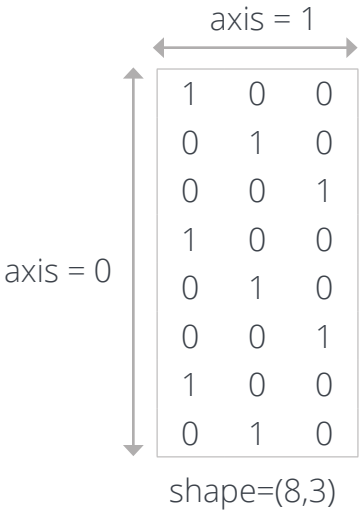
TOOL

NumPy Array Tip Sheet

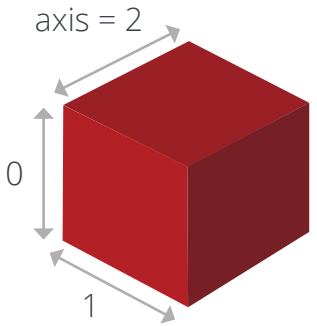
Arrays are the central data type introduced in the NumPy package. Technically, array objects are of type **`numpy.ndarray`**, which stands for "n-dimensional array." Arrays are accessible by importing the NumPy module, although we will use the conventional shorthand here: **`import numpy as np`**. Arrays are similar in some respects to Python lists but are multidimensional, homogeneous in type, and support compact and efficient array-level manipulations. Documentation can be found online at www.numpy.org/doc.

Anatomy of an array



The **axes** of an array describe the order of indexing into the array; e.g., axis=0 refers to the first index coordinate, axis=1 the second, etc.

The **shape** of an array is a tuple indicating the number of elements along each axis. An existing array **a** has an attribute **a.shape** which contains this tuple.



- All elements must be the same dtype (data type).
- The default dtype is float.
- Arrays constructed from a list of mixed dtype will be upcast to the "greatest" common type.

Constructing arrays

- `np.array(alist):` Construct an n-dimensional array from a Python list (all elements of list must be of same length).

```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([i*i for i in range(100) if
i%2==1])
c = b.tolist()                          # convert array back to Python list
```

- `np.zeros(shape, dtype=float):` Construct an n-dimensional array of the specified shape, filled with zeros of the specified dtype.

```
a = np.zeros(100)                       # a 100-element array of float zeros
b = np.zeros((2,8), int)                # a 2x8 array of int zeros
c = np.zeros((N,M,L), bool)             # a NxMxL array of bool zeros
```

- `np.ones(shape, dtype=float):` Construct an n-dimensional array of the specified shape, filled with ones of the specified dtype.

```
a = np.ones(10, int)                    # a 10-element array of int ones
b = np.pi * np.ones((5,5))              # a useful way to fill up an array with a specified value
```

- `np.transpose(a)`

```
b = np.transpose(a)                     # return new array with a's dimensions reversed
b = a.T                                 # equivalent to np.transpose(a)
```

- np.arange and np.linspace

```
a = np.arange(start, stop, increment)        # like Python range, but with (potentially) real-valued arrays
b = np.linspace(start, stop, num_elements)   # create array of equally spaced points based on specifed number of points
```

- Random array constructors in np.random

```
a = np.random.random((100,100))              # 100x100 array of floats uniform on [0.,1.)
b = np.random.randint(0,10, (100,))          # 100 random ints uniform on [0, 10); i.e., not including the upper bound 10
c = np.random.standard_normal((5,5,5))       # zero-mean, unit-variance Gaussian random numbers in a 5x5x5 array
```

Indexing arrays

- Multidimensional indexing

```
elem = a[i,j,k]                              # extract element at position (i,j,k) in array
```

- "Negative" indexing (wrap around the end of the array)

```
last = a[-1]                                 # the last element of the array (returns array if a.ndim > 1)
last_elem = a[-1,-1]                         # the lower-right element of an array (returns array if a.ndim > 2)
```

- Arrays as indices

```
i = np.array([0,1,2,1])                    # array of indices for the first axis
j = np.array([1,2,3,4])                    # array of indices for the second axis
a[i,j]                                     # return array([a[0,1], a[1,2], a[2,3], a[1,4]]) based on
#                                            i and j above
b = np.array([True, False, True, False])
a[b]                                       # return array([a[0], a[2]]) since only b[0] and b[2] are
                                             True
```

Slicing arrays (extracting subsections)

- Slice a defined subblock

```
section = a[10:20, 30:40]                  # 10x10 subblock starting at [10,30]
```

- Grab everything up to the beginning/end of the array

```
asection = a[10:, 30:]                     # missing stop index implies until end of array
bsection = b[:10, :30]                     # missing start index implies until start of array
```

- Grab an entire column

```
x = a[:, 0]                                # get everything in the 0th column (missing start and
                                             stop)
y = a[:, 1]                                # get everything in the 1st column
```

- Slice off the tail end of an array

```
tail = a[-10:]                             # grab the last 10 elements of the array
slab = b[:, -10:]                          # grab a slab of width 10 off the “side” of the array
interior = c[1:-1, 1:-1, 1:-1]             # slice out everything but the outer shell
```

Element-wise functions on arrays

- Arithmetic operations

```
c = a + b                                  # add a and b element-wise (must be same shape)
d = c + 2                                  # add 2 to every element of c
h = e * f                                  # multiply e and f element-wise (NOT matrix multiplication)
g = -h                                     # negate every element of h
m = c ** 2                                 # compute the square of every element of c
z = w > 0.0                                # return Boolean array indicating which elements are > 0.0
logspace = 10**np.linspace(-6, -1, 50)     # array of 50 equally spaced-in log points between 1.0e-06 and 1.0e-01
```

