Python Basics

In [1]:
```
# Printing a string
print("Hello, Python!")
print('hi')
```

```
Hello, Python!
hi
```

Variables

In [2]:
```
# defining a variable : In Python there is no need to mention the data type

var1 = 10      # An integer assignment
var2 = 3.146   # A floating point
var3 = "Hello" # A string

print(var1,' ',var2,' ',var3)
```

```
10   3.146   Hello
```

In [3]:
```
pi = 3.14
print ("Value of Pi is",pi)
```

```
Value of Pi is 3.14
```

Assignment

In [3]:
```
# Assigning same value to multiple variables

var1 = var2 = var3 = 1
print(var1,' ',var2,' ',var3)

# Assigning Different values to variable in a single expression

var1, var2, var3 = 1, 2.5, "john"
print(var1,' ',var2,' ',var3)

# Note: commas can be used for multi-assignments
```

```
1   1   1
1   2.5   john
```

Slicing

In [4]:
```
# String operations

str = 'Hello World!'  # A string

print(str)          # Prints complete string
print(str[0])       # Prints first character of the string
print(str[2:5])     # Prints characters starting from 3rd to 5th element
print(str[2:])      # Prints string starting from 3rd character
print(str[:2])
print(str * 2)      # Prints string two times
print(str + "TEST") # Prints concatenated string
```

```
Hello World!
H
llo
llo World!
He
Hello World!Hello World!
Hello World!TEST
```

Data types

In [5]:
```
# Python Lists
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  # A list
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  ) # A tuple. Tuples are immutable, i

print(list)            # Prints complete list
print(list[0])         # Prints first element of the list
print(tuple[1:3])        # Prints elements starting from 2nd till 3rd
```

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
(786, 2.23)
```

In [6]:
```
# Lists are ordered sets of objects, whereas dictionaries are unordered sets. Bu
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
print(tel)
print(tel['jack'])
del tel['sape']
tel['irv'] = 4127
print(tel)
print(tel.keys())
print(sorted(tel.keys()))
print(sorted(tel.values()))
print('guido' in tel)
print('jack' not in tel)
```

```
{'jack': 4098, 'sape': 4139, 'guido': 4127}
4098
{'jack': 4098, 'guido': 4127, 'irv': 4127}
dict_keys(['jack', 'guido', 'irv'])
['guido', 'irv', 'jack']
[4098, 4127, 4127]
True
False
```

Conditioning and looping

In [7]:
```
# Square of Even numbers

for i in range(0,10):

    if i%2 == 0:
        print("Square of ",i," is :",i*i)

    else:
        print(i,"is an odd number")
```

```
Square of  0  is : 0
1 is an odd number
Square of  2  is : 4
3 is an odd number
Square of  4  is : 16
5 is an odd number
Square of  6  is : 36
7 is an odd number
Square of  8  is : 64
9 is an odd number
```

Built-in Functions

In [8]:
```
print("Sum of array: ",sum([1,2,3,4]))
print("Length of array: ",len([1,2,3,4]))
print("Absolute value: ",abs(-1234))
print("Round value: ",round(1.2234))

import math as mt      # importing a package
print("Log value: ",mt.log(10))
```

```
Sum of array:  10
Length of array:  4
Absolute value:  1234
Round value:  1
Log value:  2.302585092994046
```

Functions

In [9]:
```
def area(length,width):
    return length*width
are = area(10,20)
print("Area of rectangle:",are)
```

```
Area of rectangle: 200
```

Broadcasting

- Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes

NumPy

- Numpy is the fundamental package for numerical computing with Python. It contains among other things:
- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

In [10]:
```
import numpy as np   # Importing Libraries

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

print("Matrix A\n", a)
print("Matrix B\n", b)

print("Regular matrix addition A+B\n", a + b)

print("Addition using Broadcasting A+5\n", a + 5)
```

```
Matrix A
 [0 1 2]
Matrix B
 [5 5 5]
Regular matrix addition A+B
 [5 6 7]
Addition using Broadcasting A+5
 [5 6 7]
```

Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1

In [11]:

```
# Lets go for a 2D matrix
c = np.array([[0, 1, 2],[3, 4, 5],[6, 7, 8]])
d = np.array([[1, 2, 3],[1, 2, 3],[1, 2, 3]])

e = np.array([1, 2, 3])

print("Matrix C\n", c)
print("Matrix D\n", d)
print("Matrix E\n", e)

print("Regular matrix addition C+D\n", c + d)

print("Addition using Broadcasting C+E\n", c + e)
```

```
Matrix C
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Matrix D
 [[1 2 3]
 [1 2 3]
 [1 2 3]]
Matrix E
 [1 2 3]
Regular matrix addition C+D
 [[ 1  3  5]
 [ 4  6  8]
 [ 7  9 11]]
Addition using Broadcasting C+E
 [[ 1  3  5]
 [ 4  6  8]
 [ 7  9 11]]
```

In [12]:

```
M = np.ones((3, 3))
print("Matrix M:\n",M)
```

```
Matrix M:
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

In [13]:

```
print("Dimension of M: ",M.shape)
print("Dimension of a: ",a.shape)
print("Addition using Broadcasting")
print(M + a)
# Broadcasting array with matrix
```

```
Dimension of M:  (3, 3)
Dimension of a:  (3,)
Addition using Broadcasting
[[1. 2. 3.]
 [1. 2. 3.]
 [1. 2. 3.]]
```

All in one program

In [14]:
```
# Importing libraries
import timeit

# Usage of builtin functions
start = timeit.default_timer()

# Defining a list
array_list = [10,11,15,19,21,32]
array_np_list = []

# Print the list
print("Original List",array_list,"\n")

# Defining a function
def prime(num):
    if num > 1:

        # check for factors
        # Iterating a range of numbers
        for i in range(2,num):
            if (num % i) == 0:

                # Appending data to list
                array_np_list.append(num)
                print(num,"is not a prime number (",i,"times",num//i,"is",num,")")

                # Terminating a loop run
                break
        else:
            print(num,"is a prime number")

# Iterating a list
for item in array_list:

    # Calling a function
    prime(item)

print("\nNon-prime List",array_np_list,"\n")

end = timeit.default_timer()

# Computing running time
print("Time Taken to run the program:",end - start, "seconds")
```

```
Original List [10, 11, 15, 19, 21, 32]

10 is not a prime number ( 2 times 5 is 10 )
11 is a prime number
15 is not a prime number ( 3 times 5 is 15 )
19 is a prime number
21 is not a prime number ( 3 times 7 is 21 )
32 is not a prime number ( 2 times 16 is 32 )

Non-prime List [10, 15, 21, 32]

Time Taken to run the program: 0.002303900000153464 seconds
```

Note:

- Python is a procedural Language
- Two versions of Python 2 vs 3
- No braces. i.e. indentation
- No need to explicitly mention data type

Unvectorized vs Vectorized Implementations

In [17]:
```
# Importing libraries
import numpy as np

# Defining matrices
mat_a = [[6, 7, 8],[5, 4, 5],[1, 1, 1]]
mat_b = [[1, 2, 3],[1, 2, 3],[1, 2, 3]]

# Getting a row from matrix
def get_row(matrix, row):
    return matrix[row]

# Getting a coloumn from matrix
def get_column(matrix, column_number):
    column = []

    for i in range(len(matrix)):
        column.append(matrix[i][column_number])

    return column

# Multiply a row with coloumn
def unv_dot_product(vector_one, vector_two):
    total = 0

    if len(vector_one) != len(vector_two):
        return total

    for i in range(len(vector_one)):
        product = vector_one[i] * vector_two[i]
        total += product

    return total

# Multiply two matrixes
def matrix_multiplication(matrix_one, matrix_two):
    m_rows = len(matrix_one)
    p_columns = len(matrix_two[0])
    result = []

    for i in range(m_rows):
        row_result = []

        for j in range(p_columns):
            row = get_row(matrix_one, i)
            column = get_column(matrix_two, j)
            product = unv_dot_product(row, column)

            row_result.append(product)
        result.append(row_result)

    return result

print("Matrix A: ", mat_a,"\n")
print("Matrix B: ", mat_b,"\n")

print("Unvectorized Matrix Multiplication\n",matrix_multiplication(mat_a,mat_b),
```

```
Matrix A:  [[6, 7, 8], [5, 4, 5], [1, 1, 1]]

Matrix B:  [[1, 2, 3], [1, 2, 3], [1, 2, 3]]

Unvectorized Matrix Multiplication
 [[21, 42, 63], [14, 28, 42], [3, 6, 9]]
```

