CSE 572: Lab 1

This lab contains two modules:

1. Module 1: Introduction to Python
2. Module 2: Introduction to Numpy and Pandas

At the end there is a practice exercise in which you will use some of the operations from Modules 1 and 2.

To execute and make changes to this notebook, click File > Save a copy to save your own version in your Google Drive or Github. Read the step-by-step instructions below carefully. To execute the code, click on each cell below and press the SHIFT-ENTER keys simultaneously or by clicking the Play button.

When you finish executing all code/exercises, save your notebook then download a copy (.ipynb file). Submit 1) a link to your Colab notebook, 2) the .ipynb file, and **3) a pdf of the executed notebook** on Canvas.

To generate a pdf of the notebook, click File > Print > Save as PDF.

Module 1: Introduction to Python

Python is a high-level programming language with extensive libraries available to perform various data analysis tasks. The following tutorial contains examples of using various data types, functions, and library modules available in the standard Python library.

We begin with some basic information about Python:

1. Python is an interpreted language, unlike other high-level programming languages such as C or C++. You only need to submit your Python program to an interpreter for execution, without having to explicitly compile and link the code first.
2. Python is a dynamically typed language, which means variable names are bound to their respective types during execution time. You do not have to explicitly declare the type of a variable before using it in the code unlike Java, C++, and other statically-typed languages.
3. Instead of using braces '{' and '}', Python uses whitespace indentation to group together related statements in loops or other control-flow statements.
4. Python uses the hash character ('#') to precede single-line comments. Triple-quoted strings (''') are commonly used to denote multi-line comments (even though it is not part of the standard Python language) or docstring of functions.
5. Python uses pass by reference (instead of pass by value) when assigning a variable to another (e.g., a = b) or when passing an object as input argument to a function. Thus, any modification to the assigned variable or to the input argument within the function will affect the original object.
6. Python uses `None` to denote a null object (e.g., a = None). You do not have to terminate each statement with a terminating character (such as a semicolon) unlike other languages.
7. You may access the variables or functions defined in another Python program file using the `import` command. This is analogous to the `import` command in Java or the `#include` command in C or C++.

1.1 Elementary Data Types

The standard Python library provides support for various elementary data types, including including integers, booleans, floating points, and strings. A summary of the data types is shown in the table below.

	Data Type	Example
Number	Integer	x = 4
	Long integer	x = 15L
	Floating point	x = 3.142
	Boolean	x = True
Text	Character	x = 'c'
	String	x = "this" or x = 'this'

```
x = 4              # integer
print(x, type(x))
```

```
y = True          # boolean (True, False)
print(y, type(y))

z = 3.7           # floating point
print(z, type(z))

s = "This is a string"    # string
print(s, type(s))
```

```
4 <class 'int'>
True <class 'bool'>
3.7 <class 'float'>
This is a string <class 'str'>
```

The following are some of the arithmetic operations available for manipulating integers and floating point numbers

```
x = 4           # integer
x1 = x + 4      # addition
x2 = x * 3      # multiplication
x += 2          # equivalent to x = x + 2
x3 = x
x *= 3          # equivalent to x = x * 3
x4 = x
x5 = x % 4      # modulo (remainder) operator

z = 3.7         # floating point number
z1 = z - 2      # subtraction
z2 = z / 3      # division
z3 = z // 3     # integer division
z4 = z ** 2     # square of z
z5 = z4 ** 0.5  # square root
z6 = pow(z,2)   # equivalent to square of z
z7 = round(z)   # rounding z to its nearest integer
z8 = int(z)     # type casting float to int

print(x,x1,x2,x3,x4,x5)
print(z,z1,z2,z3,z4)
print(z5,z6,z7,z8)
```

```
18 8 12 6 18 2
3.7 1.7000000000000002 1.2333333333333334 1.0 13.690000000000001
3.7 13.690000000000001 4 3
```

The following are some of the functions provided by the math module for integers and floating point numbers

```
import math

x = 4
print(math.sqrt(x))      # sqrt(4) = 2
print(math.pow(x,2))     # 4**2 = 16
print(math.exp(x))       # exp(4) = 54.6
print(math.log(x,2))     # log based 2  (default is natural logarithm)
print(math.fabs(-4))     # absolute value
print(math.factorial(x)) # 4! = 4 x 3 x 2 x 1 = 24

z = 0.2
print(math.ceil(z))      # ceiling function
print(math.floor(z))     # floor function
print(math.trunc(z))     # truncate function

z = 3*math.pi            # math.pi = 3.141592653589793
print(math.sin(z))       # sine function
print(math.tanh(z))      # arctan function

x = math.nan             # not a number
print(math.isnan(x))

x = math.inf             # infinity
print(math.isinf(x))
```

```
2.0
16.0
54.598150033144236
2.0
4.0
24
```

```
1
0
0
3.6739403974420594e-16
0.9999999869751758
True
True
```

The following are some of the logical operations available for booleans

```
y1 = True
y2 = False

print(y1 and y2)       # logical AND
print(y1 or y2)        # logical OR
print(y1 and not y2)   # logical NOT
```

```
False
True
True
```

The following are some of the operations and functions for manipulating strings

```
s1 = "This"

print(s1[1:])                    # print last three characters
print(len(s1))                               # get the string length
print("Length of string is " + str(len(s1))) # type casting int to str
print(s1.upper())                            # convert to upper case
print(s1.lower())                            # convert to lower case

s2 = "This is a string"
words = s2.split(' ')             # split the string into words
print(words[0])
print(s2.replace('a','another'))  # replace "a" with "another"
print(s2.replace('is','at'))      # replace "is" with "at"
print(s2.find("a"))               # find the position of "a" in s2
print(s1 in s2)                   # check if s1 is a substring of s2

print(s1 == 'This')               # equality comparison
print(s1 < 'That')                # inequality comparison
print(s2 + " too")                # string concatenation
print((s1 + " ")* 3)              # replicate the string 3 times
```

```
his
4
Length of string is 4
THIS
this
This
This is another string
That at a string
8
True
True
False
This is a string too
This This This
```

1.2 Compound Data Types

The following examples show how to create and manipulate a list object

```
intlist = [1, 3, 5, 7, 9]
print(type(intlist))
print(intlist)
intlist2 = list(range(0,10,2))   # range[startvalue, endvalue, stepsize]
print(intlist2)

print(intlist[2])                # get the third element of the list
print(intlist[:2])               # get the first two elements
print(intlist[2:])               # get the last three elements of the list
print(intlist[-2:])              # get the last two elements of the list
```

```
print(len(intlist))             # get the number of elements in the list
print(sum(intlist))             # sums up elements of the list

intlist.append(11)              # insert 11 to end of the list
print(intlist)
print(intlist.pop())            # remove last element of the list
print(intlist)
print(intlist + [11,13,15])     # concatenate two lists
print(intlist * 3)              # replicate the list
intlist.insert(2,4)             # insert item 4 at index 2
print(intlist)
intlist.sort(reverse=True)      # sort elements in descending order
print(intlist)
```

```
<class 'list'>
[1, 3, 5, 7, 9]
[0, 2, 4, 6, 8]
5
[1, 3]
[5, 7, 9]
[7, 9]
5
25
[1, 3, 5, 7, 9, 11]
11
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9, 11, 13, 15]
[1, 3, 5, 7, 9, 1, 3, 5, 7, 9, 1, 3, 5, 7, 9]
[1, 3, 4, 5, 7, 9]
[9, 7, 5, 4, 3, 1]
```

```
mylist = ['this', 'is', 'a', 'list']
print(mylist)
print(type(mylist))

print("list" in mylist)         # check whether "list" is in mylist
print(mylist[2])                # show the 3rd element of the list
print(mylist[:2])               # show the first two elements of the list
print(mylist[2:])               # show the last two elements of the list
mylist.append("too")            # insert element to end of the list

separator = " "
print(separator.join(mylist))   # merge all elements of the list into a string

mylist.remove("is")             # remove element from list
print(mylist)
```

```
['this', 'is', 'a', 'list']
<class 'list'>
True
a
['this', 'is']
['a', 'list']
this is a list too
['this', 'a', 'list', 'too']
```