- Trigonometric operations

```
y = np.sin(x)                              # sin of every element of x
w = np.cos(2*np.pi*x)                      # cos of 2*pi*x
```

Summation of arrays

- Simple sums

```
s  = np.sum(a)                             # sum all elements in a, returning a scalar (single number)
s0 = np.sum(a, axis=0)                     # sum elements along specified axis (=0), returning an array of remaining shape
s1 = np.sum(a, axis=1)                     # sum elements along axis=1; i.e., over rows if a is 2-dimensional
```

- Averaging, etc.

```
m = np.mean(a, axis)                    # compute mean along the specified axis (over entire
                                          array if axis=None)
s = np.std(a, axis)                     # compute standard deviation along the specified axis
                                          (over entire array if axis=None)
```

- Cumulative sums

```
s0 = np.cumsum(a, axis=0)               # cumulatively sum over 0 axis, returning array with
                                          same shape as a
s0 = np.cumsum(a)                       # cumulatively sum over 0 axis, returning 1D array of
                                          length shape[0]*shape[1]*...*shape[dim-1]
```

Some other useful functions and methods

Many of these work both as separate functions (e.g., `np.sum(a)`) as well as array methods (e.g., `a.sum()`).

- `np.any(a):` Return True if any element of a is True.
- `np.all(a):` Return True if all elements of a are True.
- `np.concatenate((a1, a2, ...), axis):` Concatenate tuple of arrays along specified axis.
- `np.min(a, axis=None), np.max(a, axis=None):` Get min/max values of a along specified axis (global min/max if axis=None).
- `np.argmin(a, axis=None), np.argmax(a, axis=None):` Get indices of min/max of a along specified axis (global min/max if axis=None).
- `np.reshape(a, newshape):` Reshape a to new shape (must conserve total number of elements).
- `np.histogram, np.histogram2d, np.histogramdd:` 1-dimensional, 2-dimensional, and d-dimensional histograms, respectively.
- `np.round(a, decimals=0):` Round elements of array a to specified number of decimals.
- `np.sign(a):` Return array of same shape as a, with -1 where a < 0, 0 where a = 0, and +1 where a > 0.
- `np.abs(a):` Return array of same shape as a, with the absolute value of each corresponding element.
- `np.unique(a):` Return sorted unique elements of array a.
- `np.where(condition, x, y):` Return array with same shape as condition, where values from x are inserted in positions where condition is True, and values from y where condition is False.

PRACTICE NUMPY

BROADCASTING: Arrays w/ diff sizes can't be +/- or generally in arithmetic means. To overcome this, we duplicate the smaller array so that it's the same dimensionality and size as the larger array. This is available in Numpy when performing array arithmetic.

VECTORIZATION: Numpy provides functions that can perform mathematical operations on arrays that would otherwise be performed through the use of for loops. Essentially, instead of using loops to perform certain computations, you can use Numpy functions.

Importing numpy using its conventional shortname np. You can choose your own shortname.

```
import numpy as np
```

numpy.arange() method: Returns evenly spaced values within a given interval

Values are generated within the [start, stop) : excluding the stop

```
numpy.arange([start,] stop, [step,] dtype=None, *, like=None)
```

- start: Integer or real [OPTIONAL]
- stop: integer or real
- step: Integer or real [OPTIONAL]
- dtype: The type of the output array. If it's not given, infer the data type from the other input arguments.
- like: Reference objects allow the creation of arrays which are not NumPy arrays.

Ex) `np.arange(0, 5, 0.5, dtype=int)` # `array([0,0,0,0,0,0,0,0,0,0])`

Ex) `np.arange(-3, 3, 0.5, dtype=int)` # `array([-3,-2,-1,0,1,2,3,4,5,6,7,8])`

Ex) `np.arange(3)` # `array([0,1,2])`

Ex) `np.arange(3.0)` # `array([0., 1., 2.])`

Ex) `np.arange(3,7)` # `array([3,4,5,6])`

Ex) `np.arange(3,7,2)` # `array([3,5])`

REMEMBER: [start, stop) — start: inclusive! stop: exclusive!

numpy.shape(): Returns the shape of an array

```
numpy.shape(a)
```

a: input array

Returns a TUPLE of ints. The elements of the shape tuple give the lengths of the corresponding array dimensions

numpy.array(): Creates an array

```
numpy.array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0, like=None)
```

- **object**: An array, any object exposing the array interface, any object whose `__array__` method returns an array or any sequence.
- **dtype**: Desired data type for the array. [OPTIONAL]
- **copy**: True = object copied. False = a copy will be made if `__array__` returns a copy. If obj is a nested sequence, or a copy is needed to satisfy any other requirements. [OPTIONAL]
- **order**: [OPTIONAL]
- **subok**: True = subclasses will be passed through. False = The returned array will be forced to be a base-class array [default]. [OPTIONAL]
- **ndmin**: specifies the min # of dimensions that the resulting array should have. 1's will be pre-pended to the shape as needed to make the requirement [int, OPTIONAL]
- **like**: Reference object to allow the creations of arrays which are NOT NumPy arrays.

