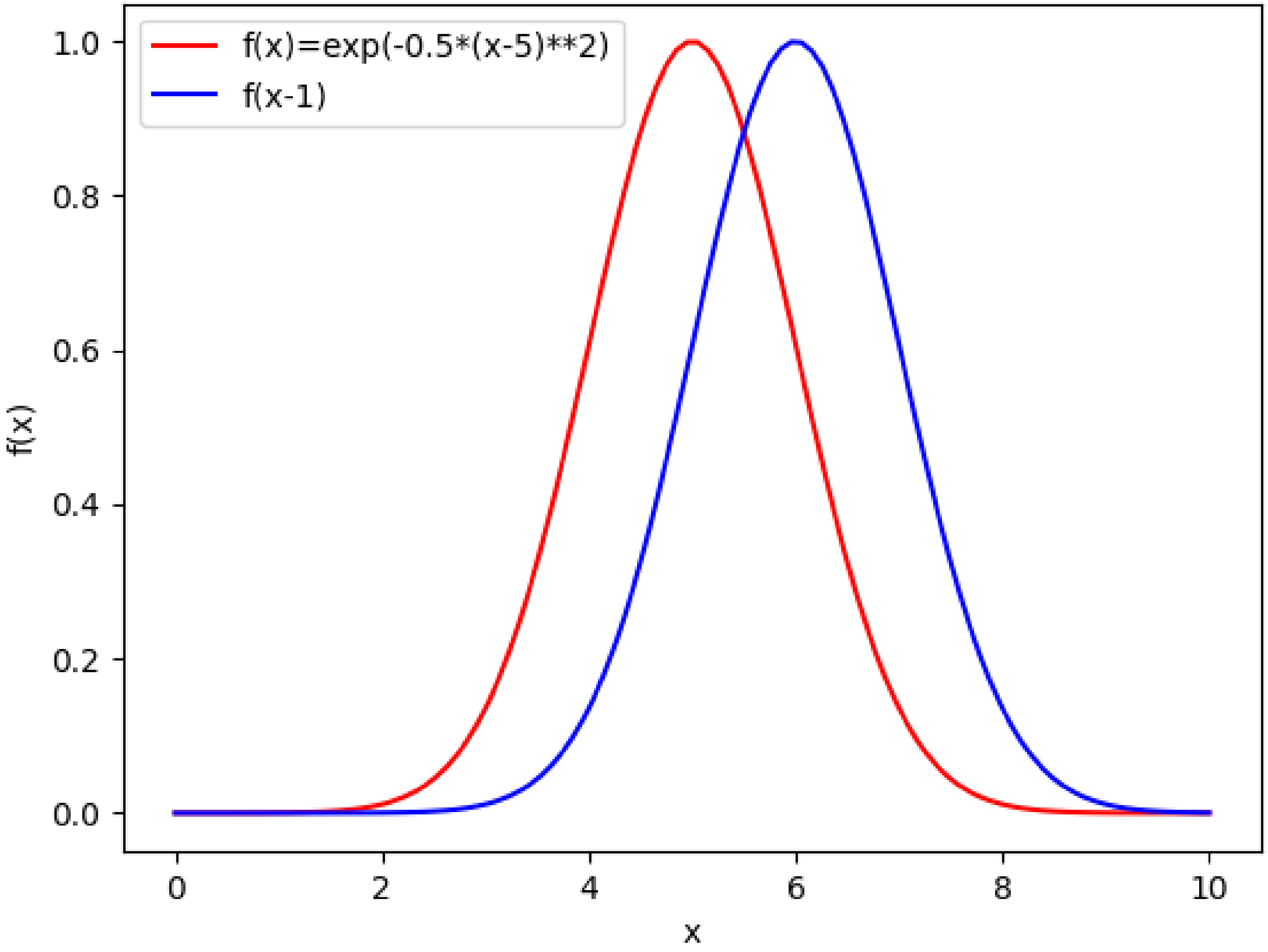
Programming with Python

Andri Hendriyana

2021

Case 1:
Write a program that can produce this plot



Python program

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

xx = np.linspace(0,10,100)
x0=5
x1 = (xx-x0)**2
y1 = np.exp(-0.5*x1)
# shifted Gaussian
x2 = (xx-x0-1)**2
y2 = np.exp(-0.5*x2)
plt.plot(xx,y1,'r-',xx,y2,'b-')
plt.legend(['f(x)=exp(-0.5*(x-5)**2)','f(x-1)'])
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

Copying an array

```
In [15]: from numpy import copy

In [16]: x = linspace(0, 2, 3)          # x becomes array([ 0.,  1.,  2.])

In [17]: y = copy(x)

In [18]: y
Out[18]: array([ 0.,  1.,  2.])

In [19]: y[0] = 10.0

In [20]: y
Out[20]: array([ 10.,   1.,   2.])  # ...changed

In [21]: x
Out[21]: array([ 0.,  1.,  2.])     # ...unchanged
```

