Reading and Writing Files

Han-fen Hu

UNLV

Outline

- Files and File Paths
- os.path Module
- File Read/Write Process
- Saving variables

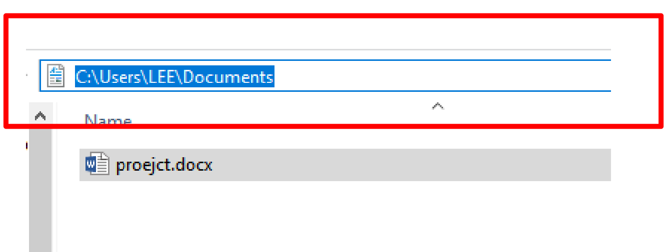
UNLV

Files

- A way to persistently save the data
- A way to have input from a static storage
- An essential step for automate things
- A file's contents can be considered as a single string value

Files and File Paths (1)

- A file has two key properties: a filename (usually written as one word) and a path
  - Filename includes the name and file extension
  - Path specifies the location of a file on the computer



Path

Files and File Paths (2)

- On Windows, paths are written using backslashes (\) as the separator
- OS X and Linux, however, use the forward slash (/) as their path separator.
- If you want your programs to work on all operating systems, you will have to write your Python programs to handle both cases
- `os.sep` variable
  - set to the correct folder-separating slash for the computer running the program

Files and File Paths (3)

- Current Working Directory
  - Every program that runs on your computer has a current working directory (cwd)
  - Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory

cwd Example (1)

- Get the current working directory as a string value with the `os.getcwd()`
- Change the working directory with os.chdir()

```python
import os
print(os.getcwd())

# change the working directory
os.chdir('C:\\Users\\Default\\Documents')

print(os.getcwd())
```

```
D:\backup\Fall2020_MIS740
C:\Users\Default\Documents
```


cwd Example (2)

- Python will display an error if you try to change to a directory that does not exist

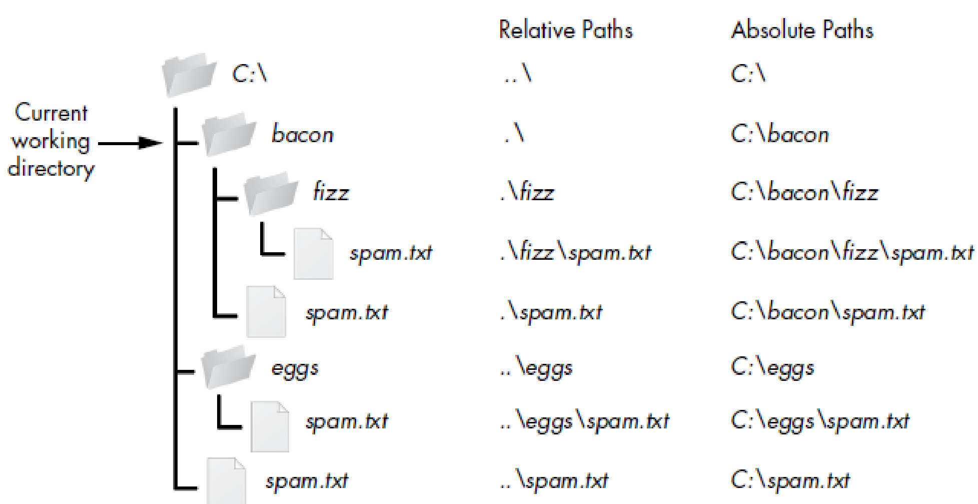
```python
import os
print(os.getcwd())

# change the working directory
os.chdir('C:\\Users\\HH\\Documents')

print(os.getcwd())
```

```
C:\Users\Default\Documents

---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-3-d724a238135f> in <module>
      3 
      4 # change the working directory
----> 5 os.chdir('C:\\Users\\HH\\Documents')
      6 
      7 print(os.getcwd())

FileNotFoundError: [WinError 3] The system cannot find the path specified: 'C:\\Users\\HH\\Documents'
```

Absolute vs. Relative Paths

- Absolute path always begins with the root folder
- Relative path is relative to the program's current working directory
  - dot (.) folder: shorthand for "this directory."
  - dot-dot (..) folders: Means "the parent folder."
  - Not real folders but special names that can be used in a path.

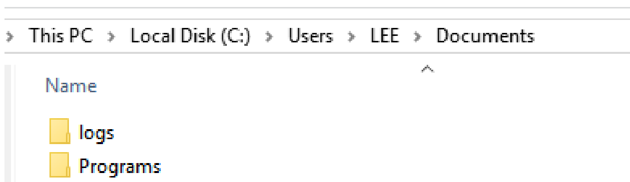
Absolute vs. Relative Paths: Example



Creating New Folders

- Your programs can create new folders (directories) with the `os.makedirs()`

```python
import os
# change the working directory
os.chdir('C:\\Users\\LEE\\Documents')

# create a folder under current wokring directory
os.makedirs('logs')

# create a folder with absolue path
os.makedirs('C:\\Users\\LEE\\Documents\\Programs')
```



`os.path` Module (1)

- Contains many helpful functions related to filenames and file paths
  - merging, normalizing and retrieving path names in python
- Full documentation: http://docs.python.org/3/library/os.path.html

os.path Module (2)

- Checking Path Validity
  - Many Python functions will crash with an error if you supply them with a path that does not exist.
- os.path.exists(*path*)
  - Return True if the file or folder referred to in the argument exists and will return False if it does not exist

```python
import os
# change the working directory
os.chdir('C:\\Users\\LEE\\Documents')

if not os.path.exists('logs'):
    # create a folder under current wokring directory
    os.makedirs('logs')
    print('Foleder "logs" created')
else:
    print('Foleder "logs" already exists')
```

UNLV

os.path Module (3)

- os.path.join()
  - Build paths in a way that will work on any operating system
    - \ for windows; / for OS X and Linux

```python
import os # for file and path operation
import datetime # for getting current date time

# get the current month
currentMonth = str(datetime.datetime.today().month)

# generate the path for storing the log files
path = os.path.join('app','logs', currentMonth)

# print it out to verify its correctness
print(path)
```

```
app\logs\10
```

UNLV

os.path Module (4)

- `os.path.abspath(path)`
  - Return a string of the absolute path of the argument
  - Converts a relative path into an absolute one
- `os.path.isabs(path)`
  - Return True if the argument is an absolute path and False if it is a relative path
- `os.path.relpath(path, start)`
  - Return a string of a relative path from the *start* path to path.
  - If *start* is not provided, the current working directory is used as the start path

os.path Module (5)

```
import os
# show the current working directory
print("Current working directory: "+os.getcwd())

# convert  .. to absotue path, save the string to the variable
absolutePath = os.path.abspath('..')
# print the variable
print("The absolute path of .. : "+absolutePath)

# convert .\\logs to absotue path and print it
print("The absolute path of .\\logs : "+os.path.abspath('.\\logs'))

# show whether . is an absolute path, should be false
print(os.path.isabs('.'))
# convert . to absolute path,
# and check whether the conversion result is an absolute path
print(os.path.isabs(os.path.abspath('.')))

```

```
Current working directory: C:\Users\LEE\Documents
The absolute path of .. : C:\Users\LEE
The absolute path of .\logs : C:\Users\LEE\Documents\logs
False
True
```

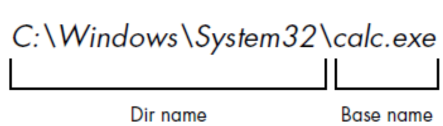
os.path Module (6)

```python
import os
# show the current working directory (CWD)
print("Current working directory: "+os.getcwd())

# get the path of CWD, relative to C:\\, assign the result ot a variable
relativePath = os.path.relpath(os.getcwd(), 'C:\\')
# print the variable
print(relativePath)

# show the path of CWD, relative to C:\\Windowss
print(os.path.relpath(os.getcwd(), 'C:\\Windows'))


# show the path of C:\\Windowss, relative to CWD
print(os.path.relpath('C:\\Windows', os.getcwd()))
```

```
Current working directory: C:\Users\LEE\Documents
Users\LEE\Documents
..\Users\LEE\Documents
..\..\..\Windows
```


os.path Module (7)

- os.path.dirname(*path*)
  - Return a string of everything that comes before the last slash in the path argument.
- os.path.basename(*path*)
  - Return a string of everything that comes after the last slash in the path argument



os.path Module (8)

```
dataFilePath = 'C:\\Users\\LEE\\Documents\\sales2019.xlsx'
pathTuple = os.path.split(dataFilePath)
print(pathTuple)
```

```
('C:\\Users\\LEE\\Documents', 'sales2019.xlsx')
```

Is equivalent to

```
dataFilePath = 'C:\\Users\\LEE\\Documents\\sales2019.xlsx'
pathTuple = (os.path.dirname(dataFilePath), os.path.basename(dataFilePath))
print(pathTuple)
```

UNLV

os.path Module (9)

- os.sep variable
  - set to the correct folder-separating slash for the computer running the program
    - For Windows

```
dataFilePath = 'C:\\Users\\LEE\\Documents\\sales2019.xlsx'
print(dataFilePath.split(os.sep))
```

```
['C:', 'Users', 'LEE', 'Documents', 'sales2019.xlsx']
```

- For OS X and Linux

```
dataFilePath = '/usr/bin/sales2019.xlsx'
print(dataFilePath.split(os.sep))
```

```
['', 'usr', 'bin', 'sales2019.xlsx']
```

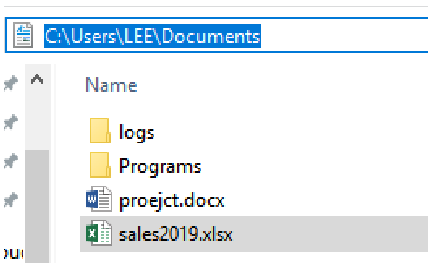
UNLV

`os.path` Module (10)

- os.listdir(*path*)
  - Return a list of filename strings for each file in the *path* argument.

```
fileList = os.listdir(os.getcwd())
print(fileList)
```

```
['desktop.ini', 'logs', 'My Music', 'My Pictures', 'My Videos', 'proejct.docx', 'Programs', 'sales2019.xlsx']
```



Exercise / Question

- What does the following program do?

```
for filename in os.listdir(os.getcwd()):
    print(os.path.join(os.getcwd(), filename))
```

Types of Files

- Plain text files
  - Contain only basic text characters and do not include font, size, or color information
  - Can be opened with Windows's Notepad or OS X's TextEdit application
  - With .txt or .csv file extension
- Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs
  - Every different type of binary file must be handled in its own way

File Reading/Writing Process

1. Call the `open()` function to return a File object.
2. Call the `read()` or `write()` method on the `File` object.
3. Close the file by calling the `close()` method on the `File` object.

Open Files

- open()
  - Pass it a string path indicating the file you want to open
  - The path can be either an absolute or relative path
  - It returns a `File` object
    - A File object is simply another type of value in Python, much like the lists and dictionaries
  - The file will be opened in "reading plaintext" mode
    - Can't write or modify it in any way

Reading the Contents of Files

- read()
  - Read the entire contents of a file as a string value
- readlines()
  - get a list of string values from the file, one string for each line of text

Close the File

- close()
  - Close the File.

- Lab
  - ReadFileContent
    This program reads the content of a file and show it on the screen

Writing to Files (1)

- The file needs to be open in “write plaintext” mode or “append plaintext” mode
  - Write mode
    - Overwrite the existing file and start from scratch
    - Pass 'w' as the second argument to open() to open the file in write mode.
  - Append mode
    - Append text to the end of the existing file
    - Pass 'a' as the second argument to open() to open the file in append mode.

UNLV

Writing to Files (2)

- `write(string)`
  - Write *string* to the file
  - It does **not** automatically add a newline character to the end of the string

UNLV

Writing to Files: Example

```
# open the file in the write mode
baconFile = open('bacon.txt', 'w')
# write to the file; overwrite everything
baconFile.write('Hello world!\n')
# close the file
baconFile.close()

# open the file in the append mode
baconFile = open('bacon.txt', 'a')
# write to the file, append the new content to the end
baconFile.write('Bacon is not a vegetable.')
# close the file
baconFile.close()

# open the file in the read mode
baconFile = open('bacon.txt')
# read the content
content = baconFile.read()
# close the file
baconFile.close()

# print the content
print(content)
```

```
Hello world!
Bacon is not a vegetable.
```


- Lab
  - Degree Courses
    In this program, the user will enter names of the courses he/she is taking this semester. Write the input value to a file

- Exercise
  - Movie Record
    Please write a program that reads the movie record from a file (MovieBoxOffice.txt) and shows the content to the user. The user can add a record by entering a move name and its box office.

NumPy (1)

Han-fen Hu

UNLV

Outline

- Introduction to NumPy
- NumPy Arrays
- Computation on NumPay Arrays

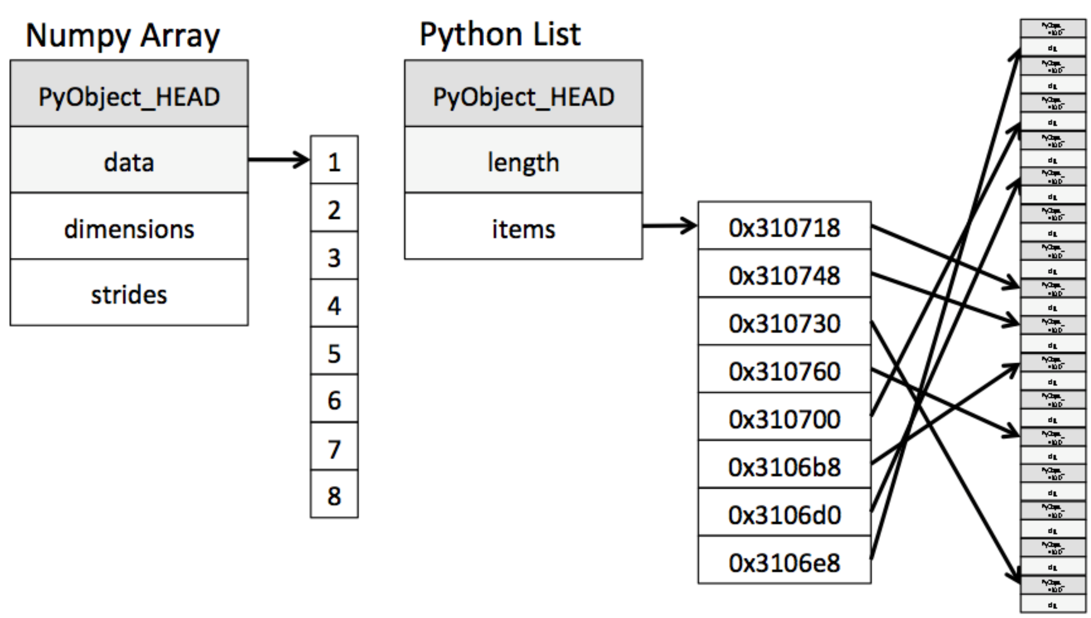
UNLV

Introduction to NumPy (1)

- Why NumPy?
  - One of the most powerful Python libraries
  - Improve how data is stored and manipulated
  - Contains a multi-dimensional array and matrix data structures
  - Pandas relies heavily on NumPy
- Purpose
  - Store in-memory data in a more efficient way
  - Includes a large number of mathematical, algebraic and transformation functions
- Installed with Anaconda

Introduction to NumPy (2)

- A more efficient way to store data



NumPy Arrays (1)

- A collection of relevant data
- Fixed-Type: All items in the array are of the same data type
  - Items of a Python list can be of different data types

UNLV

Popular NumPy Data Types

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (normally either int64 or int32)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa

UNLV

Creating NumPy Arrays (1)

- From Python lists

```
# import numpy, set the alias as np
import numpy as np

# declare a list
ageList = [58,69,32,53,81,60,18,25]

# convert the list to a numpy array
ageArray = np.array(ageList)
print(ageArray)

ageArray = np.array(ageList, dtype='float32')
print(ageArray)
```

```
[58 69 32 53 81 60 18 25]
[58. 69. 32. 53. 81. 60. 18. 25.]
```


