CHAPTER

33 SQL Server Machine Learning Services: Python Support

In This Chapter

- Python: An Introduction
- Data Visualization with Python
- Predictive Modeling with Python in SQL Server

Microsoft introduced support for Python with its Machine Learning Services in SQL Server 2017. The integration of Python in SQL Server allows developers to access the extensive Python libraries that are available in the open source community. Microsoft has added two SQL Server–specific features. First, the **sp_execute_external_script** stored procedure (introduced in Chapter 32) allows you to integrate Python scripts in SQL Server. Second, Python resources can be throttled using Resource Governor. That way, database administrators can make optimal decisions concerning Python processing workloads. (For the description of Resource Governor, see Chapter 20.)

This chapter covers the integration of the Python language into the Database Engine. The first major section introduces the Python language, explains generally how the language is embedded in the SQL Server system, and defines the concept of a data frame in relation to Python. The second major section shows you how to visualize data using data frames. The chapter wraps up with an exploration of predictive modeling with R and presents an example of solving linear regression problems with R.

Python: An Introduction

Python is a programming language that can be used for many purposes, including machine learning. The most important advantages of the language are

- Simple syntax
- High readability
- A large collection of libraries

Python aims to be simple in the design of its syntax, encapsulated in the slogan "There should be one *obvious* way to do it." Because of its simplicity, Python is known as a beginner's level programming language.

If you ask Python programmers what they like most about Python, they often cite its high readability. One reason for the high readability of Python code is its complete set of code style guidelines. Because of its simplicity, Python has attracted many developers to create new libraries for machine learning. Also, because of the existence of these libraries, Python is becoming very popular among machine learning experts.

NOTE To be able to use the **sp_execute_external_script** system stored procedure, you need to enable it first by running the following batch:

```
EXEC sp_configure 'external scripts enabled', 1
GO
RECONFIGURE WITH OVERRIDE
GO
```

Getting Started with Python

If you are new to the Python programming language, it is important to know that Python code blocks are specified by their indentation. In other words, indentation of lines of code is a *requirement* and not a matter of style as in many other programming languages. Example 33.1 is a very simple example that shows how you can use the **sp_execute_external_script** system stored procedure to display values of the **project** table using Python. (For the syntax and description of this system procedure, see the section "Getting Started with R in SQL Server" toward the beginning of Chapter 32.)

Example 33.1

```
USE sample;
DECLARE @pscript NVARCHAR(MAX);
SET @pscript = N'
df = InputDataSet
OutputDataSet = df';
DECLARE @select NVARCHAR(MAX);
SET @select = N'
  SELECT project_no, budget
  FROM project';
EXEC sp_execute_external_script
  @language = N'Python',
  @script = @pscript,
  @input_data_1 = @select;
GO
```

As you can see from Example 33.1, the **sp_execute_external_script** system stored procedure has, among others, the following three parameters: **@language**, **@script**, and **@input_data_1**. The **@language** parameter specifies the name of the language integrated in

SQL Server. The value "Python" specifies that Python is used for implementation. The **@script** parameter defines the external language script as a string. You can specify it using a literal or a variable. The **@input_data_1** parameter defines the input data used by the external script in the form of a Transact-SQL query. In other words, the query generates a result set that is used as input data. The data type of this parameter is NVARCHAR(max), meaning that the query is passed as a string.

Two important parameters are related to the **@script** variable: **@input_data_1_name** and **@output_data_1_name**. The first parameter specifies the name of the variable used to represent the input dataset (i.e., the dataset defined by **@input_data_1**). You can omit the use of **@input_data_1_name**, in which case the default value **InputDataSet** is used. The second parameter, **@output_data_1_name**, specifies the name of the variable used to represent the output dataset (i.e., the result of the script). If you do not assign a value to **@output_data_1_name**, the default name **OutputDataSet** is used.