The following examples show how to create and manipulate a dictionary object

```
abbrev = {}
abbrev['MI'] = "Michigan"
abbrev['MN'] = "Minnesota"
abbrev['TX'] = "Texas"
abbrev['CA'] = "California"

print(abbrev)
print(abbrev.keys())
print(type(abbrev.keys()))          # get the keys of the dictionary
print(abbrev.values())          # get the values of the dictionary
print(len(abbrev))              # get number of key-value pairs

print(abbrev.get('MI'))
print("FL" in abbrev)
print("CA" in abbrev)
```

```
{'MI': 'Michigan', 'MN': 'Minnesota', 'TX': 'Texas', 'CA': 'California'}
dict_keys(['MI', 'MN', 'TX', 'CA'])
<class 'dict_keys'>
dict_values(['Michigan', 'Minnesota', 'Texas', 'California'])
```

```
4
Michigan
False
True
```

```
keys = ['apples', 'oranges', 'bananas', 'cherries']
values = [3, 4, 2, 10]
fruits = dict(zip(keys, values))
print(fruits)
print(sorted(fruits))     # sort keys of dictionary

from operator import itemgetter
print(sorted(fruits.items(), key=itemgetter(0)))    # sort by key of dictionary
print(sorted(fruits.items(), key=itemgetter(1)))    # sort by value of dictionary
```

```
{'apples': 3, 'oranges': 4, 'bananas': 2, 'cherries': 10}
['apples', 'bananas', 'cherries', 'oranges']
[('apples', 3), ('bananas', 2), ('cherries', 10), ('oranges', 4)]
[('bananas', 2), ('apples', 3), ('oranges', 4), ('cherries', 10)]
```

The following examples show how to create and manipulate a tuple object. Unlike a list, a tuple object is immutable, i.e., they cannot be modified after creation.

```
MItuple = ('MI', 'Michigan', 'Lansing')
CAtuple = ('CA', 'California', 'Sacramento')
TXtuple = ('TX', 'Texas', 'Austin')

print(MItuple)
print(MItuple[1:])

states = [MItuple, CAtuple, TXtuple]    # this will create a list of tuples
print(states)
print(states[2]) # print the third tuple in the list
print(states[2][:]) # print all the values in the third tuple
print(states[2][1:]) # print the last two values in the third tuple

states.sort(key=lambda state: state[2])  # sort the states by their capital cities (last value in list)
print(states)
```

```
('MI', 'Michigan', 'Lansing')
('Michigan', 'Lansing')
[('MI', 'Michigan', 'Lansing'), ('CA', 'California', 'Sacramento'), ('TX', 'Texas', 'Austin')]
('TX', 'Texas', 'Austin')
('TX', 'Texas', 'Austin')
('Texas', 'Austin')
[('TX', 'Texas', 'Austin'), ('MI', 'Michigan', 'Lansing'), ('CA', 'California', 'Sacramento')]
```

1.3 Control Flow Statements

Similar to other programming languages, the control flow statements in Python include if, for, and while statements. Examples on how to use these statements are shown below.

```
# using if-else statement

x = 10

if x % 2 == 0:
    print("x =", x, "is even")
else:
    print("x =", x, "is odd")

if x > 0:
    print("x =", x, "is positive")
elif x < 0:
    print("x =", x, "is negative")
else:
    print("x =", x, "is neither positive nor negative")
```

```
x = 10 is even
x = 10 is positive
```

```
# using for loop with a list

mylist = ['this', 'is', 'a', 'list']
for word in mylist:
    print(word.replace("is", "at"))

mylist2 = [len(word) for word in mylist]   # number of characters in each word
print(mylist2)
```

```
that
at
a
latt
[4, 2, 1, 4]
```

```
# using for loop with list of tuples

states = [('MI', 'Michigan', 'Lansing'),('CA', 'California', 'Sacramento'),
          ('TX', 'Texas', 'Austin')]

sorted_capitals = [state[2] for state in states]
sorted_capitals.sort()
print(sorted_capitals)
```

```
['Austin', 'Lansing', 'Sacramento']
```

```
# using for loop with dictionary

fruits = {'apples': 3, 'oranges': 4, 'bananas': 2, 'cherries': 10}
fruitnames = [k for (k,v) in fruits.items()]
print(fruitnames)
```

```
['apples', 'oranges', 'bananas', 'cherries']
```

```
# using while loop

mylist = list(range(-10,10))
print(mylist)

i = 0
while (mylist[i] < 0):
    i = i + 1

print("First non-negative number:", mylist[i])
```

```
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
First non-negative number: 0
```

1.4 User-Defined Functions

You can create your own functions in Python, which can be named or unnamed. Unnamed functions are defined using the lambda keyword as shown in the previous example for sorting a list of tuples.

```
myfunc = lambda x: 3*x**2 - 2*x + 3      # example of an unnamed quadratic function

print(myfunc(2))
```

```
11
```

```
import math

# The following function will discard missing values from a list
def discard(inlist, sortFlag=False):    # default value for sortFlag is False
    outlist = []
    for item in inlist:
        if not math.isnan(item):
            outlist.append(item)

    if sortFlag:
        outlist.sort()
    return outlist
```

```
mylist = [12, math.nan, 23, -11, 45, math.nan, 71]

print(discard(mylist,True))
```

```
[-11, 12, 23, 45, 71]
```

1.5 File I/O

You can read and write data from a list or other objects to a file.

```
states = [('MI', 'Michigan', 'Lansing'),('CA', 'California', 'Sacramento'),
          ('TX', 'Texas', 'Austin'), ('MN', 'Minnesota', 'St Paul')]

with open('states.txt', 'w') as f:
    f.write('\n'.join('%s,%s,%s' % state for state in states))

with open('states.txt', 'r') as f:
    for line in f:
        fields = line.split(sep=',')    # split each line into its respective fields
        print('State=',fields[1],'(',fields[0],')','Capital:', fields[2])
```

```
State= Michigan ( MI ) Capital: Lansing

State= California ( CA ) Capital: Sacramento

State= Texas ( TX ) Capital: Austin

State= Minnesota ( MN ) Capital: St Paul
```

Module 2: Introduction to Numpy and Pandas

The following tutorial contains examples of using the numpy and pandas library modules.

2.1 Introduction to Numpy

Numpy, which stands for numerical Python, is a Python library package to support numerical computations. The basic data structure in numpy is a multi-dimensional array object called ndarray. Numpy provides a suite of functions that can efficiently manipulate elements of the ndarray.

2.1.1 Creating ndarray

An ndarray can be created from a list or a tuple object as shown in the examples below. It is possible to create a 1-dimensional or multi-dimensional array from the list objects as well as tuples.

```
import numpy as np

oneDim = np.array([1,2,3,4,5])   # a 1-dimensional array (vector)
print(oneDim)
print("#Dimensions =", oneDim.ndim)
print("Dimension =", oneDim.shape)
print("Size =", oneDim.size)
print("Array type =", oneDim.dtype, '\n')

twoDim = np.array([[1,2],[3,4],[5,6],[7,8]])  # a two-dimensional array (matrix)
print(twoDim)
print("#Dimensions =", twoDim.ndim)
print("Dimension =", twoDim.shape)
print("Size =", twoDim.size)
print("Array type =", twoDim.dtype, '\n')

arrFromTuple = np.array([(1,'a',3.0),(2,'b',3.5)])  # create ndarray from tuple
print(arrFromTuple)
print("#Dimensions =", arrFromTuple.ndim)
print("Dimension =", arrFromTuple.shape)
print("Size =", arrFromTuple.size)
```

```
[1 2 3 4 5]
#Dimensions = 1
Dimension = (5,)
Size = 5
Array type = int64
```

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
#Dimensions = 2
Dimension = (4, 2)
Size = 8
Array type = int64

[['1' 'a' '3.0']
 ['2' 'b' '3.5']]
#Dimensions = 2
Dimension = (2, 3)
Size = 6
```

There are also built-in functions available in numpy to create the ndarrays.

