Hacettepe University

Computer Engineering Department

Programming in  python

BBM103 Introduction to Programming Lab 1

Week 11

Fall 2016

Scientific Computing with Python

- **NumPy** (http://www.numpy.org) is the fundamental package for scientific computing with Python.

- NumPy provides the most important data types for econometrics, statistics and numerical analysis – **arrays and matrices**.

Array Creation

```
67 import numpy as np
68 x = np.array([1, 2, 3])
69 print(x)
70 y = np.arange(10)  # like Python's range, but returns an array
71 print(y)
```

Output:

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

Basic Operations

```
75 a = np.array([1, 2, 3, 6])
76 b = np.linspace(0, 2, 4)  # create an array with four equally spaced points starting with 0 and ending with 2.
77 c = a - b
78 print(c)
79 print(a**2)
```

Output:

```
[ 1.          1.33333333  1.66666667  4.        ]
[ 1  4  9 36]
```

Universal functions

```
84 a = np.linspace(-np.pi, np.pi, 100)
85 b = np.sin(a)
86 c = np.cos(a)
87
88 print (a)
```

Output:

```
[-3.14159265 -3.07812614 -3.01465962 -2.9511931  -2.88772658 -2.82426006
 -2.76079354 -2.69732703 -2.63386051 -2.57039399 -2.50692747 -2.44346095
 -2.37999443 -2.31652792 -2.2530614  -2.18959488 -2.12612836 -2.06266184
 -1.99919533 -1.93572881 -1.87226229 -1.80879577 -1.74532925 -1.68186273
 -1.61839622 -1.5549297  -1.49146318 -1.42799666 -1.36453014 -1.30106362
 -1.23759711 -1.17413059 -1.11066407 -1.04719755 -0.98373103 -0.92026451
 -0.856798   -0.79333148 -0.72986496 -0.66639844 -0.60293192 -0.53946541
 -0.47599889 -0.41253237 -0.34906585 -0.28559933 -0.22213281 -0.1586663
 -0.09519978 -0.03173326  0.03173326  0.09519978  0.1586663   0.22213281
  0.28559933  0.34906585  0.41253237  0.47599889  0.53946541  0.60293192
  0.66639844  0.72986496  0.79333148  0.856798    0.92026451  0.98373103
  1.04719755  1.11066407  1.17413059  1.23759711  1.30106362  1.36453014
  1.42799666  1.49146318  1.5549297   1.61839622  1.68186273  1.74532925
  1.80879577  1.87226229  1.93572881  1.99919533  2.06266184  2.12612836
  2.18959488  2.2530614   2.31652792  2.37999443  2.44346095  2.50692747
  2.57039399  2.63386051  2.69732703  2.76079354  2.82426006  2.88772658
  2.9511931   3.01465962  3.07812614  3.14159265]
```

Linear Algebra

```
93 from numpy.random import rand
94 from numpy.linalg import solve, inv
95 a = np.array([[1, 2, 3], [3, 4, 6.7], [5, 9.0, 5]])
96 print(a)
97 print(a.transpose())
98 print(inv(a))
99 b =  np.array([3, 2, 1])
100 print(b)
101 print(solve(a, b))  # solve the equation ax = b
102 c = rand(3, 3)  # create a 3x3 random matrix
103 print(c)
104 print(np.dot(a, c))  # matrix multiplication
105 print(a @ c) # Starting with Python 3.5 and NumPy 1.10
106
```

print(a)

print(a.transpose())

print(inv(a))

print(b)

print(solve(a,b))

print(c)

print(np.dot(a,c))

print(a@c)

```
Output:
[[ 1.   2.   3. ]
 [ 3.   4.   6.7]
 [ 5.   9.   5. ]]
[[ 1.   3.   5. ]
 [ 2.   4.   9. ]
 [ 3.   6.7  5. ]]
[[-2.27683616  0.96045198  0.07909605]
 [ 1.04519774 -0.56497175  0.1299435 ]
 [ 0.39548023  0.05649718 -0.11299435]]
[3 2 1]
[-4.83050847  2.13559322  1.18644068]
[[ 0.83843849  0.99200584  0.3483117 ]
 [ 0.1630258   0.54165777  0.34870755]
 [ 0.02351926  0.67038873  0.28482933]]
[[ 1.23504786  4.08648756  1.90021478]
 [ 3.32499769  9.63425306  4.34812178]
 [ 5.77702094 13.18689276  6.30407306]]
[[ 1.23504786  4.08648756  1.90021478]
 [ 3.32499769  9.63425306  4.34812178]
 [ 5.77702094 13.18689276  6.30407306]]
```

Elementary statistics

Example: Mean

```
def mean_func(lst):
    mn = sum(lst)/len(lst)
    return mn
data_in = [2.75, 2.86, 3.37, 2.76, 2.62, 3.49, 3.05, 3.12]
print ("The mean is %.4f." % mean_func(data_in))
```

Output:

```
The mean is 3.0025.
```

Example: Mean with NumPy

```
import numpy as np


data_in = [2.75, 2.86, 3.37, 2.76, 2.62, 3.49, 3.05, 3.12]
mean_data = np.mean(data_in)
print ("The mean is ",mean_data)
```

Output:

```
The mean is 3.0025
```

Example: Median

```
def median(lst):
    quotient, remainder = divmod(len(lst), 2)
    if remainder:
        return sorted(lst)[quotient]
    return sum(sorted(lst)[quotient - 1:quotient + 1]) / 2.

data_in = [2.75, 2.86, 3.37, 2.76, 2.62, 3.49, 3.05, 3.12]
print(median(data_in))
```

Output:

```
2.955
```

Example: Median with NumPy

```
data_in = [2.75, 2.86, 3.37, 2.76, 2.62, 3.49, 3.05, 3.12]
median_data = np.median(data_in)
print ("The median is:", median_data)
```

Output:

```
The median is:
2.955
```

Example: Mod

```
from collections import Counter
mode_data = [1, 2, 4, 1, 5, 3, 2, 1, 5]
mode_counts = Counter(mode_data) # turn the list into a Counter objectprint mode_counts #  a python dictionary
print(mode_counts)
max_counts = mode_counts.most_common(1)
print(max_counts)
```

Output:

```
Counter({1: 3, 2: 2, 5: 2, 3: 1, 4: 1})
[(1, 3)]
```

Example: Correlation with numpy

```
108 a=np.correlate([1, 2, 3], [0, 1, 0.5])
109 print(a)
110 b=np.correlate([1, 2, 3], [0, 1, 0.5], "same")
111 print(b)
112 c=np.correlate([1, 2, 3], [0, 1, 0.5], "full")
113 print(c)
```

Output:

```
[ 3.5]
[ 2.   3.5  3. ]
[ 0.5  2.   3.5  3.   0. ]
```

Example: Variation with numpy

```
117 a = np.array([[1, 2], [3, 4]])
118 print(np.var(a))
119 print(np.var(a, axis=0))
120 print(np.var(a, axis=1))
```

Output:

```
1.25
[ 1.   1.]
[ 0.25  0.25]
```

In single precision, var() can be inaccurate:

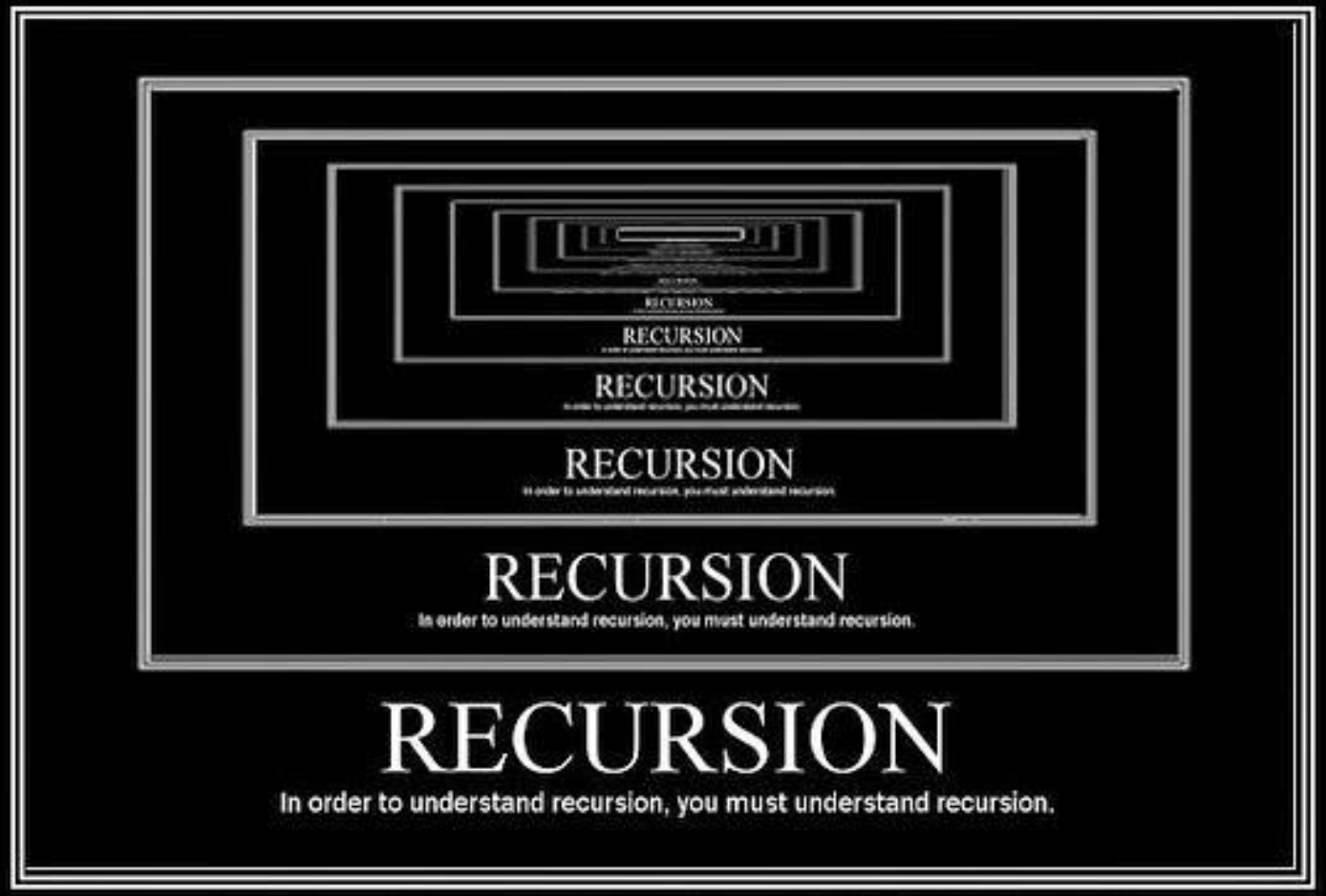
```
123 a = np.zeros((2, 512*512), dtype=np.float32)
124 a[0, :] = 1.0
125 a[1, :] = 0.1
126 print(a)
127
128 print(np.var(a))
```

```
[[ 1.   1.   1.  ...,  1.   1.   1. ]
 [ 0.1  0.1  0.1 ...,  0.1  0.1  0.1]]
0.2025
```

Computing the variance in float64 is more accurate:

```
131 print(str(np.var(a, dtype=np.float64)))
```

```
0.202499999329
```



RECURSION

In order to understand recursion, you must understand recursion.

Example: Fibonacci Numbers

The Fibonacci numbers are the numbers of the following sequence of integer values:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
The Fibonacci numbers are defined by:
$F_n = F_{n-1} + F_{n-2}$
with $F_0 = 0$ and $F_1 = 1$

```
def fibonacci(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
number=input("Please enter a number to print fibonacci numbers!")
print(fibonacci(int(number)))
```

Output:

```
Please enter a number to print fibonacci numbers!4
3
```

Example: Calculate factorial

We can track how the function works by adding two print() function to the previous function definition:

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(" ,n-1, "): ",res)
        return res

number=input("Please enter a number to caluclate factorial!")
print(factorial(int(number)))
```

Output:

```
Please enter a number to calculate factorial!5
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for  2  * factorial( 1 ):  2
intermediate result for  3  * factorial( 2 ):  6
intermediate result for  4  * factorial( 3 ):  24
intermediate result for  5  * factorial( 4 ):  120
120
```

Example: Calculate factorial

```
1  def factorial(n):
2      if n == 1:
3          return 1
4      else:
5          return n * factorial(n-1)
6
7  number=input("Please enter a number to caluclate factorial!")
8  print(factorial(int(number)))
9
```

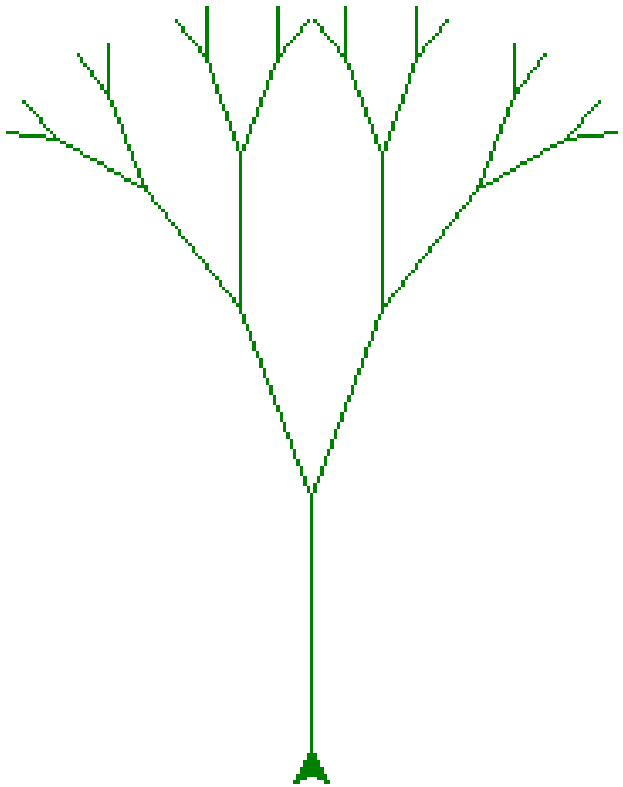
Output:

```
Please enter a number to calculate factorial! 5
120
```

Example: Visualizing Recursion

```
34 import turtle
35
36 def tree(branchLen,t):
37     if branchLen > 5:
38         t.forward(branchLen)
39         t.right(20)
40         tree(branchLen-15,t)
41         t.left(40)
42         tree(branchLen-15,t)
43         t.right(20)
44         t.backward(branchLen)
45
46 def main():
47     t = turtle.Turtle()
48     myWin = turtle.Screen()
49     t.left(90)
50     t.up()
51     t.backward(100)
52     t.down()
53     t.color("green")
54     tree(75,t)
55     myWin.exitonclick()
56
57 main()
```

Output:



Example: Computing Exponent

```
def exp(x, n):
    """
    Computes the result of x raised to the power of n.

        >>> exp(2, 3)
        8
        >>> exp(3, 2)
        9
    """
    if n == 0:
        return 1
    else:
        return x * exp(x, n-1)

number1=input("print a number as base")
number2=input("print a number as exponent")
print(exp(int(number1),int(number2)))
```

Lets look at the execution pattern.

```
exp(2, 4)
+-- 2 * exp(2, 3)
|       +-- 2 * exp(2, 2)
|       |       +-- 2 * exp(2, 1)
|       |       |       +-- 2 * exp(2, 0)
|       |       |       |       +-- 1
|       |       |       +-- 2 * 1
|       |       |       +-- 2
|       |       +-- 2 * 2
|       |       +-- 4
|       +-- 2 * 4
|       +-- 8
+-- 2 * 8
+-- 16
```

We can compute exponent in fewer steps if we use successive squaring.

```
def fast_exp(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return fast_exp(x*x, n/2)
    else:
        return x * fast_exp(x, n-1)

number1=input("print a number as base")
number2=input("print a number as exponent")
print(fast_exp(int(number1),int(number2)))

```

Lets look at the execution pattern now.

```
fast_exp(2, 10)
+-- fast_exp(4, 5) # 2 * 2
|   +-- 4 * fast_exp(4, 4)
|   |       +-- fast_exp(16, 2) # 4 * 4
|   |       |   +-- fast_exp(256, 1) # 16 * 16
|   |       |   |   +-- 256 * fast_exp(256, 0)
|   |       |   |               +-- 1
|   |       |   |   +-- 256 * 1
|   |       |   |   +-- 256
|   |       |   +-- 256
|   |       +-- 256
|   +-- 4 * 256
|   +-- 1024
+-- 1024
1024
```

Example: Flatten a List

```
39 def flatten_list(a, result=None):
40     if result is None:
41         result = []
42 
43     for x in a:
44         if isinstance(x, list):
45             flatten_list(x, result)
46         else:
47             result.append(x)
48 
49     return result
50 listToFlat=[ [1, 2, [3, 4] ], [5, 6], 7]
51 print(listToFlat)
52 faltList=flatten_list(listToFlat)
53 print(faltList)
```

Output:

```
[[1, 2, [3, 4]], [5, 6], 7]
[1, 2, 3, 4, 5, 6, 7]
```