As you can see from the SELECT statement in Example 33.1, the input data contains all values of the columns **project_no** and **budget** of the **project** table. These values are assigned to the variable called **df**. After that, the input data is passed to the output data (**OutputDataSet = df**) without any modification. (As you can see from the following result, neither column has an explicit name. Example 33.2 shows how you can assign particular names to the columns in a result set.)

The result is

(No Column Name)	(No Column Name)
p1	120000
p2	95000
p3	186500

I present the script in Example 33.1 only to demonstrate how to run the **sp_execute_external_script** system stored procedure and to explain the meaning of some parameters of the procedure. In other words, Example 33.1 simply assigns query results to the Python script, which returns the same results as that of the SELECT statement. This is something you can do without using a Python script, but being able to assign your query results to the Python script means you can then use the analytical power built into Python to apply it to that data.

The most important advantage of the integration of Python in SQL Server is the capability to use data stored persistently in the database system, manipulate the data using Python functions, and return it to the Database Engine. Example 33.2 shows this.

Example 33.2

```
USE sample;
DECLARE @pscript NVARCHAR(MAX);
SET @pscript = N'
df1 = InputDataSet
OutputDataSet = round(df1/7, 2)';
DECLARE @select NVARCHAR(MAX);
```

```
SET @select = N'
  SELECT budget AS Balanced_budget
  FROM project';
 EXEC sp_execute_external_script
  @language = N'Python',
  @script = @pscript,
  @input_data_1 = @select
  WITH RESULT SETS ((Balanced_budget FLOAT));
GO
```

First, note the following two lines of code from Example 33.2:

```
df1 = InputDataSet
OutputDataSet = round(df1/7, 2)';
```

The SELECT statement assigned to **@input_data_1** using the **@select** variable generates the input dataset. The dataset contains values of the **budget** column of the **project** table. The second line applies a Python function to the **df1** variable, to which the input data is assigned. Precisely, each input value is then divided by 7 and rounded using the **round** function. After that, the modified values of the **budget** column are assigned to the result set, which is sent to the database system.

The last line of the code in Example 33.2 uses the WITH RESULT SETS clause to name a column of the result set. (By default, a result set returned by a Python script is displayed as a table with unnamed columns.)

The result is

Balanced_budget
17142.86
13571.43
26642.86

Python Data Frames

A data frame is used to store input data sent from the Database Engine, pass the data to Python, modify it using Python functions, and send it back to the database system. The structure of a data frame is "two-dimensional," meaning that each frame is made up of rows and columns.

In case of Python, input data is passed to a script and converted to a **DataFrame** object. Also, the data returned by a Python script is passed to the output variable as a **DataFrame** object. All **DataFrame** objects belong to the class with the same name, which is a part of the **pandas** library. (This library provides data structures designed to allow you to work with "relational" data—data provided in the table form.)

Example 33.3 shows the use of data frames.

Example 33.3

```
USE AdventureWorks;
DECLARE @pscript NVARCHAR(MAX);
SET @pscript = N'
df1 = InputDataSet
OutputDataSet = df1.groupby("Units", as_index=False).max()';
DECLARE @select NVARCHAR(MAX);
SET @select = N'
SELECT  v.UnitMeasureCode AS Units
  FROM Purchasing.PurchaseOrderHeader h
   INNER JOIN Purchasing.PurchaseOrderDetail d
        ON h.PurchaseOrderID = d.PurchaseOrderID
   INNER JOIN Purchasing.ProductVendor v ON d.ProductID=v.ProductID';
 EXEC sp_execute_external_script
   @language = N'Python',
   @script = @pscript,
   @input_data_1 = @select;
```

The input data of the script in Example 33.3 is generated from the tables of the **AdventureWorks** sample database. The SELECT statement displays the total purchase per unit measure code. To retrieve data, we join three tables: **PurchaseOrderHeader**, **PurchaseOrderDetail**, and **ProductVendor**. The single column in the SELECT list has the alias **Units**. The results of this query are all distinct unit measure codes.