Creating NumPy Arrays (2)

- Creating Arrays from Scratch
  - Specify the size
  - Specify the data type
  - Specify the values

```
# import numpy, set the alias as np
import numpy as np

# create an array with all 0s, single-dimension
zeroArray = np.zeros(10, dtype='int')
# create an array with all 1s, a 3X5 array
oneArray = np.ones((3,5), dtype='float')
# Create a 3x3 identity matrix
eyeArray = np.eye(3)
# create an array filled with 3.14, a 2X6 array
piArray = np.full((2,6), 3.14)

print(zeroArray)
print()
print(oneArray)
print()
print(eyeArray)
print()
print(piArray)
```

```
[0 0 0 0 0 0 0 0 0 0]

[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

[[3.14 3.14 3.14 3.14 3.14 3.14]
 [3.14 3.14 3.14 3.14 3.14 3.14]]
```

Creating NumPy Arrays (3)

- `arange(`*`start, stop, step`*`)`
  - Create an array filled with a linear sequence, with *start*, *stop*, and *step* values

```
# Create an array filled with a linear sequence
# Starting at 20, ending at 65, stepping by 5
checkInAges = np.arange(20, 65, 5)
print(checkInAges)
```

```
[20 25 30 35 40 45 50 55 60]
```

- `linspace(`*`lowerBound, upperBound, numberOfValues`*`)`
  - Create an array with values between two numbers

```
# Create an array of 6 values evenly spaced between 5 and 20
spacedNumbers = np.linspace(5, 20, 6)
print(spacedNumbers)
```

```
[ 5.  8. 11. 14. 17. 20.]
```


Creating NumPy Arrays (4)

- `random.random(`*`numberOfRows, numberOfColumns`*`)`
  - Create an array of uniformly distributed random values between 0 and 1
- `random.normal(`*`mean, standardDeviation, size`*`)`
  - Create an array of normally distributed random values with *mean* and *standard deviation*
- `random.randint`*`(lowerBound, upperBound, size)`*
  - *Create an array of random integers in the interval*

Creating NumPy Arrays (4)

```
# Create a 2x6 array of uniformly distributed
# random values between 0 and 1
randNumbers = np.random.random((2, 6))
print(randNumbers)
```

```
[[0.62425223 0.58639327 0.90045935 0.65651807 0.61259153 0.95764916]
 [0.57857881 0.51963987 0.52926542 0.12380843 0.69996986 0.24306753]]
```

```
# Create a 3x5 array of normally distributed random values
# with mean 0 and standard deviation 1
normalRandNumbers = np.random.normal(0, 1, (3, 5))
print(normalRandNumbers)
```

```
[[ 1.90726418  0.85365738  1.22608166 -0.3359527  -0.28892326]
 [-1.58078791  1.1592405   0.25509415  0.75614964  0.97189673]
 [ 0.07902027 -2.62662344 -0.67627699  1.98106937 -0.49556422]]
```

```
# Create a 3x2 array of random integers in the interval 1 and 53
intRandNumbers = np.random.randint(1, 53, (3, 2))
print(intRandNumbers)
```

```
[[19 32]
 [20 41]
 [43 35]]
```


NumPy Array Attributes

- ndim: number of dimensions
- shape: the size of each dimension
- size: the total size of the array
- dtype: the data type of the array items

```
# Create a 3X4X5 array of random integers in the interval 1 and 100
threeDArray = np.random.randint(1, 100, (3, 4, 5))
print("arry dimension: ", threeDArray.ndim)
print("array shape:", threeDArray.shape)
print("array size: ", threeDArray.size)
print("array data type: ", threeDArray.dtype)
```

```
arry dimension:  3
array shape: (3, 4, 5)
array size:  60
array data type:  int32
```

Accessing Items with Index (1)

- In a one-dimensional array, the *i*th value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists
- NumPy slicing syntax follows that of the standard Python list

Accessing Items with Index (2)

```python
# some array
randArray = np.random.randint(1, 100, (10))
print(randArray)

print('The third: '+ str(randArray[2]))
print('The second from the right: '+ str(randArray[-2]))
print('First three item: ', randArray[:3])
print('Item 5 to 7: ', randArray[4:7])
print('update the value of the 3 item as 200')
randArray[2] =200 # assign the value to a specific item
print(randArray)
```

```
[32 23  3 87 40 29 15 31 90 51]
The third: 3
The second from the right: 90
First three item:  [32 23  3]
Item 5 to 7:  [40 29 15]
update the value of the 3 item as 200
[ 32  23 200  87  40  29  15  31  90  51]
```

Question: What's the result?

```
import numpy as np
twoDArray = np.random.randint(1, 100, (2,5))

print (twoDArray)

print(twoDArray[0,1])
print(twoDArray[1,-1])
```

```
[[77 14 92  3 10]
 [84 85 31 53 34]]
```


UNLV

Accessing Items with Index (2)

- We can also slice multi-dimensional arrays

```
twoDArray = np.random.randint(1, 100, (6,7))
print(twoDArray)

print(twoDArray[:2, :3]) # first two rows, first three columns
print(twoDArray[:,-1:]) # all rows, last column
```

```
[[34 20 59 82 31  2 69]
 [ 2 85 73 82 32 37 91]
 [85 82 53 67 91 91 92]
 [85 69 76 96 25 37 17]
 [ 5 77 75 50 62 60 27]
 [97 87 90 92 55  8  3]]
[[34 20 59]
 [ 2 85 73]]
[[69]
 [91]
 [92]
 [17]
 [27]
 [ 3]]
```


UNLV

Subarrays as No-Copy Views (1)

- One important–and extremely important–feature about array slices is that they return *views* rather than *copies* of the array data.
  - Key aspect of NumPy array slicing that differs from Python list slicing
  - When the value in the sliced subarray is updated, the value is reflected in the original array as well
- It is useful when we work with large datasets. We can access and process pieces of these datasets without the need to copy the entire dataset.

Subarrays as No-Copy Views (2)

```python
twoDArray = np.random.randint(1, 100, (6,7))
print(twoDArray)

subArray = twoDArray[:2, :2] # 2X2 subarray
print(subArray)

print('Update subarray[1,1] as 300')
subArray[1,1]=300

print(twoDArray)
```

```
[[72 33 98 37 11 74  7]
 [33 66 49 71 40  9 30]
 [ 2 39 32 66 11 50 75]
 [43 13 71 89 19 60 88]
 [ 4 21 47 32 51 18 61]
 [54 49 67 41 33  1 14]]
[[72 33]
 [33 66]]
Update subarray[1,1] as 300
[[ 72  33  98  37  11  74   7]
 [ 33 300  49  71  40   9  30]
 [  2  39  32  66  11  50  75]
 [ 43  13  71  89  19  60  88]
 [  4  21  47  32  51  18  61]
 [ 54  49  67  41  33   1  14]]
```

Subarrays as No-Copy Views (3)

- It is sometimes useful to instead explicitly copy the data within an array or a subarray

```
twoDArray = np.random.randint(1, 100, (6,7))
print(twoDArray)

subArrayCopy = twoDArray[:2, :2].copy() # 2X2 subarray, but as a copy
print(subArrayCopy)

print('Update subarrayCopy[1,0] as 400')
subArrayCopy[1,0]=400

print('Updated copied subarray')
print(subArrayCopy)
print('Original Array:')
print(twoDArray)
```

```
[[98 43 35 88 63 80 97]
 [53 25 23 59 90 21 61]
 [57 98 33 29 88 68 43]
 [19 20  9 44 69 93 78]
 [79 21 88  1 73 77 78]
 [87 48 37  7 97 13 99]]
[[98 43]
 [53 25]]
Update subarrayCopy[1,0] as 400
Updated copied subarray
[[ 98  43]
 [400  25]]
Original Array:
[[98 43 35 88 63 80 97]
 [53 25 23 59 90 21 61]
 [57 98 33 29 88 68 43]
 [19 20  9 44 69 93 78]
 [79 21 88  1 73 77 78]
 [87 48 37  7 97 13 99]]
```


Reshape Array

- `reshape(`*`newRowSize, newColumnSize`*`)`
  – the size of the initial array must match the size of the reshaped array

```
twoDArray = np.random.randint(1, 100, (2,5))

print(twoDArray)
# row vector via reshape
row=twoDArray.reshape((1, 10))
print(row)
```

```
[[34 41  4 89  4]
 [60 76  9 29 28]]
[[34 41  4 89  4 60 76  9 29 28]]
```

Array Concatenation (1)

- np.concatenate(*array1, array2, ….*)
  - For uni-dimensional array, add the items to the same array
  - For multi-dimensional array, concatenate along a specific axis

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
grid2 = np.array([[11, 12, 13],
                  [14, 15, 16]])

# concatenate along the first axis (i.e., adding rows)
print(np.concatenate([grid, grid], axis=0))

# concatenate along the first axis (i.e., adding columns)
print(np.concatenate([grid, grid2], axis=1))
```

```
[[1 2 3]
 [4 5 6]
 [1 2 3]
 [4 5 6]]
[[ 1  2  3 11 12 13]
 [ 4  5  6 14 15 16]]
```


Array Concatenation (2)

- np.vstack (array1, array2, ….)
- np.hstack (array1, array2, ….)
  - Concatenarate arrays of mixed dimensions

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

# vertically stack the arrays
print(np.vstack([x, grid]))

# horizontally stack the arrays
y = np.array([[99],
              [99]])
print(np.hstack([grid, y]))
```

```
[[1 2 3]
 [9 8 7]
 [6 5 4]]
[[ 9  8  7 99]
 [ 6  5  4 99]]
```

Computation on NumPay Arrays

- Universal Functions
  - Vectorized operations to improve the performance of calculation
  - Instead of using for loops to process each item in an array, the universal function can be used to make repeated calculations on array elements much more efficient
- Array arithmetic
- Absolute value
- Exponents and logarithms
- Other functions: Trigonometric functions

Array Arithmetic (1)

- Standard addition, subtraction, multiplication, and division

```
x = np.arange(5)
print("x     =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)   # floor division
print("x ** 2 =", x ** 2)   # exponentiation
print("x % 2  =", x % 2)   # modulus
```

```
x     = [0 1 2 3 4]
x + 5 = [5 6 7 8 9]
x - 5 = [-5 -4 -3 -2 -1]
x * 2 = [0 2 4 6 8]
x / 2 = [0.  0.5 1.  1.5 2. ]
x // 2 = [0 0 1 1 2]
x ** 2 = [ 0  1  4  9 16]
x % 2  = [0 1 0 1 0]
```

Array Arithmetic (2)

- The operators can be used together in an expression.
  - Standard order of operations is respected

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)
	np.sqrt	Square root (e.g. np.sqrt(9) = 3

- Lab
  - Grade Curving
    Write a program that reads a file with scores and applied an equation to curve the grade. The result contains the original and curved grade is written to a new file.

UNLV

- Exercise
  - Height Converter
    Please write a program that read a file with some heights in centimeters. Please convert the heights into feet and inches. The result contains the original and converted height should be written to a new file and separated with a comma.

NumPy (2)

Han-fen Hu

UNLV

Outline

- Computation on NumPay Arrays
- Aggregations
- Broadcasting
- Comparisons

UNLV

Computation on NumPay Arrays

- Universal Functions
  - Vectorized operations to improve the performance of calculation
  - Instead of using for loops to process each item in an array, the universal function can be used to make repeated calculations on array elements much more efficient
- Array arithmetic
- Absolute value
- Exponents and logarithms
- Other functions: Trigonometric functions

Absolute Value

- np.absolute()
  np.abs()
  - Returns the absolute values of the items

```
x = np.array([-2, -1, 0, 1, 2])
print(np.absolute(x))
```

```
[2 1 0 1 2]
```

Exponents and Logarithms

- ufunc also provides an efficient way to do exponentials and logarithms

```
x = [1, 2, 4, 10]
print("x      =", x)
print("e^x    =", np.exp(x))
print("3^x    =", np.power(3, x))
print("ln(x)     =", np.log(x)) #natural logarithm
print("log2(x)   =", np.log2(x)) #base-2 logarithm
print("log10(x) =", np.log10(x)) #base-10 logarithm
```

```
x     = [1, 2, 4, 10]
e^x   = [2.71828183e+00 7.38905610e+00 5.45981500e+01 2.20264658e+04]
3^x   = [    3     9    81 59049]
ln(x)    = [0.         0.69314718 1.38629436 2.30258509]
log2(x)  = [0.         1.         2.         3.32192809]
log10(x) = [0.         0.30103    0.60205999 1.        ]
```


Aggregations

- Summing the Values in an Array
  - Multi dimensional aggregates
  - Other aggregation functions
- Minimum and Maximum

Lab

- Average Height of US Presidents
  This program reads the height data from a csv file and show the statistics.
- NOTE: This program uses pandas and matplotlib that we will cover later in this class

UNLV

Exercise

- Grade Curving
- In the curvedGrade.csv file, the first column is the original scores, and the second column shows the curved scores.

  Use the curvedGrade.csv and show how the curving changes the distribution of the scores, including the min, max, mean, standard deviation, and the median.

UNLV

Binary Operation on Arrays

- Binary operation on NumPy arrays are performed on an element-by-element basis

```
import numpy as np

a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
sum = a+b
print(sum)

```

```
[5 6 7]
```

```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])

difference = b-a
print(difference)
```

```
[5 4 3]
```


Broadcasting

- A set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.)
- Apply to arrays of different sizes
  - Here the one-dimensional array `a` is stretched, or broadcast across the second dimension in order to match the shape of `c`

```
a = np.array([0,1,2])
c = np.array([[23,59,61],
              [55,46,69],
              [18,43,36]])

unevenSum = a+c
print(unevenSum)
```

```
[[23 60 63]
 [55 47 71]
 [18 44 38]]
```

Rules of Broadcasting (1)

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

UNLV

Question/ Exercise (1)

- What is the result of m+a?

```
m = np.ones((2, 3))
a = np.array([0,1,2])

print(m)
print(a)

print(m+a)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
[0 1 2]
```

UNLV

Question/ Exercise (2)

- What is the result of a+b?

```
a = ([[0],[1],[2]])
b = np.arange(3)
print(a)
print(b)
print(a+b)
```

```
[[0], [1], [2]]
[0 1 2]
```

[[0]
[1]
[2]]

[0 1 2]

[[0] [0] [0]
[1] [1] [1]
[2] [2] [2]]

[0 1 2
0 1 2
0 1 2]

Question/ Exercise (3)

- What is the result of m+a?

```
m = np.ones((3, 2))
a = np.arange(3)

print(m)
print(a)

print(m+a)
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
[0 1 2]
```

Rules of Broadcasting (2)

- Broadcasting rules apply to any binary ufunc

```
ages = np.array([[80,86,88,30],
                 [50,62,75,23],
                 [66,58,40,36],
                 [98,68,93,40]])
# using the mean aggregate across the first dimension
ageMean = ages.mean(0)
print(ageMean)

# Centering the array
agesCentered = ages - ageMean
print(agesCentered)
```

```
[73.5  68.5  74.   32.25]
[[  6.5   17.5   14.    -2.25]
 [-23.5   -6.5    1.    -9.25]
 [ -7.5  -10.5  -34.     3.75]
 [ 24.5   -0.5   19.     7.75]]
```

UNLV

- Lab
  - Curving a Series of Scores

    The program reads the original scores from a file and then ask the user to enter the percentage of curving he/she wants to apply. The program then prints the updated score.

    NOTE: This program uses pandas that we will cover later in this class

UNLV

Comparison Operators as ufuncs (1)

- NumPy also implements comparison operators as element-wise ufuncs.
  - All six of the standard comparison operations are available
- The result of these comparison operators is always an array with a Boolean data type.