In [18]:

```
# Vectorized Implementation
npm_a = np.array(mat_a)
npm_b = np.array(mat_b)

print("Vectorized Matrix Multiplication\n",npm_a.dot(npm_b),"\n")
# A.dot(B) is a numpy built-in function for dot product
```

```
Vectorized Matrix Multiplication
 [[21 42 63]
 [14 28 42]
 [ 3  6  9]]
```

Tip:

- Vectorization reduces number of lines of code
- Always prefer libraries and avoid coding from scratch

Essential Python Packages: Numpy, Pandas, Matplotlib

In [19]:

```
# Load library
import numpy as np
```

In [20]:

```
# Create row vector
vector = np.array([1, 2, 3, 4, 5, 6])
print("Vector:",vector)

# Select second element
print("Element 2 in Vector is",vector[1])
```

```
Vector: [1 2 3 4 5 6]
Element 2 in Vector is 2
```

In [21]:
```
# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

print("Matrix\n",matrix)

# Select second row
print("Second row of Matrix\n",matrix[1,:])
print("Third coloumn of Matrix\n",matrix[:,2])
```

```
Matrix
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Second row of Matrix
 [4 5 6]
Third coloumn of Matrix
 [3 6 9]
```

In [22]:
```
# Create Tensor
# multi dimensional array
tensor = np.array([ [[[1, 1], [1, 1]], [[2, 2], [2, 2]]],
                    [[[3, 3], [3, 3]], [[4, 4], [4, 4]]] ])
print("Tensor\n",tensor)
print(tensor.shape)
```

```
Tensor
 [[[[1 1]
   [1 1]]

  [[2 2]
   [2 2]]]


 [[[3 3]
   [3 3]]

  [[4 4]
   [4 4]]]]
(2, 2, 2, 2)
```

Matrix properties

In [24]:
```
# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

print("Matrix Shape:",matrix.shape)
print("Number of elements:",matrix.size)
print("Number of dimentions:",matrix.ndim)
print("Average of matrix:",np.mean(matrix))
print("Maximum number:",np.max(matrix))
print("Coloumn with minimum numbers:",np.min(matrix, axis=1))
print("Diagnol of matrix:",matrix.diagonal())
print("Determinant of matrix:",np.linalg.det(matrix))
```

```
Matrix Shape: (3, 3)
Number of elements: 9
Number of dimentions: 2
Average of matrix: 5.0
Maximum number: 9
Coloumn with minimum numbers: [1 4 7]
Diagnol of matrix: [1 5 9]
Determinant of matrix: -9.51619735392994e-16
```

Matrix Operations

In [25]:
```
print("Flattened Matrix\n",matrix.flatten())
print("Reshaping Matrix\n",matrix.reshape(9,1))
print("Transposed Matrix\n",matrix.T)
```

```
Flattened Matrix
 [1 2 3 4 5 6 7 8 9]
Reshaping Matrix
 [[1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]
 [9]]
Transposed Matrix
 [[1 4 7]
 [2 5 8]
 [3 6 9]]
```