The line of code following the assignment of the input data to the **df1** variable needs some explanation:

```
OutputDataSet = df1.groupby("Units", as_index=False).max()';
```

The **groupby** function is applied to the values stored in the **df1** variable. This function allows you to group records using distinct values. The first argument of the function specifies the object whose values will be grouped. In this case, the grouping is done on values of the **UnitMeasureCode** column of the **ProductVendor** table. (The alias of this column is **Units**.) To get one value from each group, the **max** function is applied to the result. (While all values in each group are identical, the function returns just one of them.)

The second argument of the **groupby** function is **as_index**. When using this function, the **as_index** parameter can be set to either True or False, depending on whether you want the grouping column to be the index of the output or not, respectively. When **as_index=False** is used, the key(s) you use in the **groupby** function is generated by the system and added as an additional column of the output data. (Taking a look at the following result, you can see that, besides the values of the **UnitMeasureCode** column, there is an additional column with index values: 1, 2, 3,…)

NOTE The use of **as_index=True** is discussed shortly in relation to Example 33.5.

The result is

	(NoColumnName)
1	CAN
2	CS
3	CTN
4	DZ
5	EA
6	GAL
7	PAK

As you already know, data frames provide data in table form. Example 33.3 uses only column data of the data frame for calculation. Example 33.4 uses the same input data as Example 33.3 but displays data in table form, with both row values and column values of the data frame.

Example 33.4

```
USE AdventureWorks;
DECLARE @pscript NVARCHAR(MAX);
SET @pscript = N'
df1 = InputDataSet
df2 = df1.groupby("Units", as_index=False).sum()
OutputDataSet = df2';
DECLARE @select NVARCHAR(MAX);
SET @select = N'
SELECT CAST(h.subtotal AS FLOAT)  AS Total, v.UnitMeasureCode AS Units
  FROM Purchasing.PurchaseOrderHeader h
    INNER JOIN Purchasing.PurchaseOrderDetail d
                ON h.PurchaseOrderID = d.PurchaseOrderID
    INNER JOIN Purchasing.ProductVendor v ON d.ProductID=v.ProductID';
 EXEC sp_execute_external_script
   @language = N'Python',
   @script = @pscript,
   @input_data_1 = @select
    WITH RESULT SETS((UnitCodes NVARCHAR(50), TotalSales MONEY));
```

First, the SELECT list of the query, assigned to the **@select** variable, contains two columns with the aliases **Total** and **Units**, respectively. In the following code,

```
df1 = InputDataSet
df2 = df1.groupby("Units", as_index=False).sum()
OutputDataSet = df2
```

the first line passes the input dataset to the **df1** data frame. In the second line, the **groupby** function is applied to the values of the unit measure codes and, for each distinct unit, the sum of all subtotals (the values of the **subtotal** column of the **PurchaseOrderHeader** table) is calculated. Finally, the content of the **df2** data frame is passed to the output data.

The result is

	UnitCodes	TotalSales
1	CAN	9609758.613
2	CS	584962.963
3	CTN	4408404.588
4	DZ	958018.1625
5	EA	267092933.073
6	GAL	7526088.36
7	PAK	206096.94

One more feature of the query in Example 33.4 requires further explanation. Note that the SELECT list of the query uses the CAST operator to select values from the **subtotal** column of the **PurchaseOrderHeader** table and to convert them into the values with the data type FLOAT. This is necessary, because if you do not use casting, the system displays an error with a message similar to the following: "Unsupported input data type in column 'total'. Supported types: bit, tinyint, smallint, int, bigint, uniqueidentifier, real, float, char, varchar, nchar, nvarchar, varbinary."

The reason for this error message is that the set of standard data types in Python is significantly smaller than the corresponding set for Transact-SQL. For this reason, the Python system cannot convert each Transact-SQL data type into an appropriate Python type. One example of such a data type is MONEY, which is the data type of the **subtotal** column. In such cases, you have to use the CAST operator and to explicitly convert the column's data type into a data type that is supported by Python.