Slicing an array

```
In [1]: from numpy import linspace

In [2]: x = linspace(11, 16, 6)

In [3]: x
Out[3]: array([  11.,   12.,   13.,   14.,   15.,   16.])

In [4]: y = x[1:5]

In [5]: y
Out[5]: array([ 12.,  13.,  14.,  15.])
```

```
In [6]: y[0] = -1.0

In [7]: y
Out[7]: array([-1.,  13.,  14.,  15.])       # ...changed

In [8]: x
Out[8]: array([  11.,  -1.,   13.,   14.,   15.,   16.])   # ...changed
```

Matrix-vector multiplication

```
In [1]: import numpy as np

In [2]: I = np.zeros((3, 3))      # create matrix (note parentheses!)

In [3]: I
Out[3]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

In [4]: type(I)                       # confirm that type is ndarray
Out[4]: numpy.ndarray

In [5]: I[0, 0] = 1.0;  I[1, 1] = 1.0;  I[2, 2] = 1.0  # identity matrix

In [6]: x = np.array([1.0, 2.0, 3.0])   # create vector

In [7]: y = np.dot(I, x)                # computes matrix-vector product

In [8]: y
Out[8]: array([ 1.,  2.,  3.])
```

I=np.eye(3)

Matrix-vector multiplication

```
In [1]: import numpy as np

In [2]: I = np.eye(3)          # create identity matrix

In [3]: I
Out[3]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

In [4]: type(I)                           # confirm that type is ndarray
Out[4]: numpy.ndarray

In [5]: I = np.matrix(I)                  # convert to matrix object

In [6]: type(I)                           # confirm that type is matrix
Out[6]: numpy.matrixlib.defmatrix.matrix

In [7]: x = np.array([1.0, 2.0, 3.0])     # create ndarray vector

In [8]: x = np.matrix(x)          # convert to matrix object (row vector)

In [9]: x = x.transpose()         # convert to column vector

In [10]: y = I*x                  # computes matrix-vector product

In [11]: y
Out[11]:
matrix([[ 1.],
        [ 2.],
        [ 3.]])
```

Summation

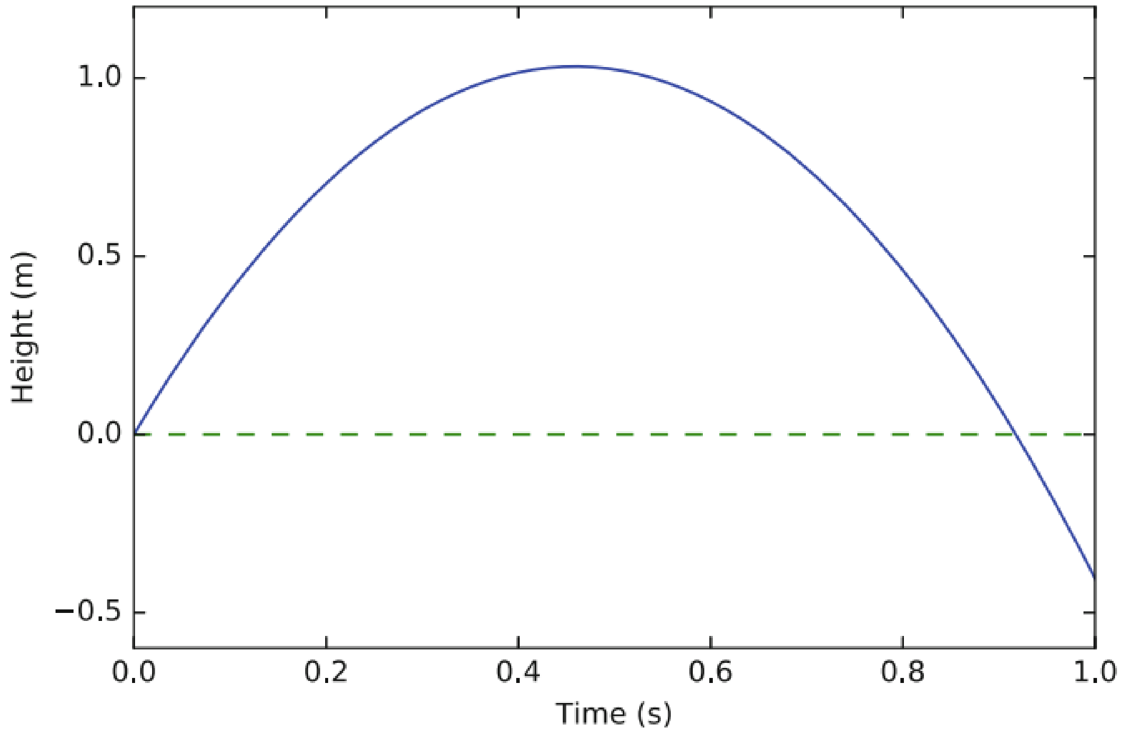
- Create a vector consisting of 100 random numbers (floats between 0 – 1)
- Compute mean of the vector:

$$\frac{1}{N}\sum_{i=1}^{N} x_i$$

```python
import numpy as np

nn=100
data=np.random.random(nn)
ndata=len(data)
sum=0
for i in range(0,ndata):
    sum=sum+data[i]
mean=sum/ndata
```