```
print('Array of random numbers from a uniform distribution')
print(np.random.rand(5))      # random numbers from a uniform distribution between [0,1]

print('\nArray of random numbers from a normal distribution')
print(np.random.randn(5))     # random numbers from a normal distribution

print('\nArray of integers between -10 and 10, with step size of 2')
print(np.arange(-10,10,2))    # similar to range, but returns ndarray instead of list

print('\n2-dimensional array of integers from 0 to 11')
print(np.arange(12).reshape(3,4))  # reshape to a matrix

print('\nArray of values between 0 and 1, split into 10 equally spaced values')
print(np.linspace(0,1,10))    # split interval [0,1] into 10 equally separated values

print('\nArray of values from 10^-3 to 10^3')
print(np.logspace(-3,3,7))    # create ndarray with values from 10^-3 to 10^3
```

```
Array of random numbers from a uniform distribution
[0.27367601 0.52689961 0.47604609 0.68836304 0.06507451]

Array of random numbers from a normal distribution
[ 0.40668244 -0.13211243 -0.2559828   0.24271227  0.79552091]

Array of integers between -10 and 10, with step size of 2
[-10  -8  -6  -4  -2   0   2   4   6   8]

2-dimensional array of integers from 0 to 11
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

Array of values between 0 and 1, split into 10 equally spaced values
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]

Array of values from 10^-3 to 10^3
[1.e-03 1.e-02 1.e-01 1.e+00 1.e+01 1.e+02 1.e+03]
```

```
print('A 2 x 3 matrix of zeros')
print(np.zeros((2,3)))        # a matrix of zeros

print('\nA 3 x 2 matrix of ones')
print(np.ones((3,2)))         # a matrix of ones

print('\nA 3 x 3 identity matrix')
print(np.eye(3))              # a 3 x 3 identity matrix
```

```
A 2 x 3 matrix of zeros
[[0. 0. 0.]
 [0. 0. 0.]]

A 3 x 2 matrix of ones
[[1. 1.]
 [1. 1.]
 [1. 1.]]

A 3 x 3 identity matrix
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

2.1.2 Element-wise Operations

You can apply standard operators such as addition and multiplication on each element of the ndarray.

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

print('x =', x)
print('x + 1 =', x + 1)      # addition
print('x - 1 =', x - 1)      # subtraction
print('x * 2 =', x * 2)      # multiplication
print('x // 2 =', x // 2)     # integer division
print('x ** 2 =', x ** 2)     # square
print('x % 2 =', x % 2)      # modulo
print('1 / x =', 1 / x)      # division
```

```
x = [1 2 3 4 5]
x + 1 = [2 3 4 5 6]
x - 1 = [0 1 2 3 4]
x * 2 = [ 2  4  6  8 10]
x // 2 = [0 1 1 2 2]
x ** 2 = [ 1  4  9 16 25]
x % 2 = [1 0 1 0 1]
1 / x = [1.         0.5        0.33333333 0.25       0.2       ]
```

```
x = np.array([2,4,6,8,10])
y = np.array([1,2,3,4,5])

print('x =', x)
print('y =', y)
print('x + y =', x + y)      # element-wise addition
print('x - y =', x - y)      # element-wise subtraction
print('x * y =', x * y)      # element-wise multiplication
print('x / y =', x / y)      # element-wise division
print('x // y =', x // y)    # element-wise integer division
print('x ** y =', x ** y)    # element-wise exponentiation
```

```
x = [ 2  4  6  8 10]
y = [1 2 3 4 5]
x + y = [ 3  6  9 12 15]
x - y = [1 2 3 4 5]
x * y = [ 2  8 18 32 50]
x / y = [2. 2. 2. 2. 2.]
x // y = [2 2 2 2 2]
x ** y = [     2     16    216   4096 100000]
```

2.1.3 Indexing and Slicing

There are various ways to select a subset of elements within a numpy array. Assigning a numpy array (or a subset of its elements) to another variable will simply pass a reference to the array instead of copying its values. To make a copy of an ndarray, you need to explicitly call the .copy() function.

```
x = np.arange(-5,5)
print('Before: x =', x)

y = x[3:5]     # y is a slice, i.e., pointer to a subarray in x
print('        y =', y)
y[:] = 1000    # modifying the value of y will change x
print('After : y =', y)
print('        x =', x, '\n')

z = x[3:5].copy()   # makes a copy of the subarray
print('Before: x =', x)
print('        z =', z)
z[:] = 500          # modifying the value of z will not affect x
print('After : z =', z)
print('        x =', x)
```

```
Before: x = [-5 -4 -3 -2 -1  0  1  2  3  4]
        y = [-2 -1]
After : y = [1000 1000]
        x = [  -5   -4   -3 1000 1000    0    1    2    3    4]

Before: x = [  -5   -4   -3 1000 1000    0    1    2    3    4]
```

```
          z = [1000 1000]
After : z = [500 500]
        x = [  -5   -4   -3 1000 1000    0    1    2    3    4]
```

There are many ways to access elements of an ndarray. The following example illustrates the difference between indexing elements of a list and elements of ndarray.

```
my2dlist = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]  # a 2-dim list
print('my2dlist =', my2dlist)
print('my2dlist[2] =', my2dlist[2])            # access the third sublist
print('my2dlist[:][2] =', my2dlist[:][2])      # can't access third element of each sublist
# print('my2dlist[:,2] =', my2dlist[:,2])      # invalid way to access sublist, will cause syntax error

my2darr = np.array(my2dlist)
print('\nmy2darr =\n', my2darr)

print('my2darr[2][:] =', my2darr[2][:])      # access the third row
print('my2darr[2,:] =', my2darr[2,:])        # access the third row
print('my2darr[:][2] =', my2darr[:][2])      # access the third row (similar to 2d list)
print('my2darr[:,2] =', my2darr[:,2])        # access the third column
print('my2darr[:2,2:] =\n', my2darr[:2,2:])     # access the first two rows & last two columns
```

```
my2dlist = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
my2dlist[2] = [9, 10, 11, 12]
my2dlist[:][2] = [9, 10, 11, 12]

my2darr =
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
my2darr[2][:] = [ 9 10 11 12]
my2darr[2,:] = [ 9 10 11 12]
my2darr[:][2] = [ 9 10 11 12]
my2darr[:,2] = [ 3  7 11]
my2darr[:2,2:] =
 [[3 4]
 [7 8]]
```

Numpy arrays also support boolean indexing.

```
my2darr = np.arange(1,13,1).reshape(3,4)
print('my2darr =\n', my2darr)

divBy3 = my2darr[my2darr % 3 == 0]
print('\nmy2darr[my2darr % 3 == 0] =', divBy3)            # returns all the elements divisible by 3 in an ndarray

divBy3LastRow = my2darr[2:, my2darr[2,:] % 3 == 0]
print('my2darr[2:, my2darr[2,:] % 3 == 0] =', divBy3LastRow)    # returns elements in the last row divisible by 3
```

```
my2darr =
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

my2darr[my2darr % 3 == 0] = [ 3  6  9 12]
my2darr[2:, my2darr[2,:] % 3 == 0] = [[ 9 12]]
```

More indexing examples.

```
my2darr = np.arange(1,13,1).reshape(4,3)
print('my2darr =\n', my2darr)

indices = [2,1,0,3]    # selected row indices
print('indices =', indices, '\n')
print('my2darr[indices,:] =\n', my2darr[indices,:])  # this will shuffle the rows of my2darr

rowIndex = [0,0,1,2,3]     # row index into my2darr
print('\nrowIndex =', rowIndex)
columnIndex = [0,2,0,1,2]  # column index into my2darr
print('columnIndex =', columnIndex, '\n')
print('my2darr[rowIndex,columnIndex] =', my2darr[rowIndex,columnIndex])
```

```
my2darr =
 [[ 1  2  3]
```

```
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
indices = [2, 1, 0, 3]

my2darr[indices,:] =
 [[ 7  8  9]
 [ 4  5  6]
 [ 1  2  3]
 [10 11 12]]

rowIndex = [0, 0, 1, 2, 3]
columnIndex = [0, 2, 0, 1, 2]

my2darr[rowIndex,columnIndex] = [ 1  3  4  8 12]
```

2.1.4 Numpy Arithmetic and Statistical Functions

Numpy provides many built-in mathematical functions available for manipulating elements of an ndarray.