Data Visualization with Python

As mentioned earlier in the chapter, one of the advantages of Python is that it supports a large number of open source libraries, which you can use for many different purposes. One of these libraries, **matplotlib**, provides extensive support to display data in different graphic forms, such as histograms, bar charts, and pie charts.

As with the R language, there are two general ways to integrate Python graphics with SQL Server:

- Output a dataset by using the **sp_execute_external_script** system stored procedure and apply one of many Python packages that support data visualization. In other words, you do data visualization *inside* Python.
- Integrate Python into a SQL Server tool, such as Power BI, and use its functions to visualize data.

NOTE Data visualization using Power BI in relation to Python is very similar to the same process using the R language. Please read the section "Integrate R in Power BI Desktop" in Chapter 32 for full details.

Example 33.5 uses the result set generated in Example 33.4 and displays it graphically using a bar chart.

NOTE When executing Example 32.5, you might get an error similar to this:
Msg 39004, Level 16, State 20, Line 448 A 'R' script error occurred during execution of 'sp_execute_external_script' with HRESULT.

This error is related to the SQL Server Launchpad service. This service is used to start Advanced Analytics Extensions processes, which are necessary for integration of the R system and Python with the Database Engine. The program in Example 33.5 writes the file in the C:\temp directory, which is by default not accessible by the Launchpad service. You can either grant access to this directory or use another directory; for instance, C:\PythonScripts. To grant access to a directory, right-click that directory, select Grant Access To, and click Add Everyone. This also applies later to Example 33.8.

Example 33.5

```
Use AdventureWorks;
DECLARE @pscript NVARCHAR(MAX);
SET @pscript = N'
import matplotlib
matplotlib.use("PDF")
import matplotlib.pyplot as plt
df1 = InputDataSet
df2 = df1.groupby("Units", as_index=True).sum()
pt = df2.plot.barh()
# Set title
pt.set_title (label="Total Purchases per Unit Code", y=1.1)
# Set labels for x and y axes
pt.set_xlabel("Purchase Amounts")
pt.set_ylabel("Unit Measure Codes")
# Set names for all items of Unit Code
pt.set_yticklabels (labels=df2.index, fontsize=8, color="green")
# Save bar chart to .pdf file
plt.savefig("c:\\temp\\Figure33_1.pdf",  bbox_inches="tight")';
DECLARE @sql NVARCHAR(MAX);
 SET  @sql = N'
SELECT CAST(h.subtotal AS FLOAT)  AS Total, v.UnitMeasureCode AS Units
  FROM Purchasing.PurchaseOrderHeader h
    INNER JOIN Purchasing.PurchaseOrderDetail d
                ON h.PurchaseOrderID = d.PurchaseOrderID
    INNER JOIN Purchasing.ProductVendor v ON d.ProductID=v.ProductID';
```

```
EXEC sp_execute_external_script
  @language = N'Python',
  @script = @pscript,
  @input_data_1 = @sql;
GO
```

First, the following lines of code from Example 33.5 are related to the visualization of data:

```
import matplotlib
matplotlib.use("PDF")
import matplotlib.pyplot as plt
```

The first line imports the **matplotlib** library into the script. The second line applies the **use** function to the library to specify the format of the output file. This example uses the .pdf format to store the bar chart, but in practice you can choose among the many formats that are supported. The third line of the code imports the **pyplot** plotting framework that is included in the **matplotlib** library. You can use this framework when you intend to do simple plotting.

After executing the preceding three lines of code, all necessary information regarding the libraries is provided, and you can start setting up the data frame used in the script, as previously described in relation to Example 33.4.

