



# **Intro to Python 3 with Numerical Analysis**

## **Participant Guide**



#### Copyright

This subject matter contained herein is covered by a copyright owned  
by: Copyright © 2019 Robert Gance, LLC

This document contains information that may be proprietary. The contents of this document may not be duplicated by any means without the written permission of TEKsystems.

TEKsystems, Inc. is an Allegis Group, Inc. company. Certain names, products, and services listed in this document are trademarks, registered trademarks, or service marks of their respective companies.

All rights reserved

IN1658-SG-02 / 11.01.2019

©2019 Robert Gance, LLC

ALL RIGHTS RESERVED

This course covers Introduction to Python 3 with Numerical Analysis

---

No part of this manual may be copied, photocopied, or reproduced in any form or by any means without permission in writing from the author—Robert Gance, LLC, all other trademarks, service marks, products or services are trademarks or registered trademarks of their respective holders.

This course and all materials supplied to the student are designed to familiarize the student with the operation of the software programs. THERE ARE NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, MADE WITH RESPECT TO THESE MATERIALS OR ANY OTHER INFORMATION PROVIDED TO THE STUDENT. ANY SIMILARITIES BETWEEN FICTITIOUS COMPANIES, THEIR DOMAIN NAMES, OR PERSONS WITH REAL COMPANIES OR PERSONS IS PURELY COINCIDENTAL AND IS NOT INTENDED TO PROMOTE, ENDORSE, OR REFER TO SUCH EXISTING COMPANIES OR PERSONS.

This version updated: 11/1/2019

# *Notes*

---

# *Chapters at a Glance*

Chapter 1	Python Language Overview	15
Chapter 2	Modular Python	83
Chapter 3	Numerical Analysis Using Python	95
Chapter 4	Introducing Matplotlib	147
Chapter 5	Pandas	175
	Course Summary	263
Appendix A	Seaborn and Scikit-learn	269

# *Notes*

---

# Introduction to Python 3 with Numerical Analysis



# Course Objectives

Establish foundational **language fundamentals** including **idioms** and **best practices**

Explore techniques for **acquisition**, **analysis** and **formatting** of data

Utilize **data analytic libraries** such as **NumPy**, and **Pandas** to perform scientific and statistical analysis

Incorporate visualization tools such as **Matplotlib** to create charts and integrate results into notebooks



# Course Agenda

## Day 1

- Language Overview
  - Data Types
  - Data Structures
  - Flow Control
- Modular Python
  - Functions
  - Modules
  - Classes



# Course Agenda

## Day 2

- Using NumPy
- Introducing Matplotlib
- Pandas



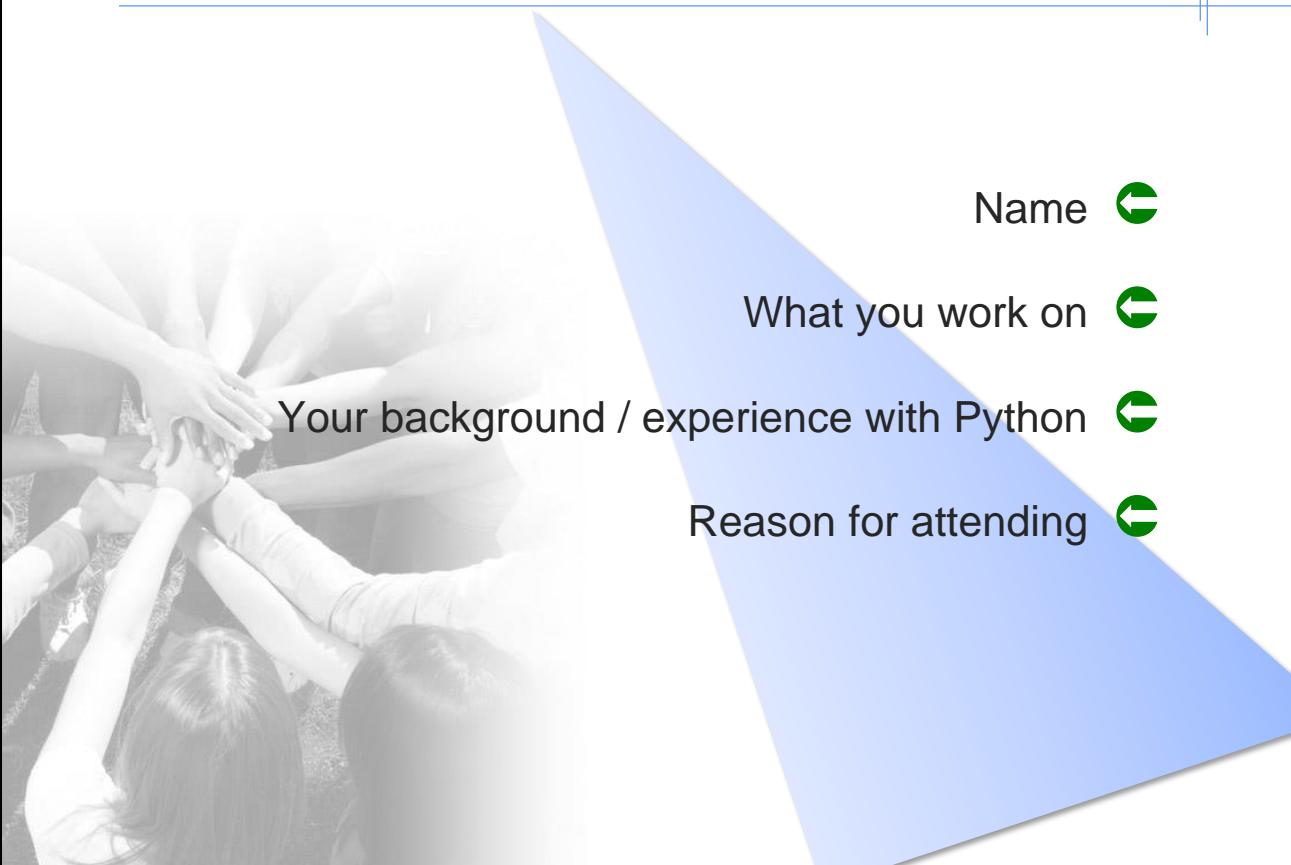
# Course Agenda

## Day 3

- More with Pandas
- Seaborn and Other Tools



# Introductions





# Logistics



## ***Typical Daily Schedule\****

9:00	Start Day
10:15	Morning Break
12:00 – 1:00	Lunch
2:15	Afternoon Break 1
3:30	Afternoon Break 2
5:00	End of Day

\* Your schedule may vary



# Get the Most from Your Experience



# Chapter 1

# Python Language

# Overview

Core Features of the Language



# Overview

Python Scripting and Environment

Data Types

Control Structures

Data Structures



# Introducing Python

- *High-level* programming language
- Supports *imperative*, *object-oriented*, and *functional* programming
- Automatic memory management
- Dynamic typing
- It is often referred to as a scripting language, much like Ruby, JavaScript, Perl, Tcl, others

Python is a very expressive language. It supports numerous styles, though to a lesser degree the functional style. Python does not require the use of classes and object-oriented techniques such as inheritance and polymorphism. However, support for the OO programming style exists and has been improved upon in the move from Python 2 to 3.



# Version History

Which version do I have? Specify: **which python** or **where python**, and **python -V**

Version	Date	Comments
0.9	1991	Pre-1.0 release
1.5	1999	Unicode, list comprehensions
1.6	2000	
2.0	2000	
2.1	2001	Nested function scoping rules, warnings added
2.2	2001	Subclasses, super(), multiple inheritance rules
2.3	2002	Sets and generators
2.4	2004	Decorators
2.5	2006	with statement, try/except/finally combo
2.6	2008	print() function, string formatting methods
<b>2.7</b>	<b>2010</b>	<b>Last 2.x release, several 3.x backported feat.</b>
<b>3.0</b>	<b>2008</b>	<b>Not 2.x compatible, avoid using 3.0-3.2</b>
3.3	2012	Made easier to migrate 2.7 to 3.3 code
3.4	2014	asyncio, statistics modules
3.5	2015	type hints, performance improvements
3.6	2016	String literals, async generators/ comprehensions, variable annotations
3.7	2018 (Jun)	
3.8	2019 (Oct)	Assignment expressions (walrus operators), Positional-only arguments

Major changes occurred in the change from Python 2 to 3. Even as of this writing, more Python 2 code exists than Python 3 code. Though Python 2.7 will be the last 2.x release, and no more maintenance releases will occur on Python 2.7 after 2020, many solutions have been written with Python 2. Python 3.3 and later makes it easier to migrate Python 2.7 code by backporting Python 3 features while being more accepting to 2.7 syntax. Most Python 3<sup>rd</sup> party libraries also exist with 3.x versions now. Python 3 is being implemented more and more.

In addition to Python versions, there are also Python "flavors". These include:  
 CPython - the typical default Python. The interpreter is written in C and is the one most used when executing Python scripts.

IronPython - Written in C# and runs within the .NET VM. C# classes can be imported as well as limited Python Standard Libraries.

Jython - compiles Python code into Java ByteCode. Uses JDK classes and Python libraries but compiles into .class files and runs within a JVM.

Other versions exist: ActivePython (by ActiveState), Stackless (supporting micro-threads), winpython (scientific windows portable version), PyObjC (Objective-C version of Python), Brython (JavaScript), RubyPython.



# Executing Scripts

- There are multiple ways to execute Python code:
  - Within the Python shell

```
c:\progs\python38>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)]
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- From the command-line

```
c:\>python myscript arg1 arg2
```

- As a shell script file (shebang line)

```
#!/usr/bin/python
```

Assumes this exact  
location for python

```
#!/usr/bin/env python
```

Use first python  
found in environment

In addition to the approaches listed above, .py files can be double-clicked in GUI-based environments when an association is established with them. Also, specialized IDEs that support and understand Python can typically run files in standard and debug modes.

For more options on executing scripts from the command-line, refer to the documentation, specifically the following resource:  
<https://docs.python.org/3/using/cmdline.html>



# Python's Interactive Shell

- The CPython distribution ships with an interactive shell
  - Script code can be executed line-by-line

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)]
Type "help", "copyright", "credits" or "license" for more information.
>>> f_temp = 70.0
>>> c_temp = (f_temp - 32.0) * 5.0/9.0
>>> c_temp
21.11111111111111
>>>
```

- To open the **Python Shell**:

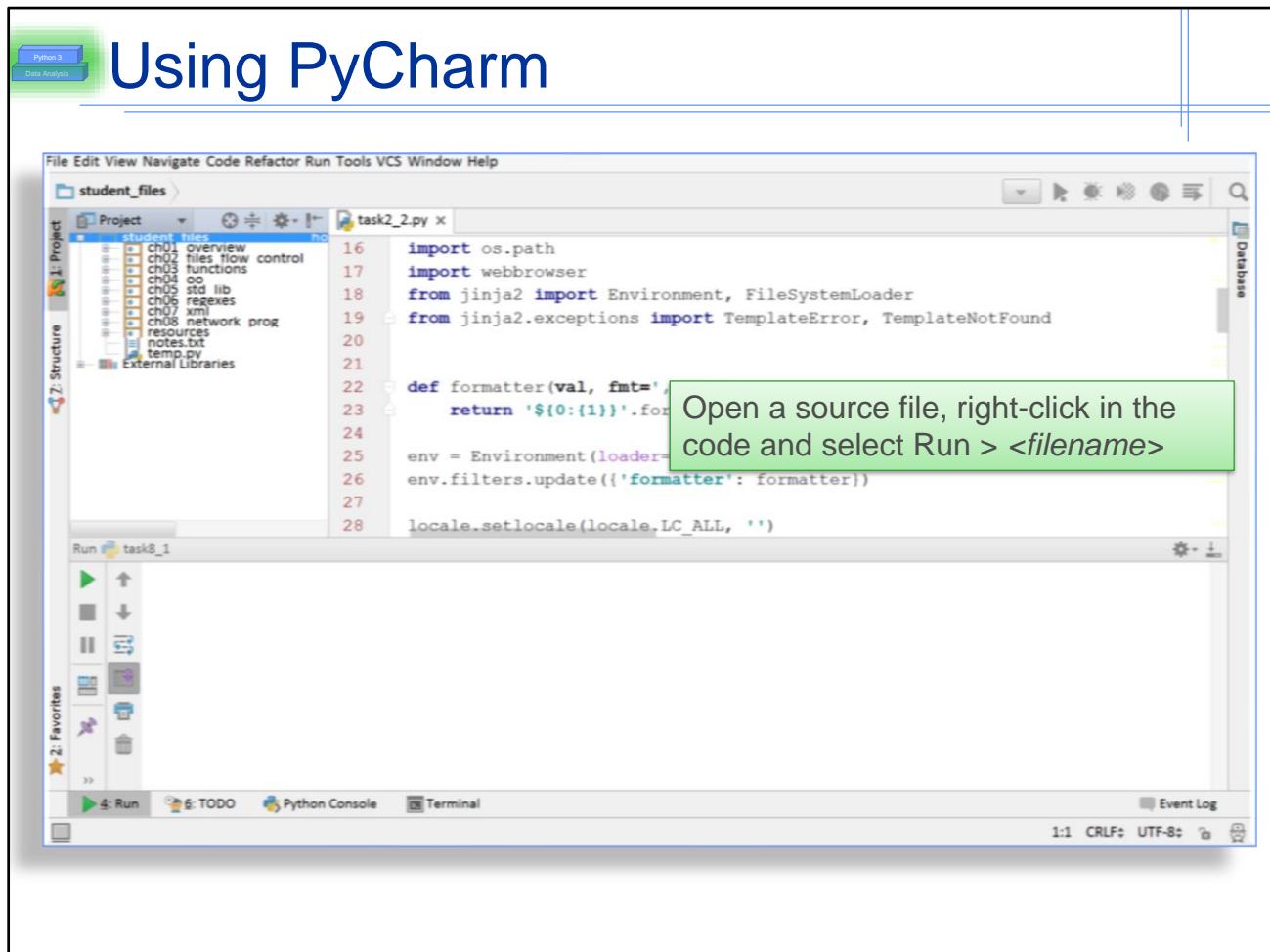
- Open a terminal (console) and type: **python**

Note: on some OS X operating systems with multiple Python distributions installed, you may have to type **python3**, **python3.8**, **python3.7**, etc.

To run a script from within the Python 3 shell, type:  
`exec(open('<path>/filename.py').read())`

The `execfile()` command was removed in Python 3.

The Python shell doesn't have full features like an ordinary OS command prompt.

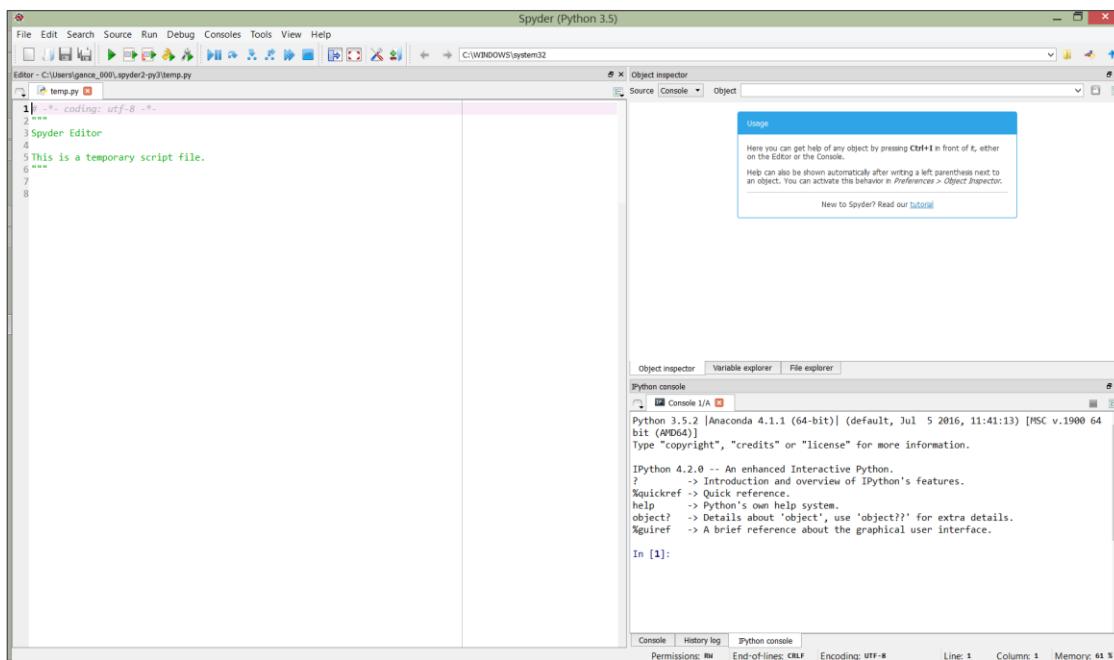


To work with PyCharm, launch the program and select Open... from the options on the right and then locate the student files directory.

Nice editing features include:

- Selecting code by hitting Tab (or Shift-Tab) to move blocks of code right and left.
- Ctrl-/ (Cmd-/ on OS X) to block comment out code.

# A Hidden IDE



When using Anaconda4+, the Spyder IDE is embedded in the distribution. From a terminal window, type **spyder** and it should launch.

Note, if the command is not recognized, you will need to edit your PATH environment variable and add the <ANACONDA\_HOME>/Scripts directory to it.

On Windows platforms, do this by going into the Control Panel and selecting the System icon. From there, select the Advanced tab followed by the Environment Variables... button. You should be able to edit your path here.

OS X users can open their .bash\_profile file (it's a hidden file found in your home (~) directory). One way to do this is to type the following from a terminal window:

```
touch ~/.bash_profile
open ~/.bash_profile
```

This will typically open *TextEdit* upon which you can add a statement like the following:

```
export PATH=$PATH:<ANACONDA_HOME>/bin
(<ANACONDA_HOME> should be the path to your Anaconda installation directory).
```



## Your Turn! Task 1-1

### Python Environment Setup and Test

- Install Python
- Test the shell environment
- Set up the student files and an IDE

Locate the instructions for this exercise in the Student Manual

On OS X, there is usually a link to python at: /usr/local/bin/python. To see where this points, type ls -l /usr/local/bin/python at a command prompt. It should reveal the actual Python interpreter location.

Most of the time, depending on how it was installed on OS X, you may find your actual Python interpreter located at:

/System/Library/Frameworks/Python.framework/Versions/3.x/bin/python



# Python Data Types

- Python has many built-in data types
- **type()** provides insight about a variable's type
  - Use **isinstance()** in practice to verify a variable's type

```
s1 = 'hello'  
s2 = b'goodbye'  
type(s1)                                # <class 'str'>  
type(s2)                                # <class 'bytes'>  
type(s1) is str                          # True  
isinstance(s1, str)                      # True  
  
isinstance(s2, str)                      # False
```

`isinstance()` can check for inheritance, as shown above.

In general it is not common to check types, so don't place a lot of statements doing so.

Note: `basestring`, a class in Python 2 that acted as the parent to both the `str` and `unicode` types is gone in Python 3.

# Built-in Data Types

Value	Description	Principal Types:
int	integer	Numerics
float	floating point	Sequences
complex	complex numbers	Mappings
bool	booleans	Classes
str	strings	Instances
bytes	fixed array of bytes	Exceptions
bytearray	array of bytes	
object	objects	
function	functions	
list	ordered item sequence (arrays)	
tuple	fixed order item sequence (fixed arrays)	
dict	unordered collection of key/value pairs (hashes)	
set	unordered collection of unique items	
file	File objects	

Python's principal types can be categorized into the following: numeric, sequences, mappings, classes, instances, exceptions.

The builtins module provides a list of the built-in types and classes. Use the following within the Python shell to view the contents of this module:

```
import builtins
dir(builtins)
```

The long type from Python 2 is no longer present because int types in Python 3 are equivalent to the long type in Python 2. Python 3 provides a bytes and bytearray type to deal with the fact that str types are now represented using Unicode characters.

# Identifiers and Naming Conventions

- Identifiers can be named using Unicode letters, digits, and underscore

– Cannot begin with a digit

```
greet = 'hello'      # okay
1greet = 'hello'    # bad
_greet = 'hello'    # okay
```

- Identifiers are case sensitive

```
greet, Greet, GREET = 'hello', 'howdy', 'hola'
```

- Python has style guidelines

– Known as PEP 8

<https://www.python.org/dev/peps/pep-0008/>

– Rules are only a guide but encouraged and followed

ch01\_overview/styleguide.py

Variable naming conventions are similar to those encountered in other languages.

One note related to numeric values: Python 3.6 introduced the use of underscores to separate groupings for easier readability. So, the following items are legal and equivalent:

x = 1000000

y = 1\_000\_000

When y is printed, it will print without the underscores.

# PEP 8 Conventions and Styles

```
say_hello = 'hi'      # underscores to separate identifier words
```

```
def my_func():          # underscores for function names
    print(say_hello)
```

```
class SomeFunClass(object):  # CapWords syntax for class names
    pass
```

```
def my_func2(arg1, arg2, arg3, arg4,
             arg5, arg6): pass
```

start arguments beneath other arguments

```
days = [
    1, 2, 3
]
```

or

```
days = [
    1, 2, 3
]
```

Indentation is not critical on continuation lines

ch01\_overview/00\_styleguide.py

While these rules are defined in the PEP 8, they are only suggestions. Most people follow these naming conventions. The main reason for this guide is to promote readability.



# Additional PEP 8 Formatting

- ❑ Use 4 spaces for indentations
- ❑ Never mix spaces and tabs. Prefer spaces over tabs.

```
if datetime.now().isoweekday() == 5:  
    print("TGIF!")
```

*4 spaces, no tabs*

- ❑ Two blank lines between top-level functions and classes
- ❑ One blank line between methods in classes
- ❑ Use blank lines in functions to group logical sections
- ❑ Put imports at the top of a file
- ❑ Put multiple imports on separate lines
- ❑ Name classes using CamelCase notation
- ❑ Name variables and free-functions using underscores

`import os, sys`



`import os  
import sys`



ch01\_overview/00\_styleguide.py

These are only a few suggestions from the PEP 8 guide.



# Python Keywords

- Don't use these reserved words as identifiers

<code>and</code>	<code>False</code>	<code>nonlocal</code>
<code>as</code>	<code>finally</code>	<code>not</code>
<code>assert</code>	<code>for</code>	<code>or</code>
<code>break</code>	<code>from</code>	<code>pass</code>
<code>class</code>	<code>global</code>	<code>raise</code>
<code>continue</code>	<code>if</code>	<code>return</code>
<code>def</code>	<code>import</code>	<code>True</code>
<code>del</code>	<code>in</code>	<code>try</code>
<code>elif</code>	<code>is</code>	<code>while</code>
<code>else</code>	<code>lambda</code>	<code>with</code>
<code>except</code>	<code>None</code>	<code>yield</code>

- Don't use any of Python's built-in types, classes, functions, exceptions, etc.
  - For example: `open`, `len`, `list`, etc...

In addition to the keywords listed above, you should also not create identifiers that use any of the names from the `__builtins__` module. Perform a `dir(__builtins__)` to view these.

# Strings

- Strings are immutable **sequences** of Unicode characters
  - Type: **str**
  - Formal notation:
  - Literal notation:

```
my_str = str('Python is great!')
my_str = 'Python is great!'
```

```
my_str = 'Python is fun'      my_str = 'Python is very fun'
                                         ↑
                                         Creates a new string object
```

- May use single or double quotes

PEP-8 does not have a preference

Triple quoted strings can span multiple lines. A common use is for documentation (called a docstring).

```
my_str = 'Python\'s great fun'
my_str = "Python is great fun"
my_str = """Python is so much fun"""
```

In Python 3, strings are Unicode characters. Some characters must be escaped to use them. Escape characters include:

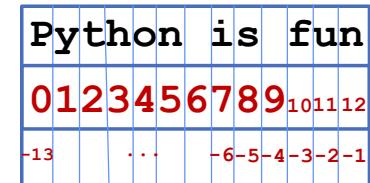
\\	backslash
\'	single quote (as shown in the slide)
\\"	double quote
\b	backspace character (ascii)
\n	linefeed
\r	carriage return
\t	tab

\uXXXX	16-bit Unicode character
\uXXXXXXXXXX	32-bit Unicode character
\N{name}	Unicode named character

# String Random Access and Slicing

- Strings are sequences and thus support random access:

```
my_str = 'Python is fun'  
print(my_str[0])           # 'P'
```



- Sequences may be sliced (sub-sequenced):

slice = string[start : end : step]

startval (included)

endval (excluded)

```
print(my_str[0:9])      # 'Python is'  
print(my_str[:3])       # 'Pyt'  
print(my_str[3:])        # 'hon is fun'  
print(my_str[-3:])       # 'fun'
```

[ch01\\_overview/01\\_strings.py](#)

Two things to note about slicing:

1. Negative values supplied will wrap around from the end.
2. Slices return new objects in memory.



# String Features

- Strings may be concatenated (creates new string):

```
my_str = 'Python is '  
new_str = my_str + 'fun'
```

- Long strings may be continued:

```
my_str = 'I just cannot seem \  
to finish this \  
darn string!'
```

- Strings (sequences) may be replicated:

```
'hello' * 3
```

```
hellohellohello
```

If lots of string concatenation will be performed, it is generally more performance oriented to use join().

In addition to strings, any long line may be continued using the backslash (\) character.

# String Methods

- **join()**

```
nums = ['1', '2', '3']
' '.join(nums)      # 1 2 3
```

Concatenates a list of strings using the specified separator string

- **split()**

```
my_str = "Python is great"
my_list = my_str.split(' ')  # Results: ['Python', 'is', 'great']
```

Creates a list of strings based on a specified separator string

- **replace()**

```
Returns a new string with all matching substrings replaced by the new one
```

```
my_str.replace('is', 'is still')  # yields: 'Python is still great'
```

These are a few of the helpful methods built into the string class.

By default, split() will break a string up based on any whitespace characters and it will split it as many times as necessary. split() supports a second argument, maxsplit, that can limit the number of times a split occurs.

# String Methods

- **strip()**

Returns a new string with whitespace removed from each end

```
new_str = '      Whitespace will be removed.      '.strip()
```

- **find()**

```
my_str = 'Python is great'  
my_str.find('is')      # returns 7  
my_str.find('not')    # returns -1
```

Finds the index of the first occurrence of the substring

- **str()**

String class constructor accepts any object returns a string object

```
s = str(55)          # creates a string from the int  
s = str(3.14)        # creates a string from the float
```

The str() method is a special case that invokes the string class constructor (`__init__`) creating a new string.

# Conversions

- Use the type name as a function to perform conversions

```
s1 = str(55)          # '55'  
s2 = str(3.14)        # '3.14'  
  
i1 = int('37')        # 37  
i2 = int(3.14)        # 3  
  
f1 = float(55)        # 55.0  
f2 = float('3.14')    # 3.14
```

- If a conversion cannot be made, a ValueError is raised:

```
int('hello')  
ValueError: invalid literal for int() with base 10: 'hello'
```

Errors, such as ValueErrors, can be properly handled. Exception handling will be discussed later.

# String format()

- A preferred way to format data is to use the **format()** method of the string class

```
s = 'It has been raining for {0} {1} and {0} {2}'
```

```
new_str = s.format(40, 'days', 'nights')
```

'It has been raining for 40 days and 40 nights'

{ fieldname | index [!spec] [:format] }

```
s6 = '{lang} is over {0:0.2f} {date} old.'
       .format(20, date='years', lang='Python')
```

Python is over 20.00 years old.

# more examples:

```
print('{0:>8}'.format('101.55')) # 101.55      (field-width of 8)
print('{0:-^20}'.format('hello')) # -----hello-----
```

ch01\_overview/01\_strings.py

[!spec] refers to either: [!s] which executes the str() function on the value before inserting into the string, or [!r] repr(), or [!a] ascii().

If you are using Python 3.6+, prefer f-strings (mentioned shortly) over the use of format().

More on the format() method can be found at:

<https://docs.python.org/3/library/string.html>

# Other String Methods

- A number of remaining string class methods include:

capitalize()  
 center(width, char)  
 count(str)  
**endswith(str)**  
 expandtabs()  
 index()  
 join(sequence)  
 ljust(width, char)  
**lower()**

maketrans()  
 partition(str)  
 splitlines(bool)  
**startswith(str)**  
 swapcase()  
 title()  
 translate()  
**upper()**  
 zfill(width)

isalnum()  
 isalpha()  
 isdecimal()  
 isdigit()  
 isidentifier()  
 islower()  
 isnumeric()  
 isprintable()  
 isspace()  
 istitle()  
 isupper()

Numerous additional string class methods exist. These are listed above.

str.count(substr)	returns the number of occurrences of the substr
expandtabs()	returns new string with tabs replaced by 8 spaces
ljust(width, fillchar)	returns new string with the string left aligned in a string the size of width. fillchar can be a padding character.
maketrans(), translate()	returns string with chars translated to new values
partition(substr)	returns a 3-item tuple: string before substr, substr, string after substr
splitlines(bool)	returns list of strings split on str, strips whitespace if bool is false (default)
title()	uppercases first letter of each word
zfill(width)	pads a string with zeroes if string is shorter than width
endswith(substr), startswith(substr)	returns true or false if the substr ends or begins the string, substr may be a tuple of strings



# Python 3.6 Formatted Strings (f-strings)

- Starting with Python 3.6, formatted string literals may be used
  - This syntax won't work in earlier Python versions

```
name, age = 'Tom', 42
```

```
s = f'Hello {name}. Ten years ago, you were \  
{age - 10} years old.'
```

```
print(s)
```

You must place an 'f' at the beginning of the literal.

f-strings must begin with the letter 'f'. On a performance note, f-strings are faster than using format(), %, or + in string operations.

# Sequences

- Sequences are ordered collections of objects
  - Types include: **str, list, tuple, range**

```
dirs = ['North', 'South', 'East', 'West']
```

- Common features include:
  - Random access, slicing

```
dirs[2] # 'East'
```

```
dirs[-2:] # ['East', 'West']
```

- Concatenation

```
dirs + ['NW', 'NE', 'SW', 'SE']
```

- Replication

```
dirs * 2 # ['North', 'South', 'East', 'West', 'North', 'South', 'East', 'West']
```

- Membership

```
if 'east'.capitalize() in dirs:
    print(dirs.index('east'.capitalize()))
```

Technically, `range()` is a class that generates sequence values on-demand. Since it generates a sequence of values, it can usually be used like any sequence. Several other specialized sequence types, such as `bytes`, `bytearray`, and `memoryview`, also exist.

Sequences exhibit similar behaviors such as the ability to be randomly accessed, perform slicing, or be replicated through multiplication of a scalar value.

# Lists

- Lists - mutable, ordered collections of objects

```
my_list = []
```

```
my_list = [1, 3, 5]
```

```
my_list = [3.3, 'hello', Person()]
```

```
my_list = list()
```

```
my_list = list('hello')
```

Empty lists

someList = list(sequence)

```
my_list = [3.3, 'hello', Person(), 3.3, Person()]
```

Lists may contain duplicates

```
my_list = [1, 2, 3]
```

```
my_list.append(10)
```

```
my_list.insert(1, 'hello')
```

1, 'hello', 2, 3, 10

ch01\_overview/02\_iterating.py

Lists may be created using literal notation with square brackets [ ]. Lists may also be created using the list() constructor.

Use append() to add items onto the end of the list.

To add a value into the middle of a list, use insert(position, value) keeping in mind that sequences are zero-based in Python.

# Tuples

- Tuples - *immutable*, ordered collections of objects

```
my_tuple = ()
```

Empty tuple

```
my_tuple = (1, 3, 5)
```

```
my_tuple = (3.3, 'hello', Person())
```

```
my_tuple = tuple()
```

Empty tuple

```
my_tuple = tuple('hello')
```

tuple(sequence)

```
my_tuple = (1,)
```

tuple with one element

Tuples do not have `append()`, `insert()`, or other methods that attempt to modify the data as this would violate the concept of a tuple.

Some like to describe the difference between tuples and lists as:

- tuples have structure while lists have order, or
- tuples are heterogeneous (hold different kind of things) while lists are homogeneous (hold same kind of things)

Example: a database in Python commonly returns a list of records (tuples). The list contains same type records while the tuple contains different fields.

# Useful Sequence Functions

- **len()**

```
dirs = ['North', 'South', 'East', 'West']
print(len(dirs), len(dirs[0]))
```

4, 5

- **max()**

```
max([1973, 2001, 2015, 2013, 1994])
```

2015

- **min()**

```
min([1973, 2001, 2015, 2013, 1994])
```

1973

These functions are all global. They do not have to be imported to be used. How they perform can be customized for custom types such as custom class types.

`min()` and `max()` also support a key function that can be supplied as an argument to define what biggest or smallest means in sequences where custom behavior is needed.

Consider the following example that uses a dictionary:

```
weights = {'Paul': 195, 'Mary': 165, 'Tom': 182}
print(min(weights.keys(), key=lambda key: weights[key]))
```

Note the key as an argument to `min()`.  
The result displays 'Mary'.

# Illegal Operations

- The following operations may not be performed on sequences:

```
[1, 2, 3] + 'Bob'  
[1, 2, 3] + (4, 5, 6)
```

Can't concatenate different sequence types

```
my_list = []  
my_list[0] = 'Bob'
```

Empty list to begin with, so there is no 0<sup>th</sup> address to assign something into

```
my_list = [None] * 5  
my_list[0] = 37  
my_list[1] = 'Bob'
```

This approach is legal because the list is sized to begin

While sequences of similar types can be concatenated, sequences of different types may not be. Also, assigning a value into a sequence in a position that doesn't exist cannot be accomplished using random access (index) notation.



# Control Structures

- Conditional

There may be zero or more `elif` branches

The `else` is optional

```
if test:  
    <one or more statements>  
elif test2:  
    <one or more statements>  
else:  
    <one or more statements>
```

Beginning and ending of blocks are determined by indentations (use 4 spaces)

```
if test:  
    print('This is part of the if block!')  
    print('So is this!')  
print('This is not!')
```

# Truthy / Falsey Values

- Test conditions in control structures expect a *bool*
  - However, everything in Python can evaluate to either a *True* or *False* value
- These values are all considered *False*:

Everything else  
evaluates to True!

*False*  
*None*  
0  
0.0  
''  
[]  
()  
{}

Empty string

```
data = [1, 2, 3]
if len(data):
    print('Data not empty.')
```

In the example above, when the list is cleared ( `data.clear()` ), all of its elements are removed, making the list empty. The empty list evaluates to *False*, the `not` operator inverts it and the `print` statement is executed.

# Comparison Operators

- Below are a few of the comparison operators that may be used in tests:

<code>a == b</code>	
<code>a &lt; b</code>	
<code>a &gt; b</code>	
<code>a &gt;= b</code>	
<code>a &lt;= b</code>	
<code>a != b</code>	
<code>a is b</code>	(same object)
<code>a is not b</code>	
<code>a in y</code>	(a member of)
<code>a not in y</code>	
<code>not a</code>	
<code>a and b</code>	(and, or both short circuit)
<code>a or b</code>	

The `is` operator checks memory locations (meaning same object), the `==` operator looks at members to perform item by item comparison. `list1` and `list3` below are different objects but have the same members so `==` is True while `is` returns False.

```
list1 = [1,2,3]
list2 = list1

print(list1 == list2)          # True
print(list1 is list2)          # True

list3 = list1[:]               # slice makes copy

print(list1 == list3)          # True
print(list1 is list3)          # False
```

# Iterative Control Structures

```
while test:  
    <one or more statements>
```

```
else:
```

```
    <one or more statements>
```

Optional else block is only executed if conditional terminates naturally (without a break)

else block not commonly used

```
for one_item in iterable:  
    <one or more statements>
```

```
else:
```

```
    <one or more statements>
```

Optional else only executed if conditional terminates naturally (rarely used)



# Iterative Control Structures

```
while True:
    name = input('Enter name: ')
    if not name:
        break
    process_name(name)
```

This loops until just the Enter key is pressed

`process_name()` is just a user-defined function here

```
week = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
for day in week:
    if day != 'Sun' and day != 'Sat':
        print('Weekday: ' + day)
```

Weekday: Mon  
 Weekday: Tue  
 Weekday: Wed  
 Weekday: Thu  
 Weekday: Fri

ch01\_overview/02\_iterating.py

Python's for and while loops are the only iterative control structures. There is not a classic 3 part `for( )` loop as seen in C++, Java, and so many other languages. Both while and for support the break and continue statements.



## Your Turn! Task 1-2

- Write a script that finds the *domain name* of a URL
  - Your results should not contain *the protocol*, any *subdirectories*, *port*, or *query string arguments*
  - Should not use regular expressions at this time

`https://docs.python.org/3/`

`https://www.google.com?gws_rd=ssl#q=python`

`http://localhost:8005/contact/501`

### What level are you?

Experienced Folks: Attack this task on your own

Some programming experience: There are additional hints in the source code  
to help you get going (`ch01_overview/task1_2_starter.py`)

Need a little additional help: Step-by-step instructions are always provided in the  
workbook in the back of the manual

Since URLs can be a variety of formats, for this exercise, we will limit URLs to the ones shown above.

# Working with Files

- Use the built-in `open()` command to work with files

```
# read text
file = open(filename, encoding='utf8')
```

Default mode is 'rt' (read text)

```
# write text
file = open(filename, 'w', encoding='utf8')
```

Default encoding is platform-specific

mode = 'r' or 'w' or 'r+' or 'a'

(read, write, read+write, or append)

Note: other mode options exist  
including: rb, rb+, wb, wb+, ab, ab+

**f** is a file object

```
f = open('myfile.txt')
entire_file = f.read()
```

String containing  
entire file contents

The other arguments that may be passed into the `open()` method are mode='r', buffering=1, errors=None, newline=None, closefd=True, opener=None

mode defines how a file is opened, 'r' is read, 'a' is append, 'w' is write, 'r+' is read or write, 'b' is binary, 't' is text (default).

buffering is either 1 or 0 in text mode or a chunk size in binary mode (e.g. 4096).

`errors='strict'` will raise a `ValueError` exception if encoding errors occur.  
`errors='ignore'` will ignore encoding errors (and likely lead to problems).

`newline=None` or " or '\n' or '\r' or '\r\n'. If None, universal newline support is enabled which reads platform-specific newlines in as \n and writes them out as platform-specific. If "", universal newline support is disabled. Text mode only.  
`encoding='utf8'`, the default is platform-specific. Only valid for text mode.

`closefd=False` will leave file descriptors open after files are closed. Rarely used.  
`opener` allows another function to serve as the `open` function().



# Reading from Text Files

- Some file object methods:

f.readline() reads one line from a file, retains newline

f.readline(10) reads first 10 characters from the line

my\_list = f.readlines() reads all lines, puts them in a list

f.writelines(list) writes a list, one item per line to a file

- A common way of processing file data iteratively:

```
for line in open('myfile.txt', encoding='utf8'):  
    # process line from file
```

line will include the termination character (\n)

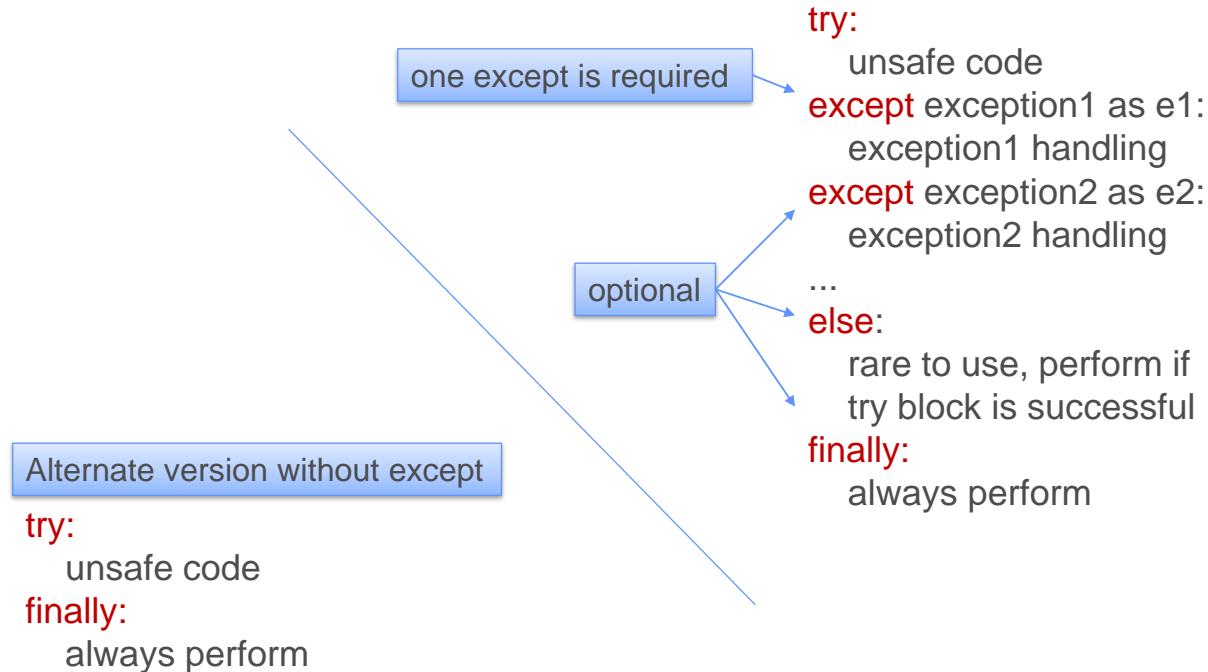
The code above works because the open() method returns a file object and the file object is iterable.

Note: Python recognizes 'utf-8' and 'utf8' equally!



# Exception Handling Structure

- Exceptions are handled using the **try - except** mechanism



Exceptions in Python are used to capture and respond to unexpected (exceptional) conditions. The flow would be as follows:

Under normal conditions:

try then else then finally

Under exceptional conditions:

try then except then finally

It is rare to use the **else** block for two reasons:

1. Arguably it is not commonly needed.
2. People aren't sure of when to use the else block (or why to use it).

A potential use for the else block might be as follows:

```

try:
    do_stuff that can raise a ValueError
except ValueError:
    handle the ValueError
else:
    do more stuff that can raise a ValueError,
    but this won't be handled by the first except,
    yet is still covered by the finally
finally:
    always do this
    
```

Regardless, the else is still rare to use

# Handling Exceptions

```
a = input('Number 1: ')
b = input('Number 2: ')
try:
    result = float(a) / float(b)
    print('Division result is: {}'.format(result))

except ValueError:
    print('Improper value entered.')
```

Illegal values, such as 'hello'  
generate a **ValueError**  
Flow jumps to except block

```
a = input('Number 1: ')
b = input('Number 2: ')
try:
    print('Division result is: {}'.format(float(a) / float(b)))

except ValueError:
    print('Improper value entered.')
except ZeroDivisionError:
    print('Number 2 may not be zero.')
```

Zero causes ZeroDivisionError,  
hits the second except block

When an exception occurs, flow jumps to the except block. If the potential for multiple errors can exist, multiple except blocks should be used. Try to use the most specific exception types as possible. List the more specific exception types first.



# Grouping and Generic Handling

```
a = input('Number 1: ')
b = input('Number 2: ')
try:
    result = float(a) / float(b)
    print('Division result is: {}'.format(result))

except (ValueError, ZeroDivisionError):
    print('Improper value entered.')
```

When exception types will be handled similarly, they can be grouped

```
a = input('Number 1: ')
b = input('Number 2: ')
try:
    result = float(a) / float(b)
    print('Division result is: {}'.format(result))

except Exception:
    print('Improper value entered.')
```

Legal, but as a rule of thumb, try to handle the more specific exception type and avoid the generic usage shown here

# The Exception Object

```
a = input('Number 1: ')
b = input('Number 2: ')
try:
    result = float(a) / float(b)
    print('Division result is: {}'.format(result))

except Exception as err:
    print(err, type(err))
```

could not convert  
string to float: 'hello'

<class 'ValueError'>

Use the exception object to  
extract additional information  
about the error condition

ch01\_overview/03\_exceptions.py

The preferred format is the one shown above using the 'as' phrase followed by the exception object.

err also has a property called args which represents a tuple or error messages. It is possible that one exception is wrapped inside another, but creating an exception chain. The err.args tuple provides a means for iterating through multiple messages if desired.

# Proper IO Error Handling

```
try:
    f = open('my_file.txt')
    try:
        for data in f:
            # work with data
    finally:
        f.close()
except IOError as e:
    print(e)
```

If file is not found, open() fails, f will be None therefore close() can't be called, so a nested try-finally structure is required

```
f = None
try:
    f = open('my_file.txt')
    for data in f:
        # work with data
except IOError as e:
    print(e)
finally:
    if f:
        f.close()
```

Avoiding the nested try-finally approach leads to somewhat clumsy code

In the first example, a nested try-finally is used because in the event that the file is never opened in the first place, you can't close() it. So, while an IOError will still be raised for failing to open(), the file object, f, will only conditionally be valid. So, the lower version also satisfies this check for f, but might not look as clean.

Neither solution is ideal, which leads to an overall better approach...



# Initialization and Cleanup

- We always want to properly clean up when working with files
  - Also true for many other resources, such as: database and network connections, sockets, etc.
- The following is a common programming paradigm:

```
do some initialization  
do some work  
do some cleanup
```

It would be nice if a control structure could take care of this for us...



# Introducing 'with'

- **with** is a control structure that performs the following:

```
do some initialization  
do work  
do some cleanup
```

- Requires a context manager object to be supplied

```
with contextmanager as obj:  
    do_work
```

A context manager is a special object that defines how to initialize at the beginning and clean up afterwards

The 'with' control is a somewhat complex, yet versatile structure. It requires the use of a special object called a context manager. The context manager requires two methods to be present (discussed on the next page).

# Using 'with'

- The file object can be used in a with control
  - It is a context manager because it has an `__enter__()` and `__exit__()` method defined

After 'with'

```
lines = []
try:
    with open('alice.txt') as f:
        lines = f.readlines()
except IOError as e:
    print('Handled {0}'.format(e))

print('{0} lines read.'.format(len(lines)))
```

Before 'with'

```
lines = []
try:
    f = open('alice.txt')
    try:
        lines = f.readlines()
    finally:
        f.close()
except IOError as e:
    print('Handled {0}'.format(e))
```

ch01\_overview/04\_contextmanager.py

Both versions are equivalent and will result in the same output and proper file closing whether there is an exception or not. The difference is the second version uses the with control providing a cleaner solution overall.



# Under the Sheets: How 'with' Works

- Context managers are objects that define two methods: `__enter__` and `__exit__`
- `__enter__` is always called at the beginning
  - Return value becomes the optional 'as' object
- `__exit__` is called *no matter what happens*

```
class CtxMgr(object):
    def __enter__(self):
        print('enter')
        return 'foo'

    def __exit__(self, typ, value, traceback):
        print('exiting')

with CtxMgr() as obj:
    print(obj)
```

Outputs:  
enter  
foo  
exiting

ch01\_overview/05\_contextmanager.py

A context manager defines both an `__enter__` and an `__exit__` method in a class. Though classes haven't been discussed thus far, the class concept is irrelevant to this discussion. Completely ignore the references to `self` at this time. When the `with` statement is encountered, the context manager's `__enter__` method is invoked. The return value is passed to the 'as' portion of the `with` control.

The 'as' portion of the control is optional but receives the return value from `__enter__()`.

When the block within the `with` control is finished executing, the `__exit__()` method will be invoked. The `__exit__()` will be called no matter whether an exception is raised or not.

# More with Lists

- Useful list methods:

<code>append(item)</code>	adds item to the end of the list
<code>insert(pos, item)</code>	adds item at pos location
<code>index(item)</code>	finds the first occurrence or -1
<code>sort()</code>	sorts in-place
<code>extend(lst2)</code>	similar to <code>list1 += list2</code>
<code>reverse()</code>	reverses the list order in-place
<code>pop()</code>	returns/removes last item
<code>remove(item)</code>	removes first occurrence of item

Most of the list methods are self-explanatory or easily learned and used. They are very useful at times. The last two items can remove objects from a list. In addition to the methods above, the `del` statement can remove items (and slices from a list):

```
a = [1, 2, 3, 4, 5]
del a[1:3]                      # a is now [1, 4, 5]
```

# Additional Iterating Techniques

- Use `enumerate()` to obtain access to both the **index** and the **value** when iterating:

```
weekdays = [
    'Monday', 'Tuesday',
    'Wednesday', 'Thursday', 'Friday'
]

for idx, value in enumerate(weekdays):
    print(idx, value)
```

o Monday  
1 Tuesday  
2 Wednesday  
3 Thursday  
4 Friday

- Use `reversed()` to iterate from the end to start:

```
for value in reversed(weekdays):
    print(value)
```

Friday  
Thursday  
Wednesday  
Tuesday  
Monday

ch01\_overview/06\_iterating.py

These are often overlooked, but handy globally available functions that accept a sequence and return values in a different way.

Note: the rules of these functions:

`enumerate()` takes any iterable while `reversed()` takes any sequence.

What's the difference?

Sequences are all iterable, but not all iterables are sequences. For example, if you create your own class, you have the ability to define how to make it iterable.

However this class would not be a sequence of objects.



# Iterating Multiple Collections

- Another useful function to assist with processing lists in parallel is `zip()`:

```
fruit = ['Apple', 'Orange', 'Banana', 'Watermelon']
color = ['red', 'orange', 'yellow', 'green', 'blue']

for f, c in zip(fruit, color):
    print('The {} is {}'.format(f, c))
```

The Apple is red  
The Orange is orange  
The Banana is yellow  
The Watermelon is green

ch01\_overview/06\_iterating.py

`zip()` accepts any number of iterables and provides a variable for each one. The function will end when the SHORTEST iterable completes.

# Sorting

- Use **sort()** for simple, in-place sorting:

```
items = [37, 2, 0, -14]  
items.sort()  
print(items) # [-14, 0, 2, 37]
```

For lists only

- Use **sorted()** to return a *new sequence*
  - sorted() is ideal for immutable types such as tuples

```
new_items = sorted(items)  
print(new_items) # [-14, 0, 2, 37]
```

Works for any iterable

ch01\_overview/07\_sorting.py

When determining whether to use sort() or sorted() consider your needs. If a new list is desired, with the original remaining intact, then use sorted(). However, if the added overhead of creating a new list doesn't make sense, when simply modifying the one in memory will suffice, then use sort(). Performance-wise, both are about the same with sort() perhaps having a slight advantage.

# Sorting in Reverse

- Sorting in **reverse** will sort from largest to smallest:

```
items = [37, -14, 0, 2]
items.sort(reverse=True)
new_items = sorted(items, reverse=True)
print(items, new_items)
```

```
[37, 2, 0, -14] [37, 2, 0, -14]
```

Sorts items in-place from largest to smallest

Sorts items from largest to smallest but returns a new list

ch01\_overview/07\_sorting.py

Use `reverse=True` to reverse the sort order from smallest to largest to largest to smallest.

`reversed()` is not the same however. It accepts a sequence and iterates it from the end to the start.

# Sorting Using a Key

- Specialized sorting can be accomplished using a function (key) that defines what to compare

```
nums = ['13', '1', '11', '4']
nums.sort()
print(nums)                                     # ['1', '11', '13', '4']
```

*Probably not what we wanted!*

```
def sort_func(val):
    return int(val)

nums.sort(key=sort_func)
print(nums)
# ['1', '4', '11', '13']
```

```
def sort_func(val):
    return int(val)

nums2 = sorted(nums, key=sort_func)
print(nums2)
# ['1', '4', '11', '13']
```

ch01\_overview/07\_sorting.py

Sometimes the default sort is not ideal. In this example, we have a list of numeric strings. The default `sort()` sorts it using hexadecimal character values, which results in an ascii type sort not a numerical sort.

To fix this, we can provide a means for sorting. Using the `key=` keyword parameter, we can sort using a function that returns the value that should be used to compare each item in the list.



# Introducing Lambdas

- Lambdas are *inline-functions*
  - For quick, short, throw away uses such as sorting, closures, functional programming

```
funcName = lambda <arguments> : <expression>
```

- Can be used anywhere functions are used
  - type() for a lambda returns `<type 'function'>`
- Example:

<pre>def sort_func(val):     return int(val)</pre>	...equivalent to...	<code>lambda val : int(val)</code>
--	---------------------	------------------------------------

ch01\_overview/07\_sorting.py

In general, use lambdas sparingly, but use them if the use case presents itself for an elegant solution.

Lambda limitations:

- The expression must be a single expression, not a whole statement (no equals sign)
- No variable assignments allowed, no if or for loops allowed

# Sorting Using a Key (continued)

- Sorting a list of records by age (descending)

```
records = [  
    ('John', 'Smith', 43, 'jsbrony@yahoo.com'),  
    ('Ellen', 'James', 32, 'jamestel@google.com'),  
    ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),  
    ('Keith', 'Cramer', 29, 'kcramer@sintech.com')  
]  
  
records.sort(key=lambda one_rec: one_rec[2], reverse=True)  
print(records)
```



```
[('John', 'Smith', 43, 'jsbrony@yahoo.com'),  
 ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),  
 ('Ellen', 'James', 32, 'jamestel@google.com'),  
 ('Keith', 'Cramer', 29, 'kcramer@sintech.com')]
```

ch01\_overview/07\_sorting.py

This example sorts a list of tuples. It sorts records in descending order of their age (reverse=True) by using the key parameter. The lambda receives one tuple at a time, upon which it returns a[2] or the age field. This means that it will sort by the ages.



# Helpful tools for Task 1-3

- To get a list of items in a directory:

```
import os
os.listdir(directory_name)
```

- To check if a directory item is a file:

```
if os.path.isfile(filename):
    # must be a file
```

- To get file statistics:

```
import os
os.stat(filename)
```

- To get the file size (in bytes):

`os.path.getsize('filename')`

also returns the file size.

```
import os
os.stat(filename).st_size
```

Another function, `glob()` from the `glob` module can be used to retrieve items in a directory. Under the hood, `glob.glob()` uses `os.listdir()`.

Other uses:

```
glob.glob('*')      # matches all files
glob.glob('*.py')  # matches all .py files
```

`os.stat()` returns a struct with the following details:

```
(st_mode, st_ino, st_dev, st_nlink, st_uid, st_gid, st_size, st_atime, st_mtime,
st_ctime) = os.stat(path)
```



## Your Turn! Task 1-3

- Create a script that reads files from a given directory
- It then sorts and displays the files by size (*largest first*)
- We won't consider sub-directories, so we will use the following to verify an item is a file:

`os.path.isfile(filename)`

True if it is a file

Follow additional instructions within ch01\_overview/task1\_3\_starter.py

- *Advanced: can you also display the file sizes?*

# List Comprehensions

- List comprehensions provide a Pythonic way to make lists from other lists
  - Similar to loops and they take the following form:

```
newlist = [ expression for var in iterable]
```

```
list1 = [1, 2, 3, 4, 5]
list2 = [x*2 for x in list1]
print(list2)
```

[2, 4, 6, 8, 10]

---

```
newlist = [ expression for var in iterable test_condition ]
list1 = [1, 2, 3, 4, 5]
list2 = [x*2 for x in list1 if x % 2 == 0]
print(list2)
```

[4, 8]

Conditional says, only consider the even numbers

ch01\_overview/08\_listcomp.py

The expanded version of the second example above would look like this:

```
list1 = [1, 2, 3, 4, 5]
for x in list1:
    if x % 2 == 0
        list2.append(x*2)
print(list2)
```

# File Sorting Task Revisited (List Comprehension)

- Task 1-3 could have utilized a list comprehension:

```
dir_contents = os.listdir('.')

files = [(os.path.basename(item), os.stat(item).st_size)
          for item in dir_contents if os.path.isfile(item)]

files.sort(key=lambda fileinfo: fileinfo[1], reverse=True)

for item in files:
    print('{0:<20}{1:10}'.format(*item))
```

*What does the \* mean?*

It expands a sequence when used in a function call, as in:

```
def avg_temperatures(fri_temp, sat_temp, sun_temp):
    return (fri_temp + sat_temp + sun_temp) / 3

temp = [88, 92, 94]
print(avg_temperatures(*temp))
```

ch01\_overview/08\_listcomp.py

# Dictionaries

- Dictionaries are collections of name/value pairs
  - Unordered, mutable, iterable
  - Supports `len()` and `in` comparisons
  - Keys should be hashable (immutable)
  - Values may be anything

```
d = { key1 : value1, key2 : value2, ... }
```

```
my_dict = {}
my_dict = dict()
my_dict = { 'item1' : 'value1', 'item2' : 'value2' }
my_dict = dict(item1='value1', item2='value2')
my_dict['item3'] = 'value3'
print(my_dict['item2'])
```

empty dicts

adding / changing values

accessing values

Dictionaries are useful data structures as they provide a means to store any kind of data yet access that data via a key. Because dictionary keys must be unique, attempting to add an entry to a dictionary that has the same key in existence will replace the value with the new value.

# Accessing Dictionaries

```
d1 = { 'Smith': 43, 'James': 32, 'Edwards': 36, 'Cramer': 29 }
```

- Direct access can generate a **KeyError**

```
d1['Smith'] # returns 43
d1['Green'] # generates a KeyError
```

- Use exception handling to deal with a **KeyError**

```
try:
    value = d1['Green'] # generates a KeyError
except KeyError:
    value = 0
```

- **dict.get(key)** to retrieve values also:

```
d1.get('Edwards') # returns 36
d1.get('Green') # returns None
```

- **dict.get(key, default)** is safest

```
d1.get('Cramer', 0) # returns 29
d1.get('Green', 0) # returns 0
```

ch01\_overview/10\_dicts.py

Accessing a dictionary in a for loop will return the keys. It is the same effect as `d1.keys()`, which is arguably more understandable but also more verbose.

A `defaultdict()` is a collection that instead of raising a `KeyError` when an invalid key is requested, it will invoke a specified function and return a value from that function.

Example:

```
from collections import defaultdict
d5 = defaultdict(str)
d5['greet1'] = 'hello'
print(d5['greet1'], d5['greet2'])
```

Prints 'hello' and empty string as the `str()` function returns an empty string by default.

# Iterating Over Dictionaries

```
d1 = { 'Smith': 43, 'James': 32, 'Edwards': 36, 'Cramer': 29 }
```

- Iterating a dictionary directly returns keys:

```
for item in d1:  
    print(item)
```

*Smith*  
*Edwards*  
*James*  
*Cramer*

- Iterating a dictionary's values:

```
for val in d1.values():  
    print(val)
```

43  
32  
29  
36

ch01\_overview/10\_dicts.py

The above example indicates how a dictionary used within a `for` control always returns the keys.

# Iterating Dictionaries

- Accessing both key and value simultaneously:

```
for key, val in d1.items():
    print('Key: {}, Value: {}'.format(key, val), end=' ')
```

items() returns a (view of) list of tuples

## Other dict methods

```
d1.copy()
d1.clear()
d1.fromkeys([key_seq])
d1.pop(key, def)
d1.setdefault(key, def)
d1.update(d2)
```

ch01\_overview/10\_dicts.py

Technically speaking, the items() method returns a "view" object instead of a copy of a list of tuples.

fromkeys() returns a dictionary based on the supplied sequence of keys.

setdefault() returns the value for the key or if not found, will add the key and set the value to def. (or None).

update() merges d2 onto d1.

Note about dictionary views:

dict.items(), dict.keys(), and dict.values() all return "views" that are iterable. A view is a read-only iterable, however if the dict the view refers to is changed, the view is changed also.



# Sorted Dictionaries

```
d1 = { 'Smith': 43, 'James': 32, 'Edwards': 36, 'Cramer': 29 }
```

- Dictionaries are unordered and cannot be sorted
- d1.keys() or d1.values() can be sorted

```
sorted(d1.keys())
```

```
['Cramer', 'Edwards', 'James', 'Smith']
```

```
sorted(d1.values())
```

```
[29, 32, 36, 43]
```

- Sorting values according to the keys:

```
list3 = [value for (key, value) in sorted(d1.items())]
```

```
[29, 36, 32, 43]
```

ch01\_overview/10\_dicts.py

In the second example, list3 is made of values sorted according to the keys (alphabetically ascending).



# Python 3.6 Dictionaries Have Order

- In attempting to make dictionaries more compact, their implementation was changed in Python 3.6
- As a result, *keys now have an order*
- Because dictionaries in Python have always been without order, you should continue to treat dictionaries as if they do not have an order for now

The docs state:

The dict type now uses a “compact” representation. The memory usage of the new [dict\(\)](#) is between 20% and 25% smaller compared to Python 3.5.

The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon (this may change in the future, but it is desired to have this new dict implementation in the language for a few releases before changing the language spec to mandate order-preserving semantics for all current and future Python implementations; this also helps preserve backwards-compatibility with older versions of the language where random iteration order is still in effect, e.g. Python 3.5).

For more on this, refer to the documentation:

<https://docs.python.org/3.6/whatsnew/3.6.html#pep-468-preserving-keyword-argument-order>



## Your Turn! - Task 1-4

- Read the **ch01\_overview/alice.txt** file, counting the number of occurrences of words
- Print the top 100 words (5 characters or more) that occur the most

Working with files use:

- Hint: 

```
for line in open(filename, encoding='utf-8'):
    words = line.split()
```

words will be a list of strings

Read all words from the file, add each word into a dictionary. On repeated words, increment a count value.

Afterwards, sort the dictionary by its values.

Follow additional instructions within ch01\_overview/task1\_4\_starter.py

For this task, don't use regular expressions. Also, don't worry about exception handling or closing files as we haven't discussed it yet. As a side note: handling of punctuation and uppercase letters is handled by regular expressions (not discussed in this course, however).

Help: the following sorts a dictionary by its values returning a list of the following: [ (key1, value1), (key2, value2), ... ]

```
sorted( dictionary.items(), key=lambda a:a[1] )
```



# Print Formatting

- The print function has the following syntax:

```
print(val1, val2, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

String inserted  
between values

end= defines a string  
appended to the end of  
the string

Defines where  
output will be  
sent

Force flush the  
stream buffer



# Summary

- Python provides an exception handling system as well as a means for building custom exceptions
- Always use the 'with' control when working with files
- Techniques for comparing, sorting, and managing data are provided via the many built-in functions, classes, and properties



# Chapter 2

## Modular Python

Introductory and Intermediate-Level  
Functions



# Overview

Functions

Default, Keyword Arguments

Modules

*pip*

Introducing Classes



# Function Definitions

- Functions provide a means to write reusable code
  - Python affords powerful features to functions--some are discussed in a later course

```
def func_name(arg0, arg1, arg2, ..., argN):  
    statements  
    return value
```

Function statements must be indented

Return values are optional. A value of 'None' is returned when a return statement is omitted

List of parameters must be supplied or () if none

- Functions must be defined before they can be used

# Calling Functions

- Functions must be defined before they can be called

```
def display_results(customer, purchase_amount):  
    print('Customer: {first} {last}, amount: ${p_amt:,.2f}'  
          .format(first=customer['first'],  
                  last=customer['last'],  
                  p_amt=purchase_amount))  
  
cust = {  
    'first': 'James',  
    'last': 'Smith'  
}  
  
display_results(cust, 1108.23)
```

Customer: James Smith, amount: \$1,108.23

ch02\_functions/01\_functions.py

Functions must be declared (or imported) before they can be called. Parameters passed into the function must also match what is declared in the function.

# Using Default Arguments

```
def convert_file_size(filesize, precision=1, override=None):
    # details of this function are not relevant

print(convert_file_size(12))                                # 0 KB

print(convert_file_size(1200))                               # 1.2 KB

print(convert_file_size(1200, 3, 'MB'))                      # 0.001 MB

print(convert_file_size(12000000000))                      # 11.2 GB

print(convert_file_size(12000000000, 2, 'TB'))            # 0.01 TB

print(convert_file_size(12000000000, 2, 'KB'))            # 11,718,750 KB
```

ch02\_functions/02\_default\_args.py

An explanation of this code:

This function takes a number (filesize) and converts it to either KB, MB, GB, TB, etc. It will automatically return the largest prefix type possible keeping the numeric value between 0 and 1024.

The point of this example is the use of default arguments. Notice that the precision has a default argument set to 1, therefore by default a single decimal place is displayed, but this can be changed by passing a value into the second argument. The override has a None value by default, but can be used to render values using a predetermined unit system.

At the bottom, the `convert_file_size()` function is invoked many different (legal) ways due to the use or overriding of default values.

For the specific details of this function, see the code in the student files.

# Keyword Arguments

- All functions support keyword parameters
  - Function calls become more flexible using this technique

```
def convert_file_size(filesize, precision=1, override=None):  
    # details not shown ...
```

All of these would be valid calls:

```
convert_file_size(1200)  
convert_file_size(1200, 3, 'MB')  
convert_file_size(1200, precision=3, override='MB')  
convert_file_size(filesize=1200, precision=3, override='MB')  
convert_file_size(precision=3, override='MB', filesize=1200)  
convert_file_size	override='MB', filesize=1200)
```

The first two examples rely solely on positional and default arguments. The third example uses mixed arguments: positional & keyword. The remaining 3 examples use keyword arguments exclusively.



# Installing 3<sup>rd</sup> Party Tools: pip

- 3<sup>rd</sup> party tools are external to Python and must be installed
- Most packages can be installed using a special tool called **pip** (python package installation tool)
- To use pip:

To use pip, ensure <PYTHON\_HOME>/Scripts directory is on your path

`pip install package`

# pip install prettytable

`python -m pip install package`

# another way to run pip

`pip uninstall package`

`pip3.6 install package`

# pip3.6 install prettytable

Python 3.6 or later already provides the pip tool

An older, though still used, tool for installing 3<sup>rd</sup> party packages is called `easy_install`. `easy_install` is not set up automatically within older Python versions. Python 3.6 has it already installed.

If you wish to install `easy_install`, visit the following page:

<https://pypi.python.org/pypi/setuptools>

Go to the section on this page that says,  
"Installation Instructions"

The recommended way to bootstrap setuptools on any system is to download `ez_setup.py` and run..."

Download the `setup.py` file and run it using the `python` command. You will need to ensure your <PYTHON\_HOME>/Scripts directory is on your path.



# What Is PyPI?

- PyPI is the *Python Package Index Repository*
- Contains 3<sup>rd</sup> party resources
  - Sometimes called the CheeseShop  
(named after a *Monty Python* skit)

<http://pypi.python.org/pypi>

pip search package

# pip search prettytable



Performs a search on PyPI

The Python Package Index contains nearly 100000 3<sup>rd</sup> party packages for use or download with varying licensing, documentation, support, etc.



# Introducing Modules

- Modules are **namespaces** in Python
  - Physically, each **.py** file represents a module
  - **Functions, variables, classes** declared at the top of a module can be made available to other modules
  - These **attributes** must be imported before being used
- Most modules of Python's standard library are found in the **<PYTHON\_HOME>/Lib** directory

```
import os  
import sys  
print(environ, path)
```



```
import os  
import sys  
print(os.environ, sys.path)
```



Creates a variable with properties representing top-level items declared within this module

# Other Import Methods

`from <moduleName> import <attributes>`

`from <moduleName> import <attributes> as <alias>`

```
from os import listdir
```

Does not create a top-level variable name for the module itself

Creates a variable called listdir, points it to the resource found in this module

Use `dir(moduleName)` to determine what can be found in a module:

```
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__clear_type_cache__', '__current_frames__', '__debugmallocstats__', '__getframe',
 '__home__', '__mercurial__', '__options__', 'api_version', 'argv', 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle', 'dont_write_bytecode', 'exc_info', 'exec_',
 'exechook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info', 'float_repr_style', 'getallocatedblocks', 'getcheckinterval',
 'getdefaultencoding', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',
 'getswitchinterval', 'gettrace', 'getwindowsversion', 'hash_info', 'hexversion', 'implementation', 'int_info', 'intern',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix',
 'ps1', 'ps2', 'setcheckinterval', 'setprofile', 'setrecursionlimit', 'setsswitchinterval', 'settrace', 'stderr', 'stdin',
 'stdout', 'thread_info', 'version', 'version_info', 'warnoptions', 'winver']
```

To determine the content of a module, issue the `dir( )` command after importing the module.

# Python Classes

```

class Contact:
    """ Defines a Contact type """
    def __init__(self, name='', address='', phone='',
                 email='', company='', position):
        self.name = name
        self.address = address
        self.phone = phone
        self.email = email
        self.company = company
        self.position = position

    def __str__(self):
        return '{0}'.format(self.name)

c = Contact('John Smith', '123 Main St.', '(970) 322-9088',
             'jsmith433@yahoo.com', 'Acme Inc.',
             'Rubber Hole Engineer')

c.alt_email = 'jsmith433@gmail.com'
print(c.name, c.alt_email)
print(c, type(c))

```

Instantiation

John Smith jsmith433@gmail.com  
 John Smith <class '\_\_main\_\_.Contact'>

ch02\_modular/03\_classes.py

This example illustrates a Python class. It contains a couple of magic methods, `__init__` and `__str__`. These will be discussed shortly. The class is used to "instantiate" an object. The instantiation occurs on the line: `c = Contact()`. This creates an object which, after a type check, shows that it is a Contact type.

Note that in other languages you might be tempted to use the `new` operator. But Python doesn't define a new operator. So this statement:

`c = new Contact( )`

is actually

`c = Contact( )` in Python

Notice that, like any object, properties such as `alt_email` can be added at any time.



# Summary

- Python provides several tools to improve reusability and modularization:
  - Modules
  - Classes
  - Functions
- Functions support default arguments, keyword arguments, multiple positional arguments, and multiple keyword arguments
- Custom module locations should be identified on the PYTHONPATH
- Classes provide a similar functionality to modules when it comes to organizing code structure

# Chapter 3

## Numerical Analysis

### Using NumPy

Introducing Data Analysis Tools and  
Techniques in Python

# Overview

The Anaconda Distribution

IPython and Jupyter Notebooks

NumPy



# The Anaconda Python Distribution

- Anaconda is a Python (and R) distribution used within **data science** application development
  - Distribution is provided by *Continuum Analytics*
    - Ships with a BSD license for Windows, Linux, OS X
    - Valid for Python 2.7, 3.5, 3.6, 3.7 (as of this writing)
  - Ships with over **100 pre-installed packages**
- Anaconda provides its own package installer:  
**conda**

```
conda install <packagename>
```

Example: `conda install paramiko`

Other conda subcommands include: remove (uninstall), update (upgrade), list, search, info, create

Anaconda is a Python-based data science platform featuring numerous pre-installed Python packages and hundreds of additional packages that may be installed using the provided conda package management tool.

The conda package installer/build tool will ping its own repository (<https://api.anaconda.org>) rather than relying on PyPI.

conda can also be used to create environments with installed packages using:

```
conda create --name envName package1 package2 ...
```

It may be necessary to run the conda command as administrator.



# Anaconda Packages

Some of the available packages in the Anaconda3 distribution

<code>astropy</code>	- astronomy	<code>notebook</code>	- interactive computing
<code>babel</code>	- internationalization	<code>numpy</code>	- number & array processing
<code>beautifulsoup4</code>	- html parsing	<code>openpyxl</code>	- work with .xls files
<code>bitarray</code>	- arrays of Booleans	<code>pandas</code>	- data analysis tools
<code>blaze</code>	- big data for numpy, pandas	<code>pathlib2</code>	- file path manipulation
<code>boto</code>	- AWS tools	<code>path.py</code>	- os.path wrapper
<code>bottleneck</code>	- fast numpy arrays	<code>pep8, pyflakes, pylink</code>	- style checkers
<code>chest</code>	- like shelve	<code>pexpect</code>	- command line manager
<code>cloudpickle, pickleshare</code>	- pickling tools	<code>pymysql, pyodbc</code>	- db tools
<code>colorama</code>	- colored text	<code>pywin32</code>	- Windows extensions
<code>configobj</code>	- file reading/writing	<code>pyyaml</code>	- yaml support
<code>decorator</code>	- decorators	<code>pyreadline, readline</code>	- read files
<code>docutils</code>	- tools for doc generation	<code>redis</code>	- key-value database
<code>fastcache</code>	- faster cache	<code>scipy</code>	- scientific tools
<code>flask</code>	- web microframework	<code>six</code>	- 2, 3 compatibility
<code>gevent/greenlets</code>	- concurrency	<code>sphinx</code>	- documentation generator
<code>imagesize jpeg, pillow</code>	- image manipulation	<code>spyder</code>	- scientific tools
<code>ipython, jupyter</code>	- interactive shell	<code>sqlalchemy</code>	- Python-database ORM
<code>jinja2</code>	- templating	<code>statsmodels</code>	- statistical models
<code>jsonpath</code>	- json query language	<code>tornado</code>	- asynchronous server
<code>lxml</code>	- xml parser	<code>unicodecsv</code>	- csv unicode strings
<code>markupsafe</code>	- xml as strings	<code>xlsxwriter</code>	- create xls files
<code>matplotlib</code>	- 2D plots	<code>yaml</code>	- yaml manipulation/creation
<code>nose, pytest</code>	- testing	<code>pip, setuptools</code>	- package management

These are just a few of the modules available in the Anaconda distribution. Keep in mind that hundreds of additional modules may be installed using the conda tool later.



# IPython and Jupyter

- IPython is a Python-based interactive computing environment
  - Coupled with Jupyter Notebooks, it is possible to share web-based projects
  - Some publicly shared Jupyter Notebooks:

## A gallery of interesting Jupyter Notebooks

Scott Cole edited this page on Jul 31 · 68 revisions

This page is a curated collection of Jupyter/IPython notebooks that are notable. Feel free to add new content here, but please try to only include links to notebooks that include interesting visual or technical content; this should *not* simply be a dump of a Google search on every ipynb file out there.

**Important contribution instructions:** If you add new content, please ensure that for any notebook you link to, the link is to the rendered version using nbviewer, rather than the raw file. Simply paste the notebook URL in the nbviewer box and copy the resulting URL of the rendered version. This will make it much easier for visitors to be able to immediately access the new content.

Note that Matt Davis has conveniently written a set of [bookmarks and extensions](#) to make it a one-click affair to load a Notebook URL into your browser of choice, directly opening into nbviewer.

## Table of Contents

1. Entire books or other large collections of notebooks on a topic
  - Introductory Tutorials
  - Programming and Computer Science
  - Statistics, Machine Learning and Data Science
  - Mathematics, Physics, Chemistry, Biology
  - Earth Science and Geo-Spatial data
  - Linguistics and Text Mining
  - Signal Processing
  - Engineering Education
2. Scientific computing and data analysis with the SciPy Stack
  - General topics in scientific computing

<https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks>

IPython is a Python-based kernel which provides the engine for Jupyter. Jupyter Notebook provides the non-Python features such as notebook creation and management, qtconsole, web-based interface, and more. IPython provides only the Python kernel.

Typing `ipython` from a command-line launches an IPython interactive shell. Other ways to start IPython include:

`ipython qtconsole` (which starts an enhanced console in a new GUI window)

`ipython notebook` (which is deprecated and replaced by techniques mentioned on subsequent slides)

Two other noteworthy notebook galleries:

<http://nbviewer.jupyter.org/>

<https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks>



# Launching Jupyter

- Anaconda ships with IPython built into it
  - If not using Anaconda, other distributions may install separately:

`pip install ipython`

`pip install jupyter`

- Launch a notebook (from a specific directory):

`jupyter notebook`

```
c:\temp\notebooks>jupyter notebook
[W 14:12:24.203 NotebookApp] Unrecognized JSON config file version, assuming version 1
[I 14:12:25.132 NotebookApp] [nb_conda_kernels] enabled, 1 kernels found
[I 14:12:25.576 NotebookApp] [nb_anacondacloud] enabled
[I 14:12:25.581 NotebookApp] [nb_conda] enabled
[I 14:12:25.647 NotebookApp] \u2713 nbpresent HTML export ENABLED
[W 14:12:25.648 NotebookApp] \u2717 nbpresent PDF export DISABLED: No module named 'nbbrowserpdf'
[I 14:12:25.767 NotebookApp] Serving notebooks from local directory: c:\temp\notebooks
                           <App> 0 active kernels
                           <App> The Jupyter Notebook is running at: http://localhost:8888
                           <App> Use Control-C to stop this server and shut down all confirmation).
```

Once the server launches, it can be accessed via the browser at:  
**localhost:8888** (by default)

While IPython provides an interactive shell (and prompt), it can also run from within a browser. To do this, you can use the command: `jupyter notebook`.

Options include:

- runtime\_dir (shows the location of the runtime directory)
- data\_dir (shows the location of the data directory)
- config\_dir (shows the location of the config file directory)

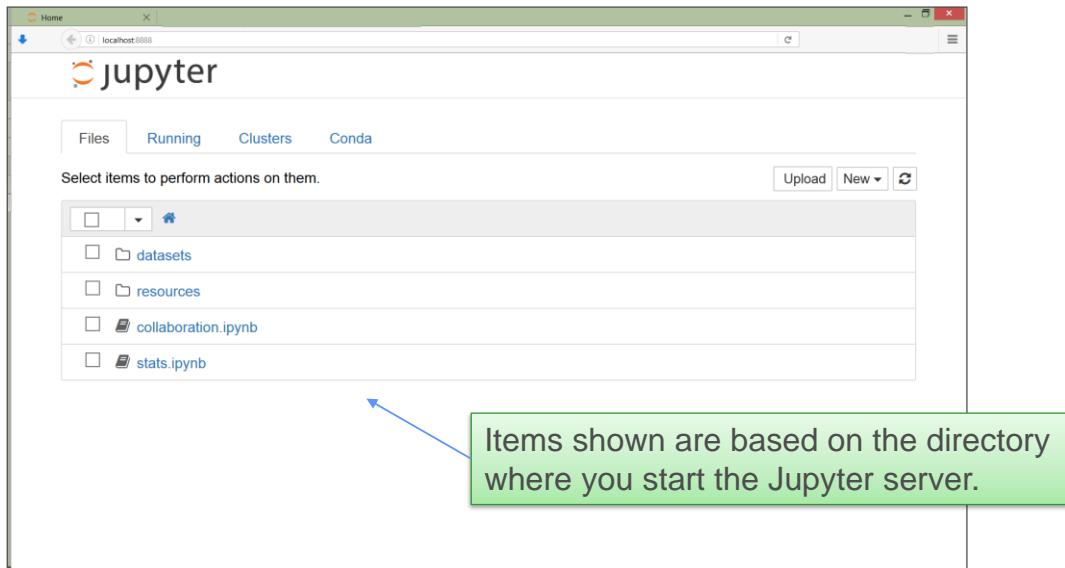
To override the defaults, set environment variables JUPYTER\_RUNTIME\_DIR, JUPYTER\_PATH, and JUPYTER\_CONFIG\_DIR respectively.

Additional startup options include:

- help (to provide additional help with commands)
- port 8889 (to launch the server on a different port)
- no-browser (to avoid launching the browser when the server starts)
- version (displays the version number)
- paths (displays all Jupyter-related paths)
- json (prints paths in JSON format)

# Jupyter Dashboard

- The web-based dashboard allows for creating and running code in notebooks



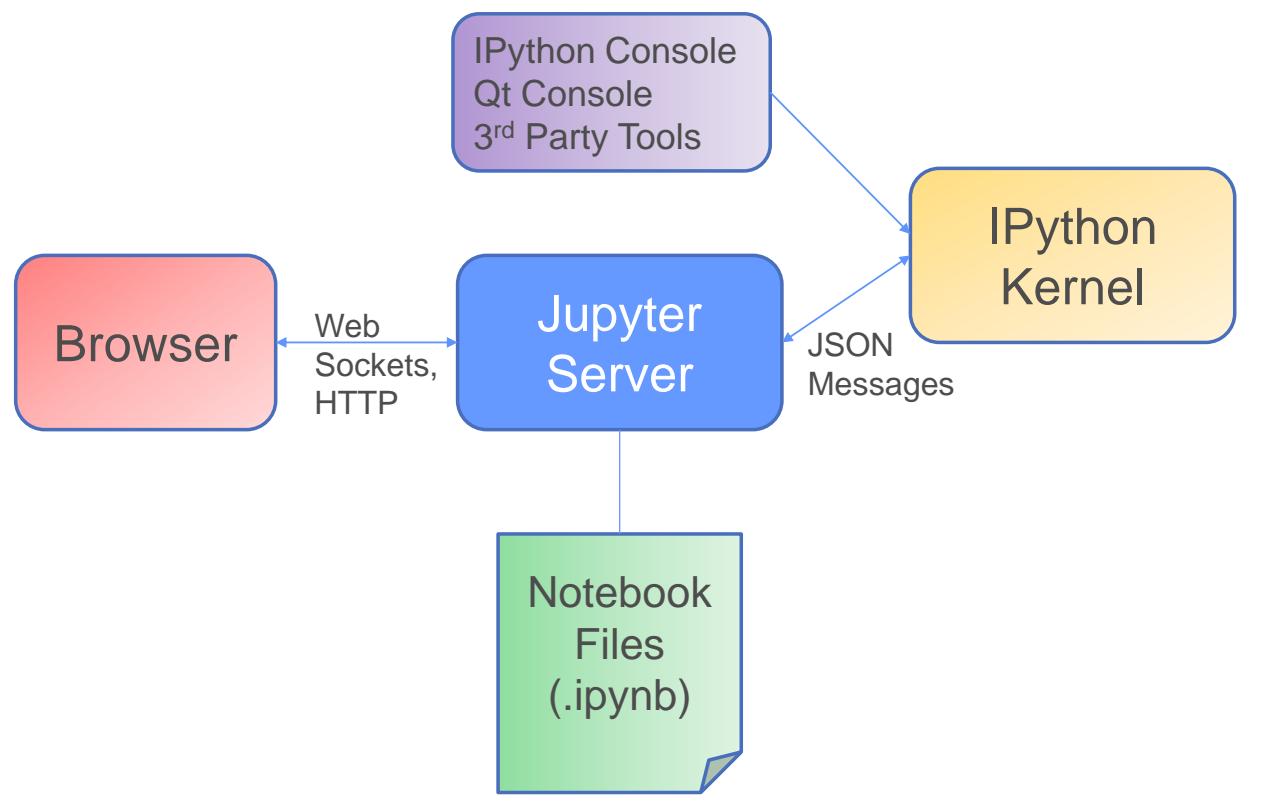
You should execute the `jupyter notebook` command from the directory you wish to start your Jupyter server. After starting Jupyter, the dashboard will appear when you browse to `localhost:8888` in your browser.

Note: In moving from version 3 to version 4, IPython separated from Jupyter. This resulted in numerous differences and changes in both configuration and commands. A first run of the Jupyter server (tool) should cause any older resources to automatically migrate. For more on this see:  
<https://blog.jupyter.org/2015/04/15/the-big-split/>.

Jupyter 4 projects did not inherit all things from IPython, therefore configuration has changed from earlier versions. Under version 4, by default, Jupyter config files can be found in `~/.jupyter`. Previously (IPython 3), these would have been located in `~/.ipython`.

To start Jupyter with a custom config file, invoke `jupyter notebook --generate-config` and then modify the settings found in `jupyter_notebook_config.py` found in the config directory (mentioned on the previous slide).

# IPython Jupyter Relationship



The Jupyter server, combined with the user's browser, acts as another form of frontend to the IPython kernel. The Jupyter server is responsible for loading and saving .ipynb files from the file system.



# Notebook Components

- Notebooks are documents that contain live code, images, video, text, graphs, widgets and more
  - Notebooks can be run from within the browser using different kernels (programming languages)
  - Create .ipynb files on the local file system
- Notebooks are made up of cells
- Two most common cells:
  - Markdown cells (narrative LaTeX-based text)
  - Code cells (cells containing code from a language)

Heading cells have been deprecated now in favor of using markdown cells. Use # through ##### to create <h1> through <h6> header elements.

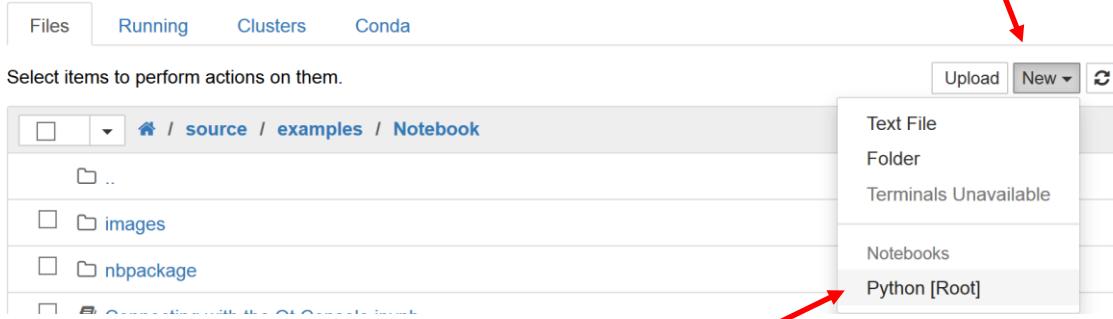
Markdown cells support HTML and LaTeX formatting.

Here's some markdown cell rules:

```
<http://someurl>
*this makes test italic*
**this makes it bold**
> This is a blockquote
1. This is an ordered list
+ This increments the list
* This is an unordered list
* This increments it
![alt_text](http://link_to_image)
```

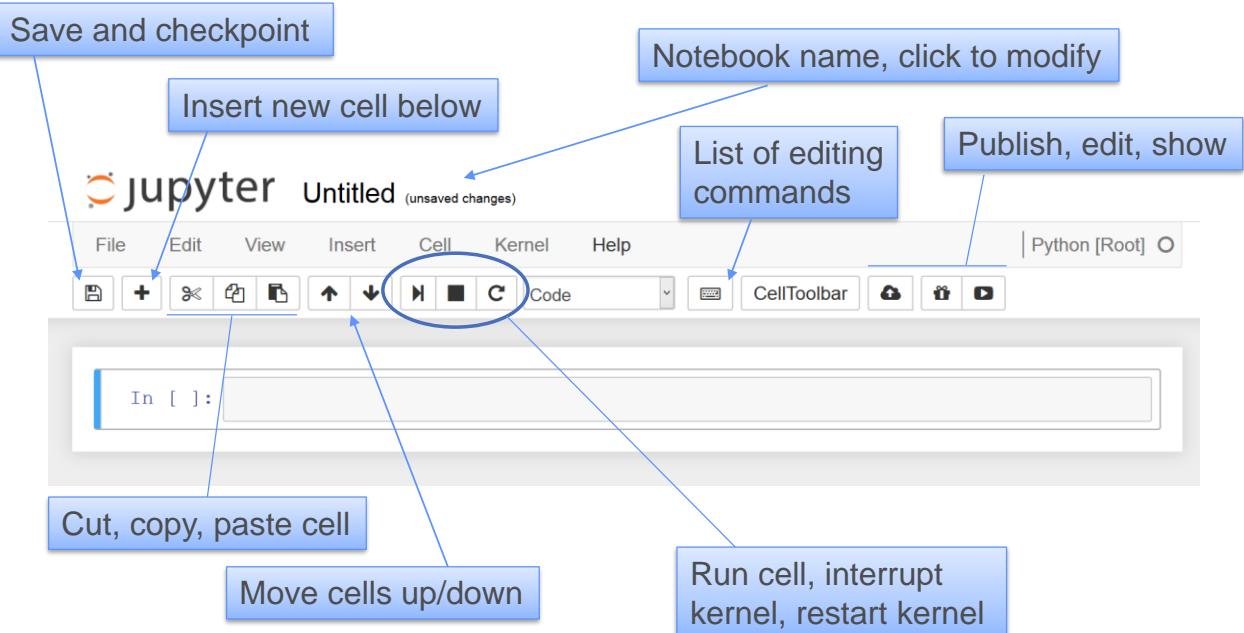
# Creating Notebooks

- Create a new notebook by selecting "New" from the dropdown on the right



# Notebook Cells Toolbar

- The Notebook menu and toolbar provide the ability to modify, execute, and save Notebooks



Note: Your student files contain some sample tutorials on Notebook creation and management in the form of Notebooks! To view these:

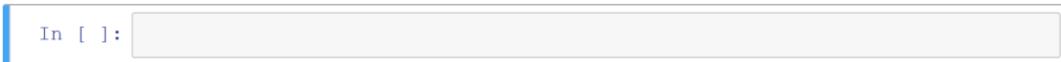
- Open a command window,
- Browse (cd) to <student\_files>/ch03\_numpy/notebook\_docs
- Run jupyter notebook
- Within a browser, drill down into source > examples > Notebook



# Notebook Editor Modes

- **Command mode**

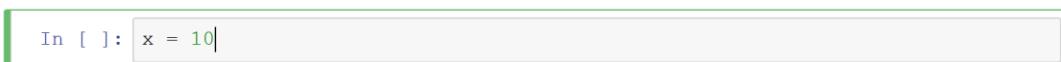
ESC key on a cell initiates this mode



- In this mode, a cell responds to special commands  
(see the list of editing commands )

- **Edit mode**

Enter key on a cell initiates this mode



- In this mode, a cell can be typed into (like an editor)

Mouse clicking in the appropriate areas will also cause the cell to go into Edit or Command Mode

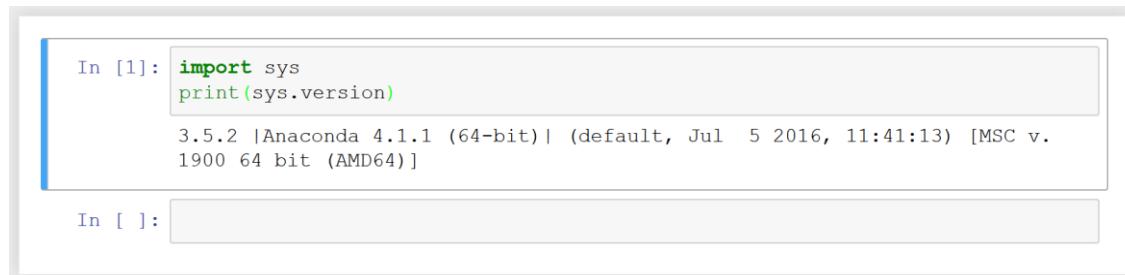
The notebook editor supports two modes: command and edit. In command mode, special commands exist that can quickly perform tasks, such as deleting a cell. To delete a cell, select it, make sure the right margin is blue (as shown in the diagram above), and then press d-d (in somewhat rapid succession).

Additional useful command mode commands include:

b - creates a cell below  
dd - deletes the current cell  
h - to get more commands

# Running Notebooks

- Pressing **Shift-Enter** or clicking  while the selected cell is in command mode will run the cell

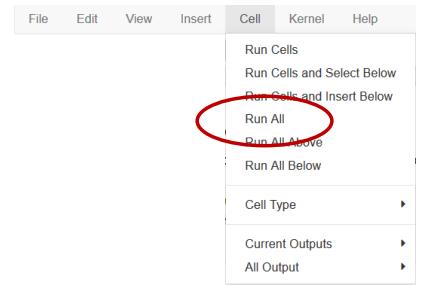


In [1]: `import sys  
print(sys.version)`

3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 5 2016, 11:41:13) [MSC v. 1900 64 bit (AMD64)]

In [ ]:

- To run all cells, use the menu item:



Some other useful IPython and Jupyter hints:

Use ?? to view a function's source code, as in:

`np.linspace??`

<-- type into a cell and run that cell



# Markdown Notation

- **Markdown** cells allow for rendering text into notebooks
  - Markdown notation include:

H1      #

H2      ##

H3      ###

H4      #####

H5      #####

H6      #####

Headers

Lists

1. Ordered lists

+ Increments the ordered list

\* Unordered lists

\*Some Italic Text\*

Italic Text

![Some link](http://url)

\*\*Some BoldText\*\*

Bold Text

Latex

\$inline\_expr\$

\$\$expr\\_on\\_own\\_line\$\$

Markdown syntax within cells can be created using shortcuts like those shown on the slide.

Latex expressions (discussed later) provide a way of depicting mathematical equations in a prettier format. A single dollar sign allows expressions to be placed inline with other text. Using a double dollar sign creates expressions on their own line.



# IPython/Jupyter Magic Commands

- Numerous "special" commands, often called **magic commands**, can be used within the Jupyter browser interface:

```
%matplotlib inline
```

- These commands often begin with a % or %% syntax

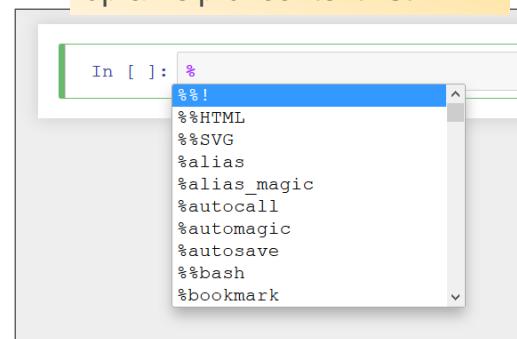
The following URL provides a summary of magic commands:  
<https://ipython.org/ipython-doc/3/interactive/magics.html>



# Some Magic Commands

%cd	Change the working directory
%config	Run this to see which objects can be configured
%config <obj>	Run this to see which properties within an obj can be configured
%history	View a history of commands
%load <file>	Loads a file into the cell
%matplotlib	Configure matplotlib features
%pprint	Toggle pretty printing
%precision 3	Sets the output precision
%quickref	Magic Command Quick Reference
%sx dir .	Perform a system command
!!dir .	!! performs a system command also
%timeit <stmt>	Performs a Python timeit execution
%%capture	Captures stdout, stderr
%%writefile [-a] <file>	Write contents of cell to a file

Note: pressing <Tab> brings up a helpful context list



Note: to debug in Jupyter, you can pip install ipdb. Place the following (boldfaced lines) into your code:

```
from IPython.core.debugger import Tracer
a=4
Tracer()()    <-- debugger will activate here, commands below may then be used
b=5
```

Commands include: p <var\_name>, n (next stmt), s (step into), exit (to quit)

You can find a cheatsheet for ipdb here:  
<https://nblock.org/2011/11/15/pdb-cheatsheet/>

Timeit and capture example:

```
%%capture out
%timeit -n 4 -r 4 print('hello')
out.show()
```



# Code Assistance within Jupyter

- Jupyter supports viewing "popup" code assistance:

Now to select the temperature data and calculate the averages...

```
In [9]: temps = arr[:, (1,2)].astype(float)
average_daily = temps.mean(axis=1)
average_daily
Out[9]: array([ 78.,  74.5,  76.,  78.,  81.]
In [12]: np.column_stack?
```

Type a '?' and then Shift-Enter to bring up a docked "window" displaying documentation

**Signature:** np.column\_stack(tup)  
**Docstring:**  
Stack 1-D arrays as columns into a 2-D array.  
Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with hstack . 1-D arrays are turned into 2-D columns first.  
**Parameters**  
-----  
tup : sequence of 1-D or 2-D arrays.  
Arrays to stack. All of them must have the same first dimension.  
**Returns**

Now to select the temperature data and calculate the averages...

```
In [9]: temps = arr[:, (1,2)].astype(float)
average_daily = temps.mean(axis=1)
Out[9]:
In [12]: np.astype?
In [ ]:
```

Shift-Tab-Tab will extend a "popup" for lines of code you've already completed

**Docstring:**  
Copy of the array, cast to a specified type.  
**Parameters**  
-----  
dtype : str or dtype  
Typecode or data-type to which the array is cast.  
order : {'C', 'F', 'A', 'K'}, optional

Documentation can be viewed in Jupyter by pressing Shift-Enter after typing a question mark. This is best done while you are entering code. On the other hand, Shift-Tab (with an optional second Tab press) will examine the cursor's location and provide documentation for the attribute/method found there.



# What Are NumPy and SciPy?



- **NumPy** is a Python-based framework providing multi-dimensional array creation and manipulation tools
- **SciPy** is a set of extension frameworks that can support and aid NumPy:
  - SciPy
  - Matplotlib
  - IPython
  - Pandas
  - Sympy
  - Nose

**Python Distributions That Include the SciPy Stack**

Anaconda	400+ Math/science packages
Canopy	Free & commercial
Python(x,y)	Free, incl. Spyder, Qt, SciPy Stack
WinPython	Free, Windows only
Pyzo	Free Python alternative to Matlab

If not using a Python distribution that includes the full SciPy stack, then additional downloads are necessary. These individual downloads have different installations per tool. You should visit each tool's homepage for installation instructions for each one.

SciPy actually imports NumPy when it loads (refer to the `scipy __init__.py` file).

# The NumPy Array

- NumPy's main object is the array (ndarray type)
  - Array dimensions in NumPy are called **axes**
  - The number of axes is its **rank**

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
```

axes = 1 or rank = 1  
Length (of axis 0): 5

```
arr2 = np.array([[1, 2, 3, 4, 5],
                 [6, 7, 8, 9, 10]])
```

axes = 2 or rank = 2  
Length of axis 0: 2  
Length of axis 1: 5

```
arr3 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9],
                 [10, 11, 12]])
```

axes = 2 or rank = 2  
Length (of axis 0): 4  
Length (of axis 1): 3  
Shape: (4, 3)

ch03\_numpy/01\_numpy\_array.py

Note: axes has nothing to do with coordinates in space. Here it defines the number of subscripts needed to access a value within the array:

arr[0][2][1]

This would indicate an array with 3 axes (a 3 dimensional array).

# Array Terms (Definitions)

- An **ndarray** (NumPy array type) has several attributes:
  - ndarray.ndim** number of axes or dimensions or rank
  - ndarray.shape** length of array in each dimension
    - For m rows and n columns shape will be (m, n)
  - ndarray.size** total number of elements
  - ndarray.strides** number of bytes to get to the next element in that dimension

```
arr3 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9],
                 [10, 11, 12]])

print('Shape: {0}, Size:{1}, Axes:{2}, Types:{3}, Strides:{4}'
      .format(arr3.shape, arr3.size, arr3.ndim, arr3.dtype,
              arr3.strides))
```

Shape: (4, 3), Size: 12, Axes: 2,  
Types: int32, Strides: (12, 4)

ch03\_numpy/01\_numpy\_array.py

Displaying array values is easy:

print(arr3)

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

print(arr3[0])

[1 2 3]

print(arr3[0][2])

3

In the example, we have a 4 x 3 array. Each element is an int32 or 4 bytes in length. This means it takes 4 bytes to get to the next element (column). There are 3 elements per row, so  $3 \times 4$  bytes = 12 bytes. It would take 12 bytes to get to the next row. Therefore, the strides of arr3 would be described as (12, 4)

# Array Values and dtype

- NumPy arrays are homogeneous

```
import numpy as np

arr10 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])

arr11 = np.array([[1, 2, 3],
                  [4, 5., 6],
                  [7, 8, 9],
                  [10, 11, 12]])

arr12 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]], dtype=float)

print(arr10.dtype, arr11.dtype,
      arr12.dtype) # int32 float64 float64
```

The types within a NumPy array are described (or set) via the **dtype** attribute

The array value types can be specified using the **dtype** argument

ch03\_numpy/02\_numpy\_dtotypes.py

NumPy's data types extend beyond what Python normally has to offer. NumPy includes `bool_`, `int_`, `int8` (-128 to 127), `int16` (-32768 to 32767), `int32`, `int64`, `uint8` (0 to 255), `uint16` (0 to 65535), `uint32`, `uint64`, `float_`, `float16`, `float32`, `float64`, `complex64`, `complex128`. Use them by specifying the module name followed by the `dtype`, as in `np.int16`.

In addition to establishing the `dtype` (as shown in the third example above), each `dtype` can be invoked like a function:

```
arr13 = np.float64([1, 3, 5]) # arr13.dtype = float64
```

## np.astype()

- **astype()** can be used to change the array types

```
arr10 = np.array([[1, 2, 3],  
                  [4, 5, 6],  
                  [7, 8, 9],  
                  [10, 11, 12]])  
  
print(arr10.dtype)                      # int32  
arr_new = arr10.astype(np.float64)       # new array created  
print(arr_new.dtype)                    # float64
```

ch03\_numpy/02\_numpy\_dtotypes.py

NumPy array types are fixed once created and can't be changed. Attempting to change the dtype causes a new array to be created. In fact, the copy=False argument will be ignored in such cases. In the above example, the following would be true:

```
arr_new is arr10 == False
```

# Creating Arrays

- Several approaches exist for creating NumPy arrays:

```
import numpy as np

arr4 = np.zeros((2,2))
print(arr4)                                     # [[ 0.  0.] [ 0.  0.]]

arr5 = np.ones((2,2))
print(arr5)                                     # [[ 1.  1.] [ 1.  1.]]

arr6 = np.full((2,2), 6)                         # np.full(shape, fill_val)
print(arr6)                                     # [[ 6.  6.], [ 6.  6.]]

arr7 = np.eye(2)
print(arr7)                                     # 2x2 identity matrix

arr8 = np.empty((2,2))
print(arr8)                                     # 2x2 random numbers
```

ch03\_numpy/01\_numpy\_array.py

Shown here are additional ways of initializing arrays in NumPy. Zeros fills the specified shaped array with all zero values. Ones does the same with one values. Full fills it with a specified value. Eye creates an identity matrix of that size. Empty leaves random values in the array.

Note that **np.identity(3)** generates a 3 x 3 identity matrix also. eye() can additionally perform an offset as follows:

<code>np.eye(3)</code>	<code>[[1.  0.  0.]</code> <code>[0.  1.  0.]</code> <code>[0.  0.  1.]]</code>
------------------------	---

---

<code>np.eye(3, 3, 1)</code> <i># defines 3 rows, 3 cols, and the diagonal offset:</i>	<code>[[0.  1.  0.]</code> <code>[0.  0.  1.]</code> <code>[0.  0.  0.]]</code>
--	---

---

<code>np.eye(3, k=-1)</code> <i># defines 3 rows/cols, diagonal offset is negative:</i>	<code>[[0.  0.  0.]</code> <code>[1.  0.  0.]</code> <code>[0.  1.  0.]]</code>
---	---

# Other Array Creation Techniques

- **linspace()** - creates an array with `num_items` number of evenly spaced elements

```
np.linspace(start, end, num_items)
```

end IS included!

```
arr14 = np.linspace(0, 20, 5)      # [ 0.  5. 10. 15. 20. ]
```

- **arange()** - generates array values—similar to Python's `range()`

```
np.arange(start, end, step)
```

end NOT included!

```
arr15 = np.arange(0, 15, 3)      # [0 3 6 9 12]
```

- **diag()** - creates a diagonal array

```
np.diag([vector_values])
```

```
arr16 = np.diag(np.arange(1, 6, 2))  # [[1 0 0] [0 3 0] [0 0 5]]
```

ch03\_numpy/03\_array\_functions.py

`num_items` for `linspace()` should be an int.

A technique, not shown in the slide above, is `np.random.rand(dimension_size1, dim_size2, ...)`. Use this to generate an array of random values from the specified shape. Example: `np.random.rand(2, 3)` generates:

```
[[ 0.30623218 0.26506357 0.19606006] [ 0.43052148 0.02311355 0.19578192]]
```

Other array creation methods include: `logspace()`, `meshgrid()`, `fromfunction()`, `fromfile()`, `genfromtxt()`, `loadtxt()`. Some of these are discussed later.



# Combining Arrays: Stacking

- `np.stack([arrs], axis)` - joins arrays along a specified axis

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
result = np.stack([v1, v2])
```

When not specified,  
default axis is 0

[1 2 3]

v1

[4 5 6]

v2



[[1 2 3]
[4 5 6]]

result

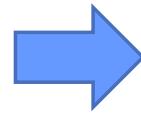
```
result = np.stack([v1, v2], axis=1)
```

[1 2 3]

v1

[4 5 6]

v2



[[1 4]
[2 5]
[3 6]]

result

ch03\_numpy/15\_stacking.py

`np.stack()` can join a sequence of arrays along a specified axis.

## Array Stacking (continued)

- `np.column_stack([arrs])` - stacks 1D arrays as columns:

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
result = np.column_stack([v1, v2])
```

[1 2 3]  
v1

[4 5 6]  
v2



[[1 4]  
[2 5]  
[3 6]]  
result

- `np.hstack([arrs])` - stacks arrays in seq. horizontally  
*(column wise)*

`np.hstack((v1, v2))`

← [1 2 3 4 5 6]

`np.hstack((result, result))`

← [[1 4 1 4]  
[2 5 2 5]  
[3 6 3 6]]

ch03\_numpy/15\_stacking.py

`column_stack()` stacks along the second axis (`axis=1`).

# Array Stacking (continued)

- **np.vstack()** - stacks arrays in seq. vertically (row wise)

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
result = np.vstack([v1, v2])
```

[1 2 3]

v1

[4 5 6]

v2



[[1 2 3]  
[4 5 6]]

result

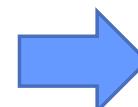
- **np.concatenate([arrs], axis)**

The previous stacking methods are all convenience methods that wrap concatenate()

```
np.concatenate([v1, v2])
```

[1 2 3]

[4 5 6]



[1 2 3 4 5 6]

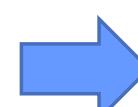
```
np.concatenate([result, result])
```

[[1 2 3]

[4 5 6]]

[[1 2 3]

[4 5 6]]



[[1 2 3]  
[4 5 6]  
[1 2 3]  
[4 5 6]]

ch03\_numpy/15\_stacking.py

np.stack([arrs], axis=0) and np.vstack([arrs]) are similar.

np.concatenate([arrs], axis=1) and np.hstack([arrs]) are similar. In the bottom example, the following:

np.concatenate([result, result], axis=1) yields the following result:

```
[[1, 2, 3, 1, 2, 3],
 [4, 5, 6, 4, 5, 6]]
```

The vstack(), hstack(), column\_stack(), and append() methods are just convenience functions (wrappers) for np.concatenate().

# Array Shifting

- **np.roll(arr, n, axis)** - causes the elements of the array to rotate n elements along axis
  - By default, arr is flattened first if no axis is specified

```
v1 = np.array([1, 2, 3])           [[1 2]
a1 = np.array([[1, 2], [3, 4], [5, 6]])    [3 4]
                                         [5 6]]
```

a1

<code>np.roll(v1, 1)</code>	[3 1 2]	Shifts right one
<code>np.roll(v1, -1)</code>	[2 3 1]	Shifts left one
<code>np.roll(a1, -1, axis=0)</code>	[[3 4] [5 6] [1 2]]	Shifts rows up one
<code>np.roll(a1, 1, axis=1)</code>	[[2 1] [4 3] [6 5]]	Shifts columns right one

ch03\_numpy/15\_stacking.py

The roll() function can shift elements in the specified axis direction. If no axis is specified, it will be flattened first. Negative shift values may be used which causes either a shift upward or to the left.

# Accessing (Indexing) Values

- Access arrays using subscripts
  - A single pair of square brackets works
  - Use Python's slice notation also

```
arr19 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])
```

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

arr19[3, 1]  
arr19[3][1]

	-3	-2	-1
-4	1	2	3
-3	4	5	6
-2	7	8	9
-1	10	11	12

arr19[-1]  
arr19[-3, -3]

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

arr19[2:, 1:]

ch03\_numpy/04\_accessing\_arrays.py

The examples show (from left to right) the use of direct access with a single pair of square brackets, negative index values used, and slice notation.

## More with Slicing: Stepping

- `np.tile(arr, reps)` - repeats an array reps times
  - Reps can be an array indicating the number of repeats in each dimension

```
arr20 = np.tile([1, 2, 3, 4], [4, 1])
```

	0	1	2	3	4
1	1	2	3	4	
2	1	2	3	4	
3	1	2	3	4	

arr20[1::2, 1::2] # [[2 4] [2 4]]

- Slicing operations create a *view* of the original array (a copy of the slice is *not* made)

In the example shown here, you can use slicing with each axis. Here we sliced the first axis, or rows, and then we took those rows and further sliced the second axis, or columns. The slice notation is `array_name[ start : stop : step ]`.

To illustrate the slicing view concept, consider the following example:

```
import numpy as np
a = np.ones((3,3))
b = a[:,1]
b += 1
print(a)          [[ 1.  2.  1.]
                   [ 1.  2.  1.]
                   [ 1.  2.  1.]]
```

# Selecting, Modifying Columns

Use slicing to modify multiple values at once

```
arr20[:,1] = 8
```

	0	1	2	3
0	1	8	3	4
1	1	8	3	4
2	1	8	3	4
3	1	8	3	4

Slicing & selecting specific column indices

```
arr20[:, [0,3]]
```

	0	1	2	3
0	1	8	3	4
1	1	8	3	4
2	1	8	3	4
3	1	8	3	4

Selecting specific rows and all columns

```
arr20[(1, 2), :]
```

	0	1	2	3
0	1	8	3	4
1	1	8	3	4
2	1	8	3	4
3	1	8	3	4

ch03\_numpy/04\_accessing\_arrays.py

These examples illustrate the use of slicing and selecting along a specified axis. Modifying an entire column is easily accomplished with slicing. Selecting columns using specific column indices or via Boolean-valued arrays is valid (as shown in the second example above).

# Selecting Using Booleans

Slicing & selecting via Boolean-valued arrays

```
arr20[:,  
       np.array([True, False, True, False])]
```

	0	1	2	3
0	1	8	3	4
1	1	8	3	4
2	1	8	3	4
3	1	8	3	4

Slicing & selecting via Boolean-valued arrays

```
arr20[:, np.array([False, False, False, True])]  
= [[6], [7], [8], [9]]
```



	0	1	2	3
0	1	8	3	4
1	1	8	3	4
2	1	8	3	4
3	1	8	3	4

	0	1	2	3
0	1	8	3	6
1	1	8	3	7
2	1	8	3	8
3	1	8	3	9

ch03\_numpy/04\_accessing\_arrays.py

Selecting columns using Boolean-valued arrays will select only the columns that are True.

# Array Element Math Operations

- NumPy arrays support basic arithmetic operations

```
arr22 = np.array([[1, 2], [3, 4]])
arr23 = np.array([[5, 6], [7, 8]])
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

```
arr24 = arr22 + arr23
```

Same as np.add(arr22, arr23)

$$\begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

```
arr26 = arr22 - arr23
```

Same as np.subtract(arr22, arr23)

$$\begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}$$

```
arr27 = arr22 * arr23
```

Same as np.multiply(arr22, arr23)

$$\begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

Do arrays have to  
be the same size?

ch03\_numpy/05\_array\_math.py

The addition, subtraction, and multiplication operators shown here perform **element-by-element** operations. Note that, if necessary, the data type (dtype) might get promoted if the operation warrants it.

For arrays of different sizes use rules of array broadcasting. There are a few rules associated with array broadcasting. Array shapes **must match exactly**, or the array dimensions (working backward) must match or be a 1. The resulting shape after the operation will be the size of the maximum array.

$$\begin{bmatrix} [2. 3. 4.] \\ [4. 5. 6.] \end{bmatrix}$$

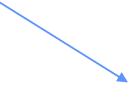
**Array broadcasting** - operating on arrays with different sizes:

```
arr = np.array([[1, 2, 3],
               [4, 5, 6]])
arr + np.ones((1, 3))
```

# Transpose

- `np.transpose()` - transpose an array, also `array.T`

```
arr35 = np.array([[1, 2, 3],  
                  [4, 5, 6],  
                  [7, 8, 9],  
                  [10, 11, 12]])  
  
print(arr35.shape)                                     # (4, 3)  
arr36 = arr35.transpose()  
print(arr36.shape)                                    # (3, 4)  
print(arr36)
```



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

ch03\_numpy/08\_transpose.py

`array.T` is the same as `np.transpose()`.



# Your Turn! - Task 3-1

Part 1

- Create a "checkerboard" array ( $8 \times 8$ ) filled with 1's and 0's

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

- Make it work for an  $n \times n$  board

Part 2

Hint: you may find `np.logical_not(arr)` useful in flipping 1's to 0's and vice versa

	0	1	2
0	0	1	0
1	1	0	1
2	0	1	0

This exercise can be accomplished either using PyCharm or Jupyter Notebook.

For the advanced solution, you may find `np.logical_not(arr)` helpful which inverts the values of a 0 /1 or True/False array.

# NumPy Functions (1 of 2)

- NumPy ships with dozens of helpful functions:

abs()	argmin()	bitwise_xor()	cumprod()	fill()	identity()
absolute()	argsort()	bmat()	cumsum()	finfo()	ifft()
accumulate()	array()	broadcast()	delete()	fix()	imag
add()	arrayrange()	bytes()	det()	flat	index_exp[]
all()	array_split()	c_[]	diag()	flatten()	indices()
allclose()	asarray()	cast[]()	diagflat()	fliplr()	inf
alltrue()	asanyarray()	ceil()	diagonal()	flipud()	inner()
angle()	asmatrix()	choose()	diff()	floor()	insert()
any()	astype()	clip()	digitize()	fromarrays()	inv()
append()	atleast_1d()	column_stack()	dot()	frombuffer()	iscomplex()
arange()	atleast_2d()	compress()	dsplit()	fromfile()	iscomplexobj()
arccos()	atleast_3d()	concatenate()	dstack()	fromfunction()	item()
arccosh()	average()	conj()	dtype()	fromiter()	ix_()
arcsin()	beta()	conjugate()	empty()	generic	lexsort()
arcsinh()	binary_repr()	copy()	empty_like()	gumbel()	linspace()
arctan()	bincount()	corrcoef()	eye()	histogram()	loadtxt()
arctan2()	binomial()	cos()	fft()	hsplit()	logical_and()
arctanh()	bitwise_and()	cov()	fftfreq()	hstack()	logical_not()
argmax()	bitwise_or()	cross()	fftshift()	hypot()	logical_or()

NumPy ships with a **HUGE** number of helpful functions.

# NumPy Functions (2 of 2)

logical_xor()	ogrid()	ranf()	slice()	tril()
logspace()	ones()	ravel()	solve()	trim_zeros()
ltsq()	ones_like()	real()	sometrue()	triu()
mat()	outer()	recarray()	sort()	typeDict()
matrix()	permutation()	reduce()	split()	uniform()
max()	piecewise()	repeat()	squeeze()	unique()
maximum()	pinv()	reshape()	std()	unique1d()
mean()	poisson()	resize()	sum()	vander()
median()	poly1d()	rollaxis()	svd()	var()
mgrid[]	polyfit()	round()	swapaxes()	vdot()
min()	prod()	rot90()	T	vectorize()
minimum()	ptp()	s_[]	take()	view()
multiply()	put()	sample()	tensordot()	vonmises()
nan	putmask()	savetxt()	tile()	vsplit()
ndenumerate()	r_[]	searchsorted()	tofile()	vstack()
ndim	rand()	seed()	tolist()	weibull()
ndindex()	randint()	select()	trace()	where()
newaxis	randn()	shape	transpose()	zeros()
nonzero()	random_sample()	shuffle()	tri()	zeros_like()

# Aggregation Functions

- NumPy provides functions for performing tasks on all selected elements within the array

```

arr31 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])

np.sum(arr31)           # 78
arr31.sum()             # 78                                     Sums all elements

np.mean(arr31[:,0])     # 5.5
arr31[:,0].mean()       # 5.5                                     Average of column 0

np.var(arr31[1])        # 0.667
arr31[1].var()          # 0.667                                    Variance of row 1

np.std(arr31[1])        # 0.816
arr31[1].std()          # 0.816                                    Standard deviation of row 1

np.prod(arr31[:2, :2])   # 40
arr31.prod()             # 40                                     Product of upper right 4 nums

arr31.prod(axis=1)       # [6 120 504 1320]                                Product of each row

```

ch03\_numpy/07\_aggregating.py

Remember, variance is the average of the squared differences from the mean:

A *population* standard deviation, a measure of how spread out numbers are, is the square root of the variance:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

A *sample* standard deviation divides by N-1 instead of N!

Numerous additional functions also perform aggregation operations, including:

`np.min()`, `np.max()` - find the min and max values within an array

`np.argmin()`, `np.argmax()` - find the index of the min and max values within an array

## Your Turn! - Task 3-2

- Use Jupyter Notebook for this task
- Create (manually) a NumPy array from the following data:

Day	Temp (High)	Temp (Low)	Humidity	Winds	Outlook	Red Flag
1	88	68	25	10	Sunny	False
2	84	65	31	5	Cloudy	False
3	86	66	32	5	Light Rain	False
4	89	67	26	5	Rain	False
5	92	70	22	10	Sunny	False
6	95	71	18	20	Sunny	True
7	94	69	27	10	Sunny	False

- Determine the **shape**, **size**, **strides**, and **number of dimensions** of the array
- Determine the average daily temperature and the average temperature for the week

Note: for average, use the `mean()` function (ex: `arr.mean()` ).

Hint: because the array is not homogeneous, use a `dtype=object`.



# Reshaping Arrays

- `ndarray.reshape(shape)` - can reshape an array's dimensions

```
arr32 = np.arange(1, 17).reshape((4,4))
```

New shape

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

**arr32**

- `np.ravel(array)` - treats a multidimensional array like a 1-D array

```
np.ravel(arr32)[8] # 9
```

ch03\_numpy/08\_reshaping.py

Using reshaping, we can see an easier way of creating a multidimensional array, however, this comes at the cost of extra time and resources consumed as two arrays will be created here.

`ravel()` has the effect of treating an array as a single-dimensional array, however, it doesn't change the original shape.



## Reshaping Arrays (continued)

- `np.squeeze(shape)` - removes any additional axes of length 1 from an array:

```
arr33 = np.array([[[[0], [1], [2]], [[3], [4], [5]]]])  
  
arr33.shape  
arr34 = arr33.squeeze()  
arr34.shape
```

# (1, 2, 3, 1)  
# [[0 1 2] [3 4 5]]  
# (2, 3)

ch03\_numpy/08\_reshaping.py

In the above example, any axis that only contains one element will be reduced (factored out). In arr33, there are two axes with only 1 element, therefore they are reduced to the 2 x 3 matrix. Note that arr33 is not modified, only the new arr34 will be created. arr33 is still the same shape.

# Boolean Indexing (Masking)

- ndarray values can be *Boolean indexed* yielding new "filtered" arrays

```
arr = np.arange(1, 7)
```

[1 2 3 4 5 6]

```
mask = arr <= 3
```

[ True True True False False False]

```
new_arr = arr[mask]
```

[1 2 3]

```
arr2 = np.arange(11, 17)
```

[11 12 13 14 15 16]

```
new_arr = arr2[mask]
```

[11 12 13]

Note that `~arr` performs a Boolean negation of `arr`:  
`~np.array([True, False])` → [False, True]

ch03\_numpy/16\_comparing\_values.py

In the first example above, the test, `arr <=3`, returns a new array with Boolean values that were determined by testing each individual value in the array. This Boolean array is then "plugged back into" the original array, yielding a new array that contains values only where "True" values existed in the Boolean array.

In the second example, a different array is used, but the same "mask" from the first example is passed into the array. This causes the same kind of filtering.

Additional logical functions include: `np.logical_not(arr)`, `np.logical_xor(test)`, and `np.logical_and(test)`

# Sorting Based on Values

- To sort an array based on a specific column:

```
arr = np.roll(np.arange(1, 10).reshape((3, 3)), 2)
```

Specifies the first column to sort on

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

```
sort_by_col0 = arr[arr[:, 0].argsort()]
```

Must take the results of the sort  
and plug it back into the original  
array to cause the sort to occur

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

```
sort_by_col2 = arr[arr[:, 2].argsort()]
```

Sorts on the third column

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

ch03\_numpy/18\_sorting.py

argsort() returns the indices that would sort an array.

In the first example, an array is created (as shown in the top, right box). The array is sorted using argsort() which returns the indices upon which to sort on. Plugging this back into the array will cause it to sort and create a new array.

The second example sorts on the third column.

# Reading Data Using genfromtxt()

- `genfromtxt()` can load data from a source file

```
np.genfromtxt(fname, dtype='float', delimiter=None,
              converters=None, missing_values=None,
              filling_values=None, usecols=None,
              skip_header=0, skip_footer=0)
```

**filling\_values** - default values to use when data is missing

**converters** - set of functions that can convert data to desired values

- Basic example:

```
result = np.genfromtxt('tennis.dat', delimiter=',',
                      usecols=(1,2), dtype=float)
```

```
print(result)      [[9.3 45.] [7.4 25.] [5.1 22.]]
```

```
print(result.dtype) float64
```

tennis.dat

Roger Federer,9.3,45
Rafael Nadal,7.4,25
Maria Sharipova,5.1,22

Using `dtype=None`  
causes NumPy to  
guess column types  
(here it would read in  
as structured data)

ch03\_numpy/14\_genfromtxt.py

`skip_header` = the number of lines to skip at the beginning of the file

`skip_footer` = the number of lines to skip at the end of the file

Numerous additional arguments exist as well, including: `names`, `defaultfmt`, `autostrip`, `replace_space`, `case_sensitive`, `loose`, `max_rows`.

In the example, `tennis.dat` file indicates player, on-court earnings and off-court earnings. For this example, we only read the second and third columns. Using a `dtype=None` value NumPy detects the column types.



# Reading As Structured or Object Data

- Since the data contains different data types, the following example can be read two ways:

city\_data.dat

```
Colorado Springs,6172,431000.0
Phoenix,1132,1488000.0
Raleigh,437,423000.0
Milwaukee,723,599000.0
Seattle,429,634000.0
```

Indicates a 50 byte string, 4-byte int, and 8-byte float will be read as one structure

- As *structured data*

```
data = np.genfromtxt('city_data.dat', delimiter=',', dtype='|U50', '<i4', '<f8'))
```

data.shape = (5, )

```
[ ('Colorado Springs', 6172, 431000.)
 ('Phoenix', 1132, 1488000.)
 ('Raleigh', 437, 423000.)
 ('Milwaukee', 723, 599000.)
 ('Seattle', 429, 634000.) ]
```

- As *objects*

```
data = np.genfromtxt('city_data.dat', delimiter=',', dtype=object)
```

data.shape = (5, 3), data.dtype=object

ch03\_numpy/14\_genfromtxt.py

When the file is read, because we have mixed data types, a record will be treated as multiple items (name, elevation, and population in this case). So, the shape of the data is (5,) because there is only one column of 5 structures that each contain 5 sub-items. But how to get to the sub-items within each structure? Give them names (see upcoming slide). If names are not given, automatically generated names are used (i.e., f0, f1, i1, i2, ....).

# NumPy *dtypes*

- Structured data types can be indicated using Python, NumPy, or shorthand values:
  - Legal dtype values in structures include:

Note:

< (little-endian) LSB first  
 > (big-endian) MSB first  
 | (vertical bar) byte order doesn't apply

b1, i1, i2, i4, i8, u1, u2, u4, u8, f2, f4,  
 f8, c8, c16  
 S50 (50-byte string), U50 (unicode)  
 np.int8, np.int16, np.int32, np.int64,  
 np.int128, np.float16, np.float32,  
 np.float64, np.float128, np.float256,  
 np.uint8, np.uint16, np.uint32,  
 np.uint64, np.uint128  
 object, int, float, others

```
data = np.genfromtxt('city_data.dat', delimiter=',',
                     dtype=('U50', np.int32, float))

print(data.dtype)    [('f0', '<U50'), ('f1', '<i4'), ('f2', '<f8')]
```

ch03\_numpy/14\_genfromtxt.py

Notice how 'f0', 'f1', and 'f2' are created automatically. What are these? These are the names that can be used to access the columns (sub-items) within a structure.

Not shown are additional type values such as np.complex32, np.complex64, ..., np.complex512. The 'b1' represents a single signed byte.

For more on constructing dtypes, refer to the documentation:  
<https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.dtypes.html>

# Using *names* with Structured Data

- The **names** param or the **dtype labels** provide access to the structured data values

city\_data.dat

```
Colorado Springs,6172,431000.0
Phoenix,1132,1488000.0
Raleigh,437,423000.0
Milwaukee,723,599000.0
Seattle,429,634000.0
```

```
data = np.genfromtxt('city_data.dat', delimiter=',',
                     names=['name', 'elev', 'pop'],
                     dtype=['|U50', '<i4', '<f8'])
```

```
print(data['name'])
```

```
['Colorado Springs' 'Phoenix' 'Raleigh'
'Milwaukee' 'Seattle']
```

```
data = np.genfromtxt('city_data.dat', delimiter=',',
                     dtype=[('name', '|U50'), ('elev', '<i4'),
                            ('pop', '<f8')])
```

```
print(data['elev'][0])
```

```
6172
```

ch03\_numpy/14\_genfromtxt.py

Here, we present two examples. The first makes use of the *names* attribute to assign column names to the structured data that was read. The lower example describes the *names* field within the *dtype* parameter. Both versions are equivalent.

# Handling Missing Values

- The data file has missing items
- It also consists of mixed data types

city\_data2.dat

```
Colorado Springs,6172,431000.0
Phoenix,1132,1488000.0
Raleigh,437,423000.0
Milwaukee,,599000.0
Seattle,429,
```

```
data = np.genfromtxt('city_data2.dat', delimiter=',',
                     filling_values=['', -999, -1],
                     dtype=('|U50', '<i4', '<f8'))
```

```
[('Colorado Springs', 6172, 431000.0) ('Phoenix', 1132, 1488000.0)
 ('Raleigh', 437, 423000.0) ('Milwaukee', -999, 599000.0)
 ('Seattle', 429, -1.0)]
```

**filling\_values** defines what to do with values not present when reading from the file.

ch03\_numpy/14\_genfromtxt.py

In this example, the specified data file is read, however it has two missing values. The `filling_values` list defines how to handle those values. When the data is read, the missing data will be replaced with the default values, as shown in the resulting output.

# NaN and Infinity

- NumPy defines NaN (`np.nan`) and infinity (`np.inf`)
  - Use `np.isnan()` or `np.isinf()` to check for these values
- To remove rows where a NaN or infinity exist:

data

```
[[ 6172.  431000.]
 [ 1132.      inf]
 [ 437.  423000.]
 [ nan   599000.]
 [ 429.      nan]]
```

We manually generated this value with:  
`data[1, 1] = data[1, 1]/0`

```
no_nans = data[~np.isnan(data).any(axis=1)]
```

*Removes NaN rows*

```
[[ 6172. 431000.]
 [ 1132. inf]
 [ 437. 423000.]]
```

```
no_infs = data[~np.isinf(data).any(axis=1)]
```

*Removes inf rows*

```
[[ 6172. 431000.]
 [ 437. 423000.]
 [ nan 599000.]
 [ 429. nan]]
```

```
neither = data[np.isfinite(data).all(axis=1)]
```

*Removes NaN and inf rows*

```
[[ 6172. 431000.]
 [ 437. 423000.]]
```

ch03\_numpy/19\_nans.py

NaN values may only exist in float-based columns. They represent missing values or results of operations that are not valid.

NumPy also has a `np.isfinite()` method that checks for finite real numbers.

Let's examine the example that removes the NaN rows above. A check is made against all elements in the data array which generates a new array of Booleans. A `False` will appear where NaN values exist (due to the `~`). `any()` is used as a filter to remove any rows that have NaN values in them.

To better understand the use of `any()`, here's another example:

```
data = np.array([[True, False, False],
                [False, False, True],
                [False, False, False]])
print(np.any(data))                  # True
print(np.any(data, axis=0))          # [True False True]
print(np.any(data, axis=1))          # [True True False]
```



# Summary

- NumPy provides a powerful array type called an ndarray
- It supports numerous operations to manipulate, transform, and reshape data
- The NumPy array is used as a foundational data structure with other frameworks
  - It integrates nicely with other tools such as Matplotlib and Pandas



## Your Turn! - Exercise 3-3

- Obtain and display the top 100 batting averages from the provided data file
  - Read from <student\_files>/resources/baseball/Batting.csv
  - Read columns 6 (atbats) and 8 (hits)
  - Calculate the batting average:  $\frac{\text{hits}}{\text{atbats}}$
- Remove rows where the *atbats* is less than 502  
(this is an official MLB minimum needed to count for the batting average title)
- Remove rows where the year is prior to 1957  
(The modern era has a 162-game schedule which began in this year)  
(The second column in the data file defines the year)



# Chapter 4

## Introducing

## Matplotlib

Making Use of Other Tools  
Available for Python



# Overview

Data Visualization

Introducing Matplotlib

Plotting Graphs

Figures and Axes



# Data Visualization and Python

- Numerous frameworks exist for creating 2D and 3D charts:
  - Matplotlib de facto standard for 2D Python visualization
  - Pandas provides data structures and wrappers for Matplotlib API
  - Seaborn wrapper for Matplotlib providing simpler code to generate charts
  - Ggplot tool that works with Pandas DataFrames and provides layered charts
  - Bokeh Python charting tool allowing for various level of control

The tools mentioned above are common and readily recognized. Numerous additional tools exist, including: pygal, geoplotlib, and more.



# Introducing Matplotlib

- Matplotlib is the Python **de facto** standard API for creating 2D graphs

- Integrates with **NumPy arrays**

- Supports numerous chart types (kinds):

- Bar

- Line

- Scatter

- Pie

- Log

- Polar

- Streamplots

- Ellipses

- Table Demo

- ...and more...

- To render charts *inline* within Jupyter when using notebooks, specify the following command:

```
%matplotlib inline
```

Because Notebooks runs within a browser, graphs will not show up automatically unless special steps are taken. If the `%matplotlib inline` statement is provided within a notebook code cell, then graphical output will show up in the notebook web page. If the top statement is left out when running Notebooks from a browser, you may not see any graphical output. So, be sure to include this statement (generally before any of your import statements).



# Matplotlib Modules

- Common conventions for importing primary modules:

```
import matplotlib as mpl
```

We will refer to  
this module as **plt**  
from now on

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

The **pyplot** submodule contains the  
primary functionality for plotting

For reference, the PyPlot API docs can be found here:

[http://matplotlib.org/api/pyplot\\_api.html](http://matplotlib.org/api/pyplot_api.html)

While the syntax of the imports can vary, the examples shown on this slide represent commonly used Matplotlib module names.

# A First Simple Plot

```

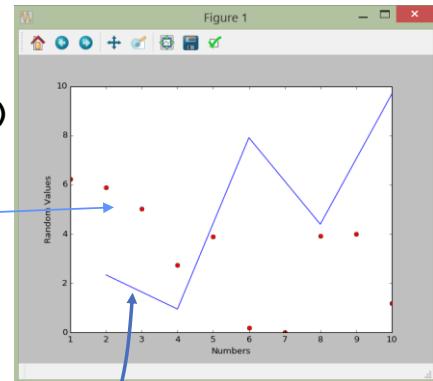
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

x1 = np.arange(1, 11)
y1 = 10 * np.random.rand(1, 10).squeeze()
plt.ylabel('Random Values')
plt.xlabel('Numbers')
plt.plot(x1, y1, 'ro')

x2 = np.arange(2, 11, 2)
y2 = 10 * np.random.rand(1, 5).squeeze()
plt.plot(x2, y2)

plt.show()

```



ch04\_matplotlib/01\_simple\_plot.py

In the example, two pairs of NumPy arrays are provided to the `plt.plot()` method. The first, `x1`, is a range of values from 1 to 10, while the second, `y1`, is an ndarray of 10 random values. They are rendered using red circles.

The second plot is overlaid onto the first. This one plots the `x2` and `y2` ndarrays using the default blue line. More on the line and marker styles is shown on the next slide.

Labels have been added and then `show()` is invoked to cause the back end to render the graph.

# The Plot Method

- The plot() method supports numerous arguments:

`plt.plot(x, y, style)`

An array of data on the x-axis

An array of data on the y-axis

The style is a string that controls the color and markers

## Chart Styles

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

'-'	solid line style
'--'	dashed line style
'-. '	dash-dot line style
' : '	dotted line style
' . '	point marker
' , '	pixel marker
' o '	circle marker
' v '	triangle_down marker
' ^ '	triangle_up marker
' < '	triangle_left marker
' > '	triangle_right marker
' 1 '	tri_down marker
' 2 '	tri_up marker

'3'	tri_left marker
'4'	tri_right marker
's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
' '	vline marker
'_'	hline marker

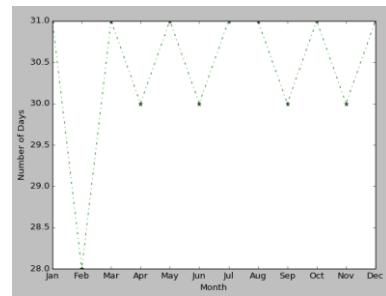
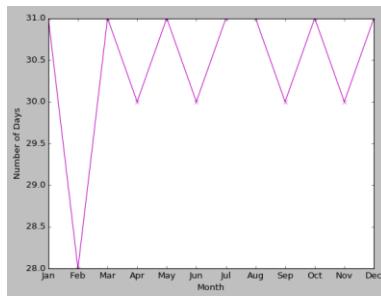
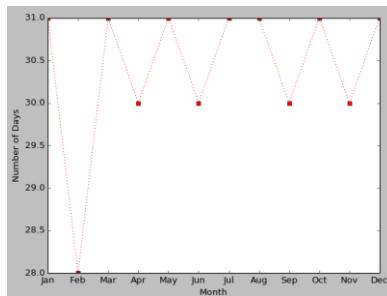
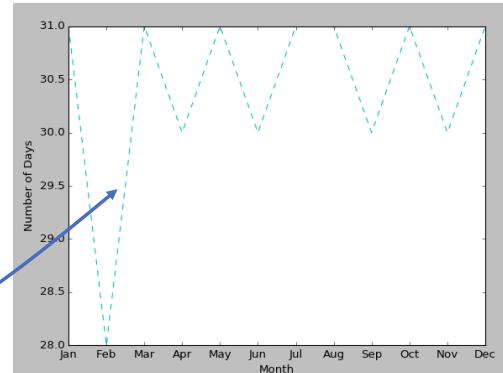
When creating a graph using Matplotlib, you will typically supply a list, ndarray for the x-axis and the y-axis. Other structures, yet to be discussed, can also be supplied. In addition to the x and y axes, the style of marker/line can be provided. This argument is a string containing a color symbol (a letter from the color list on the left above) followed by a line style (an option from the right two lists). The default is 'b-', which refers to a solid blue line. Other examples are shown on the next slide.

Also, instead of supplying two individual arrays, a single, two-dimensional array may be supplied for the x and y axis. Examples of this will be shown later.

# Plot Style Variations

```
x_ticks = ['Jan', 'Feb', ..., 'Dec']
x = np.arange(1, 13)
y = [31, 28, 31, 30, ..., 31]

plt.xlabel('Month')
plt.ylabel('Number of Days')
plt.xticks(x, x_ticks)
plt.plot(x, y, 'c--')
plt.show()
```



```
plt.plot(x, y, 'rs:') plt.plot(x, y, 'mx-') plt.plot(x, y, 'g*-.' )
```

ch04\_matplotlib/02\_style\_variations.py

In this example, the last day of the month is plotted against the month name. Because Matplotlib wants numerical values for the axes, we can apply a range for the x-axis, but provide names for the tick marks on the x-axis. This causes the month names to be filled in. We can do this using plt.xticks(), which accepts as arguments the location of ticks followed by a list of labels to display for those ticks.

Notice in the top example the use of 'c--' for the style argument. This displays a cyan colored dashed line.

The bottom examples (from left to right) display as: red square markers with a red dotted line, magenta solid line with x markers, and green dash-dot line with star markers.

# Bar Graphs

- Use the **bar()** method to create a bar graph:

```
plt.bar(left, height, width, bottom, ...)
```

```
cities = ['Colorado Springs', 'Phoenix',
          'Raleigh', 'Milwaukee', 'Seattle']
cities_idx = np.arange(5)
elevations = [6172, 1132, 437, 723, 429]

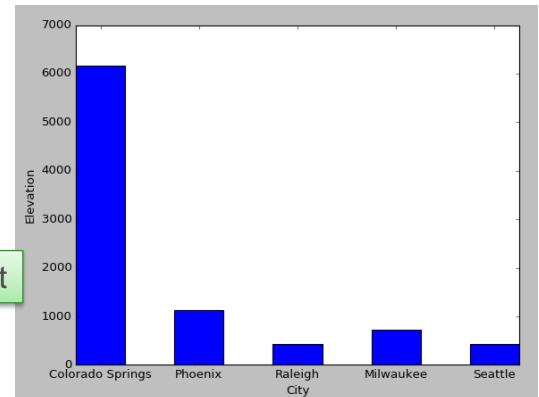
plt.xlabel('City')
plt.ylabel('Elevation')

bar_width = 0.5
plt.xticks(cities_idx + bar_width/2,
           cities)
plt.bar(cities_idx, elevations,
        bar_width)

plt.show()
```

Diagram illustrating the bounding boxes for a bar plot:

- left, bottom**: The bottom-left corner of the bar.
- left+width**: The right edge of the bar.
- bottom+height**: The top edge of the bar.



ch04\_matplotlib/03\_bar\_plots.py

Rectangles in a bar plot are bounded by: left, left+width, bottom, bottom+height

Some of the available arguments to `bar()` are:

`left` - (sequence of scalars) this represents the x-coordinates of the left sides of the bars

`height` - (sequence of scalars) represents the height of each bar

`width` - the width of the bar (def. is 0.8)

`bottom` - the y-coordinates of the bars (def. is None, normally not needed)

`color` - color of the bar area

`linewidth` - width of the bar edges

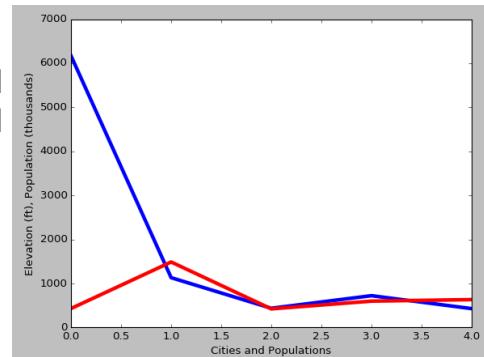
`edgecolor` - color of the edges of the bars

# Plotting Multiple Data Sets

- The `plot()` method supports plotting multiple data sets by specifying additional arguments:

```
plt.plot(x1, y1, style1, x2, y2, style2, ...)
```

```
cities = ['Colorado Springs', ...]
cities_idx = np.arange(5)
elevations = [6172, 1132, 437, 723, 429]
populations = [431, 1488, 423, 599, 634]
plt.xlabel('Cities and Populations')
plt.ylabel('Elevation (ft), ...')
plt.plot(cities_idx, elevations, 'b-',
         cities_idx, populations, 'r-',
         linewidth=4.0)
plt.show()
```



- For more control, however, Matplotlib supports rendering multiple plots using **Figures** and **Axes**...

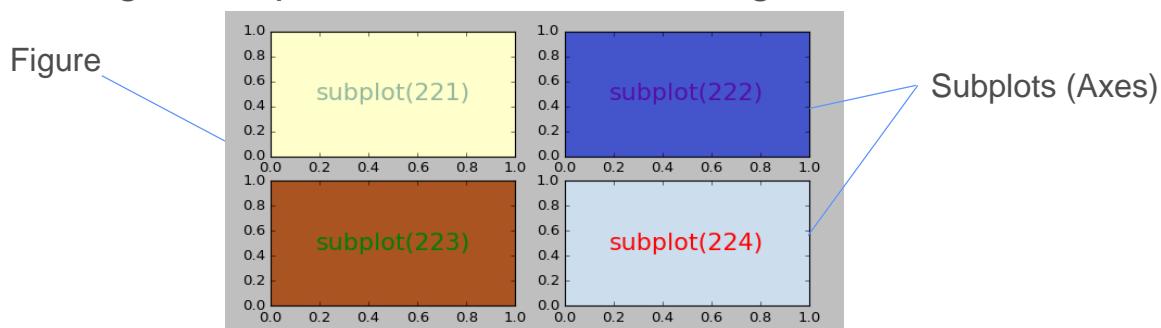
ch04\_matplotlib/04\_multiplots.py

To create multiple plots on a single axis, you can use multiple sets of x-data, y-data, and style values.



# Figures and Subplots

- A Matplotlib graph is made up of a Figure instance and one or more Axes instances
  - Figures represent the *entire drawing canvas*



- **Axes** represent a *coordinate system for plotting data*
- **Subplots** are special cases of (inherit from) Axes
- Subplots can be placed onto a figure using a layout system: **nrows ncols plot\_number**
  - Use subplots when laying out multiple plots on a grid

ch04\_matplotlib/05\_figure\_axes.py

The Figure object manages all graphs and content. Plots may be made on a figure, generally created by calling `add_subplot()`. The code used to create the diagram above is shown on the next slide.

# Adding Subplots to a Figure

- `plt.figure()` is a factory for generating figure objects:

```
plt.figure(num, figsize, dpi, facecolor, edgecolor)
```

```
figure = plt.figure(figsize=(8, 4))

sub221 = figure.add_subplot(221, facecolor='#ffffcc')
sub221.text(0.5, 0.5, 'subplot(221)', ha='center',
            va='center', fontsize=20, alpha=.5)

sub222 = figure.add_subplot(222, facecolor='#4455cc')
sub222.text(0.5, 0.5, 'subplot(222)', ...)

sub223 = figure.add_subplot(223, facecolor='#33aa22')
sub223.text(0.5, 0.5, 'subplot(223)', ...)

sub224 = figure.add_subplot(224, facecolor='#ccddcc')
sub224.text(0.5, 0.5, 'subplot(224)', ...)

plt.show()
```

Canvas size in inches.  
See footnotes for support for pixel sizing.

ch04\_matplotlib/05\_figure\_axes.py

The num associated with a figure is generally created automatically and managed internally. figsize defines the size of the figure in inches as an iterable (usually a tuple). Matplotlib figures don't support pixel sizes, but you can use the following to help try to calculate sizes in pixels:

```
plt.figure(figsize=(pix_wdth/dpi, pix_ht/dpi), dpi=dpi)
```

The syntax and arguments to the `text()` method shown above are:

`plot.text(x, y, s, fontdict, withdash)`

x, y - coordinates of the textbox

s - string of text

fontdict - dictionary to override the default font properties

withdash - creates text with dashes

ha and va are shorthands for horizontalalignment and verticalalignment.

# Plotting Text

- `plt.text()` allows any string to be added to a plot

```
plt.text(x, y, s, ...)
```

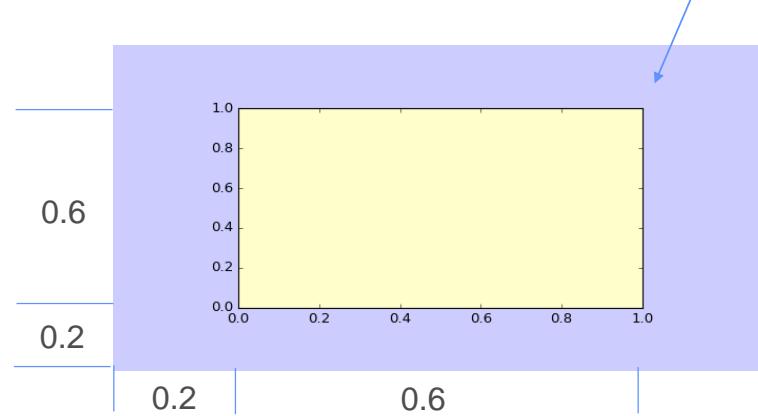
<code>x, y</code> -	coordinates of the textbox
<code>s</code> -	the string of text to display
<code>horizontalalignment, ha</code> -	'center', 'left', 'right'
<code>verticalalignment, va</code> -	'center', 'top', 'bottom', 'baseline'
<code>color</code> -	#hex_value or single letter (e.g. 'b')
<code>backgroundcolor</code> -	same values allowed as color above
<code>alpha</code> -	an alpha-transparency, 0 is transparent, 1 is opaque
<code>fontname, family</code> -	valid font name
<code>fontsize, size</code> -	a pixel size or 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'
<code>rotation</code> -	degrees or 'horizontal' or 'vertical'
<code>style</code> -	'normal', 'italic'

From the previous example, the `text()` method supports adding text to graphs. Many of the arguments used to display the text are shown here.

# Managing Axes

- Use `add_axes()` to create a new axis within a Figure

```
figure = plt.figure(figsize=(8, 4), facecolor='#ccccff')
```



```
figure.add_axes((0.2, 0.2, 0.6, 0.6), facecolor='#ffffcc')
plt.show()
```

(left, bottom, width, height)

All values are fractions of the figure's total size

ch04\_matplotlib/06\_axes.py

What is the difference between `add_subplot()` and `add_axes()`?

Subplot instances inherit from Axes. Subplots manage plots in a grid format, so they should be used when the desire is to place graphs in a uniform order.

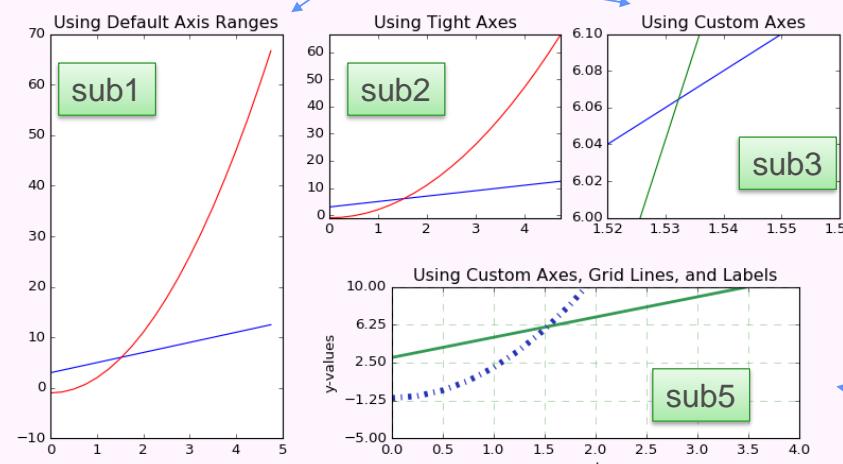
However, if finite control is desired, use `add_axes()` to create the axis. This approach allows for placing a plot at exactly the desired location within the figure.

In the diagram above, the lower, left corner of the plot will appear 20% from the bottom and 20% from the left. It will span 60% the width of the figure and 60% the height of the figure. Finally, the `facecolor` argument can supply a color value to be displayed.

# Custom Layouts (1 of 3)

- Layouts can be created using `add_subplot()` and then specifying 3 scalar values
  - This provides the ability to span multiple columns or rows

These are laid out using  
`add_subplot()` rows and columns



This plot is laid out using axes() and specifying l, b, w, h

ch04\_matplotlib/07\_layouts.py

In this example, a  $2 \times 3$  layout system grid is used to place a series of plots. The top three plots are all placed nicely into the grid, therefore, they are created using the figure's `add_subplot()` method (shown on the next slide). For reference, the variables representing these plots are shown in the green boxes above (sub1, sub2, sub3, sub5).

The last diagram shows a plot placed at a specific location using the figure's `add_axes()` method.

## Custom Layouts (2 of 3)

```

x = np.arange(0, 5, 0.25)
y1 = 2*x + 3
y2 = 3*x**2-1

figure = plt.figure(figsize=(12, 6), facecolor='#ffff55ff')

sub1 = figure.add_subplot(2, 3, (1, 4))
sub1.plot(x, y1, 'b-', x, y2, 'r-')
sub1.set_title('Using Default Axis Ranges')

sub2 = figure.add_subplot(2, 3, 2)
sub2.plot(x, y1, 'b-', x, y2, 'r-')
sub2.axis('tight')
sub2.set_title('Using Tight Axes')

sub3 = figure.add_subplot(2, 3, 3)
sub3.plot(x, y1, 'b-', x, y2, 'g-')
sub3.set_xticks(np.linspace(1.52, 1.56, 5))
sub3.set_xlim((1.52, 1.56))
sub3.set_ylim((6.0, 6.1))
sub3.set_title('Using Custom Axes')

```

ch04\_matplotlib/07\_layouts.py

The above code represents the top 3 plots. Notice that we can think of our  $2 \times 3$  grid as follows:

231    232    233

234    235    236

Therefore, to cause the first plot to cover 231 and 234, we can write:  
`figure.add_subplot(2, 3, (1, 4))`.

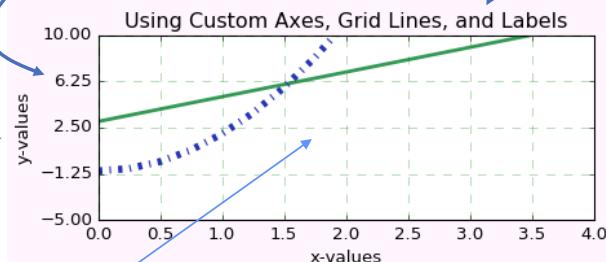
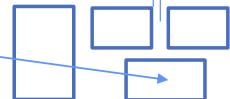
The other two plots fit nicely into a single grid cell (232 and 233). The final plot doesn't fit nicely, we'll place it where we want using `add_axes()`.

# Custom Layouts (3 of 3)

```
sub5 = figure.add_axes((0.46, 0.1, 0.4, 0.3))
sub5.plot(x, y1, 'b-', color='#339955', lw=2.5)
sub5.plot(x, y2, 'r-', color='#2533b7', lw=5.0, ls='dashdot')
sub5.set_yticks(np.linspace(-5, 10, 5))
sub5.set_xlim((0, 4))
sub5.set_ylim((-5, 10))
sub5.set_xlabel('x-values')
sub5.set_ylabel('y-values')
```

`set_xlim()`, `set_ylim()` set the range for the x and y axes, while `set_xticks()` and `set_yticks()` set the frequency and label of the ticks

```
sub5.grid(color='g', alpha=0.5, ls='dashed', lw=0.5)
sub5.set_title('Using Custom Axes, Grid Lines, and Labels')
```



ch04\_matplotlib/07\_layouts.py

The `add_axes()` call places the last plot exactly where we want it. 0.46 and 0.1 refer to the left, bottom corner of the plot with respect to the left, bottom corner of the figure (as a fraction of the total size). The 0.4 and 0.3 values represent the width and height of the plot. That's it! The remaining items are similar to the other plots.

The `grid()` call allows for establishing line properties such as color and alpha transparency. Shorthands for linestyle (`ls`) and linewidth (`lw`) are also used here.

Notice that the line properties can be changed. In the `sub5.plot()` statement, we performed two calls to `plot()`, once for each formula plotted. In each case, the line properties were changed, resulting in different visual lines. Valid line properties are shown on the next slide.

# Helpful Line Properties

- The previous slide illustrated a few line properties, here are some valid values that may be used:

```
plt.plot(xdata, ydata, ...)
```

color, c -	any '#hex_value', or single letter ('r'),
linestyle, ls -	'solid', 'dashed', 'dotted', '-', '--', '-.', '.', or other valid symbol
linewidth, lw -	float value for the width
alpha -	0.0 (transparent) to 1.0 (opaque)
label -	text placed on the line
fillstyle -	'full', 'left', 'right', 'bottom', 'top', 'none'
marker -	any marker style described on an earlier slide (e.g. 'v' or 'o')
drawstyle -	'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'
markersize, ms -	float value for the size of the markers
markerfacecolor -	a color for the markers
markeredgecolor -	a color for the edges of the markers
markeredgewidth -	width of the line that draws the markers

Visit here for more marker styles:

[http://matplotlib.org/api/markers\\_api.html#module-matplotlib.markers](http://matplotlib.org/api/markers_api.html#module-matplotlib.markers)

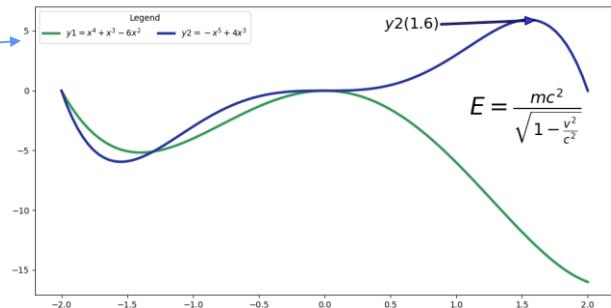
The drawstyle is the line type used to connect the data points. Steps look like stairs.

While the slide shows a call to plt.plot(), these values apply to any subplot/axes object as well.

# Legends

- Use `plt.legend()` to customize a legend

A title, two columns, and optimized positioning creates this legend.



```
x = np.linspace(-2, 2, 100)
y1 = x**4 - x**3 - 6*x**2
y2 = -x**5 + 4*x**3

figure = plt.figure(figsize=(12, 6), facecolor='#ffff55ff')

sub1 = figure.add_axes((0.1, 0.1, 0.8, 0.8))
sub1.plot(x, y1, color='#339955', lw=3.0, ls='solid')
sub1.plot(x, y2, color='#253397', lw=3.0, ls='solid')
sub1.legend([r'$y1 = x^4 + x^3 - 6x^2$',  

            r'$y2 = -x^5 + 4x^3$'],  

            loc='best', title='Legend', ncol=2)
```

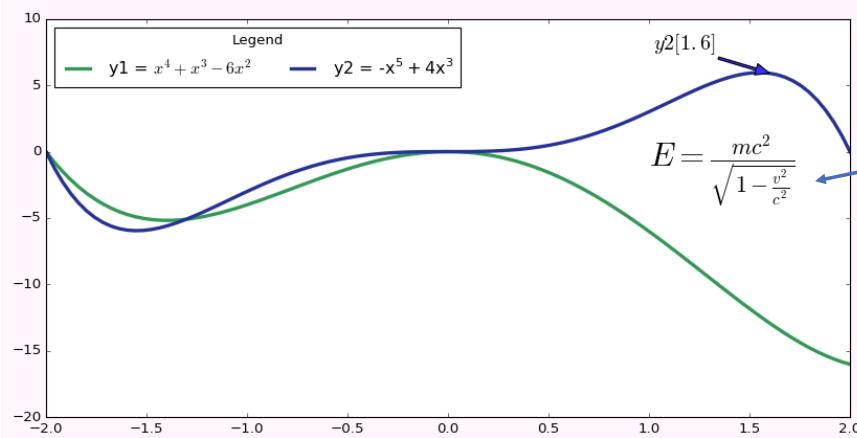
LaTeX, a special markup for displaying numerical values.

ch04\_matplotlib/08\_legends.py

The `loc` argument allows for specifying the location of the legend. Other values include: 'upper right', 'upper left', 'lower right', 'lower left', 'right', 'center left', 'center right', 'lower center', 'upper center', 'center'. The `ncol` argument defines how many columns to display for the legend. The default value is 1.

# Annotations

```
sub1.annotate(s=r'$E = \frac{mc^2}{\sqrt{1-\frac{v^2}{c^2}}}$',
              xy=(0.7, 0.6), xycoords='figure fraction',
              fontsize=28)
```



Annotations can be used to give plots additional flare or highlight noteworthy points

```
sub1.annotate(r'$y2(1.6)$',
              xy=(1.6, -1.6**5 + 4*1.6**3), xycoords='data',
              xytext=(-100, +20), textcoords='offset points',
              fontsize=18, arrowprops=dict(facecolor='#3333ff',
                                           headwidth=10, frac=0.3, width=2))
```

ch04\_matplotlib/08\_legends.py

Two annotations have been added to our example, one is purely for illustration to show the structure of a TeX statement. The first annotation, ( $E=mc^2$ ), places the formula onto the plot specifying the `xycoords` value of 'figure fraction', which means it will be placed as a percentage of the figure. The `xy` argument places the text at 0.7 across the width and 0.6 up the height of the figure. Other values for `xycoords` can be 'figure points', 'figure pixels', 'figure fraction', 'axes points', 'axes pixels', 'axes fraction', 'data', 'offset points', and 'polar'.

The lower example uses the coordinate 1.6,  $y_2(1.6)$  and then creates an annotation that is offset from that location. `arrowprops` identifies properties for the arrow that points to the data value.



## savefig

- Save figures using `figure.savefig()`:

```
figure.savefig(fname, dpi=None, facecolor='w',
               edgecolor='w', format=None
               orientation='portrait')
```

```
figure.savefig('legendary.png')
```

ch04\_matplotlib/08\_legends.py

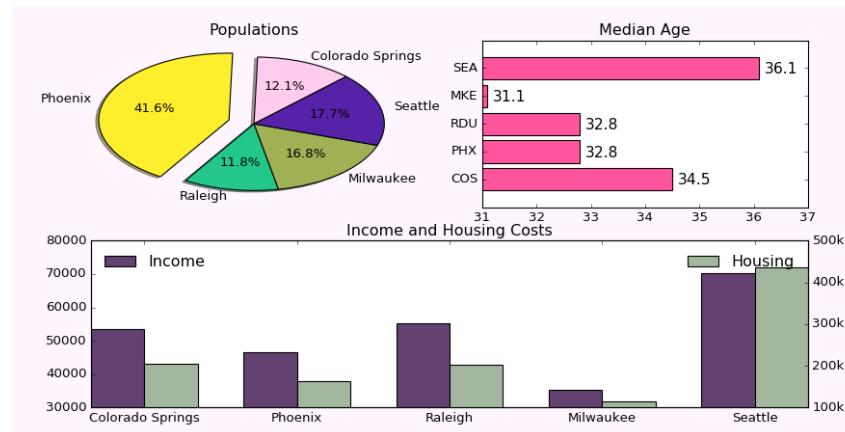
`fname` is the name of the file the plot will be saved to.

The format can be one of the formats supported by the backend. Most back ends will support 'pdf', 'ps', 'svg', 'png', and 'eps'.

The orientation may be 'portrait' or 'landscape'.

The `papertype` argument (not shown above) can be 'legal', 'letter', 'executive', 'ledger', or another standard size, such as 'a0'.

# Other Chart Examples (1 of 3)



```
figure = plt.figure(figsize=(12, 6), facecolor='#ffff55ff')

cities = ['Colorado Springs', 'Phoenix', 'Raleigh',
          'Milwaukee', 'Seattle']
cities_abbr = ['COS', 'PHX', 'RDU', 'MKE', 'SEA']
elevations = [6172, 1132, 437, 723, 429]
populations = [431000, 1488000, 423000, 599000, 634000]
median_age = [34.5, 32.8, 32.8, 31.1, 36.1]
median_income = [53550, 46601, 55170, 35186, 70172]
median_house = [205600, 162300, 202800, 113900, 436600]
```

ch04\_matplotlib/09\_other\_charts.py

The figure shown above is created using the data provided here. The subsequent two slides discuss the pie, horizontal bar, and grouped bar plots.

# Other Chart Examples (2 of 3)

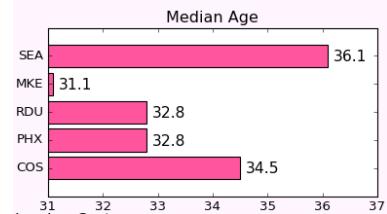
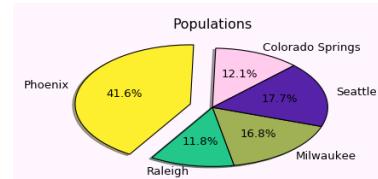
```

sub1 = figure.add_subplot(221)
colors = ['#ffccce', '#fdef2e', '#23c78a', '#a0b055', '#5623a8']
sub1.pie(populations, labels=cities,
          colors=colors,
          explode=[0, 0.2, 0, 0, 0],
          autopct='%.1f%%',
          shadow=True, startangle=45)

sub1.set_title('Populations')

sub2 = figure.add_subplot(222)
y_ticks = np.arange(len(cities))
sub2.barh(y_ticks, median_age, height=0.8,
           align='center', color='#ff559e')
sub2.set_yticks(y_ticks)
sub2.set_xlim((31, 37))
sub2.set_yticklabels(cities_abbr)
sub2.set_title('Median Age')
for loc, age in enumerate(median_age):
    sub2.annotate(str(age), xy=(median_age[loc], loc),
                  xycoords='data', xytext=(+5, -5),
                  textcoords='offset points', fontsize=14)

```

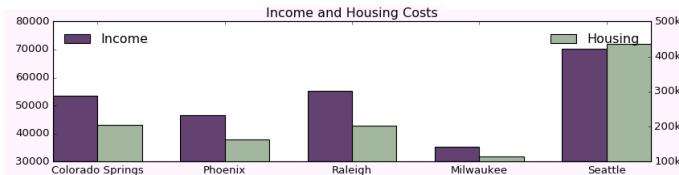


ch04\_matplotlib/09\_other\_charts.py

There are two plots shown here. On top, the pie chart is created by invoking `pie()`. The `colors` options provide the shading for each pie piece. The `explode` option identifies which pieces to "break off" and by how much (fractionally). `autopct` defines the numeric value placed within the pie piece.

On the bottom, the `barh()` method call creates a horizontal bar graph. Take note of the placement of annotations to identify the values of data points at the end of each bar. This was done by plotting the value of the `median_age` and then adding a small offset to move the text slightly further to the right.

# Other Chart Examples (3 of 3)



Grouped Bar,  
Twin Axis

```
sub3 = figure.add_subplot(2, 2, (3, 4))
bar_width = 0.35
x_ticks = np.arange(len(cities))
sub3.bar(x_ticks, median_income, width=bar_width, color="#634170")
sub3.set_xticks(x_ticks + bar_width)
sub3.set_xticklabels(cities)
sub3.set_xlim(30000, 80000)
sub3.set_title('Income and Housing Costs')
sub3.legend(['Income'], loc='upper left', frameon=False)

sub3b = sub3.twinx()                                     ← Creates axis on
sub3b.bar(x_ticks + bar_width, median_house,           other side of x
          width=bar_width, color='#a2b79e')
sub3b.set_xlim(100000, 500000)
sub3b.set_yticks(np.arange(100000, 600000, 100000))
sub3b.set_yticklabels(['100k', '200k', '300k', '400k', '500k'])
sub3b.legend(['Housing'], loc='upper right', frameon=False)
```

ch04\_matplotlib/09\_other\_charts.py

In this example, the main "new" concepts are the fact that there are two bars per data point (city) and that two axes are used on the left and right sides. Use the `twinx()` method to generate a new sub-axis that allows data to be plotted on the right side.

Note that there are two calls to `bar()` in this example to generate the dual bar graphs.

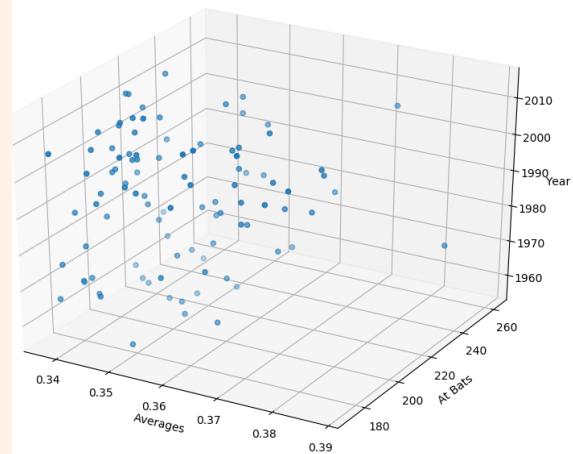
# 3D Plots

- **mpl\_toolkits** (included within Anaconda) provides additional tools for creating 3D plots
  - Plots include:
 

• 3D Scatter	• Wireframe
• 3D Line	• Polygon
• Surface	• Contour

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d

...
fig = plt.figure(figsize=(8, 6))
ax = axes3d.Axes3D(fig)
x = data[-1:-100:-1, 3]
y = data[-1:-100:-1, 2]
z = data[-1:-100:-1, 0]
ax.scatter(x, y, z)
ax.set_xlabel('Averages')
ax.set_ylabel('At Bats')
ax.set_zlabel('Year')
plt.show()
```



ch04\_matplotlib/10\_3D\_plots.py

Use `mpl_toolkits.mplot3d` to generate 3D plots. This is included in Anaconda. Other `mpl_toolkits` include classes for displaying multiple axes in Matplotlib (`axes_grid1`) and tools for displaying curvilinear grids (`axisartist`).

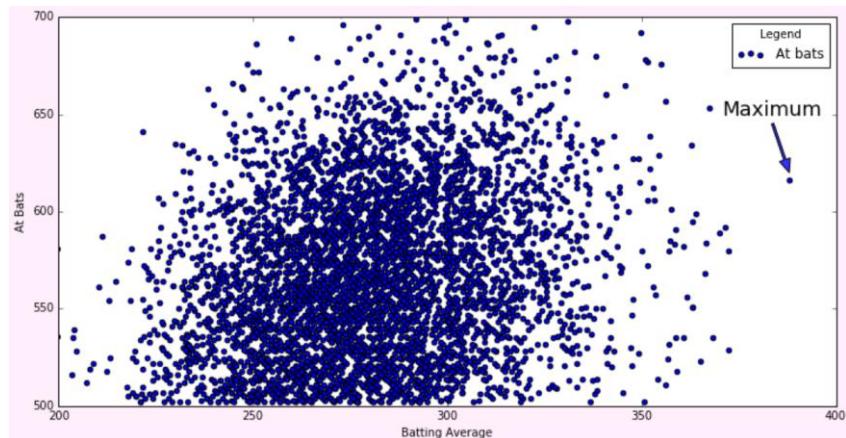


# Summary

- Matplotlib is one of the most commonly used visualization libraries
- It supports a huge number of customization features and plot types
- Matplotlib understands and integrates with NumPy

## Your Turn! - Task 4-1

- Create a **scatter plot** that plots batting average (x-axis) against atbats (y-axis)
  - Use `plt.scatter()`, options are similar to other plots
  - Provide appropriate labels for each axis
  - Create a legend and annotate the maximum batting average





# Chapter 5

## Pandas

Working with the Python Pandas Tool

# Overview

Introducing Pandas

Series

DataFrames

Reading from Files and Other  
Sources



# Introducing Pandas

- Pandas is a data analysis library for Python and part of the SciPy suite
- Features include:
  - APIs for reading data into data structures
  - Data merging, reshaping and pivoting of data
  - Split-apply-combine operations
- Data structures include:
  - Series
  - DataFrame
  - Panel

Common import syntax  
We will refer to this module as pd in further examples

`import pandas as pd`

Panel is deprecated as of version 0.20



# Pandas Series

- A Pandas **Series** is a *one-dimensional ndarray*
  - Series have an index (like a dictionary) called **labels**
  - Many ndarray methods are overridden to better support Series operations

```
s = pd.Series(data, index, dtype)
```

```
import pandas as pd  
  
ser1 = pd.Series([212, 32, 0, -273])  
print(ser1)
```

Output:  
0 212  
1 32  
2 0  
3 -273  
dtype: int64

ch05\_pandas/01\_creating\_series.py

A Pandas Series is based upon the NumPy array. NumPy does not have to be imported to create a Series, however.

# Creating Series

- Additional ways to create series:

```
ser2 = pd.Series(name='City Elevations',
                  index=['Colorado Springs', 'Phoenix',
                         'Raleigh', 'Milwaukee', 'Seattle'],
                  data=[6172, 1132, 437, 723, 429])
```

```
print(ser2)
```

Colorado Springs	6172
Phoenix	1132
Raleigh	437
Milwaukee	723
Seattle	429
Name: City Elevations, dtype: int64	

```
city_elevations = {'Denver': 5883, 'Austin': 632,
                   'Philadelphia': 21, 'Billings': 3649, 'Anchorage': 144}
ser3 = pd.Series(city_elevations, name='City Elevations')
```

```
print(ser3)
```

Anchorage	144
Austin	632
Billings	3649
Denver	5883
Philadelphia	21
Name: City Elevations, dtype: int64	

ch05\_pandas/01\_creating\_series.py

A Series is a typical Python class, therefore it follows conventional class usage rules. Keywords may be used when instantiating objects as shown in the top example. Similarly, a dictionary may be provided when creating a Series which uses the keys as the labels. If an index is provided along with the dictionary, the index will override the keys from the dictionary.



# Accessing and Working with Series

- Series allow access to values using methods similar to Python dictionaries:

```
ser4 = pd.Series([218, 15, 619, 13, 1295],
                 index=['Mobile', 'San Diego', 'Chicago',
                         'New York City', 'Oklahoma City'],
                 name='City Elevations')

ser4['Mobile']                                # 218
ser4.get('New York City')                     # 13
ser4['Albuquerque']                           # KeyError
ser4.get('Albuquerque')                       # None
ser4.get('Albuquerque', -1)                   # -1
ser4[['Chicago', 'Oklahoma City']]            # new series
ser4.Chicago                                  # 619

ser4[1:3]                                     # San Diego 15
                                                # Chicago 619
ser4.name                                     # City Elevations
ser4.median()                                 # 218.0
```

ch05\_pandas/02\_accessing\_series.py

Series work very much like lists and/or dictionaries. They support access using `get()`, but they also support slice notation in a similar fashion to lists. Series can also use the axis labels as properties (as demonstrated with the `ser4.Chicago` example above).



# Accessing and Working with Series

- `series.describe()` returns a *new summary Series* of the original Series:

```
ser4.describe()
```

count	5.000000
mean	432.000000
std	541.983395
min	13.000000
25%	15.000000
50%	218.000000
75%	619.000000
max	1295.000000
Name:	City Elevations, dtype: float64
<class 'pandas.core.series.Series'>	

```
ser4.describe() [ 'mean' ]
```

432.0

```
ser4.index
```

Index(['Mobile', 'San Diego', 'Chicago', 'New York City', 'Oklahoma City'], dtype='object')

```
ser4.values
```

[ 218 15 619 13 1295]

ch05\_pandas/02\_accessing\_series.py

When accessing a Series (beyond what was mentioned on the previous slide), the index and values attributes as well as the describe() method can provide helpful information. Results of these are shown in the slide above.

# Plotting Series

- Pandas data structures support Matplotlib data plotting arguments:

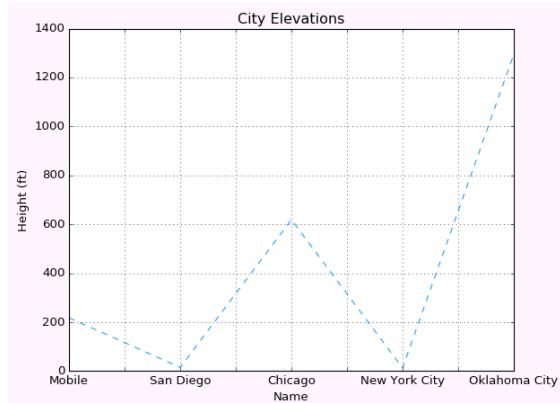
```
import matplotlib.pyplot as plt
import pandas as pd

ser4 = pd.Series([218, 15, 619, 13, 1295],
                 index=['Mobile', 'San Diego', 'Chicago',
                         'New York City', 'Oklahoma City'],
                 name='City Elevations')

figure = plt.figure()
axis = figure.add_subplot(111)
axis.set_title('City Elevations')
axis.set_xlabel('Name')
axis.set_ylabel('Height (ft)')

# same as ser4.plot.line()
ser4.plot(style='--', kind='line',
           ax=axis, color='#24a1f2',
           grid=True)

plt.show()
```



ch05\_pandas/03\_plotting.py

Pandas supports plotting Series as part of the API. It uses the Matplotlib API, therefore all of the options discussed in the previous chapter are available as well. These options include: kind ('bar', 'line', 'barh', 'hist', 'box', 'kde', 'density', 'area', 'pie'), ax, figsize, use\_index, legend, grid, title, style, xticks, yticks, xlim, ylim, rot (rotation), label, fontsize, xerr, yerr, secondary\_y, and more.

# DataFrames

- Pandas **DataFrames** are two-dimensional data structures containing labels on each axis

```
pd.DataFrame(data, index, columns, dtype)
```

- Conceptually similar to a spreadsheet
- DataFrames support an **index** (rows labels) and **columns** (column labels)

```
df5 = pd.DataFrame(np.arange(16).reshape(4, 4),
                   index=['row0', 'row1', 'row2', 'row3'],
                   columns=['col0', 'col1', 'col2', 'col3'])
```

	col0	col1	col2	col3
row0	0	1	2	3
row1	4	5	6	7
row2	8	9	10	11
row3	12	13	14	15

DataFrames are the primary data structure of Pandas. They are essentially a Series of Series (similar to a dictionary of dictionaries). Each column can be a different data type and both rows and columns can be inserted and deleted.

# Creating DataFrames

- DataFrames can be created several ways:

```
df1 = pd.DataFrame([['Colorado Springs', 'Phoenix', 'Raleigh',
                     'Milwaukee', 'Seattle'],
                     [6172, 1132, 437, 723, 429]])
```

```
print(df1)
```

Positional-based index and columns

	0	1	2	3	4
0	Colorado Springs	Phoenix	Raleigh	Milwaukee	Seattle
1	6172	1132	437	723	429

```
df1.index = ['City', 'Elevation']
```

```
print(df1)
```

	0	1	2	3	4
City	Colorado Springs	Phoenix	Raleigh	Milwaukee	Seattle
Elevation	6172	1132	437	723	429

Label-based indexing

ch05\_pandas/04\_dataframes.py

The top half of the example shows a DataFrame created using two nested lists. The lists each form a row within the DataFrame. No index or column index is defined so positional numerical values (0, 1) and (0, 1, 2, 3, 4) are used automatically.

The bottom half of the example defines an index for the DataFrame. Notice how the index is used to define the rows.

# Creating DataFrames Using a Dict of Lists

```
import pandas as pd

data = {
    'elevation': [6172, 1132, 437, 723, 429],
    'median_age': [34.5, 32.8, 32.8, 31.1, 36.1],
    'median_income': [53550, 46601, 55170, 35186, 70172],
}

df2 = pd.DataFrame(data, index=['Colorado Springs', 'Phoenix',
                                 'Raleigh', 'Milwaukee', 'Seattle'])

print(df2)
```

A dict of lists can be used to create a DataFrame

	elevation	median_age	median_income
Colorado Springs	6172	34.5	53550
Phoenix	1132	32.8	46601
Raleigh	437	32.8	55170
Milwaukee	723	31.1	35186
Seattle	429	36.1	70172

ch05\_pandas/04\_dataframes.py

Shown here is yet another way to create a DataFrame. A dictionary whose values are lists is provided when instantiating the DataFrame. An index is also provided resulting in the diagram shown at the bottom. Notice how the keys of the dictionary become the column names.

# Creating DataFrames Using ndarrays

```

Python 3
Data Analysis

data = [
    [6172, 1132, 437, 723, 429],
    [34.5, 32.8, 32.8, 31.1, 36.1],
    [53550, 46601, 55170, 35186, 70172]
]

columns = ['elevation', 'median_age', 'median_income']
index = ['Colorado Springs', 'Phoenix', 'Raleigh',
         'Milwaukee', 'Seattle']

arr = np.array(data, dtype=int)
df3 = pd.DataFrame(arr.transpose(), index=index, columns=columns)

print(df3)

```

A NumPy array can be used to create a DataFrame

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

ch05\_pandas/04\_dataframes.py

In this last version, a DataFrame is created by taking a list of lists containing our data. In its current form, the data would display in the wrong order. The columns would appear as rows instead. So, we need to modify the data a bit. One way to do this is to create a NumPy array and transpose the array so that rows become columns. This yields a DataFrame with the structure shown. Index and column values are applied as well.

# Working with DataFrames

df4

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

**df4.values**

Returns a NumPy Array of just the DataFrame data

```
[ [ 6172    34  53550]
  [ 1132    32  46601]
  [ 437     32  55170]
  [ 723     31  35186]
  [ 429     36  70172]]
```

**df4.T**

Returns the transpose (view) of the array

	Colorado	Springs	Phoenix	Raleigh	Milwaukee	Seattle
elevation		6172	1132	437	723	429
median_age		34	32	32	31	36
median_income		53550	46601	55170	35186	70172

ch05\_pandas/05\_dataframe\_info.py

In the same way that there are numerous ways to create DataFrames, there are lots of techniques for accessing and utilizing DataFrames.

If just the DataFrame values are desired, a NumPy array can be extracted from the DataFrame using the `values` attribute.

If statistical summary information is desired, use the `describe()` function which returns a new DataFrame.



# DataFrame Summary and Metadata

`df4.describe()` Provides summary statistical information

You can transpose  
this as well!

	elevation	median_age	median_income
count	5.000000	5.0	5.000000
mean	1778.600000	33.0	52135.800000
std	2472.633057	2.0	12791.018555
min	429.000000	31.0	35186.000000
25%	437.000000	32.0	46601.000000
50%	723.000000	32.0	53550.000000
75%	1132.000000	34.0	55170.000000
max	6172.000000	36.0	70172.000000

`df4.info(verbose=True)` Provides metadata info about a DataFrame

```
<class 'pandas.core.frame.DataFrame'>
Index: 5 entries, Colorado Springs to Seattle
Data columns (total 3 columns):
elevation      5 non-null int32
median_age     5 non-null int32
median_income   5 non-null int32
dtypes: int32(3)
memory usage: 100.0+ bytes
```

ch05\_pandas/05\_dataframe\_info.py

The `info()` function can provide information about the structure of the DataFrame including column information and datatype information.

The `T` attribute can provide a transpose view of the DataFrame. It doesn't, however, modify the original DataFrame.

# Accessing DataFrames

- Data from the DataFrame can be accessed numerous ways

Get elevation column,  
returns a Series

df4

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

`df4['elevation']`  
`df4.get('elevation')`

```
Colorado Springs      6172
Phoenix              1132
Raleigh              437
Milwaukee            723
Seattle              429
Name: elevation, dtype: int32
```

`df4['elevation'].Phoenix`

1132

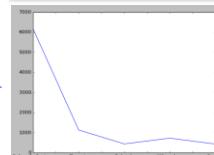
Select column, then row as attribute

`df4.get('population')`

None

Column doesn't exist

`df4.get('elevation').plot()`  
`plt.show()`



ch05\_pandas/06\_accessing\_dataframes.py

In the examples above, `df4['elevation']` returns a Series object. You can perform either a direct access or use the `get()` method. In the event that the column is not there (as in the case with the 'population' query), either a `KeyError` occurs or a `None` value is returned.

Also note how easy it is to select a column and plot it.



# Accessing DataFrames (continued)

Specifying a range of rows

```
df4[1:3]
```

	elevation	median_age	median_income
Phoenix	1132	32	46601
Raleigh	437	32	55170

Specifying multiple columns

```
df4[['elevation', 'median_income']]
```

	elevation	median_income
Colorado Springs	6172	53550
Phoenix	1132	46601
Raleigh	437	55170
Milwaukee	723	35186
Seattle	429	70172

Select a column and then a row

```
df4['elevation']['Colorado Springs'] 6172
```

ch05\_pandas/06\_accessing\_dataframes.py

Here we can see three more ways to access DataFrames. First, a slice of rows is selected, returning a new DataFrame. In the middle example, multiple columns are selected using a nested sequence. This also returns a new DataFrame. Finally, both a column and row can be selected to retrieve an individual data value.

# Accessing DataFrames (continued)

Specifying a range of rows, and a specific column

```
df4[1:4]['elevation']
```

```
Phoenix      1132
Raleigh      437
Milwaukee    723
Name: elevation, dtype: int32
```

Notes on selecting:

When labels are specified first, columns are selected.

When a range of indices are specified first, rows are selected.

ch05\_pandas/06\_accessing\_dataframes.py

This example uses slicing to retrieve multiple rows and then selects the whole 'elevation' column.

Note: as mentioned earlier, **df.values** may also be used to obtain the NumPy Array.



# Improving Access with loc[] and iloc[]

- loc[] and iloc[] provide better consistency for selecting data from DataFrames

```
df5 = pd.DataFrame(np.arange(16).reshape(4, 4),
                   index=['row0', 'row1', 'row2', 'row3'],
                   columns=['col0', 'col1', 'col2', 'col3'])
```

	col0	col1	col2	col3
row0	0	1	2	3
row1	4	5	6	7
row2	8	9	10	11
row3	12	13	14	15

```
df5.iloc[1:3]
```

	col0	col1	col2	col3
row1	4	5	6	7
row2	8	9	10	11

```
df5.loc['row1']
```

	col0	col1	col2	col3
row1	4	5	6	7
Name: row1, dtype: int32				

```
df5.iloc[2:, 2:]
```

	col2	col3
row2	10	11
row3	14	15

```
df5.loc['row2':, 'col2':]
```

	col2	col3
row2	10	11
row3	14	15

ch05\_pandas/07\_indexing.py

The two indexing methods shown here are similar, but have some differences. iloc[index] only supports integer-based, positional index values even if there are labels provided. The loc[index] is used with labels in the index.

Depending on what is selected, the results will either be a new Series, DataFrame, or individual value.

A third technique for accessing data, ix[], is now deprecated.



# Labels vs Positions with loc[] and iloc[]

```
ser = pd.Series(1, index=[10, 9, 8, 1, 2, 3, 4, 5])
```

10	1
9	1
8	1
1	1
2	1
3	1
4	1
5	1

dtype: int64

ser.iloc[:3]

10	1
9	1
8	1

ser.loc[:3]

10	1
9	1
8	1
1	1
2	1
3	1

```
df = pd.DataFrame(1, columns=[4, 3, 2, 1],
                  index=[10, 9, 8, 1, 2, 3, 4, 5])
```

	4	3	2	1
10	1	1	1	1
9	1	1	1	1
8	1	1	1	1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	1	1	1	1

df.iloc[:3, :3]

	4	3	2
10	1	1	1
9	1	1	1
8	1	1	1

df.loc[:3, :3]

	4	3
10	1	1
9	1	1
8	1	1
1	1	1
2	1	1
3	1	1

ch05\_pandas/22\_loc\_vs\_iloc.py

There are two examples here each illustrating how loc[] and iloc[] work. The first example shows a series, while the second depicts a DataFrame. Using the same indexing values for loc[] and iloc[] notice how the results are different. loc[] focuses on the values as labels. iloc[] only considers actual index values and not the provided labels.

# Index (and Column) Sorting

- Data can be "reordered" using `pd.sort_index()`

`df4`

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

`df6 = df4.sort_index()``df6`

Reorders the index,  
df4 is unchanged

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Milwaukee	723	31	35186
Phoenix	1132	32	46601
Raleigh	437	32	55170
Seattle	429	36	70172

Use `df4.sort_index(inplace=True)` to change df4 directly

`df7 = df4.sort_values(by='median_age')`

Sorts the DataFrame on  
the median\_age column

Use `ascending=False` to sort from largest to smallest

ch05\_pandas/08\_index\_sorting.py

In the example in the middle of the slide, rows are reordered using the `sort_index()` method. Using `sort_values()`, as in the bottom example, will sort the DataFrame by the specified column value.



## DataFrame min() and max()

- `df.max(axis=None)` - returns the maximum values in a DataFrame
- `df.min(axis=None)` - returns the minimum values in a DataFrame
- Use `df.idxmax()` or `df.idxmin()` to get the associated index for the max or min value

```
df = pd.DataFrame(index=['Colorado Springs', 'Phoenix', 'Raleigh',
                         'Milwaukee', 'Seattle'],
                   data=[6172, 1132, 437, 723, 429])

print(df.max()[0])                                # 6172
print(df.idxmax()[0])                            # Colorado Springs
```

df.max() and df.idxmax() each return a Series



# Pandas read\_csv()

- `pd.read_csv()` reads data files into a DataFrame

– Useful arguments include:

- `delimiter=','` data item separator character
- `skiprows=None` how many rows to skip at the beginning
- `encoding='utf-8'` file encoding type
- `nrows` number of rows to read
- `header` which row to use for the column headers

Doesn't consider the first  
line as the header now

```
contacts = pd.read_csv('contacts.dat', header=None, dtype='str')
contacts.columns = ['name', 'address', 'state', 'zip',
                    'area_code', 'phone', 'email', 'company', 'position']
```

ch05\_pandas/09\_read\_csv.py

The signature of the `read_csv()` method includes many optional arguments:

```
pd.read_csv(filepath_or_buffer, sep, delimiter, header, names,
            index_col, usecols, squeeze, prefix, mangle_dupe_cols,
            dtype, engine, converters, true_values, false_values,
            skipinitialspace, skiprows, skipfooter, nrows,
            na_values, keep_default_na, na_filter, verbose,
            skip_blank_lines, parse_dates, infer_datetime_format,
            keep_date_col, date_parser, dayfirst, iterator,
            chunksize, compression, thousands, decimal,
            lineterminator, quotechar, quoting, escapechar,
            comment, encoding, dialect, tupleize_cols,
            error_bad_lines, warn_bad_lines, skip_footer,
            doublequote, delim_whitespace, as_recarray,
            compact_ints, use_unsigned, low_memory,
            buffer_lines, memory_map, float_precision)
```



# Boolean Indexing (Masking) in Pandas

- Data values that match the given Boolean criterion will be seen in the resulting data structure
  - Example: Find only records where *BOTH* the high and low temps are above 51 degrees:

```
sat_temps = pd.DataFrame(data=[(78, 50), (82, 52), (83, 53)],
                           index=['Colorado Springs', 'Canon City', 'Pueblo'],
                           columns=['High', 'Low'])
```

	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53

```
sat_temps.loc[(sat_temps['High'] >= 51) & (sat_temps['Low'] >= 51)]
```

	High	Low
Canon City	82	52
Pueblo	83	53

ch05\_pandas/13\_boolean\_indexing.py

Note: if you are wondering why you can't simply use the Python **and** operator instead of **&**, it is because **and** has been written to take the left and right sides, convert them to Booleans first, and then evaluate the expression. This behavior can't be changed in Python. However, the **&** operator's behavior can be changed. The same is true for the **or** operation, use **/** instead.

## head() and tail()

- Use `df.head(n=5)` or `df.tail(n=5)` to display the first n or last n rows of a DataFrame

```
import pandas as pd
contacts = pd.read_csv('contacts2.dat', header=None, dtype='str')
contacts.columns = ['name', 'address', 'state', 'zip', 'area_code',
                    'phone', 'email', 'company', 'position']

print(contacts.head(3))
```

Displays the first 3 rows of the DataFrame

Bob Green, 4517 Elm St. Riverside, NJ,08075,301,356-8921,bob@abc.com, ...
Violet Smith, 220 E. Main Ave Philadelphia, PA,09119,202,421-9008, ...
John Brown, 231 Oak Blvd. Black Hills, SD,82101,719,303-1219, ...

`head()` or `tail()` can provide a partial view of a data structure. By default, each returns either the first 5 or the last 5 rows in the DataFrame.



## Your Turn! - Task 5-1

- Repeat the Batting Average exercise this time using only Pandas
  - Use pd.read\_csv() to read data from the file
  - Keep only records where atbats is 502 or more
  - Keep only records where the year is 1957 or later
  - Sort the records using Pandas sort\_values() method
  - Display the top 100 batting averages
- Determine the 50th percentile batting average
- Create a scatter plot from the DataFrame data
  - Use:

```
df.plot(kind='scatter', x='averages', y='atbats')
```

Work from the task5\_1\_starter.py, or create a new Jupyter Notebook file



## Additional Features Using Pandas

- Concatenating DataFrames
- MultiIndexing and Boolean indexing
- Renaming Columns
- Imputing Values
- Groupby:
  - Split
  - Apply
  - Combine
- apply()
- Other read methods



# Merging DataFrames

- The correct choice of method (and options) to combine data sets can greatly affect the resulting DataFrame
- `concat()` - joins DataFrames *along a given axis*
  - Use this approach when "stacking" records
- `merge()` - joins two DataFrames *by aligning values in a column or index*
  - Best suited when joining by values found in *common columns*, similar to a database style join

Two additional methods, `join()` and `append()`, can also be used to concatenate. `pd.join()` is a shorthand version of `merge()` with `left_index=True`. `pd.append()` is similar to `pd.concat()` with `axis=0` and `join='outer'` parameters. Because of this, `join()` and `append()` won't be discussed further on subsequent slides.

# concat() Arguments

- Syntax for concat() is:

```
pd.concat(objs, axis=0, join='outer | inner', join_axes=None,  
          ignore_index=False, keys=None, levels=None,  
          names=None, verify_integrity=False, copy=True)
```

objs -	sequence of Series or DataFrames to join
axis -	concatenate along this axis
join -	'inner' or 'outer' (def.), outer=union, inner=intersection
ignore_index -	True   False (def.) True=don't use idx values on concat axis
join_axes -	join these specific axes instead of join 'inner' or 'outer'
keys -	sequence, constructs hierarchical index of joined data
levels -	list of sequences, identifies levels of multi-index
names -	list of names for the levels in hierarchy
verify_integrity -	boolean (def. False), check if new concat contains dups.
copy -	boolean (def. True), False=do not copy data

# Basic concat() Example

- Use **concat()** for most DataFrame concatenation requirements:

```
sat_temps = pd.DataFrame(data=[(78, 50), (82, 52), (83, 53)],
                           index=['Colorado Springs', 'Canon City', 'Pueblo'],
                           columns=['High', 'Low'])

sun_temps = pd.DataFrame(data={'High':(77, 81, 84), 'Low':(48, 49, 50)},
                           index=['Colorado Springs', 'Canon City', 'Pueblo'])

merged = pd.concat([sat_temps, sun_temps])
```

sat_temps		High	Low
Colorado Springs		78	50
Canon City		82	52
Pueblo		83	53

sun_temps		High	Low
Colorado Springs		77	48
Canon City		81	49
Pueblo		84	50



	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53
Colorado Springs	77	48
Canon City	81	49
Pueblo	84	50

ch05\_pandas/11\_merging\_dataframes.py

In the example above, two DataFrames are created and then merged, relying on all default values. The results of the concatenation are shown in the lower right table.



# concat() with keys, ignore\_index

```
merged = pd.concat([sat_temps, sun_temps], keys=['Saturday', 'Sunday'])
```

			High	Low
Saturday	Colorado Springs	78	50	
	Canon City	82	52	
	Pueblo	83	53	
Sunday	Colorado Springs	77	48	
	Canon City	81	49	
	Pueblo	84	50	

`merged.loc['Saturday', 'Colorado Springs']['High']`

```
merged = pd.concat([sat_temps, sun_temps], ignore_index=True)
```

	High	Low
0	78	50
1	82	52
2	83	53
3	77	48
4	81	49
5	84	50

keys will be ignored if  
ignore\_index is used

ch05\_pandas/11\_merging\_dataframes.py

By supplying a *keys* list, a hierarchy of the merged data can be created. Accessing the hierarchy is managed by `loc[]`.

Indices can also be ignored using `ignore_index=True` and replaced with numerical index values.

# MultIndex DataFrames

- The previous merged DataFrame uses hierarchical indexing (or MultIndexing)

`merged`

			High	Low
Saturday	Colorado Springs		78	50
	Canon City		82	52
	Pueblo		83	53
Sunday	Colorado Springs		77	48
	Canon City		81	49
	Pueblo		84	50

`merged.index`

`Level 0`      `Level 1`

```
MultiIndex(levels=[['Saturday', 'Sunday'], ['Colorado Springs', 'Canon City', 'Pueblo']],
           labels=[[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]])
```

---

		High	Low
Saturday	Colorado Springs	78	50
Sunday	Colorado Springs	77	48

ch05\_pandas/11\_merging\_dataframes.py

Our merged DataFrame has a two-level index, or a MultIndex. The level 0 indices are 'Saturday' and 'Sunday', while the level 1 indices are 'Colorado Springs', 'Canon City', 'Pueblo'.

The bottom example uses Boolean Indexing to determine how to filter values out. We used the MultIndex object method, `get_level_values(n)`, which provides the ability to check for only 'Colorado Springs' index values at level 1. At level 0, we could check for 'Saturday' and 'Sunday' values.

To give names to the MultIndex column names, use the `names` property:  
`merged.index.names = ['Day', 'City']`

# concat() Along an Axis

```
sat_temps = pd.DataFrame(data=[(78, 50), (82, 52), (83, 53)],
                           index=['Colorado Springs', 'Canon City', 'Pueblo'],
                           columns=['High', 'Low'])

sat_humidity = pd.DataFrame([22, 18, 19, 25],
                            index=['Colorado Springs', 'Canon City', 'Pueblo', 'Denver'],
                            columns=['Humidity'])

merged = pd.concat([sat_temps, sat_humidity], axis=1)
```

sat_temps	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53

sat_humidity	Humidity
Colorado Springs	22
Canon City	18
Pueblo	19
Denver	25



	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

ch05\_pandas/11\_merging\_dataframes.py

In the example, we have concatenated humidity data onto the Saturday temperature data along axis 1. This results in the Denver row of data having no temperature information (since it didn't exist in the first DataFrame). A NaN value is placed into these positions.

# concat() Using join=

sat\_temps

	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53

sat\_humidity

	Humidity
Colorado Springs	22
Canon City	18
Pueblo	19
Denver	25

```
merged = pd.concat([sat_temps, sat_humidity], axis=1, join='inner')
```

	High	Low	Humidity
Colorado Springs	78	50	22
Canon City	82	52	18
Pueblo	83	53	19

Only common rows are retained. (def. is 'outer' which is a union of the rows)

ch05\_pandas/11\_merging\_dataframes.py

Concatenation can be performed by specifying exactly which rows to retain or discard. By default, the *join* argument is set to the value "outer" which performs a *union* of the two DataFrames. By using a value of "inner," the *intersection* of row indices will be used. Finally, by specifying *join\_axes*, the common rows from the stated indices will be used.



# Merging DataFrames with merge()

- `merge()` behaves more like SQL

```
merge(left, right, how='inner', on=None, left_on=None,  
right_on=None, left_index=False, right_index=False,  
sort=False, suffixes=('_x', '_y'), copy=True, indicator=False)
```

left -	a left DataFrame
right -	a right DataFrame
on -	column names to join on. Must be in both left & right
left_on -	columns from the left DataFrame to use as keys
right_on -	columns from the right DataFrame to use as keys
left_index -	if True, use the index (row labels) from the left DataFrame as the keys
right_index -	same as left_index, but for the right DataFrame
how -	type of join operation. Either: 'left', 'right', 'outer', 'inner' (def.)
sort -	sort the result DataFrame by the join keys (def. is True)
suffixes -	tuple of strings to apply to overlapping columns
copy -	always copy data (def. is True) from the passed DataFrame
indicator -	add a column called _merge with info on the source of rows

# Using merge()

```
data1 = np.arange(1, 10).reshape(3,3)
df1 = pd.DataFrame(data1,
                    columns=['a', 'b', 'c'])
data2 = np.arange(1, 13).reshape(4,3)
df2 = pd.DataFrame(data2, columns=['d', 'b', 'e'])
```

df1

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

df2

	d	b	e
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

```
merged1 = pd.merge(df1, df2, on='b')
```

merged1

	a	b	c	d	e
0	1	2	3	1	3
1	4	5	6	4	6
2	7	8	9	7	9

merged2

```
merged2 = pd.merge(df1, df2, left_on='a',
                    right_on='d', how='outer')
```

	a	b_x	c	d	b_y	e
0	1.0	2.0	3.0	1	2	3
1	4.0	5.0	6.0	4	5	6
2	7.0	8.0	9.0	7	8	9
3	NaN	NaN	NaN	10	11	12

ch05\_more\_pandas/11\_merge.py

There are two merge examples shown here. The first merges on a column that exists in both DataFrames, column b.

Given the two DataFrames shown (df1 and df2), you can merge on a common column name, such as column 'b'. Because the *how* attribute defaults to the value "inner", which means "keep the rows containing common values," the row containing the value 11 found in df2 only is dropped.

In the second example, we merged on values from column 'a' in the first DataFrame and column 'd' in the second DataFrame. Because both DataFrames have a column entitled 'b' and because we are not merging on 'b' this time, both 'b' columns are brought in but given a naming convention (\_x and \_y) to differentiate them.

# Using merge() (continued)

df1

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

df2

	d	b	e
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

how='inner'  
(default)

```
merged3 = pd.merge(df1, df2, left_on='a',
                    right_on='d')
```

merged3

	a	b_x	c	d	b_y	e
0	1	2	3	1	2	3
1	4	5	6	4	5	6
2	7	8	9	7	8	9

merged4

```
merged4 = pd.merge(df1, df2, left_index=True,
                    right_index=True)
```

	a	b_x	c	d	b_y	e
0	1	2	3	1	2	3
1	4	5	6	4	5	6
2	7	8	9	7	8	9

ch05\_more\_pandas/11\_merge.py

In the third example (continuing from the previous slide), we used the default how='inner' which results in dropping the NaN row because the 10 value in column 'd' is not common between the 'a' and 'd' columns.

In the bottom example, we simply joined by index values. With how='inner' again, the non-common row is removed.

# Renaming Columns (2 ways)

- The **columns** attribute to rename columns:

	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

	High	Low	Pct Hum
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

```
cols = temps_df.columns.values
cols[2] = 'Pct Hum'
temps_df.columns = cols
print(temps_df)
```

This is an ndarray. The 3<sup>rd</sup> col value is changed and set back to the columns object.

- You can also use the **rename()** method

From this	To this
<code>temps_df.rename(columns={'Pct Hum': '% Hum'}, inplace=True)</code>	

	High	Low	% Hum
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

ch05\_pandas/11\_merging\_dataframes.py

In the example above, the 3<sup>rd</sup> column (Humidity) is renamed to 'Pct Hum' by acquiring the column names (tempo\_df.columns.values), changing the 3<sup>rd</sup> column value, and then reassigning it back to the columns attribute of the DataFrame.

The **rename()** technique accepts a dictionary of column value names to change. Use **inplace=True** to avoid creating a new DataFrame.

# Imputing Missing Values

- Use df.fillna() to deal with missing data values

```
df.fillna(value=None, axis=None, inplace=False)
```

temps\_df

	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

Use temps\_df.dropna() to drop rows that contain NaN values

```
temps_df['High'].fillna(value=80, inplace=True)
temps_df['Low'].fillna(52, inplace=True)
```

temps\_df

	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	80.0	52.0	25
Pueblo	83.0	53.0	19

ch05\_pandas/12\_imputing\_copying\_deleting.py

In this example, columns have been selected individually. fillna() has been applied to each column. Using fillna() on the entire DataFrame will apply the changes throughout.

# Pandas NaN and Infinity

- Pandas uses NumPy values for representing NaN and infinity (`np.nan`, `np.inf`):

	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

```
temps_df.loc[(tempo_df['High'] > 80), 'High'] = np.inf
```

Converts any values in the 'High' column > 80 to infinity

	High	Low	Humidity
Canon City	inf	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	inf	53.0	19

ch05\_pandas/19\_dropna.py

In this example, we took our `tempo_df` from previous examples and set any value in the 'High' column to infinity using NumPy's inf. The shaded area indicates where Boolean indexing is being used.

# Removing Infinity Values

- Pandas doesn't have a specific method for removing infinity values
  - Use `replace()` to first convert the values to NaN, then use `dropna()` to remove them

`temps_df`

	High	Low	Humidity
Canon City	inf	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	inf	53.0	19



```
temps_df.replace([np.inf, -np.inf], np.nan, inplace=True)
```

`temps_df`

	High	Low	Humidity
Canon City	NaN	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	NaN	53.0	19

ch05\_pandas/19\_dropna.py

Note: You can get rid of values containing +inf or -inf values, by converting these columns to NaN values first and then dropping the rows (next slide).

# Dropping Rows Containing NaNs

temps\_df

		High	Low	Humidity
Canon City		NaN	52.0	18
Colorado Springs		78.0	50.0	22
Denver		NaN	NaN	25
Pueblo		NaN	53.0	19



```
temps_df.dropna(subset=['Low'], inplace=True)
```

temps\_df

		High	Low	Humidity
Canon City		NaN	52.0	18
Colorado Springs		78.0	50.0	22
Pueblo		NaN	53.0	19

Note: use **subset=** to only consider certain columns, leave off to examine all columns

ch05\_pandas/19\_dropna.py

Now, rows containing NaN values can be dropped.



## "Group By" Operations

- Much like SQL queries, DataFrame data can be grouped and then operated upon in groups
  - Often this process can be broken down further into discrete steps referred to as:

**Split - Apply - Combine**

- **Split** - groups data according to some criteria
- **Apply** - performs operations on the grouped data
- **Combine** - places resulting values into a new structure

# Contacts Sample Data

Bob Green,	4517 Elm St. Riverside,	<i>NJ</i> , 08075, 301, 356-8921
Violet Smith,	220 E. Main Ave Philadelphia,	<i>PA</i> , 09119, 202, 421-9008
John Brown,	231 Oak Blvd. Black Hills,	<i>SD</i> , 82101, 719, 303-1219
Ed Blumenthal,	3012 Briarwood Ln. Denver,	<i>CO</i> , 80101, 719, 422-8091
Rosey Englund,	1818 Mockingbird Ln. Aurora,	<i>CO</i> , 82101, 719, 286-1920
Tori Gray,	2218 Masengild Ave.,	<i>NJ</i> , 08075, 301, 338-6571
Lisa Black,	89 Prince Dr. Philadelphia,	<i>PA</i> , 09119, 202, 419-0650
Tom Redford,	2323 Nicholas St. Newark,	<i>NJ</i> , 07101, 862, 227-8022
Sally White,	3345 Spruce Cir. Harrisburg,	<i>PA</i> , 17105, 717, 429-1217
Goldy Simpson,	4430 Mountainside Creek Rd Custer,	<i>SD</i> , 57730, 605, 689-3131
O. Range,	1703 Treeline Dr. Denver,	<i>CO</i> , 80101, 719, 429-1356
Sil Verna,	557 Pine Ave Aurora	<i>CO</i> , 82101, 719, 286-1920
Pinky Tuscadero,	601 Sapling Blvd.,	<i>NJ</i> , 08501, 609, 227-6001
Hazel Sanford.	27 Musket Dr. Pittsburg.	<i>PA</i> , 15201, 412, 389-7711
<i>bob@abc.com</i> ,	ABC Inc.,	President
<i>ssmithj@hypex.org</i> ,	FakeCo Inc.,	Janitor
<i>vivoj@wandergem.com</i> ,	Wandergem LLC,	Sr. Analyst
<i>ep20002@gmail.com</i> ,	Hanibow & Delite,	Programmer
<i>ke7001@yahoo.com</i> ,	Wadlow Inc.,	Administrative Lead
<i>tjames@acme.com</i> ,	Acme Inc.,	Inventor
<i>victors89@glaser.org</i> ,	Glaser Properties LLC.,	Manager
<i>tom.redford@gmail.com</i> ,	Illustrative Studio Systems,	Graphics Engineer
<i>swhope@ggworth.com</i> ,	Bond Appliances,	Administrative Specialist
<i>simpson@yahoo.com</i> ,	Crater Construction,	Owner
<i>ffnine27@hotmail.com</i> ,	n/a,	Retired
<i>sil@yahoo.com</i> ,	Music Enthusiasts,	Salesperson
<i>pinky@freedom.net</i> ,	Self-employed,	Vocal Artist
<i>hazel@outlook.com</i> ,	Bourne Legal Associates,	Paralegal

ch05\_pandas/contacts2.dat

Each record in our file has a name, address (includes city), state, zip, area code, phone, email, company, and position.

# Grouping Contacts By State

- To segment (split) data into groups, use the **groupby()** method:

```
df.groupby(key)
df.groupby([key1, key2])
df.groupby(key, axis=1)
```

```
contacts = pd.read_csv('contacts2.dat', header=None,
                       names=['name', 'address', 'state', 'zip', 'area_code',
                              'phone', 'email', 'company', 'position'],
                       converters={'state': lambda txt: txt.strip()})
```

Used to strip whitespace from specified column

```
bystate = contacts.groupby('state')
print(bystate.groups)
```

```
{'NJ': [0, 5, 7, 12], 'SD': [2, 9],
 'CO': [3, 4, 10, 11], 'PA': [1, 6, 8, 13]}
```

ch05\_pandas/10\_grouping.py

The default of the `groupby()` is to sort according to the selected *key(s)* and make them the index of the resulting grouped DataFrame. In the example above, the indices of the resulting dictionary from the `groups` attribute are the keys from the `groupby()` call.

# "Grouped" Operations

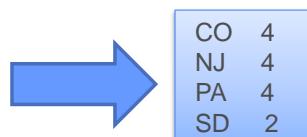
- Once data is "split" (grouped), it can be operated upon using a number of methods:

group.agg()	group.hist()
group.boxplot()	group.max() / min()
group.cumin()	group.idxmax() / idxmin()
group.describe()	group.nth(n)
group.filter()	group.prod()
group.get_group()	group.resample()
group.median()	group.sum()
group.ngroups	group.var()
group.plot()	group.apply()
group.rank()	group.cummax()
group.std()	group.cumsum(0)
group.transform()	group.fillna()
group.aggregate()	group.head() / tail()
group.count()	group.indices
group.cumprod()	group.mean()
group.first() / last()	group.name
group.groups	group.size()

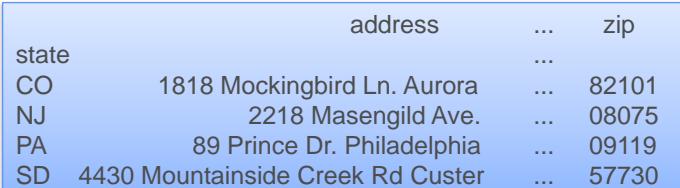
# Operations on "Grouped" Items

`bystate.size()`

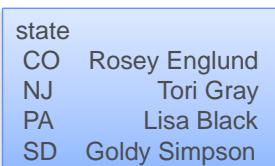
Returns a new Series

`bystate.first()`

Returns a DataFrame

`bystate.nth(1)``bystate.nth(1) ['name']`

Returns a Series



ch05\_more\_pandas/10\_grouping.py

In the examples above, we have grouped items from the original DataFrame using different functions. The `size()` function returns a new series with the length of each grouping.

In the second example, the first record from each group is returned. In the third example, the second record from each group is returned using `nth()` which re-orders the columns. If there is no second record, the group is removed from the result.

The last one simply extracts the `name` column from the DataFrame provided in the previous example.

# Selecting a Group

- Use `get_group()` to select just a single group to operate on:

```
bystate.get_group('CO')
```

	address	area_code	...	position	zip
3	3012 Briarwood Ln. Denver	719	...	Programmer	80101
4	1818 Mockingbird Ln. Aurora	719	...	Administrative Lead	82101
10	1703 Treeline Dr. Denver	719	...	Retired	80101
11	557 Pine Ave Aurora	719	...	Salesperson	82101

```
colorado = bystate.get_group('CO')
colorado[colorado['zip'] == 82101]
```

	address	area_code	...	position	zip
4	1818 Mockingbird Ln. Aurora	719	...	Administrative Lead	82101
11	557 Pine Ave Aurora	719	...	Salesperson	82101

ch05\_more\_pandas/10\_grouping.py

When only one of the groups is desired, use `get_group('key')` to select only the desired one. The result here is a DataFrame we called `colorado`. Now, all DataFrame-related actions can be performed on this new DataFrame.

# Pivot Tables (1 of 2)

- Pivot tables are summary tables created from an original table
  - Usually perform grouping operations on columns
  - Usually perform averages, sums, sorts on other columns
- Example:

Day	Highs	Lows	Humidity	Wind Speed	Outlook	Red Flag
1	88	68	25	10	Sunny	False
2	84	65	31	5	Cloudy	False
3	86	66	32	5	Light Rain	False
4	89	67	26	5	Rain	False
5	92	70	22	10	Sunny	False
6	95	71	18	20	Sunny	True
7	94	69	27	10	Sunny	False
8	93	72	25	10	Rain	False
9	98	76	16	5	Cloudy	True
10	94	72	22	10	Sunny	False

```
df = pd.DataFrame(data, columns=['Day', 'Highs', 'Lows', 'Humidity',
                                 'Wind Speed', 'Outlook', 'Red Flag'])
df.set_index('Day', inplace=True)
```

ch05\_pandas/23\_pivot\_table.py

The data shown here is similar to the data we encountered in an earlier exercise. A few additional records have been added. The Day column is established as the index in this example.

---

Pandas can do the simpler cross-tabulation tables also, here's an example:

```
cross = pd.crosstab(df.Outlook, df['Wind Speed'], margins=True)
```

Wind Speed	5	10	20	All
Outlook				
Sunny	0	4	1	5
Rain	1	1	0	2
Light Rain	1	0	0	1
Cloudy	2	0	0	2
All	4	5	1	10



## Pivot Tables (2 of 2)

```
df['Outlook'] = df['Outlook'].astype('category')
df['Outlook'].cat.set_categories(
    ['Sunny', 'Rain', 'Light Rain', 'Cloudy'], inplace=True)

pivot = pd.pivot_table(data=df, index=['Outlook'],
                       values=['Highs', 'Lows'], aggfunc=[np.mean, len])
```

Outlook	mean		len	
	Highs	Lows	Highs	Lows
Sunny	92.6	70.0	5	5
Rain	91.0	69.5	2	2
Light Rain	86.0	66.0	1	1
Cloudy	91.0	70.5	2	2

Set the 'Outlook' column to **category** and limit its values

index = column(s) we want to get summary info on  
values = the column(s) we want to apply aggregation functions to  
aggfunc = the functions to apply to the values columns

```
pivot.query('Outlook == ["Sunny"]')
```

Outlook	mean		len	
	Highs	Lows	Highs	Lows
Sunny	92.6	70.0	5	5

ch05\_pandas/23\_pivot\_table.py

The pivot\_table() function allows for specifying a column or columns that will be summarized using the index= parameter. The values parameter identifies the column or columns to apply aggregation functions to.

Not shown here is an option use of a dictionary for the aggfunc parameter. A dictionary could be used to individually specify different functions for different values columns.

The bottom illustrates how to filter desired rows from the pivot table, if desired.

# Working with Log Files

Note: for the complete solution, refer to:  
10\_reading\_a\_log\_file.ipynb

- Our data is a typical log file *not* a CSV

resources/new\_access.log

```
109.169.248.247 -- [12/Dec/2015:18:25:11 +0100] "GET /administrator/ HTTP/1.1" 200 4263 ...
109.169.248.247 -- [12/Dec/2015:18:25:11 +0100] "POST /administrator/index.php HTTP/1.1" 200 4494 ...
46.72.177.4 -- [12/Dec/2015:18:31:08 +0100] "GET /administrator/ HTTP/1.1" 200 4263 ...
46.72.177.4 -- [12/Dec/2015:18:31:08 +0100] "POST /administrator/index.php HTTP/1.1" 200 4494 ...
83.167.113.100 -- [12/Dec/2015:18:31:25 +0100] "GET /administrator/ HTTP/1.1" 200 4263 ...
83.167.113.100 -- [12/Dec/2015:18:31:25 +0100] "POST /administrator/index.php HTTP/1.1" 200 4494 ...
```

```
log = pd.read_csv('../resources/new_access.log', sep='\s+',
                  usecols=(0, 3, 5, 6, 7, 9),
                  names=['addr', 'req_date', 'request', 'status',
                         'size', 'browser'],
                  error_bad_lines=False)
print(log.shape) → (463915, 6)
print(log.head())
```

	addr	req_date	request	status	size	browser
0	109.169.248.247	[12/Dec/2015:18:25:11	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
1	109.169.248.247	[12/Dec/2015:18:25:11	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2	46.72.177.4	[12/Dec/2015:18:31:08	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
3	46.72.177.4	[12/Dec/2015:18:31:08	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...

ch05\_more\_pandas/20\_reading\_a\_log\_file.ipynb

Explanation of code:

To read from this file (it isn't a CSV, but we used `read_csv()` anyways), the `sep` attribute allowed us to specify a regular expression. Using `sep='\s+'`, we could break the line up by whitespace. This would, of course, yield more columns than we desired (such as the columns containing purely dashes). However, we can use the `usecols` attribute to select only the desired columns. Also, quoted items (items inside of the double-quotes) would be treated as one value when using the regular expression approach.



# Log Files Info & Converting the Date

- Learn about columns using info():

```
log.info()
```

This will happen **after** the code below executes!

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 463915 entries, 0 to 463914
Data columns (total 6 columns):
addr           463915 non-null object
req_date       463915 non-null datetime64[ns]
request        463915 non-null object
status          463915 non-null int64
size            463914 non-null object
browser         463860 non-null object
dtypes: datetime64[ns](1), int64(1), object(4)
memory usage: 21.2+ MB
```

- Convert the req\_date column into a date type:

```
log.req_date = pd.to_datetime(log.req_date, format='[%d/%b/%Y:%H:%M:%S']')
```

ch05\_more\_pandas/20\_reading\_a\_log\_file.ipynb

Explanation of code:

It is helpful to get a better understanding of how Pandas views your data. Use info() to get an idea of the memory footprint and column types.

The field holding the date gets converted into a datetime64 field. First, the leading square bracket, [, is stripped from the data value then it is converted to a Datetime object.

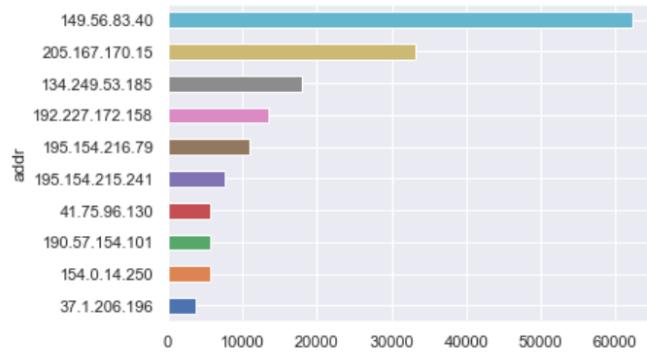


# Log Files: Grouping (by Address)

- You can group results in many ways:

```
top10_addr = log.groupby('addr').size().sort_values(ascending=False).head(10)
print(top10_addr)
```

```
addr
149.56.83.40      62178
205.167.170.15    33302
134.249.53.185    17904
192.227.172.158    13474
195.154.216.79    10996
195.154.215.241    7705
41.75.96.130      5664
190.57.154.101    5662
154.0.14.250      5659
37.1.206.196      3780
dtype: int64
```



```
top10_addr[::-1].plot(kind='barh')
```

Allows largest to appear on top

ch05\_more\_pandas/20\_reading\_a\_log\_file.ipynb

Explanation of code:

Eventually the data is grouped using `groupby()`. The grouping occurs by address. The groups are sized and sorted from largest to smallest. This yields the top 10 IP addresses that appear in the log file. We plotted those using Pandas and Matplotlib.



# Log Files: Grouping (by Browser)

```
log.groupby('browser').size()  
    .sort_values(ascending=False).head(10) [::-1].plot(kind='barh')
```



ch05\_more\_pandas/20\_reading\_a\_log\_file.ipynb

Explanation of code:

Grouping once again, this time we use the browser column.

# Custom Column Operations

- Use `apply()` to execute a custom function on cells
  - Here, the GET and POST values are extracted from the request column and placed into a new `http-method` column

```
log['http-method'] = log.request.apply(lambda s: s.split()[0])
print(log.head())
```

	addr	req_date	request	status	size	browser	http-method
0	109.169.248.247	2015-12-12 18:25:11	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	GET
1	109.169.248.247	2015-12-12 18:25:11	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	POST
2	46.72.177.4	2015-12-12 18:31:08	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	GET
3	46.72.177.4	2015-12-12 18:31:08	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	POST
4	83.167.113.100	2015-12-12 18:31:25	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	GET

ch05\_more\_pandas/20\_reading\_a\_log\_file.ipynb

Explanation of code:

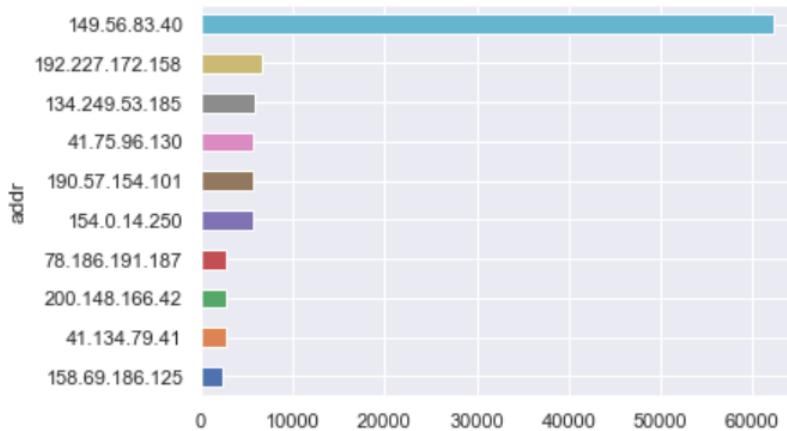
In this example, we used the `apply()` method to execute a custom function. The results of the function are placed into a new column called 'http-method'.



# Grouping Selected Records

```
top10_POSTS = log[log['http-method'] == 'POST'].groupby('addr').size()  
              .sort_values(ascending=False).head(10)  
top10_POSTS[::-1].plot(kind='barh')
```

Only top 10 POST requests are viewed



ch05\_more\_pandas/20\_reading\_a\_log\_file.ipynb

Explanation of code:

Using techniques previously discussed, this time we filtered out all but the POST style requests.

# Constructing New Data

```

import pygeoip
GEOIP = pygeoip.GeoIP('../resources/GeoLiteCity.dat')
GEOIP.record_by_addr('149.56.83.40') →
{'postal_code': 'H3A',
'country_code': 'CA',
'country_code3': 'CAN',
'country_name': 'Canada',
'continent': 'NA',
'region_code': 'QC',
'city': 'Montréal',
'latitude': 45.50399999999999,
'longitude': -73.5747,
'time_zone': 'America/Montreal'}

def get_location(addr):
    results = ['', '', 0, 0]
    try:
        info = GEOIP.record_by_addr(addr)
        if info:
            results = [info.get('country_name'), info.get('city'),
                       info.get('latitude'), info.get('longitude')]
    except pygeoip.GeoIPError:
        pass

    return results

results = log.addr.map(get_location)

new_df = pd.DataFrame(results.values.tolist(),
                      columns=['country', 'city', 'latitude', 'longitude'])
print(new_df.head())

```

map() is similar to apply() except for Series

	country	city	latitude	longitude
0	Russian Federation	Chapaevsk	52.9781	49.7197
1	Russian Federation	Chapaevsk	52.9781	49.7197
2	Russian Federation	Kursk	51.8830	36.2659
3	Russian Federation	Kursk	51.8830	36.2659
4	Russian Federation	Moscow	55.7522	37.6156

ch05\_more\_pandas/20\_reading\_a\_log\_file.ipynb

Explanation of code:

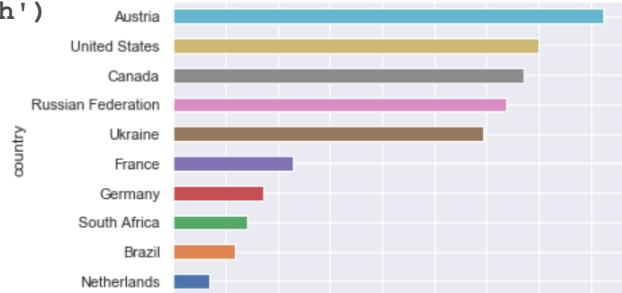
For fun, we've brought in a 3<sup>rd</sup> party tool called pygeoip. It is used to read a special file containing information about IP addresses. Given an IP address (as shown at the top of the slide) a dictionary of results can be returned (shown at the top right). By using the map() function, we call the get\_location() function for every value in the address (addr) column. The function returns a list of values each time. Those values include the country name, city, latitude and longitude. The results are used to create a new DataFrame (called new\_df).

# Merging the Two DataFrames

```
merged = pd.concat([log, new_df])
print(merged.head())
```

	addr	browser	city	country	http-method	latitude	longitude	req_date	request	size	status
0	109.169.248.247	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	GET	NaN	NaN	2015-12-12 18:25:11	GET /administrator/ HTTP/1.1	4263	200.0
1	109.169.248.247	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	POST	NaN	NaN	2015-12-12 18:25:11	POST /administrator/index.php HTTP/1.1	4494	200.0
2	46.72.177.4	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	GET	NaN	NaN	2015-12-12 18:31:08	GET /administrator/ HTTP/1.1	4263	200.0
3	46.72.177.4	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	POST	NaN	NaN	2015-12-12 18:31:08	POST /administrator/index.php HTTP/1.1	4494	200.0
4	83.167.113.100	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	GET	NaN	NaN	2015-12-12 18:31:25	GET /administrator/ HTTP/1.1	4263	200.0

```
top10_countries =
    merged.groupby('country').size().sort_values(ascending=False).head(10)
top10_countries[::-1].plot(kind='barh')
```



ch05\_more\_pandas/20\_reading\_a\_log\_file.ipynb

Explanation of code:

We took our log file DataFrame and concatenated it (along the column direction) with the new DataFrame data obtained on the previous slide. We then grouped results by country to see which countries made the most requests.

# Final Results of Log File Analysis

```
top10_cities =
    merged.groupby('city').size().sort_values(ascending=False).head(10)
top10_cities[::-1].plot(kind='barh');

from gmplot import GoogleMapPlotter
lats, longs = [], []
for city in top10_cities.index:
    latitude = merged.loc[city == merged['city'], 'latitude']
    longitude = merged.loc[city == merged['city'], 'longitude']
    lats.append(latitude.iloc[0])
    longs.append(longitude.iloc[0])

g_map = GoogleMapPlotter(37, 95, 3)
g_map.heatmap(lats, longs, radius=20)
g_map.draw('results.html')
```

City	Count
Montréal	~68,000
Alexandria	~35,000
Vienna	~28,000
Lviv	~22,000
Buffalo	~15,000
Moscow	~10,000
Graz	~8,000
Quito	~5,000
Johannesburg	~4,000
Centurion	~3,000

ch05\_more\_pandas/20\_reading\_a\_log\_file.ipynb

Explanation of code:

We repeated our groupings one last time, this time by city and the results are shown in the upper right. For fun, a 3<sup>rd</sup> party tool, called gmplot, was pip installed and used to make a heatmap of the latitude and longitude values of our top 10 cities. The generated HTML file can be found in the chapter 5 folder.

# Binning

- Binning is the conversion of *continuous* (or many categorical) values into smaller sets of categorical values

```
df = pd.DataFrame(data,
                   columns = ['gender', 'age', 'state'])

bins = [0, 18, 25, 35, 45, 55, 150]
df['age_group'] = pd.cut(df['age'], bins=bins)

labels = ['kid', 'early adult', 'young adult',
          'middle adult', 'older adult', 'senior']
df['age_name'] = pd.cut(df['age'], bins=bins, labels=labels)
print(df)
```

	gender	age	state	age_group	age_name
0	M	33	California	(25, 35]	young adult
1	M	55	Florida	(45, 55]	older adult
2	F	44	Maine	(35, 45]	middle adult
3	F	43	Idaho	(35, 45]	middle adult
4	F	64	Alaska	(55, 150]	senior
5	F	49	Ohio	(45, 55]	older adult
6	F	13	New York	(0, 18]	kid
7	M	37	California	(35, 45]	middle adult
8	M	61	Texas	(55, 150]	senior
9	M	27	Washington	(25, 35]	young adult
10	F	22	Florida	(18, 25]	early adult
11	M	55	New Jersey	(45, 55]	older adult
12	F	18	Nevada	(0, 18]	kid
13	M	27	Oregon	(25, 35]	young adult
14	F	26	Arizona	(25, 35]	young adult
15	M	21	Utah	(18, 25]	early adult
16	F	19	Oregon	(18, 25]	early adult
17	M	67	Colorado	(55, 150]	senior

ch05\_more\_pandas/25\_binning.py

In the example, we actually "binned" our age column twice. In the first attempt, we created a new column called age\_group. The second example used our own custom labels for the bin names and created a column called age\_name.

Not shown on the slide is our data. Here it is:

```
data = [('M', 33, 'California'), ('M', 55, 'Florida'), ('F', 44, 'Maine'),
        ('F', 43, 'Idaho'), ('F', 64, 'Alaska'), ('F', 49, 'Ohio'),
        ('F', 13, 'New York'), ('M', 37, 'California'), ('M', 61, 'Texas'),
        ('M', 27, 'Washington'), ('F', 22, 'Florida'), ('M', 55, 'New Jersey'),
        ('F', 18, 'Nevada'), ('M', 27, 'Oregon'), ('F', 26, 'Arizona'),
        ('M', 21, 'Utah'), ('F', 19, 'Oregon'), ('M', 67, 'Colorado')]
```

# Your Turn! - Task 5-2

## Pandas Groupby

Using Pandas to read Batting.csv, answer the two following questions:

1. Which team has hit the most home runs (cumulative)?
2. Which team hit the most home runs in 2015?

Then, plot the total home runs hit per decade to see which decade had the most home runs

	playerID	yearID	stint	teamID	lgID	G	AB	R	H	2B	...	RBI	SB	CS	BB	SO	IBI
0	abercda01	1871	1	TRO	Nan	1	4.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	Nal
1	addybo01	1871	1	RC1	NaN	25	118.0	30.0	32.0	6.0	...	13.0	8.0	1.0	4.0	0.0	Nal
2	allisar01	1871	1	CL1	NaN	29	137.0	28.0	40.0	4.0	...	19.0	3.0	1.0	2.0	5.0	Nal
3	allisdo01	1871	1	WS3	NaN	27	133.0	28.0	44.0	10.0	...	27.0	1.0	1.0	0.0	2.0	Nal
4	ansonca01	1871	1	RC1	NaN	25	120.0	29.0	39.0	11.0	...	16.0	6.0	2.0	2.0	1.0	Nal

Work from the task5\_2\_starter.py, or copy its contents into a new Notebook file if you prefer to work within Jupyter.

# Time Series and DataFrames

- Time Series and DataFrames are structures whose index is time-based

```
log = pd.read_csv('../resources/new_access.log', sep='\s+',
                  usecols=(0, 3, 5, 6, 7, 9),
                  names=['addr', 'req_date', 'request', 'status', 'size', 'browser'],
                  error_bad_lines=False)
log.req_date = pd.to_datetime(log.req_date, format='[%d/%b/%Y:%H:%M:%S]')
date_based_log = log.set_index('req_date') ←
print(date_based_log.info())
print(date_based_log.head())
```

Notice we made our `req_date` column an index now

	addr	request	status	size	browser
<b>req_date</b>					
2015-12-12 18:25:11	109.169.248.247	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:25:11	109.169.248.247	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:31:08	46.72.177.4	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:31:08	46.72.177.4	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:31:25	83.167.113.100	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...

## ch05\_more\_pandas/26\_time\_series.py

In this example, we have taken our original log file DataFrame and converted the `req_date` column (a datetime object) into the index of the DataFrame. This creates a special DataFrame sometimes referred to as a time series because its index is now based on time.

# Using the Time Series

- Time Series has special capabilities for matching values:

```
print(date_based_log['2015-12-12'].shape)
```

(358, 5)

Gives back just records that fall on this date

```
print(date_based_log['2015'].shape)
```

(14148, 5)

Just records from that year

```
print(date_based_log['2016-01'].shape)
```

(28224, 5)

Just records from that month

```
print(date_based_log['2015-12-12 18:00:00':'2015-12-12 18:30:00'])
```

req_date	addr	request	status	size	browser
2015-12-12 18:25:11	109.169.248.247	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:25:11	109.169.248.247	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...

Just records within the given 30 minute range

ch05\_more\_pandas/26\_time\_series.py



# Reading Excel Spreadsheet Files

- `read_excel()` can read a Microsoft Excel spreadsheet table into a DataFrame

```
pd.read_excel(io, sheetname=0, header=0, skiprows=None, skip_footer=0,  
             index_col=None, names=None, na_values=None,  
             convert_float=True, thousands=None, converters=None)
```

<code>io</code> -	URL or path to an Excel-compatible file
<code>sheetname</code> -	string name or int for the sheet to read
<code>header</code> -	row to use for the column labels
<code>skiprows</code> -	rows to skip at the beginning
<code>skip_footer</code> -	rows to skip at the end
<code>index_col</code> -	column to use as the column labels of the DataFrame
<code>names</code> -	list of column names to use
<code>converters</code> -	dict of functions for converting values in columns
<code>thousands</code> -	thousands separator for parsing string columns to numeric
<code>convert_float</code> -	convert floats to int
<code>na_values</code> -	strings to use in place of NaN or NA values

# Using read\_excel()

temperatures.xlsx

	High	Low		High	Low		Humidity
Colorado Springs	78	50	Colorado Springs	77	48	Colorado Springs	22
Canon City	82	52	Canon City	81	49	Canon City	18
Pueblo	83	53	Pueblo	84	50	Pueblo	19
						Denver	25

```
sat_temps      = pd.read_excel('temperatures.xlsx', sheetname='Saturday')
sun_temps      = pd.read_excel('temperatures.xlsx', sheetname=1)
sat_humidity = pd.read_excel('temperatures.xlsx', sheetname='Humidity')

sat_merged = pd.concat([sat_temps, sat_humidity], axis=1)
merged = pd.concat([sat_merged, sun_temps], keys=['Saturday', 'Sunday'])
```

			High	Humidity	Low
Saturday	Canon City		82.0	18.0	52.0
	Colorado Springs		78.0	22.0	50.0
	Denver		NaN	25.0	NaN
Sunday	Pueblo		83.0	19.0	53.0
	Colorado Springs		77.0	NaN	48.0
	Canon City		81.0	NaN	49.0
	Pueblo		84.0	NaN	50.0

ch05\_pandas/15\_apply.py

In the above example, three sheets can be found in temperatures.xlsx. Each is read into different DataFrames and then concatenated into the resulting DataFrame shown in the lower right.

The results of this example are continued on the next page.

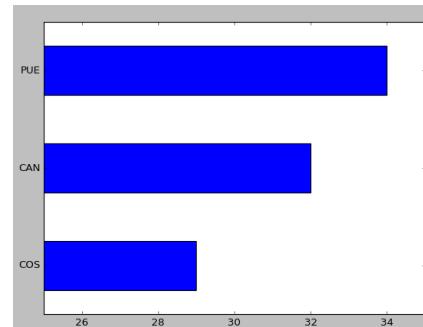
# Using apply()

- **apply()** can execute a function for each row or column in a DataFrame

merged

		High	Humidity	Low
Saturday	Canon City	82.0	18.0	52.0
	Colorado Springs	78.0	22.0	50.0
	Denver	NaN	25.0	NaN
	Pueblo	83.0	19.0	53.0
Sunday	Colorado Springs	77.0	NaN	48.0
	Canon City	81.0	NaN	49.0
	Pueblo	84.0	NaN	50.0

Specify **axis=0** to apply the function to **each column** and **axis=1** to apply the function to **each row**.



```
def get_temp_delta(row):
    return row['High'] - row['Low']

temperature_deltas = merged.apply(get_temp_delta, axis=1)

bar_chart = temperature_deltas['Sunday'].plot(kind='barh')
bar_chart.set_xlim(25, 35)
bar_chart.set_yticklabels(['COS', 'CAN', 'PUE'])
plt.show()
```

ch05\_pandas/15\_apply.py

After reading the data from the Excel spreadsheet on the previous slide, we can apply a function that calculates the high-low temperature delta for each row. It is important to specify the `axis=1` attribute in the `apply()` call or else columns will be selected instead.

# Analysis: Black Friday vs Cyber Monday

- We'll use Pandas `read_html()` to compare and analyze data related to Black Friday sales vs Cyber Monday sales:

```
url1 = 'https://en.wikipedia.org/wiki/Black_Friday_(shopping)'
url2 = 'https://en.wikipedia.org/wiki/Cyber_Monday'
```

```
bfriday = pd.read_html(url1, header=0)[1]
cyber = pd.read_html(url2, header=0)[1]
```

`print(bfriday)`

	Year	Date	Survey published	Shoppers (millions)	Average spent	Total spent	Consumers polled	Margin for error
0	2020	Nov 27	NaN	NaN	NaN	NaN	NaN	NaN
1	2019	Nov 29	NaN	NaN	NaN	NaN	NaN	NaN
2	2018	Nov 23	NaN	NaN	NaN	NaN	NaN	NaN
3	2017	Nov 24	Nov 28[130]	174.0	\$335.47	\$58.3 billion	3242.0	+/- 1.7%
4	2016	Nov 25	NaN	NaN	NaN	NaN	NaN	NaN
5	2015	Nov 27	NaN	NaN	NaN	NaN	NaN	NaN
6	2014[131]	Nov 28	Nov 30	233.0	\$380.95	\$50.9 billion	4631.0	1.5%
7	2013	Nov 29	Dec 1	249.0	\$407.02	\$57.4 billion	4864.0	1.7%
8	2012	Nov 23	Nov 25	247.0	\$423.66	\$59.1 billion	4005.0	1.6%
9	2011	Nov 25	Nov 27	226.0	\$398.62	\$52.5 billion	3826.0	1.6%
10	2010	Nov 26	Nov 28	212.0	\$365.34	\$45.0 billion	4306.0	1.5%
11	2009	Nov 26	Nov 29	195.0	\$343.31	\$41.2 billion	4985.0	1.4%
12	2008	Nov 28	Nov 30	172.0	\$372.57	\$41.0 billion	3370.0	1.7%
13	2007	Nov 23	Nov 25	147.0	\$347.55	\$34.6 billion	2395.0	1.5%
14	2006	Nov 24	Nov 26	140.0	\$360.15	\$34.4 billion	3090.0	1.5%
15	2005	Nov 25	Nov 27	132.0	\$301.81	\$26.8 billion	NaN	NaN

We took the second table on the page and used row 0 for the column names

This data requires LOTS of cleaning!

ch05\_more\_pandas/27\_Black\_Friday\_vs\_Cyber\_Monday.ipynb

In the following example, we will bring in data from two Wikipedia sources: one capturing information from Cyber Monday sales and the other capturing sales data for Black Friday.

# Cleaning the Black Friday Table

```
print(bfriday.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16 entries, 0 to 15
Data columns (total 8 columns):
Year           16 non-null object
Date           16 non-null object
Survey published 11 non-null object
Shoppers (millions) 11 non-null float64
Average spent   11 non-null object
Total spent     11 non-null object
Consumers polled 10 non-null float64
Margin for error 10 non-null object
dtypes: float64(2), object(6)
memory usage: 1.1+ KB
```

Drop unneeded columns and rows with NaN values in the remaining columns

```
bfriday = bfriday.drop(['Margin for error', 'Survey published', 'Date',
                       'Consumers polled'], axis=1).dropna()

bfriday.Year = bfriday.Year.map(lambda yr: str(yr).split('[')[0]).astype(int)

bfriday['Average spent'] = bfriday['Average spent'].str.lstrip('$').astype(float)

bfriday['Total spent (billions)'] = bfriday['Total spent'].str.lstrip('$')

.bstr.strip('billion').astype(float)

bfriday.drop(['Total spent'], axis=1, inplace=True)
```

Remove the footnotes and \$ and typed our columns

Removed the 'billion' word

ch05\_more\_pandas/27\_Black\_Friday\_vs\_Cyber\_Monday.ipynb

Get in the habit of analyzing the initial data structure. Use info() to see what you are working with. We removed unneeded columns and NaN rows from the remaining DataFrame.



## Black Friday DataFrame Cleaned Up

	Year	Shoppers (millions)	Average spent	Total spent (billions)
3	2017	174.0	335.47	58.3
6	2014	233.0	380.95	50.9
7	2013	249.0	407.02	57.4
8	2012	247.0	423.66	59.1
9	2011	226.0	398.62	52.5
10	2010	212.0	365.34	45.0
11	2009	195.0	343.31	41.2
12	2008	172.0	372.57	41.0
13	2007	147.0	347.55	34.6
14	2006	140.0	360.15	34.4
15	2005	132.0	301.81	26.8

ch05\_more\_pandas/27\_Black\_Friday\_vs\_Cyber\_Monday.ipynb

After the cleaning step, here's our new Black Friday table.

# Cleaning the Cyber Monday Table

Don't need all these columns

	Day	Year	Sales(millionsof USD)	% Change
0	November 27	2006	\$610	NaN
1	November 26	2007	\$730	20%
2	December 1	2008	\$887	Need to remove \$, commas, footnotes
3	November 30	2009	\$887	4.7%
4	November 29	2010	\$1,028	16%
5	November 28	2011	\$1,251	22%
6	November 26	2012	\$1,465	17%
7	December 2	2013	\$1,735	18%
8	December 1	2014	\$2,038[23]	17%
9	November 30	2015	\$2,280	12%
10	November 28	2016	\$2,671	17%
11	November 27	2017	\$3,364	26%
12	November 26	2018	\$7,900	19.3%

ch05\_more\_pandas/27\_Black\_Friday\_vs\_Cyber\_Monday.ipynb

The Cyber Monday table is equally riddled with data cleaning issues.



# Cleaning the Cyber Monday DataFrame

```

cyber.drop(['Day', '% Change'], axis=1, inplace=True) Drop unneeded columns
cyber.dropna(inplace=True)
cyber = cyber.rename({'Sales(millionsof USD)': 'Cyber Mon. (millions)'},
                     axis=1) Rename other columns

cyber['Cyber Mon. (millions)'] =
    cyber['Cyber Mon. (millions)'].str.lstrip('$')
    .str.replace(',', '')
    .map(lambda val: str(val).split('[')[0])
    .astype(int)

Year  Cyber Mon. (millions)
0    2006          610
1    2007          730
2    2008          846
3    2009          887
4    2010         1028
5    2011         1251
6    2012         1465
7    2013         1735
8    2014         2038 Cleaned Up Version
9    2015         2280
10   2016         2671
11   2017         3364
12   2018         7900

```

Remove unwanted \$ signs, replace commas with empty strings, and remove the footnotes

ch05\_more\_pandas/27\_Black\_Friday\_vs\_Cyber\_Monday.ipynb

In a similar fashion, we cleaned the Cyber Monday Table. Time to bring the two tables together...

# Merging the DataFrames

```
merged = pd.merge(bfriday, cyber, on='Year').sort_values(by='Year')
```

```
merged.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10 entries, 9 to 0
Data columns (total 5 columns):
Year          10 non-null int32
Shoppers (millions) 10 non-null float64
Average spent   10 non-null float64
Total spent (billions) 10 non-null float64
Cyber Mon. (millions) 10 non-null int32
dtypes: float64(3), int32(2)
memory usage: 400.0 bytes
```

Provides another way to see if there are any NaN values

```
pd.isnull(merged).sum()
```

Year	0
Shoppers (millions)	0
Average spent	0
Total spent (billions)	0
Cyber Mon. (millions)	0
dtype: int64	

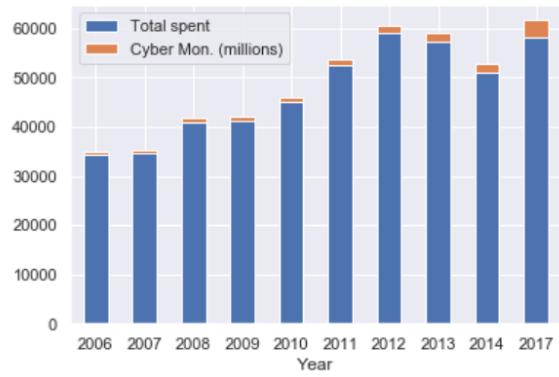
ch05\_more\_pandas/27\_Black\_Friday\_vs\_Cyber\_Monday.ipynb

The two DataFrames are merged on the Year columns. We can see there are no NaN values in any of these columns.

# Final Results: Black Friday vs Cyber Monday

```
merged['Total spent'] = merged['Total spent (billions)'] * 1000
merged.plot(kind='bar', x='Year',
            y=['Total spent', 'Cyber Mon. (millions)'], rot=0, stacked=True)
```

Year	Shoppers (millions)	Average spent	Total spent (billions)	Cyber Mon. (millions)
9 2006	140.0	360.15	34.4	610
8 2007	147.0	347.55	34.6	730
7 2008	172.0	372.57	41.0	846
6 2009	195.0	343.31	41.2	887
5 2010	212.0	365.34	45.0	1028
4 2011	226.0	398.62	52.5	1251
3 2012	247.0	423.66	59.1	1465
2 2013	249.0	407.02	57.4	1735
1 2014	233.0	380.95	50.9	2038
0 2017	174.0	335.47	58.3	3364



ch05\_more\_pandas/27\_Black\_Friday\_vs\_Cyber\_Monday.ipynb

Our final results are plotted as a bar chart and illustrate that Cyber Monday is only a fraction of what Black Friday sales amount to.

# Using read\_json()

- Pandas `read_json()` supports reading JSON-based data even in various formats:

```
import pandas as pd
data = '''
[
    {"name": "Fang", "type": "Dog", "age": 3},
    {"name": "Aragog", "type": "Spider", "age": 1},
    {"name": "Hedwig", "type": "Owl", "age": 2}
]
'''
```

`df = pd.read_json(data, orient='records')`

Different values for orient are used for different JSON structures: `records`, `split`, `index`, `columns`, and `values`

	age	name	type
0	3	Fang	Dog
1	1	Aragog	Spider
2	2	Hedwig	Owl

For data that doesn't match any of these formats, use Pandas' `json_normalize()` method. An example of retrieving live JSON data and using `json_normalize()` can be found in [14\\_normalizing\\_json\\_data.ipynb](#)

ch05\_pandas/17\_read\_json.py

Refer to the example within the student files to view how the value of the orient attribute should change based on different JSON formats.

# Pandas Idioms

```
df = pd.DataFrame(data=np.arange(1, 10).reshape((3, 3)),
                   columns=['first', 'second', 'third'])

   first  second  third
0      1       2       3
1      4       5       6
2      7       8       9
```

- Split DataFrame based on a criterion

```
criterion = (df['first'] < 4)
df_one = df[criterion]
df_two = df[~criterion]
```

	first	second	third
0	1	2	3
1	4	5	6
2	7	8	9

- Assign values to a column based on another column

```
df.loc[df['second'] >= 5, 'third'] = -1
```

	first	second	third
0	1	2	3
1	4	5	-1
2	7	8	-1

ch05\_pandas/21\_idioms.py

The bottom example is the Pandas version of an if-then.

# Pandas Idioms (*continued*)

```
df = pd.DataFrame(data=np.arange(1, 10).reshape((3, 3)),
                   columns=['first', 'second', 'third'])

   first  second  third
0      1       2       3
1      4       5       6
2      7       8       9
```

- Membership

```
df[df.index.isin([0, 1]) & df['second'].isin([1, 2, 3])]
```

	first	second	third
0	1	2	3

- Complement

```
df[~(df.index.isin([0, 1]) & df['second'].isin([1, 2, 3]))]
```

	first	second	third
1	4	5	6
2	7	8	9

ch05\_pandas/21\_idioms.py

When determining membership, any iterable, dictionary, Series or DataFrame may be used as an argument to `isin()`.

# Pandas Idioms (continued)

- Find first of each group

```
data = [
    ['Dick Cabbage', 'Tulsa'],
    ['Tina Turnip', 'Oklahoma City'],
    ['Elvis Parsley', 'Tulsa'],
    ['Antonio Banana', 'Norman'],
    ['Howie Mango', 'Oklahoma City'],
    ['Tom Shanks', 'Norman'],
]
df2 = pd.DataFrame(data=data, columns=['name', 'city'])
city_groups = df2.groupby('city')
print(city_groups.first().reset_index())
```

Use `city_groups.nth(1)` to get the second item of each group



	city	name
0	Norman	Antonio Banana
1	Oklahoma City	Tina Turnip
2	Tulsa	Dick Cabbage

ch05\_pandas/21\_idioms.py

The `reset_index()` in this example makes it so that the city column is NOT the index, but rather a column within the data values section of the DataFrame.



# Notes on Pandas and Database Access

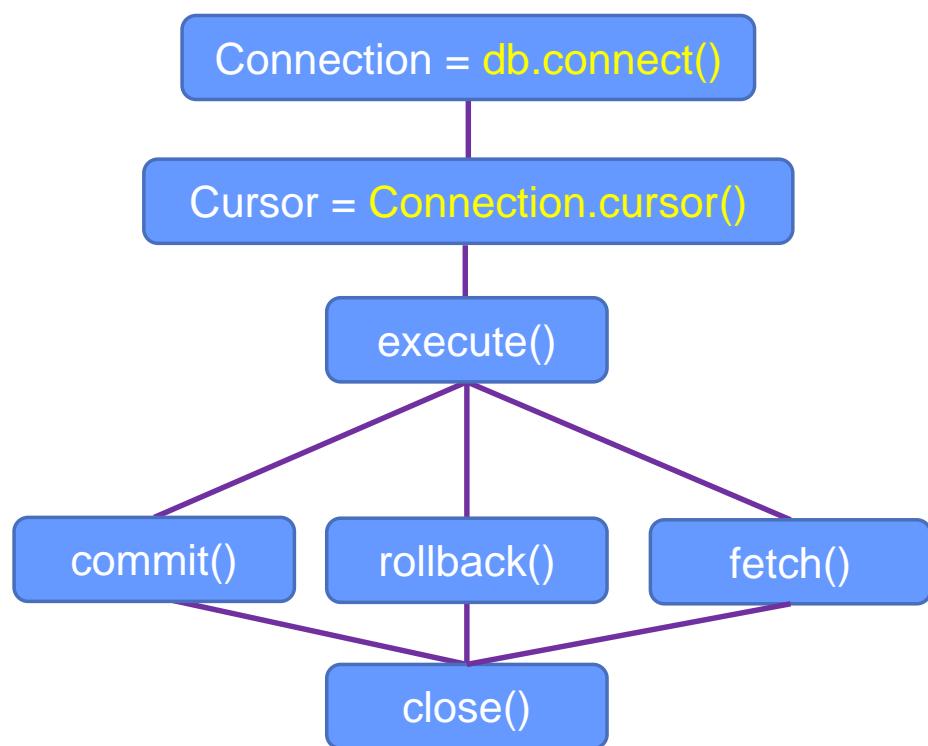
- The Python DB API 2.0 defines a common interface for modules to connect to and work with relational databases
- The interface defines:
  - Connection Objects and Transactions
  - Cursors Object Operations
  - Input, Output Data Types
  - Error Handling
  - Two-Phase Commits

There is no standard library module. Vendors or 3<sup>rd</sup> parties must provide the needed module

The Python Database API 2.0 is Python's version of Java's JDBC. It defines the interface for working with a driver that communicates with a specific database.



# A Typical Database Interaction





# Cursors

- **Cursors** are used for fetch and execution
  - Multiple cursors made from the same connection are not isolated
  - Cursors support the following methods:

`fetchone()`  
`fetchmany(size=cursor.arraysize)`  
`fetchall()`  
`execute(sql, params)`  
`executemany(sql, [params])`  
`callproc(name, params)`

`nextset()`  
`setinputsizes(sizes)`  
`setoutputsizes(size, col)`  
`rowcount`  
`description`  
`close()`

**Atomicity** - all parts of a transaction succeed or fail together

**Consistency** - the database will always be consistent

**Isolation** - no transactions can interfere with another

**Durability** - once the transaction is committed, it won't be lost

# Execute Methods

- `execute(sql, params)` – executes the provided sql

```
data = ('Bob', 100.0, 0.05, 'C')
cursor.execute(
    "INSERT INTO accounts (name, balance, rate, acct_type) VALUES \
    (?, ?, ?, ?)", data)
```

- `executemany(sql, [params])` – execute sql repeatedly against all supplied params

```
data = [('John Smith', 5500.0, 0.025, 'C'),
        ('Sally Jones', 6710.11, 0.025, 'C'),
        ('Fred Green', 2201.73, 0.035, 'S'),
        ('Ollie Engle', 187.30, 0.025, 'S'),
        ('Gomer Pyle', 12723.10, 0.015, 'C')]
cursor.executemany("INSERT INTO accounts \
    (name, balance, rate, acct_type) VALUES \
    (?, ?, ?, ?)", data)
```



## SQLite

- **SQLite** is an in-process relational database
  - Don't have to start a separate application to run it
  - Runs entirely within your current Python app
  - To use it import the module `sqlite3`
  - Can be an in-memory or file-based database
  - Not typically used in large-scale production, but useful for development, testing, smaller efforts
- Connect using the `sqlite.connect()` method

```
import sqlite3
connection = sqlite3.connect('mysqlite.db')
```

Note: to work with the database as an in-memory database, connect using the string ':memory:'.

# Using 'with' in SQLite3

```

try:
    with sqlite3.connect('schools.db') as connection:
        cursor = connection.cursor()
        cursor.execute(DROP_SCHOOLS_SQL)
        cursor.execute(CREATE_SCHOOLS_SQL)
        cursor.executemany(INSERT_RECORD, school_data)
        print('Data loaded into schools table')
except sqlite3.Error as err:
    print('Data not loaded into schools table')
    print('Error: {}'.format(err))
finally:
    if connection:
        connection.close()

```

The `with` control can be used on the connection object. It will `commit()` if no errors occur otherwise it will automatically `rollback()` if an error occurs

The connection is not automatically closed by the with control

ch05\_pandas/28\_creating\_schools\_db\_using\_with.py

The `with` control when using `sqlite3` can be used to automatically commit or rollback the transaction depending on whether an error occurs internally or not. Worth noting however, the connection is not closed by the `with` control.

# Accessing Data

```
import sqlite3

school_data = []
connection = None
state = input('Schools from which state: ').upper()
try:
    connection = sqlite3.connect('schools.db')
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM schools WHERE state=?', (state,))
    for sch in cursor:
        school_data.append((sch[0], sch[1], sch[2], sch[3],
                           sch[4]))
except sqlite3.Error as e:
    print('Error: {}'.format(e))
finally:
    if connection:
        connection.close()
```

or append(sch)

ch07\_database/02\_fetching\_rows.py

This solution reads back records from the database into a tuple. Once the cursor is executed the rows can be retrieved with fetch methods or via the cursor directly.



# Pandas and the Database

- Use `pd.read_sql(sql, conn, params)` to read data directly from a database into a DataFrame
- Arguments to `read_sql()` include:
  - `sql` - the sql statement to perform
  - `conn` - the already created connection to the database
  - `index_col` - the db column to use for the DataFrame index
  - `params` - list of parameters to use
  - `columns` - list of column names to use from the table

Note: a `read_sql_table()` method also exists which reads all records from a table.



# Pandas Database Example

```

import pandas as pd
import sqlite3

with sqlite3.connect('batting.db') as conn:
    df = pd.read_sql('SELECT hits, atbats FROM batting WHERE year >= ?',
                      conn, params=['1957'])

df = df[df.hits != 0]
df = df[df.atbats >= 502]           ← Removes undesired data rows

avgs = df['hits'] / df['atbats']
df.loc[:, 'averages'] = pd.Series(avgs, index=df.index)
print(df.describe())
print('50% batting average: {0}'           ← Add the new
      .format(df.describe()['averages']['50%']))

```

Loads up selected query  
data into a DataFrame

Removes undesired data rows

Add the new  
averages column  
to the DataFrame

ch07\_database/06\_pandas\_sql\_batting.py

The example is similar to the earlier tasks except that we are now reading from a database directly into a DataFrame. In addition, zero values for hits and atbats are removed while only atbats  $\geq 502$  are kept.



# Summary

- Merging Pandas DataFrames can be performed in a similar way to operating on tables within a database
- Pandas provides great data access methods to acquire data from external sources:
  - These include:
    - `read_csv()`
    - `read_json()`
    - `read_html()`
    - `read_sql()`
    - Other methods also

## Task 5-3 - Pandas and the Database

- Using *batting.db*, determine the highest paid baseball player overall and the year played

Salaries table

year	team	league	playerid	salary

Players table

playerid	firstname	lastname

- Hints:

- Read data from batting.db (a sqlite3 database)
- Use `df['salary'].idxmax()` to determine the row for the max salary
- Use the row for the max salary to get the playerid and year
  - Consider using `df['playerid'][idxmax]` and `df['year'][idxmax]`
- Using the playerid, query the Players table to get the firstname and lastname

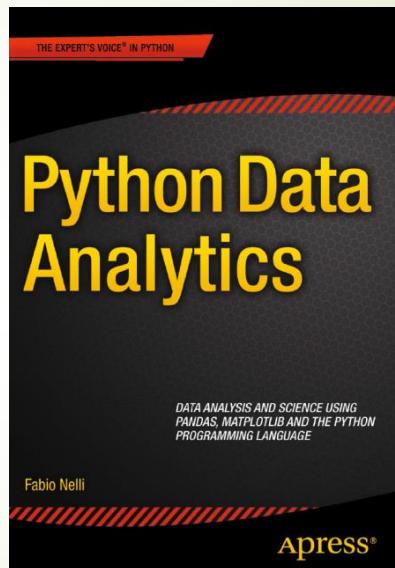
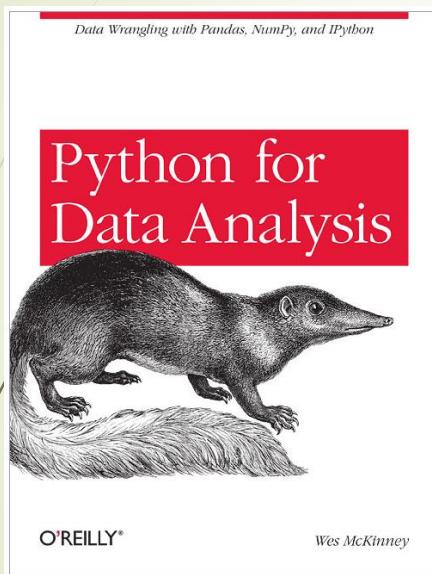


# Course Summary

# What did we learn ?

- 
- Data Types
  - Operations of Sequences
  - Slicing
  - Strings and Formatting Values
  - Comments and Docstrings
  - Importing and Managing Modules
  - Control Structures
  - Advanced Iteration Techniques
  - List Comprehensions
  - Dictionary Manipulation
  - Exception Handling
  - Creating Classes
  - Creating, Working with Threads
  - IPython
  - Jupyter Notebooks
  - Data Analysis Libraries
  - Using NumPy, ndarray
  - Pandas Data Structures
  - Series
  - DataFrames
  - Sorting DataFrame Data
  - Grouping Data: Splitting,  
Applying, Combining
  - Matplotlib
  - Plotting Different Types
  - Python Database APIs
  - Pandas and Databases
  - Working with CSV and Excel Data  
Files
  - Indexing using loc[], iloc[]
  - MultilIndexing
  - Merging DataFrames
  - Other Data Analysis Tools

## Recommended Sources



100 NumPy Exercises to keep you tuned:

[http://www.labri.fr/perso/nrougier/teaching\(numpy.100/](http://www.labri.fr/perso/nrougier/teaching(numpy.100/)

# Evaluations

- ▶ Please take the time to fill out an evaluation
- ▶ All evaluations are read and considered

Thank you for your response and feedback which are critical to this process.

# Questions





# Appendix A

## Seaborn and

## Scikit-learn

Additional Data Visualization Tools and  
an Intro to Data Mining with Python



# Overview

Seaborn vs Matplotlib

Seaborn APIs

Introducing Scikit-learn



# Seaborn vs Matplotlib

- **Seaborn** is a Python data visualization library built upon Matplotlib
  - Works with NumPy and Pandas
  - Understands Pandas labels when defining axes
  - Provides additional features beyond Matplotlib such as:
    - Easier use of color palettes and themes
    - Visualizations for linear regressions
    - Additional data plot types
- Seaborn, like Matplotlib, must be installed:

`conda install seaborn` or `pip install seaborn`

  - If using Anaconda 4.3+, it will already be installed

Seaborn requires Matplotlib to be installed. Anaconda now ships with both installed already. Seaborn provides numerous feature enhancements over Matplotlib. When compared to Matplotlib, it creates more stylized charts with less coding required.



# Starting with Seaborn

- Seaborn is imported using:  
`import seaborn as sns`
- If using it within Jupyter, it will require:  
`%matplotlib inline`
- Use the `sns.set()` call to configure different styles  
`sns.set(context='notebook', style='darkgrid', palette='deep', ...)`
  - `sns.set()` by itself sets styles to the default values

The `set()` method allows for setting the look-and-feel of a visualization all at once. Individual methods allow for setting aesthetics one feature at a time (e.g., `sns.set_style()`, `sns.color_palette()`, and `sns.plotting_context()` for example).



# Retrieving Seaborn Datasets

- Seaborn has the ability to retrieve datasets from a github repository using `load_dataset()`
  - Returns a Pandas DataFrame

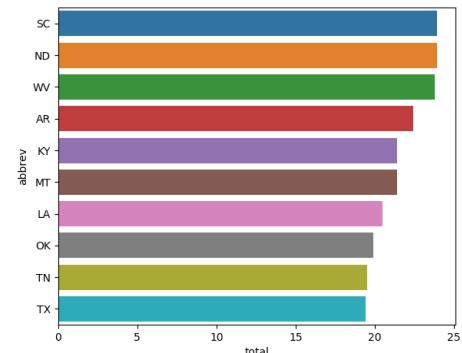
<https://github.com/mwaskom/seaborn-data>

<a href="#">anscombe.csv</a>	Add anscombe dataset	4 years ago
<a href="#">attention.csv</a>	Add attention dataset	4 years ago
<a href="#">brain_networks.csv</a>	Add brain networks dataset	4 years ago
<a href="#">car_crashes.csv</a>	Add 538 car crash dataset	3 years ago
<a href="#">dots.csv</a>	Add dots dataset	8 months ago
<a href="#">exercise.csv</a>	Add exercise dataset	4 years ago
<a href="#">flights.csv</a>	Add flights dataset	4 years ago
<a href="#">fmri.csv</a>	Change sorting of events in fmri data	7 months ago
<a href="#">gammas.csv</a>	Make fake fmri data make a bit more sense	4 years ago
<a href="#">iris.csv</a>	Add iris dataset	4 years ago
<a href="#">planets.csv</a>	Add planets dataset	4 years ago
<a href="#">tips.csv</a>	Add tips dataset	4 years ago
<a href="#">titanic.csv</a>	Update titanic dataset to remove index variable	4 years ago

# Plotting Seaborn Datasets

```
import matplotlib.pyplot as plt
import seaborn as sns

crashes = sns.load_dataset('car_crashes')
print(crashes.head())
print(crashes.shape)
most = crashes.sort_values(by='total', ascending=False).head(10)
sns.barplot(x='total', y='abbrev', data=most)
plt.show()
```



	total	speeding	alcohol	not_distracted	no_previous	ins_premium	ins_losses	abbrev
0	18.8	7.332	5.640	18.048	15.040	784.55	145.08	AL
1	18.1	7.421	4.525	16.290	17.014	1053.48	133.93	AK
2	18.6	6.510	5.208	15.624	17.856	899.47	110.35	AZ
3	22.4	4.032	5.824	21.056	21.280	827.34	142.39	AR
4	12.0	4.200	3.360	10.920	10.680	878.41	165.63	CA

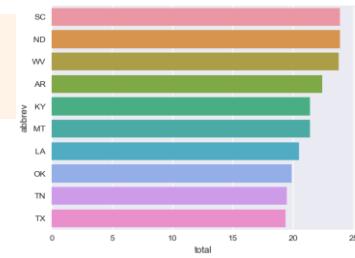
(51, 8)

ch06\_seaborn\_sklearn/01\_loading\_datasets.py

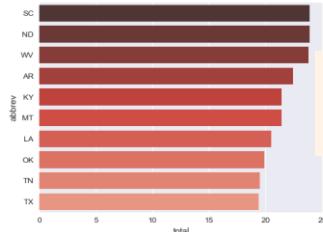
The example loads the `car_crashes.csv` file from GitHub. It displays the first 5 records and the shape (shown at the bottom on the slide). Finally, it sorts the DataFrame by total (crashes) and renders it as a bar plot.

# Different Seaborn Sets

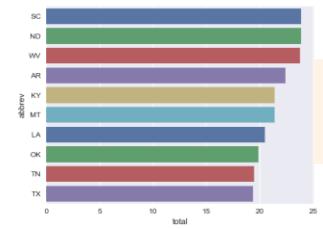
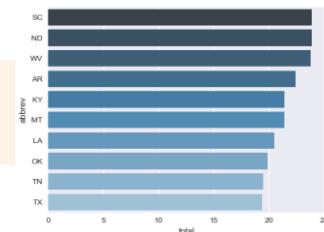
```
sns.set()
sns.barplot(x='total', y='abbrev', data=most)
plt.show()
```



```
sns.barplot(x='total', y='abbrev', data=most,
            palette='Reds_d')
plt.show()
```



```
sns.barplot(x='total', y='abbrev', data=most,
            palette='Blues_d')
plt.show()
```



```
sns.barplot(x='total', y='abbrev', data=most,
            palette='deep')
plt.show()
```

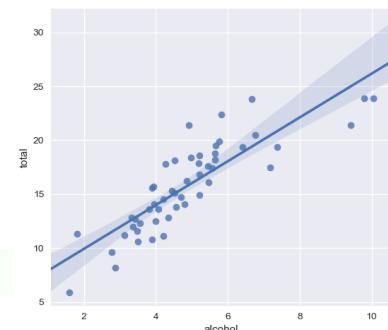
Different palettes can be specified to easily change the color scheme of any visualization. The variations of the default theme are 'deep', 'muted', 'dark', 'bright', 'colorblind', and 'pastel'. The 'deep' variation is shown at the bottom.

For more options on changing palettes, visit the Seaborn documentation at:  
[https://seaborn.pydata.org/tutorial/color\\_palettes.html](https://seaborn.pydata.org/tutorial/color_palettes.html).

# Seaborn Plotting

- Seaborn provides numerous plot functions
  - Most accept a DataFrame using the **data=** parameter
  - Provide **x=** and **y=** parameters to define axis data values

```
sns.lmplot(x='alcohol', y='total', data=crashes)
```



- Seaborn provides the following types of plots:

countplot()	violinplot()	boxplot()	pairplot()	factorplot()
Categorical			Axis	
pointplot()	stripplot()	barplot()	jointplot()	lmplot()
stripplot()	swarmplot()			
kdeplot()	rugplot()	heatmap()	regplot()	
Distribution		Matrix	Regression	
distplot()		clustermap()		residplot()

ch06\_seaborn\_sklearn/02\_seaborn\_syntax.py

The example shows one of the many Seaborn plot types (a linear regression, discussed in more detail in the next chapter). It illustrates how a DataFrame can be applied to the **data=** parameter while the names of its columns can be supplied to the **x=** and **y=** arguments.

# Understanding Your Data

- Seaborn can provide a fast visualization of our data to help get a better understanding of it
  - We can plot multiple **features** against our chosen **response** (we are using **total**)

	Our Response Variable		Our Features							
0	<b>total</b>	speeding	alcohol	not_distracted	no_previous	ins_premium	ins_losses	abbrev		
1	18.8	7.332	5.640	18.048	15.040	784.55	145.08	AL		
2	18.1	7.421	4.525	16.290	17.014	1053.48	133.93	AK		
3	18.6	6.510	5.208	15.624	17.856	899.47	110.35	AZ		
4	22.4	4.032	5.824	21.056	21.280	827.34	142.39	AR		
5	12.0	4.200	3.360	10.920	10.680	878.41	165.63	CA		

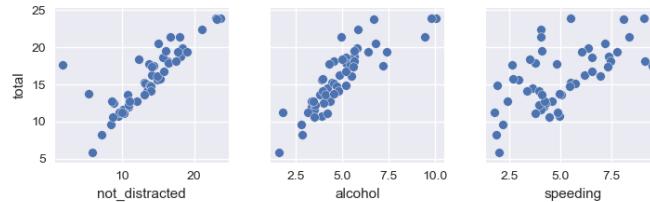
Because **total** is continuous, we can use a regression...

A regression is a machine learning model that assumes a continuous response value (as opposed to a categorical response).

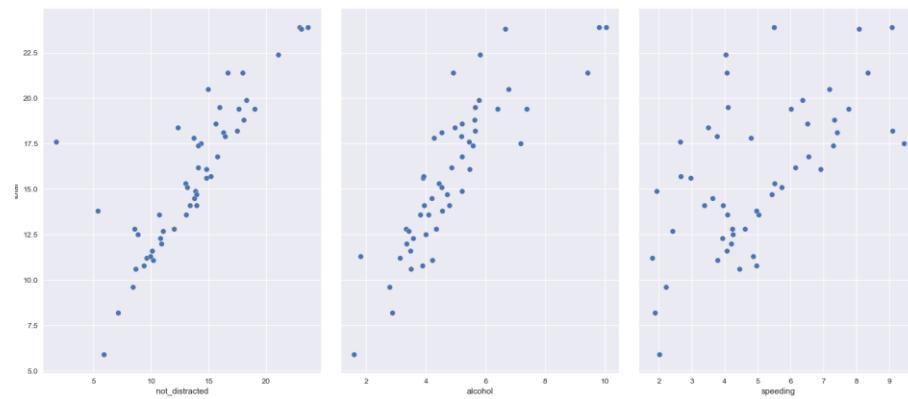
The total number of crashes, or the total column, will represent our response variable. For simplicity, we'll ignore the other columns for now. We want to predict total crashes, the response, based on the values of other features.

# Using pairplot() to Understand Features

```
sns.pairplot(crashes, x_vars=['not_distracted', 'alcohol', 'speeding'], y_vars='total')
```



```
sns.pairplot(crashes, x_vars=['not_distracted', 'alcohol', 'speeding'], y_vars='total', aspect=0.75, size=7.5)
```

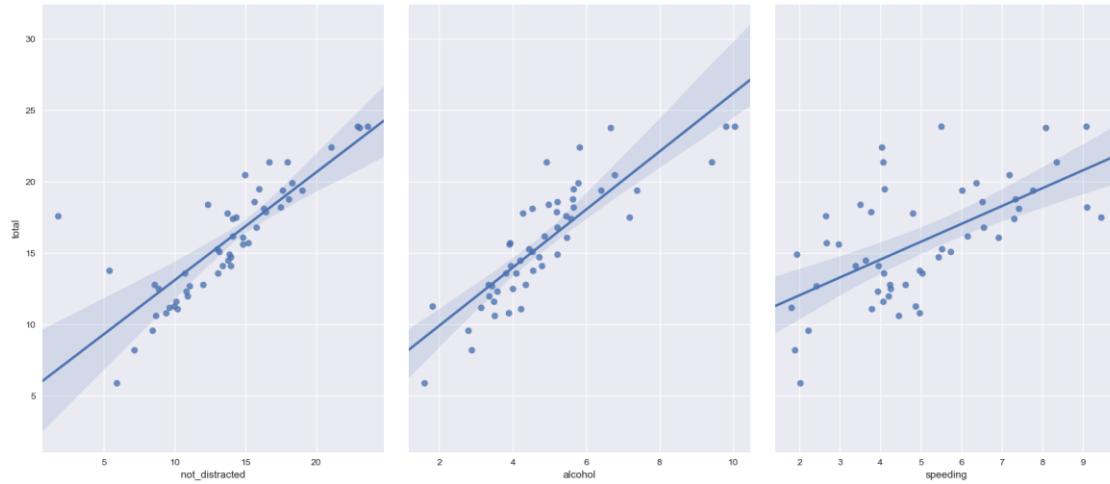


ch06\_seaborn\_sklearn/03\_pairplot.py

Using `pairplot()`, we plot the features: `not_distracted`, `alcohol`, and `speeding` against total crashes. The results can be made larger by adjusting the aspect and size values. From the results above, we can see there appears to be somewhat of a linear relationship between not being distracted and the total number of crashes. The relationship is less strong between alcohol-related crashes and total crashes, and finally a weaker relationship exists between speeding and total crashes.

## Adding a Linear Model and Confidence Interval

```
sns.pairplot(crashes, x_vars=['not_distracted', 'alcohol', 'speeding'], y_vars='total', kind='reg')
```



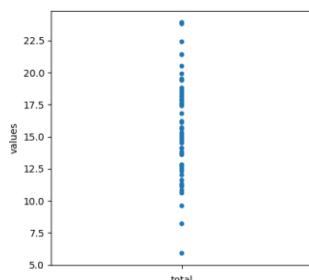
ch06\_seaborn\_sklearn/03\_pairplot.py

Values for the kind parameter can be "scatter" or "reg". With the "reg" kind, Seaborn adds a linear "best fit" and 95% confidence band. From the visualizations, we can infer that a linear regression might make sense to help predict values (refer to the Scikit-learn chapter for a continuation of this discussion).

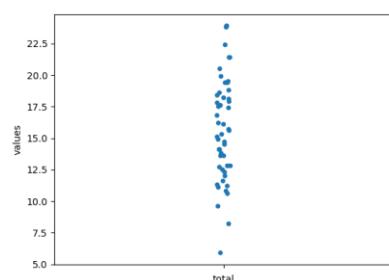
# Strip Plots and Swarm Plots

- Strip plots show a scatter plot when one axis is categorical
  - Jitter provides easier view when data points overlap

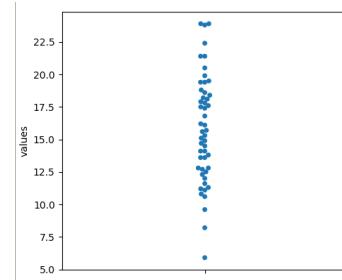
```
crashes = sns.load_dataset('car_crashes')
```



```
sns.stripplot(y='total',  
               data=crashes)
```



```
sns.stripplot(y='total',  
               data=crashes,  
               jitter=0.02)
```



```
sns.swarmplot(y='total',  
               data=crashes)
```

ch06\_seaborn\_sklearn/04\_stripplot.py

Strip plots show a scatter plot where one axis is categorical. Jitter can be added to better visualize the distribution of values. A swarm plot is similar to a strip plot but its points are not allowed to overlap. Points are placed non-overlapping but give a "swarm" effect to better indicate denser distributions.

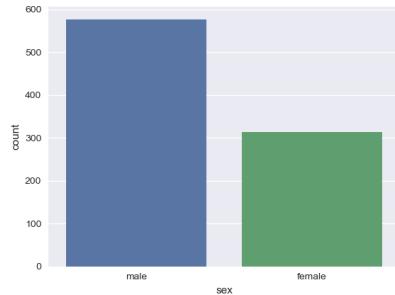
# Count Plots

- Count plots can show the count totals for categories as a bar

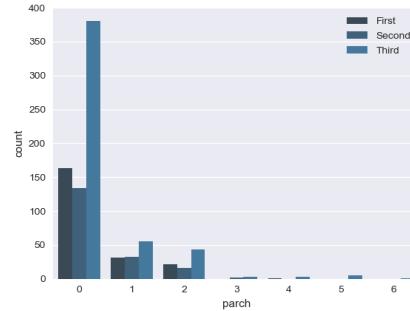
```
titanic = sns.load_dataset('titanic')
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

```
sns.countplot(x='sex',  
               data=titanic)
```



```
sns.set(palette='Blues_d')  
sns.countplot(x='parch', data=titanic, hue='class')
```



[ch06\\_seaborn\\_sklearn/07\\_countplot.py](#)

Titanic data here is used to create a count plot. Some of the columns explained:

survived - (0=no, 1=yes)

pclass - passenger class (1=1<sup>st</sup>, 2=2<sup>nd</sup>, 3=3<sup>rd</sup>)

sibsp - number of siblings and/or spouses aboard

parch = number of parents and/or children aboard

# Heatmap

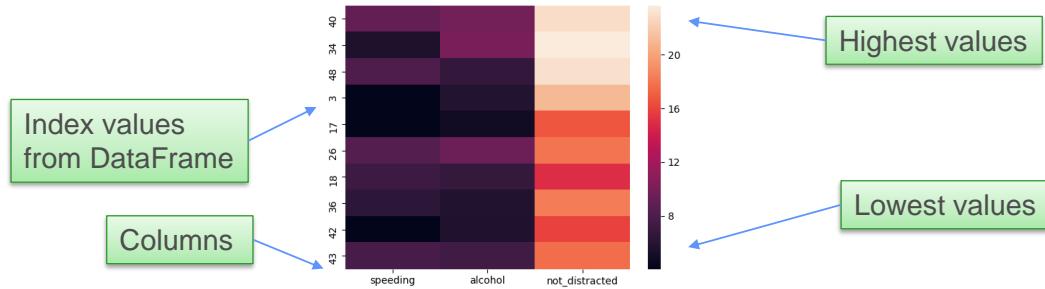
Heatmaps provide quick visual representation, through colors, of the intensities of data values

```
import matplotlib.pyplot as plt
import seaborn as sns

crashes = sns.load_dataset('car_crashes')
most = crashes.sort_values(by='total', ascending=False).head(10)

sns.heatmap(most[['speeding', 'alcohol', 'not_distracted']])
plt.show()
```

	speeding	alcohol	not_distracted
40	9.082	9.799	22.944
34	5.497	10.038	23.661
48	8.092	6.664	23.086
3	4.032	5.824	21.056
17	4.066	4.922	16.692
26	8.346	9.416	17.976
18	7.175	6.765	14.965
36	6.368	5.771	18.308
42	4.095	5.655	15.990
43	7.760	7.372	17.654



ch06\_seaborn\_sklearn/06\_heatmap.py

Heatmaps can provide fast insight as to the relationship of higher and lower values within features.



## Your Turn! - Task 6-1

- Using Seaborn pairplot() plot the response (averages) against the features year, atbats, and hits
  - We will only examine year > 2000
  - Use a hue in the pairplot() set to the year
  - Does there appear to be a relationship between any of these features and the response?
  - Work from and complete task6\_1\_starter.py



# What is a Scikit?

- **Scikits** (SciPy Toolkits) are a series of Python-based addons to SciPy
  - There are over 70+ Scikits
  - They are too specific or too large to be included in SciPy
  - List of toolkits can be found here: <https://scikits.appspot.com/scikits>
  - *Two toolkits* that predominate include:
    - **Scikit-learn**
      - Data mining / Machine learning toolkit
      - Allows for the creation and application of models against datasets to predict outcome
    - **Scikit-image**
      - Provides tools for image filtering, morphing, exposure, etc.

Scikit has two major subprojects that get most users attention: sklearn and skimage. Scikit-learn is sometimes shortened to sklearn (it's also the package name to import into Python scripts).

Scikit-image is used for image processing (e.g., filtering, morphing, exposure manipulation). An example of its use is shown on the next slide.



## Scikit-learn (sklearn)

- Provides tools for **data mining** and **machine learning**
  - Built upon SciPy  
(Pandas, NumPy, Matplotlib)
- Emphasizes modeling
  - *Data loading and manipulation* should be accomplished beforehand using Pandas or NumPy
- Already installed in Anaconda
  - If not installed, you can use pip:

Created by David Cournapeau  
in 2007 as a Google Summer  
of Code Project

`pip install scikit-learn`

Scikit-learn comes pre-packaged with the Anaconda distribution, however, it can also be installed with `pip install scikit-learn`.



# What is Machine Learning?

- Machine Learning is the act of predicting output from unknown data
  - The computer "learns" from the data it is provided by considering patterns, likelihood, and groupings
- ML models are used to *predict output responses* or *identify structure from the data*
  - Most ML models are mathematically described
- Machine learning commonly falls into two broad categories:
  - Supervised learning (most common learning form)
  - Unsupervised learning

The term machine learning dates back as far as 1959. It is one form of artificial intelligence. Typically, ML involves creating a mathematical formula that can describe (predict) how an output will respond to a given input.

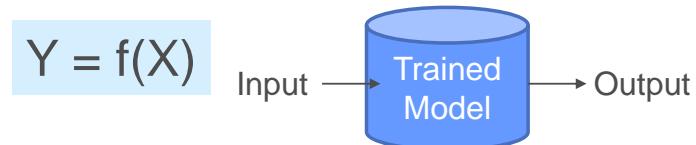
A third broad category of ML is often considered as well, semi-supervised learning, which is a mixture of both supervised and unsupervised techniques.

# Supervised Learning

- Sometimes referred to as ***predictive modeling***
- Supervised learning involves creating a model that has been "trained" based on known data
  - It is called supervised because we have knowledge of the response for the provided inputs



- That model when properly expressed will accept input variables ( $X$ ) and produce an predictive output ( $Y$ ):

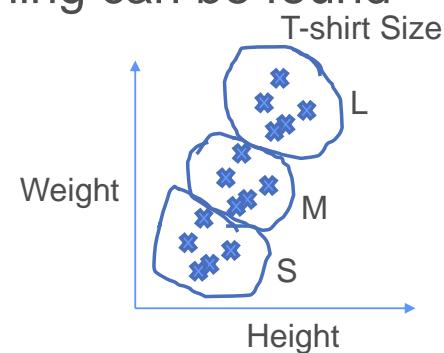


Supervised learning relies on known output values. In other words, given an input, we then must also know the result. The input and output values are then used to develop a model than can accurately predict results for future unknown data values. This occurs every day in practical business situations. For example, financial companies will offer a credit card to someone with a job because they have learned (via supervised data results) that people with jobs are more likely to pay back a credit card than those without jobs.

*Features* represent the number of columns while *samples* represents the number of rows or records. The *response* (or labels) represents the output result.

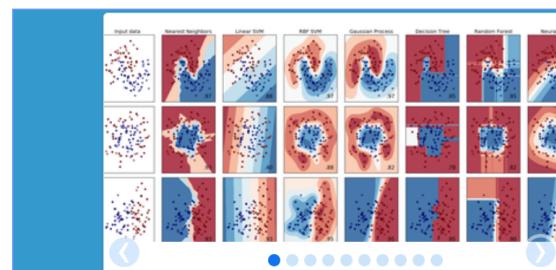
# Unsupervised Learning

- Unsupervised learning uses features but *without a known response*
  - There's no knowledge of the outcome or results
  - Data is **unlabeled** (*there is no Y*)
- **Structure is extracted** (learned) from the data
- Examples of unsupervised learning can be found in consumer behavior models
  - Segmenting consumers, voters, coins, pictures, animals or any items helps us determine how other items with similar characteristics may behave



Unsupervised learning has unlabeled data. There are no pre-discovered classifications. By identifying groups and looking for patterns in the input values we can develop structure about the data. This can be used to determine how other consumers might behave if they have similar characteristics to other consumers. If many teen consumers purchase role playing board games AND wooden fighting swords, it might be deemed reasonable that another teen customer who purchases a role-playing board game could be interested in wooden fighting swords. In the above example, if we know the height and weight of an individual, we may be able to reasonably determine that person's t-shirt size.

# Machine Learning and Scikit-learn Tools



## scikit-learn

*Machine Learning in Python*

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

<http://scikit-learn.org>

### Classification

Identifying to which category an object belongs to.

**Applications:** Spam detection, Image recognition.

**Algorithms:** SVM, nearest neighbors, random forest, ...

[— Examples](#)

### Regression

Predicting a continuous-valued attribute associated with an object.

**Applications:** Drug response, Stock prices.

**Algorithms:** SVR, ridge regression, Lasso, ...

[— Examples](#)

### Clustering

Automatic grouping of similar objects into sets.

**Applications:** Customer segmentation, Grouping experiment outcomes

**Algorithms:** k-Means, spectral clustering, mean-shift, ...

[— Examples](#)

### Dimensionality reduction

Reducing the number of random variables to consider.

**Applications:** Visualization, Increased efficiency

**Algorithms:** PCA, feature selection, non-negative matrix factorization.

[— Examples](#)

### Model selection

Comparing, validating and choosing parameters and models.

**Goal:** Improved accuracy via parameter tuning

**Modules:** grid search, cross validation, metrics.

[— Examples](#)

### Preprocessing

Feature extraction and normalization.

**Application:** Transforming input data such as text for use with machine learning algorithms.

**Modules:** preprocessing, feature extraction.

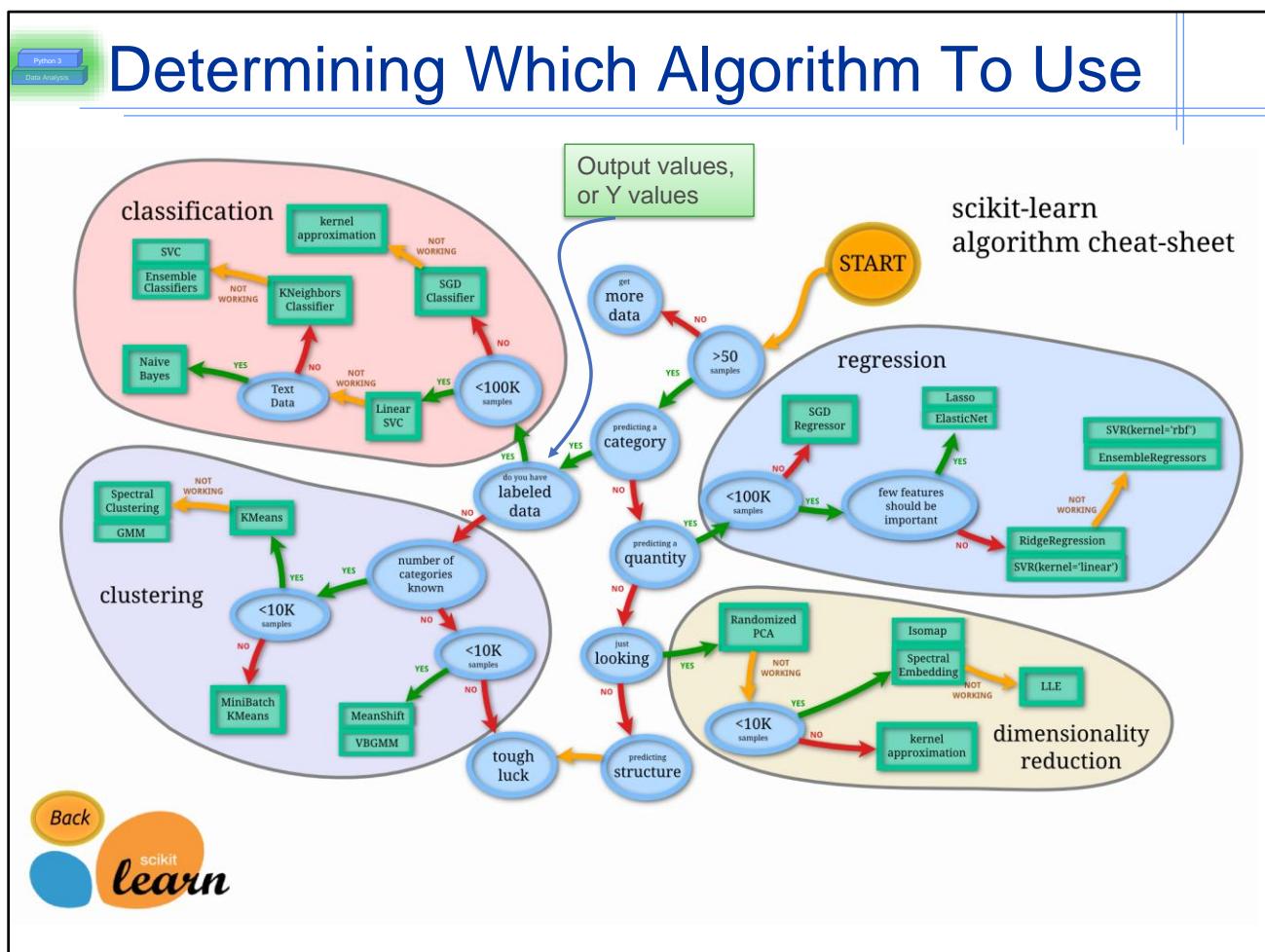
[— Examples](#)

Classification is the act of solving a problem by identifying to which category an observation belongs (uses supervised learning). Regressions are algorithms to assist in predictive analysis where we have a continuous response (non-discrete). Clustering is an unsupervised learning technique (several algorithms exist for this) for learning how to simplify complex data into groups.



# Scikit Supervised Learning Support

- Supervised learning comes in two broad forms:
  - **Regressions** - involve continuous response values
  - **Classifications** - discrete-valued (finite) responses
- Sklearn supervised learning algorithms include:
  - Ordinary Least Squares (OLS)
  - Ridge
  - Lasso
  - Elastic Net
  - Bayesian
  - Logistic
  - Support Vector Machines
  - Decision Trees
  - Ensemble Methods
  - Others...



The chart shown above is developed and provided by the Scikit-learn development team and can be viewed online at:

[http://scikit-learn.org/stable/tutorial/machine\\_learning\\_map/index.html](http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html)



# Regressions and Linear Regressions

- **Regressions** are a type of *supervised learning* in which the *response* is continuous
- **Linear Regressions** are a specific type of model used in regression problems
  - Advantages:
    - Easy to implement, well-understood by many
    - Fast algorithm (important for large datasets)
    - No tuning required, unlike KNN
    - Easy to interpret/understand
  - Disadvantages:
    - Not as accurate as other models due to its linear nature which doesn't often model the real-world

ch06\_seaborn\_sklearn/10\_simple\_regression.py

Linear regressions are useful tools for forecasting data sets such as stocks or weather. Linear Regressions attempt to predict scores on a response variable (Y) based on scores of a predictor (feature) variable (X). When only one predictor variable exists, it is called a *simple regression*.

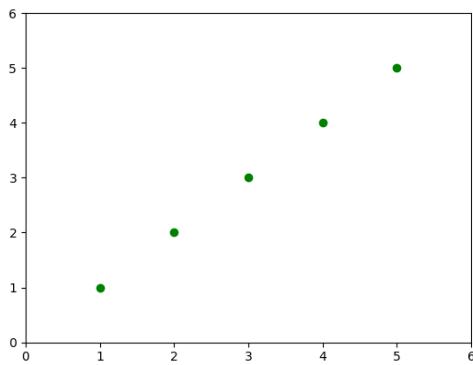
Linear regressions are used when the target value is expected to be a linear combination of the input variables.

KNN = K Nearest Neighbors. It is a form of classification that looks at all responses and then classifies new responses based on proximity to other responses. Tuning the K value is required with KNN solutions.

# Linear Regressions

- Linear Regressions use a "best fit" model (smallest sum of squares difference  $Y - Y'$ )
- Consider the following data points:

```
points = [(1, 1), (2, 2),
           (3, 3), (4, 4), (5, 5)]
data_array = np.array(points)
```



```
x = data_array[:, 0]
y = data_array[:, 1]
plt.plot(X, y, 'go')
plt.xlim(0, 6)
plt.ylim(0, 6)
plt.show()
```

x-values  
[1 2 3 4 5]

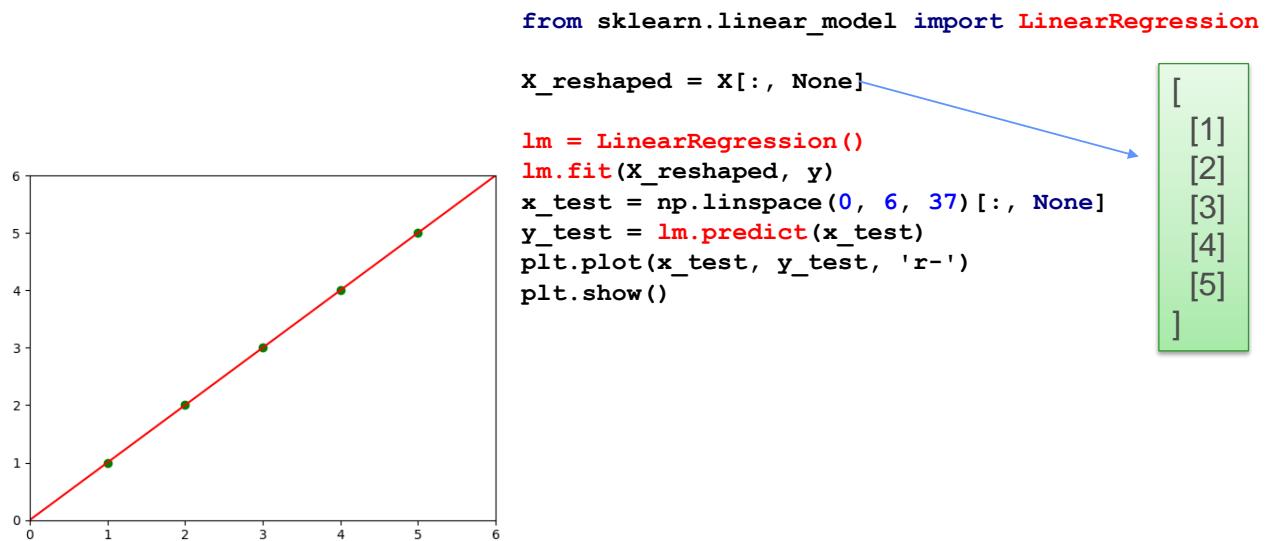
y-values  
[1 2 3 4 5]

ch06\_seaborn\_sklearn/10\_simple\_regression.py

The example uses 5 data points (plotted by Matplotlib). The data exists within a  $5 \times 2$  Numpy array. The X values are extracted using `data_array[:, 0]`.

# Linear Regression Plotted

- A line can be made that creates the smallest value of the difference of the sum of the squares
  - $\sum (Y - Y')^2 = 0$  for all data values in this example



ch06\_seaborn\_sklearn/10\_simple\_regression.py

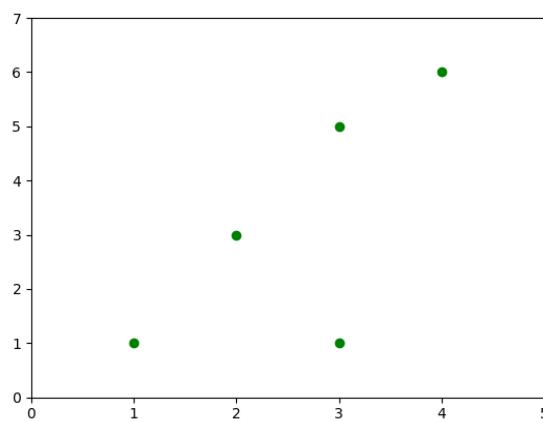
Because all data points lie on the generated line, we can accurately predict where other (future) values will appear.

## Non-Linear Data Points

- If data points are not exactly on a line, a best fit will be made:

```
points = [(1, 1), (2, 3),  
          (3, 1), (3, 5), (4, 6)]
```

```
data_array = np.array(points)
```



```
X = data_array[:, 0]  
y = data_array[:, 1]  
plt.plot(X, y, 'go')  
plt.xlim(0, 5)  
plt.ylim(0, 7)  
plt.show()
```

[1 2 3 3 4]  
[1 3 1 5 6]

ch06\_seaborn\_sklearn/10\_simple\_regression.py

# Creating the Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

```
x_reshaped = x[:, None]
```

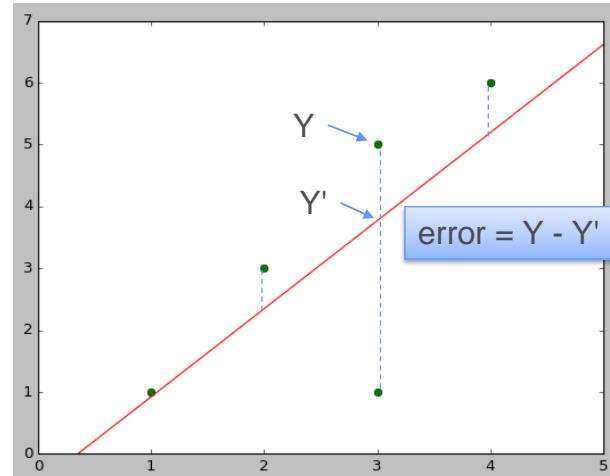
X
[1 2 3 3 4]

X-reshaped

[
[1]
[2]
[3]
[3]
[4]
]

```
lm = LinearRegression()
lm.fit(x_reshaped, y)
x_test =
    np.linspace(0, 5, 31)[:, None]
y_test = lm.predict(x_test)
plt.plot(x_test, y_test, 'r-')
plt.show()
```

predict() can be used to "predict" future Y values from the model.



ch06\_seaborn\_sklearn/10\_simple\_regression.py

In the example, a linear regression is created from the data points encountered on the previous slide. However, before a "best fit" can be made by invoking the `fit()` method, the X data must be reshaped so that all of its values are each in their own row. Now a `fit()` is applied. Finally, any new value can be used with the `predict()` method (such as `x_test` values) to yield predictive results (`y_test`). Matplotlib shows the regression line.

The `LinearRegression()` constructor accepts an argument: `fit_intercept`. `fit_intercept` is normally set to `True`, which means it will take the intercept value into consideration, otherwise `fit_intercept=False` will use a `0` value for the intercept. This may be true if the when x and y are assumed to be proportional.

# Model Coefficients and Scoring

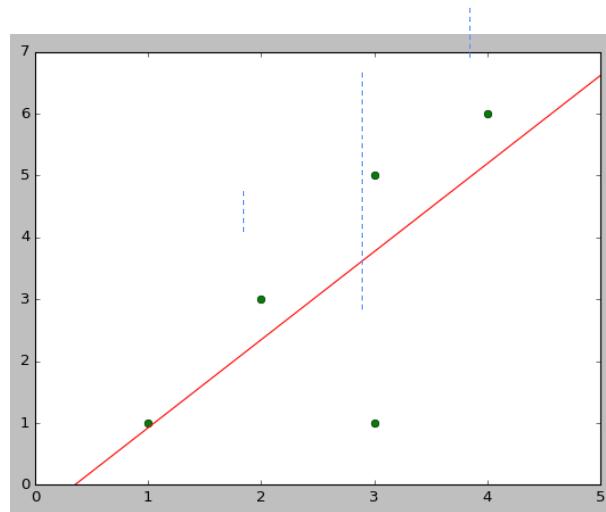
- Coefficients from the model may be obtained:

```
print(lm.intercept_) -0.5
```

```
print(lm.coef_) [1.42307]
```

```
print(lm.score(x_values,  
y_values))
```

0.77



- The model may be evaluated for accuracy by examining its score
  - The score for linear regressions is a value between 0 and 1 (higher = better)

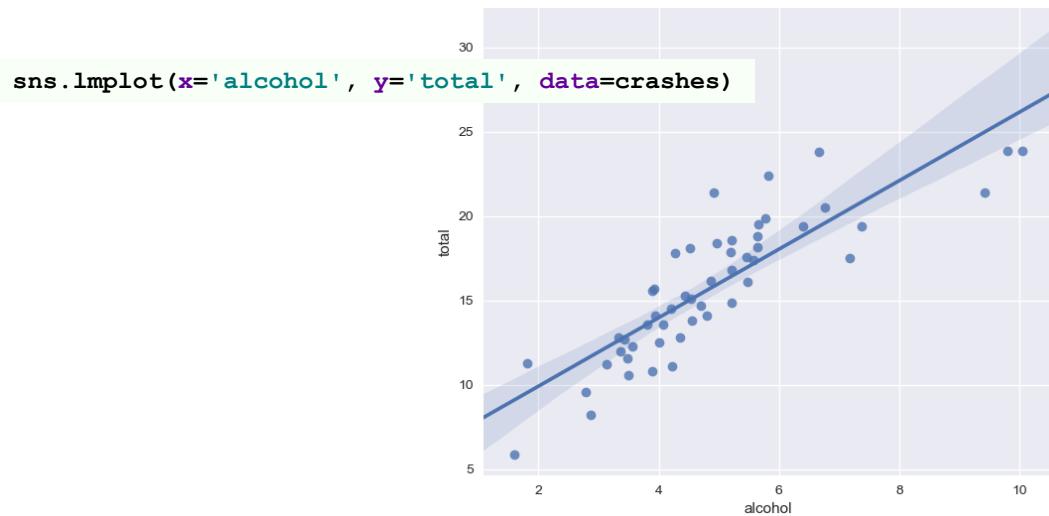
ch06\_seaborn\_sklearn/10\_simple\_regression.py

Note: values that end with an underscore (as in lm.coef\_) is Scikit's way of indicating the value is an estimation.

The intercept value is our value when X=0. The coefficients refer to the slope of the line for a simple regression. The score() method returns the r-squared value, or coefficient of determination. This value can be from 0.0 to 1.0 where 1.0 arises when all data values fall onto the regression.

# Alcohol-Related Crashes

- As an example, we'll fit a model to our car crashes dataset by examining only the alcohol-related crashes against the total crashes



ch06\_seaborn\_sklearn/11\_linear\_regression.py

# Creating Test and Training Data

- For proper model evaluation, samples should be broken into training data (to construct the model) and test data (to evaluate it)

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

crashes = sns.load_dataset('car_crashes')

X = crashes['alcohol']
y = crashes['total']

# splits 75-25 train-test by default
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)

lm = LinearRegression()
lm.fit(X_train[:, None], y_train)

y_pred = lm.predict(X=X_test[:, None])
```

These values are predicted based on the model,  
but how well do they hold up?

ch06\_seaborn\_sklearn/11\_linear\_regression.py

# Evaluating the Model

- In a regression, accuracy is not useful, instead we'll use continuous value metrics for evaluating the model

```
print(y_pred) [ 13.9 16.5 15.1 25.4 16.5 9.4 12.2 17.3 13.1 16.5 17.8 12.9 12.9]
print(y_test.values) [ 14.1 18.6 13.8 21.4 16.8 11.3 11.2 17.4 12.3 14.9 22.4 11.6 10.6]
```

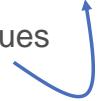
- We'll use the `sklearn.metrics` module to provide our error metric functions:

```
from sklearn.metrics import mean_squared_error
np.sqrt(mean_squared_error(y_test.values, y_pred))
```

2.11655482118

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^n (P_i - O_i)^2}{n}}$$

The RMSE can be directly compared to the Y test values to see how much error is introduced in this model



ch06\_seaborn\_sklearn/11\_linear\_regression.py

Several values exist to check the error. MAE (mean absolute error), MSE (mean squared error), and RMSE (root mean squared error). Generally the RMSE value provides the best metric. It takes the square root of the  $Y - Y'$  values at the end of summing all the error values. The RMSE penalizes the errors that are larger in a harsher way. The RMSE value can be compared to the response variable ( $Y$ ) because it is in the same units.



# Summary

- Seaborn can use several datasets for test purposes
  - Seaborn provides numerous plot types
  - These plot types are usually simpler to use than attempting the plots directly using Matplotlib
- Toolkits such as Scikit-learn provide ML algorithms that help build models that can accurately predict the behavior of systems given unknown data
- Both supervised and unsupervised algorithms exist within Scikit-learn



# Intro to Python 3 with Numerical Analysis

Exercise Workbook

# Task 1-1

## Python Environment Setup and Test



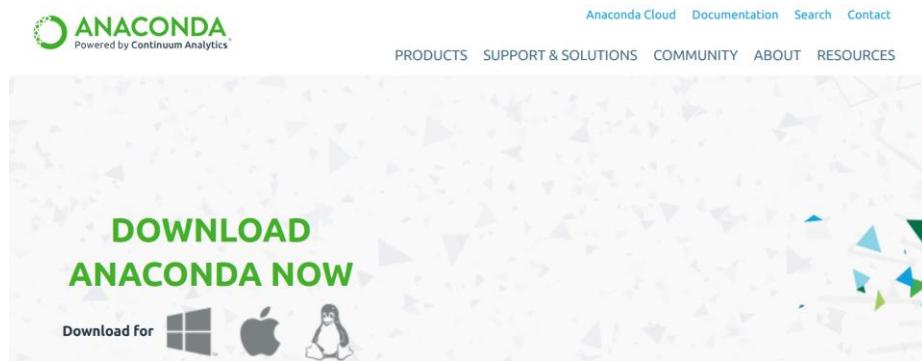
### Overview

This task is designed to help you setup and establish your Python environment.



### Install Python

If you have not already done so, install the proper Python Distribution, the Anaconda Distribution, by visiting  
<https://www.anaconda.com/distribution/>.



Click the link to download the appropriate version for your platform.

Watch for options to "install for all users" and to add the Python distribution to your PATH environment variable.

## Anaconda 2018.12 for Windows Installer

### Python 3.7 version

[Download](#)

64-Bit Graphical Installer (614.3 MB)  
32-Bit Graphical Installer (509.7 MB)

### Python 2.7 version

[Download](#)

64-Bit Graphical Installer (560.6 MB)  
32-Bit Graphical Installer (458.6 MB)

Or

## Anaconda 2018.12 for macOS Installer

### Python 3.7 version

[Download](#)

64-Bit Graphical Installer (652.7 MB)  
64-Bit Command Line Installer (557 MB)

### Python 2.7 version

[Download](#)

64-Bit Graphical Installer (640.7 MB)  
64-Bit Command Line Installer (547 MB)

After the installation, **open a console** or terminal window and type:

`python -v`

If the command is not recognized or the wrong Python version is displayed, you will need to modify your PATH environment variable to include the <PYTHON\_HOME> directory. Setting the PATH will vary from system to system so ask for help if assistance is needed.

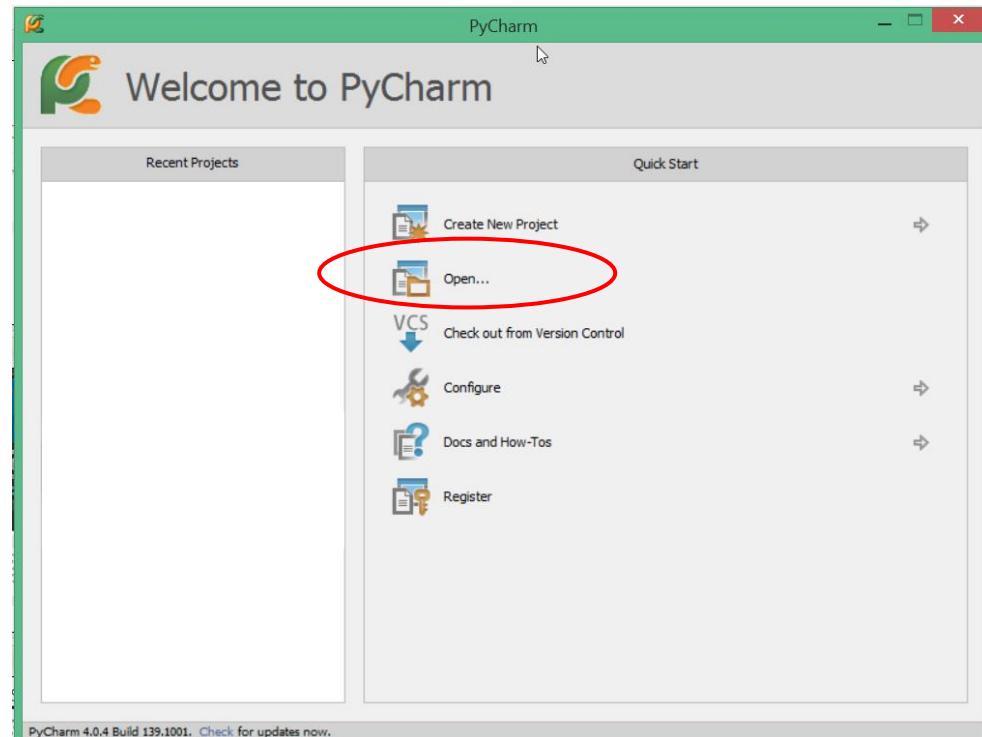
### Note for Macs

Use the command: **python3.6** after installing to invoke the 3.6 version of Python instead of the installed default, 2.7. For Python 3.7, replace 3.6 with 3.7.

## Launch PyCharm, Set Up the Student Files



PyCharm Community Edition is a free Python IDE created by JetBrains. After launching PyCharm, a "Welcome to PyCharm" dialog will appear. Select "Open..." and browse to the directory where your student files are.



## Run a Test Script

Open the **02\_iterating.py** file found in the **student\_files/ch01\_overview** folder.

Right-click in the source code and select Run 02\_iterating.py.

You should see results from running this script.

This concludes Task 1-1.

That's It!

# Task 1-2

## Working with Strings and Controls



### Overview

This exercise is intended to provide practice with Python Strings. You will input a user string URL, parse it, and extract ONLY the domain name portion of it.



#### Obtain a URL from the Standard Input

Query for user input, asking the user to input a proper URL. Do this using the `input()` method.



#### Remove the Protocol Prefix

If the URL begins with "http://" or "https://" remove it from the input string. Do this using `startswith()` and slicing. Iterate over the provided prefixes checking to see if the string starts with that prefix.

```
for prefix in prefixes:  
    if url.startswith(prefix):  
        domain = url[len(prefix):]
```



#### Remove the URL Suffix

Repeat the above step this time removing any content after the domain name. A list of suffixes was provided, use this list invoking the string class's `find()` method. If `find()` returns -1 then the suffix is not in the string. Otherwise, `find()` will return

the position of the suffix character within the string. Take the slice from the beginning of the string to this position.

```
for suffix in suffixes:  
    pos = domain.find(suffix)  
    if pos != -1:  
        domain = domain[:pos]
```



## Print the String

Print the remaining string after the prefix and suffix have been stripped off.

# Task 1-3

## Lists and Sorting



### Overview

This exercise lists files (not directories) from largest to smallest when given a specified input directory to read from. It uses the lists and sorting techniques discussed in class to accomplish the task.



### Prompt for a Search a Path

Prompt the user for a file path and pattern to search. It can be relative or absolute (for example: `../resources` or `/temp`).



### Obtain a List of Directory Items

Use `os.listdir(path)` as discussed in the hints to extract a list of directories and files that match. Store the results into a list variable.



### Iterate Over Results, Check If They Are Files

The results from the `os.listdir()` call may include directory names. Iterate over the results, checking each item to see if they are files. If so, add them into a new *list*. Here's one way to do it:

```
path = input('Path to search: ')
dir_contents = os.listdir(path)
files = []
for file_item in dir_contents:
    fullpath = os.path.join(path, file_item)
    if os.path.isfile(fullpath):
        # to be completed in the next step
```

Note: the `os.path.join()` method is not necessary, you can also just add the path and filename together using plus (+).



## Create a List of File Names Only

Within the `if` statement (on the next line and indented), append the files into a new, empty list. You should store two things into the list: the file name and the file size. Do this by storing things into a tuple (ex: `(filename, filesize)` ).

```
files.append((file_item, os.stat(fullpath).st_size))
```



## Sort (by Size) and Display the List

With the new list (of files only), create a function (a lambda) that will sort based on file-size. Use `os.stat(filename).st_size` for this.

```
files.sort(reverse=True,  
          key=lambda name_size: name_size[1])
```

Print the resulting list.

```
for name, size in files:  
    print('{name:<30}{size:10}'  
          .format(name=name, size=size))
```

That's it. Test it out!

# Task 1-4

## Working with Dictionaries



### Overview

This exercise will read all of the words within a text file. It then places the words into a dictionary, counts the word occurrences, and finally displays the top 100 most frequent words that are five letters or more.



#### Create a Dictionary to Store Words

```
wordcount = {}
```



#### Read Lines from the Entire File

Even though we've touched on files briefly in previous exercises, we haven't officially introduced working with them. So, at this point, we will ignore exception handling. For this step, iterate over the file reading line-by-line. Break each line up into individual words using the `split()` method as follows:

```
for line in open('alice.txt'):
    words = line.split()
```



## Store Words in the Dictionary

For each of the words, add them to a dictionary as the key to the dictionary. The dictionary will store the number of occurrences of words. If the word is already in the dictionary, increment its count:

```
for word in words:  
    if word in wordcount:  
        wordcount[word] += 1  
    else:  
        wordcount[word] = 1
```



## Sort the Dictionary Items Based on Occurrences (Frequency Most to Least)

A dictionary cannot be sorted, but `dictionary.items()` can. `dictionary.items()` returns a list of tuples in the form of:

`[(word1, count1), (word2, count2), ...]`

You will want to sort each item based on the count value. The count value in each case is the second item in each tuple, so the following key function should work:

```
key=lambda a:a[1]
```

where "a" in this case is a tuple as shown in the discussion above.

Don't forget to sort in reverse order (descending):

```
sortedwords = sorted(wordcount.items(), key=lambda a:a[1], reverse=True)
```



## Obtain Only Words 5 Letters or Greater

sortedwords, from the previous step is a list of (word, count) tuples sorted in order of most-to-least frequent. Create a new list that only contains words that are 5 letters or greater. A list comprehension can do this for you. Can you create this on your own first? If you need help, look down at the bottom of the page.

Finally, the list is sorted from most frequent to least. Use slicing to print the top 100 items in the list. Again, can you do this on your own?

```
five_letters = [(word, count) for word, count in  
                sortedwords if len(word) >= 5]
```

# Task 3-1

## Checkerboards and ndarrays



### Overview

This exercise consists of two sub-tasks broken into the following:

Task 3-1 (Part 1) - creates an  $8 \times 8$  checkerboard ndarray

Task 3-1 (Part 2) - modifies the solution to work for an  $n \times n$  checkerboard

### Task 3-1: $8 \times 8$ Checkboard (Part 1)



#### Begin Creating the ndarray

For this exercise, you may work from your ch03\_numpy/task3\_1\_starter.py file. Open this file and locate the section marked # Part 1. You will create your checkerboard array here. Begin by creating a one-dimensional ndarray with eight elements alternating between 0 and 1. Try this on your own before viewing the solution to this step below.

Solution to step 1 (note: there are many ways to do this):

```
row = np.array([0,1]*4)
```



## Create Row 2

Repeat the previous step, this time creating a one-dimensional ndarray with 8 values that begins with a 1 and ends with 0 (i.e., it should be the opposite, or inverted, from the one you created in step 1).

*Can you do this on your own before looking below?*

Solution to step 2:

```
row_shifted = np.roll(row, 1)
```



## Combine the Two Rows

Combine the two rows created in the previous steps. Hint: stack them.

*Can you do this on your own before looking onward?*

Solution to step 3:

```
partial_checkerboard =
np.column_stack((row, row_shifted))
```



## Complete the Checkerboard

You now have two combined rows of alternating values for the checkerboard. Complete the checkerboard by repeating it 4 more times. Hint: use `np.tile()`.

*Can you do this on your own before looking below?*

Solution to step 4:

```
checkerboard = np.tile(partial_checkerboard, [1, 4])
```

That's it with Part 1. Next, we'll attempt Part 2...

## Task 3-1: An $n \times n$ Checkboard (Part 2)



### Begin Creating the ndarray

For this exercise, you may continue working from the same file (`ch03_numpy/task3_1_starter.py`). Create a variable called `width` and assign it a value (any value is okay). Create a 1-D array of all zeros:

```
width = 5  
  
row = np.zeros(width, dtype=int)
```



### Insert 1's in Every Other Position

Use slicing to add 1's to every other position in the array.

*Can you do this on your own before looking onward?*

Solution to step 2

```
row[1::2] = 1
```



## Create a Second Row

Create another array, this time with 0's and 1's inverted. You can use `np.logical_not(arr)` to help with this.

*Can you do this on your own before looking below?*

Solution to step 3:

```
row_inverted = np.logical_not(row)
```



## Combine the Two Rows

Combine the two arrays previously created so that they stack upon each other. Hint: use `np.stack()`.

*Can you do this on your own before looking onward?*

Solution to step 4:

```
two_rows = np.stack([row, row_inverted], axis=0)
```



## Repeat the "Two Rows" Vertically

With the two rows created in the previous step, repeat them vertically (width/2 times). Hint: use `np.tile()`. Note that since width can be an odd number, 5, for example, you should round up when dividing the width in half.

```
larger_board =
    np.tile(two_rows, [int(np.ceil(width/2)),1])
```



## Slice the Board to the Size of "Width"

The board may currently be larger than required (if width is odd), therefore we will take a slice of the board so that only "width" number of rows are used.

```
checkerboard = larger_board[:width]
```

This will slice only width number of rows.

That's it! Test it out by running the script and printing your board.

# Task 3-2

## Jupyter Notebook



### Overview

In this exercise you will use Jupyter Notebook to create the temperatures NumPy Array. You will then calculate the average daily temperature and average temperature for the week.



### Launch Jupyter Notebook

Where you launch your Jupyter Notebook server is not critical, however, the directory where the server is started will determine what files you can view. Since there are no starter files for this particular exercise, you may start your server in a temporary directory.

Change to a temporary directory, such as c:\temp.

```
cd c:\temp      (if using a Windows-based computer)
```

Launch the Jupyter server.

```
c:\temp> jupyter notebook
```

Once the server starts, your browser should open automatically.

From the server's start page, select the "**New**" dropdown menu item on the right-side of the page. Next, select **Python [root]**.



## Create the Code Cells Containing Temperature Data

By creating Code cells within your new page, enter the temperature data as shown on the slide (or as shown below). You should also import necessary modules, such as numpy.

```
import numpy as np
```

```
data = [[1, 88, 68, 25, 10, 'Sunny', False],  
        [2, 84, 65, 31, 5, 'Cloudy', False],  
        [3, 86, 66, 32, 5, 'Light Rain', False],  
        [4, 89, 67, 26, 5, 'Rain', False],  
        [5, 92, 70, 22, 10, 'Sunny', False],  
        [6, 95, 71, 18, 20, 'Sunny', True],  
        [7, 94, 69, 27, 10, 'Sunny', False]]
```

Create the Numpy array from the data you entered. Note: generally it is best to break these tasks into separate cells for ease of maintenance.

```
weather_data = np.array(data, dtype=object)
```

By declaring values as `dtype=object`, we can create arrays that contain various types of values.

Display the array characteristics in another cell:

```
weather_data.ndim, weather_data.shape,  
weather_data.size, weather_data.strides
```



## Obtain the High and Low Temperatures

Using slicing, we can obtain just the high and low temperatures and treat them as float types only. From this, we can get the average of each day.

```
high_low = np.array(weather_data[:,1:3], dtype=float)
```

```
daily_avg = high_low.mean(axis=1)
```

Place the above code in individual cells and test that they work.

Finally, we can get the average temperature for the week from this resulting data.

```
weekly_avg = daily_avg.mean()
```

That's it! Clean your solution up and test it out!

# Task 3-3

## *genfromtxt() and Batting Averages*



### Overview

In this exercise, you will determine the top 100 batting averages from the provided data found in <student\_files>/resources/baseball/Batting.csv. To do this, you must read the atbats column (column 6) and the hits column (column 8) from the file. Average is found by dividing hits by atbats.



### Read the Batting.csv File Data

For this exercise, you may work from either Jupyter Notebooks (starting a new notebook page) or from task3\_3\_starter.py.

Read the data from the file. You can use a relative path and forward slashes for path separators.

```
data = np.genfromtxt(  
    '../resources/baseball/Batting.csv',  
    skip_header=1, usecols=(1, 6, 8),  
    delimiter=',', dtype=float)
```



### Remove Unwanted Records (atbats < 502)

We need to remove the rows where the number of atbats is less than 502. Or, another way to say this is we want to keep rows where atbats is greater than or equal to 502.

The following can help accomplish this:

```
data = data[data[:, 1] >= 502]
```



## Remove Records from Before 1957

After the 502 atbats filtering statement, add the following new criterion that removes records from before 1957:

```
data = data[data[:, 0] >= 1957]
```



## Calculate the Averages, Join the Hits, Atbats, and Averages Columns

With the file data read into an array, you can now calculate the averages. Numpy will perform this division on an element-by-element basis:

```
avgs = data[:, 2] / data[:, 1]
```

After performing the calculation, add the new column data to the array of hits and atbats:

```
data = np.column_stack([data, avgs])
```



## Sort the Averages

We can use the **argsort()** method to sort according to the desired column. The desired column, in this case, is the 4th column or the averages column.

```
data = data[data[:, 3].argsort()]
```

## Get the Final Results

The final results have been obtained. We should now be able to get the top 100 batting averages:

```
print(data[-1:-100:-1, 3])
```

That's it. Test it out!

# Task 4-1

## Scatter Plots and Matplotlib



### Overview

In this exercise, you will create a scatter plot that plots the batting average (x-axis) against the atbats (y-axis) values obtained during the previous exercise. You will use Matplotlib for this task.

You may work from either the task4\_1\_starter.py or you may continue from your task3\_3\_starter.py file that you worked on previously.



### Dynamically Find the Max Average

First, add the appropriate Matplotlib import:

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

Using the argmax() function, we can find the row where the maximum average occurs. Then we can plug that back into the array to get the max average value and the atbats value where this occurs:

```
max_avg_idx = data[:, 3].argmax()  
max_avg = data[max_avg_idx, 3]  
atbats = data[max_avg_idx, 1]
```



## Create a Figure and Axes

For plotting, we will multiply the averages column by 1000 (it is common to report baseball averages in the hundreds).

```
data[:,3] = data[:,3] * 1000
```

Next, create the figure, add a plot using `add_axes()`, and then set the xlabel, ylabel, xlim, and ylim values:

```
fig = plt.figure(figsize=(8,6), facecolor='#ccccff')
ax = fig.add_axes((0.1, 0.1, 0.8, 0.8))
```



## Invoke scatter() from the Figure

We have the figure information established, it's time to create the plot. We will create a scatter diagram for our visualization. The following plots the average on the x-axis and atbats on the y-axis:

```
ax.scatter(data[:, 3], data[:, 1],
           marker='.', c='#cdcd24')
```

# Task 5-1

## Using Pandas and DataFrames



### Overview

In this exercise, you will repeat the batting averages we worked on in the numpy chapters. This time we will use only the Pandas framework. Using Pandas, you will find out what a 50 percentile MLB baseball player hits. Finally, we'll use Pandas built-in plotting features to create a scatter plot of the batting averages again.



### Create the Pandas DataFrame Using `read_csv()`

Create the DataFrame using `read_csv()`. Supply the `usecols` parameter. Assign new column names after reading the data:

```
data =  
pd.read_csv('.../.../resources/baseball/Batting.csv',  
            usecols=('yearID', 'AB', 'H'))
```



### Filter Records Using Boolean Indexing

Filter records using Boolean indexing. Use `.loc[]` and Boolean Indexing. Keep records with 502 or more atbats and year greater than or equal to 1957:

```
data = data.loc[(data['yearID'] >= 1957) &  
                (data['AB'] >= 502)]
```



## Calculate the Averages

Calculate the averages. You can easily create a new column in Pandas by referencing a new column name:

```
data['avgs'] = data.loc[:, 'H'] / data.loc[:, 'AB']
```



## Sort the DataFrame Values and Find the 50% Value

Sort the array values as described in our manual using the `sort_values()` method. Obtain the 50% value using the `describe()` function:

```
data.sort_values(by='avgs', inplace=True,  
                 ascending=False)  
  
print(data.describe().loc['50%', 'avgs'])
```



## Plot the Values Using Pandas Plot Method

Plot the values as we did in task 4-1. Use  
df.plot(kind='scatter', x='col', y='col'). Don't forget to show() it!

```
data.plot(kind='scatter', x='avgs', y='AB', marker='.')

plt.xlabel('Batting Averages')
plt.ylabel('At Bats')

plt.show()
```

# Task 5-2

## *Split-Apply-Combine Operations*



### Overview

In this exercise, you will re-visit our batting.csv file this time answering two questions:

1. Which team has hit the most home runs (cumulative)?
2. Which team hit the most home runs in 2015?

Answering these questions will involve loading the DataFrame and performing groupby() operations.



### Create the Initial DataFrame

Read all column-based data from batting.csv using read\_csv().

```
batting = pd.read_csv(filename)
```



### Check the Shape and info()

It's always good to verify the data by examining the shape and the DataFrame's info():

```
print(batting.shape)
print(batting.info())
```



## Answer Question 1

Group the results together by teamID. Then sum() them to get totals. Note: this will give you totals in other columns that may be meaningless, ignore those columns. Find the team with the highest sum in the HR column.

```
most_hrs = batting.groupby('teamID').sum()
               .sort_values('HR', ascending=False)
print(most_hrs.head())
print('Question 1 answer: {}'.format(most_hrs.index[0]))
```



## Answer Question 2

This answer will be similar to step 3 except we must first filter out all records that are not in 2015:

```
hrs_2015 = batting[batting['yearID'] == 2015]
               .groupby('teamID').sum().sort_values('HR', ascending=False)
print(hrs_2015.head())
print('Question 2 answer: {}'.format(hrs_2015.index[0]))
```



## Create the Bins for Plotting HRs by Decade

We'll define bins using a Python list going from 1870 to 2020:

```
bins = [1870, 1880, 1890, 1900, 1910, 1920, 1930, 1940,  
        1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020]
```



## "Cut" the yearID Column Into Bins

Break the yearID column into the respective bins defined in step 5 above.

*Can you do all of this on your own before looking below?*

```
battting['hr_decade'] =  
    pd.cut(battting['yearID'], bins=bins)
```



## Plot the HRs for Each Decade as a 'barh' Plot

First, use groupby() on the 'hr\_decade' column. Then, sum() the groups and plot only the HR column:

```
batting.groupby('hr_decade').sum()['HR']
    .plot(kind='barh')
```

If running this outside of Jupyter, don't forget to plt.show() it!

That's it! Display your plot and question answers.

# Task 5-3

## The Database and Pandas DataFrames



### Overview

In this exercise, you will connect to a database to retrieve baseball information. The data is contained within the **batting.db** SQLite database found in the current ch05 folder.

Your task is to determine the highest-paid player in MLB by querying the salaries table within the database using Panda's `read_sql()` method. This will create a Pandas DataFrame from which the `max()` method can be invoked to determine the maximum salary.



### Connect to the Database

Use the 'with' control to connect to the database. Refer to the Pandas example within the student manual on how to do this. Proper syntax uses the following structure:

```
with <connection_obj> as conn:  
    do stuff
```

```
with sqlite3.connect('batting.db') as conn:  
    <do stuff here shortly>
```

All remaining code should be placed inside of the 'with' control.



## Read the Data, Create the DataFrame

Read the playerid, salary, and year columns from the salaries table within the database.

The following sql should accomplish this task:

```
SELECT playerid, salary, year FROM salaries
```

*Can you accomplish this task on your own? Look below when finished.*

```
df = pd.read_sql(  
    'SELECT playerid, salary, year FROM salaries',  
    conn)  
print(df.shape)
```



## Determine the Maximum Salary, Year, and PlayerID

Once you have the DataFrame, use the max() function to retrieve the maximum value for the salary column. Next, use idxmax() to obtain the associated index for that maximum salary.

```
max_salary_idx = df['salary'].idxmax()
```

Use the index value that you just obtained to determine the year and playerid for the row where the max salary was earned.

```
max_salary = df.iloc[max_salary_idx, 1]
playerid = df.iloc[max_salary_idx, 0]
```



## Obtain the Player's Name

Once you have the playerid that represents the highest-paid player, obtain the player's firstname and lastname from the players table. You can do this with a standard SQL query, but let's load the result into a DataFrame (mostly just for the practice)...

```
SELECT firstname, lastname FROM players WHERE playerid=?
```

```
df2 = pd.read_sql(
    'SELECT firstname, lastname FROM players where playerid = ?',
    conn, params=[playerid])
```



## Display the Results

The last task is to display the results. That's it. Test it out when finished!

```
print(df2.iloc[0, 0], df2.iloc[0, 1])
```

# Task 6-1 (Appendix A) *pairplot() with Seaborn*



## Overview

In this exercise, you use Seaborn's `pairplot()` utility to create a visualization of three plots: averages vs. year, averages vs. atbats, and averages vs. hits.



## Write the `pairplot()` Call

There are only two lines of code to write in this exercise. The first is a Seaborn `pairplot()` call and the second uses Matplotlib to display it. Begin by writing the two lines of code at the bottom of the starter file:

```
sns.pairplot()  
plt.show()
```

Next, specify the DataFrame to be plotted:

```
sns.pairplot(batting)  
plt.show()
```

For all three plots, the y-axis will be the 'averages' column. Using Seaborn's `y_vars` parameter, plot the y-axis column.

```
sns.pairplot(batting, y_vars='averages')  
plt.show()
```

Define the columns that will comprise the x-axis of each of the three plots. Provide them as list to the `x_vars` parameter:

```
sns.pairplot(batting,  
             x_vars=['year', 'atbats', 'hits'], y_vars='averages')  
plt.show()
```

As mentioned on the requirements slide, add a hue for the 'year' column.

If you wish, you can also change the aspect and size parameters:

```
sns.pairplot(batting, x_vars=['year', 'atbats', 'hits'],  
             hue='year', aspect=0.7, size=7.5, y_vars='averages')  
plt.show()
```

That's it! Test it out!