While loop




```
import numpy as np

v0 = 4.5                        # Initial velocity
g = 9.81                        # Acceleration of gravity
t = np.linspace(0, 1, 1000)    # 1000 points in time interval
y = v0*t - 0.5*g*t**2          # Generate all heights

# Find index where ball approximately has reached y=0
i = 0
while y[i] >= 0:
    i = i + 1

# Since y[i] is the height at time t[i], we do know the
# time as well when we have the index i...
print('Time of flight (in seconds): {:g}'.format(t[i]))

# We plot the path again just for comparison
import matplotlib.pyplot as plt
plt.plot(t, y)
plt.plot(t, 0*t, 'g--')
plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.show()
```

Branching (if, elif, else)

```
if condition_1:            # testing condition 1
    <code line 1>
    <code line 2>
    ...
elif condition_2:          # testing condition 2
    <code line 1>
    <code line 2>
    ...
elif condition_3:          # testing condition 3
    <code line 1>
    <code line 2>
    ...
else:
    <code line 1>
    <code line 2>
    ...
# First line after if-elif-else construction
```

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

v0 = 5                          # Initial velocity
g = 9.81                        # Acceleration of gravity
t = np.linspace(0, 1, 1000)     # 1000 points in time interval
y = v0*t - 0.5*g*t**2           # Generate all heights

# At this point, the array y with all the heights is ready,
# and we need to find the largest value within y.

largest_height = y[0]           # Starting value for search
for i in range(1, len(y), 1):
    if y[i] > largest_height:
        largest_height = y[i]

print('The largest height achieved was {:g} m'.format(largest_height))

# We might also like to plot the path again just to compare
plt.plot(t,y)
plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.show()
```

Function

```
def y(v0, t):
    g = 9.81                          # not in main
    return v0*t - 0.5*g*t**2          # not in main

v0 = 5

time = 0.6
print(y(v0, time))
time = 0.9
print(y(v0, time))
```

Function (two return values)

```
def xy(v0x, v0y, t):
    """Compute horizontal and vertical positions at time t"""
    g = 9.81                                        # acceleration of gravity
    return v0x*t, v0y*t - 0.5*g*t**2

v_init_x = 2.0                                      # initial velocity in x
v_init_y = 5.0                                      # initial velocity in y
time = 0.6                                          # chosen point in time

x, y = xy(v_init_x, v_init_y, time)
print('Horizontal position: {:g} , Vertical position: {:g}'.format(x, y))
```

Exercise 3.1: A `for` Loop with Errors

Assume some program has been written for the task of adding all integers $i = 1, 2, \ldots, 10$ and printing the final result:

```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sum = Sum + x
print 'sum: ', sum
```

a) Identify the errors in the program by just reading the code.
b) Write a new version of the program with errors corrected. Run this program and confirm that it gives the correct output.

Exercise 3.2: The `range` Function

Write a slightly different version of the program in Exercise 3.1. Now, the `range` function should be used in the `for` loop header, and only the even numbers from [2, 10] should be added. Also, the (only) statement within the loop should read `sum = sum + i`.

Exercise 3.3: A while Loop with Errors

Assume some program has been written for the task of adding all integers $i = 1, 2, \ldots, 10$:

```
some_number = 0
i = 1
while i < 11
    some_number += 1
print some_number
```

a) Identify the errors in the program by just reading the code.
b) Write a new version of the program with errors corrected. Run this program and confirm that it gives the correct output.

Exercise 3.7: Frequency of Random Numbers

Write a program that takes a positive integer N as input and then draws N random integers from the interval $[1, 6]$. In the program, count how many of the numbers, M, that equal 6 and print out the fraction M/N. Also, print all the random numbers to the screen so that you can check for yourself that the counting is correct. Run the program with a small value for `N` (e.g., `N = 10`) to confirm that it works as intended.

Hint Use `random.randint(1,6)` to draw a random integer between 1 and 6.

Exercise 3.10: Sort Array with Numbers

Write a script that uses the `uniform` function from the `random` module to generate an array of 6 random numbers between 0 and 10.

The program should then sort the array so that numbers appear in increasing order. Let the program make a formatted print of the array to screen both before and after sorting. Confirm that the array has been sorted correctly.

Exercise 3.11: Compute π

Up through history, great minds have developed different computational schemes for the number π. We will here consider two such schemes, one by Leibniz (1646–1716), and one by Euler (1707–1783).

The scheme by Leibniz may be written

$$\pi = 8\sum_{k=0}^{\infty} \frac{1}{(4k+1)(4k+3)},$$

while one form of the Euler scheme may appear as

$$\pi = \sqrt{6\sum_{k=1}^{\infty} \frac{1}{k^2}}.$$

If only the first N terms of each sum are used as an approximation to π, each modified scheme will have computed π with some error.

Write a program that takes N as input from the user, and plots the error development with both schemes as the number of iterations approaches N. Your program should also print out the final error achieved with both schemes, i.e. when the number of terms is `N`. Run the program with $N = 100$ and explain briefly what the graphs show.