After setup of the data frame, the following lines of code describe several properties of the bar chart:

```
pt = df2.plot.barh()
pt.set_title (label="Total Purchases per Unit Code", y=1.1)
pt.set_xlabel("Purchase Amounts")
pt.set_ylabel("Unit Measure Codes")
```

First, the **barh** function uses the modified input data and generates a graphic that contains a bar chart, which is then saved to the **pt** variable. After that, several properties of the bar chart are specified using subplots. One of the important features of the **matplotlib** library is that you can add multiple plots, called *subplots*, within a graphic. These subplots are helpful when you want to show different data presentations in a single view. So, the final three lines of code are regarded as the definition of subplots: the **set_title** function specifies the title of the bar chart, while **set_xlabel** and **set_ylabel** define the names of the X and Y coordinates, respectively. The second parameter of the **set_title** function specifies the font size.

After the properties of the bar chart are configured, the following line of code sets the tick labels in the bar chart:

```
pt.set_yticklabels (labels=df2.index, fontsize = 8, color="green")
```

The **set_yticklabels** function is used to adjust the tick labels in relation to the Y axis. The first argument of the function, **labels**, specifies which column will be used as the index of the result. Therefore, **labels = df2.index** specifies that the values of the **UnitMeasureCode** column will be used as the names of the tick labels on the Y axis in the bar chart.

NOTE The **labels** parameter is related to the **as_index** option of the **groupby** function (see Example 33.3). The use of the **labels** parameter implicitly specifies that an existing column will be the index of the output. Therefore, this is equivalent to the specification of **as_index=True**.

The next line of code in Example 33,

```
plt.savefig("c:\\temp\\Figure33_1.pdf",  bbox_inches="tight")';
```

uses the **savefig** function to store the graphic on a disk. (This function belongs to the **pyplot** framework.) The first parameter of the function specifies the file in which the corresponding bar chart will be stored. (Note that you have to escape the backslashes in the file path by doubling them and, as mentioned earlier, change the path name if your system does not have the **C:\temp** directory.)

The second argument of the **savefig** function, **bbox_inches**, is used to control how many whitespace characters are generated around the displayed graphics. The default value for **bbox_inches** is **None**, meaning that no action will take place. (The other meaningful value is **tight**, which removes unnecessary whitespace characters.) Figure 33-1 shows the output of Example 33.5.

NOTE If you want to create a visualization other than a bar chart, simply call another function by specifying the data frame and the name of the new function. For instance, if you want to display the data in Example 33.5 as a histogram, use the **hist** function instead of the **barh** function.

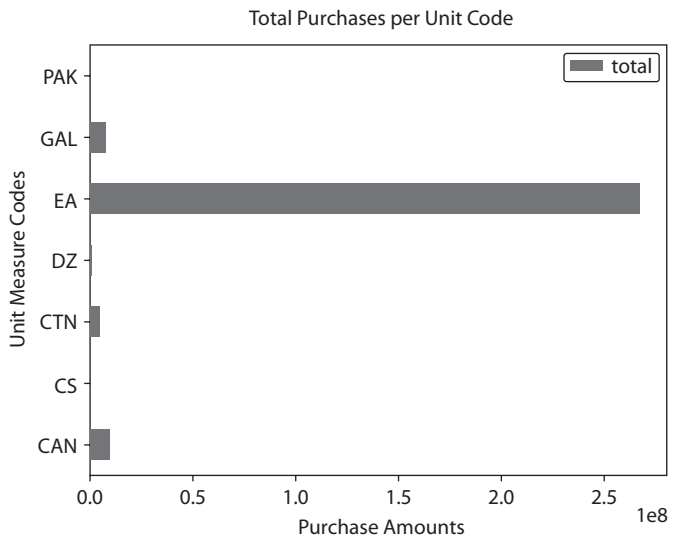

Figure 33-1 The output of Example 33.5

Predictive Modeling with Python in SQL Server

As briefly introduced in Chapter 32, predictive analytics comprises a variety of statistical techniques from machine learning that analyze current and historical data to make predictions about future events. In the business sector, for example, predictive models analyze patterns found in historical and transactional data to identify possible risks and/or opportunities. To do this, a predictive model examines relationships among several parameters. After the analysis, the predictive model presents the various scenarios for evaluation.