UNLV

Comparison Operators as ufuncs (2)

```
x = np.array([1, 2, 3, 4, 5])
print(x < 3)
```

```
[ True  True False False False]
```

```
x = np.array([1, 2, 3, 4, 5])
print(x > 3)
```

```
[False False False  True  True]
```

```
x = np.array([1, 2, 3, 4, 5])
print(x >= 3)
```

```
[False False  True  True  True]
```

```
x = np.array([1, 2, 3, 4, 5])
print(x <= 3)
```

```
[ True  True  True False False]
```

```
x = np.array([1, 2, 3, 4, 5])
print(x != 3)
```

```
[ True  True False  True  True]
```

```
x = np.array([1, 2, 3, 4, 5])
print(x == 3)
```

```
[False False  True False False]
```


UNLV

Comparison Operators as ufuncs (3)

- Element-wise comparison of two arrays

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

print (x == y)
```

```
[False False  True False False]
```

- Use the comparison with other functions

```
ages = np.array([80,86,88,30])

print(ages >= ages.mean(0))

```

```
[ True  True  True False]
```

UNLV

Comparison Operators as ufuncs (3)

- Comparison Operator will work on arrays of any size and shape

```
ages = np.array([[80,86,88,30],
                 [50,62,75,23],
                 [66,58,40,36],
                 [98,68,93,40]])

# mean of the mean
print(ages.mean(0).mean(0))
print(ages >= ages.mean(0).mean(0))
```

```
62.0625
[[ True  True  True False]
 [False False  True False]
 [ True False False False]
 [ True  True  True False]]
```

UNLV

Working with Boolean Arrays (1)

- Counting entries with `np.sum()`
  - False is interpreted as 0, and True is interpreted as 1

```
ages = np.array([[80,86,88,30],
                 [50,62,75,23],
                 [66,58,40,36],
                 [98,68,93,40]])

# mean of the mean
print(ages.mean(0).mean(0))
aboveAve = np.sum(ages >= ages.mean(0).mean(0))
print(aboveAve)
```

```
62.0625
8
```

UNLV

Working with Boolean Arrays (2)

- Check whether any or all the values are true using `np.any()` or `np.all()`

```
ages = np.array([[80,86,88,30],
                 [50,62,75,23],
                 [66,58,40,36],
                 [98,68,93,40]])
print(np.all(ages > 40))
print(np.all(ages > 20))
```

```
False
True
```

```
ages = np.array([[80,86,88,30],
                 [50,62,75,23],
                 [66,58,40,36],
                 [98,68,93,40]])
print(np.any(ages > 90))
print(np.any(ages < 20))
```

```
True
False
```

UNLV

- Lab
  - Counting Rainy Days
    The program read a file containing a series of data that represents the amount of precipitation each day for a year in a given city.

NumPy (3)

Han-fen Hu

Outline

- Boolean Logic
- Masks
- Fancy Indexing
- Fast Sorting Arrays
- Partial Sorts
- Structured Arrays

Boolean Operators (1)

- Bitwise logic operators can be used together with the comparison operators
  - & : and
  - | : or
  - ~ : not
  - ^ : xor (exclusive or)
    - the result evaluates to True if only exactly *one* of the value is True.

a	b	a ^ b
False	False	False
True	False	True
False	True	True
True	True	False

UNLV

Boolean Operators (2)

- Using the Keywords `and/or` Versus the Operators `&/|`
  - `and` and `or` gauge the truth or falsehood of entire object,
  - `&` and `|` refer to elements within each object

```
a = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
b = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
print(a | b)
```

```
[ True  True  True False  True  True]
```

```
a = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
b = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
print(a or b)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-55-5f3e72e4d095> in <module>
      1 a = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
      2 b = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
----> 3 print(a or b)

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

Comparisons and Masks (1)

- Boolean Arrays as Masks
  - Use Boolean arrays as masks, to select particular subsets of the data themselves
- Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion
  - Can be used to remove all outliers that are above some threshold

Comparisons and Masks (2)

- Masking operation
  - Index on a Boolean array to filter the data
  - Returns a one-dimensional array filled with all the values that meet this condition; i.e., all the values in positions at which the mask array is True.

Comparisons and Masks: Example (1)

```
# generate integers (<10) of random number to fill a 3X4 array
x = np.random.randint(10, size=(3, 4))

print(x)
# a Boolean array showing whether the number is less than 5
print(x< 5)
# use the Boolean array to index the array x
print(x[x < 5])
```

```
[[3 0 7 7]
 [7 2 4 8]
 [5 9 3 0]]
[[ True  True False False]
 [False  True  True False]
 [False False  True  True]]
[3 0 2 4 3 0]
```


UNLV

Comparisons and Masks: Example (2)

```
ages = np.array([[80,86,88,30],
                 [50,62,75,23],
                 [66,58,40,36],
                 [98,68,93,40]])
print(ages <= 30)

youngAges = ages[ages <= 30]
print(youngAges)
```

```
[[False False False  True]
 [False False False  True]
 [False False False False]
 [False False False False]]
[30 23]
```


UNLV

- Lab
  - Counting Summer Rainy Days
    The program read a file containing a series of data that represents the amount of precipitation each day for a year in a given city.

    Use a mask to show rainy days in the summer.

UNLV

Fancy Indexing

- Ways to access and modify portions of arrays
  - Index
  - Slicing
  - Boolean masks
  - Fancy indexing
    - Pass arrays of indices in place of single scalars

UNLV

Fancy Indexing: Example (1)

```
import numpy as np

x = np.random.randint(10, size=10)
print(x)
ind = [3, 7, 4]
print(x[ind])
```

```
[6 5 8 4 3 8 3 0 6 4]
[4 0 3]
```

```
x = np.random.randint(10, size=10)
print(x)
ind = np.array([[3, 7],
                [4, 5]])
print(x[ind])
```

```
[2 7 9 3 3 3 5 6 7 0]
[[3 6]
 [3 3]]
```

When using fancy indexing, the shape of the result reflects the shape of the *index arrays*
rather than the shape of the *array being indexed*

Fancy Indexing: Example (2)

```
x = np.array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
print(x[row, col])
```

```
[ 2  5 11]
```

- Like with standard indexing, the first index refers to the row, and the second to the column
  - The first value in the result is x[0, 2],
  - The second is x[1, 1],
  - and the third is x[2, 3]

Combined Indexing (1)

- Combine fancy and simple indices
  - What is the result?

```
x = np.array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
print(x[2, [2, 0, 1]])
```


UNLV

Combined Indexing (2)

- Combine fancy indexing with slicing
  - What is the result?

```
x = np.array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
print(x[1:, [3, 1, 2]])
```


UNLV

Combined Indexing (3)

- Combine fancy indexing with masking
  - What is the result?

```
x = np.array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
row = np.array([[0],
                [1]])
mask = np.array([1, 0, 1, 0], dtype=bool)

print(x[row, mask])
```


UNLV

Updating Values with Fancy Indexing (1)

- Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array

```
x = np.arange(10)
i = np.array([2, 1, 8, 4])
x[i] = 99
print(x)
```

```
[ 0 99 99  3 99  5  6  7 99  9]
```


UNLV

Updating Values with Fancy Indexing (2)

- Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array

```
x = np.arange(10)
i = np.array([2, 1, 8, 4])
# assign the value 99 to index 2, 1, 8, 4
x[i] = 99
print(x)
# subtract 10 from index 2, 1 8, 4
x[i] -= 10
print(x)
```

```
[ 0 99 99  3 99  5  6  7 99  9]
[ 0 89 89  3 89  5  6  7 89  9]
```

UNLV

Fast Sorting in NumPy

- `np.sort()`: return a sorted array
- `np.argsort()`: return the *indices* of the sorted elements

```
x = np.random.randint(10, size=10)
print(x)
# print the sorted array
print(np.sort(x))
# print the indices of the osrted elements
print(np.argsort(x))
```

```
[1 1 3 2 2 5 6 1 0 4]
[0 1 1 1 2 2 3 4 5 6]
[8 0 1 7 3 4 2 9 5 6]
```

UNLV

Sorting along Rows or Columns

- Adding the axis argument to the `sort()` function

```
s = np.random.randint(0, 10, (4, 6))
print(s)
print('sorted by row (within column)')
print(np.sort(s, axis=0))
print('sorted by column (within row)')
print(np.sort(s, axis=1))
```

```
[[6 0 4 2 1 5]
 [1 6 0 5 4 7]
 [7 2 9 2 7 5]
 [5 0 4 5 3 5]]
sorted by row (within column)
[[1 0 0 2 1 5]
 [5 0 4 2 3 5]
 [6 2 4 5 4 5]
 [7 6 9 5 7 7]]
sorted by column (within row)
[[0 1 2 4 5 6]
 [0 1 4 5 6 7]
 [2 2 5 7 7 9]
 [0 3 4 5 5 5]]
```


Question

- How do you sort an array by row and column?

```
s = np.random.randint(0, 10, (4, 6))
print(s)
print('sorted by row and column')
print(np.sort(np.sort(s, axis=0), axis=1))

```

Sort an Array in Descending Order

- Apply the `sort()` function and negative (-) ufunc

```
# a random number array with 15 values between 0 and 99

s = np.random.randint(0, 100, (15))
print(s)

print('transform the values to the opposite sign and then sort it')
print(np.sort(-s))

print('transform the sorted values to the opposite sign')
print(-np.sort(-s))

print('Completed reverse sorting')
```

```
[95 53 23 71 84 43 98 25 52 95 61 95 15 83 80]
transform the values to the opposite sign and then sort it
[-98 -95 -95 -95 -84 -83 -80 -71 -61 -53 -52 -43 -25 -23 -15]
transform the sorted values to the opposite sign
[98 95 95 95 84 83 80 71 61 53 52 43 25 23 15]
Completed reverse sorting
```


Partial Sorts: Partitioning (1)

- Sometimes we're not interested in sorting the entire array, but simply want to find the *k* smallest values in the array
- `np.partition()`
  - Takes an array and a number *K*
  - The result is a new array with the smallest *K* values to the left of the partition, and the remaining values to the right in arbitrary order
  - Can partition along different axis of a multidimensional array

Partial Sorts: Partitioning (2)

```python
import numpy as np
# a random number array with 10 values between 0 and 99
s = np.random.randint(0, 100, (10))
print("Original array: ")
print(s)
# partition to get the smallest 3 number
p = np.partition(s, 3)
print("particall sorted array: ")
print(p)
```

```
Original array:
[90 37 54 65 50 25 78 39  1 83]
particall sorted array:
[37  1 25 39 50 54 78 65 83 90]
```

The first three values in the resulting array are the three smallest in the array. The remaining array positions contain the remaining values

Partial Sorts: Partitioning (3)

```python
# a 4x6 random number array with values between 0 and 99
twoDArray = np.random.randint(0, 100, (4,6))
print("Original array: ")
print(twoDArray)

# partition to get the smallest 3 number
# when axis is not provided, it is default to axis = 0
# sorted by row value
p = np.partition(twoDArray, 3, axis = 0)
print("particall sorted array (the 3 smallest value of each column on the top): ")
print(p)

# partition to get the smallest 3 number
# when axis  = 1, sorted by column value
p2 = np.partition(twoDArray, 3, axis = 1)
print("particall sorted array (the 3 smallest value of each row on the left): ")
print(p2)
```

```
Original array:
[[67 31 46  5 88 75]
 [28 50  6 22 82 22]
 [15 69 66 97 28 79]
 [54  9 17  9 20 64]]
particall sorted array (the 3 smallest value of each column on the top):
[[15  9  6  5 28 22]
 [28 31 17  9 20 64]
 [54 50 46 22 82 75]
 [67 69 66 97 88 79]]
particall sorted array (the 3 smallest value of each row on the left):
[[ 5 46 31 67 88 75]
 [22  6 22 28 50 82]
 [28 15 66 69 79 97]
 [ 9  9 17 20 54 64]]
```

axis = 1
the first three slots in each row contain the smallest values from that row

Partial Sorts: Partitioning (4)

- `np.argpartition()`
  - Similar to how `np.argsort()` works
  - return the *indices* of the sorted elements

UNLV

Structured Arrays (1)

- To store heterogeneous data
  - As compared to `np.array()` that stores only data of the same type
- Arrays with compound data types
- Store associated data in the same array

UNLV

Example: Regular Array for Associated Date

- Three arrays
- Use the index to reference values for the same individual

```python
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
salary = [55000.0, 85500.0, 68000.0, 61500.0]

print('First individual : ')
print(name[0]+", "+str(age[0])+", "+str(salary[0]))
print('\nAll: ')
for i in range(len(name)):
    print(name[i]+", "+str(age[i])+", "+str(salary[i]))
```

```
First individual :
Alice, 25, 55000.0


All:
Alice, 25, 55000.0
Bob, 45, 85500.0
Cathy, 37, 68000.0
Doug, 19, 61500.0
```


UNLV

Example: Structured Array for Associated Date

- Manage only one array
- Efficient: Arranged together in one convenient block of memory

```python
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
salary = [55000.0, 85500.0, 68000.0, 61500.0]

data = np.zeros(4, dtype={'names':('name', 'age', 'salary'),
                          'formats':('U10', 'i4', 'f8')})

data['name'] = name
data['age'] = age
data['salary'] = salary
print('First individual : ')
print(data[0])
print('\nAll: ')
for person in data:
    print(person)
```

```
First individual :
('Alice', 25, 55000.)

All:
('Alice', 25, 55000.)
('Bob', 45, 85500.)
('Cathy', 37, 68000.)
('Doug', 19, 61500.)
```


UNLV

Structured Arrays (2)

- Data type used in defining the compound data type
  - U: Unicode string
  - i: integer
  - f: float
- Example
  - 'U10': Unicode string of maximum length 10
  - 'i4': 4-byte (i.e., 32 bit) integer
  - 'f8': 8-byte (i.e., 64 bit) float

Structured Arrays (2)

- Individual attributes are still available
- We can refer to values either by index or by name

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
salary = [55000.0, 85500.0, 68000.0, 61500.0]

data = np.zeros(4, dtype={'names':('name', 'age', 'salary'),
                          'formats':('U10', 'i4', 'f8')})
data['name'] = name
data['age'] = age
data['salary'] = salary
print('All names ')
print(data['name'])
print('\nName of last person: ')
print(data[-1]['name'])
```

```
All names
['Alice' 'Bob' 'Cathy' 'Doug']

Name of last person:
Doug
```

Structured Arrays (3)

- Boolean masking can be applied to filter data

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
salary = [55000.0, 85500.0, 68000.0, 61500.0]

data = np.zeros(4, dtype={'names':('name', 'age', 'salary'),
                          'formats':('U10', 'i4', 'f8')})
data['name'] = name
data['age'] = age
data['salary'] = salary
# Get names where salary is less then 65000
print(data[data['salary'] < 65000]['name'])
```

```
['Alice' 'Doug']
```

UNLV

Define Structured Array Data Types

- `np.dtype()`
- Method 1:
  – Provide the *names* and the *formats*

```
employeeDType = np.dtype({'names':('name', 'age', 'salary'),
                          'formats':('U10', 'i4', 'f8')})
print(employeeDType)
```

```
[('name', '<U10'), ('age', '<i4'), ('salary', '<f8')]
```

- Method 2
  – Each attribute is specified as a tuple

```
employeeDType = np.dtype([('name', 'U10'), ('age', 'i4'), ('salary', 'f8')])

print(employeeDType)
```

```
[('name', '<U10'), ('age', '<i4'), ('salary', '<f8')]
```

- Lab
  - Loyal Customer Lookup
    This program read the customer data from a file. The user can enter a criterion to filter customer data by the loyalty points earned.