Ex) `numpy_array = np.array(new_list, dtype=int)`

↳ This takes a NORMAL, REGULAR LIST called new_list & returns a numpy array w/ data type of int.

numpy.random.normal(): Draws random samples from normal (Gaussian) distribution

```
numpy.random.normal(loc=0.0, scale=1.0, size=None)
```

- **loc**: float OR array of floats. It's the mean centre of distribution.
- **scale**: float OR array of floats. standard deviation of distribution must be (+) #s
- **size**: int OR tuple of ints. output shape.

numpy.ones(): Returns a new array of given shape + type, FILLED WITH ONES.

```
numpy.ones(shape, dtype=None, order="C", *, like=None)
```

- **shape**: int or sequence of int. Ex. (2,3) Ex. 2
- **dtype**: [OPTIONAL]
- **order**: [OPTIONAL] Default: C

numpy.full(): Returns a new array of given shape + type, filled with fill_value

```
numpy.full(shape, fill_value, dtype=None, order="C", *, like=None)
```

- **shape**: int OR sequence of int. shape of the new array. Ex) (2,3) Ex) 2
- **fill_value**: scalar OR array_like
- **dtype**: [OPTIONAL]
- **order**: [OPTIONAL]

numpy.eye(): Returns 2D arrays with 1's on the diagonal & 0 everywhere else.

```
numpy.eye(N, M=None, k=0, dtype=<class 'float'>, order="C", *, like=None)
```

- **N**: int: # of rows in the output
- **M**: [OPTIONAL] int: # columns in the output. If None, defaults to N.
- **k**: [OPTIONAL] int: index of diagonal (+) = upper diagonal (-) = lower diagonal
- **dtype**: default is float [OPTIONAL]
- **order**: [OPTIONAL]

numpy.full(): Returns a new array of given shape & type filled with fill_value

```
numpy.full(shape, fill_value, dtype=None, order="C", like=None)
```

- **shape**: int OR sequence of int. Shape of new array
- **fill_value**: scalar OR array-like. Fill value

numpy.where(): Return elements chosen from x or y depending on condition

```
numpy.where(condition, [x, y, ]/)
```

- **condition**: array-like, bool. Where True, yield x. Where False, yield y.
- **x, y**: array-like. values from which to choose. x, y, and condition need to be broadcasted to some shape.

numpy.round(): Rounds an array to a given # of decimals

```
numpy.round(a, decimals=0, out=None)
```

- a: an array
- decimals: number of decimals

numpy.random.rand(): Random values at given shape

```
numpy.random.rand(d0, d1, ..., dn)
```

← Returns random values

- d0, d1, ..., dn: int: [OPTIONAL] Dimensions of array must be non-negative.

numpy.random.randint(): Random integers from low (inclusive) to high (exclusive)

```
numpy.random.randint(low, high=None, size=None, dtype=int)
```

- low: int OR array of int. Drawn from distribution
- high: int OR array of int [OPTIONAL]
- size: int OR tuple of int. output shape [OPTIONAL]

numpy.triu(): Upper triangle of an array.

```
numpy.triu(m, k=0)
```

- m: An multi-D array

numpy.unique(): Finds the unique elements of an array.

Returns the sorted unique elements of an array. There are also 3 optional outputs:

- The indicies of the input array that give the unique values.
- The indicies of the unique array that reconstruct the input array.
- The number of times each unique values comes up in the input array.

```
numpy.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None)
```

- ar: input array
- return_index: bool [OPTIONAL] True = returns the indicies of ar that result in unique array
- return_inverse: bool [OPTIONAL] True = return indices of the unique array that can be used to reconstruct ar
- return_counts: bool [OPTIONAL] True = returns the # of times each unique item appears in ar.
- axis: int OR none [OPTIONAL]

numpy.any(): Tests whether any array along axis evaluates to True

```
numpy.any(a, axis=None, out=None, keepdims=<no value>, where=<no value>)
```

- a: array-like. input array or object that can be converted into an array.
- axis: [OPTIONAL] axis where logical OR reduction is performed
- out, keepdims, where: [OPTIONAL]

numpy.mean(): Computes arithmetic mean along specified axis

```
numpy.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>, where=<no value>)
```

- a: array-like
- axis: None OR int OR tuple of int [OPTIONAL]

numpy.percentile(): Compute the q-th percentile of the data along the specified axis.

```
numpy.percentile(a, q, axis=None, overwrite_input=False, method='linear', keepdims=False, interpolation=None)
```

- a: array-like. input array or objects that can be converted into arrays.
- q: array-like of floats. percentile or sequence of percentile to compute.
- axis: int, tuple of int [OPTIONAL]. Axis or axes along which the percentiles are computed

numpy.random.choice(): Generates a random sample from a given 1D array.

```
numpy.random.choice(a, size=None, replace=True, p=None)
```

- a: 1D array-like
- size: int OR tuple of int [OPTIONAL]. output shape
- replace: [OPTIONAL] whether a sample is with/out replacement. True = a value of a can be selected multiple times
- p: 1-D like array [OPTIONAL] The probabilities associated with each entry in a.

numpy.percentile(): Compute the q-th percentile of the data along the specified axis.

numpy.percentile(a, q, axis=None, overwrite_input=False, method='linear', keepdims=False, interpolation=None)

- a: array-like — input array or objects that can be converted into arrays.
- q: array-like of float — percentile or sequence of percentile to compute.
- axis: int, tuple of int [OPTIONAL] — Axis or axes along which the percentiles are computed.

numpy.cov(): Estimates a covariance matrix, given data and weights

numpy.cov(m, y=None, rowvar=True, bias=False, ddof=None, fweights=None, aweights=None, dtype=None)

- m: 1D or 2D array containing multiple variables and observations.
- y: additional set of variables and observations. [OPTIONAL]
- rowvar: True = each row represents a variable, with observations in the columns; False = relationship is transpose; each column represents a variable, while rows contains observations [OPTIONAL]

numpy.ndarray.T : The transpose of an array

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

x.T # array([[1., 2.], [2., 4.]])

numpy.round_(): Rounds an array to a given number of decimals

numpy.round_(a, decimals=0, out=None)

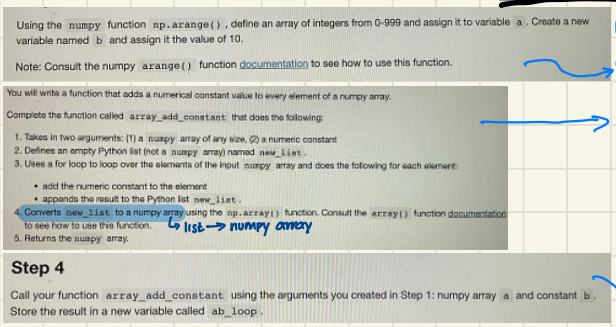
numpy.dot(): Dot product of 2 arrays

numpy.dot(a, b, out=None)

dot(a,b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])

numpy.linalg.inv(): Computes the (multiplicative) inverse of a matrix.

NUMPY



Using the `numpy` function `np.arange()`, define an array of integers from 0-999 and assign it to variable `a`. Create a new variable named `b` and assign it the value of 10.

Note: Consult the numpy `arange()` function documentation to see how to use this function.

You will write a function that adds a numerical constant value to every element of a numpy array.

Complete the function called `array_add_constant` that does the following:

1. Takes in two arguments: (1) a `numpy` array of any size, (2) a numeric constant
2. Defines an empty Python list (not a `numpy` array) named `new_list`.
3. Uses a for loop to loop over the elements of the input `numpy` array and does the following for each element:
   - add the numeric constant to the element
   - appends the result to the Python list `new_list`.
4. Converts `new_list` to a numpy array using the `np.array()` function. Consult the `array()` function documentation to see how to use this function. ↳ list → numpy array
5. Returns the `numpy` array.

Step 4

Call your function `array_add_constant` using the arguments you created in Step 1: numpy array `a` and constant `b`. Store the result in a new variable called `ab_loop`.

```
import numpy as np #imports numpy & we'll us np for short
a = np.arange(1000) #array on integers from 0-99
b = 10

def array_add_constant(numpy_array, constant):
    new_list = []
    result = 0
    for i in range(len(numpy_array)):
        result = constant + i
        new_list.append(result)

    numpy_array = np.array(new_list, dtype = int)
    return numpy_array

ab_loop = array_add_constant(a,b)
```

NOTE: notice how `numpy_array` & `ab_loop` was created

↳ We used the `array_add_constant`, a for loop, arithmetic addition to produce `ab_loop`

BUT We can do it in Numpy way:

```
import numpy as np #imports numpy & we'll us np for short
a = np.arange(1000) #array on integers from 0-99
b = 10
ab_numpy = a+b
```

`ab_loop` is equivalent to `ab_numpy`

Analyzing: `ab_numpy = a+b`

- **Broadcasting** b/c you perform a mathematical operation that involved an array & one numerical value. You added a value to each element of an array using `a+b` (arithmetic)
- **Vectorization** b/c that statement took the place of a for loop

NOTE: Vectorization speeds up performance ∴ ALWAYS USE NUMPY TECHNIQUE!

Data Wrangling

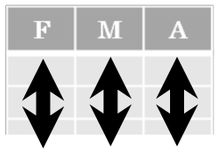
with pandas Cheat Sheet

http://pandas.pydata.org

Pandas API Reference Pandas User Guide

Tidy Data – A foundation for wrangling in pandas

In a tidy data set:

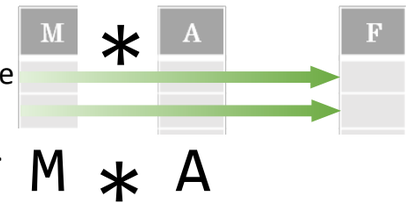


&

Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements pandas's **vectorized operations**. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.



Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4, 5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
       index = [1, 2, 3])
```

Specify values for each column.

```
df = pd.DataFrame(
    [[4, 7, 10],
     [5, 8, 11],
     [6, 9, 12]],
    index=[1, 2, 3],
    columns=['a', 'b', 'c'])
```

Specify values for each row.

		a	b	c
N	v			
D	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
       [('d', 1), ('d', 2),
        ('e', 2)], names=['n', 'v']))
```

Create DataFrame with a MultiIndex

Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

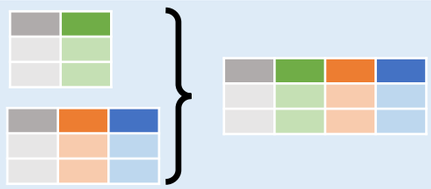
```
df = (pd.melt(df)
        .rename(columns={
                'variable':'var',
                'value':'val'})
        .query('val >= 200')
    )
```

Reshaping Data – Change layout, sorting, reindexing, renaming

pd.melt(df)
Gather columns into rows.

df.pivot(columns='var', values='val')
Spread rows into columns.

pd.concat([df1,df2])
Append rows of DataFrames



pd.concat([df1,df2], axis=1)
Append columns of DataFrames

df.sort_values('mpg')
Order rows by values of a column (low to high).

df.sort_values('mpg', ascending=False)
Order rows by values of a column (high to low).

df.rename(columns = {'y':'year'})
Rename the columns of a DataFrame

df.sort_index()
Sort the index of a DataFrame

df.reset_index()
Reset index of DataFrame to row numbers, moving index to columns.

df.drop(columns=['Length', 'Height'])
Drop columns from DataFrame

Subset Observations - rows

df[df.Length > 7]
Extract rows that meet logical criteria.

df.drop_duplicates()
Remove duplicate rows (only considers columns).

df.sample(frac=0.5)
Randomly select fraction of rows.

df.sample(n=10) Randomly select n rows.

df.nlargest(n, 'value')
Select and order top n entries.

df.nsmallest(n, 'value')
Select and order bottom n entries.

df.head(n)
Select first n rows.

df.tail(n)
Select last n rows.

Subset Variables - columns

df[['width', 'length', 'species']]
Select multiple columns with specific names.

df['width'] *or* **df.width**
Select single column with specific name.

df.filter(regex='*regex*')
Select columns whose name matches regular expression *regex*.

Using query

query() allows Boolean expressions for filtering rows.

df.query('Length > 7')

df.query('Length > 7 and Width < 8')

df.query('Name.str.startswith("abc")', engine="python")

Subsets - rows and columns

Use **df.loc[]** and **df.iloc[]** to select only rows, only columns or both.
Use **df.at[]** and **df.iat[]** to access a single value by row and column.
First index selects rows, second index columns.

df.iloc[10:20]
Select rows 10-20.

df.iloc[:, [1, 2, 5]]
Select columns in positions 1, 2 and 5 (first column is 0).

df.loc[:, 'x2':'x4']
Select all columns between x2 and x4 (inclusive).

df.loc[df['a'] > 10, ['a', 'c']]
Select rows meeting logical condition, and only the specific columns .

df.iat[1, 2] Access single value by index

df.at[4, 'A'] Access single value by label

Logic in Python (and pandas)

<	Less than	!=	Not equal to
>	Greater than	df.column.isin(*values*)	Group membership
==	Equals	pd.isnull(*obj*)	Is NaN
<=	Less than or equals	pd.notnull(*obj*)	Is not NaN
>=	Greater than or equals	&,|,~,^,df.any(),df.all()	Logical and, or, not, xor, any, all

regex (Regular Expressions) Examples

'\.'	Matches strings containing a period '.'
'Length$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]$'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
'^(?!Species$).*'	Matches strings except the string 'Species'

Cheatsheet for pandas (http://pandas.pydata.org/ originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

Summarize Data

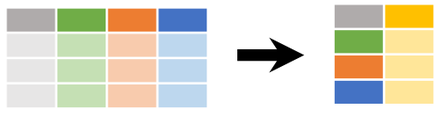
`df['w'].value_counts()`
Count number of rows with each unique value of variable

`len(df)`
# of rows in DataFrame.

`df.shape`
Tuple of # of rows, # of columns in DataFrame.

`df['w'].nunique()`
# of distinct values in a column.

`df.describe()`
Basic descriptive and statistics for each column (or GroupBy).



pandas provides a large set of summary functions that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

`sum()`
Sum values of each object.

`count()`
Count non-NA/null values of each object.

`median()`
Median value of each object.

`quantile([0.25,0.75])`
Quantiles of each object.

`apply(function)`
Apply function to each object.

`min()`
Minimum value in each object.

`max()`
Maximum value in each object.

`mean()`
Mean value of each object.

`var()`
Variance of each object.

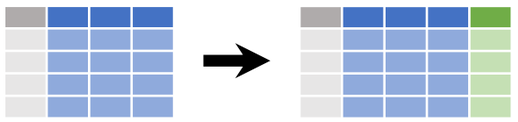
`std()`
Standard deviation of each object.

Handling Missing Data

`df.dropna()`
Drop rows with any column having NA/null data.

`df.fillna(value)`
Replace all NA/null data with value.

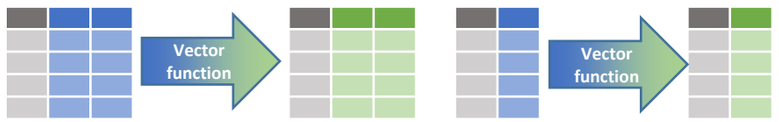
Make New Columns



`df.assign(Area=lambda df: df.Length*df.Height)`
Compute and append one or more new columns.

`df['Volume'] = df.Length*df.Height*df.Depth`
Add single column.

`pd.qcut(df.col, n, labels=False)`
Bin column into n buckets.



pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

`max(axis=1)`
Element-wise max.

`clip(lower=-10,upper=10)`
Trim values at input thresholds

`min(axis=1)`
Element-wise min.

`abs()`
Absolute value.

Group Data

`df.groupby(by="col")`
Return a GroupBy object, grouped by values in column named "col".

`df.groupby(level="ind")`
Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

`size()`
Size of each group.

`agg(function)`
Aggregate group using function.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

`shift(1)`
Copy with values shifted by 1.

`rank(method='dense')`
Ranks with no gaps.

`rank(method='min')`
Ranks. Ties get min rank.

`rank(pct=True)`
Ranks rescaled to interval [0, 1].

`rank(method='first')`
Ranks. Ties go to first value.

`shift(-1)`
Copy with values lagged by 1.

`cumsum()`
Cumulative sum.

`cummax()`
Cumulative max.

`cummin()`
Cumulative min.

`cumprod()`
Cumulative product.

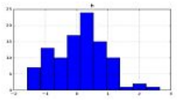
Windows

`df.expanding()`
Return an Expanding object allowing summary functions to be applied cumulatively.

