Statistics Gathering and Reporting

To make the data really useful, we need to analyze it, derive some conclusions, and present the results to the end users. In this chapter, we’ll create a simple web-based application that performs statistical analysis on the data and also generates some reports.

# Application Requirements and Design

The statistical representation system should be fairly simple and easy to use. The following is the basic functionality it needs to provide:

* The system should provide a list of all available hosts that are being monitored.
* For each available host, there should be a list of all probes (a *probe* is a simple check script running on the remote server) available for that host.
* The probes should be grouped into two criteria: probe name and data timescale.
* The data should be presented on different timescales, such as readings obtained in the last 24 hours, last 7 days, and last 30 days.
* The system should report on the number of times the set thresholds have been reached. This information can be expressed as a percentage from the number of all requests that have been made in a timescale period.
* The system should provide basic statistical analysis of the data, such as the average values, data trending, and so on.

The system will be a script that reads the data from the monitoring database and then generates the static HTML pages along with the required data graph images. This script can be run on a regular basis using system scheduling tools such as cron.

The graphing and statistical analysis will be performed by using the NumPy and matplotlib libraries.

## Using NumPy Library

The most popular statistical libraries for the Python programming language are NumPy (formerly known as Numeric), which provides high-level mathematical functions, and SciPy, which provides more than 15 different scientific modules (with various scientific algorithms for optimizations, linear algebra, signal processing and analysis, and statistical analysis).

## NumPy Examples

### Working with Arrays

>>> import numpy

>>> array\_py = [1, 4, 5, 7, 9]

>>> array\_np = numpy.array([1, 4, 5, 7, 9])

>>> type(array\_py)

<type 'list'>

>>> type(array\_np)

<type 'numpy.ndarray'>

>>> array\_np.append(2)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'numpy.ndarray' object has no attribute 'append'

>>>

>>> a1 = numpy.array([1, 4, 5, 7, 9])

>>> a1.mean() # calculate a mean value of the array

5.2000000000000002

>>> a1.std() # calculate the standard deviation

2.7129319932501073

>>> a1.var() # calculate the variance

7.3599999999999994

>>>

>>> a1 = numpy.array([[1, 2, 3], [4, 5, 6]])

>>> a1[1, 1] # second element of the second row

5

>>>

>>> a = np.arange(16)

>>> a

array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])

>>> a.reshape(2, 8)

array([[ 0, 1, 2, 3, 4, 5, 6, 7],

[ 8, 9, 10, 11, 12, 13, 14, 15]])

>>> a.reshape(4, 4)

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11],

[12, 13, 14, 15]])

>>> a.reshape(4, 5)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: total size of new array must be unchanged

>>>

>>> numpy.append(a1, [7, 8, 9])

array([1, 2, 3, 4, 5, 6, 7, 8, 9])

>>>

This is clearly wrong. We wanted a third row to appear, but instead we got a single-dimension list with the additional entries appended to it. What’s happened is that NumPy flattened the list and appended the new values to it, because that’s what the append() operation does—appends new elements, and not sublists.

Fortunately, NumPy has two other functions that allow appending not only new rows but also new columns to the lists. The vstack() function appends a new row, and the hstack() function appends a new column:

>>> numpy.vstack((a1, [7, 8, 9]))

array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9]])

>>> numpy.hstack((a1, [[7], [8]]))

array([[1, 2, 3, 7],

[4, 5, 6, 8]])

>>>

## Calculating the Mean and Standard Deviation

>>> a = np.random.randn(1000)

>>> h, b = np.histogram(a, bins=8, normed=True)

>>> h

array([ 0.00238784, 0.02268444, 0.12416748, 0.30444912, 0.37966596,

0.26146807, 0.08834994, 0.01074526])

>>> b

array([-3.63950476, -2.80192639, -1.96434802, -1.12676964, -0.28919127,

0.5483871 , 1.38596547, 2.22354385, 3.06112222])

>>>

The function numpy.random.randn(<*count*>) is used to generate a normal distribution set with the mean of 0 and the standard deviation of 1. Also keep in mind that randn() returns samples from a standard normal distribution, so the result is not guaranteed to be the same between runs.

## Finding the Trend Line of a Dataset

>>> x = np.arange(100)

>>> y = np.random.normal(4., 0.9, 100)

>>> for i in range(100):

... y[i] = y[i] + i/40

>>> a, b = np.polyfit(x, y, 1)

## Reading and Writing Data to Files

>>> a = np.arange(16).reshape(4,4)

>>> a

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11],

[12, 13, 14, 15]])

>>> np.savetxt('data.txt', a, fmt="%G", delimiter=',')

>>> b = np.loadtxt('data.txt', delimiter=',')

>>> b

array([[ 0., 1., 2., 3.],

[ 4., 5., 6., 7.],

[ 8., 9., 10., 11.],

[ 12., 13., 14., 15.]])

