CIS 41B
Advanced Python Programming

numpy

De Anza College
Instructor: Clare Nguyen

CSV file

- In addition to reading and writing plain text files, Python can read / write many other file formats.
- One common format that's used with a large numbers of data values is the CSV format or **c**omma **s**eparated **v**alue format.
- CSV is often the default data exchange format for consumer, business, and scientific applications. This is because a CSV file can be opened by spreadsheet tools such as Excel, and it can also be opened as a plain text file for reading/writing.
- An example line of a CSV file:

  Monty Python And The Holy Grail,1975,Terry Gilliam and Terry Jones

  There are 3 fields: movie title, year released, directors. The 3 fields are separated by a comma.
- The CSV format is like a spreadsheet:
  - Each line is one data record or one row of a spreadsheet.
  - The fields in a line are the columns of the spreadsheet.

Reading CSV File

- The csv module has functions to read data from a CSV file.
- The csv.reader function is an *iterator* that reads in one line at a time from the file and splits each line into a list of field values, yielding the list.
- The function signature of the reader function is:

```
csv.reader(csvfile, dialect='excel', delimiter=',')
```

where:

  - csvfile is the file handle to the csv file
  - dialect dictates the formatting standard of data in the spreadsheet, the default is excel
  - delimiter shows the delimiting character between fields, the default is comma

- Example:

```
import csv

with open('inputfile.csv') as fh :
    reader = csv.reader(fh)        # reader object is an iterator
    for row in reader :            # each row is a list of fields in a line
        print(row[0], row[-1])     # print the first and last fields
```

- How would you use csv.reader to read in one line from the file?

Writing CSV File

- The csv.writer function creates a writer object, with a writerow method.
- writerow writes a list of data into a comma separated line in the output file.
- The function signature of the writer function is:

```
csv.writer(csvfile, dialect='excel', delimiter=',' , **fmtparams)
```

where:

  - csvfile is the file handle to the csv file
  - dialect dictates the formatting standard of data in the spreadsheet
  - delimiter shows the delimiting character between fields
  - fmtparams are other parameters that are used to overwrite any formatting standard specified by the dialect

- The csv file handle must be opened with the parameter newline=' ' (empty string) so that empty lines are not inserted in between rows.
- Example:

```
import csv

with open('outputfile.csv', 'w', newline='') as fh :
    writer = csv.writer(fh)        # create a writer object
    for aList in table :           # table is a list of lists of data to be written to file
        writer.writerow(aList)     # write the list to file as a comma separated line
```

numpy

- The numpy module provides highly efficient functions for mathematical calculations, specifically for large multi-dimensional tables and matrices.
- Before we start using numpy, it's important to note that many mathematical calculations can already be done with the built-in objects in Python.
  - Python has a large number of numeric data classes (int, float, decimal, complex) and fast access to data structures (lists, dictionaries…) that store these numbers.
- Therefore, numpy is mainly used for intensive computing of large numbers of numeric data, especially multi-dimensional, large matrices.
- Getting started with numpy:
  - numpy is a long name so most programmers use `import numpy as np`
  - numpy's basic data type is an array, which is an alias to the actual data type name ndarray (for N-dimensional array) that's used internally in numpy
  - To refer to a numpy array, use the data type: `np.array`
  - A numpy array is what most programmers know as an array:
    - it can contain many numeric data values
    - the data are stored in a specific location and can be indexed

Introduction to numpy Arrays (1)

- A numpy array contains only one type of data: only integers or only floats.
  - If input data are different types, the smaller size data are promoted to a larger size.

```
arr = np.array([1, 3.4, 2.5, 7])        # create array from ints and floats
print(arr)                              # output:   [1.  3.4   2.5   7. ]
                                           # 1 and 7 are promoted to float
```

- To check the data type of an array:

```
print(array.dtype)
```

  To change the type of data in an array:

```
intArr = arr.astype(int)          # arr contains floats, intArr contains integers
```

- A numpy array and a Python list are different when we apply the * operator on them.

```
mylist = [1, 2, 3]
print(mylist * 2)       # output:   [ 1, 2, 3, 1, 2, 3 ]

myarr = np.array([1, 2, 3])
print(myarr * 2)        # output:   [ 2  4  6 ]        # Note: no comma in output
```

Introduction to numpy Arrays (2)

- A numpy array is a C array "under the hood," therefore:
  - There's no insert, append, or pop methods.
  - Changing the size (the number of elements) of a numpy array is expensive and is generally avoided.
- The advantage of an array over a list is clear in calculations with large numbers of data.

1. Code is easier to read.

If we convert 10,000 Celsius temperatures in tempC to Fahrenheit

Using a list:

```
tempF = [ temp * 9 / 5 + 32  for temp in tempC ]
```

Using an array:

```
tempF = tempC * 9 / 5 + 32
```

2. Code runs faster

Even with comprehension, using a list is slower than using an array for calculation of large numbers of data values.

numpy uses libraries of functions that were developed in C or Fortran, and many mathematicians have spent many years squeezing out every bit of inefficiencies.

- Note that there is a Python array data type, a Python array is not the same as the numpy array.

Array Shape

- An array dimension is the shape of an array:

```
print(arr.shape)     # output:  (4,)      for a 1D array with 4 elements
print(arr.shape)     # output:  (2, 5)    for a 2D array with 2 rows, 5 columns
```

- The shape output is a tuple of (number of rows, number of columns)
- We can change the dimension or shape of an array by assigning a tuple of (num rows, num cols) to the shape attribute:

```
print(arr.shape)     # output:  (6, 2)    a 2D array with 6 rows and 2 columns
arr.shape = (3,4)    # change array to 3 rows and 4 cols
```

- The new array size (which is rows * columns) must be the same before and after the dimension change.

Access Array with Integer Index (1)

- Accessing a 1D array is just as with a list, by using [index]:

```
arr[0]      # first element
arr[-1]     # last element
arr[:3]     # first 3 elements, using a slice
```

- Accessing a 2D array can be done (inefficiently) as with a list:

```
arr[2][0]    # 1 element at 3rd row, 1st column
```

  but this requires getting to a row first, and then getting to a column as a second step.

- A more efficient way for accessing multi-dimensional arrays is by using the [row,col] syntax:

```
arr[2, 0]          # 1 element at 3rd row, 1st column
arr[-1, :3]        #  last row, first 3 elements
arr[:2, 4:8]       #  elements in the first 2 rows, from 5th to 8th columns
```

- To get an entire row of data, we can omit the col in the index.

```
arr = np.array([ [1, 2, 3], [4, 5, 6], [7, 8 , 9]])
print(arr[:2]       # row index only, omit col index    output:   [ [1  2  3]
                    # comma is optional                            [4  5  6] ]
print(arr[,:2])         # Error! Can't omit the row index
```

Access Array with Integer Index (2)

- We can also use an array of indices to access data in an array:

```
arr = np.array([9, 2, 13, 4, 5, 10, 43, 22, 9, 17, 35, 1])
indices = np.array([2, 4, 2, 5])        # want data at these indices
print(arr[indices])                      # output:   [ 13  5  13  10 ]
```

- To access an array in reverse, we can use the same slice technique as with a Python sequence:

```
arr = np.array([9, 2, 13, 4, 5])
print(arr[::-1])                        # output:   [ 5  4 13  2  9]
```

Access Array with Boolean Index (1)

- We can also use boolean indexing with arrays.
  This means we can choose data in the array that meet a certain condition.
- When we use an array in a boolean expression:
  - Each element in the array is evaluated with the boolean expression
  - The boolean result is stored in an output array that is the same shape as the input array

```
arr = np.array([ [4, 7, 3, 4, 2], [2, 6, 4, 9, 8] ])      # array with 2 rows x 5 columns
print(arr <= 4)                    # output:   [ [True  False  True  True   True]
                                                 [True  False  True  False  False] ]
```

- Re

- Taking advantage of the beh `<  <=  >  >=  ==  !=` xpression, we can use boolean indexing of arrays:

```
arr = np.array([ [4, 7, 3, 4, 2], [2, 6, 4, 9, 8] ])      # same array as above
print(arr[arr <= 4])               # output:   [ 4  3  4  2  2  4  ]
                                   # which are all values <= 4 in the array
```

Access Array with Boolean Index (2)

- We can also use boolean indexing with the logical methods any and all

```
arr = np.array([ [4, 7, 3, 4, 2], [2, 6, 4, 9, 8] ])
print(arr <= 4)                  # output:   [ [True  False  True  True   True]
                                                [True  False  True  False  False] ]
print(np.any(arr<=4))            # output: True
print(np.all(arr<=4))            # output: False
```

Initialize Array with List

- Arrays can be easily created from lists by using the array function.
- Converting a list to a 1D array, and a 1D array to a list:

```
arr = np.array([1, 3.4, 2.5, 7])      # input is list with ints and floats
print(arr)                            # output:   [1.  3.4  2.5  7. ]
L = list(arr)                         # input is array with floats
print(L)                              # output:   [1.0, 3.4, 2.5, 7.0]
```