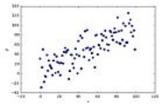
`df.rolling(n)`
Return a Rolling object allowing summary functions to be applied to windows of length n.

Plotting

`df.plot.hist()`
Histogram for each column



`df.plot.scatter(x='w',y='h')`
Scatter chart using pairs of points



Combine Data Sets

adf

x1	x2
A	1
B	2
C	3

+

bdf

x1	x3
A	T
B	F
D	T

=

Standard Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NaN

`pd.merge(adf, bdf, how='left', on='x1')`
Join matching rows from bdf to adf.

x1	x2	x3
A	1.0	T
B	2.0	F
D	NaN	T

`pd.merge(adf, bdf, how='right', on='x1')`
Join matching rows from adf to bdf.

x1	x2	x3
A	1	T
B	2	F

`pd.merge(adf, bdf, how='inner', on='x1')`
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NaN
D	NaN	T

`pd.merge(adf, bdf, how='outer', on='x1')`
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

`adf[adf.x1.isin(bdf.x1)]`
All rows in adf that have a match in bdf.

x1	x2
C	3

`adf[~adf.x1.isin(bdf.x1)]`
All rows in adf that do not have a match in bdf.

ydf

x1	x2
A	1
B	2
C	3

+

zdf

x1	x2
B	2
C	3
D	4

=

Set-like Operations

x1	x2
B	2
C	3

`pd.merge(ydf, zdf)`
Rows that appear in both ydf and zdf (Intersection).

x1	x2
A	1
B	2
C	3
D	4

`pd.merge(ydf, zdf, how='outer')`
Rows that appear in either or both ydf and zdf (Union).

x1	x2
A	1

```
pd.merge(ydf, zdf, how='outer',
         indicator=True)
.query('_merge == "left_only"')
.drop(columns=['_merge'])
```
Rows that appear in ydf but not zdf (Setdiff).

TOOL

Structure of a DataFrame Object

df.columns: list-like group of column names

	Date	Size	Base Topping #1	Topping #2	Topping #3	Topping #4	Price
0	2018-02-20	L	Cheese	Garlic	NaN	NaN	15.0
1	2018-02-20	M	Cheese	Onions	NaN	NaN	13.5
2	2018-02-20	S	Margherita	NaN	NaN	NaN	10.0
3	2018-02-20	M	Margherita	NaN	NaN	NaN	12.0
4	2018-02-21	L	Bianco	NaN	NaN	NaN	14.5
5	2018-02-21	M	Marinara	Anchovies	NaN	NaN	13.5
6	2018-02-21	L	Cheese	Pepperoni	Peppers	Mushrooms	16.5

df.index: list-like group of row names

df.values: NumPy array of table data

array dtype is generic “object” type if table data is heterogeneous

array dtype is specific (e.g., int, float) if table data is homogeneous

PANDAS

Pandas is a package used for data manipulation & analysis and offers many unique features.

↳ Working w/ structured, tabular data.

↳ DATAFRAME: data structure which is a 2D table where data is organized in rows + columns

pandas.DataFrame(): 2D mutable size, potentionally heterogenous tabular data.

pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=None)

- data: Iterable, dictionary, or DataFrame
  - ↳ Dictionary can contain series, arrays, constants, dataclasses, or list-like objects.
- index: index or array-like used for resulting frame
- columns: index or array-like column labels
- dtype: default is None. DataType to force.
- copy: bool or None copy data from inputs.

Ex)
```
d = {"col1": [1,2],
     "col2": [3,4]}
df = pandas.DataFrame(data=d)
# converts dictionary to DataFrame
```

	col1	col2
0	1	3
1	2	4

Ex) `df2 = pandas.DataFrame(numpy.array([[1,2,3],[4,5,6],[7,8,9]]), columns=["a","b","c"])`

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

pandas.DataFrame.head(): Returns the first n rows of tabular data.

pandas.DataFrame(n=5) ← default is 5 (starting from 0). But you can change n value

pandas.DataFrame.shape(): Returns tuple representing dimensionality of the DataFrame.

Ex)
```
df = pandas.DataFrame({"col1": [1,2], "col2": [3,4], "col3": [5,6]})
df.shape # (2,3)
```

DATAFRAME SLICING & INDEXING

DataFrames have the ability to slice & index based on labels, or positions of rows & columns.

Slicing allows you to select a subset of rows or columns from the DataFrame.

Indexing & slicing can be done by:

- [ ]
  - ↳ selecting a particular column based on labels.
  - ↳ Ex. df["x2"] ← extracts column labeled x2 & returns the series
  - ↳ Ex. df[:3] ← return first 3 rows in DataFrame
- loc[ ]
  - ↳ Access both rows & columns based on their labels.
  - ↳ First entry in index = rows; 2nd index = column
  - ↳ Syntax: DataFrame_obj.loc[] OR DataFrame_obj.loc[][]
- iloc[ ]
  - ↳ Allows you to access rows & columns w/ specific integer positions
  - ↳ Syntax: DataFrame_obj.iloc[] OR DataFrame_obj.iloc[][]

pandas.DataFrame.sort_values(): sort by the values along either axis.

```
DataFrame.sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last', ignore_index=False, key=None)
```

- `by`: str OR list of str
- `axis`: 0 = index, 1 = columns. Axis to be sorted
- `ascending`: bool OR list of bool. Sort by ascending vs. descending
- `inplace`: If True, perform operation in place
- `kind`: quicksort, mergesort, heapsort. Stable
- `na_position`: puts NaNs at the beginning == FIRST; puts NaNs at the end == LAST
- `ignore_index`: True. The resulting axis will be labeled 0, 1, ..., n-1
- `key`: [OPTIONAL]

pandas.concat(): concatenate pandas object along a particular object axis

```
pandas.concat(objs, axis=0, join='outer', ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, sort=False, copy=True)
```

- `objs`: sequence OR DF obj OR mapping of series
- `axis`: 0 = index, 1 = columns. The axis to concatenate along
- `join`: 'inner' OR 'outer'. How to handle indexes on other axis
- `keys`: sequence : if multiple levels passed, should contain tuples
- `levels`: list of sequence. Specific levels to use for constructing a MultiIndex.
- `names`: list. Names for the levels in the resulting hierarchical index

NOTE: pd.concat() will NOT change the values in the OG DF

pandas.DataFrame.merge(): Merge DF or named series obj w/ database style join.

```
DataFrame.merge(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, validate=None)
```

- `right`: DF or named series obj to merge with
- `how`: 'left', 'right', 'outer', 'inner', 'cross'. Type of merge to be performed.
- `on`: label OR list. Column or index level names to join on. Must be found in both DF.
- `left_on`: label OR list. Columns or index level names to join on. In the left DF.
- `right_on`: label OR list. Columns or index level names to join on. In the right DF.
- `left_index`: use the index from the left DF as the join key(s).
- `right_index`: use the index from the right DF as the join key

pandas.DataFrame.rename(): alter axes labels

```
DataFrame.rename(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False, level=None, errors='ignore')
```

- `mapper`: dict-like or function
- `index`: dict-like or function. Alternative to specifying axis
- `columns`: dict-like or function. Alternative to specifying axis.
- `axis`: 0 = index, 1 = columns. Axis to target with mapper.
- `inplace`: Whether to return new DF. If True, value of copy is ignored

pandas.DataFrame.head(): Returns the first n rows of tabular data.

pandas.DataFrame(n=5) ← default is 5 (starting from 0). But you can change n value

pandas.DataFrame.columns(): The column labels of the DataFrame.

pandas.DataFrame.groupby(): group DF using mapper OR by series of columns

```
DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=NoDefault.no_default, observed=False=dropna=True)
```

pandas.DataFrame.describe(): Generates descriptive statistics

```
DataFrame.describe(percentiles=None, include=None, exclude=None, datetime_is_numeric=False)
```

- `percentiles`: list-like of num [OPTIONAL]. The percentiles to include in the output.
- `include`: [OPTIONAL]. list of dtypes to include in result. 'all' = all columns of the input that'll be included in the output.
- `exclude`: [OPTIONAL]. A list of dtypes to omit from the result.
- `datetime_is_numeric`: Whether to treat datetime dtypes as numeric. DEFAULT: False

NOTE: more detailed in Prelab 2: get summary statistics using describe exercise

pandas.dataFrame.isnull(): Detects any missing values in the data by returning T/F.

we can take the last data: by using .tail() method: nbdh_reviews.tail(5) #last 5 data

pandas.Series.fillna(): used to fill in missing values in a Series or DataFrame object.

```
Series.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None)
```

- `value`: scalar, dict, Series, or DF. Value to fill holes
- `method`: "backfill", "bfill", "pad", "ffill", None. Method to use for filling holes in reindexed Series pad/ffill
- `axis`: axis along which to fill missing value
- `inplace`: True = fill in place. You're not just creating a copy object w/ filled values, but change the og values

NOTE: Object with missing values filled, None if inplace=True

pandas.Series.mean(): Returns the mean of the values over the requested axis.

```
Series.mean(axis=NoDefault.no_default, skipna=True, level=None, numeric_only=None)
```

- `axis`: for the function to be applied on.
- `skipna`: Exclude NA/null values when computing the result

pandas.DataFrame.corr(): provides a list of correlation coefficients of pairwise of columns, excluding NA/null values

```
DataFrame.corr(method="pearson", min_periods=1)
```

- `method`: Method of correlation "pearson", "kendall", "spearman"
  - "pearson": standard correlation coefficient
  - "kendall": Kendall Tau correlation coefficients
  - "spearman": Spearman rank correlation
- `min_periods`: int [OPTIONAL]. Minimum # of observations required per pair of columns to have a valid result

pandas.DataFrame.sample(): Returns a random sample of items from an axis of object.

DataFrame_object.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None, ignore_index=False)

- n: int [OPTIONAL] — number of items from axis to return
- frac: float [OPTIONAL] — fraction of axis items to return
- replace: [OPTIONAL] — Allow/disallow sampling of the same row more than once.

pandas.notnull(): Detect non-missing values for an array-like objects.

pandas.notnull(obj) — array like object → object to check for not null or non missing values

pandas.api.types.infer_dtype(): Describing the common type of the input data. (Return a string)

pandas.api.types.infer_dtypes(value) — scalar, list, ndarray, or pandas type

pandas.DataFrame.dtypes(): Returns the dtypes in the DataFrame

pandas.Categorical(): Represents a categorical variable in classic R/S-plus fashion.

pandas.Categorical(values, categories=None, ordered=None, dtype=None, fastpath=False, copy=True)

- values: list-like — The values of the categorical.
- categories: index-like [OPTIONAL] — The unique categories for this categorical.
- ordered: bool; DEFAULT=False — Whether this categorical is treated as a ordered categorical. TRUE = resulting categorical will be ordered
- fastpath, copy: OPTIONAL

pandas.get_dummies(): Converts categorical variable into dummy/indicator variables

pandas.get_dummies(data, prefix=None, prefix_sep="_", dummy_na=False, columns=None, sparse=False, drop_first=False, dtype=None)

- data: array-like, series or DF — Data in which to get dummy indicators
- prefix: str, list of str, dict of str; DEFAULT=None — String to append DF column names
- prefix_sep: str — If appending prefix
- dummy_na: bool; DEFAULT=False — Add column to indicate NaNs
- columns: list-like — Column names in the DF to be encoded

pandas.Series.idxmax(): Retrieves the index (or a name) of the location where the maximum value in a series was found.

