ECE368: Probabilistic Reasoning
A Quick Tutorial on Python for Labs

January 31, 2019

We will use Python for all labs in this course. This document aims to help you get familiar with Python and its scientific computation libraries NumPy and Matplotlib that will be used in the labs. This document provides a quick introduction. To learn more about the Python and its scientific computing tools, the following two links are very helpful:

https://docs.python.org/3/tutorial/, official Python tutorial
https://scipy-lectures.org/, Python Scientific Lecture Notes

1 Python Environment Setup

It is recommended to install Anaconda, which is a open-source and cross-platform Python distribution suitable for scientific computing. Please make sure to download the Python 3 version (latest 3.7) of Anaconda, as all programs in the labs are required to be written and run in Python 3.

Once Anaconda is installed, open Spyder from Anaconda Navigator. You can write, debug, and test the Python programs conveniently in Spyder. To run your code in Spyder, click Run file in the toolbar or press F5. You can also run your code from the console window by typing run filename.py.

2 Python Basics

This section focuses on some basics of Python, including data types, if tests and for loops, functions, classes, and input/output operations.

2.1 Types and Operations

Numbers: Numbers include *integers* and *floating-point* numbers. Normal mathematical operations are supported in Python. For example, the plus sign + performs addition, one star ∗ performs multiplication, and two stars ∗∗ performs exponentiation.

Example 1: numbers

```
# numbers
a = 1  # integer
b = 2.0 # float

# some operations on numbers
print(a + 1) # addition
print(a * 2) # multiplication
print(a ** 2) # a to the power 2
print(a / b) # division, the answer is a floating-point number
```

Strings: Strings are used to record textual information, which can be seen as sequences of characters. Note that strings can NOT be modified by any of the operations run on the strings.

Example 2: strings

```
# string
word = 'spam'

# some operations on strings
print(word[0]) # fetch the first letter in 'spam' (in Python index starts from 0)
print(word[0:3]) # slice of 'spam' from offsets 0 through 2 (not 3), result is 'spa'
print(word + 'email') # concatenation, result is 'spamemail'
```

Lists: Lists are ordered collections of arbitrarily typed objects. Lists have no fixed size and lists can be modified, which differ from strings.

Example 3: lists

```
# list creation
l = [1,'spam',2.0]  # a list of three items
m = [[1,2,3],[4,5,6],[7,8,9]] # nested list, which is similar to a matrix
r = list(range(4)) # [0,1,2,3]

# some operations on lists
l.append('ham') # add an item at end of l
print(l) # print the modified list
r.pop(0) # delete the item in r at position 0
print(r) # print the modified list
print([row[0] for row in m]) # comprehension expression, show the first column of m
```

Dictionaries: Dictionaries are collections of "key: value" pairs without order. Dictionaries can be modified.

Example 4: dictionaries

```
# dictionary creation
dict = {'spam': 1, 'ham': 2} # 'spam' and 'ham' are keys, 0 and 1 are the values

# some operations on dictionaries
print(dict['spam']) # fetch the value of key 'spam'; nonexistent key returns an error
print(dict.get('msg', 0))  # fetch the value of 'msg'; return 0 if 'msg' not exist
dict['spam'] += 1 # change the value of key 'spam'
dict['msg'] = 3 # add a new key 'msg' with value 2
```

Tuples and sets: Tuples are like lists but cannot be modified. Sets are unordered collections of unique objects.

Example 5: sets and tuples

```
# tuple creation and some operations
t = (1,2,3,4)
print(len(t)) # length of t
print(t[0]) # fetch the first item in t

# set creation and membership testing
x = {'s','p', 'a', 'm'}
print('p' in x) # membership testing, the result is True
```

2.2 if Tests and for Loops

if Statements: if statements select actions to perform. It takes the form of an if test, followed by one or more optional elif tests and a final optional else block.

Example 6: if statement

```
# multiway branching
x = 'red'
if x == 'red':
    print('The color is red')
elif x == 'blue':
    print('The color is blue')
else:
    print('Unknown color')
```

for Loops: for loops are used to step across the items in any iterable objects such as strings, lists, tuples, and dictionaries (keys and values).