>>>

## Representing Data with matplotlib

### Plotting Graphs

import matplotlib.pyplot as plt

import numpy as np

fig = plt.figure()

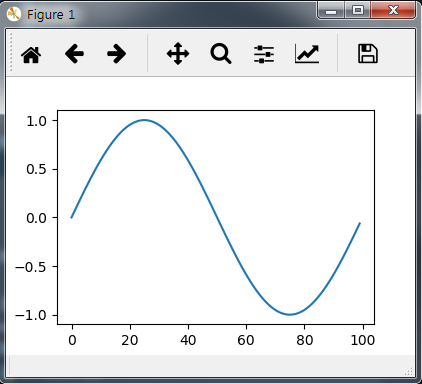
ax = fig.add\_subplot(1, 1, 1)

x = np.arange(100)

y = np.sin(2 \* np.pi \* x / 100)

ax.plot(y)

plt.show()

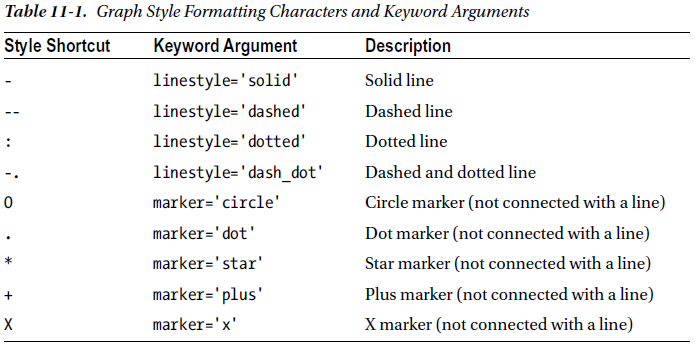


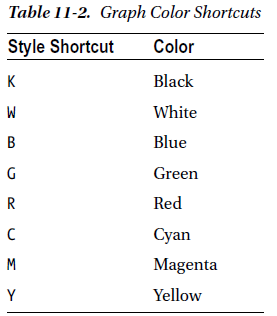
### Changing the Appearance of the Plot Primitives

x = np.arange(100)

y = np.sin(2 \* np.pi \* x / 100)

ax.plot(x, y, 'r:')





### Drawing Bars and Using Multiple Axes

import matplotlib.pyplot as plt

import numpy as np

fig = plt.figure()

ax = fig.add\_subplot(2, 1, 1, polar=True)

x = np.arange(25)

y = np.sin(2 \* np.pi \* x / 25)

ax.bar(x \* np.pi \* 2/ 25, abs(y), width=0.3, alpha=0.3)

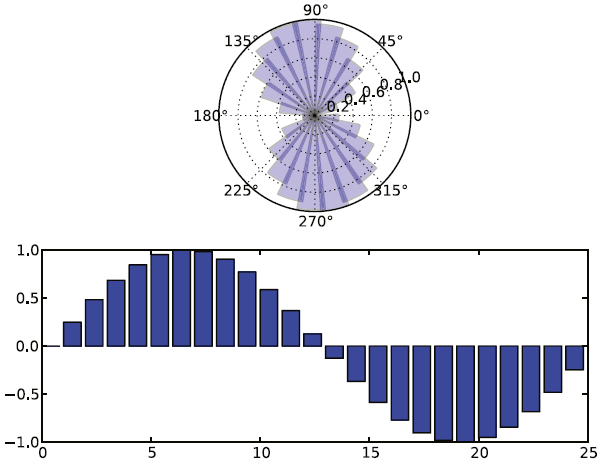
ax2 = fig.add\_subplot(2, 1, 2)

x2 = np.arange(25)

y2 = np.sin(2 \* np.pi \* x2 / 25)

ax2.bar(x2, y2)

plt.show()



The polar keyword argument indicates whether the axis will have the Cartesian coordinate system or polar

coordinates. If you set the coordinate system to polar, keep in mind that the full circle range is from 0 to 2\*π.

### Working with Text Strings

import matplotlib.pyplot as plt

import numpy as np

fig = plt.figure()

ax = fig.add\_subplot(1, 1, 1,

title="Fourth degree polynomial",

xlabel='X Axis',

ylabel='Y Axis')

x = np.linspace(-5., 3)

y = 0.2 \* x\*\*4 + 0.5 \* x\*\*3 - 2.5 \* x\*\*2 - 1.2 \* x - 0.6

ax.plot(x, y)

ax.grid(True)

ax.text(-4.5, 6, r'$y = 0.2 x^4 + 0.4 x^3 - 2.5 x^2 - 1.2 x - 0.6$', fontsize=14)

ax.annotate('Turning point',

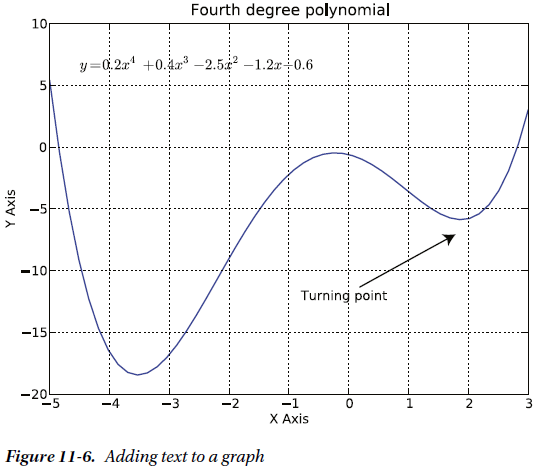
xy=(1.8, -7),

xytext=(-0.8, -12.6),

arrowprops=dict(arrowstyle="->",)

)

plt.show()



### Saving Plots to a File

import matplotlib

matplotlib.use('Agg')

import matplotlib.pyplot as plt

import numpy as np

fig = plt.figure()

ax = fig.add\_subplot(1, 1, 1)

x = np.arange(100)

y = np.sin(2 \* np.pi \* x / 100)

ax.plot(y)

plt.savefig('sin-wave.png')

plt.savefig('sin-wave.pdf')

## Graphing Statistical Data