- A 2D array needs to be created from a list of lists:

```
arr = np.array( [[2, 4, 6], [5, 6, 7]] )      # input is list of lists
print(arr)                                     # output:  [ [2  4  6]
                                                          [5  6  7] ]
```

But converting a 2D array back to a list of lists requires converting each list

```
L = [list(row) for row in list(arr)]          # input is 2D array
print(L)                                      # output:   [[2, 4, 6], [5, 6, 7]]
```

Initialize Array with Literal Values

- Because an array of 0's or an array of 1's are often used in linear algebra, there are special functions to create these arrays.

```
zeroArr = np.zeros((4, 7))        # array with 4 rows x 7 columns, filled with 0.0
                                  # float is the default data type
oneArr = np.ones((3,5), dtype=int)     # array with 3 rows x 5 coumnls, filled with 1
```

  - Specify the keyword argument dtype for integers, otherwise floats is the default data type for both functions.
  - The first input argument is a tuple for the array shape (row, column)

- Creating and initializing an array with random numbers:

```
arr = np.random.random(10)                # array of 10 floats between 0.0 and 1.0
arr = np.random.randint(1, 7, size=12)    # array of 12 integers in the range
                                          # 1 <= num <= 6
```

- Create and initialize an array with a range of data:

```
arr =  np.linspace(2.0, 3.0, 50)          # array of 50 floats, equally spaced in the
                                          # interval 2.0 <= num < = 3.0
arr = np.arange(10, 50, 2)                # arange works like the range function, but
                                          # produces an array instead of an iterator
```

Initialize Array with CSV File

- Because numerical data are formatted into CSV file, numpy has a simple way to read in data from a CSV file.
- CSV files with numerical data can then be read in by the loadtxt function into a numpy array:

```
data = np.loadtxt(filename, delimiter=",")
```

- The default data type is float, but we can use the dtype argument to specify integer data or even string data:

```
data = np.loadtxt(filename, dtype=int, delimiter=",")
```

```
data = np.loadtxt(filename, dtype=str, delimiter=",")
```

- The resulting numpy array will have the same shape as the CSV file: same number of rows and columns.
- Note that since numpy's strength is numerical calculations, most of the time data that are in a numpy array are numeric data. But strings can also be stored in numpy arrays to take advantage of the flexible ways to index numpy arrays.

Overwrite Data in an Existing Array

- We can assign a new data value to any element that we can access.
- Assign value to one element:

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
arr[1,0] = 0
print(arr)                    # output:  [ [1  2  3]
                                          [0  5  6] ]
```

- Assign value to a slice of elements:

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
arr[1:,] = 0
print(arr)                    # output:  [ [1  2  3]
                                          [0  0  0] ]
```

- Assign va

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
arr[arr>2] = 10
print(arr)                    # output:  [ [  1   2  10]
                                          [10  10  10] ]
```

Copy and View Array

- Shallow copy an array is just like shallow copying a Python mutable object:

```
newArr = arr              # newArr is another name for arr
```

- Use the copy method for deep copying an array:

```
newArr = arr.copy()       # newArr is separate but duplicate of arr
```

- Arrays are typically large but calculation often needs to be done on only one part of the array. For example, find the sum of all data in each of the first 5 columns of a 2D array.
- If the calculation doesn't modify the data, such as calculating the sum, numpy offers a way to select a view into a part of the array.
  - A view of an array means to select a section of the array and view it as a new array, but without making a copy of all the data in the subset.
- Use a slice with an array to create a view:

```
arr[row slice, column slice]
```

```
newArr = arr[2:4, :5]          # newArr is a view of arr,  which consists of
                               # the first 5 columns of the 3rd  and 4th rows
```

- A view is not a separate copy of the original array. If we modify data in the view, we will change data in the original array.

Arithmetic with Array

- Python arithmetic operators and the re-assignment operators can be used with an array.

```
+    -    *    /    //   %    **
+=   -=   *=   /=   //=  %=   **=
```

- If the operators are used with an array and a scalar (which is a single number):
  - The arithmetic expression is applied to every element of the array:

```
arr = np.array([1, 2, 3, 4])
print(arr / 2)                    # output:  [ 0.5  1.0  1.5  2.0 ]
```

- If the operators are used with two arrays, then the 2 arrays must have the same shape.
  - The arithmetic expression works on corresponding elements from each array:

```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([-1, 0, 1, 2])
print(arr1 * arr2)                # output:   [ -1  0  3  8 ]
```

  - FYI for those working in linear algebra: The * operator is not the same as:
    - np.dot() calculates the dot product of 2 arrays
    - np.mat() converts a multidimensional array to a matrix for matrix multiplication.

Compare Array

- We can compare elements of 2 arrays that have the same shape by using the relational operators:

```
<   <=   >   >=   ==   !=
```

  - Each pair of corresponding elements in the 2 arrays are compared, resulting in a boolean value, and the boolean value is put in an array of the same shape as the 2 input arrays:

```
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([1, 1, 3, 5, 7])
print(arr1 < arr2)                  # output:  [ False False False True True ]
```

- It is also possible to compare 2 arrays for exact match in values:

```
print(np.array_equal(arr1, arr2))          # output:   False
```

Common Math Methods (1)

- Find min or max value:
  - In each row or column:

```
np.min(arr, 0)      # return 1D array of smallest value of each column
```

```
np.max(arr,1)       # return 1D array of largest value of each row
```

axis = 0 for column, 1 for row, and can be higher for multi-D array

  - In the entire array:

```
np.min(arr)         # return smallest value in array
```

```
np.max(arr)         # return largest value in array
```

- Find the sum:
  - In each row or column:

```
np.sum(arr, 0)      # return 1D array of sum of each column
```

```
np.sum(arr, 1)      # return 1D array of sum of each row
```

axis = 0 for column, 1 for row, and can be higher for multi-D array

  - In the entire array:

```
np.sum(arr)         # return sum of all values in array
```

Common Math Methods (2)

- Sort values:
  - In each row or column:

```
np.sort(arr, 0)          # return arr sorted by column
```

```
np.sort(arr, 1)          # return arr sorted by row
```

axis = 0 for column, 1 for row, and can be higher for multi-D array

  - In the entire array:

```
np. sort(arr)            # return arr sorted by the largest axis
```

- Sort values to get the index of the sorted array
  - In each row or column:

```
np.argsort(arr, 0)       # return 1D array of indices that would sort
                         # arr by column
```

```
np.argsort(arr, 1)       # return 1D array of indices that would sort
                         # arr by row
```

  - In the entire array:

```
np.argsort(arr)          # return array of indices that would sort
                         # arr largest axis
```

Common Statistics Methods (1)

- The median is the middle value if the array was sorted

```
np.median(arr, 0)     # return 1D array of median of each column
```

```
np.median(arr, 1)     # return 1D array of median of each row
```

axis = 0 for column, 1 for row, and can be higher for multi-D array

```
np.median(arr)        # return 1D median of all elements
```

- The mean, commonly known as the average, is the sum of elements divided by the number of elements

```
np.mean(arr, 0)       # return 1D array of mean of each column
```

```
np.mean(arr, 1)       # return 1D array of mean of each row
```

axis = 0 for column, 1 for row, and can be higher for multi-D array

```
np.mean(arr)          # return the mean of all elements
```

Common Statistics Methods (2)

- The variance and standard deviation together show whether the data points are close together or spread out.
- The variance is the squared deviation (or how far away) from the mean.

```
np.var(arr, 0)        # return 1D array of variance for each column
```

```
np.var(arr, 1)        # return 1D array of variance for each row
```

axis = 0 for column, 1 for row, and can be higher for multi-D array

```
np.var(arr)           # return the variance for all elements
```

- The standard deviation is the square root of the variance. If it's a low value compared to the data range, then the data are close together. If it's a high value relative to the data range, then the data is spread out.

```
np.std(arr, 0)        # return 1D array of standard dev for each column
```

```
np.std(arr, 1)        # return 1D array of standard dev for each row
```

axis = 0 for column, 1 for row, and can be higher for multi-D array

```
np.std(arr)           # return standard dev for all elements
```

Going further…

- numpy has many more features than what is covered here.
  - For linear algebra: matrix and vector calculations
  - For probability and statistics: functions and generated data for statistical distributions, weighted probability, synthetic data
- It is also important to note that for intensive numerical analyses, numpy is a subset of other tools that Python provides:
  - scipy: a module for scientific calculation that includes all of numpy plus additional support for optimization, integration, interpolation, linear algebra, signal and image processing, among many others.
  - pandas is a superset that includes numpy *and* scipy modules plus additional support for manipulating numerical tables (such as dataframe objects that allow alignment, reshaping, pivoting, merging, slicing of datasets), and time series functionality (such as frequency conversion, linear regression, date shifting)
- The competition with the Python numerical stack (numpy, scipy, pandas) is, in order of how long they've been around, Matlab, R, and Julia programming languages, which also specialize in numerical calculation and analysis.

Up next: Plotting