Example 7: for loop

```
# for loop working on list
l = [1,2,3,4,5,6] # a list
for x in l:
    print(x ** 2) # print 1,4,9,16,25,36 sequentially

# for loop working on dictionary
dict = {'a': 0, 'b': 1, 'c': 2, 'd': 3} # a dictionary
for key in dict:
    print(key, '=', dict[key]) # print a=0,b=1,c=2,d=3 sequentially

# for loop with function range
for i in range(10):
    print('index',i)  # print index 0, index 1,...,index 9 sequentially
```

2.3 Functions and Classes

Functions: Functions in Python are created by def statements, which consist of a header line followed by a block of statements which is the main body of the function.

Example 8: create a Python function

```
# define a function that computes the product of two inputs
def product(x, y):
    return x * y

# call the function in a for loop
for x in range(5):
    print(product(x, 2)) # print 0,2,4,6,8
```

Classes: Classes are created by class statements. A class contains several attributes and methods. Once a class is defined, subclasses can be created by inheritance.

Example 9: define a class and a subclass

```
class Book:
    # constructor
    def __init__(self,bookname,price): # constructor takes two arguments
        self.bookname = bookname  # self is the instance object
        self.price = price   # fields are filled out when created

    # method
    def promotion(self,percent):
        self.price = int(self.price * (1 - percent))

# subclass inherits class Book
class Novel(Book):

    # method
    def promotion(self,percent):
        Book.promotion(self,percent + 0.05)

# instance creation
b_1 = Book('ABC',20) # create an instance of class Book
b_1.promotion(0.1) # call the instance method
print(b_1.price) # result is 18

b_2 = Novel('XYZ',40) # construct an instance of of class Manager
b_2.promotion(0.1) # call the instance method
print(b_2.price) # result is 34
```

2.4 Input and Output

The function open can be used for reading and writing files.

Example 10: reading words from a text file

```
with open('words.txt','r') as f: # open file words.txt, which contains one sentence
                                 # "Python numerical modules are efficient."
    read_data = f.read()  # the entire contents of words.txt is read
                          # return a string

for word in read_data.split(): # break up the string by using whitespace as separator
    print(word) # Prints "Python" "numerical" "modules" "are" "efficient."
```

3 NumPy

NumPy is a library for scientific computing with Python. NumPy's main object is multi-dimensional array, which is a table of elements with the same type and indexed by a tuple of positive integers. This section focuses on array creation and array operations.

Note that to use library NumPy, the following line should be included at the head of the Python script.

```
import numpy as np
```

3.1 Array Creation

Numpy arrays can be created from Python lists or tuples using the np.array function, or initialized by several other functions such as np.zeros and np.ones.

Example 11: array creation

```
import numpy as np

# Array creation from (nested) lists
A1 = np.array([1, 2, 3])  # one-dimensional array
A2 = np.array([[1, 2],[3, 4]]) # two-dimensional array

# Array creation by initialization
A3 = np.ones(10) # one-dimensional array of all ones
A4 = np.ones((10, 2))  # two-dimensional (10-by-2) array of all ones
A5 = np.zeros((2, 10)) # two-dimensional (2-by-10) array of all zeros
A6 = np.eye(3) # 3-by-3 identity array
A7 = np.arange(0,10) # array whose entries are 0,1,2,...9 (not 10)
A8 = np.random.normal(0,1,(4,3)) # 4-by-3 array from normal distribution N(0,1)
```

3.2 Math Operations

Math operators apply on arrays elementwise. For example, the operator $*$ performs elementwise product in arrays. The regular matrix product can be performed using the function np.dot.

Example 12: math operations

```
import numpy as np

A = np.array([[1,2],[3,4]]) # two-dimensional array
B = np.array([[2,0],[5,1]]) # two-dimensional array
C = np.array([10,100]) # one-dimensional array

# matrix operations
print(A.T) # array transpose
print(A + B) # elementwise addition
print(A * B) # elementwise product
print(A.dot(B)) # regular matrix product
print(np.dot(A, B)) # regular matrix product (another way)
print(A.dot(C)) # inner product of C and every row in A, since C is a 1D array

# sum operations on arrays
print(np.sum(A)) # sum of all entries in the array
print(np.sum(A, axis=0)) # sum of each column, the result is a 1D array
print(np.sum(A, axis=1)) # sum of each row, the result is a 1D array
```

3.3 Indexing and Slicing

NumPy arrays have one index per axis, and the indices are given in a tuple separated by commas. The items of arrays can be accessed with the indices. Arrays also can be sliced similar to Python lists.

Example 13: array indexing and slicing

```
import numpy as np

A = np.array([[1,2,3], [4,5,6], [7,8,9]])
a = A[0,1] # fetch the entry located in the first row and the second column
print(a)   # the result is 2
Asub = A[0:2,1:3] # the first 2 rows and columns 1 and 2 of A
print(Asub) # The result is [[2 3]
            #                 [5 6]]
row_A1 = A[0,:]    # the first row of A, as 1D array
print(row_A1)      # prints [1 2 3]
row_A2 = A[0:1,:]  # the first row of A, as 2D array
print(row_A2)      # prints [[1 2 3]], differs from [1 2 3] above
b = A[np.where(A > 4)] # select the entries of A that are greater than 4
print(b)               # prints [5,6,7,8,9]
```

3.4 Stacking and Shape Manipulation

Arrays can be stacked together along different axes by using np.vstack and np.hstack. The shape of an array can be modified with data unchanged using np.reshape.

Example 14: array stacking and reshaping

```
import numpy as np

A = np.array([1,2,3,4,5,6])
B = A.reshape(2,3)  # change the shape of A to 2-by-3
print(B)   # prints [[1 2 3]
           #          [4 5 6]]

C = np.array([[1,2],[3,4]])
D = np.array([[5,6],[7,8]])

V = np.vstack((C,D)) # stack arrays vertically
print(V)  # the result is [[1 2]
          #                 [3 4]
          #                 [5 6]
          #                 [7 8]]
H = np.hstack((C,D)) # stack arrays horizontally
print(H)  # the result is [[1 2 5 6]
          #                 [3 4 7 8]]
```

4 Matplotlib

Matplotlib is a Python library produceing 2D figures. We will make use of the pyplot module in Matplotlib, which provides a MATLAB-like interface for plotting. The plotting commands in pyplot are very similar to those in MATLAB.

Note that to make use of pyplot, the following line should be included at the head of the Python script.

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

4.1 Simple 2D Plots

The most commonly used plotting function may be plt.plot.

Example 15: a simple 2D plot

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

# generate x and y from a sine curve
x = np.arange(0,2 * np.pi,0.1)
y = np.sin(x)

# plot the x versus y curve with red
plt.plot(x, y, 'r')   # 'r' means red
plt.show()
```

The figure produced by the code above is shown in Fig 4.1

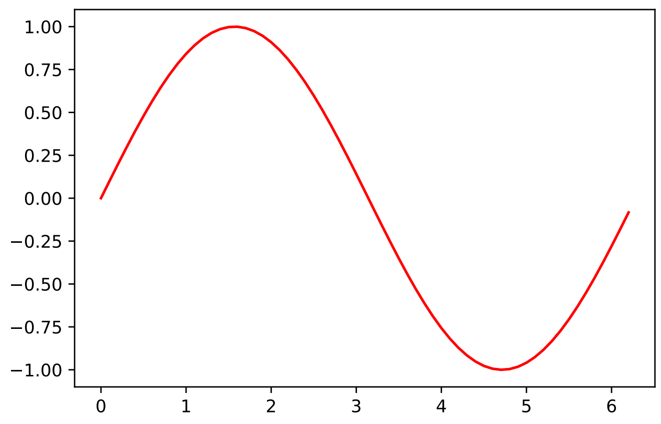

Figure 4.1: A 2D plot of a sine curve.

4.2 Scatter Plots

It is convenient to use function plt.scatter to create a scatter plot in which each circle represents a data point.

Example 16: a scatter plot of random data

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

# generate x and y from normal distribution N(0,1)
x = np.random.normal(0,1,20) # x specifies the locations on x-axis
y = np.random.normal(0,1,20) # y specifies the locations on y-axis

plt.scatter(x,y,color = 'blue') # the color of the circles is blue
plt.show()
```

The figure produced by the code above is shown in Fig 4.2

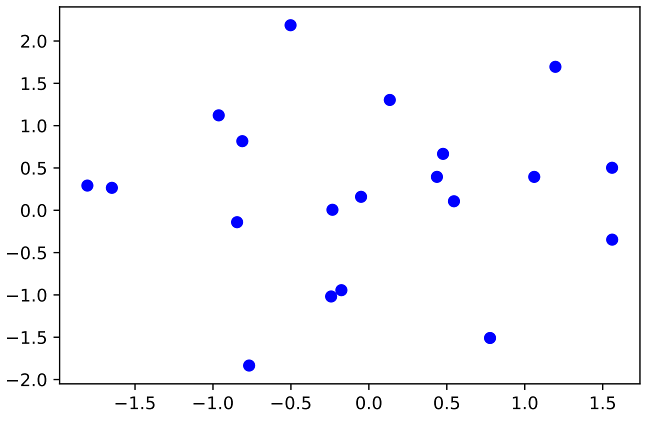

Figure 4.2: A scatter plot of random data

4.3 Contour Plots

The function plt.contour creates a contour plot of a two-variable function $z = f(x, y)$. Note that before calling plt.contour, it is necessary to build a grid first via function plt.meshgrid in the x-y plane, such that a two-dimensional array of z can be computed.

Example 17: a contour plot of $z = \sin(x^2 + y^2)$

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

# generate the x, y coordinates to build a grid
x = np.arange(-1, 1, 0.1)  # x = [-1,-0.9,...,0.8,0.9]
y = np.arange(-1, 1, 0.1)  # y = [-1,-0.9,...,0.8,0.9]

# build a grid on x-y plane, and compute values of z for all points in the grid
X, Y = np.meshgrid(x, y) # X and Y contains the x and y coordinates of all points
Z = np.sin(X ** 2 + Y ** 2) # Z is a two-dimensional array

# plot the contours
plt.contour(X,Y,Z)
plt.show()
```

The figure produced by the code above is shown in Fig 4.3

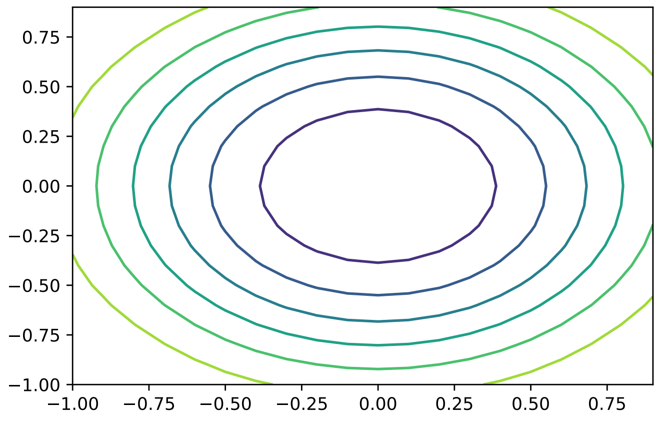

Figure 4.3: A contour plot of $z = \sin(x^2 + y^2)$

4.4 Multiple Plots in One Python Script

Two or more figures can be created in one Python script using the method in the following example.

Example 18: two plots (a filled contour plot, and a unfilled contour plot with specified level)

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

x = np.arange(-4, 4, 0.1)
y = np.arange(-4, 4, 0.1)
X, Y = np.meshgrid(x, y)
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

plt.figure(1) # figure 1
plt.contourf(X,Y,Z)  # filled contour plot

plt.figure(2) # figure 2
plt.contour(X,Y,Z,0) # the contour line with specified level z = 0

plt.show()
```

The two figures produced by the code above are shown in Fig 4.4 and Fig 4.5.

4.5 Adding Title, Legend, and Axis Labels

Here is an example on how to add title, legend, and axis labels to a figure.

Example 19: title, legend, and axis labels

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

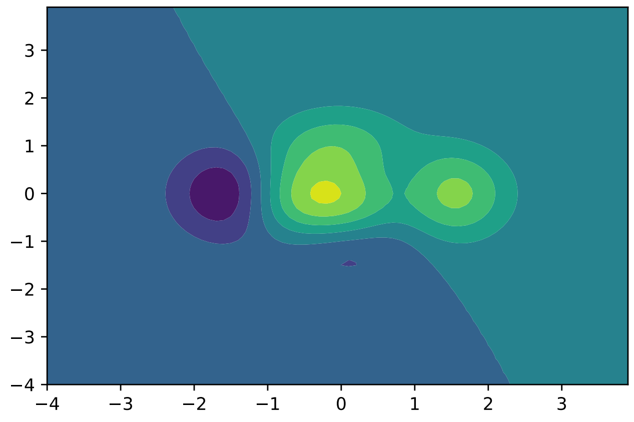

Figure 4.4: A filled contour plot.

```
x = np.arange(0,2 * np.pi,0.1)
y_s, y_c = np.sin(x), np.cos(x)
plt.plot(x, y_s,'r') # red
plt.plot(x, y_c,'g') # green

# adding labels, title and legends
plt.xlabel('x')
plt.ylabel('y')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

The figure produced by the code above is shown in Fig 4.6

4.6 Saving Plots as PDF files

The function plt.savefig can be used to save the figures as PDF files.

Example 20: saving figures

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3], [2, 3, 4], 'r')

# save the plot in a pdf file
plt.savefig("line.pdf")
```

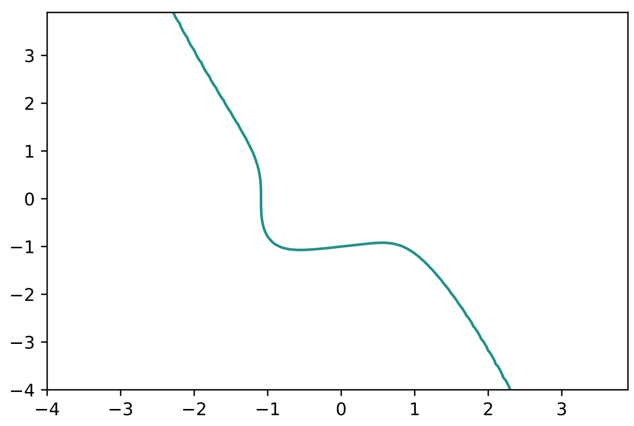

Figure 4.5: A unfilled contour plot with specified level $z = 0$.

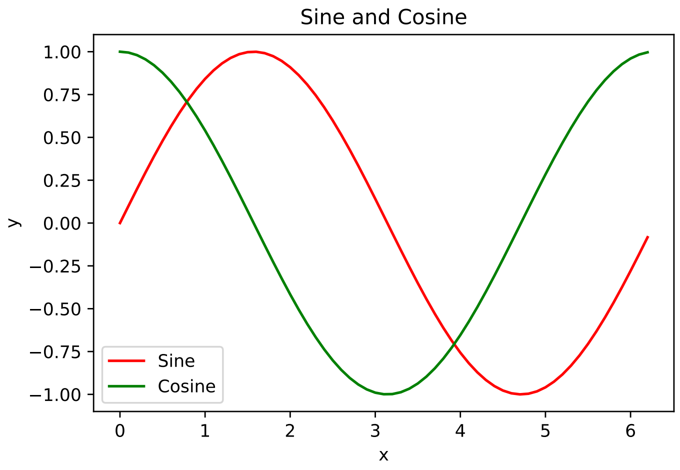

Figure 4.6: A plot with title, legend, and axis labels.