```
y = np.array([-1.4, 0.4, -3.2, 2.5, 3.4])
print('y =', y, '\n')

print('np.abs(y) =', np.abs(y))                # convert to absolute values
print('np.sqrt(abs(y)) =', np.sqrt(abs(y)))    # apply square root to each element
print('np.sign(y) =', np.sign(y))              # get the sign of each element
print('np.exp(y) =', np.exp(y))                # apply exponentiation
print('np.sort(y) =', np.sort(y))              # sort array
```

```
y = [-1.4  0.4 -3.2  2.5  3.4]

np.abs(y) = [1.4 0.4 3.2 2.5 3.4]
np.sqrt(abs(y)) = [1.18321596 0.63245553 1.78885438 1.58113883 1.84390889]
np.sign(y) = [-1.  1. -1.  1.  1.]
np.exp(y) = [ 0.24659696  1.4918247   0.0407622  12.18249396 29.96410005]
np.sort(y) = [-3.2 -1.4  0.4  2.5  3.4]
```

```
x = np.arange(-2,3)
y = np.random.randn(5)
print('x =', x)
print('y =', y, '\n')

print('np.add(x,y) =', np.add(x,y))                # element-wise addition       x + y
print('np.subtract(x,y) =', np.subtract(x,y))      # element-wise subtraction    x - y
print('np.multiply(x,y) =', np.multiply(x,y))      # element-wise multiplication x * y
print('np.divide(x,y) =', np.divide(x,y))          # element-wise division       x / y
print('np.maximum(x,y) =', np.maximum(x,y))        # element-wise maximum        max(x,y)
```

```
x = [-2 -1  0  1  2]
y = [-0.14836267  1.2563624  -0.84783783 -0.52857267  0.65294457]

np.add(x,y) = [-2.14836267  0.2563624  -0.84783783  0.47142733  2.65294457]
np.subtract(x,y) = [-1.85163733 -2.2563624   0.84783783  1.52857267  1.34705543]
np.multiply(x,y) = [ 0.29672534 -1.2563624  -0.         -0.52857267  1.30588914]
np.divide(x,y) = [13.48048004 -0.79594868 -0.         -1.89188744  3.06304714]
np.maximum(x,y) = [-0.14836267  1.2563624   0.          1.          2.        ]
```

```
y = np.array([-3.2, -1.4, 0.4, 2.5, 3.4])
print('y =', y, '\n')

print("Min =", np.min(y))             # min
print("Max =", np.max(y))             # max
print("Average =", np.mean(y))        # mean/average
print("Std deviation =", np.std(y))   # standard deviation
print("Sum =", np.sum(y))             # sum
```

```
y = [-3.2 -1.4  0.4  2.5  3.4]

Min = -3.2
Max = 3.4
Average = 0.34000000000000014
Std deviation = 2.432776191925595
Sum = 1.7000000000000006
```

2.1.5 Numpy linear algebra

Numpy provides many functions to support linear algebra operations.

```
X = np.random.randn(2,3)                         # create a 2 x 3 random matrix
print('X =\n', X, '\n')
print('Transpose of X, X.T =\n', X.T, '\n')      # matrix transpose operation X^T

y = np.random.randn(3) # random vector
print('y =', y, '\n')

print('Matrix-vector multiplication')
print('X.dot(y) =\n', X.dot(y), '\n')            # matrix-vector multiplication  X * y

print('Matrix-matrix product')
print('X.dot(X.T) =', X.dot(X.T))        # matrix-matrix multiplication  X * X^T
print('\nX.T.dot(X) =\n', X.T.dot(X))      # matrix-matrix multiplication  X^T * X
```

```
X =
 [[-0.90806503 -1.03436344  0.3110096 ]
 [ 1.28307678 -1.41856072  1.57543665]]

Transpose of X, X.T =
 [[-0.90806503  1.28307678]
 [-1.03436344 -1.41856072]
 [ 0.3110096   1.57543665]]

y = [1.45833659 0.69836767 0.3559364 ]

Matrix-vector multiplication
X.dot(y) =
 [-1.93593081  1.4412361 ]

Matrix-matrix product
X.dot(X.T) = [[1.9912168  0.79216611]
 [0.79216611 6.14060118]]

X.T.dot(X) =
 [[ 2.47086812 -0.88085305  1.73898924]
 [-0.88085305  3.08222225 -2.55654951]
 [ 1.73898924 -2.55654951  2.57872761]]
```

```
X = np.random.randn(5,3)
print('X =\n', X, '\n')

C = X.T.dot(X)               # C = X^T * X is a square matrix
print('C = X.T.dot(X) =\n', C, '\n')

invC = np.linalg.inv(C)      # inverse of a square matrix
print('Inverse of C = np.linalg.inv(C)\n', invC, '\n')

detC = np.linalg.det(C)      # determinant of a square matrix
print('Determinant of C = np.linalg.det(C) =', detC)

S, U = np.linalg.eig(C)      # eigenvalue S and eigenvector U of a square matrix
print('Eigenvalues of C =\n', S)
print('Eigenvectors of C =\n', U)
```

```
X =
 [[ 1.1950961  -1.41438183  2.04052226]
 [-0.16963088  1.30844031  0.63923653]
 [-0.41502443 -0.5141118  -0.93847537]
 [-0.6829059   0.22118507  0.20729381]
 [ 0.2119748   0.96182991 -2.14700872]]

C = X.T.dot(X) =
 [[ 2.14056839 -1.64607003  2.12300225]
 [-1.64607003  4.95084255 -3.58640042]
 [ 2.12300225 -3.58640042 10.1057076 ]]

Inverse of C = np.linalg.inv(C)
 [[ 0.67662126  0.16421104 -0.08386765]
 [ 0.16421104  0.31173455  0.07613366]
 [-0.08386765  0.07613366  0.14359182]]

Determinant of C = np.linalg.det(C) = 54.933980999304595
Eigenvalues of C =
 [12.61209759  1.34385538  3.24116558]
Eigenvectors of C =
 [[ 0.24502994 -0.93628122 -0.25167002]
```

```
[-0.45372873 -0.34014374  0.82367012]
[ 0.85679085  0.08763392  0.50816309]]
```

2.2 Introduction to Pandas

Pandas provide two convenient data structures for storing and manipulating data: Series and DataFrame. A Series is similar to a one-dimensional array whereas a DataFrame is a tabular representation akin to a spreadsheet table.

2.2.1 Series

A Series object consists of a one-dimensional array of values, whose elements can be referenced using an index array. A Series object can be created from a list, a numpy array, or a Python dictionary. You can apply most of the numpy functions on the Series object.

```
from pandas import Series

s = Series([3.1, 2.4, -1.7, 0.2, -2.9, 4.5])   # creating a series from a list
print('Series, s =\n', s, '\n')

print('s.values =', s.values)     # display values of the Series
print('s.index =', s.index)       # display indices of the Series
print('s.dtype =', s.dtype)       # display the element type of the Series
```

```
Series, s =
 0    3.1
1    2.4
2   -1.7
3    0.2
4   -2.9
5    4.5
dtype: float64

s.values = [ 3.1  2.4 -1.7  0.2 -2.9  4.5]
s.index = RangeIndex(start=0, stop=6, step=1)
s.dtype = float64
```

```
import numpy as np

s2 = Series(np.random.randn(6))   # creating a series from a numpy ndarray
print('Series s2 =\n', s2, '\n')
print('s2.values =', s2.values)   # display values of the Series
print('s2.index =', s2.index)     # display indices of the Series
print('s2.dtype =', s2.dtype)     # display the element type of the Series
```

```
Series s2 =
 0    0.187385
1    1.298862
2    0.822272
3    1.004246
4    0.840090
5    0.842281
dtype: float64

s2.values = [0.18738485 1.29886227 0.82227185 1.00424636 0.84009013 0.84228101]
s2.index = RangeIndex(start=0, stop=6, step=1)
s2.dtype = float64
```

```
s3 = Series([1.2,2.5,-2.2,3.1,-0.8,-3.2],
            index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6',])
print('Series s3 =\n', s3, '\n')
print('s3.values =', s3.values)   # display values of the Series
print('s3.index =', s3.index)     # display indices of the Series
print('s3.dtype =', s3.dtype)     # display the element type of the Series
```

```
Series s3 =
 Jan 1    1.2
Jan 2    2.5
Jan 3   -2.2
Jan 4    3.1
Jan 5   -0.8
Jan 6   -3.2
dtype: float64

s3.values = [ 1.2  2.5 -2.2  3.1 -0.8 -3.2]
s3.index = Index(['Jan 1', 'Jan 2', 'Jan 3', 'Jan 4', 'Jan 5', 'Jan 6'], dtype='object')
s3.dtype = float64
```

```
capitals = {'MI': 'Lansing', 'CA': 'Sacramento', 'TX': 'Austin', 'MN': 'St Paul'}

s4 = Series(capitals)   # creating a series from dictionary object
print('Series s4 =\n', s4, '\n')
print('s4.values =', s4.values)   # display values of the Series
print('s4.index=', s4.index)      # display indices of the Series
print('s4.dtype =', s4.dtype)     # display the element type of the Series
```

```
Series s4 =
 MI       Lansing
CA    Sacramento
TX        Austin
MN       St Paul
dtype: object

s4.values = ['Lansing' 'Sacramento' 'Austin' 'St Paul']
s4.index= Index(['MI', 'CA', 'TX', 'MN'], dtype='object')
s4.dtype = object
```

```
s3 = Series([1.2,2.5,-2.2,3.1,-0.8,-3.2],
            index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6',])
print('s3 =\n', s3, '\n')

# Accessing elements of a Series

print('s3[2]=', s3[2])        # display third element of the Series
print('s3[\'Jan 3\']=', s3['Jan 3'])   # indexing element of a Series

print('\ns3[1:3]=')             # display a slice of the Series
print(s3[1:3])
print('\ns3.iloc([1:3])=')      # display a slice of the Series
print(s3.iloc[1:3])
```

```
s3 =
 Jan 1    1.2
Jan 2    2.5
Jan 3   -2.2
Jan 4    3.1
Jan 5   -0.8
Jan 6   -3.2
dtype: float64

s3[2]= -2.2
s3['Jan 3']= -2.2

s3[1:3]=
Jan 2    2.5
Jan 3   -2.2
dtype: float64

s3.iloc([1:3])=
Jan 2    2.5
Jan 3   -2.2
dtype: float64
```

There are various functions available to find the number of elements in a Series. Result of the function depends on whether null elements are included.

```
s3['Jan 7'] = np.nan
print('Series s3 =\n', s3, '\n')

print('Shape of s3 =', s3.shape)   # get the dimension of the Series
print('Size of s3 =', s3.size)     # get the number of elements of the Series
print('Count of s3 =', s3.count()) # get the number of non-null elements of the Series
```

```
Series s3 =
 Jan 1    1.2
Jan 2    2.5
Jan 3   -2.2
Jan 4    3.1
Jan 5   -0.8
Jan 6   -3.2
Jan 7    NaN
dtype: float64

Shape of s3 = (7,)
Size of s3 = 7
Count of s3 = 6
```

A boolean filter can be used to select elements of a Series

```
print(s3[s3 > 0])   # applying filter to select non-negative elements of the Series
```

```
Jan 1    1.2
Jan 2    2.5
Jan 4    3.1
dtype: float64
```

Scalar operations can be performed on elements of a numeric Series

```
print('s3 + 4 =\n', s3 + 4, '\n')
print('s3 / 4 =\n', s3 / 4)
```

```
s3 + 4 =
 Jan 1    5.2
Jan 2    6.5
Jan 3    1.8
Jan 4    7.1
Jan 5    3.2
Jan 6    0.8
Jan 7    NaN
dtype: float64

s3 / 4 =
 Jan 1    0.300
Jan 2    0.625
Jan 3   -0.550
Jan 4    0.775
Jan 5   -0.200
Jan 6   -0.800
Jan 7      NaN
dtype: float64
```

Numpy functions can be applied to pandas Series.

```
print('np.log(s3 + 4) =\n', np.log(s3 + 4), '\n')    # applying log function to a numeric Series
print('np.exp(s3 - 4) =\n', np.exp(s3 - 4), '\n')    # applying exponent function to a numeric Series
```

```
np.log(s3 + 4) =
 Jan 1    1.648659
Jan 2    1.871802
Jan 3    0.587787
Jan 4    1.960095
Jan 5    1.163151
Jan 6   -0.223144
Jan 7         NaN
dtype: float64

np.exp(s3 - 4) =
 Jan 1    0.060810
Jan 2    0.223130
Jan 3    0.002029
Jan 4    0.406570
Jan 5    0.008230
Jan 6    0.000747
Jan 7         NaN
dtype: float64
```

The value_counts() function can be used for tabulating the counts of each discrete value in the Series.

```
colors = Series(['red', 'blue', 'blue', 'yellow', 'red', 'green', 'blue', np.nan])
print('colors =\n', colors, '\n')

print('colors.value_counts() =\n', colors.value_counts())
```

```
colors =
 0       red
1      blue
2      blue
3    yellow
4       red
5     green
```

```
6      blue
7       NaN
dtype: object

colors.value_counts() =
 blue      3
red       2
yellow    1
green     1
dtype: int64
```

2.2.2 DataFrame

A DataFrame object is a tabular, spreadsheet-like data structure containing a collection of columns, each of which can be of different types (numeric, string, boolean, etc). Unlike Series, a DataFrame has distinct row and column indices. There are many ways to create a DataFrame object (e.g., from a dictionary, list of tuples, or even numpy's ndarrays).

```
from pandas import DataFrame

cars = {'make': ['Ford', 'Honda', 'Toyota', 'Tesla'],
       'model': ['Taurus', 'Accord', 'Camry', 'Model S'],
       'MSRP': [27595, 23570, 23495, 68000]}
carData = DataFrame(cars)            # creating DataFrame from dictionary
carData                              # display the table
```

	make	model	MSRP
0	Ford	Taurus	27595
1	Honda	Accord	23570
2	Toyota	Camry	23495
3	Tesla	Model S	68000

```
print('carData.index =', carData.index)         # print the row indices
print('carData.columns =', carData.columns)     # print the column indices

carData.index = RangeIndex(start=0, stop=4, step=1)
carData.columns = Index(['make', 'model', 'MSRP'], dtype='object')
```

Inserting columns to an existing dataframe

```
carData2 = DataFrame(cars, index = [1,2,3,4])  # change the row index
carData2['year'] = 2018    # add column with same value
carData2['dealership'] = ['Courtesy Ford','Capital Honda','Spartan Toyota','N/A']
carData2                   # display table
```

	make	model	MSRP	year	dealership
1	Ford	Taurus	27595	2018	Courtesy Ford
2	Honda	Accord	23570	2018	Capital Honda
3	Toyota	Camry	23495	2018	Spartan Toyota
4	Tesla	Model S	68000	2018	N/A

Creating DataFrame from a list of tuples.

```
tuplelist = [(2011,45.1,32.4),(2012,42.4,34.5),(2013,47.2,39.2),
              (2014,44.2,31.4),(2015,39.9,29.8),(2016,41.5,36.7)]
columnNames = ['year','temp','precip']
weatherData = DataFrame(tuplelist, columns=columnNames)
weatherData
```

	year	temp	precip
0	2011	45.1	32.4
1	2012	42.4	34.5

Creating DataFrame from numpy ndarray

3	2014	44.2	31.4

```
import numpy as np

npdata = np.random.randn(5,3)  # create a 5 by 3 random matrix
columnNames = ['x1','x2','x3']
data = DataFrame(npdata, columns=columnNames)
data
```

	x1	x2	x3
0	0.589216	-0.187104	0.156143
1	-1.253231	-1.892100	-1.023850
2	1.017221	-0.533214	-0.824243
3	0.160781	-0.031562	-0.080451
4	-1.840945	0.266364	-1.539405

There are many ways to access elements of a DataFrame object.

```
# accessing an entire column will return a Series object

print(data['x2'])
print(type(data['x2']))
```

```
0   -0.187104
1   -1.892100
2   -0.533214
3   -0.031562
4    0.266364
Name: x2, dtype: float64
<class 'pandas.core.series.Series'>
```

```
# accessing an entire row will return a Series object

print('Row 3 of data table:')
print(data.iloc[2])       # returns the 3rd row of DataFrame
print(type(data.iloc[2]))

print('\nRow 3 of car data table:')
print(carData2.iloc[2])   # row contains objects of different types
```

```
Row 3 of data table:
x1    1.017221
x2   -0.533214
x3   -0.824243
Name: 2, dtype: float64
<class 'pandas.core.series.Series'>

Row 3 of car data table:
make                  Toyota
model                  Camry
MSRP                   23495
year                    2018
dealership    Spartan Toyota
Name: 3, dtype: object
```

```
# accessing a specific element of the DataFrame

print('carData2 =\n', carData2)

print('\ncarData2.iloc[1,2] =', carData2.iloc[1,2])                # retrieving second row, third column
print('carData2.loc[1,\'model\'] =', carData2.loc[1,'model'])    # retrieving second row, column named 'model'

# accessing a slice of the DataFrame
```

```
print('\ncarData2.iloc[1:3,1:3]=')
print(carData2.iloc[1:3,1:3])
```

```
carData2 =
     make    model   MSRP  year      dealership
1    Ford   Taurus  27595  2018   Courtesy Ford
2   Honda   Accord  23570  2018   Capital Honda
3  Toyota    Camry  23495  2018  Spartan Toyota
4   Tesla  Model S  68000  2018             N/A

carData2.iloc[1,2] = 23570
carData2.loc[1,'model'] = Taurus

carData2.iloc[1:3,1:3]=
    model   MSRP
2  Accord  23570
3   Camry  23495
```

```
print('carData2 =\n', carData2, '\n')

print('carData2.shape =', carData2.shape)
print('carData2.size =', carData2.size)
```

```
carData2 =
     make    model   MSRP  year      dealership
1    Ford   Taurus  27595  2018   Courtesy Ford
2   Honda   Accord  23570  2018   Capital Honda
3  Toyota    Camry  23495  2018  Spartan Toyota
4   Tesla  Model S  68000  2018             N/A

carData2.shape = (4, 5)
carData2.size = 20
```

```
# selection and filtering

print('carData2 =\n', carData2, '\n')

print('carData2[carData2.MSRP > 25000] =')
print(carData2[carData2.MSRP > 25000])
```

```
carData2 =
     make    model   MSRP  year      dealership
1    Ford   Taurus  27595  2018   Courtesy Ford
2   Honda   Accord  23570  2018   Capital Honda
3  Toyota    Camry  23495  2018  Spartan Toyota
4   Tesla  Model S  68000  2018             N/A

carData2[carData2.MSRP > 25000] =
    make    model   MSRP  year     dealership
1   Ford   Taurus  27595  2018  Courtesy Ford
4  Tesla  Model S  68000  2018            N/A
```

2.2.3 Arithmetic Operations

```
print(data)

print('\nData transpose operation: data.T')
print(data.T)    # transpose operation

print('\nAddition: data + 4')
print(data + 4)    # addition operation

print('\nMultiplication: data * 10')
print(data * 10)   # multiplication operation
```

```
         x1        x2        x3
0  0.589216 -0.187104  0.156143
1 -1.253231 -1.892100 -1.023850
2  1.017221 -0.533214 -0.824243
3  0.160781 -0.031562 -0.080451
4 -1.840945  0.266364 -1.539405

Data transpose operation: data.T
           0         1         2         3         4
x1  0.589216 -1.253231  1.017221  0.160781 -1.840945
x2 -0.187104 -1.892100 -0.533214 -0.031562  0.266364
x3  0.156143 -1.023850 -0.824243 -0.080451 -1.539405

Addition: data + 4
```

```
         x1        x2        x3
0  4.589216  3.812896  4.156143
1  2.746769  2.107900  2.976150
2  5.017221  3.466786  3.175757
3  4.160781  3.968438  3.919549
4  2.159055  4.266364  2.460595

Multiplication: data * 10
          x1         x2         x3
0   5.892164  -1.871040   1.561433
1 -12.532310 -18.921003 -10.238503
2  10.172209  -5.332138  -8.242431
3   1.607811  -0.315621  -0.804514
4 -18.409446   2.663637 -15.394055
```

```
print('data =\n', data)

columnNames = ['x1','x2','x3']
data2 = DataFrame(np.random.randn(5,3), columns=columnNames)
print('\ndata2 =')
print(data2)

print('\ndata + data2 = ')
print(data.add(data2))

print('\ndata * data2 = ')
print(data.mul(data2))
```

```
data =
          x1        x2        x3
0  0.589216 -0.187104  0.156143
1 -1.253231 -1.892100 -1.023850
2  1.017221 -0.533214 -0.824243
3  0.160781 -0.031562 -0.080451
4 -1.840945  0.266364 -1.539405

data2 =
         x1        x2        x3
0 -1.142969  0.457884 -0.398623
1 -1.562523 -0.041757  0.079921
2  1.328234  0.609033  1.417474
3  0.578005  0.134623 -1.061997
4 -1.077484 -0.548816 -0.545277

data + data2 =
         x1        x2        x3
0 -0.553753  0.270780 -0.242480
1 -2.815754 -1.933857 -0.943930
2  2.345455  0.075819  0.593231
3  0.738786  0.103061 -1.142448
4 -2.918429 -0.282452 -2.084683

data * data2 =
         x1        x2        x3
0 -0.673456 -0.085672 -0.062242
1  1.958202  0.079008 -0.081827
2  1.351107 -0.324745 -1.168343
3  0.092932 -0.004249  0.085439
4  1.983588 -0.146185  0.839403
```

```
print(data.abs())    # get the absolute value for each element

print('\nMaximum value per column:')
print(data.max())    # get maximum value for each column

print('\nMinimum value per row:')
print(data.min(axis=1))    # get minimum value for each row

print('\nSum of values per column:')
print(data.sum())    # get sum of values for each column

print('\nAverage value per row:')
print(data.mean(axis=1))    # get average value for each row

print('\nCalculate max - min per column')
f = lambda x: x.max() - x.min()
print(data.apply(f))

print('\nCalculate max - min per row')
```

```
f = lambda x: x.max() - x.min()
print(data.apply(f, axis=1))
```

```
          x1        x2        x3
0  0.589216  0.187104  0.156143
1  1.253231  1.892100  1.023850
2  1.017221  0.533214  0.824243
3  0.160781  0.031562  0.080451
4  1.840945  0.266364  1.539405

Maximum value per column:
x1    1.017221
x2    0.266364
x3    0.156143
dtype: float64

Minimum value per row:
0   -0.187104
1   -1.892100
2   -0.824243
3   -0.080451
4   -1.840945
dtype: float64

Sum of values per column:
x1   -1.326957
x2   -2.377617
x3   -3.311807
dtype: float64

Average value per row:
0    0.186085
1   -1.389727
2   -0.113412
3    0.016256
4   -1.037995
dtype: float64

Calculate max - min per column
x1    2.858165
x2    2.158464
x3    1.695549
dtype: float64

Calculate max - min per row
0    0.776320
1    0.868250
2    1.841464
3    0.241232
4    2.107308
dtype: float64
```

The value_counts() function can also be applied to a pandas DataFrame

```
objects = {'shape': ['circle', 'square', 'square', 'square', 'circle', 'rectangle'],
           'color': ['red', 'red', 'red', 'blue', 'blue', 'blue']}

shapeData = DataFrame(objects)
print('shapeData =\n', shapeData, '\n')

print('shapeData.value_counts() =\n', shapeData.value_counts().sort_values())
```

```
shapeData =
         shape color
0     circle   red
1     square   red
2     square   red
3     square  blue
4     circle  blue
5  rectangle  blue

shapeData.value_counts() =
 shape      color
circle     blue     1
           red      1
rectangle  blue     1
square     blue     1
           red      2
dtype: int64
```

2.2.4 Plotting Series and DataFrame

There are many built-in functions available to plot the data stored in a Series or a DataFrame.

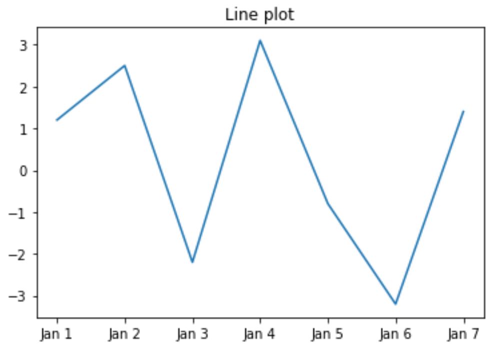
(a) Line plot

```
%matplotlib inline

s3 = Series([1.2,2.5,-2.2,3.1,-0.8,-3.2,1.4],
            index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6','Jan 7'])


s3.plot(kind='line', title='Line plot')
```

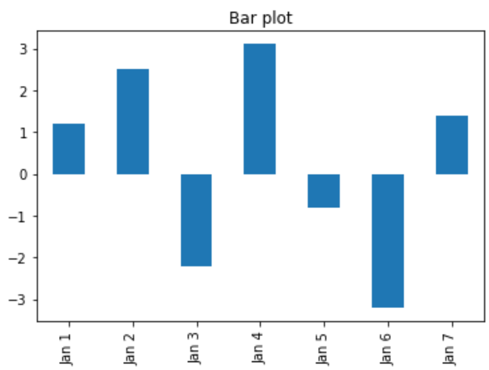
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f78dc5d0be0>
```



(b) Bar plot

```
s3.plot(kind='bar', title='Bar plot')
```

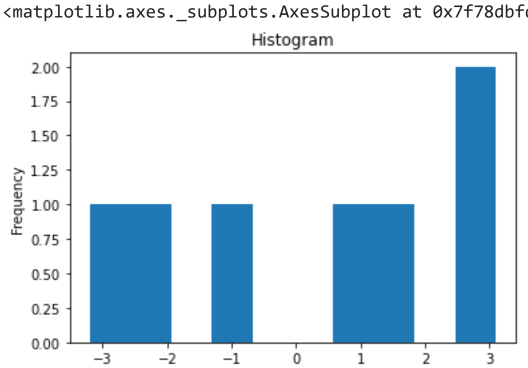
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f78dc49de20>
```



(c) Histogram

```
s3.plot(kind='hist', title = 'Histogram')
```

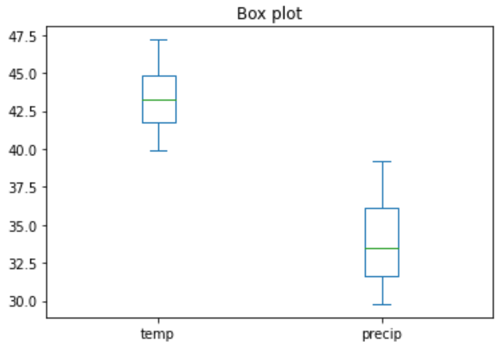
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f78dbfd36d0>
```



(d) Box plot

```
tuplelist = [(2011,45.1,32.4),(2012,42.4,34.5),(2013,47.2,39.2),
             (2014,44.2,31.4),(2015,39.9,29.8),(2016,41.5,36.7)]
columnNames = ['year','temp','precip']
weatherData = DataFrame(tuplelist, columns=columnNames)
weatherData[['temp','precip']].plot(kind='box', title='Box plot')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f78dbf4cd00>
```

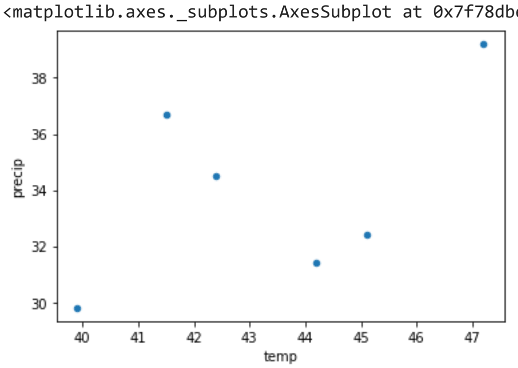


(e) Scatter plot

```
print('weatherData =\n', weatherData)

weatherData.plot(kind='scatter', x='temp', y='precip')
```

```
weatherData =
    year  temp  precip
0  2011  45.1    32.4
1  2012  42.4    34.5
2  2013  47.2    39.2
3  2014  44.2    31.4
4  2015  39.9    29.8
5  2016  41.5    36.7
<matplotlib.axes._subplots.AxesSubplot at 0x7f78dbed7b20>
```



Exercise: Explore crop production dataset

The dataset `Production_Crops_E_Asia.csv` contains crop production statistics from the United Nations Food and Agriculture Organization (UN FAO) in East Asian countries from 1961 to 2019 (source: https://data.world/agriculture/crop-production). In this exercise, you'll explore the dataset using some of the operations from Modules 1-2 in this lab.

```
import pandas as pd
```

```
# Load the CSV file from its storage location on Google Drive
prod = pd.read_csv('https://docs.google.com/uc?export=download&id=1r2DzEFS5tb5NjoIyAl6FvUDYW5DOKm5q')
```

```
prod
```

	Area Code	Area	Item Code	Item	Element Code	Element	Unit	Y1961	Y1961F	Y1962	...	Y2015	Y2015F	Y2016	Y2016F	
0	2	Afghanistan	221	Almonds, with shell	5312	Area harvested	ha	NaN	NaN	NaN	...	14676.0	NaN	19481.0	NaN	1
1	2	Afghanistan	221	Almonds, with shell	5419	Yield	hg/ha	NaN	NaN	NaN	...	16521.0	Fc	16859.0	Fc	1
2	2	Afghanistan	221	Almonds, with shell	5510	Production	tonnes	NaN	NaN	NaN	...	24246.0	NaN	32843.0	NaN	2
3	2	Afghanistan	711	Anise, badian, fennel, coriander	5312	Area harvested	ha	NaN	M	NaN	...	25000.0	F	25787.0	Im	2
4	2	Afghanistan	711	Anise, badian, fennel, coriander	5419	Yield	hg/ha	NaN	NaN	NaN	...	7200.0	Fc	6982.0	Fc	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
9633	249	Yemen	1729	Treenuts,	5419	Yield	hg/ha	NaN	NaN	NaN	...	18805.0	Fc	18061.0	Fc	1

How many unique countries are in the dataset? What are the names of the countries? Print a list of the unique countries from the 'Area' column as well as the number of unique countries.

```
UnqCountries = prod['Area'].unique()
NoOfUnqCountries = len(UnqCountries)
print('List of Countries: {}'.format(UnqCountries))
print('Number of unique countries: {}'.format(NoOfUnqCountries))
```

```
List of Countries: ['Afghanistan' 'Armenia' 'Azerbaijan' 'Bahrain' 'Bangladesh' 'Bhutan'
 'Brunei Darussalam' 'Cambodia' 'China, Hong Kong SAR' 'China, Macao SAR'
 'China, mainland' 'China, Taiwan Province of' 'Cyprus'
 "Democratic People's Republic of Korea" 'Georgia' 'India' 'Indonesia'
 'Iran (Islamic Republic of)' 'Iraq' 'Israel' 'Japan' 'Jordan'
 'Kazakhstan' 'Kuwait' 'Kyrgyzstan' "Lao People's Democratic Republic"
 'Lebanon' 'Malaysia' 'Maldives' 'Mongolia' 'Myanmar' 'Nepal' 'Oman'
 'Pakistan' 'Palestine' 'Philippines' 'Qatar' 'Republic of Korea'
 'Saudi Arabia' 'Singapore' 'Sri Lanka' 'Syrian Arab Republic'
 'Tajikistan' 'Thailand' 'Timor-Leste' 'Turkey' 'Turkmenistan'
 'United Arab Emirates' 'Uzbekistan' 'Viet Nam' 'Yemen']
Number of unique countries: 51
```

What are the most common crops included in the dataset across all countries? Use value_counts() to print the number of occurrences of each crop category (column 'Item') in the dataframe.

```
# YOUR CODE HERE
prod.value_counts('Item')
```

```
Item
Roots and Tubers, Total    152
Vegetables, fresh nes      152
Vegetables Primary         152
Fruit Primary              150
Cereals, Total             144
                          ...
Pyrethrum, dried             3
Gooseberries                 3
Agave fibres nes             3
Karite nuts (sheanuts)       2
Cassava leaves               2
Length: 165, dtype: int64
```

Print a list of the countries in the dataframe sorted by the number of rows in which it appears.

```
from operator import itemgetter
sorted([(country, df.shape[0]) for country, df in prod.groupby('Area')], key = itemgetter(1))
```

```
[('China, Macao SAR', 23),
 ('China, Hong Kong SAR', 59),
 ('Mongolia', 74),
 ('Singapore', 76),
 ('Bahrain', 89),
 ('Maldives', 93),
```

```
 ('Qatar', 101),
 ('Oman', 102),
 ('Brunei Darussalam', 105),
 ('Saudi Arabia', 112),
 ('United Arab Emirates', 112),
 ('Kuwait', 119),
 ('Timor-Leste', 119),
 ('Turkmenistan', 119),
 ('Cambodia', 128),
 ("Democratic People's Republic of Korea", 138),
 ("Lao People's Democratic Republic", 138),
 ('Afghanistan', 145),
 ('Armenia', 164),
 ('Viet Nam', 165),
 ('Myanmar', 168),
 ('Tajikistan', 172),
 ('Malaysia', 174),
 ('Bhutan', 178),
 ('Sri Lanka', 180),
 ('Georgia', 190),
 ('Nepal', 199),
 ('Yemen', 208),
 ('Iraq', 220),
 ('Indonesia', 221),
 ('Bangladesh', 223),
 ('Jordan', 224),
 ('Lebanon', 228),
 ('Azerbaijan', 230),
 ('Kazakhstan', 230),
 ('Republic of Korea', 230),
 ('Palestine', 243),
 ('Kyrgyzstan', 251),
 ('Philippines', 254),
 ('Syrian Arab Republic', 254),
 ('Thailand', 257),
 ('Israel', 259),
 ('Pakistan', 263),
 ('Uzbekistan', 264),
 ('China, Taiwan Province of', 267),
 ('Cyprus', 273),
 ('India', 284),
 ('Japan', 288),
 ('Iran (Islamic Republic of)', 302),
 ('Turkey', 325),
 ('China, mainland', 398)]
```

What are the trends in crop production for cereal crops (includes maize, wheat, etc.) from 1961 to 2019? Plot a line plot of crop production for the following countries: India, Japan, Iran, Turkey, and China (mainland). Only plot the values in the columns with format 'YXXXX'.

Step 1: filter the dataframe to only include rows for the specified countries.

```
COUNTRY_LIST = ['India', 'Japan', 'Iran (Islamic Republic of)', 'Turkey','China, mainland']
sel_prod = prod[prod['Area'].isin(COUNTRY_LIST)]
```

Step 2: filter the columns to only include Area, Item, Element, and those matching the pattern 'YXXXX'.

```
year_cols = [col for col in sel_prod.columns if (col.startswith('Y') and len(col) == 5)]

sel_prod = sel_prod[['Area', 'Item', 'Element'] + year_cols]

sel_prod
```

	Area	Item	Element	Y1961	Y1962	Y1963	Y1964	Y1965	Y1966	Y1967	...	Y2010	Y2011
1344	China, mainland	Almonds, with shell	Area harvested	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	13000.0	14000.0
1345	China, mainland	Almonds, with shell	Yield	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	29231.0	30000.0
1346	China, mainland	Almonds, with shell	Production	5000.0	5500.0	5800.0	6500.0	7500.0	8000.0	7800.0	...	38000.0	42000.0
1347	China, mainland	Anise, badian, fennel, coriander	Area harvested	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	37000.0	37522.0
1348	China, mainland	Anise, badian, fennel, coriander	Yield	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	11892.0	12153.0
...	...	...	...	...	...	...	...	...	...	...	...	...	..
8765	Turkey	Treenuts, Total	Yield	6935.0	8080.0	6837.0	9836.0	5754.0	9213.0	6391.0	...	16450.0	13676.0
8766	Turkey	Treenuts, Total	Production	227780.0	272310.0	232590.0	344415.0	204670.0	331900.0	233800.0	...	1023186.0	857854.0

Step 3: filter that dataframe to include only the rows for which Element has the value 'Production' and Item contains 'Cereals, Total'.

```
filterCols =  (sel_prod['Element'] == 'Production') & (sel_prod['Item'].str.contains("Cereals, Total"))
sel_prod = sel_prod[filterCols]
sel_prod
```

	Area	Item	Element	Y1961	Y1962	Y1963	Y1964	Y1965	Y1966	Y1967	...	Y20
1708	China, mainland	Cereals, Total	Production	107000000.0	117640008.0	134720016.0	149430000.0	159090000.0	174500000.0	178020016.0	...	496343192
2860	India	Cereals, Total	Production	87376496.0	87257552.0	90373008.0	93706000.0	79699504.0	80137608.0	95453504.0	...	267838308
3383	Iran (Islamic Republic of)	Cereals, Total	Production	4303119.0	4402740.0	4101140.0	4299578.0	5642713.0	6588982.0	5833000.0	...	19597232
4150	Japan	Cereals, Total	Production	20318660.0	20635150.0	19470310.0	19079380.0	18925506.0	18915600.0	21023600.0	...	11454614
8736	Turkey	Cereals, Total	Production	12729100.0	14728000.0	17487100.0	14454700.0	14756700.0	16511200.0	16962300.0	...	32764875

5 rows × 62 columns

Step 4: Plot the line plot of production values from 1961 to 2019 for each of the 5 countries (each country should be a separate line). Only the 'YXXXX' columns should be used for the line plot values.

Note: pandas by default plots by column, so to plot each of the countries as its own line, we need to transpose the dataframe so that each column contains the values for one country.

```
sel_prod_t = sel_prod.T
```

```
sel_prod_t
```

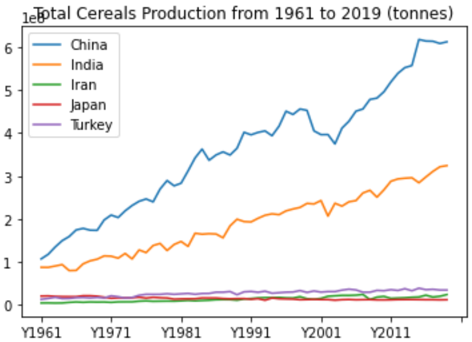
	1708	2860	3383	4150	8736
Area	China, mainland	India	Iran (Islamic Republic of)	Japan	Turkey
Item	Cereals, Total	Cereals, Total	Cereals, Total	Cereals, Total	Cereals, Total
Element	Production	Production	Production	Production	Production
Y1961	107000000.0	87376496.0	4303119.0	20318660.0	12729100.0
Y1962	117640008.0	87257552.0	4402740.0	20635150.0	14728000.0
...	...	...	...	...	...
Y2015	618165315.0	284333000.0	18255385.0	12141378.0	38632438.0

```
sel_prod_t.columns = ['China', 'India', 'Iran', 'Japan', 'Turkey']
```

Y2017	614037296.0	310782330.0	18120141.0	11903422.0	36126157.0

```
sel_prod_t[3:].plot(kind='line', title='Total Cereals Production from 1961 to 2019 (tonnes)')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f78dc6a55e0>
```



✓ 0s completed at 12:29 AM