Pandas (1)

Han-fen Hu

UNLV

Outline

- Introduction to Pandas
- Series Objects
- DataFrame Object
- Indexing and Selection
- Operating on Data in Pandas

UNLV

Introduction to Pandas

- Pandas is an open-source library providing high-performance, easy-to-use data structures and data analysis tools for the Python
- Designed to make working with "relational" or "labeled" data easy and intuitive
- https://pandas.pydata.org/

Pandas Objects

- Pandas objects are enhanced versions of NumPy structured arrays
  - Rows and columns are identified with labels
- Three fundamental Pandas data structures
  - Series: one-dimensional array of indexed data
  - DataFrame: a generalization of a NumPy array, or as a specialization of a Python dictionary
  - Index: an *immutable array* or as an *ordered set*

Pandas Series Object (1)

- One-dimensional array of indexed data
  - `Series` wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes

```python
import pandas as pd

data = pd.Series([0.25, 0.5, 0.75, 1.0])
print("The series object: ")
print(data)
print("\nThe values only: ")
print(data.values)
print("\nThe indices: ")
print(data.index)
```

```
The series object:
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64

The values only:
[0.25 0.5  0.75 1.  ]

The indices:
RangeIndex(start=0, stop=4, step=1)
```

UNLV

Pandas Series Object (2)

- Data can be accessed by the associated index via the familiar Python square-bracket notation
  - While the Numpy Array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values

```python
import pandas as pd

data = pd.Series([0.25, 0.5, 0.75, 1.0])
print("The second value: ")
print(data[1])
print("The second to third value: ")
print(data[1:3])
```

```
The second value:
0.5
The second to third value:
1    0.50
2    0.75
dtype: float64
```

UNLV

Pandas Series Object (3)

- The index need not be an integer, but can consist of values of any desired type

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
print(data)

print('\nThe value for index b: ')
print(data['b'])
```

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64

The value for index b:
0.5
```


Pandas Series Object (4)

- Series can be considered as specialized dictionary
  - A dictionary is a structure that maps arbitrary keys to a set of arbitrary values
  - A Series is a structure which maps typed keys to a set of typed values
    - Makes it much more efficient than Python dictionaries for certain operations
  - Unlike a dictionary, though, the Series also supports array-style operations such as slicing

Slicing of Series Object: Example

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)

print(population['California'])
print(population['California':'New York'])
```

```
38332521
California    38332521
Texas         26448193
New York      19651127
dtype: int64
```


UNLV

Creating Series Objects

- `pd.Series(data, index=index)`

```
# index defaults to an integer sequence
# data can be a list or array
data1= pd.Series([2, 4, 6])
print(data1)
```

```
0    2
1    4
2    6
dtype: int64
```

```
# data can be a scalar, which is repeated to fill the specified index
data2 = pd.Series(5, index=[100, 200, 300])
print(data2)
```

```
100    5
200    5
300    5
dtype: int64
```

```
# data can be a dictionary
# in which index defaults to the sorted dictionary keys
data3 = pd.Series({2:'a', 1:'b', 3:'c'})
print(data3)
```

```
2    a
1    b
3    c
dtype: object
```


UNLV

Pandas DataFrame Object (1)

- A generalization of a NumPy array
  - A two-dimensional array with both flexible row indices and flexible column names
  - A sequence of Series objects that share the same index.

```
population_dict = {'California': 38332521,'Texas': 26448193,'New York': 19651127,
                   'Florida': 19552860,'Illinois': 12882135}
population = pd.Series(population_dict)
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
states = pd.DataFrame({'population': population,
                       'area': area})
print(states)
```

```
            population    area
California    38332521  423967
Texas         26448193  695662
New York      19651127  141297
Florida       19552860  170312
Illinois      12882135  149995
```


Pandas DataFrame Object (2)

- DataFrame has an index attribute that gives access to the index labels
- DataFrame has a columns attribute, which is an Index object holding the column labels

```
population_dict = {'California': 38332521,'Texas': 26448193,'New York': 19651127,
                   'Florida': 19552860,'Illinois': 12882135}
population = pd.Series(population_dict)
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
states = pd.DataFrame({'population': population,
                       'area': area})
print(states.index)
print(states.columns)
```

```
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')
Index(['population', 'area'], dtype='object')
```

Pandas DataFrame Object (3)

- Can be considered as a generalization of a two-dimensional NumPy array
  - A DataFrame has labels for the columns
- Can also think of a DataFrame as a specialization of a dictionary
  - a DataFrame maps a column name to a Series of column data

UNLV

Creating DataFrame Objects (1)

- Method 1: From a single Series object

```
population_dict = {'California': 38332521,'Texas': 26448193,'New York': 19651127,
                   'Florida': 19552860,'Illinois': 12882135}
population = pd.Series(population_dict)

s = pd.DataFrame(population, columns=['population'])
print(s)
```

```
            population
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
```


UNLV

Creating DataFrame Objects (2)

- Method 2: From a list of dictionaries
  - Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e., "not a number") values

```
s = pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
print(s)
```

```
     a  b    c
0  1.0  2  NaN
1  NaN  3  4.0
```

- Method 3: From a dictionary of Series objects
  - See the example on slide 11&12

UNLV

Creating DataFrame Objects (3)

- Method 4: From a two-dimensional NumPy array

```
s = pd.DataFrame(np.random.rand(3, 2), # a 3x2 array of random numbers
                 columns=['foo', 'bar'], # column labels
                 index=['a', 'b', 'c']) # index
print(s)
```

```
        foo       bar
a  0.479797  0.440195
b  0.272726  0.658412
c  0.943578  0.228896
```

- Method 5: From a NumPy structured array

```
a = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
s = pd.DataFrame(a)
print(s)
```

```
   A    B
0  0  0.0
1  0  0.0
2  0  0.0
```

UNLV

Read Data from File into DataFrame (1)

- The first step to any data science project is to import your data
- `read_csv()` function
  - File path is the argument
    - full file path which is prefixed by a / and includes the working directory in the specification,
    - or use the relative file path which doesn't.
  - The data will be read into a DataFrame

UNLV

Read Data from File into DataFrame (2)

- The index will be automatically assigned.
- `set_index()`
  - Set the index to an existing column

UNLV

- Lab
  - Demographic Statistics
    The program read the data from a CSV file into a DataFrame and show the statistics

Data Selection in Series (1)

- Series object acts in many ways like a standard Python dictionary
  - Use the key to access the value
  - `in` and `not in` operator
  - Assign new value
  - Add a new key-value pair

Data Selection in Series (2)

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
print(data['b']) # use key to access the value
if 'a' in data: # the in operator
    print('a is in data')

# update the value of b
data['b']=1.5

# assign a new key-value pair
data['e'] = 1.25

print(data)
```

```
0.5
a is in data
a    0.25
b    1.50
c    0.75
d    1.00
e    1.25
dtype: float64
```


UNLV

Data Selection in Series (3)

- Series object acts in many ways like a one-dimensional NumPy array
  - Slices
    - slicing with an explicit index (i.e., data['a':'c']), the final index is included in the slice
    - slicing with an implicit index (i.e., data[0:2]), the final index is excluded from the slice
  - Masking
  - Fancy indexing

UNLV

Data Selection in Series (4)

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
# slicing by explicit index
print(data['a':'c'])
# slicing by implicit integer index
print(data[0:2])
# masking
print(data[(data > 0.3) & (data < 0.8)])
# fancy indexing
print(data[['a', 'e']])
```

```
a    0.25
b    0.50
c    0.75
dtype: float64
a    0.25
b    0.50
dtype: float64
b    0.50
c    0.75
dtype: float64
a    0.25
e     NaN
dtype: float64
```


Data Selection in DataFrame (1)

- DataFrame acts in many ways like a dictionary of Series structures sharing the same index
  - Use the column label to retrieve the values
  - Use the dictionary-style syntax to modify the object (e.g., add a new column)

Data Selection in DataFrame (2)

```python
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})

# get the value for the key "area"
print(data['area'])

# add a new key-value pair.
# the value here is a Series, calculated based on pop and area
data['density'] = data['pop'] / data['area']

print(data)
```

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
              area       pop     density
California  423967  38332521   90.413926
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Florida     170312  19552860  114.806121
Illinois    149995  12882135   85.883763
```


Data Selection in DataFrame (3)

- DataFrame acts in many ways like a two-dimensional or structured array
  - For array-style indexing, we need another convention: `loc`, `iloc`, and `ix`
    - `loc:` index the data using the explicit index and column names
    - `iloc:` index the array using the implicit Python-style index
  - With `loc` and `iloc`,
    - Can combine masking and fancy indexing
    - Can update the value

Data Selection in DataFrame (4)

```
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135
```

```python
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})

# use the explicit index and column names
print(data.loc[:'Illinois', :'pop'])
# use the implicit Python-style index
print(data.iloc[:3, :2])
```

```
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
```


Data Selection in DataFrame (5)

```
              area       pop     density
California  423967  38332521   90.413926
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Florida     170312  19552860  114.806121
Illinois    149995  12882135   85.883763
```

```python
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']

# use the masking and fancy indexing
print(data.loc[data.density > 100, ['pop', 'density']])
```

```
               pop     density
New York  19651127  139.076746
Florida   19552860  114.806121
```

Data Selection in DataFrame (6)

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135

```python
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})

data.iloc[0, 1] = 90
print(data)

```

```
            area       pop
California  423967        90
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135
```


Data Selection in DataFrame (7)

- Note that *slicing* and *mask* refers to rows

```python
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area']

print('slicing refers to rows. Using key')
print(data['Florida':'Illinois'])
print('\nslicing refers to rows. implicit index')
print(data[1:3])
print('\nMasking operations refers to rows')
print(data[data.density > 100])
```

```
slicing refers to rows. Using key
            area       pop     density
Florida   170312  19552860  114.806121
Illinois  149995  12882135   85.883763

slicing refers to rows. implicit index
            area       pop     density
Texas     695662  26448193   38.018740
New York  141297  19651127  139.076746

Masking operations refers to rows
            area       pop     density
New York  141297  19651127  139.076746
Florida   170312  19552860  114.806121
```

- Lab
  - Demographic Statistics (cont’d)
    The program read the data from a CSV file into a DataFrame and show the statistics

Operating on Data in Pandas (1)

- Pandas inherits much of this functionality from NumPy, and the ufuncs
  - Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects

Operating on Data in Pandas (2)

- For binary operations on two Series or DataFrame objects, Pandas will align indices in the process of performing the operation
  - The resulting array contains the *union* of indices of the two input arrays
  - Any item for which one or the other does not have an entry is marked with NaN, or "Not a Number,"

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                        'New York': 19651127}, name='population')

density = population / area
print(density)
```

```
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```


Operating on Data in Pandas (3)

```
a = pd.Series([2, 4, 6], index=[0, 1, 2])
b = pd.Series([1, 3, 5], index=[1, 2, 3])

print(a+b)
```

```
0    NaN
1    5.0
2    9.0
3    NaN
dtype: float64
```

```
a = pd.Series([2, 4, 6], index=[0, 1, 2])
b = pd.Series([1, 3, 5], index=[1, 2, 3])

# for NaN, fill in with 0
c = a.add(b, fill_value=0)
print(c)
```

```
0    2.0
1    5.0
2    9.0
3    5.0
dtype: float64
```

Operating on Data in Pandas (4)

Python Operator	Pandas Method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Operating on Data in Pandas (5)

- A similar type of alignment takes place for both columns and indices for operations on DataFrame

```
x = pd.DataFrame(np.random.randint(0, 20, (2, 2)),
                 columns=list('AB'))
print("Content of x")
print(x)
y = pd.DataFrame(np.random.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
print("\nContent of y")
print(y)
print("\nContent of x+y")
print(x+y)
```

```
Content of x
    A   B
0  19  18
1   9   4

Content of y
   B  A  C
0  4  8  0
1  2  0  2
2  6  7  2

Content of x+y
      A     B   C
0  27.0  22.0 NaN
1   9.0   6.0 NaN
2   NaN   NaN NaN
```

- Lab
  - Demographic Statistics (cont'd)
    The program read the data from a CSV file into a DataFrame and show the statistics

Pandas (2)

Outline

- Missing Data
- Operating on Null Values
- MultiIndex

Missing Data (1)

- Real-world data is rarely clean and homogeneous
- Missing Data Conventions
  - Using a *mask* that globally indicates missing values
  - Choosing a *sentinel value* that indicates a missing entry
    - Such as -9999 or some rare pattern.

UNLV

Missing Data (2)

- In Pandas
  - NaN (acronym for Not a Number) for missing numeric values
  - None object for others missing values
- Note: an Empty string ("") is not equivalent to NaN or None

UNLV

NaN: Missing Numerical Data (1)

- Acronym for Not a Number
- Special floating-point value
  - Can be used to represent missing floating-point value

```
import numpy as np # import numpy
import pandas as pd # import pandas

# declare a numeric array, the second element is a missing value
exampNumArray = np.array([1, np.nan, 3, 4])
# print the array
print(exampNumArray)
# print the data type of the array
print(exampNumArray.dtype)
```

```
[ 1. nan  3.  4.]
float64
```


NaN: Missing Numerical Data (2)

- Data Virus: Regardless of the operation, the result of arithmetic with `NaN` will be another `NaN`
  - Aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

```
import numpy as np
import pandas as pd

vals = np.array([1, np.nan, 3, 4])


print(vals.sum())
print(vals.max())
print(vals.min())
print(1+ vals)
print(0* vals)

```

```
nan
nan
nan
[ 2. nan  4.  5.]
[ 0. nan  0.  0.]
```

NaN: Missing Numerical Data (3)

- NumPy does provide some special aggregations that will ignore these missing values
  - np.nansum()
  - np.nanmin()
  - np.nanmax()

```python
import numpy as np
import pandas as pd

vals = np.array([1, np.nan, 3, 4])


print(np.nansum(vals))
print(np.nanmin(vals))
print(np.nanmax(vals))

```

```
8.0
1.0
4.0
```

UNLV

NaN and None

- NaN: Missing floating-point values
- None: Missing object value
  - For example, string
- Pandas is built to handle the two of them nearly interchangeably

Type	Conversion	Storing Value
floating	No change	np.nan
object/string	No change	None or np.nan
integer	Cast to float64	np.nan
boolean	Cast to object	None or np.nan

UNLV

Operating on Null Values (1)

- Several useful methods for detecting, removing, and replacing null values in Pandas data structures
  - `isnull()`: Generate a Boolean mask indicating missing values
  - `notnull()`: Opposite of isnull()
  - `dropna()`: Return a filtered version of the data
  - `fillna()`: Return a copy of the data with missing values filled or imputed

UNLV

Operating on Null Values (1): Example

```python
data = pd.Series([1, np.nan, 'hello', None])
# function that returns Boolean mask
print(data.isnull())
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

```python
data = pd.Series([1, np.nan, 'hello', None])
# function that returns Boolean mask
print(data.notnull())
```

```
0     True
1    False
2     True
3    False
dtype: bool
```

```python
# use the mask to filter data
print(data[data.notnull()])
# It is is equivalent to the following
print(data.dropna())
```

```
0        1
2    hello
dtype: object
0        1
2    hello
dtype: object
```

Detecting null values
The boolean array can be used for masking

Dropping null values

UNLV

Operating on Null Values (2)

- For a DataFrame, there are more options
  - We cannot drop single values from a DataFrame; we can only drop full rows or full columns
  - By default, dropna() will drop all rows in which any null value is present
  - Alternatively, we can drop NA values along a different axis

Operating on Null Values (2): Example

```python
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])

print(df.dropna())
```

```
     0    1  2
1  2.0  3.0  5
```

Default action
Drop the rows containing null value

```python
print(df.dropna(axis='columns'))
```

```
   2
0  2
1  5
2  6
```

Drop the columns containing null value

```python
print(df.dropna(axis='rows'))
```

```
     0    1  2
1  2.0  3.0  5
```

Explicitly specification
Drop the rows containing null value

Operating on Null Values (3)

- You might rather be interested in dropping rows or columns with *all* NA values, or a majority of NA values
  - how parameters
    - `how = 'all'`: drop rows/columns that are *all* null values
    - `how = 'any'`: drop rows/columns that contain *any* null values
  - `thresh` parameter
    - Specify a minimum number of non-null values for the row/column to be kept

UNLV

Operating on Null Values (3): Example

```
df = pd.DataFrame([[1,      np.nan, 2, np.nan],
                   [2,      3,      5, np.nan],
                   [np.nan, 4,      6, np.nan],
                   [np.nan, np.nan, 7, np.nan]])
# drop the columns with all na values
print(df.dropna(axis='columns', how='all'))
```

```
     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6
3  NaN  NaN  7
```

Drop the column
if all the values in the column is null

```
# drop the columns with any na values
print(df.dropna(axis='columns', how='any'))
```

```
   2
0  2
1  5
2  6
3  7
```

Drop the column
if any value in the column is null

```
# drop the rows with less than 2 valid values
print(df.dropna(axis='rows', thresh=2))
```

```
     0    1  2   3
0  1.0  NaN  2 NaN
1  2.0  3.0  5 NaN
2  NaN  4.0  6 NaN
```

Drop the row
if it contains more than 2 non-null values

UNLV

- Lab
  - Medical Tracking Data
  - The program reads a csv file containing the data for a medical experiment. Several thousands of people received a treatment and came back for 37 tests. The test results were recorded. There are missing data in the file.

Filling Null Values

- Sometimes rather than dropping NA values, we'd rather replace them with a valid value.
  - `fillna(value)`: fill NA entries with a single value
  - `fillna(method='ffill')`: forward-fill to propagate the previous value forward
  - `fillna(method='bfill')`: back-fill to propagate the next values backward

Filling Null Values: Example

```python
data = pd.Series([1, np.nan, 2,np.nan, 3],
                 index=list('abcde'))
print(data.fillna(0))
```

```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

```python
print(data.fillna(method='ffill'))
print(data.fillna(method='bfill'))
```

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

```python
df = pd.DataFrame([[1,      np.nan, 2, np.nan],
                   [2,      3,      5, np.nan],
                   [np.nan, 4,      6, np.nan],
                   [np.nan, np.nan, 7, np.nan]])
print(df.fillna(method='ffill', axis="columns"))
```

```
     0    1    2    3
0  1.0  1.0  2.0  2.0
1  2.0  3.0  5.0  5.0
2  NaN  4.0  6.0  6.0
3  NaN  NaN  7.0  7.0
```

The fills take place by taking the value of previous columns

Note that if there isn't a previous column, the null value remain unfilled.