In [26]:
```
# Create matrix
matrix_a = np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 2]])

# Create matrix
matrix_b = np.array([[1, 3, 1],
                     [1, 3, 1],
                     [1, 3, 8]])

print("Matrix Addition\n",np.add(matrix_a, matrix_b))
print("Scalar Multiplication\n",np.multiply(matrix_a, matrix_b))
print("Matrix Multiplication\n",np.dot(matrix_a, matrix_b)) # vector or inner pro
```

```
Matrix Addition
 [[ 2  4  2]
 [ 2  4  2]
 [ 2  4 10]]
Scalar Multiplication
 [[ 1  3  1]
 [ 1  3  1]
 [ 1  3 16]]
Matrix Multiplication
 [[ 3  9 10]
 [ 3  9 10]
 [ 4 12 18]]
```

Pandas

In [27]:
```
import pandas as pd
```

In [29]:
```
df=pd.read_csv("Income.csv")
print("Data\n")
df
```

```
Data
```

Out[29]:

	GEOID	State	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	04000US01	Alabama	37150	37952	42212	44476	39980	40933	42590	43464	41381
1	04000US02	Alaska	55891	56418	62993	63989	61604	57848	57431	63648	61137
2	04000US04	Arizona	45245	46657	47215	46914	45739	46896	48621	47044	50602
3	04000US05	Arkansas	36658	37057	40795	39586	36538	38587	41302	39018	39919
4	04000US06	California	51755	55319	55734	57014	56134	54283	53367	57020	57528
5	04000US07	Chicago	-999	-999	-999	-999	-999	-999	-999	-999	-999

In [30]:
```
print("Top Elements\n")
df.head(3)
```

```
Top Elements
```

Out[30]:

	GEOID	State	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	04000US01	Alabama	37150	37952	42212	44476	39980	40933	42590	43464	41381
1	04000US02	Alaska	55891	56418	62993	63989	61604	57848	57431	63648	61137
2	04000US04	Arizona	45245	46657	47215	46914	45739	46896	48621	47044	50602

In [31]:
```
print("Bottom Elements\n")
df.tail(3)
```

```
Bottom Elements
```

Out[31]:

	GEOID	State	2005	2006	2007	2008	2009	2010	2011	2012	2013
3	04000US05	Arkansas	36658	37057	40795	39586	36538	38587	41302	39018	39919
4	04000US06	California	51755	55319	55734	57014	56134	54283	53367	57020	57528
5	04000US07	Chicago	-999	-999	-999	-999	-999	-999	-999	-999	-999

In [32]:
```
print("Specific Coloumn\n")
df['State'].head(3)
```

```
Specific Coloumn
```

Out[32]:
```
0    Alabama
1     Alaska
2    Arizona
Name: State, dtype: object
```

In [34]:
```
print("Replace negative numbers with NaN\n")
df.replace(-999,np.nan)
```

```
Replace negative numbers with NaN
```

Out[34]:

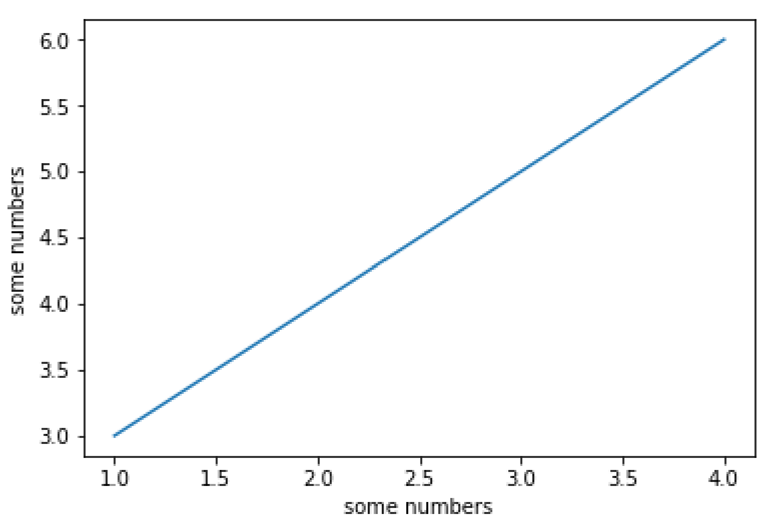
	GEOID	State	2005	2006	2007	2008	2009	2010	2011	2012	
0	04000US01	Alabama	37150.0	37952.0	42212.0	44476.0	39980.0	40933.0	42590.0	43464.0	4
1	04000US02	Alaska	55891.0	56418.0	62993.0	63989.0	61604.0	57848.0	57431.0	63648.0	6
2	04000US04	Arizona	45245.0	46657.0	47215.0	46914.0	45739.0	46896.0	48621.0	47044.0	5
3	04000US05	Arkansas	36658.0	37057.0	40795.0	39586.0	36538.0	38587.0	41302.0	39018.0	3
4	04000US06	California	51755.0	55319.0	55734.0	57014.0	56134.0	54283.0	53367.0	57020.0	5
5	04000US07	Chicago	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