Solving Linear Regression Problems Using Python

Linear regression is one of the simplest techniques used in predictive modeling. For this reason, this section presents an example of using linear regression to explain predictive modeling with Python.

NOTE See the Chapter 32 section "Solving Linear Regression Problems with R" for an explanation of the two forms of regression analysis, linear and nonlinear.

For the following example, assume that for each given value, the corresponding value is measured. Measurements in real life, such as this one, usually do not have perfect linear relationships between the values of the independent variable and the values of the dependent variable. For this reason, the goal of linear regression is to find the straight line that best fits the given values (called the *best-fit line*). In other words, linear regression identifies the equation that produces the smallest difference between all the observed values and their fitted values.

The table created in Example 33.6 will be used to show how linear regression can be calculated using Python.

Example 33.6

```
USE sample;
CREATE TABLE Measures (x_value INT, y_value DEC (6,2));
INSERT INTO Measures VALUES (1,   33.5);
INSERT INTO Measures VALUES (2,   35.9);
INSERT INTO Measures VALUES (3,   37.9);
INSERT INTO Measures VALUES (4,   39.8);
INSERT INTO Measures VALUES (5,   41.6);
INSERT INTO Measures VALUES (6,   45.4);
INSERT INTO Measures VALUES (7,   44.6);
INSERT INTO Measures VALUES (8,   47.4);
INSERT INTO Measures VALUES (9,   48.2);
INSERT INTO Measures VALUES (10, 50.3);
```

The ten measured values are stored in the **Measures** table created at the beginning of Example 33.6. Our task is to find linear regression—the formula for the corresponding straight line to which all the given points are at a minimum distance. Example 33.7 calculates the line and displays its formula, together with some other parameters.

Example 33.7

```
USE sample;
EXEC sp_execute_external_script
      @language = N'Python' , @script = N'
     from revoscalepy import rx_lin_mod, rx_predict
linearmodel=rx_lin_mod(formula ="Y_Value~X_Value",data=InputDataSet);
print(linearmodel.summary())',
@input_data_1 = N'SELECT  x_value  AS X_Value,
        CAST (y_value AS FLOAT) AS Y_Value FROM Measures'
```

Let's take a look at the following three lines of code in Example 33.7:

```
from revoscalepy import rx_lin_mod, rx_predict
linearmodel=rx_lin_mod(formula="Y_Value~X_Value",data=InputDataSet);
print(linearmodel.summary())
```

The first line imports the **rx_lin_mod** and **rx_predict** functions from the **revoscalepy** module. (The **revoscalepy** module is a collection of Python functions that you can use for statistics-related tasks, such as linear models, regression, and classification.) The second line incorporates the **rx_lin_mod** function, which is used to fit linear models on small or large data sets. The two most important parameters of the function are **formula** and **data**. The former specifies which statistical model is used. The formula

```
"Y_Value ~ X_Value"
```

defines that the alphanumerical string on the right side of the ~ operator is the name of the independent variable, while the string on the left side is the name of the dependent variable. The **data** argument specifies the input data. The last line of the code prints the summary for the generated model.

After execution of this script, the result looks similar to the following (several lines of the result have been omitted):

```
Linear regression Results for: Y_Value ~ X_Value
Dependent variable(s): ['Y_Value']
Total independent variables: 2
Number of valid observations: 10
Number of missing observations: 0
(Intercept)  (Intercept)  32.360000
X_Value          X_Value   1.836364
Residual standard error: 0.8482 on 8.0 degrees of freedom
Multiple R-squared:0.9797
```

As you already know, when applying regression analysis, the program calculates the straight line that is generated so that all the data points are at a minimum distance from the line. Generally, the formula for the straight line is $y = a + bx$, where a is the y-intercept and b is the

slope of the particular straight line. Therefore, by applying the algorithm for linear regression, we derive the values of the coefficients *a* and *b*. As you can see from the result of Example 33.7, the y-intercept of the straight line is 32.36 and the corresponding slope is 1.83.