Series.idxmax(axis=0, skipna=True)

- axis: int [DEFAULT=0]
- skipna: bool [DEFAULT=True] — Excludes NA/null values

pandas.DataFrame.drop(): Drop specified labels from rows or columns

DataFrame.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')

- labels: single label OR list-like — index or column labels to drop
- axis: whether to drop labels from the index — 0=index, 1=columns

pandas.Series.sort_values(): Sort a series/values in ascending/descending order

Series.sort_values(axis=0, ascending=True, inplace=False, kind="quicksort", na_position="last", ignore_index=False, key=None)

- axis: 0 or index — Axis to direct sorting
- ascending: True=ascending, False=descending
- inplace: True=perform operation in place, False=opposite

PRACTICE PANDAS & NUMPY

Step 1

The code cell below imports the `pandas` package, using its conventional shorthand name, `pd`. It then imports the `numpy` package using its conventional shorthand name, `np`. Run the cell below.

```
import pandas as pd #we'll use pandas as pd
import numpy as np #we'll use numpy as np
```

Step 2

In the code cell below, use numpy's `np.random.normal()` function to create two arrays, each with size=100, loc=0.0, and scale=1.0. Name your arrays `a` and `b`.

If you would like to learn how to use the `np.random.normal()` function, use the help `?` functionality by running the line `np.random.normal?` in the code cell below. Recall that you can use help by specifying the function name (in this case, `np.random.normal`), followed by `?`. You can also access the online documentation.

```
a = np.random.normal(size=100, loc=0.0, scale=1.0)
b = np.random.normal(size=100, loc=0.0, scale=1.0)
```

Step 3

Now we'll convert our data to a Pandas DataFrame. The `DataFrame()` function is the primary way to create DataFrames. You will use this approach to create a DataFrame out of the two arrays created in the previous step.

Your DataFrame will contain two columns, named `x1` and `x2`.

1: First create a Python dictionary in which the keys are the names of the columns you want to create, and the values are the arrays. Column `x1` will contain array `a` and column `x2` will contain array `b`. Assign this dictionary to variable `d`.

2: Next use the `pd.DataFrame()` function to convert dictionary `d` to a DataFrame, and assign the result to variable `df`.

For more information on how to use the `DataFrame()` function, consult the online documentation.

```
d = {"x1": a,
     "x2": b}
df = pd.DataFrame(data=d)
```

Step 4

The pandas `head()` method allows you to view the first *n* rows in a DataFrame. The syntax is `DataFrame_obj.head()`. You can consult the online documentation to see how to use this method. You can also use IPython's `help` functionality by typing `df.head?` in the cell below. Recall that you can obtain documentation by specifying the object name (in this case, `df`), followed by the method or property (in this case, `head`), followed by `?`.

In the code cell below, use the `head()` method to view the first 5 rows of dataframe `df`.

```
df.head
```

	x1	x2
0	-0.044563	-2.558524
1	-0.006936	0.746383
2	0.020261	0.090749
3	0.656970	0.212481
4	3.090097	1.123328

Step 5

Add a new column to your `df` DataFrame. You can add this new column to `df` by:

1. using bracket notation and specifying the label of the new column: `df['new_column_label']`
2. assigning it the values of the column: `df['new_column_label'] = column values`

Name your new column **x3**. Column **x3** will consist of the sum of the values in column **x1** and column **x2**. Hint: You can access each column using bracket notation as seen above, and you can add entire columns together using the addition operation (`+`).

After you have added the new column `x3` to DataFrame `df`, use the `head()` method to examine the new values in `df`.

```
df["x3"] = df["x1"] + df["x2"]
df.head()
```

	x1	x2	x3
0	-0.044563	-2.558524	-2.603087
1	-0.006936	0.746383	0.739447
2	0.020261	0.090749	0.111010
3	0.656970	0.212481	0.869451
4	3.090097	1.123328	4.213425

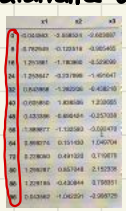
- To retrieve value we can access a row like a list using iloc or loc.
  - `df.loc[5]["x1"]` # -0.9412724229462 98 ← using label 5 to access row
  - `df.iloc[5][0]` # -0.9412724229462 98

- To access a subset of rows & columns in a DataFrame using slicing.
  - `df[10:14]`

	x1	x2	x3
10	1.736734	0.163871	1.900605
11	2.099049	-0.344945	1.754104
12	1.545302	-1.250911	0.294392
13	0.827693	0.255183	1.082876

← starting at row 10 & ending at row 13.

  - `df[::8]` outputs every 8th row.



  - `df.iloc[10:14, [2]]`

	x3
10	1.900605
11	1.754104
12	0.294392
13	1.082876

Returns column with index of 2 (column x3) in rows 10-13.

- Selecting rows based on conditions:

Syntax: `DataFrame_obj[condition]` ← boolean condition on the data

  - `d = df[df["x3"] > 1]`

means column index 2 so column labeled x3

  - `df[df["x3"]>1].shape[0]` #29

rows that have values > 1 in column 3 return that row + other columns x1, x2, x3.