- Lab
  - Vaccination Data
    The program read a csv file containing number of people received vaccinations (in thousands).
    There are missing data in the file.

- Exercise
  - BMI for US Presidents
    Write a program that reads president_heights_updated.csv, and shows the names of US presidents with the highest and lowest BMI from the dataset.

MultiIndex (1)

- Datasets are not limited to one-dimensional and two-dimensional
- MultiIndex data type contains multiple levels of indexing

```
California   2000    33871648
             2010    37253956
New York     2000    18976457
             2010    19378102
Texas        2000    20851820
             2010    25145561
```

MultiIndex (1): Example

```
import pandas as pd
import numpy as np

# stats and year
ind = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
# population for the states
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]

# convert the index to a MultiIndex
ind = pd.MultiIndex.from_tuples(ind)
# use the MultiIndex as the index for the data
pop = pd.Series(populations, index=ind)

pop
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```


MultiIndex (2)

- We can add dimensions to the data frame

```
import pandas as pd
import numpy as np

# stats and year
ind = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
# population for the states
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
# population for people under 18
minor= [9267089, 9284094,
           4687374, 4318033,
           5906301, 6879014]

# convert the index to a MultiIndex
ind = pd.MultiIndex.from_tuples(ind)
# use the MultiIndex as the index for the data
pop = pd.Series(populations, index=ind)
print(pop)
pop_df = pd.DataFrame({'total': pop,
                       'under18': minor})
print(pop_df)
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
                    total  under18
California 2000  33871648  9267089
           2010  37253956  9284094
New York   2000  18976457  4687374
           2010  19378102  4318033
Texas      2000  20851820  5906301
           2010  25145561  6879014
```

MultiIndex (3)

- all the ufuncs and other functionality for data frame work with hierarchical indices as well.

```
# apply ufunc for calculation
pop_df['ratio'] = pop_df['under18'] / pop_df['total']
print(pop_df)
```

```
                    total  under18     ratio
California 2000  33871648  9267089  0.273594
           2010  37253956  9284094  0.249211
New York   2000  18976457  4687374  0.247010
           2010  19378102  4318033  0.222831
Texas      2000  20851820  5906301  0.283251
           2010  25145561  6879014  0.273568
```

Pandas (3)

UNLV

Outline

- Combining Datasets
  - Concat
  - Merge
- Data Manipulation
  - Drop
  - Replace
- High-Performance Pandas: query()

UNLV

Combining Datasets

- Some of the most interesting studies of data come from combining different data sources
- Pandas includes functions and methods that make this sort of data wrangling fast and straightforward.

Concatenation with `pd.concat` (1)

- `pd.concat()`: By default, the concatenation takes place row-wise but allows specification of an axis

```python
df1=pd.DataFrame([['A1', 'B1'], ['A2', 'B2']], columns=list('AB'))
df2=pd.DataFrame([['C3', 'D3'], ['C4', 'D4']], columns=list('AB'))

print(df1)
print(df2)
print(pd.concat([df1, df2]))
print(pd.concat([df1, df2], axis=1))
```

```
    A   B
0  A1  B1
1  A2  B2
    A   B
0  C3  D3
1  C4  D4
    A   B
0  A1  B1
1  A2  B2
0  C3  D3
1  C4  D4
    A   B   A   B
0  A1  B1  C3  D3
1  A2  B2  C4  D4
```

Concatenation with pd.concat (2)

- pd.concat(): preserves indices, even if the result will have duplicate indices
  - To ignore the index, use the ignore_index flag

```
df1=pd.DataFrame([['A1', 'B1'], ['A2', 'B2']], columns=list('AB'))
df2=pd.DataFrame([['C3', 'D3'], ['C4', 'D4']], columns=list('AB'))

print(df1)
print(df2)
print(pd.concat([df1, df2],ignore_index=True))
```

```
    A   B
0  A1  B1
1  A2  B2
    A   B
0  C3  D3
1  C4  D4
    A   B
0  A1  B1
1  A2  B2
2  C3  D3
3  C4  D4
```

UNLV

Concatenation with Joins (1)

- In practice, data from different sources might have different sets of column names
  - By default, the entries for which no data is available are filled with NaN values

```
df3=pd.DataFrame([['A1','B1','C1'], ['A2','B2','C2']]
                  , columns=list('ABC')
                  , index = list('01'))
df4=pd.DataFrame([['B3','C3','D3'], ['B4','C4','D4']]
                  , columns=list('BCD')
                 , index = list('34'))

print(df3)
print(df4)
print(pd.concat([df3, df4]))
```

```
    A   B   C
0  A1  B1  C1
1  A2  B2  C2
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
     A   B   C    D
0   A1  B1  C1  NaN
1   A2  B2  C2  NaN
3  NaN  B3  C3   D3
4  NaN  B4  C4   D4
```

UNLV

Concatenation with Joins (2)

- We can specify the options of `join` and `join_axes` parameters of the concatenate function.
  - By default, the join is a union of the input columns (`join='outer'`)
  - We can change this to an intersection of the columns using `join='inner'`
  - We can also specify the index of the remaining columns using the `join_axes` argument

Concatenation with Joins: Example (1)

Get the intersection of the columns using `join='inner'`

```
df3=pd.DataFrame([['A1','B1','C1'], ['A2','B2','C2']]
                 , columns=list('ABC')
                 , index = list('01'))
df4=pd.DataFrame([['B3','C3','D3'], ['B4','C4','D4']]
                 , columns=list('BCD')
                , index = list('34'))

print(df3)
print(df4)
print(pd.concat([df3, df4], join='inner'))
```

```
    A   B   C
0  A1  B1  C1
1  A2  B2  C2
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
    B   C
0  B1  C1
1  B2  C2
3  B3  C3
4  B4  C4
```

Concatenation with Joins: Example (2)

Specify the index of the remaining columns using `join_axes`

```
df3=pd.DataFrame([['A1','B1','C1'], ['A2','B2','C2']]
                 , columns=list('ABC')
                 , index = list('01'))
df4=pd.DataFrame([['B3','C3','D3'], ['B4','C4','D4']]
                 , columns=list('BCD')
                , index = list('34'))

print(df3)
print(df4)
print(pd.concat([df3, df4], join_axes=[df3.columns]))
```

```
    A   B   C
0  A1  B1  C1
1  A2  B2  C2
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
     A   B   C
0   A1  B1  C1
1   A2  B2  C2
3  NaN  B3  C3
4  NaN  B4  C4
```

UNLV

Merge: One-to-One

- Merge two DataFrame objects
  - Recognize the common column and use the column as a key to merge
  - After merge, the order of entries in each column is not necessarily maintained
  - The merge in general discards the index

UNLV

One-to-One Merge: Example

```
emp1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
emp2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                     'hire_date': [2004, 2008, 2012, 2014]})
print(emp1)
print(emp2)
print(pd.merge(emp1, emp2))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```


UNLV

Merge: Many-to-One

- Many-to-one merge happens when one of the two key columns contains duplicate entries.
  - The resulting DataFrame will preserve those duplicate entries as appropriate.

UNLV

Many-to-One Merge: Example

```
emp1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
emp2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                     'hire_date': [2004, 2008, 2012, 2014]})
emp3 = pd.merge(emp1, emp2)
emp4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                     'supervisor': ['Carly', 'Guido', 'Steve']})
print(emp3)
print(emp4)
print(pd.merge(emp3, emp4))
```

```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
         group supervisor
0   Accounting      Carly
1  Engineering      Guido
2           HR      Steve
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

"supervisor" information is repeated as needed

Merge: Many-to-Many

- If the key column in both the left and right DataFrame contains duplicates, then the result is a many-to-many merge
  - Some values are repeated as needed

Many-to-Many Merge: Example

```
emp1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
emp5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                               'Engineering', 'Engineering', 'HR', 'HR'],
                     'skills': ['math', 'spreadsheets', 'coding', 'linux',
                                'spreadsheets', 'organization']})
print(emp1)
print(emp5)
print(pd.merge(emp1, emp5))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
         group        skills
0   Accounting          math
1   Accounting  spreadsheets
2  Engineering        coding
3  Engineering         linux
4           HR  spreadsheets
5           HR  organization
  employee        group        skills
0      Bob   Accounting          math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering        coding
3     Jake  Engineering         linux
4     Lisa  Engineering        coding
5     Lisa  Engineering         linux
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```


Keyword on

- on: specify the name of the key column for merge

```
emp1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
emp2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                     'hire_date': [2004, 2008, 2012, 2014]})
print(emp1)
print(emp2)
print(pd.merge(df1, df2, on='employee'))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

Keyword left_on and right_on

- left_on and right_on: merge two datasets with different column names

```
emp1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
emp3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'salary': [70000, 80000, 120000, 90000]})
print(emp1)
print(emp3)
print(pd.merge(emp1, emp3, left_on="employee", right_on="name"))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
   name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
  employee        group  name  salary
0      Bob   Accounting   Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```


Keyword left_index and right_index

- Rather than merging on a column, we can merge on an index

```
emp1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
emp2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                     'hire_date': [2004, 2008, 2012, 2014]})
emp1 = emp1.set_index('employee')
emp2 = emp2.set_index('employee')
print(emp1)
print(emp2)
print(pd.merge(emp1, emp2, left_index=True, right_index=True))
```

```
                group
employee
Bob        Accounting
Jake      Engineering
Lisa      Engineering
Sue                HR
          hire_date
employee
Lisa           2004
Bob            2008
Jake           2012
Sue            2014
                group  hire_date
employee
Bob        Accounting       2008
Jake      Engineering       2012
Lisa      Engineering       2004
Sue                HR       2014
```

drop() method (1)

- Drop a column from the DataFrame

```
emp1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
emp3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'salary': [70000, 80000, 120000, 90000]})
emp6 = pd.merge(emp1, emp3, left_on="employee", right_on="name").drop('name', axis="columns")
print(emp6)
```

```
  employee        group  salary
0      Bob   Accounting   70000
1     Jake  Engineering   80000
2     Lisa  Engineering  120000
3      Sue           HR   90000
```


Inner and Outer Merge (1)

- When a value appears in one key column but not the other, we need to consider how to merge the data

```
guest1 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                       'food': ['fish', 'beans', 'bread']},
                      columns=['name', 'food'])
guest2 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                       'drink': ['wine', 'beer']},
                      columns=['name', 'drink'])
print(guest1)
print(guest2)
print(pd.merge(guest1, guest2))
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
     name drink
0    Mary  wine
1  Joseph  beer
   name   food drink
0  Mary  bread  wine
```

By default, the result contains the *intersection* of the two sets of inputs;
this is what is known as an *inner join*

Inner and Outer Merge (2)

- By using the keyword how, we can define outer, left, and right join

```
guest1 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                       'food': ['fish', 'beans', 'bread']},
                      columns=['name', 'food'])
guest2 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                       'drink': ['wine', 'beer']},
                      columns=['name', 'drink'])

print(pd.merge(guest1, guest2, how = 'outer'))
print(pd.merge(guest1, guest2, how = 'left'))
print(pd.merge(guest1, guest2, how = 'right'))
```

```
     name   food
0   Peter   fish
1    Paul  beans
2    Mary  bread
     name drink
0    Mary  wine
1  Joseph  beer
```

```
     name   food drink
0   Peter   fish   NaN
1    Paul  beans   NaN
2    Mary  bread  wine
3  Joseph    NaN  beer
    name   food drink
0  Peter   fish   NaN
1   Paul  beans   NaN
2   Mary  bread  wine
     name   food drink
0    Mary  bread  wine
1  Joseph    NaN  beer
```


- Lab
  - US States Data
    The program reads data from three csv files, representing data from different sources.
    In the program, the data will be combined/merged into a single DataFrame.
    The program will rank US states and territories by their 2010 population density.

drop() method: Remove Rows (1)

- `drop()` method can also be used to drop a row from a DataFrame

```
emp = pd.DataFrame({'employee':['Bob', 'Jake', 'Lisa','Sue','Ann','John'],
                   'group':['Accounting', 'Engineering','Engineering','HR','RD','RD'],
                    'salary':[70000, 80000, 120000, 90000,85000,85000]})

emp = emp.set_index('employee')
emp1 = emp.drop("Bob")
emp1
```

employee	group	salary
Jake	Engineering	80000
Lisa	Engineering	120000
Sue	HR	90000
Ann	RD	85000
John	RD	85000

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000
4	Ann	RD	85000
5	John	RD	85000

drop() method: Remove Rows (2)

- `drop()` method can be used to drop multiple rows
  - Use fancy indexing (explicit index)

```
emp = pd.DataFrame({'employee':['Bob', 'Jake', 'Lisa','Sue','Ann','John'],
                   'group':['Accounting', 'Engineering','Engineering','HR','RD','RD'],
                    'salary':[70000, 80000, 120000, 90000,85000,85000]})

emp = emp.set_index('employee')
emp2 = emp.drop(["Bob", "Ann"])
emp2
```

employee	group	salary
Jake	Engineering	80000
Lisa	Engineering	120000
Sue	HR	90000
John	RD	85000

drop() method: Remove Rows (3)

- `drop()` method can be used to drop multiple rows
  - Use fancy indexing (implicit index)

```
emp = pd.DataFrame({'employee':['Bob', 'Jake', 'Lisa','Sue','Ann','John'],
                    'group':['Accounting', 'Engineering','Engineering','HR','RD','RD'],
                     'salary':[70000, 80000, 120000, 90000,85000,85000]})

emp = emp.set_index('employee')
emp3 = emp.drop(emp.index[[1, 4]])
print(emp3)

emp4 = emp.drop(emp.index[:3])
print(emp4)
```

```
                group  salary
employee
Bob        Accounting   70000
Lisa      Engineering  120000
Sue                HR   90000
John               RD   85000
         group  salary
employee
Sue         HR   90000
Ann         RD   85000
John        RD   85000
```