Matplotlib

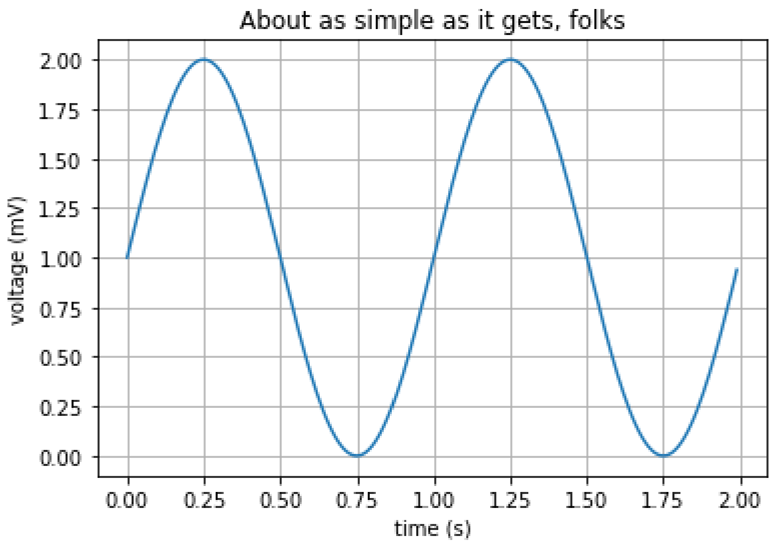
In [35]:
```
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
```

Line Plot

In [36]:
```
# Line plot
plt.plot([1,2,3,4],[3,4,5,6])
plt.xlabel('some numbers')
plt.ylabel('some numbers')
plt.show()
```

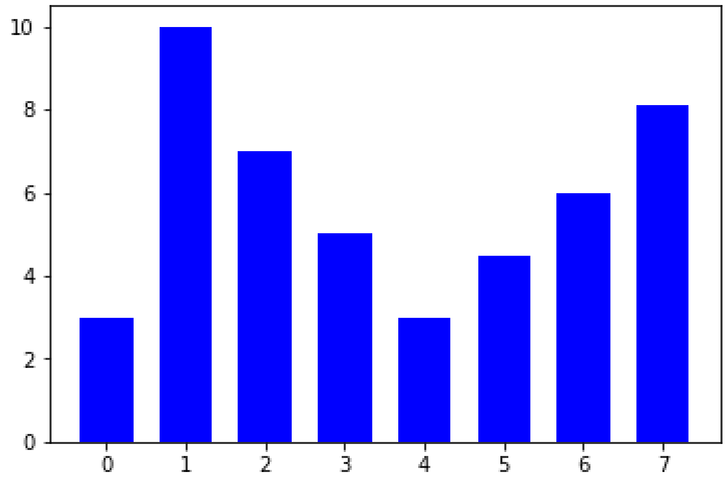


In [37]:
```
### Adding elements to line plots
t = np.arange(0.0, 2.0, 0.01) # Generate equally space numbers between 0 and 2
s = 1 + np.sin(2*np.pi*t)  # Apply sin function to the random numbers
plt.plot(t, s)

plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('About as simple as it gets, folks')
plt.grid(True)
plt.savefig("test.png") # Save a plot. Check the directory
plt.show()
```