Multiple R-squared (the last line of output) evaluates the scatter of the data points around the fitted regression line. It is also called the *coefficient of determination*. For the same data set, higher R-squared (aka R^2) values represent smaller differences between the observed data and the fitted values. R-squared is always between 0 percent and 100 percent. (The larger the value of multiple R-squared, the better the regression model fits your observations.) Therefore, our straight line is a very good choice, because the corresponding R^2 value is almost 98 percent.

Example 33.8 uses the measurements from the **Measures** table to show how data is plotted.

Example 33.8

```
USE sample;
EXEC sp_execute_external_script
      @language = N'Python'
    , @script = N'
#Importing Packages
import matplotlib
matplotlib.use("PDF")
from revoscalepy import rx_lin_mod, rx_predict
import matplotlib.pyplot as plt
import pandas as pd
linearmodel = rx_lin_mod(formula = "Y_Value ~ X_Value", data =
InputDataSet);
df = InputDataSet
plt.scatter(df.X_Value,df.Y_Value)
plt.xlabel("Values of Independent Variable ")
plt.ylabel("Values of Dependent Variable")
#plt.title("Graphical Output of Example 33.8")
plt.plot()
plt.savefig("C:\\temp\\Figure33_2.png") ',
@input_data_1 = N'SELECT x_value  AS X_Value,
        CAST (y_value AS FLOAT) AS Y_Value FROM dbo.Measures'
```

Three lines of code in Example 33.8 merit further explanation:

```
plt.scatter(df.X_Value,df.Y_Value)
plt.plot()
plt.savefig("C:\\temp\\Figure33_2.png") ',
```

The **scatter** function of the **matplotlib.pyplot** framework generates a scatter plot. This function has several parameters, but only the first two have to be specified. These two parameters specify the X and Y coordinates of data points, respectively. The **plot** function in the second line creates the corresponding scatter plot, shown in Figure 33-2.

Finally, the **savefig** function saves the figure. Note that the **use** function of the **matplotlib** library in Example 33.8 specifies the .pdf format as the format of the output file. This value can

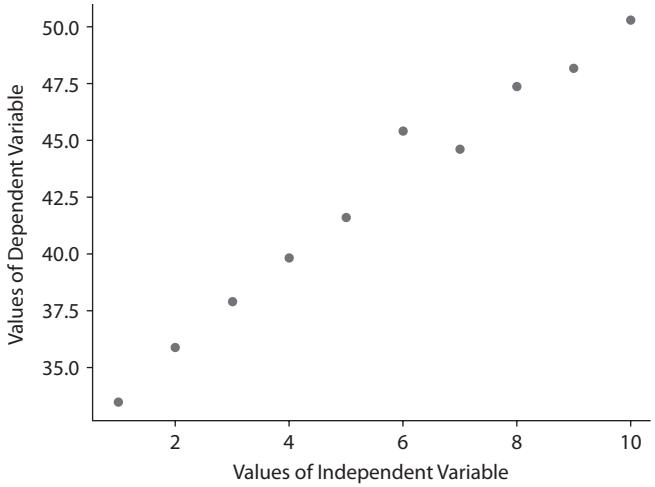

Figure 33-2 The graphical output of Example 33.8

be modified afterwards, using the **savefig** function. As you can see in the example, the format of the scatter plot is modified and stored as a .png file.

Summary

The Python programming language has the following general properties:

- Simple syntax
- High readability
- A large collection of libraries

Python aims to be simple in the design of its syntax, encapsulated in the slogan "There should be one *obvious* way to do it." (For instance, another related programming language, Perl, is known for its complexity in relation to syntax.) Python's design philosophy emphasizes code readability and the use of significant whitespace. With its extensive set of open source libraries, Python is very popular as a programming language for machine learning.

The integration between the Database Engine on one side and Python on the other is done using the **sp_execute_external_script** system stored procedure.