UNLV

Other Ways to Drop Rows from a DataFrame

- Use masking (on the value of a column)

```
emp = pd.DataFrame({'employee':['Bob', 'Jake', 'Lisa','Sue','Ann','John'],
                    'group':['Accounting', 'Engineering','Engineering','HR','RD','RD'],
                     'salary':[70000, 80000, 120000, 90000,85000,85000]})


emp5 = emp.loc[emp['employee'] != 'Bob']
print(emp5)
print('\n\n')

emp6 = emp.loc[emp['salary'] < 100000]
print(emp6)
```

```
  employee        group  salary
1     Jake  Engineering   80000
2     Lisa  Engineering  120000
3      Sue           HR   90000
4      Ann           RD   85000
5     John           RD   85000



  employee        group  salary
0      Bob   Accounting   70000
1     Jake  Engineering   80000
3      Sue           HR   90000
4      Ann           RD   85000
5     John           RD   85000
```

UNLV

Replace Values in a DataFrame

- `replace()` method can be used to update values in a DataFrame

```
emp = pd.DataFrame({'employee':['Bob', 'Jake', 'Lisa','Sue','Ann','John'],
                    'group':['Accounting', 'Engineering','Engineering','HR','RD','RD'],
                     'salary':[70000, 80000, 120000, 90000,85000,85000]})


emp = emp.replace('RD','R&D')
emp
```

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000
4	Ann	R&D	85000
5	John	R&D	85000

Replace NaN Values

- For the whole DtaFrame

```
data = pd.read_csv('bonus.csv')

data = data.replace(np.nan, -99999)
```

- For one column

```
emp['bonus'] = emp['bonus'].replace(np.nan, 0)
emp
```

	employee	group	salary	bonus
0	Bob	Accounting	70000	NaN
1	Jake	Engineering	80000	3000.0
2	Lisa	Engineering	120000	2000.0
3	Sue	HR	90000	15000.0
4	Ann	RD	85000	NaN
5	John	RD	0	NaN

	employee	group	salary	bonus
0	Bob	Accounting	70000	0.0
1	Jake	Engineering	80000	3000.0
2	Lisa	Engineering	120000	2000.0
3	Sue	HR	90000	15000.0
4	Ann	RD	85000	0.0
5	John	RD	0	0.0

Update a Cell in a DataFrame (1)

- Specify the index and column

```python
customer = pd.DataFrame({'name': ['Alan','Byona','Catherine','Dean','Franky'],
            'age': [30, 69, 40, 18, 22],
            'premium':[345,234,974,563,435]})

# update the first customer's age
customer.at[0,'age'] = 31
customer
```

	name	age	premium
0	Alan	31	345
1	Byona	69	234
2	Catherine	40	974
3	Dean	18	563
4	Franky	22	435

UNLV

Update a Cell in a DataFrame (2)

- Specify the condition and column

```python
customer = pd.DataFrame({'name': ['Alan','Byona','Catherine','Dean','Franky'],
            'age': [30, 69, 40, 18, 22],
            'premium':[345,234,974,563,435]})

# update Alan's age
customer.at[customer['name'] == 'Alan','age'] = 31
customer
```

	name	age	premium
0	Alan	31	345
1	Byona	69	234
2	Catherine	40	974
3	Dean	18	563
4	Franky	22	435

UNLV

Lab

- Player Salary Data
  The program reads data from a csv files, representing data of NBA players' salaries in different years. This program can show the salary for a certain year, and allows the user to update the salary.

DataFrame.sort_value()

- sort_value(by=column, asending=Boolean)

```python
df = pd.DataFrame({'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
                   'col2': [2, 1, 9, 8, 7, 4],
                   'col3': [0, 1, 9, 4, 2, 3]})
print(df)
print(df.sort_values(by='col1'))
```

```
  col1  col2  col3
0    A     2     0
1    A     1     1
2    B     9     9
3  NaN     8     4
4    D     7     2
5    C     4     3
  col1  col2  col3
0    A     2     0
1    A     1     1
2    B     9     9
5    C     4     3
4    D     7     2
3  NaN     8     4
```

`DataFrame.sort_value()`: Example

```
df = pd.DataFrame({'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
                   'col2': [2, 1, 9, 8, 7, 4],
                   'col3': [0, 1, 9, 4, 2, 3]})
print(df.sort_values(by=['col1', 'col2']))
```

```
  col1  col2  col3
0    A     2     0
1    A     1     1
2    B     9     9
3  NaN     8     4
4    D     7     2
5    C     4     3
```

```
  col1  col2  col3
1    A     1     1
0    A     2     0
2    B     9     9
5    C     4     3
4    D     7     2
3  NaN     8     4
```

Sort by two columns

```
df = pd.DataFrame({'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
                   'col2': [2, 1, 9, 8, 7, 4],
                   'col3': [0, 1, 9, 4, 2, 3]})
print(df.sort_values('col1', ascending=False))
```

```
  col1  col2  col3
4    D     7     2
5    C     4     3
2    B     9     9
0    A     2     0
1    A     1     1
3  NaN     8     4
```

Sort by col1, in descending order

UNLV

`DataFrame.query()` Method

- Can be used to filter data with conditions on multiple columns
  - A more efficient computation
  - Compared to the masking expression this is much easier to read and understand

UNLV

- Lab
  - US States Data (cont.)
    The program reads data from three csv files, representing data from different sources.
    In the program, the data will be combined/merged into a single DataFrame.
    The program will rank US states and territories by their 2010 population density.

UNLV

Visualization with Matplotlib (1)

UNLV

Outline

- Introduction to Matplotlib
- Simple Line Plots
- Simple Scatter Plots
- Histograms, Binnings, and Density
- Customizing Plot Legends and Colorbars

UNLV

Introduction to Matplotlib (1)

- Data visualization library built on NumPy arrays
  - Allows visual access to huge amounts of data in easily digestible visuals
  - Large user base and an active developer base
  - Predated Pandas by more than a decade, and thus is not designed for use with Pandas DataFrames
- Seaborn
  - Provides an API on top of Matplotlib that offers choices for plot style and color defaults
  - Defines simple high-level functions for common statistical plot types
  - Integrates with the functionality provided by Pandas DataFrames

Introduction to Matplotlib (2)

- Importing Matplotlib

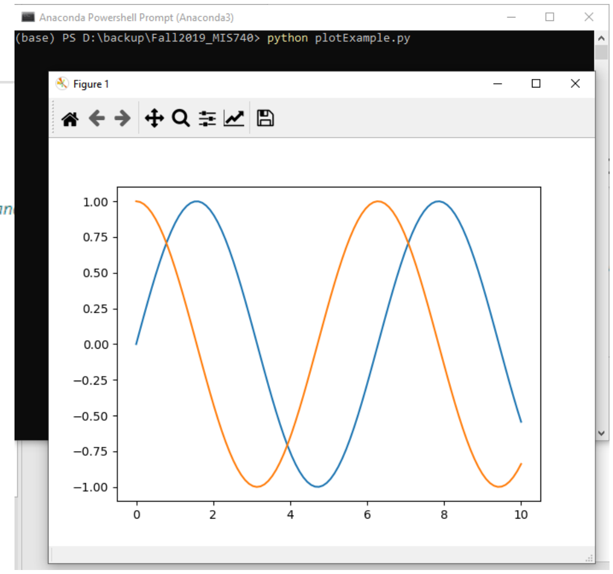
```
# import matplotlib, set the alias as mpl
import matplotlib as mpl
# import the pyplot module, set the alias as plt
import matplotlib.pyplot as plt
```

  - Pyplot is the most used module of Matplotlib
    - Provides an interface like MATLAB but instead, it uses Python and it is open source
- IPython is built to work well with Matplotlib

Introduction to Matplotlib (3)

- `plt.show()`
  - If we run the .py file from the shell, the `show()` function is needed to opens a window that display the figure

```
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as plt
import matplotlib.pyplot as plt

# an array of 100 numbers evenly distributed between 0 an
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```



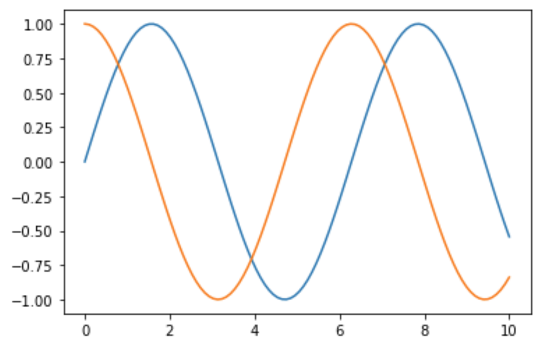
Introduction to Matplotlib (4)

- IPython is built to work well with Matplotlib if we specify Matplotlib mode
  - `%matplotlib inline` will lead to static images of the plot embedded in the notebook
  - Creating a plot will embed a PNG image of the resulting graphic
  - It needs to be done only once per kernel/session

Introduction to Matplotlib (5)

```
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as plt
import matplotlib.pyplot as plt

# an array of 100 numbers evenly distributed between 0 and 100
x = np.linspace(0, 10, 100)

%matplotlib inline
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
```

```
[<matplotlib.lines.Line2D at 0x1763436eac8>]
```



UNLV

Simple Line Plots (1)

- For all Matplotlib plots, we start by creating a figure and an axes

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# assign the figure object to a variable
fig = plt.figure()
# assign the axes object to a variable
ax = plt.axes()
```

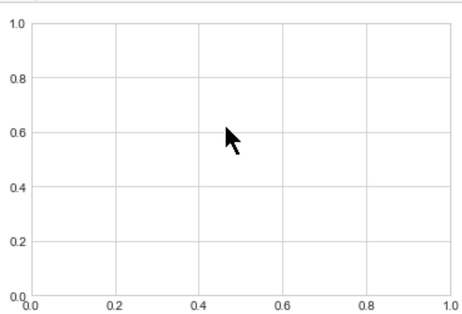


figure is a single container that contains all the objects representing axes, graphics, text, and labels.

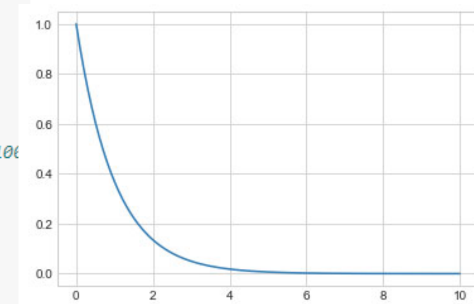
axes is a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization

UNLV

Simple Line Plots (1)

- Use the `plot()` function to draw the plot

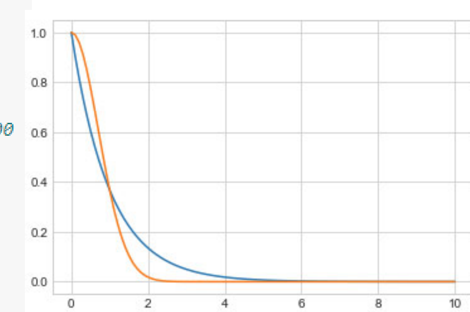
```python
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as plt
import matplotlib.pyplot as plt

%matplotlib inline

# an array of 100 numbers evenly distributed between 0 and 10
a = np.linspace(0, 10, 100)
# an array of exponential values of -a
b = np.exp(-a)

# use a as x axis and b as y axis for a line plot
plt.plot(a, b)
```



Simple Line Plots (3)

- To create a single figure with multiple lines, just simply call the plot function multiple times

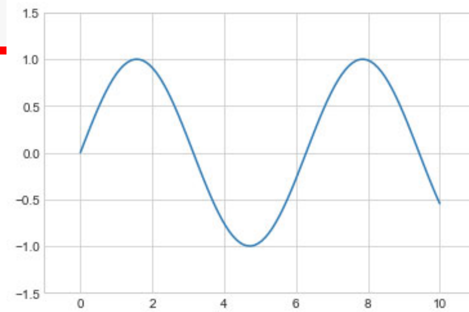
```python
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as plt
import matplotlib.pyplot as plt

# an array of 100 numbers evenly distributed between 0 and 100
a = np.linspace(0, 10, 100)
# an array of exponential values of -a
b = np.exp(-a)
c = np.exp(-a*a)
# use a as x axis and b as y axis for a line plot
plt.plot(a, b)
# call plot() again to add another line
plt.plot(a, c)
```



Adjusting the Plot: Axes Limits

- To adjust axis limits is to use the `plt.xlim()` and `plt.ylim()` methods

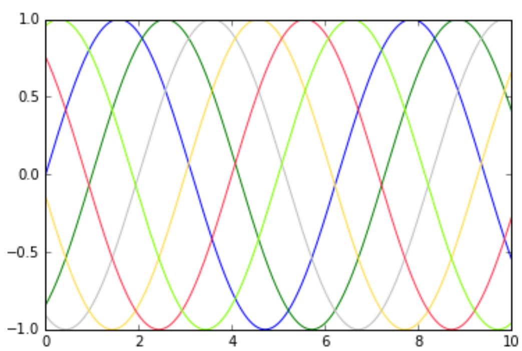
```
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as plt
import matplotlib.pyplot as plt

# an array of 100 numbers evenly distributed between 0 and 100
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))

plt.xlim(-1, 11) # set the x axis to -1 and 11
plt.ylim(-1.5, 1.5) # set the y axis to -1.5 and 1.5
```



Line Colors and Styles (1)

- `color` keyword: accepts a string argument representing virtually any imaginable color
  - If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines

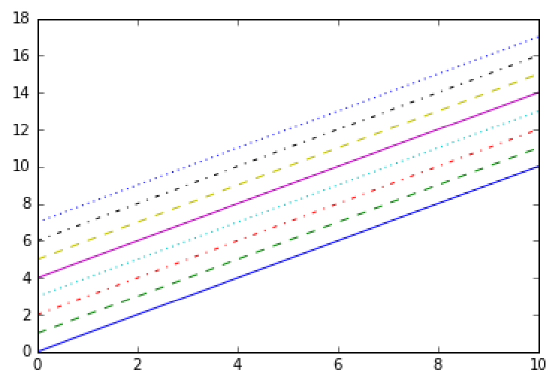
```
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x - 0), color='blue')        # specify color by name
plt.plot(x, np.sin(x - 1), color='g')           # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75')        # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')     # Hex code (RRGGBB from 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 to 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names supported
```



Line Colors and Styles (2)

- `linestyle` keyword: Specify the line style

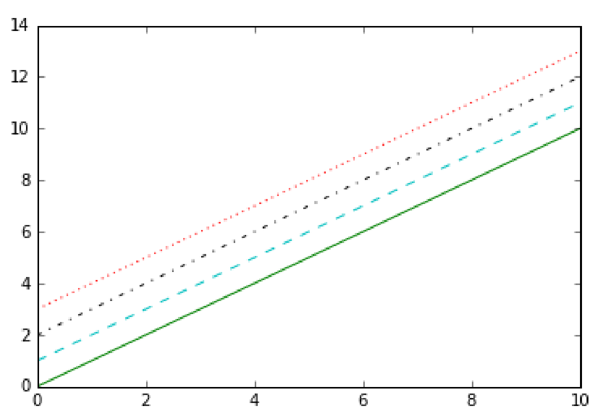
```
x = np.linspace(0, 10, 1000)
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');

# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-')  # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':');  # dotted
```



Line Colors and Styles (3)

- `linestyle` and `color` codes can be combined into a single non-keyword argument

```
x = np.linspace(0, 10, 1000)
plt.plot(x, x + 0, '-g')  # solid green
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot black
plt.plot(x, x + 3, ':r');  # dotted red
```



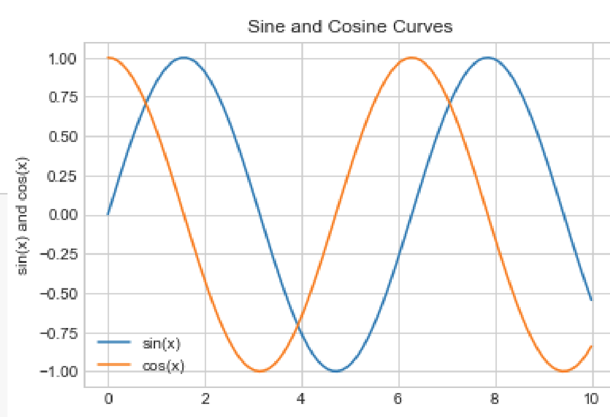
Labeling Plots

- `title()`, `xlabel()`, and `ylabel()`
  - Set the title and axis labels
- `legend()` and the keyword `label` in `plot()`
  - Set the plot legend

UNLV

Labeling Plots: Example

```
%matplotlib inline
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as
import matplotlib.pyplot as plt

# an array of 100 numbers evenly distributed
x = np.linspace(0, 10, 100)

# set the title of the plot
plt.title("Sine and Cosine Curves")
# set the label of x axis
plt.xlabel("x")
# set the label of y axis
plt.ylabel("sin(x) and cos(x)");

# set the plot legend for each line on the plot
plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.legend() # show the legend
```

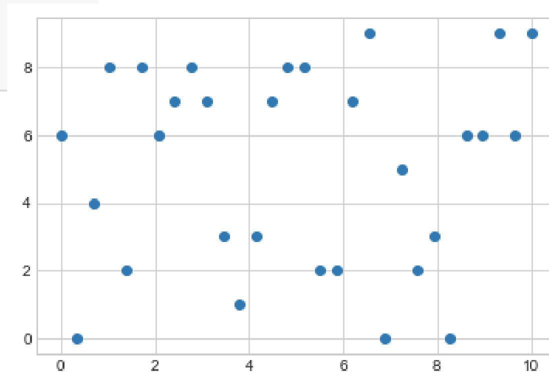


UNLV

Simple Scatter Plots

- scatter() function

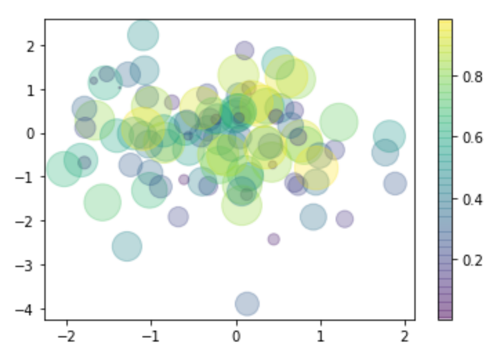
```
%matplotlib inline
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as plt
import matplotlib.pyplot as plt

# an array of 30 numbers evenly distributed between 0 and 100
x = np.linspace(0, 10, 30)
# an array of 30 random integers between 0 and 9
y = np.random.randint(0, 10, 30)

plt.scatter(x, y)

```



Scatter Plots with Color and Size

- c keyword: assign different colors to the dots
  - array or list of colors or color
- s keyword: assign different size to the dot

```
x = np.random.randn(100)
y = np.random.randn(100)
colors = np.random.rand(100)
sizes = 1000 * colors

# c is defined by the numbers in colors
# s is defined by the numbers in size
# cmap specifies the color map
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
            cmap='viridis')
plt.colorbar()  # show color scale
```



Colormaps

- Matplotlib has a number of built-in colormaps
  - They are accessible via at `matplotlib.colormaps`

```
import matplotlib.pyplot as plt


plt.colormaps()
```

```
['Accent',
 'Accent_r',
 'Blues',
 'Blues_r',
 'BrBG',
 'BrBG_r',
 'BuGn',
 'BuGn_r',
 'BuPu',
 'BuPu_r',
 'CMRmap',
 'CMRmap_r',
 'Dark2',
 'Dark2_r',
 'GnBu',
```


UNLV

- Lab
  - Height and Weight by Age Group
    This program reads the data from a csv file and then plot the relationships between height and weight

UNLV

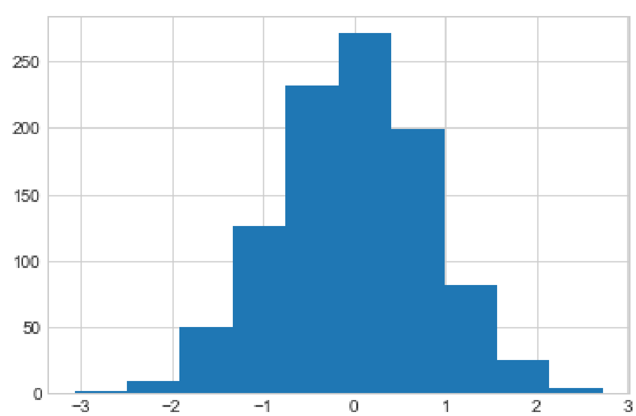
Histograms

- A simple histogram can be a great first step in understanding a dataset
- `hist()` function
  - `bins` keyword: specify the number of bins
  - `histtype` keyword:
    - '`bar`' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
    - '`barstacked`' is a bar-type histogram where multiple data are stacked on top of each other.
    - '`step`' generates a lineplot that is by default unfilled.
    - '`stepfilled`' generates a lineplot that is by default filled.
  - `alpha` keyword: set the opacity
  - `density` keyword: The area (or integral) under the histogram will sum to 1

UNLV

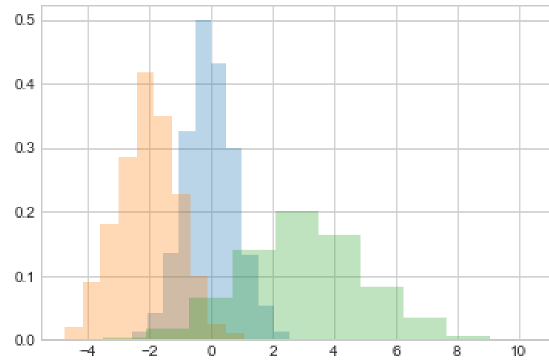
Histograms: Example (1)

```python
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as plt
import matplotlib.pyplot as plt
%matplotlib inline
# 1000 random numbers with mean=0, sd = 0.8
x1 = np.random.normal(0, 0.8, 1000)
plt.hist(x1, histtype='stepfilled', bins=10)
plt.show()
```



UNLV

Histograms: Example (2)



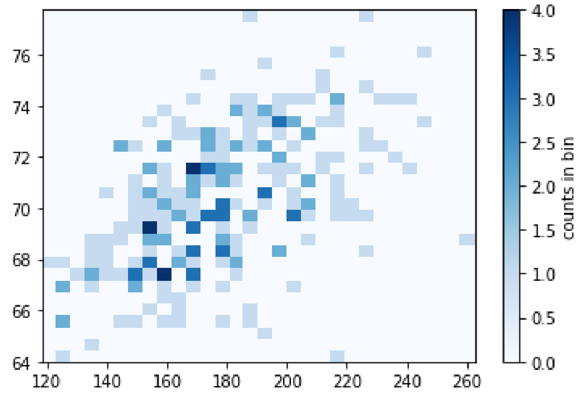

```
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as
import matplotlib.pyplot as plt
%matplotlib inline

# three random number arrays, with 1000 numbres each
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

# a shared set of keywords
kwargs = dict(histtype='stepfilled', alpha=0.3, density=True, bins=10)

# draw the three subsets of data, with the same set of keywords
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs)
```


Two-Dimensional Histograms

- We can also create histograms in two-dimensions by dividing points among two-dimensional bins
  - Create a heat map of the data
- `hist2d()` function
  - `bins` keyword: specify the number of bins for the two dimensions
  - `cmap` keyword: color theme

Two-Dimensional Histograms: Example

```
# draw the 2-d scatter plot
# 30 bins on each dimension. Color map as blue
plt.hist2d(weight, height, bins=30, cmap='Blues')
# show the colorar on the side with labels
plt.colorbar().set_label('counts in bin')
```



UNLV

Customizing Plot Style

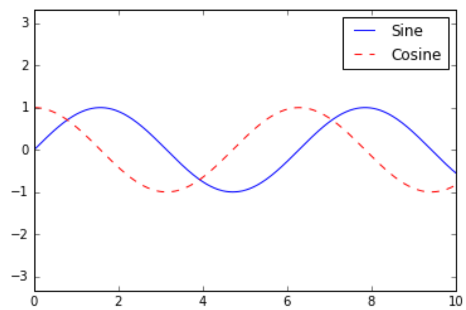
- `plt.style.use()`
  - Specify the style you would like to apply
  - You can use `plt.style.available` to see all the available styles
    - ['seaborn-ticks', 'ggplot', 'dark_background', 'bmh', 'seaborn-poster', 'seaborn-notebook', 'fast', 'seaborn', 'classic', 'Solarize_Light2', 'seaborn-dark', 'seaborn-pastel', 'seaborn-muted', '_classic_test', 'seaborn-paper', 'seaborn-colorblind', 'seaborn-bright', 'seaborn-talk', 'seaborn-dark-palette', 'tableau-colorblind10', 'seaborn-darkgrid', 'seaborn-whitegrid', 'fivethirtyeight', 'grayscale', 'seaborn-white', 'seaborn-deep']

UNLV

Customizing Plot Style: Example

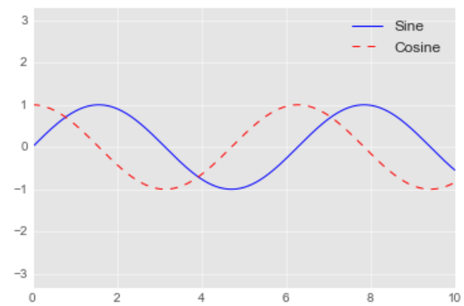
```python
plt.style.use('seaborn-muted')

x = np.linspace(0, 10, 1000)
# Get the fiure and axes from the plot
fig, ax = plt.subplots()
# draw the two lines with respective style and label
ax.plot(x, np.sin(x), '-b', label='Sine')
ax.plot(x, np.cos(x), '--r', label='Cosine')

# adjust plots with equal axis ratios
ax.axis('equal')
# show the legend
ax.legend()
```

```
<matplotlib.legend.Legend at 0x1763a39ba08>
```




```python
plt.style.use('ggplot')

x = np.linspace(0, 10, 1000)
# Get the fiure and axes from the plot
fig, ax = plt.subplots()
# draw the two lines with respective style and labels
ax.plot(x, np.sin(x), '-b', label='Sine')
ax.plot(x, np.cos(x), '--r', label='Cosine')

# adjust plots with equal axis ratios
ax.axis('equal')
# show the legend
ax.legend()
```

```
<matplotlib.legend.Legend at 0x276d1c53b48>
```



UNLV

Customizing Plot Legends

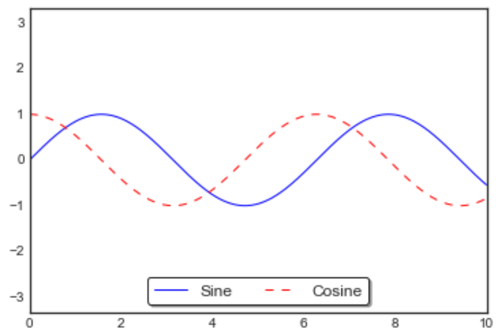
- `loc` keyword: specify the location
- `frameon` keyword: turn on or off the frame
- `ncol` keyword: specify the number of columns
- `fancybox` keyword: use a rounded box or not
- `shadow` keyword: add a shadow

UNLV

Customizing Plot Legends: Example

```python
plt.style.use('seaborn-white')

x = np.linspace(0, 10, 1000)
# Get the fiure and axes from the plot
fig, ax = plt.subplots()
# draw the two lines with respective style and labels
ax.plot(x, np.sin(x), '-b', label='Sine')
ax.plot(x, np.cos(x), '--r', label='Cosine')

# adjust plots with equal axis ratios
ax.axis('equal')
# show the legend to use fancybox, turn the frame on, add shadow
# make the location as lower center, two columns
ax.legend(fancybox=True, frameon=True, shadow=True, loc='lower center', ncol=2)
```

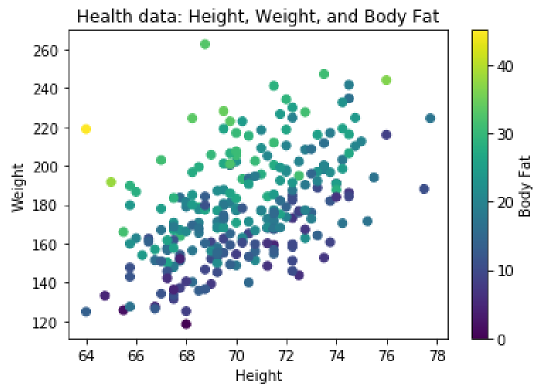
```
<matplotlib.legend.Legend at 0x276d2fd0f08>
```



- Lab
  - California Cities
    The program reads data from a csv file and plot the California cites. The size of the dots represents the area, and the color shows the population

Exercise

– Height Weight & BodyFat
Please use the data bodyData.csv to visualize the height, weight, and percentage body fat data. For example, create a figure as shown below



Visualization with Matplotlib (2)

Outline

- Pie Chart
- Multiple Subplots
- Three-Dimensional Plotting
- Visualization with Seaborn

Pie Chart

- `pie()` function to display the pie chart
  - Values and labels are lists
  - `autopct`: auto-labeling the percentage
    - Formatting string
  - explode: offsetting a slice


```python
import matplotlib.pyplot as plt

# Labels a list
categories = ['Utilities', 'Food', 'Entertainment', 'Clothing', 'Misc.']
# Values to be charted as a list
amounts = [312,658, 230, 498, 123]

# offsetting a slice; only "explode" the 3rd slice
offset = (0, 0, 0.1, 0, 0)

# draw the pie chart, showing percentage, offset the "Entertainment slice"
plt.pie(amounts, labels = categories, autopct ='%1.2f%%', explode=offset)
# add title to the chart
plt.title('Expenses')
plt.show()
```

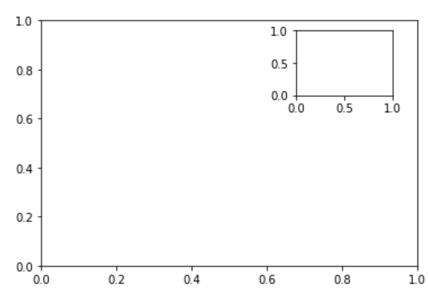
Multiple Subplots

- Subplots
  - Compare different views of data side by side
  - Groups of smaller axes that can exist together within a single figure
- Different ways to create subplots
  - Subplots by Hand
  - Grids of Subplots

Subplots by Hand (1)

- `plt.axes ()` function
  - by default this creates a standard axes object that fills the entire figure.
  - It also takes a four numbers in the figure coordinate system to represent [left, bottom, width, height]

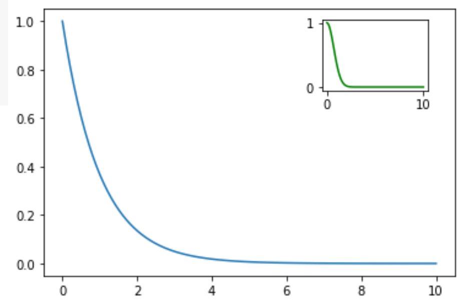
```python
# import numpy, set the alias as np
import numpy as np
# import the pyplot module, set the alias as plt
import matplotlib.pyplot as plt

# standard axes
ax1 = plt.axes()
# create an inset axes by setting the x and y position to 0.65
# and the x and y extents to 0.2
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```



Subplots by Hand (2)

- `plt.axes ()` function
  - After creating the new axes, plot the second line

```
# an array of 100 numbers evenly distributed between 0 and 100
a = np.linspace(0, 10, 100)
# an array of exponential values of -a
c = np.exp(-a)
# standard axes
ax1 = plt.axes()
# use a as x axis and b as y axis for a line plot
plt.plot(a, c)

# an array of exponential values of -a*a
d = np.exp(-a*a)
# create an inset axes by setting the x and y position to 0.65
# and the x and y extents to 0.2
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
# call plot() again to add another line
plt.plot(a, d, color = "green")
```



Simple Grids of Subplots (1)

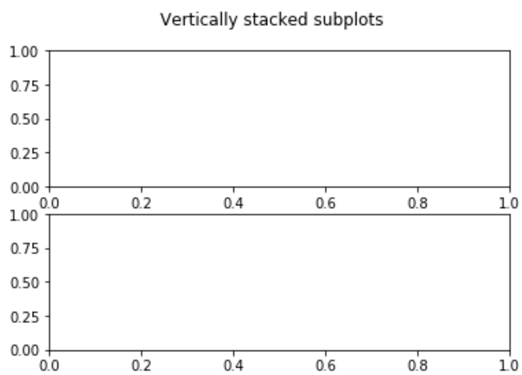
- `plt.subplots(rows, columns)`
  - Can be used to create subplots in rows and columns
  - The first optional arguments define the number of rows and columns of the subplot grid
  - The returned axes is a NumPy array containing the list of created Axes
    - When stacked in one direction, axes is a one-dimensional array
    - When a grid is created, axes is a two-dimensional array

Simple Grids of Subplots (2)

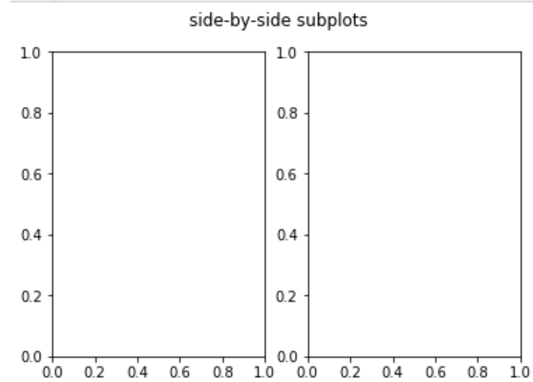
– The fig variable saves the figure object returned by subplots()

– The axs variable saves the axes array returned by subplots()

```python
# subplot in two rows.
# save the figure and axes to varaibles
fig, axs = plt.subplots(2)
# use the fig variable to set the title
fig.suptitle('Vertically stacked subplots')
```



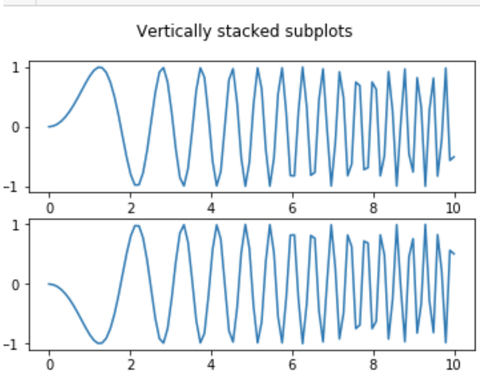

```python
# subplot in two rows.
# save the figure and axes to varaibles
fig, axs = plt.subplots(1,2)
# use the fig variable to set the title
fig.suptitle('side-by-side subplots')
```



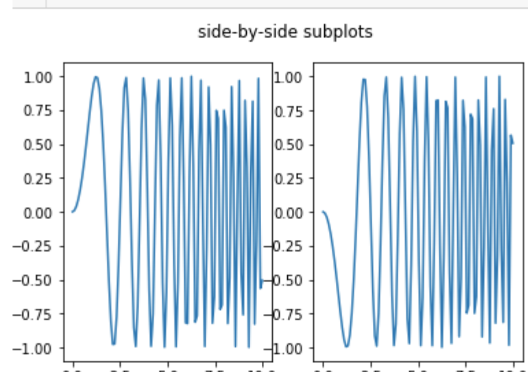
Simple Grids of Subplots (3)

– Then, use index to specify the axes to plot

```python
x = np.linspace(0, 10, 100)
# sine function of x squared
y = np.sin(x ** 2)

# subplot in two rows.
# save the figure and axes to varaibles
fig, axs = plt.subplots(2)
# use the fig variable to set the title
fig.suptitle('Vertically stacked subplots')
# Plot the figure at the first row
axs[0].plot(x, y)
# Plot the figure at the second row
axs[1].plot(x, -y)
```



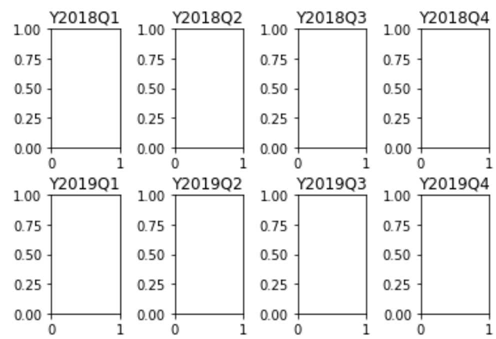

```python
x = np.linspace(0, 10, 100)
# sine function of x squared
y = np.sin(x ** 2)

# subplot in two rows.
# save the figure and axes to varaibles
fig, axs = plt.subplots(1,2)
# use the fig variable to set the title
fig.suptitle('side-by-side subplots')
# Plot the figure at the first column
axs[0].plot(x, y)
# Plot the figure at the second column
axs[1].plot(x, -y)
```



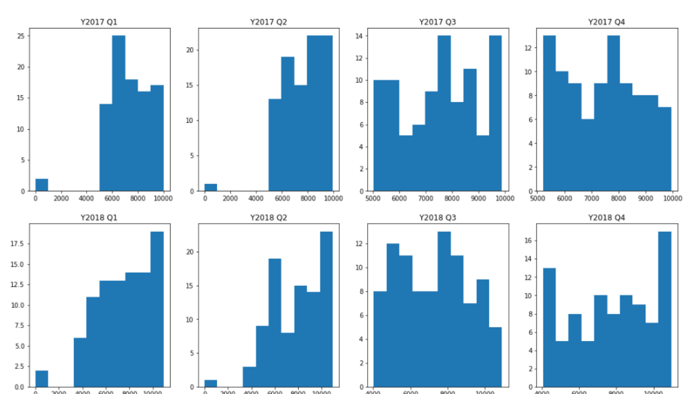
Simple Grids of Subplots (4)

- When more than one column or rows are specified, use the index of two-dimensional array to specify the subplot

```python
year = ['2018','2019']
quarter = ['Q1', 'Q2', 'Q3','Q4']

fig, axs = plt.subplots(len(year), len(quarter))
fig.subplots_adjust(hspace=0.4, wspace=0.8)

for y in range(len(year)):
    for q in range(len(quarter)):
        axs[y, q].set_title('Y'+year[y]+quarter[q])
```



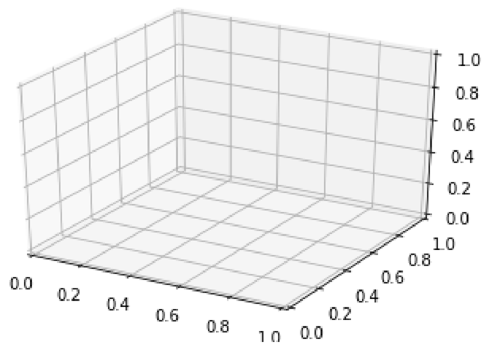
- Lab
  - Sales History Comparison
    The program reads a file containing the quarterly sales data and plot histogram for comparison.



Three-Dimensional Plotting (1)

- It is enabled by the `mplot3d` toolkit

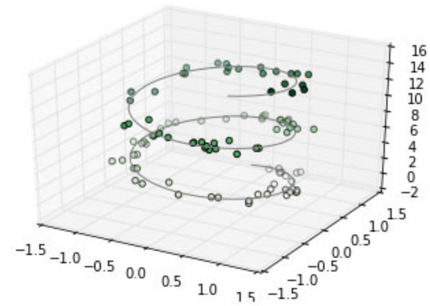
```
import matplotlib.pyplot as plt

# to enable the 3d plotting
from mpl_toolkits import mplot3d

%matplotlib inline

ax = plt.axes(projection='3d')

```



Three-Dimensional Plotting (2)

- `ax.plot3D()` and `ax.scatter3D()`
  - The most basic three-dimensional plot is a line or collection of scatter plot created from sets of (x, y, z) triples
  - The call signature for these is nearly identical to that of their two-dimensional counterparts

Three-Dimensional Plotting: Example




```python
ax = plt.axes(projection='3d')

# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
# 3d line plot
ax.plot3D(xline, yline, zline, 'gray')

# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
# 3d scatter plot
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens')
```


- Lab
  - Three-Dimensional Body Data
    This program reads the data from a csv file and then plot the data points

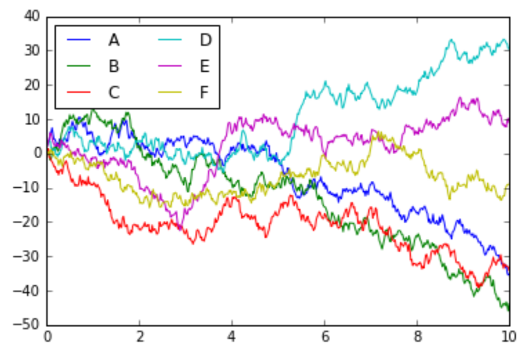
- Exercise
  - Revise Three-Dimensional Body Data program
    Please create three data sets: people less than or equal 30 years old, people between 30 and 60, and people over 60. Plot the three datasets on the 3D scatter plot.

Visualization with Seaborn

- Provides an API on top of Matplotlib that offers choices for plot style and color defaults, defines simple high-level functions for common statistical plot types
- Integrates with the functionality provided by Pandas DataFrames
- Build in some dataset that can be used for learning purposes
  - Get all the available dataset by using `sns.get_dataset_names()`

Seaborn Style

```python
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# Create some data
rng = np.random.RandomState(0)
x = np.linspace(0, 10, 500)
y = np.cumsum(rng.randn(500, 6), 0)

# The old matplotlib style
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left')
```

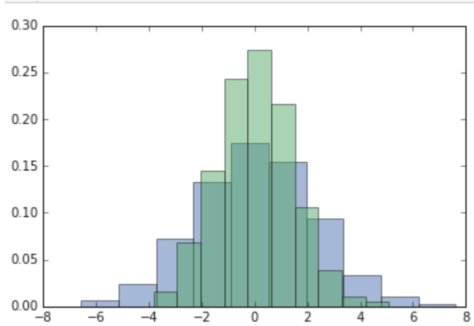



```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline

# Create some data
rng = np.random.RandomState(0)
x = np.linspace(0, 10, 500)
y = np.cumsum(rng.randn(500, 6), 0)

# use the seaborn style
sns.set()

plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left')
```

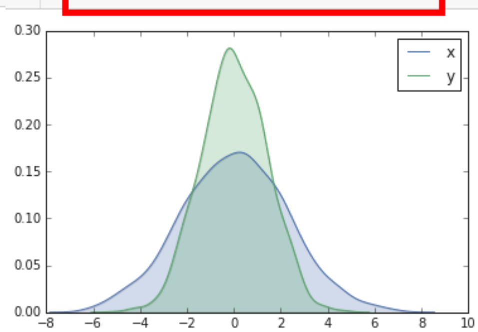
Distribution Plot

- `sns.kdeplot()`
  - Instead of histogram, get a smooth estimate of the distribution using a kernel density estimation

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]], size=2000)
data = pd.DataFrame(data, columns=['x', 'y'])

for col in 'xy':
    plt.hist(data[col], density=True, alpha=0.5)
```

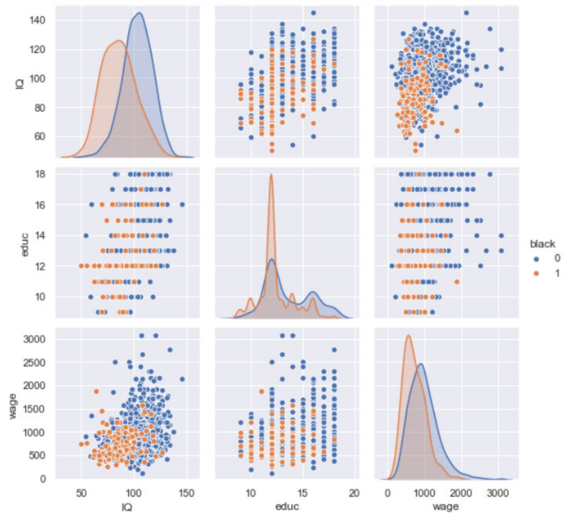


```python
for col in 'xy':
    sns.kdeplot(data[col], shade=True)
```



Exploring Data (1)

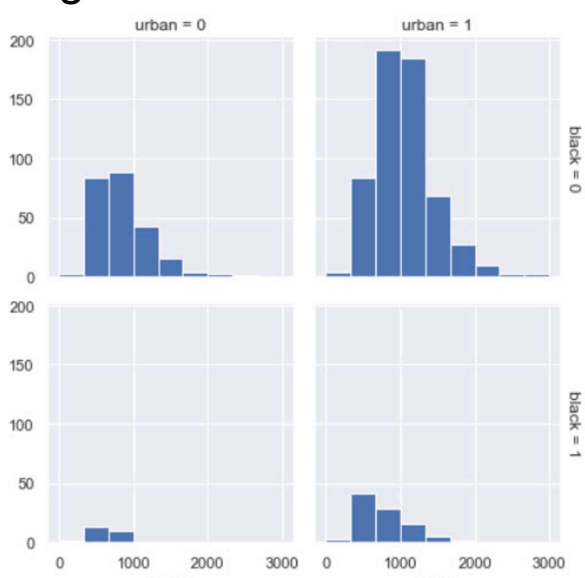
- Pair Plots: `sns.pairplot()`
  - Useful for exploring correlations between multidimensional data
  - plot all pairs of values against each other



UNLV

Exploring Data (2)

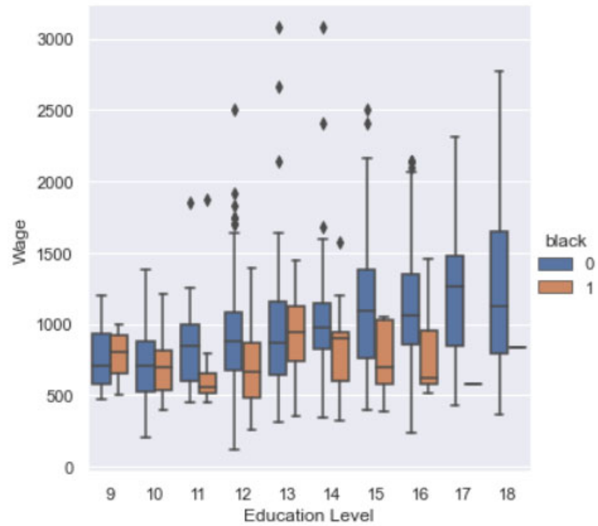
- Faceted histograms: `sns.FacetGrid()`
  - histograms of subsets



UNLV

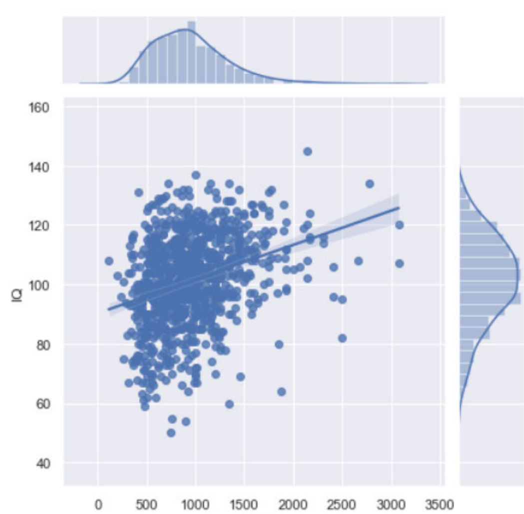
Exploring Data (3)

- Factor plots: `sns.catplot()`
  - view the distribution of a parameter within bins defined by any other parameter



Exploring Data (4)

- Joint distributions: `sns.jointplot()`
  - show the joint distribution between different datasets, along with the associated marginal distributions



Lab

- Tip Distribution
  The program is a test of many useful functions in Seaborn, including pair plot, faceted histograms, factor plots, and joint distribution

Exercise

- Wage in 1980
  Write a program that read the data "wage.csv" and create the following diagrams:
  - pair plot for four variables: IQ, education, black, and wage.
  - faceted histograms for wage, categorized by black and urban (10 bins between 0 and 3000)
  - factor plot: for each education level, compare the wage by black or not
  - joint distribution on (1) IQ and wage and (2) experience and wage
- Data source: Introductory Econometrics: A Modern Approach, 6e by Jeffrey M. Wooldridge.