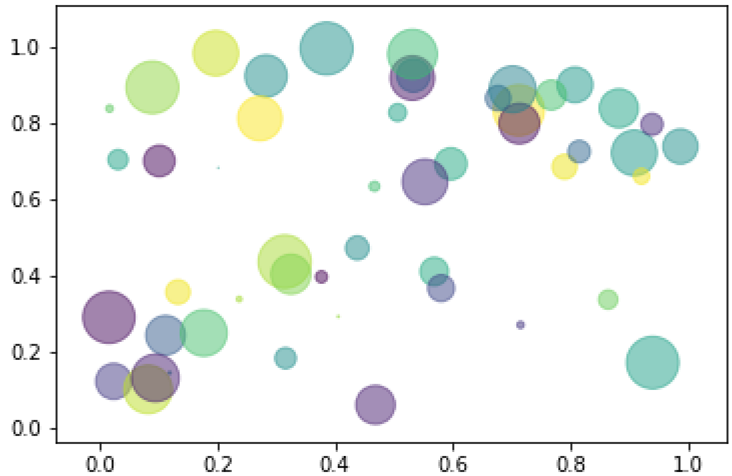
Bar Plot

In [38]:
```
y = [3, 10, 7, 5, 3, 4.5, 6, 8.1]
x = range(len(y))
width = 1/1.5
plt.bar(x, y, width, color="blue")
plt.show()
```



Scatter Plot

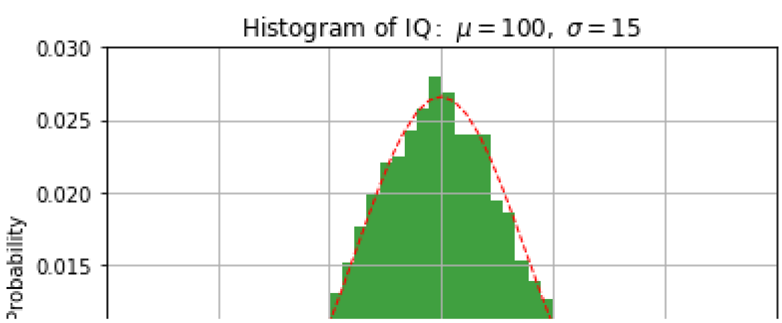
In [41]:
```
N = 50
# Generate random numbers
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radii

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



Histogram

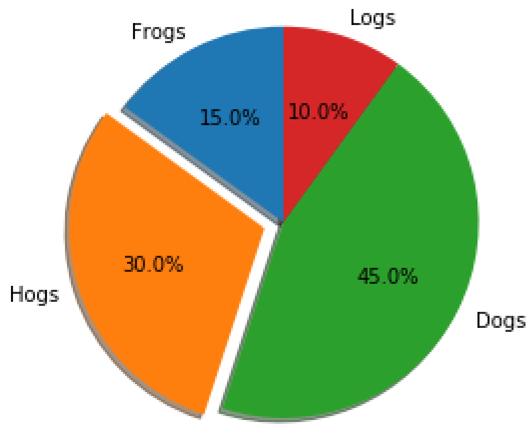
In [42]:
```
mu, sigma = 100, 15
x = mu + sigma*np.random.randn(10000) # Generate random values with some distribu

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='green', alpha=0.75)

# add a 'best fit' line
y = mlab.normpdf( bins, mu, sigma)
l = plt.plot(bins, y, 'r--', linewidth=1)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'$\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)

plt.show()
```

```
C:\Users\Arihant\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py:6521: M
atplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.
1. Use 'density' instead.
  alternative="'density'", removal="3.1")
C:\Users\Arihant\Anaconda3\lib\site-packages\ipykernel_launcher.py:8: Matplot
libDeprecationWarning: scipy.stats.norm.pdf
```



Pie Chart

In [43]:
```
# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()
```



In [ ]: