Python Cheat Sheet

JUST THE BASICS

Created By: Arianne Colton and Sean Chen

General

- Python is case sensitive
- Python index starts from 0
- Python uses whitespace (tabs or spaces) to indent code instead of using braces.

Help

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

Module (aka Library)

Python module is simply a '.py' file

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

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

Scalar Types

Check data type : `type(variable)`

Six Commonly Used Data Types

1. **int/long*** - Large int automatically converts to long
2. **float*** - 64 bits, there is no 'double' type
3. **bool*** - True or False
4. **str*** - ASCII valued in Python 2.x and Unicode in Python 3
   - String can be in single/double/triple quotes
   - String is a sequence of characters, thus can be treated like other sequences
   - Special character can be done via `\` or preface with `r`

     ```
     strl = r'this\f?ff'
     ```

   - String formatting can be done in a number of ways

     ```
     template = '%.2f %s haha $%d';
     strl = template % (4.88, 'hola', 2)
     ```

Scalar Types

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

5. **NoneType(None)** - Python 'null' value (ONLY one instance of None object exists)
   - None is not a reserved keyword but rather a unique instance of 'NoneType'
   - None is common default value for optional function arguments :

     ```
     def func1(a, b, c = None)
     ```

   - Common usage of None :

     ```
     if variable is None :
     ```

6. **datetime** - built-in python 'datetime' module provides 'datetime', 'date', 'time' types.
   - 'datetime' combines information stored in 'date' and 'time'

Create datetime from String	`dtl = datetime.strptime('20091031', '%Y%m%d')`
Get 'date' object	`dtl.date()`
Get 'time' object	`dtl.time()`
Format datetime to String	`dtl.strftime('%m/%d/%Y %H:%M')`
Change Field Value	`dt2 = dtl.replace(minute = 0, second = 30)`
Get Difference	`diff = dtl - dt2` # diff is a 'datetime.timedelta' object

Note : Most objects in Python are mutable except for 'strings' and 'tuples'

Data Structures

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

Tuple

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

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

Application of Tuple

Swap variables	`b, a = a, b`

Data Structures

List

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

Create List	`list1 = [1, 'a', 3]` or `list1 = list(tup1)`
Concatenate Lists*	`list1 + list2` or `list1.extend(list2)`
Append to End of List	`list1.append('b')`
Insert to Specific Position	`list1.insert(posIdx, 'b')` **
Inverse of Insert	`valueAtIdx = list1.pop(posIdx)`
Remove First Value from List	`list1.remove('a')`
Check Membership	`3 in list1 => True` ***
Sort List	`list1.sort()`
Sort with User-Supplied Function	`list1.sort(key = len)` # sort by length

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

** Insert is computationally expensive compared with append.

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

Built-in 'bisect module‡

- Implements binary search and insertion into a sorted list
- 'bisect.bisect' finds the location, where 'bisect.insort' actually inserts into that location.

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

Slicing for sequence types†

† Sequence types include 'str', 'array', 'tuple', 'list', etc.

Notation	`list1[start:stop]`
	`list1[start:stop:step]` (if step is used) §

Note :
- 'start' index is included, but 'stop' index is NOT.
- start/stop can be omitted in which they default to the start/end.

§ Application of 'step' :

Take every other element	`list1[::2]`
Reverse a string	`str1[::-1]`

Dict (Hash Map)

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

* 'KeyError' exception if the key does not exist.

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

*** Returns the lists of keys and values in the same order. However, the order is not any particular order, aka it is most likely not sorted.

Valid dict key types

- Keys have to be immutable like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable too)
- The technical term here is 'hashability', check whether an object is hashable with the `hash('this is string'), hash([1, 2])` - this would fail.

Set

- A set is an **unordered** collection of UNIQUE elements.
- You can think of them like dicts but keys only.

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

- **Set operations :**

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

Python For Data Science *Cheat Sheet*

Python Basics

Learn More Python for Data Science Interactively at www.datacamp.com

Variables and Data Types

Variable Assignment

```
>>> x=5
>>> x
5
```

Calculations With Variables

```
>>> x+2
7
>>> x-2
3
>>> x*2
10
>>> x**2
25
>>> x%2
1
>>> x/float(2)
2.5
```

Sum of two variables
Subtraction of two variables
Multiplication of two variables
Exponentiation of a variable
Remainder of a variable
Division of a variable

Types and Type Conversion

```
str()    '5', '3.45', 'True'    Variables to strings
int()    5, 3, 1                Variables to integers
float()  5.0, 1.0               Variables to floats
bool()   True, True, True       Variables to booleans
```

Asking For Help

```
>>> help(str)
```

Lists

Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements

Index starts at 0

```
Subset
>>> my_list[1]          Select item at index 1
>>> my_list[-3]         Select 3rd last item
Slice
>>> my_list[1:3]        Select items at index 1 and 2
>>> my_list[1:]         Select items after index 0
>>> my_list[:3]         Select items before index 3
>>> my_list[:]          Copy my_list
Subset Lists of Lists
>>> my_list2[1][0]      my_list[list][itemOfList]
>>> my_list2[1][:2]
```

List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

List Methods

```
>>> my_list.index(a)          Get the index of an item
>>> my_list.count(a)          Count an item
>>> my_list.append('!')       Append an item at a time
>>> my_list.remove('!')       Remove an item
>>> del(my_list[0:1])         Remove an item
>>> my_list.reverse()         Reverse the list
>>> my_list.extend('!')       Append an item
>>> my_list.pop(-1)           Remove an item
>>> my_list.insert(0,'!')     Insert an item
>>> my_list.sort()            Sort the list
```

Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2
'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
'thisStringIsAwesomeInnit'
>>> 'm' in my_string
True
```

String Operations

Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```

String Methods

```
>>> my_string.upper()                String to uppercase
>>> my_string.lower()                String to lowercase
>>> my_string.count('w')             Count String elements
>>> my_string.replace('e', 'i')      Replace String elements
>>> my_string.strip()                Strip whitespace from ends
```

Libraries

Import libraries

```
>>> import numpy
>>> import numpy as np
```

Selective import

```
>>> from math import pi
```

pandas — Data analysis
scikit-learn — Machine learning
NumPy — Scientific computing
matplotlib — 2D plotting

Install Python

Leading open data science platform powered by Python

Free IDE that is included with Anaconda

Create and share documents with live code, visualizations, text, ...

Numpy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3],[4,5,6]])
```

Selecting Numpy Array Elements

Index starts at 0

```
Subset
>>> my_array[1]          Select item at index 1
2
Slice
>>> my_array[0:2]        Select items at index 0 and 1
array([1, 2])
Subset 2D Numpy arrays
>>> my_2darray[:,0]      my_2darray[rows, columns]
array([1, 4])
```

Numpy Array Operations

```
>>> my_array > 3
array([False, False, False,  True], dtype=bool)
>>> my_array * 2
array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
array([6, 8, 10, 12])
```

Numpy Array Functions

```
>>> my_array.shape                  Get the dimensions of the array
>>> np.append(other_array)          Append items to an array
>>> np.insert(my_array, 1, 5)       Insert items in an array
>>> np.delete(my_array,[1])         Delete items in an array
>>> np.mean(my_array)               Mean of the array
>>> np.median(my_array)             Median of the array
>>> my_array.corrcoef()             Correlation coefficient
>>> np.std(my_array)                Standard deviation
```

DataCamp
Learn Python for Data Science Interactively

Data Science Cheat Sheet

NumPy

KEY

We'll use shorthand in this cheat sheet

`arr` - A numpy Array object

IMPORTS

Import these to start

```
import numpy as np
```

IMPORTING/EXPORTING

`np.loadtxt('file.txt')` - From a text file

`np.genfromtxt('file.csv',delimiter=',')` - From a CSV file

`np.savetxt('file.txt',arr,delimiter=' ')` - Writes to a text file

`np.savetxt('file.csv',arr,delimiter=',')` - Writes to a CSV file

CREATING ARRAYS

`np.array([1,2,3])` - One dimensional array

`np.array([(1,2,3),(4,5,6)])` - Two dimensional array

`np.zeros(3)` - 1D array of length 3 all values 0

`np.ones((3,4))` - 3x4 array with all values 1

`np.eye(5)` - 5x5 array of 0 with 1 on diagonal (Identity matrix)

`np.linspace(0,100,6)` - Array of 6 evenly divided values from 0 to 100

`np.arange(0,10,3)` - Array of values from 0 to less than 10 with step 3 (eg `[0,3,6,9]`)

`np.full((2,3),8)` - 2x3 array with all values 8

`np.random.rand(4,5)` - 4x5 array of random floats between 0-1

`np.random.rand(6,7)*100` - 6x7 array of random floats between 0-100

`np.random.randint(5,size=(2,3))` - 2x3 array with random ints between 0-4

INSPECTING PROPERTIES

`arr.size` - Returns number of elements in `arr`

`arr.shape` - Returns dimensions of `arr` (rows, columns)

`arr.dtype` - Returns type of elements in `arr`

`arr.astype(dtype)` - Convert `arr` elements to type `dtype`

`arr.tolist()` - Convert `arr` to a Python list

`np.info(np.eye)` - View documentation for `np.eye`

COPYING/SORTING/RESHAPING

`np.copy(arr)` - Copies `arr` to new memory

`arr.view(dtype)` - Creates view of `arr` elements with type `dtype`

`arr.sort()` - Sorts `arr`

`arr.sort(axis=0)` - Sorts specific axis of `arr`

`two_d_arr.flatten()` - Flattens 2D array `two_d_arr` to 1D

`arr.T` - Transposes `arr` (rows become columns and vice versa)

`arr.reshape(3,4)` - Reshapes `arr` to 3 rows, 4 columns without changing data

`arr.resize((5,6))` - Changes `arr` shape to 5x6 and fills new values with 0

ADDING/REMOVING ELEMENTS

`np.append(arr,values)` - Appends `values` to end of `arr`

`np.insert(arr,2,values)` - Inserts `values` into `arr` before index 2

`np.delete(arr,3,axis=0)` - Deletes row on index 3 of `arr`

`np.delete(arr,4,axis=1)` - Deletes column on index 4 of `arr`

COMBINING/SPLITTING

`np.concatenate((arr1,arr2),axis=0)` - Adds `arr2` as rows to the end of `arr1`

`np.concatenate((arr1,arr2),axis=1)` - Adds `arr2` as columns to end of `arr1`

`np.split(arr,3)` - Splits `arr` into 3 sub-arrays

`np.hsplit(arr,5)` - Splits `arr` horizontally on the 5th index

INDEXING/SLICING/SUBSETTING

`arr[5]` - Returns the element at index 5

`arr[2,5]` - Returns the 2D array element on index `[2][5]`

`arr[1]=4` - Assigns array element on index 1 the value 4

`arr[1,3]=10` - Assigns array element on index `[1][3]` the value 10

`arr[0:3]` - Returns the elements at indices `0,1,2` (On a 2D array: returns rows `0,1,2`)

`arr[0:3,4]` - Returns the elements on rows `0,1,2` at column 4

`arr[:2]` - Returns the elements at indices `0,1` (On a 2D array: returns rows `0,1`)

`arr[:,1]` - Returns the elements at index 1 on all rows

`arr<5` - Returns an array with boolean values

`(arr1<3) & (arr2>5)` - Returns an array with boolean values

`~arr` - Inverts a boolean array

`arr[arr<5]` - Returns array elements smaller than 5

SCALAR MATH

`np.add(arr,1)` - Add 1 to each array element

`np.subtract(arr,2)` - Subtract 2 from each array element

`np.multiply(arr,3)` - Multiply each array element by 3

`np.divide(arr,4)` - Divide each array element by 4 (returns `np.nan` for division by zero)

`np.power(arr,5)` - Raise each array element to the 5th power

VECTOR MATH

`np.add(arr1,arr2)` - Elementwise add `arr2` to `arr1`

`np.subtract(arr1,arr2)` - Elementwise subtract `arr2` from `arr1`

`np.multiply(arr1,arr2)` - Elementwise multiply `arr1` by `arr2`

`np.divide(arr1,arr2)` - Elementwise divide `arr1` by `arr2`

`np.power(arr1,arr2)` - Elementwise raise `arr1` raised to the power of `arr2`

`np.array_equal(arr1,arr2)` - Returns `True` if the arrays have the same elements and shape

`np.sqrt(arr)` - Square root of each element in the array

`np.sin(arr)` - Sine of each element in the array

`np.log(arr)` - Natural log of each element in the array

`np.abs(arr)` - Absolute value of each element in the array

`np.ceil(arr)` - Rounds up to the nearest int

`np.floor(arr)` - Rounds down to the nearest int

`np.round(arr)` - Rounds to the nearest int

STATISTICS

`np.mean(arr,axis=0)` - Returns mean along specific axis

`arr.sum()` - Returns sum of `arr`

`arr.min()` - Returns minimum value of `arr`

`arr.max(axis=0)` - Returns maximum value of specific axis

`np.var(arr)` - Returns the variance of array

`np.std(arr,axis=1)` - Returns the standard deviation of specific axis

`arr.corrcoef()` - Returns correlation coefficient of array

Numpy Cheat Sheet

Python Package

Created By: Arianne Colton and Sean Chen

Numpy (Numerical Python)

What is NumPy?

Foundation package for scientific computing in Python

Why NumPy?

- Numpy **'ndarray'** is a much more efficient way of storing and manipulating **"numerical data"** than the built-in Python data structures.
- Libraries written in lower-level languages, such as C, can operate on data stored in Numpy **'ndarray'** without copying any data.

N-DIMENSIONAL ARRAY (NDARRAY)

What is NdArray?

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

Create NdArray	`np.array(seq1)` # seq1 - is any sequence like object, i.e. [1, 2, 3]

Create Special NdArray	`1, np.zeros(10)`
	# one dimensional ndarray with 10 elements of value 0
	`2, np.ones(2, 3)`
	# two dimensional ndarray with 6 elements of value 1
	`3, np.empty(3, 4, 5) *`
	# three dimensional ndarray of uninitialized values
	`4, np.eye(N) or np.identity(N)`
	# creates N by N identity matrix

NdArray version of Python's `range`	`np.arange(1, 10)`
Get # of Dimension	`ndarray1.ndim`
Get Dimension Size	`dim1size, dim2size, .. = ndarray1.shape`
Get Data Type **	`ndarray1.dtype`
Explicit Casting	`ndarray2 = ndarray1.astype(np.int32) ***`

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

- ** Default data type is **'np.float64'**. This is equivalent to Python's float type which is 8 bytes (64 bits); thus the name 'float64'.
- *** If casting were to fail for some reason, **'TypeError'** will be raised.

SLICING (INDEXING/SUBSETTING)

- Slicing (i.e. `ndarray1[2:6]`) is a **'view'** on the original array. **Data is NOT copied**. Any modifications (i.e. `ndarray1[2:6] = 8`) to the 'view' will be reflected in the original array.
- Instead of a 'view', explicit copy of slicing via :

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

- Multidimensional array indexing notation :

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

* Boolean indexing :

```
ndarray1[(names == 'Bob') | (names == 'Will'), 2:]
# '2:' means select from 3rd column on
```

- Selecting data by boolean indexing **ALWAYS** creates a copy of the data.
- The 'and' and 'or' keywords do NOT work with boolean arrays. Use & and |.

* Fancy indexing (aka 'indexing using integer arrays')

Select a subset of rows in a particular order :

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

- Fancy indexing **ALWAYS** creates a copy of the data.

Numpy (Numerical Python)

Setting data with assignment :

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

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

COMMON OPERATIONS

1. Transposing

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

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

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

- `math.sqrt()` works on only a `scalar`

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

3. Vectorized expressions

- `np.where(cond, x, y)` is a vectorized version of the expression 'x if condition else y'

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

- Common Usages :

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

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

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

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

Compute mean	`ndarray1.mean()` or `np.mean(ndarray1)`
Compute statistics over axis *	`ndarray1.mean(axis = 1)` `ndarray1.sum(axis = 0)`

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

5. Boolean arrays methods

Count # of 'Trues' in boolean array	`(ndarray1 > 0).sum()`
If at least one value is 'True'	`ndarray1.any()`
If all values are 'True'	`ndarray1.all()`

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

6. Sorting

Inplace sorting	`ndarray1.sort()`
Return a sorted copy instead of inplace	`sorted1 = np.sort(ndarray1)`

7. Set methods

Return sorted unique values	`np.unique(ndarray1)`
Test membership of ndarray1 values in [2, 3, 6]	`resultBooleanArray = np.in1d(ndarray1, [2, 3, 6])`

- Other set methods : `intersect1d()`,`union1d()`, `setdiff1d()`,`setxor1d()`

8. Random number generation (np.random)

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

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

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

Created by Arianne Colton and Sean Chen
www.datasciencefree.com
Based on content from
'Python for Data Analysis' by Wes McKinney

Updated: August 18, 2016

Python For Data Science *Cheat Sheet*

NumPy Basics

Learn Python for Data Science Interactively at www.DataCamp.com

NumPy

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

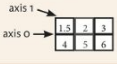
Use the following import convention:

```
>>> import numpy as np
```

NumPy Arrays

1D array

2D array



3D array

Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

```
>>> np.zeros((3,4))
>>> np.ones((2,3,4),dtype=np.int16)
>>> d = np.arange(10,25,5)
>>> np.linspace(0,2,9)
>>> e = np.full((2,2),7)
>>> f = np.eye(2)
>>> np.random.random((2,2))
>>> np.empty((3,2))
```

Create an array of zeros
Create an array of ones
Create an array of evenly spaced values (step value)
Create an array of evenly spaced values (number of samples)
Create a constant array
Create a 2X2 identity matrix
Create an array with random values
Create an empty array

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

```
>>> np.int64
>>> np.float32
>>> np.complex
>>> np.bool
>>> np.object
>>> np.string_
>>> np.unicode_
```

Signed 64-bit integer types
Standard double-precision floating point
Complex numbers represented by 128 floats
Boolean type storing `TRUE` and `FALSE` values
Python object type
Fixed-length string type
Fixed-length unicode type

Inspecting Your Array

```
>>> a.shape
>>> len(a)
>>> b.ndim
>>> e.size
>>> b.dtype
>>> b.dtype.name
>>> b.astype(int)
```

Array dimensions
Length of array
Number of array dimensions
Number of array elements
Data type of array elements
Name of data type
Convert an array to a different type

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

```
>>> g = a - b
 array([[-0.5,  0. ,  0. ],
        [-3. , -3. , -3. ]])
>>> np.subtract(a,b)
>>> b + a
 array([[ 2.5,  4. ,  6. ],
        [ 5. ,  7. ,  9. ]])
>>> np.add(b,a)
>>> a / b
 array([[ 0.66666667,  1.        ,  1.        ],
        [ 0.25      ,  0.4       ,  0.5       ]])
>>> np.divide(a,b)
>>> a * b
 array([[  1.5,   4. ,   9. ],
        [  4. ,  10. ,  18. ]])
>>> np.multiply(a,b)
>>> np.exp(b)
>>> np.sqrt(b)
>>> np.sin(a)
>>> np.cos(b)
>>> np.log(a)
>>> e.dot(f)
 array([[ 7.,  7.],
        [ 7.,  7.]])
```

Subtraction
Subtraction
Addition
Addition
Division
Division
Multiplication
Multiplication
Exponentiation
Square root
Print sines of an array
Element-wise cosine
Element-wise natural logarithm
Dot product

Comparison

```
>>> a == b
 array([[False,  True,  True],
        [False, False, False]], dtype=bool)
>>> a < 2
 array([True, False, False], dtype=bool)
>>> np.array_equal(a, b)
```

Element-wise comparison
Element-wise comparison
Array-wise comparison

Aggregate Functions

```
>>> a.sum()
>>> a.min()
>>> b.max(axis=0)
>>> b.cumsum(axis=1)
>>> a.mean()
>>> b.median()
>>> a.corrcoef()
>>> np.std(b)
```

Array-wise sum
Array-wise minimum value
Maximum value of an array row
Cumulative sum of the elements
Mean
Median
Correlation coefficient
Standard deviation

Copying Arrays

```
>>> h = a.view()
>>> np.copy(a)
>>> h = a.copy()
```

Create a view of the array with the same data
Create a copy of the array
Create a deep copy of the array

Sorting Arrays

```
>>> a.sort()
>>> c.sort(axis=0)
```

Sort an array
Sort the elements of an array's axis

Subsetting, Slicing, Indexing

Also see Lists

Subsetting

```
>>> a[2]
 3
>>> b[1,2]
 6.0
```

Select the element at the 2nd index
Select the element at row 0 column 2 (equivalent to `b[1][2]`)

Slicing

```
>>> a[0:2]
 array([1, 2])
>>> b[0:2,1]
 array([ 2.,  5.])
>>> b[:1]
 array([[1.5, 2., 3.]])
>>> c[1,...]
 array([[[ 3.,  2.,  1.],
         [ 4.,  5.,  6.]]])
>>> a[ : :-1]
 array([3, 2, 1])
```

Select items at index 0 and 1
Select items at rows 0 and 1 in column 1
Select all items at row 0 (equivalent to `b[0:1, :]`)
Same as `[1,:,:]`
Reversed array `a`

Boolean Indexing

```
>>> a[a<2]
 array([1])
```

Select elements from `a` less than 2

Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]
 array([ 4. ,  2. ,  6. ,  1.5])
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]
 array([[ 4. ,  5. ,  6. ,  4. ],
        [ 1.5,  2. ,  3. ,  1.5],
        [ 4. ,  5. ,  6. ,  4. ],
        [ 1.5,  2. ,  3. ,  1.5]])
```

Select elements `(1,0),(0,1),(1,2)` and `(0,0)`
Select a subset of the matrix's rows and columns

Array Manipulation

Transposing Array

```
>>> i = np.transpose(b)
>>> i.T
```

Permute array dimensions
Permute array dimensions

Changing Array Shape

```
>>> b.ravel()
>>> g.reshape(3,-2)
```

Flatten the array
Reshape, but don't change data

Adding/Removing Elements

```
>>> h.resize((2,6))
>>> np.append(h,g)
>>> np.insert(a, 1, 5)
>>> np.delete(a,[1])
```

Return a new array with shape (2,6)
Append items to an array
Insert items in an array
Delete items from an array

Combining Arrays

```
>>> np.concatenate((a,d),axis=0)
 array([ 1,  2,  3, 10, 15, 20])
>>> np.vstack((a,b))
 array([[ 1. ,  2. ,  3. ],
        [ 1.5,  2. ,  3. ],
        [ 4. ,  5. ,  6. ]])
>>> np.r_[e,f]
>>> np.hstack((e,f))
 array([[ 7.,  7.,  1.,  0.],
        [ 7.,  7.,  0.,  1.]])
>>> np.column_stack((a,d))
 array([[ 1, 10],
        [ 2, 15],
        [ 3, 20]])
>>> np.c_[a,d]
```

Concatenate arrays
Stack arrays vertically (row-wise)
Stack arrays vertically (row-wise)
Stack arrays horizontally (column-wise)
Create stacked column-wise arrays
Create stacked column-wise arrays

Splitting Arrays

```
>>> np.hsplit(a,3)
 [array([1]),array([2]),array([3])]
>>> np.vsplit(c,2)
[array([[[ 1.5,  2. ,  1. ],
         [ 4. ,  5. ,  6. ]]]),
 array([[[ 3. ,  2. ,  3. ],
         [ 4. ,  5. ,  6. ]]])]
```

Split the array horizontally at the 3rd index
Split the array vertically at the 2nd index

Data Analysis with PANDAS

CHEAT SHEET

Created By: Arianne Colton and Sean Chen

Data Structures

SERIES (1D)

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

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

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

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

DATAFRAME (2D)

Tabular data structure with ordered collections of columns, each of which can be different value type.

Data Frame (DF) can be thought of as a dict of Series.

Create DF
(from a dict of equal-length lists or NumPy arrays)
	`dict1 = {'state': ['Ohio', 'CA'], 'year': [2000, 2010]}`
	`df1 = pd.DataFrame(dict1)`
	# columns are placed in sorted order
	`df1 = pd.DataFrame(dict1, index = ['row1', 'row2'])`
	# specifying index
	`df1 = pd.DataFrame(dict1, columns = ['year', 'state'])`
	# columns are placed in your given order

* Create DF (from nested dict of dicts) The inner keys as row indices	`dict1 = {'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 3, 'row2': 4} }` `df1 = pd.DataFrame(dict1)`

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

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

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

PANEL DATA (3D)

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

```
import pandas_datareader.data as web
panel1 = pd.Panel({stk : web.get_data_
yahoo(stk, '1/1/2000', '1/1/2010')
for stk in ['AAPL', 'IBM']})
# panel1 Dimensions : 2 (item) * 861 (major) * 6 (minor)
```

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

```
panel1 = panel1.swapaxes('item', 'minor')
panel1.ix[:, '6/1/2003', :].to_frame() *
```

=> Stacked DF (with **hierarchical indexing** **) :

#		Open	High	Low	Close	Volume	Adj-Close
# major	minor						
# 2003-06-01	AAPL						
#	IBM						
# 2003-06-02	AAPL						
#	IBM						

Data Structures continued

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

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

INDEX OBJECTS

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

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

HIERARCHICAL INDEXING

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

MultiIndex :

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

Series Partial Indexing	`series1['b']` # Outer Level `series1[:, 2]` # Inner Level

DF Partial Indexing	`df1['outerCol3','InnerCol2']`
	Or
	`df1['outerCol3']['InnerCol2']`

Swaping and Sorting Levels

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

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

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

Summary Statistics by Level

Most stats functions in DF or Series have a "level" option that you can specify the level you want on an axis.

Sum rows (that have same 'key2' value)	`df1.sum(level = 'key2')`
Sum columns ..	`df1.sum(level = 'col3', axis = 1)`

- Under the hood, the functionality provided here utilizes panda's **'groupby'**.

DataFrame's Columns as Indexes

DF's "set_index" will create a new DF using one or more of its columns as the index.

New DF using columns as index	`df2 = df1.set_index(['col3', 'col4'])` * ‡ # col3 becomes the outermost index, col4 becomes inner index. Values of col3, col4 become the index values.

* "reset_index" does the opposite of "set_index", the hierarchical index are moved into columns.

‡ By default, 'col3' and 'col4' will be removed from the DF, though you can leave them by option : `'drop = False'`.

Missing Data

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

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

FILTERING OUT MISSING DATA

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

`df1.dropna()` # drop any row containing missing value	
`df1.dropna(axis = 1)` # drop any column containing missing values	

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

FILLING IN MISSING DATA

`df2 = df1.fillna(0)` # fill all missing data with 0	
`df1.fillna('inplace = True')` # modify in-place	
Use a different fill value for each column :	
`df1.fillna({'col1' : 0, 'col2' : -1})`	
Only forward fill the 2 missing values in front :	
`df1.fillna(method = 'ffill', limit = 2)`	
i.e. for column1, if row 3-6 are missing. so 3 and 4 get filled with the value from 2, NOT 5 and 6.	

Python For Data Science *Cheat Sheet*

Pandas Basics

Learn Python for Data Science Interactively at www.DataCamp.com

Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis** tools for the Python programming language.

Use the following import convention:

```
>>> import pandas as pd
```

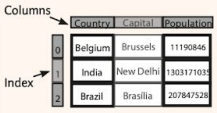
Pandas Data Structures

Series

A **one-dimensional** labeled array capable of holding any data type


```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame



A **two-dimensional** labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
            'Population': [11190846, 1303171035, 207847528]}

>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

```
>>> s['b']
 -5
```
Get one element

```
>>> df[1:]
  Country    Capital  Population
1   India  New Delhi  1303171035
2  Brazil   Brasilia   207847528
```
Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[[0],[0]]
'Belgium'
```
Select single value by row & column

```
>>> df.iat([0],[0])
'Belgium'
```

By Label

```
>>> df.loc[[0], ['Country']]
'Belgium'
```
Select single value by row & column labels

```
>>> df.at([0], ['Country'])
'Belgium'
```

By Label/Position

```
>>> df.ix[2]
Country      Brazil
Capital    Brasilia
Population  207847528
```
Select single row of subset of rows

```
>>> df.ix[:,'Capital']
0      Brussels
1    New Delhi
2      Brasilia
```
Select a single column of subset of columns

```
>>> df.ix[1,'Capital']
'New Delhi'
```
Select rows and columns

Boolean Indexing

```
>>> s[~(s > 1)]
```
Series `s` where value is not >1

```
>>> s[(s < -1) | (s > 2)]
```
`s` where value is <-1 or >2

```
>>> df[df['Population']>1200000000]
```
Use filter to adjust DataFrame

Setting

```
>>> s['a'] = 6
```
Set index `a` of Series `s` to 6

Dropping

```
>>> s.drop(['a', 'c'])
```
Drop values from rows (axis=0)

```
>>> df.drop('Country', axis=1)
```
Drop values from columns(axis=1)

Sort & Rank

```
>>> df.sort_index()
```
Sort by labels along an axis

```
>>> df.sort_values(by='Country')
```
Sort by the values along an axis

```
>>> df.rank()
```
Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
```
(rows,columns)

```
>>> df.index
```
Describe index

```
>>> df.columns
```
Describe DataFrame columns

```
>>> df.info()
```
Info on DataFrame

```
>>> df.count()
```
Number of non-NA values

Summary

```
>>> df.sum()
```
Sum of values

```
>>> df.cumsum()
```
Cummulative sum of values

```
>>> df.min()/df.max()
```
Minimum/maximum values

```
>>> df.idxmin()/df.idxmax()
```
Minimum/Maximum index value

```
>>> df.describe()
```
Summary statistics

```
>>> df.mean()
```
Mean of values

```
>>> df.median()
```
Median of values

Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
```
Apply function

```
>>> df.applymap(f)
```
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
a    10.0
b     NaN
c     5.0
d     7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
a    10.0
b    -5.0
c     5.0
d     7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> df.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
```

Read multiple sheets from the same file

```
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

`read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()`

```
>>> df.to_sql('myDf', engine)
```

Python For Data Science *Cheat Sheet*

Matplotlib

Learn Python Interactively at www.DataCamp.com

Matplotlib

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

matplotlib

1 Prepare The Data

Also see Lists & NumPy

1D Data

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

2D Data or Images

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

2 Create Plot

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

Figure

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

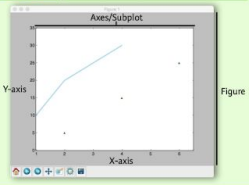
Axes

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

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

Plot Anatomy & Workflow

Plot Anatomy



Workflow

The basic steps to creating plots with matplotlib are:

1 Prepare data 2 Create plot 3 Plot 4 Customize plot 5 Save plot 6 Show plot

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

4 Customize Plot

Colors, Color Bars & Color Maps

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

Markers

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

Linestyles

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

Text & Annotations

```
>>> ax.text(1,
            -2.1,
            'Example Graph',
            style='italic')
>>> ax.annotate("Sine",
                xy=(8, 0),
                xycoords='data',
                xytext=(10.5, 0),
                textcoords='data',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="arc3"),)
```

Mathtext

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

Limits, Legends & Layouts

Limits & Autoscaling

```
>>> ax.margins(x=0.0,y=0.1)                  Add padding to a plot
>>> ax.axis('equal')                         Set the aspect ratio of the plot to 1
>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5])    Set limits for x-and y-axis
>>> ax.set_xlim(0,10.5)                      Set limits for x-axis
```

Legends

```
>>> ax.set(title='An Example Axes',          Set a title and x-and y-axis labels
           ylabel='Y-Axis',
           xlabel='X-Axis')
>>> ax.legend(loc='best')                    No overlapping plot elements
```

Ticks

```
>>> ax.xaxis.set(ticks=range(1,5),           Manually set x-ticks
                 ticklabels=[3,100,-12,"foo"])
>>> ax.tick_params(axis='y',                 Make y-ticks longer and go in and out
                   direction='inout',
                   length=10)
```

Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5,         Adjust the spacing between subplots
                         hspace=0.3,
                         left=0.125,
                         right=0.9,
                         top=0.9,
                         bottom=0.1)
>>> fig.tight_layout()                       Fit subplot(s) in to the figure area
```

Axis Spines

```
>>> ax1.spines['top'].set_visible(False)     Make the top axis line for a plot invisible
>>> ax1.spines['bottom'].set_position(('outward',10)) Move the bottom axis line outward
```

3 Plotting Routines

1D Data

```
>>> lines = ax.plot(x,y)                     Draw points with lines or markers connecting them
>>> ax.scatter(x,y)                          Draw unconnected points, scaled or colored
>>> axes[0,0].bar([1,2,3],[3,4,5])           Plot vertical rectangles (constant width)
>>> axes[1,0].barh([0.5,1,2.5],[0,1,2])      Plot horiontal rectangles (constant height)
>>> axes[1,1].axhline(0.45)                  Draw a horizontal line across axes
>>> axes[0,1].axvline(0.65)                  Draw a vertical line across axes
>>> ax.fill(x,y,color='blue')                Draw filled polygons
>>> ax.fill_between(x,y,color='yellow')      Fill between y-values and o
```

Vector Fields

```
>>> axes[0,1].arrow(0,0,0.5,0.5)             Add an arrow to the axes
>>> axes[1,1].quiver(y,z)                    Plot a 2D field of arrows
>>> axes[0,1].streamplot(X,Y,U,V)            Plot 2D vector fields
```

Data Distributions

```
>>> ax1.hist(y)                              Plot a histogram
>>> ax3.boxplot(y)                           Make a box and whisker plot
>>> ax3.violinplot(z)                        Make a violin plot
```

2D Data or Images

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

```
>>> axes2[0].pcolor(data2)                   Pseudocolor plot of 2D array
>>> axes2[0].pcolormesh(data)                Pseudocolor plot of 2D array
>>> CS = plt.contour(Y,X,U)                  Plot contours
>>> axes2[2].contourf(data1)                 Plot filled contours
>>> axes2[2]= ax.clabel(CS)                  Label a contour plot
```

5 Save Plot

Save figures

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

Save transparent figures

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

6 Show Plot

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

Close & Clear

```
>>> plt.cla()                                Clear an axis
>>> plt.clf()                                Clear the entire figure
>>> plt.close()                              Close a window
```

Python For Data Science *Cheat Sheet*

Scikit-Learn

Learn Python for data science Interactively at www.DataCamp.com

Scikit-learn

Scikit-learn is an open source Python library that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms using a unified interface.

A Basic Example

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=33)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

Loading The Data

Also see NumPy & Pandas

Your data needs to be numeric and stored as NumPy arrays or SciPy sparse matrices. Other types that are convertible to numeric arrays, such as Pandas DataFrame, are also acceptable.

```
>>> import numpy as np
>>> X = np.random.random((10,5))
>>> y = np.array(['M','M','F','F','M','F','M','M','F','F','F'])
>>> X[X < 0.7] = 0
```

Training And Test Data

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X,
                                                        y,
                                                        random_state=0)
```

Create Your Model

Supervised Learning Estimators

Linear Regression

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression(normalize=True)
```

Support Vector Machines (SVM)

```
>>> from sklearn.svm import SVC
>>> svc = SVC(kernel='linear')
```

Naive Bayes

```
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
```

KNN

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
```

Unsupervised Learning Estimators

Principal Component Analysis (PCA)

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=0.95)
```

K Means

```
>>> from sklearn.cluster import KMeans
>>> k_means = KMeans(n_clusters=3, random_state=0)
```

Model Fitting

Supervised learning
`>>> lr.fit(X, y)`
`>>> knn.fit(X_train, y_train)`
`>>> svc.fit(X_train, y_train)`
	Fit the model to the data

Unsupervised Learning
`>>> k_means.fit(X_train)`
`>>> pca_model = pca.fit_transform(X_train)`
	Fit the model to the data
	Fit to data, then transform it

Prediction

Supervised Estimators
`>>> y_pred = svc.predict(np.random.random((2,5)))`
`>>> y_pred = lr.predict(X_test)`
`>>> y_pred = knn.predict_proba(X_test)`
	Predict labels
	Predict labels
	Estimate probability of a label

Unsupervised Estimators `>>> y_pred = k_means.predict(X_test)`	Predict labels in clustering algos

Preprocessing The Data

Standardization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)
```

Normalization

```
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer().fit(X_train)
>>> normalized_X = scaler.transform(X_train)
>>> normalized_X_test = scaler.transform(X_test)
```

Binarization

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X)
```

Encoding Categorical Features

```
>>> from sklearn.preprocessing import LabelEncoder
>>> enc = LabelEncoder()
>>> y = enc.fit_transform(y)
```

Imputing Missing Values

```
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values=0, strategy='mean', axis=0)
>>> imp.fit_transform(X_train)
```

Generating Polynomial Features

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly = PolynomialFeatures(5)
>>> poly.fit_transform(X)
```

Evaluate Your Model's Performance

Classification Metrics

Accuracy Score — Estimator score method

```
>>> knn.score(X_test, y_test)
```

Metric scoring functions

```
>>> from sklearn.metrics import accuracy_score
>>> accuracy_score(y_test, y_pred)
```

Classification Report — Precision, recall, f1-score and support

```
>>> from sklearn.metrics import classification_report
>>> print(classification_report(y_test, y_pred))
```

Confusion Matrix

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(y_test, y_pred))
```

Regression Metrics

Mean Absolute Error

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2]
>>> mean_absolute_error(y_true, y_pred)
```

Mean Squared Error

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(y_test, y_pred)
```

R^2 Score

```
>>> from sklearn.metrics import r2_score
>>> r2_score(y_true, y_pred)
```

Clustering Metrics

Adjusted Rand Index

```
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(y_true, y_pred)
```

Homogeneity

```
>>> from sklearn.metrics import homogeneity_score
>>> homogeneity_score(y_true, y_pred)
```

V-measure

```
>>> from sklearn.metrics import v_measure_score
>>> metrics.v_measure_score(y_true, y_pred)
```

Cross-Validation

```
>>> from sklearn.cross_validation import cross_val_score
>>> print(cross_val_score(knn, X_train, y_train, cv=4))
>>> print(cross_val_score(lr, X, y, cv=2))
```

Tune Your Model

Grid Search

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {"n_neighbors": np.arange(1,3),
              "metric": ["euclidean", "cityblock"]}
>>> grid = GridSearchCV(estimator=knn,
                        param_grid=params)
>>> grid.fit(X_train, y_train)
>>> print(grid.best_score_)
>>> print(grid.best_estimator_.n_neighbors)
```

Randomized Parameter Optimization

```
>>> from sklearn.grid_search import RandomizedSearchCV
>>> params = {"n_neighbors": range(1,5),
              "weights": ["uniform", "distance"]}
>>> rsearch = RandomizedSearchCV(estimator=knn,
                                 param_distributions=params,
                                 cv=4,
                                 n_iter=8,
                                 random_state=5)
>>> rsearch.fit(X_train, y_train)
>>> print(rsearch.best_score_)
```

DataCamp
Learn Python For Data Science Interactively