Learning Python for PL/SQL Developers: Part 5

by Arup Nanda

Part 5 of a five-part series that presents an easier way to learn Python by comparing and contrasting it to PL/SQL.

Python with Data and Oracle Database

I hope you have enjoyed learning about Python as the complexity developed progressively in this series until this point. So far, we have covered the elements of the language and how you use them in place of PL/SQL programs you are already familiar with. But since you are most likely a PL/SQL developer, you are a data professional by trade. Your objective, it can be argued, is to manipulate data--in fact large amounts of it. Can you use Python for that? Of course you can.

Earlier you saw some examples of native data handling capabilities of Python expressed in collections (lists, tuples, dictionaries, and so on) that can store large amounts of data in memory and manipulate them well. But, there is more: much, much more. Python offers built-in modules for advanced data handling and visualization to turn data into knowledge, which is the essence of data science.

In this installment you will learn about these data manipulation modules, specifically NumPy, Pandas, MatPlotLib, SQL Alchemy, and CX_Oracle. The last two are for Python programs to connect to Oracle Database, similar to Pro*C.

A word to set expectations may be prudent here: each of these modules is vast enough to cover in a book, let alone a few sections in an article. It's not my intention to present this installment of the series as a complete treatise of these topics. Rather it's an attempt to jumpstart the learning process by providing just the right amount of information. Once you master the basics, you will find much easier to explore the subjects with the help of the documentation.

Installation

All the Python modules are not installed by default. They may need to be installed explicitly after Python is installed. The simplest way to install the modules is to use the Python Installer Package (PIP), which is

somewhat similar to what an app store is to a smartphone. To learn what modules are available for extending Python, simply visit the Python Package Index (PyPI) index page. You can invoke the PIP to install the appropriate module.

Here is how you install a module named oracle-db-query. Please note that oracle-db-query is not discussed in this article. It's merely shown here as an example of how to install a package.

```
C:\>python -m pip install oracle-db-query
Collecting oracle-db-query
  Downloading oracle_db_query-1.0.0-py3-none-any.whl
Collecting pyaml (from oracle-db-query)
  Downloading pyaml-15.8.2.tar.gz
Requirement already satisfied (use --upgrade to upgrade): pandas in c:\python34\lib\site-packages (from
oracle-db-query)

Requirement already satisfied (use --upgrade to upgrade):
cx-Oracle in
c:\python34\lib\site-packages\cx_oracle-5.2.1-py34-win32.egg (from
oracle-db-query)
Collecting PyYAML (from pyaml->oracle-db-query)
  Downloading PyYAML-3.11.zip (371kB)
    100% |################################| 378kB 2.2MB/s
Requirement already satisfied (use --upgrade to upgrade):
numpy>=1.7.0 in c:\python34\lib\site-packages (from
pandas->oracle-db-query)
Requirement already satisfied (use --upgrade to upgrade):
python-dateutil>=2 in c:\python34\lib\site-packages (from
pandas->oracle-db-query)
Requirement already satisfied (use --upgrade to upgrade):
pytz>=2011k in c:\python34\lib\site-packages (from
pandas->oracle-db-query)
Requirement already satisfied (use --upgrade to upgrade):
six>=1.5 in c:\python34\lib\site-packages (from
python-dateutil>=2->pandas->oracle-db-query)
Building wheels for collected packages: pyaml, PyYAML
  Running setup.py bdist_wheel for pyaml ... done
  Stored in directory:
C:\Users\arupnan\AppData\Local\pip\Cache\wheels
\b3\a1\20\62a3bb21201ef3d01e1f41ca396871750998bc00e6697f992d
  Running setup.py bdist_wheel for PyYAML ... done
  Stored in directory:
C:\Users\arupnan\AppData\Local\pip\Cache\wheels\4a\bf
\14\d79994d19a59d4f73efdafb8682961f582d45ed6b459420346
```

```
Successfully built pyaml PyYAML
Installing collected packages: PyYAML, pyaml, oracle-db-query
Successfully installed PyYAML-3.11 oracle-db-query-1.0.0 pyaml-15.8.2
```

Now that you leaned how to install a module, let's install the modules we will be talking out. To install CX_Oracle, NumPy and Pandas, simply use this:

```
python -m pip cx_Oracle
python -m pip pandas
python -m pip numpy
python -m pip matplotlib
```

With the installation out of the way, let's start using the modules.

NumPy

NumPy is a numerical extension to Python. It's useful in many data applications, especially in array operations. Because you are probably crunching a lot of data, mostly in arrays, using NumPy comes very handy. Here is a rudimentary example; but one that drives home the usefulness of the module. Suppose you have an bunch of accounts with their balances in an array, a tuple to be precise, as shown below:

```
>>> balances = (100,200,300,400,500)
```

And you want to raise everyone's balance by 50 percent, that is, multiply balances by 1.5. All these number will need to be updated; so, you do the following:

```
>>> new_balances = balances * 1.5
```

But it comes back with an error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'
```

What happened? Well, you can't just multiply a value directly to an array. To accomplish that you can write something like the following in Python:

```
balances=tuple(new_balances)
new_balances = list(balances)
for i in range(len(balances)):
   new_balances[i] = balances[i]*1.5
```

balances=tuple(new_balances)

But it's cumbersome and takes a lot of coding to do something relatively trivial. As you handle real-world data issues, you will use arrays more and more; so with complex arrays, this approach will be exponentially more complex and probably nonscalable. Fortunately, the NumPy module comes to the rescue. Here is how you do this in NumPy. First, import the NumPy package:

```
>>> import numpy
```

Then we convert to a NumPy array datatype and store it in a new variable called `np_balances`:

```
>>> np_balances = numpy.array(balances)
```

Go ahead and take a look in that variable to see what it contains:

```
>>> np_balances
array([100, 200, 300, 400, 500])
```

It does look amazingly similar to the Python variable, but with as a list (remember, lists are enclosed by "[" and "]") cast as `array()` or, more specifically, a NumPy array. With this in place, it's a piece of cake to perform arithmetic on this variable.

```
>>> np_balances = np_balances * 1.5
```

Take a look at the variable again:

```
>>> np_balances
array([ 150.,  300.,  450.,  600.,  750.])
```

The values are all changed. All this happened with just one multiplication as if you are multiplying a scalar value by 1.5. Actually the value multiplied by doesn't have to be a scalar either. It can be another array. For instance, suppose instead of multiplying all the balances ny 1.5, you need to multiply different values as accounts were given different amounts of raises. Here is the array of multipliers:

```
>>> multipliers = (5, 3, 2, 4, 1)
```

You want the balances to be multiplied with the respective amount, that is, 100 X 5, 200 X 3, and so on. Again, you can't do that in Python in one multiplication step with something like "balances * multipliers" because you can' t multiply an array to another. You have to write something like this:

```
>>> balances = (100,200,300,400,500)
>>> multipliers = (5, 3, 2, 4, 1)
>>> new_balances = list(balances)
```

```
>>> for i in range(len(balances)):
...     new_balances[i] = balances[i]*multipliers[i]
...
>>> balances=tuple(new_balances)
>>> balances
(500, 600, 600, 1600, 500)
```

However, why do all this when NumPy makes it really simple. You convert these into NumPy arrays and simply multiply them:

```
>>> np_balances = numpy.array(balances)
>>> np_multipliers = numpy.array(multipliers)
>>> np_balances = np_balances * np_multipliers
>>> np_balances
array([ 500,  600,  600, 1600,  500])
```

It's that simple. However, the power of the array processing doesn't stop here. You can compare the elements of the array *together* as an array. Here is how you find out which elements are more than 300.

```
>>> np_balances > 300
array([False, False, False,  True,  True], dtype=bool)
```

The result is an array showing which elements have satisfied the condition (`True`) and which ones didn't (`False`).

The NumPy array is accessed by the same indexing mechanism as a regular Python array. Let's see how we access the 0-th element of the NumPy array:

```
>>> balances = (100,200,300,400,500)
>>> np_balances = numpy.array(balances)
>>> np_balances[0]
100
```

But, the NumPy array can also be accessed by the Boolean value of a comparison, as shown below:

```
>>> np_balances[np_balances>300]
array([400, 500])
```

The output shows the elements of the array (returned as another NumPy array) that satisfied the condition we put in, that is, values greater than 300.

Be careful about the using NumPy arrays because sometimes their behavior might not be exactly the same as the regular Python array. Recall from Part 3 that you can add collections, as shown below:

```
>>> balances + multipliers
(100, 200, 300, 400, 500, 5, 3, 2, 4, 1)
```

Python merely makes one list with elements from both arrays. This is not the case in NumPy, where an arithmetic operation is executed on the elements. It adds the corresponding elements to produce another array.

```
>>> np_balances + np_multipliers
array([105, 203, 302, 404, 501])
```

Note how the elements of the new array is simply the corresponding elements added up, which behaves differently from regular Python arrays.

n-Dimensional Arrays

Now that you know how arrays are manipulated easily in NumPy with just one operation, you probably are thinking: hey, aren't these similar to tables where one update operation takes care of the update of all rows? Sure they are. But a real database table is not just one linear array of values. It has multiple columns. Consider a table to store sales for your company. Here is how the table values look:

ProductID ->	0	1	2	3
Sales	1000	2000	3000	4000

This data can be stored in a single linear array:

```
sales = (1000,2000,3000,4000)
```

But in reality, it will not be as simple. You might want to store the sales in each quarter for each product; not just consolidated sales of all quarters for each product. So it will be something like the following:

	Quarter 1	Quarter 2	Quarter 3	Quarter 4
Product 0	200	300	275	225
Product 1	400	600	550	450
Product 2	600	900	1000	500
Product 3	800	1200	1100	900

To store this set of data as a collection in Python, you can't just use a linear array. You need a multidimensional array--almost like a database table with multiple columns. In this case, you just need a two-

dimensional array, the dimensions being the product ID and the quarter. It's ridiculously easy to define an array like this in NumPy. Here is how you can define the 2-D NumPy array:

```
sales = numpy.array([
   [200,300,275,225],
   [400,600,550,450],
   [600,900,1000,500],
   [800,1200,1100,900]
   ])
```

Note how I have defined the array. The NumPy module has a *function* called `array` to which you pass the values of the array you want to define. That's why the entire array is just one parameter to the array function; hence, the parentheses enclosing the values. The actual array, if you notice, is an array of arrays, enclosed by the square brackets. After the assignment, let's check the value of the variable named `sales`:

```
>>> sales
array([[ 200,  300,  275,  225],
       [ 400,  600,  550,  450],
       [ 600,  900, 1000,  500],
       [ 800, 1200, 1100,  900]])
```

NumPy allows you to check the number of dimensions of the array as follows:

```
>>> sales.ndim
2
```

This confirms that this is a two-dimensional array. OK; that's good because that's what we planned. To address individual elements of the array, you use the normal array index notation shown below:

```
>>> sales[0][0]
200
```

NumPy also has a concept of a "shape," which is merely how many elements are defined along each dimension. To know the shape of an array, use the following:

```
>>> sales.shape
(4, 4)
```

This confirms that this is a 4 x 4 array.

Let's bring the complexity up a notch. Your company wants to further break down sales, across territories: North, South, East and West. So, you need to add one more dimension to the array: Territory. The final array has the following columns:

1. ProductID
2. Quarter
3. Territory
4. Amount

To define this array, you simply have to create yet another array within the innermost array, as shown below:

```
sales = np.array([
   [
      [50,50,50,50],
      [150,150,180,120],
      [75,70,60,70],
      [55,45,65,55]
   ],
   [
      [100,90,110,100],
      [150,160,130,170],
      [145,155,150,100],
      [110,115,105,120]
   ],
   [
      [150,140,160,150],
      [225,220,230,225],
      [250,200,300,250],
      [125,130,120,125]
   ],
   [
      [200,250,150,200],
      [300,350,250,300],
      [225,230,220,225],
      [220,225,230,200]
   ]
   ])
```

If you check the dimensions of this new array variable, you'll see this:

```
>>> sales.ndim
3
```

It's a three-dimensional array, as intended. If you want to select the sales of product ID 0 in the second quarter (quarter 1), in the third territory (territory ID 2), you would use the following:

```
>>> sales [0][1][2]
180
```

How did it get this value? We can simplify the process by selecting the arrays one by one. Let's start with just the first dimension:

```
>>> sales[0]
array([[ 50,  50,  50,  50],
       [150, 150, 180, 120],
       [ 75,  70,  60,  70],
       [ 55,  45,  65,  55]])
```

The "sales[0]" refers to the first group of numbers there or, more specifically, to the array of the arrays. The next index is [1], which is the second element of this array:

```
>>> sales[0][1]
array([150, 150, 180, 120])
```

This is an array itself. Therefore the "[2]" index refers to the third element of this array:

```
>>> sales[0][1][2]
180
```

That's it. Since the NumPy array can be an array of arrays, you can define as many dimensions as you want and each dimension will be an array enclosing the child arrays.

But why are you creating the arrays in the first place? You want to do some data analysis, right? Let's do exactly that. We will start with something very basic--determine the total sales of the company:

```
>>> sales.sum()
11215
```

Or, determine the sales mean for products for all quarters and territories:

```
>>> numpy.mean(sales)
175.234375
```

Here, `mean()` is a method in the type. There is also a function in the NumPy module to get the same data:

```
>>> numpy.mean(sales)175.234375
```

But that's not always helpful. You might want to know the mean sales of a specific product. To know mean sales for product 0, use this:

```
>>> sales[0].mean()
151.875
```

Since the data is stored in arrays, you can easily loop through the elements to find the mean sales for all products:

```
>>> for i in range(len(sales)):
...      print('Mean sales of product ',i,'=',sales[i].mean())

Mean sales of product  0 = 151.875
Mean sales of product  1 = 125.625
Mean sales of product  2 = 187.5
Mean sales of product  3 = 235.9375
```

But NumPy makes the process even simpler without loops by introducing the concept of an axis, which is just a dimension in the array. It might be difficult to grasp the axis concept in an array with greater than two dimensions. Let's see a very simple two-dimensional array:

```
>>> a = numpy.array([
   [11,12,13,14],
   [21,22,23,24],
   [31,32,33,34]
   ])
```

To see the values of the array, do this:

```
>>> a
array([[11, 12, 13, 14],
       [21, 22, 23, 24],
       [31, 32, 33, 34]])
```

To get the sum of all *columns*, which is known by the first axis, that is, `axis=0` (remember, all indexes in Python start at 0; not 1, unlike PL/SQL), do this:

```
>>> a.sum(axis=0)
array([63, 66, 69, 72])
```

The result is a another array containing the sums of all the columns. If you want the sum of all rows, which is the second axis, that is, `axis=1`, use the same technique:

```
>>> a.sum(axis=1)
array([ 50,  90, 130])
```

The result is an array with three elements--for three rows.

Now let's consider the 3-D array we defined earlier named `sales`. Figure 1 shows the elements of the array and the axes.

Figure 1. The 3-D `sales` array

The axes allow us to easily summarize the data contained in the cells on the respective axes. Take, for instance, a situation in which we want to know the sum of all sales for all products across all quarters. Note that we don't want the sum of *all* sales, which will give us just one number. We also don't want to know the sum of the sales of each product. We want the total sales of each product for each quarter; that is, we want a 2-D array. The dimensions are product and quarter. To do this, we will need to compute the sum of all numbers along `axis=0`. Figure 2 shows how one cell of the resultant 2-D array is calculated along `axis=0`. All the red-marked cells are summed to come to the number 550.

Figure 2. Calculation along `axis=0`

Likewise, all the other cells are summed in the same manner. Here is the result when we sum the array along `axis=0`.

```
>>> sales.sum(axis=0)
array([[ 550,  580,  520,  550],
       [ 975, 1030,  810,  935],
       [ 770,  725,  790,  715],
       [ 565,  560,  585,  555]])
```

But the power of array summarization is way more than this. What if we wanted to find the sum of sales for all quarters, not products? It will also be a 2-D array; but the dimensions will not be product and quarter. They will be quarter and territory. This will be summed along `axis=1`, as shown in Figure 3. The total of one such set of cells comes to 660. Likewise Python computes the rest of the sets of cells and produces the results. Here is how the final calculated values look.

```
>>> sales.sum(axis=1)
array([[ 660,  630,  550,  590],
       [ 505,  520,  495,  490],
       [ 750,  690,  810,  750],
       [ 945, 1055,  850,  925]])
```

Figure 3. Finding the sum of sales for all quarters

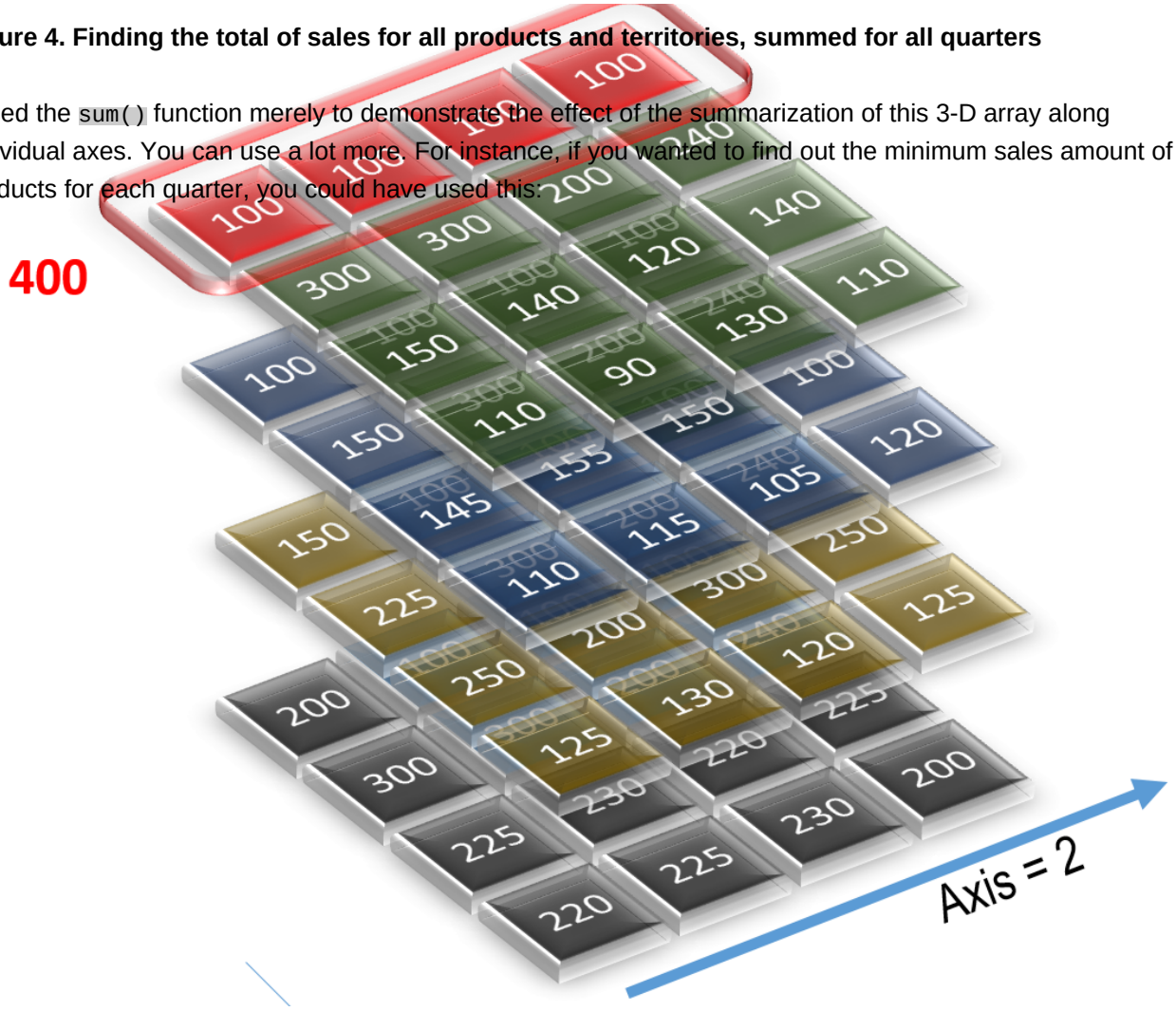
Similarly, what if we wanted to find the total sales for products and territories, summed for all quarters? We can do that by computing the sum along `axis=2`. Figure 4 shows the calculation of one set of cells. The total comes

out to be 400. Likewise, Python computes the sums for all cells and populates the 2-D array. Here is how we get the sum of all sales along `axis=2`.

```
>>> sales.sum(axis=2)
array([[ 400, 1040,  550,  440],
       [ 400,  610,  550,  450],
       [ 600,  900, 1000,  500],
       [ 800, 1200,  900,  875]])
```

Figure 4. Finding the total of sales for all products and territories, summed for all quarters

I used the `sum()` function merely to demonstrate the effect of the summarization of this 3-D array along individual axes. You can use a lot more. For instance, if you wanted to find out the minimum sales amount of products for each quarter, you could have used this:



```
>>> sales.min(axis=0)
array([[100,  90, 100, 100],
       [150, 160, 130, 170],
       [145, 140, 120, 100],
       [110,  90, 105, 110]])
```

Some other functions are `mean()` and `max()`.

Each child array of the NumPy array can be addressed individually, as shown below:

```
>>> for row in sales:
...       print (row)

[[100  90 100 100]
[150 160 130 170]
[145 140 120 100]
[110  90 105 110]]
[[100 100 110 100]
[225 220 200 225]
...
```

This function returns the output in an array form, which is useful if you want to operate on arrays as well. If you want to operate on individual elements of the array, you need to "flatten" it using the `flat()` function, as shown below:

```
>>> for sales_point in sales.flat:
...      print(sales_point)
100
90
...
```

I hope you see how NumPy is useful for data analysis. Of course, it is not possible to show the complete power of the module in this short article. My intention was to give you a glimpse into the basics of NumPy and what possibilities it offers for data professionals. I encourage you to explore more of NumPy in the official NumPy documentation. Later in this article, you will also see some advanced uses of NumPy, especially integrating with other modules.

MatPlotLib

As they say, a picture is worth thousand words. It can't be more true for a data professional. Volumes of data are facts; but they are not necessarily the knowledge that you seek. Creating a graph or a chart of the data

may show exactly what you want. Typically you probably put the data in a CSV file, export into a spreadsheet application such as MS Excel or your favorite data analysis application such as Tableau, and create the charts. Well, that is not exactly convenient. The process has many steps, leading to latency in the development of the charts and, therefore, reducing their effectiveness. It also forces you to perform an intermediate step of dumping data into a file for further analysis. Because you are using Python, you would want to complete everything in that environment without an intermediate step. Besides saving time, that also allows you to script the steps so that the process of going from data analysis to chart presentation can be automated. But how do you create a chart in Python from the data?

This is where the next module we will discuss becomes extremely handy. It's called MatPlotLib, and it's used to draw various types of charts to render data visually. Install the module first, as shown at the beginning of the article. You should also have installed NumPy, which was explained in the previous section.

Let's see the tool in action. First, we will need to import the modules into our environment. Instead of importing the entire MatPlotLib, we will import only the chart plotting functionality from it. We will also use aliases for these modules to save some typing.

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

We will use the data from the previous section where we defined a three-dimensional `sales` array. Let me define that again:

```
sales = np.array([
   [
      [50,50,50,50],
      [150,150,180,120],
      [75,70,60,70],
      [55,45,65,55]
   ],
   [
      [100,90,110,100],
      [150,160,130,170],
      [145,155,150,100],
      [110,115,105,120]
   ],
   [
      [150,140,160,150],
      [225,220,230,225],
      [250,200,300,250],
      [125,130,120,125]
   ],
   [
```

```
    [200,250,150,200],
    [300,350,250,300],
    [225,230,220,225],
    [220,225,230,200]
]
])
```

We will use the simplest charts of all--line graphs of sales of products for each territory for the first quarter. On the x-axis, we just need a series of four numbers to represent the territories. So we define a variable--named simply `x`--to hold this array.

```
>>> x = np.array([0, 1, 2, 3])
```

On the y-axis, however, we need four variables--one for each product. Let's define one array variable for each product. Each of these variables--`y0`, `y1`, `y2`, and `y3`--will be an array.

```
>>> y0 = sales[0][0]
>>> y1 = sales[0][1]
>>> y2 = sales[0][2]
>>> y3 = sales[0][3]
```

Then we will plot a chart of the x and y values. We can optionally add a label. That's useful here because there will be four lines, and we will need to know what is for what:

```
>>> plt.plot(x,y0,label='Product 0')
[<matplotlib.lines.Line2D object at 0x03370CB0>]
>>> plt.plot(x,y1,label='Product 1')
[<matplotlib.lines.Line2D object at 0x03370D90>]
>>> plt.plot(x,y2,label='Product 2')
[<matplotlib.lines.Line2D object at 0x03375710>]
>>> plt.plot(x,y3,label='Product 3')
[<matplotlib.lines.Line2D object at 0x03375BB0>]
```

The legend describing the labels might come up on the chart at an inconvenient location. So, as an optional step, we can tell MatPlotLib to put the legend in a specific location, for example, the upper left corner:

```
>>> plt.legend(loc='upper left')
```

Finally, we tell MatPlotLib to show the chart:

```
>>> plt.show()
```

And voila! This will bring up a chart shown in Figure 5. The important thing to note is that you didn't need to have any special software installed to show the graph. It comes up in a different window.

Figure 5. Chart generated by MatPlotLib

What if you wanted to show the data as a bar chart instead of lines? No worries. The `bar()` function is for that purpose. Use it instead of the `plot()` function used in the previous code. In addition to the other parameters you saw earlier, we will use a new parameter, `color`, to show what color the individual items will be displayed in. The `color` attribute is available in line chart as well.

```
plt.bar(x,y0,label='Product 0', color='blue')
plt.bar(x,y1,label='Product 1', color='red')
plt.bar(x,y2,label='Product 2', color='green')
plt.bar(x,y3,label='Product 3', color='yellow')
plt.legend(loc='upper left')plt.show()
```

The code above will bring up a bar chart (not shown). You will notice come controls on the chart window to zoom in (or out) on the chart to show the details. You can also save the chart as a picture by clicking the appropriate button. That is how I saved the image shown in Figure 5.

Like NumPy, this is merely a basic treatise on the MatPlotLib module. It can do many other things, especially 3-D graphs, which are not discussed here due to lack of space. I encourage you to explore the documentation, especially that of `pyplot`. In the next section, we will use NumPy and MatPlotLib to build something more powerful for data analysis.

Pandas

While NumPy provides many powerful data manipulation functionalities, it might not be as powerful as needed in some cases. That's were a new module--built on the top of NumPy--comes in. It's called Pandas. You have to install the module, as described at the beginning of the article. Once installed, you have to import it into your Python session. Below, we import all the modules we learned about so far:

```
>>> import pandas as pd
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

Pandas provides two types of collections for data analysis:

- **Series**, which is just a one-dimensional array
- **DataFrame**, which is two-dimensional array

Let's examine a simple series. In the earlier sections, we created a NumPy array to store the sales figure of the company for all products with a detailed breakdown for all quarters and all the four territories or zones. However, we didn't actually name the zones as North, South, and so on. Instead we just used their positional indices (0, 1, and so on). Ideally you would like to represent the data in a tabular format in the same way it is shown in table, with proper labels identifying the names of the axes such as this:

	Prod 1	**Prod 2**	**Prod 3**	**Prod 4**
North				
South				
East				
West				

First, we will create a series named `zones` to hold all the zone names. The `Series` (note the uppercase "S") function in the module Pandas creates the series.

```
>>> zones = pd.Series (['North','South','East','West'])
```

Let's take a look at the `zones` variable.

```
>>> zones
0   North
1   South
2    East
3    West
dtype: object
```

The products will be akin to the columns of a table; that is, they will be pretty much fixed. So we define a normal tuple named `products`. Below, only three products are used deliberately to contrast between the numbers of rows and columns.

```
>>> products = ('Prod 1','Prod 2','Prod 3')
```

Now that we have defined rows and columns, we need to enter the sales data. We can enter the data as a NumPy array, as we saw earlier. Let's examine a new concept. There is a built-in random number generator in NumPy. We can create a series of random numbers to store data. Because we need four rows and three columns, we will need to create a grid of that shape. The following command creates a set of random number arranged as four rows and three columns.

```
>>> data1 = np.random.randn(4,3)
```

Let's take a look at the variable named `data1`:

```
>>> data1
array([[-0.1700711 ,  1.05886638, -0.29129644],
       [ 1.5863526 , -1.10870921, -1.43207089],
       [ 0.74046798, -0.45139616,  0.90758052],
       [ 0.83318995, -0.44284258, -1.08616932]])
```

This gives us a perfect set of data to put into the empty two-dimensional array we showed earlier.

That brings up the next important topic--the two-dimensional array. In Pandas, it's known as a *DataFrame*. If you are familiar with R, you might remember this particular term from there. To create a DataFrame in Pandas, we need three things:

- The actual data. We already have it in the variable `data1`.
- The label for columns. We have also defined that as the variable `products`.
- The labels for the rows, which is called "index" in a Python DataFrame. We have the variable `zones` for that.

Here is how we define the DataFrame:

```
>>> df1 = pd.DataFrame(data1,index=zones, columns=products)
```

Let's take a look at the DataFrame we just defined:

```
>>> df1
        Prod 1    Prod 2    Prod 3
North -0.170071  1.058866 -0.291296
South  1.586353 -1.108709 -1.432071
East   0.740468 -0.451396  0.907581
West   0.833190 -0.442843 -1.086169
```

To address the specific elements, we need to have some type of Python datatype, preferably an array. To see the values, use the following:

```
>>> df1.values
array([[-0.1700711 ,  1.05886638, -0.29129644],
       [ 1.5863526 , -1.10870921, -1.43207089],
       [ 0.74046798, -0.45139616,  0.90758052],
       [ 0.83318995, -0.44284258, -1.08616932]])
```

Note how it comes back as a NumPy array, which makes it possible for it to be managed as a NumPy object. For instance, you can update all the elements to 10 times their original values using one command.

```
>>> df2 = df1 * 10
```

```
>>> df2
         Prod 1     Prod 2     Prod 3
North  -1.700711  10.588664  -2.912964
South  15.863526 -11.087092 -14.320709
East    7.404680  -4.513962   9.075805
West    8.331900  -4.428426 -10.861693
```

Now that the DataFrame is in place, let's perform some analysis. Here is an example of quick statistics on the data elements. It shows some basic statistical operations on the values, for example, the total count, mean, standard deviation, maximum, minimum, and so on.

```
>>> df1.describe()
        Prod 1    Prod 2    Prod 3
count  4.000000  4.000000  4.000000
mean   0.747485 -0.236020 -0.475489
std    0.719491  0.917874  1.038394
min   -0.170071 -1.108709 -1.432071
25%    0.512833 -0.615724 -1.172645
50%    0.786829 -0.447119 -0.688733
75%    1.021481 -0.067415  0.008423
max    1.586353  1.058866  0.907581
```

In this case, we showed products as columns. What if you want to see the territories as columns? You can easily do that using the `T` attribute (which stands for "transform"):

```
>>> df2 = df1.T
>>> df2
          North     South      East      West
Prod 1 -0.170071  1.586353  0.740468  0.833190
Prod 2  1.058866 -1.108709 -0.451396 -0.442843
Prod 3 -0.291296 -1.432071  0.907581 -1.086169
>>> df2.describe()
         North     South      East      West
count  3.000000  3.000000  3.000000  3.000000
mean   0.199166 -0.318143  0.398884 -0.231941
std    0.746985  1.657247  0.741090  0.976906
min   -0.291296 -1.432071 -0.451396 -1.086169
25%   -0.230684 -1.270390  0.144536 -0.764506
50%   -0.170071 -1.108709  0.740468 -0.442843
75%    0.444398  0.238822  0.824024  0.195174
max    1.058866  1.586353  0.907581  0.833190
```

Sometimes it's difficult to show all the results on the screen. All you want to see might be the first few lines. Here is how you can display only the first two lines:

```
>>> df1.head(2)
        Prod 1    Prod 2    Prod 3
North -0.170071  1.058866 -0.291296
South  1.586353 -1.108709 -1.432071
```

Or, the last two lines:

```
>>> df1.tail(2)
       Prod 1    Prod 2    Prod 3
East  0.740468 -0.451396  0.907581
West  0.833190 -0.442843 -1.086169
```

In Pandas, the columns are called, well, "columns." The rows are called "rows." The labels for the rows are called "indexes." To see the labels for the DataFrame, display the `index` attribute:

```
>>> df1.index
Index(['North', 'South', 'East', 'West'], dtype='object')
```

To display the columns, do this:

```
>>> df1.columns
Index(['Prod 1', 'Prod 2', 'Prod 3'], dtype='object')
```

Sorting

What if you want to display the values in a sorted manner, instead of the way in which they were put in, to see how the values progress--much like how you would have done in a spreadsheet application? It's quite simple, using the `sort_values()` function. Here is the actual DataFrame called `df1`:

```
>>> df1
        Prod 1    Prod 2    Prod 3
North -0.170071  1.058866 -0.291296
South  1.586353 -1.108709 -1.432071
East   0.740468 -0.451396  0.907581
West   0.833190 -0.442843 -1.086169
```

To sort it along the column called "Prod 1," you call the function with that column as a parameter:

```
>>> df1.sort_values('Prod 1')
        Prod 1    Prod 2    Prod 3
North -0.170071  1.058866 -0.291296
East   0.740468 -0.451396  0.907581
West   0.833190 -0.442843 -1.086169
South  1.586353 -1.108709 -1.432071
```

So far we talked about how to operate on the entire DataFrame. Quite often you might need to extract a subset of values from it. For instance, you might be asked:

- What are the sales of product Prod 1 for each territory?
- What are the sales in the North territory for each product?

This calls for selection from the DataFrame. To select columns, you would just use the DataFrame as an array and use the column name as an index:

```
>>> df1['Prod 1']
North   -0.170071
South    1.586353
East     0.740468
West     0.833190
Name: Prod 1, dtype: float64
```

To select rows, use the DataFrame as an array and use the normal Python-style index notation. For instance to select first row (`index=0`) to the second row (`index=1`), you use this:

```
>>> df1[0:2]
        Prod 1    Prod 2    Prod 3
North -0.170071  1.058866 -0.291296
South  1.586353 -1.108709 -1.432071
```

Note that I used "2" not "1" in the index. Can you tell why?

If you recall from Part 1 of this series, the indexing mechanism in Python arrays for the second part of the index is *up to*, but *not including*, the index. If you want to select until the second row, you enter "2," which signifies the third element. So, `[0:2]` means up to, but not including, the third element.

Just like NumPy, you can use boolean expressions as indices to reference the array elements. For instance, if you want to find out how many of the numbers are positive, do this:

```
>>> df1[df1>0]
        Prod 1    Prod 2    Prod 3
```

```
North       NaN  1.058866       NaN
South  1.586353       NaN       NaN
East   0.740468       NaN  0.907581
West   0.833190       NaN       NaN
```

Note that "NaN" stands for "Not available Number"-- sort of an explicit "N/A."

What if you don't like to use the positional index to get rows of the DataFrame? For instance, position "0" means "North"; but you might not know that. Or what if you don't want to have an extra step of identifying it. You want to address the row by its label, that is, "North." Is that possible? Yes, it's a piece of cake. Here is how:

```
>>> df1.loc['North']
Prod 1   -0.170071
Prod 2    1.058866
Prod 3   -0.291296
Name: North, dtype: float64
```

You can even reference a specific cell in the DataFrame using `loc`. Here is how I get the sales of Prod 1 in the North territory:

```
>>> df1.loc['North','Prod 1']
-0.17007109680992288
```

This method works for multiple cells as well:

```
>>> df1.loc['North',['Prod 1','Prod 2']]
Prod 1   -0.170071
Prod 2    1.058866
Name: North, dtype: float64
```

Another attribute, `iloc`, allows you to slice the DataFrame by positions (not labels) along both rows and columns. Here is an example of how we create a smaller DataFrame with the first two rows and two columns:

```
>>> df1.iloc[0:2,0:2]
        Prod 1    Prod 2
North -0.170071  1.058866
South  1.586353 -1.108709
```

This is the typical Python style of notation in an array. With this notation, we can do any type of slicing. For instance, to get all the rows, merely use ":" without any numbers around it:

```
>>> df1.iloc[:,0:2]
```

```
        Prod 1    Prod 2
North -0.170071  1.058866
South  1.586353 -1.108709
East   0.740468 -0.451396
West   0.833190 -0.442843
```

Similarly, to get all the columns, use ":" in the column parameter:

```
>>> df1.iloc[0:2,:]
        Prod 1    Prod 2    Prod 3
North -0.170071  1.058866 -0.291296
South  1.586353 -1.108709 -1.432071
```

This notation is great when you don't know either the names of the columns (Prod 1, Prod 2, and so on) or the rows (North, South, and so on) or the number of rows and columns.

Once you have a DataFrame, obviously you will want to make some meaningful analysis f it. There are many thing you can do to it with the associated *method* (also called *function*) of the type . One such method (or, function) is `mean()`, which is shown below:

```
>>> df1.mean()
Prod 1    0.747485
Prod 2   -0.236020
Prod 3   -0.475489
dtype: float64
```

The code above shows the mean values for each column. What if you wanted the mean along the other axis? No problem; you simply need to use the axis number as the function argument, as shown below:

```
>>> df1.mean(1)
North    0.128116
South   -0.040313
East     0.391722
West    -0.069807
```

Expanding and Contracting

Consider a case where a fourth product was introduced with the name "Prod 4." How will you add it to the existing DataFrame? Quite simple. Just like you add any variable in Python, simply assign some value to it; there is no need to define it first. In this case, we add an entire column called "Prod 4." Suppose we want to populate it with half the values from Prod 1. Here is how we do it:

```
>>> df1['Prod 4'] = df1['Prod 1'] * 0.5
```

That's it. We just defined and populated a new column. Let's see the values of the DataFrame now:

```
>>> df1
        Prod 1    Prod 2    Prod 3    Prod 4
North -0.170071  1.058866 -0.291296 -0.085036
South  1.586353 -1.108709 -1.432071  0.793176
East   0.740468 -0.451396  0.907581  0.370234
West   0.833190 -0.442843 -1.086169  0.416595
```

You can manipulate the rows as well. To drop the row "North," just use this:

```
>>> df1.drop(['North'])
        Prod 1    Prod 2    Prod 3
South  0.768935 -1.692176 -0.095580
East  -0.894624 -0.264153 -0.511328
West  -1.090884 -0.994317  0.291895
```

Once again, this was merely meant to be a primer on Pandas, not a complete treatise; not even close. You can get more information on Pandas by visiting http://pandas.pydata.org/pandas-docs/stable/.

Pandas DataReader

What can you do with Pandas? Well, analyze data, of course. What sort of data? All sorts, you might say. And where do you get the data from? From all sorts of places--from Google, Yahoo, specialized data providers, and so on--you will probably respond. A specialized module called Pandas DataReader allows you to pull data from some predefined sources without using an intermediate step of creating a text file. Let's see how.

First, you need to install the module, using PIP, as you did for all other modules described at the beginning of the article.

```
C:\>python -m pip install pandas-datareader
Collecting pandas-datareader
  Downloading pandas_datareader-0.2.1-py2.py3-none-any.whl
Requirement already satisfied (use --upgrade to upgrade): pandas in c:\python34\lib\site-packages (from
pandas-datareader)

Collecting requests-file (from pandas-datareader)
  Downloading requests-file-1.4.tar.gz
```

```
Collecting requests (from pandas-datareader)
  Downloading requests-2.11.0-py2.py3-none-any.whl (514kB)
    100% |################################| 522kB 547kB/s
Requirement already satisfied (use --upgrade to upgrade): numpy>=1.7.0 in c:\python34\lib\site-packages
(from pandas->pandas-datareader)
Requirement already satisfied (use --upgrade to upgrade): pytz>=2011k in c:\python34\lib\site-packages (from
pandas->pandas-datareader)
Requirement already satisfied (use --upgrade to upgrade): python-dateutil>=2 in c:\python34\lib\site-packages
(from pandas->pandas-datareader)
Requirement already satisfied (use --upgrade to upgrade): six in c:\python34\lib\site-packages (from requests-
file->pandas-datareader)
Building wheels for collected packages: requests-file
  Running setup.py bdist_wheel for requests-file ... done
  Stored in directory: C:\Users\arup\AppData\Local\pip\Cache\wheels\a4\4c\1c
\8f22cae3cebc5e0b7da07bbf4feee223c2c86702e391c47df0
Successfully built requests-file
Installing collected packages: requests, requests-file, pandas-datareader
Successfully installed pandas-datareader-0.2.1 requests-2.11.0 requests-file-1.4
```

After installation, import the modules we need. We will use a new module called datetime to handle date and time components. It's already installed.

```
>>> import pandas as pd
>>> from pandas_datareader import data as pdr
>>> import datetime
```

For this demo, we will pull in stock market data for a stock symbol called "HOT" from Google Finance. We will get that data from January 1, 2004 to August 1, 20016. Two variables define these boundaries:

```
>>> begin_date = datetime.datetime(2004,1,1)
>>> end_date = datetime.datetime(2016,8,1)
```

Now we will pull the data from Google Finance for these two dates:

```
>>>  hot = pdr.get_data_google("HOT", begin_date, end_date)
```

The variable `hot` is a Pandas DataFrame. Let's see the columns and rows just for two rows:

```
>>> hot.head(2)
            Open   High    Low  Close   Volume
Date
2006-04-10  57.00  57.00  53.98  54.64  3473600
2006-04-11  54.65  55.07  53.92  54.45  2833000
```

Note how easy it was. I pulled the data directly into a variable that I can manage in a Python program without storing the data at an intermediate file location. The output has many columns, as we can see: one each for opening price, highest price, lowest price, closing price, and the volume for each day for each of all the available days. Let's focus on only one--the closing price. We create a new variable to hold that collection:

```
>>> hot_close = hot['Close']
```

If you check, you will see this new variable is of the Pandas "Series" datatype.

```
>>> type(hot_close)
<class 'pandas.core.series.Series'>
```

But we are not importing the data just for the fun of it. Let's create a chart, using MatPlotLib (explained earlier in this article on page 13):

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(hot_close)
[<matplotlib.lines.Line2D object at 0x04D73330>]>>>
plt.show()
```

The code above will bring up the chart as shown in Figure 6. Note the button labeled "Zoom." If you click it, you can zoom to any area of the chart and get the details.

Figure 6. Chart showing stock data

Pandas DataReader is versatile. It can get data from sources such as World Bank, OECD, and so on. For more information visit http://pandas-datareader.readthedocs.io/en/latest/. This is an open source project and you can get the source code of many of the functionalities at https://github.com/pydata/pandas-datareader. It's in GitHub, so the list of sources keeps growing with more contributions from others. So check out the latest.

SQL Alchemy

While Python is a great data manipulation language, you need to get the data first. From the previous installments, you saw how to read from a flat file; in this installment you learned one way to get the data directly from Google Finance. Assuming you are a PL/SQL developer, you are probably dealing with Oracle Database. Surely you would have at least wondered if it were possible to get the data directly from Oracle Database without going through the intermediate steps of creating a flat file.

And the good news is that it is. Several modules exist to allow you to connect to Oracle Database directly from Python to pull data. One such tool is SQL Alchemy, which is a generic module for many databases, not specifically for Oracle Database. First , install it using PIP:

```
c:\>  python -m pip install sqlalchemy
```

In SQL Alchemy, you create a *connection object* to connect to the database, akin to a session in Oracle. The function `create_engine()` is used for that. You specify various properties of the connection object using a *connect string*. The connect string is in the following format:

```
oracle://<User>:<Password>@<Host>:<Port>/<OracleSID>
```

So, to connect to the database as HR (password HR) running on my local machine with 1521 as the listener port and service AL121, I call the function in the following manner, which returns a connection object I named `conn`:

```
>>>  conn = sqlalchemy.create_engine('oracle://hr:hr@localhost:1521/AL121')
```

Using this connection object, I query the employees table using the `read_sql_query` function in SQL Alchemy. This returns the data in a Pandas DataFrame variable I call `emp1`.

```
>>>  emp1 = pd.read_sql_query("select * from employees", conn)
```

Just to confirm, I check the datatype of the variable `emp1`:

```
>>> type(emp1)
<class 'pandas.core.frame.DataFrame'>
```

With `emp1` set, I display the variable:

```
>>> emp1
    employee_id   first_name   last_name     email       phone_number  \
0           198       Donald    OConnell  DOCONNEL       650.507.9833
1           199      Douglas       Grant    DGRANT       650.507.9844
...
```

Because `emp1` is a DataFrame, it is quite easy to manipulate the data using the techniques you learned earlier.

Here, you saw just a glimpse of the capabilities of the SQL Alchemy tool (or, module, to be precise). I don't want to explain it any further because there is a better tool to use to connect to Oracle Databases called CX_Oracle. SQL Alchemy is not database-specific; so it's appeal might be in those cases where multiple databases are involved or database independence is desired. For more information on SQL Alchemy, visit http://docs.sqlalchemy.org/en/latest/core/index.html.

CX_Oracle

CX_Oracle is a module based on the DBAPI specifications of Python, and it is specifically designed for Oracle Database, unlike SQL Alchemy, which is generic. You saw how to install it using PIP at the beginning of the article. Another way to install the module is to download it from PyPI and install manually. That is quite simple as well. Go to the website https://pypi.python.org/pypi/cx_Oracle/5.2.1 and choose the specific file for your Oracle Database version and the Python version. Because I am running Oracle Database 12*c* and Python 3.4 under Microsoft Windows (32-bit), I chose the following:

```
cx_Oracle-5.2.1-12c.win32-py3.4.exe (md5)
```

Once the file is downloaded, execute it. This actually doesn't install anything in the OS; so you don't need administrator privileges. The program will find the correct Python executable location and install in the appropriate directories.

This module needs to connect to Oracle Database using an Oracle client driver or, more specifically, files such as `oci.dll`. So, make sure the `PATH` variable includes the directory where the Oracle Database client is installed. In my case, I am using the Oracle Database Instant Client, which does not require installation. Simply download Oracle Database Instant Client from www.oracle.com, and unzip the contents into a directory. In my case, the directory is `C:\app\instantclient_12_1`; so I include it my `PATH` as follows:

```
C:\> set PATH=C:\app\instantclient_12_1;%PATH%
```

After this is done, you ready to invoke CX_Oracle. Invoke the Python command line interpreter and import the module you just installed. Typing `cx_Oracle` every time might be inconvenient. So, you can import the module as an alias, say, `cxo`.

```
>>> import cx_Oracle as cxo
```

Now that the module is imported, you can use the alias `cxo` to prefix the functions inside it. The first task is to connect to the database. Let's use the HR user. We need to use the `connect()` function in the CX_Oracle module. It accepts the parameters for the user ID, the password, and the connect string in the format <*Machine Name*>:<*Listener Port*>/<*Service Name*>. It returns an object of the `connection` class.

```
>>> conn = cxo.connect('hr','hr','localhost:1521/AL121')
```

When it connects, you can use the `conn` object to manipulate data in the database. If you want to find out later what the connection was for, simply print it:

```
>>> print(conn)
```

<cx_Oracle.Connection to hr@localhost:1521/AL121>

With the connection established, we can use it to interact with Oracle Database. Let's start with something really simple: to find the database version.

```
>>> conn.version
'12.1.0.2.0'
```

Let's execute some queries on the database now. The general steps for executing SQL queries are as follows:

1. Open a cursor object.
2. Bind a statement to that cursor.
3. Execute the cursor to fetch the results.

Do this to open a cursor:

```
>>> cursor1 = conn.cursor()
```

To bind a statement to this cursor we just opened, do this:

```
>>> cursor1.execute('select * from employees where employee_id = 198')
<cx_Oracle.Cursor on <cx_Oracle.Connection to hr@localhost:1521/AL121>>
```

Now let's fetch the data from the database using this cursor:

```
>>> for row in cursor1:
...     print(row)
...
(198, 'Donald', 'OConnell', 'DOCONNEL', '650.507.9833', datetime.datetime(2007, 6, 21, 0, 0), 'SH_CLERK',
2600.0, None, 124, 50)
```

Let's examine what we got. We got all the records from the table EMPLOYEES. If we examine the data in the database table using SQL*Plus,we see this:

```
SQL> select * from employees where employee_id = 198;
EMPLOYEE_ID FIRST_NAME          LAST_NAME
----------- -------------------- -------------------------
EMAIL                    PHONE_NUMBER        HIRE_DATE JOB_ID        SALARY
------------------------- -------------------- --------- ---------- ----------
COMMISSION_PCT MANAGER_ID DEPARTMENT_ID
-------------- ---------- -------------
       198 Donald              OConnell
DOCONNEL                 650.507.9833        21-JUN-07 SH_CLERK        2600
```

124 50

You can see the records from the table appear in the Python output as a tuple. Remember from Part 1 of the series that tuples are collections that are immutable--that is, they are not allowed to change--and they are represented as enclosed within regular parentheses. The columns are represented in Python as normal elements with the correct datatype. The only exception is DATE, where there is no Python equivalent. Therefore, Python has a module called datetime to represent various parts of date and time such as year, month, date, hour, and so on. CX_Oracle correctly translates the information. For instance HIRE_DATE is 21-JUN-07 in Oracle Database. The CX_Oracle module used in Python presented it as follows:

```
datetime.datetime(2007, 6, 21, 0, 0)
```

This is because of the way Python, especially CX_Oracle, handles time and date columns.
Instead of looping through the results in a loop as we saw earlier, you can also fetch just one row at a time using the function `fetchone()`:

```
>>> row = cursor1.fetchone()
>>> print(row)
(198, 'Donald', 'OConnell', 'DOCONNEL', '650.507.9833', datetime.datetime(2007, 6, 21, 0, 0), 'SH_CLERK',
2600.0, None, 124, 50)
```

It retrieved just one row. If you execute the function once again, you will see that the next row is retrieved.

```
>>> row = cursor1.fetchone()
>>> print(row)
(199, 'Douglas', 'Grant', 'DGRANT', '650.507.9844', datetime.datetime(2008, 1, 13, 0, 0), 'SH_CLERK', 2600.0,
None, 124, 50)
```

Remember to type the `()` at the end of the function name; otherwise, it will not work. Instead of fetching only one row, you can fetch many rows using the `fetchmany()` function:

```
>>> rows = cursor1.fetchmany()
>>> print(rows)
[(200, 'Jennifer', 'Whalen', 'JWHALEN', '515.123.4444', datetime.datetime(2003, 9, 17, 0, 0), 'AD_ASST',
4400.0, None, 101, 10),
(201, 'Michael', 'Hartstein', 'MHARTSTE', '515.123.5555', datetime.datetime(2004, 2, 17, 0, 0), 'MK_MAN',
13000.0, None, 100, 20),
(202, 'Pat', 'Fay','PFAY', '603.123.6666', datetime.datetime(2005, 8, 17, 0, 0), 'MK_REP', 6000.0, None, 201,
20), (203, 'Susan',
'Mavris', 'SMAVRIS', '515.123.7777', datetime.datetime(2002, 6, 7, 0, 0), 'HR_REP', 6500.0, None, 101, 40),
(204, 'Hermann',
'Baer', 'HBAER', '515.123.8888', datetime.datetime(2002, 6, 7, 0, 0), 'PR_REP', 10000.0, None, 101, 70), (205,
'Shelley',
```

'Higgins', 'SHIGGINS', '515.123.8080', datetime.datetime(2002,
...

As you can, see it fetched all the rows as one output, which might not be convenient to see. Optionally, you can pass a parameter to the function to limit how many rows it can retrieve. For instance, to fetch only two rows at a time, do this:

```
>>> rows = cursor1.fetchmany(numRows=2)
>>> print(rows)
[(143, 'Randall', 'Matos', 'RMATOS', '650.121.2874', datetime.datetime(2006, 3, 15, 0, 0), 'ST_CLERK', 2600.0, None, 124, 50),
(144, 'Peter', 'Vargas', 'PVARGAS', '650.121.2004', datetime.datetime(2006, 7, 9, 0, 0), 'ST_CLERK', 2500.0, None, 124, 50)]
```

If you execute it again, it will get the next two rows, and so on:

```
>>> rows = cursor1.fetchmany(numRows=2)
>>> print(rows)
[(145, 'John', 'Russell', 'JRUSSEL', '011.44.1344.429268', datetime.datetime(2004, 10, 1, 0, 0), 'SA_MAN', 14000.0, 0.4, 100, 80),
(146, 'Karen', 'Partners', 'KPARTNER', '011.44.1344.467268', datetime.datetime(2005, 1, 5, 0, 0), 'SA_MAN', 13500.0, 0.3, 100, 80)]
```

Note how it fetched the next two rows; not the same two rows. So `fetchone()` and `fetchmany()` act like cursors in PL/SQL. Here is an equivalent program in PL/SQL to fetch one record at a time:

```
declare
  cursor cursor1 is
    select * from employees;
  row     cursor1%rowtype;
begin
  open cursor1;
  while (true) loop
    fetch cursor1
    into row;
    exit when cursor1%notfound;
      dbms_output.put_line('Employee ID='||row.employee_id);
  end loop;
end;
```

Another method, `fetchall()`, also pulls all the records.

```
>>> row = cursor1.fetchall()
```

```
>>> print(row)
```

This will print all the records pulled by the cursor. However, `fetchall()` simply pulls one record at a time, making too many round trips to the database from the client. This is still inefficient. To improve the performance, CX_Oracle allows you to define an array size.

```
>>> cursor1.arraysize = 10
```

This setting forces the `fetchall()` to pull 10 rows at a time, very similar to the SQL*Plus `arraysize` property.

To see what the columns of a cursor are, you can use the attribute called `description`:

```
>>> cursor1.description
[('EMPLOYEE_ID', <class 'cx_Oracle.NUMBER'>, 7, 22, 6, 0, 0), ('FIRST_NAME', <class
'cx_Oracle.STRING'>, 20, 20, 0, 0, 1),
('LAST_NAME', <class 'cx_Oracle.STRING'>, 25, 25, 0, 0, 0), ('EMAIL', <class 'cx_Oracle.STRING'>, 25, 25, 0,
0, 0),
('PHONE_NUMBER', <class 'cx_Oracle.STRING'>, 20, 20, 0, 0, 1), ('HIRE_DATE', <class
'cx_Oracle.DATETIME'>, 23, 7, 0, 0, 0),
('JOB_ID', <class 'cx_Oracle.STRING'>, 10, 10, 0, 0, 0),('SALARY', <class 'cx_Oracle.NUMBER'>, 12, 22, 8, 2,
1),
('COMMISSION_PCT', <class 'cx_Oracle.NUMBER'>, 6, 22, 2, 2, 1), ('MANAGER_ID', <class
'cx_Oracle.NUMBER'>, 7, 22, 6, 0, 1),
('DEPARTMENT_ID', <class 'cx_Oracle.NUMBER'>, 5, 22, 4, 0, 1)]
```

In the previous case, we used `cursor1.execute()` and then `cursor1.fetchall()`. We need not have. We could have put that into one step by storing the results of the `execute()` method in a variable, which is of type tuple, as shown below.

```
>>> rows = cur1.execute('select * from employees')
```

Then we could have used the iteration approach to read the data from the variable named `rows`:

```
>>> for row in rows:
...     print(row)
```

The output would have been the same as we saw earlier. Let's see a slight variation of that.

```
>>> rows.description()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not callable
```

The error occurred because you used `description()`. The parentheses after `description` indicates that the it is a method (a function) in the object called `rows`. Actually it's not a method; it's an *attribute* of the object. Therefore, you shouldn't use the parentheses, as shown below:

```
>>> rows.description
[('EMPLOYEE_ID',
<class 'cx_Oracle.NUMBER'>, 7, 22, 6, 0, 0), ('FIRST_NAME',
<class 'cx_Oracle.STRING'>, 20, 20, 0, 0, 1),
('LAST_NAME', <class
'cx_Oracle.STRING'>, 25, 25, 0, 0, 0), ('EMAIL', <class
'cx_Oracle.STRING'>, 25, 25, 0, 0, 0),
('PHONE_NUMBER', <class
'cx_Oracle.STRING'>, 20, 20, 0, 0, 1), ('HIRE_DATE', <class
'cx_Oracle.DATETIME'>, 23, 7, 0, 0, 0),
('JOB_ID', <class
'cx_Oracle.STRING'>, 10, 10, 0, 0, 0),
('SALARY', <class
'cx_Oracle.NUMBER'>, 12, 22, 8, 2, 1), ('COMMISSION_PCT', <class
'cx_Oracle.NUMBER'>, 6, 22, 2, 2, 1),
('MANAGER_ID', <class 'cx_Oracle.NUMBER'>, 7, 22, 6, 0, 1), ('DEPARTMENT_ID', <class
'cx_Oracle.NUMBER'>, 5, 22, 4, 0, 1)]
```

Bind Variables

So far we have used very simple queries with pure literal values. As you know, this is bad for performance with Oracle Database because each query will be treated as a separate statement, creating unnecessary parsing and filling up the library cache. Besides, you might have to create multiple cursors for statements where only predicates change, adding to the complexity of coding. Well, no need to worry; bind variables come to rescue, just as in PL/SQL. You create cursors with bind variables (or placeholders) in them and pass the values at runtime, not at coding time. Let's see how to do it:

First, close the cursor:

```
>>> cursor1.close()
```

Reopen the cursor:

```
>>> cursor1 = conn.cursor()
```

This time, we have to prepare the cursor explicitly. We didn't have to do this step earlier because `execute()` took care of it:

```
>>> cursor1.prepare('select * from employees where employee_id = :emp_id')
```

Note how similar the bind variable definition is (`:VarName`) in Python compared to PL/SQL. We then execute the cursor. Because there is a bind variable `emp_id`, we have to pass the value. A cursor may have multiple bind variables; so instead of just a scalar value, we need to pass the values as a dictionary. Remember, dictionary datatypes are nothing but key-value pairs; so the bind variables are keys and their values are, well, values of the bind variables. Also remember that the dictionary datatype is represented as enclosed within curly braces ("{" and "}").

```
>>> cursor1.execute(None,{'emp_id':198})
<cx_Oracle.Cursor on <cx_Oracle.Connection to hr@localhost:1521/AL121>>
```

After that, fetch or loop through the cursor as usual:

```
>>> cursor1.fetchall()
[(198, 'Donald', 'OConnell', 'DOCONNEL', '650.507.9833', datetime.datetime(2007, 6, 21, 0, 0), 'SH_CLERK',
2600.0, None, 124, 50)]
```

If you want to display the names of the bind variables, you can call the function `bindnames()`, as shown below:

```
>>> print (cursor1.bindnames())
['EMP_ID']
```

What about dynamically constructed SQL statements, similar to Native Dynamic SQL in PL/SQL, which is a text string executed by `execute immediate`? Of course, CX_Oracle has a similar construct.

```
>>> sqlStmt = 'select '
>>> sqlStmt += ' '
>>> sqlStmt += '*'
>>> sqlStmt += ' '
>>> sqlStmt += 'from employees'
>>> sqlStmt += ' '
>>> sqlStmt += 'where employee_id = '
>>> sqlStmt += '198'
>>> cursor1 = conn.cursor()
>>> cursor1.execute(sqlStmt)
<cx_Oracle.Cursor on <cx_Oracle.Connection to hr@localhost:1521/AL121>>
>>> cursor1.fetchall()
[(198, 'Donald', 'OConnell', 'DOCONNEL', '650.507.9833', datetime.datetime(2007, 6, 21, 0, 0), 'SH_CLERK',
2600.0, None, 124, 50)]
```

But data selection is not the only thing you can do in CX_Oracle. You can issue data manipulations statements such as `update`, `delete`, and so on as well. Here is an example of `update` followed by a `select`, both using bind variables.

```
>>> cursor1 = conn.cursor()
>>> cursor1.prepare('update employees set salary = salary * 2 where employee_id = :emp_id')
>>> cursor1.execute(None,{'emp_id':198})
>>> cursor2 = conn.cursor()
>>> cursor2.prepare('select salary from employees where employee_id = :emp_id')
>>> cursor2.execute(None,{'emp_id':198})
<cx_Oracle.Cursor on <cx_Oracle.Connection to hr@localhost:1521/AL121>>
>>> cursor2.fetchall()
[(5200.0,)]
```

DML statements bring the need for another concept in Oracle Database: transactions. At the end of a transaction, you either commit or roll back. A built-in method of the `connection` object can do either of those. Note the salary is now 5200, up from 2600 earlier. At this point, you can commit:

```
>>> conn.commit()
```

Or, you can roll back:

```
>>> conn.rollback()
```

While you are no doubt happy to see the insert statements, you must be thinking: well, this might be good for some demo purpose. In real life, you would probably get lots of data in one shot (such as the stock market data from Google Finance we saw in the previous section on Pandas DataReader) and you would have to insert lots of records. Writing an insert statement for each record must be exhausting. Is there a better way?

Consider the following table:

```
SQL> create table test (c1 number, c2 varchar2(1));
Table created.
```

To insert data into this table, you will need to write something like this in CX_Oracle:

```
>>> inputData = (100,'A')
>>> cursor1.execute("insert into test values (:1,:2)", inputData)
```

Note how the values of the bind variables are not assigned individually but as a tuple in the form of the variable named `inputData`. It makes it simple to enter variable values. And then you repeat the first statement to assign all the values to the `inputData` variable and repeat the second statement thousands, perhaps hundreds of thousands, of times! That is not feasible. Not only that, you might have the data to be input in the form of an

array, which means you have to write a loop to read each element from the array and execute the insert. Not something to look forward to. Not to worry; the real power comes in bulk inserts for CX_Oracle. Let's see how it is done.

Suppose we have the data to be inserted in the form of an array (a list):

```
>>> inputData = [ (100,'A'),
... (200,'B'),
... (300,'C')]
```

With this, let's perform the database operations:

```
>>> cursor1 = conn.cursor()
>>> cursor1.executemany("insert into test values (:1,:2)", inputData)
```

Note how I used the `inputData` variable as the bind variable value, even though the variable is no longer a tuple of the bind variables. It's an array, that is, a list of many tuples. CX_Oracle understands it and executes the insert for each element of the list. After this, we can create another cursor to select from the table to confirm the insert actually worked:

```
>>> cursor2 = conn.cursor()
>>> cursor2.execute("select * from test")
<cx_Oracle.Cursor on <cx_Oracle.Connection to hr@localhost:1521/AL121>>
>>> cursor2.fetchall()
[(100, 'A'), (200, 'B'), (300, 'C')]
```

Let's check in the database, too. First, we commit:

```
>>> conn.commit()
```

Then we check in the database using SQL*Plus:

```
SQL> select * from test;

        C1 C
---------- -
       100 A
       200 B
       300 C
```

Yes; we have the records. Bulk inserts are powerful in CX_Oracle because they allow you to insert several records in one statement using a collection.

Procedures and Functions

So far, we talked about single SQL statements. As PL/SQL developers, that's not the only thing you do. Many of the applications you develop are probably in the form of stored procedures and functions in the database. CX_Oracle makes it easy to call them as well. Let's see with an example procedure:

```
create or replace procedure proc1 (
  p1 number,
  p2 varchar2
) is
begin
  dbms_output.put_line('p1='||p1||' p2='||p2);
end;
```

Sometimes you might create a procedure with an OUT parameter, which puts the value in that parameter to be used by the caller. CX_Oracle can handle that as well. Here is such a procedure:

```
create or replace procedure proc2 (
   p1 in number,
   p2 out number
) is
begin
   p2 := p1*2;
   dbms_output.put_line('p1='||p1||' p2='||p2);
end;
```

Let's create a stored function that accepts two parameters and returns a number:

```
create or replace function func1 (
    p1 number,
    p2 number
)
return number
is
begin
   return p1+p2;
end;
```

To execute stored procedures, CX_Oracle has a function called `callproc()` that accepts the name of the procedure and all the parameter values as a tuple. Here is how you execute the procedure `proc1`:

```
>>> conn = cxo.connect('hr','hr','localhost:1521/AL121')
>>> cursor1 = conn.cursor()
>>> cursor1.callproc('proc1',(1,'A'))
```

To execute the procedure with OUT parameter, you need to first define a cursor variable. Note this is in the context of Python, not the cursor variable in PL/SQL. Here is how you define the variable named `v1`:

```
>>> v1 = cursor1.var(cxo.NUMBER)
```

Note "NUMBER" is in uppercase. Use it for any number datatype in Oracle Database. For VARCHAR2, use `cxo.STRING`, for CHAR use `cxo.FIXED_CHAR`, for DATE use `cxo.DATETIME`, and for TIMESTAMP use `cxo.TIMESTAMP`. Now define the cursor:

```
>>> cursor2 = conn.cursor()
```

And call the procedure in the same way as before, except that your second parameter, which is an OUT type, is `v1`, the variable you defined earlier.

```
>>> cursor2.callproc('proc2',(1,v1))
[1, 2.0]
```

After the procedure executes, the variable `v1` has the OUT value. You can check it:

```
>>> v1
<cx_Oracle.NUMBER with value 2.0>
```

There is a built-in method for the object `getvalue()` that returns the value of the variable:

```
>>> v1.getvalue()
2.0
```

Calling a stored function is similar. The `callfunc()` method of the cursor object is used to execute the stored function. It accepts three parameters: the function name, the return datatype, and the parameters in a tuple. The method returns the return value of the stored function. This is how you execute the function:

```
>>> cursor3 = conn.cursor()
>>> result = cursor3.callfunc('func1',cx_Oracle.NUMBER,(1,1))
```

You can print the result:

```
>>> print(result)
2.0
```

As for all the other modules, this was merely an attempt to give you a jumpstart on the topic, not a comprehensive explanation. I hope your appetite was whetted enough to start exploring more. You can find more information on CX_Oracle at https://cx-oracle.readthedocs.io/en/latest/.

Summary

This article showed you a few modules for powerful data analysis. NumPy is a module for number crunching, where multidimensional arrays can be manipulated as individual elements. Pandas is another module built on the top of NumPy that takes this array concept even further with series (one-dimensional arrays) and DataFrames (two-dimensional arrays). MatPlotLib is a module for plotting charts of various kinds such as line and bars. The DataReader module allows you to read charts from public data sources such as Google Finance, World Bank, and many others. SQL Alchemy is a module to access databases such as Oracle Database, MySQL, and so on. Another module, CX_Oracle, provides the same functionality (and much more detail) as SQL Alchemy except only for Oracle Database.

As mentioned at the beginning, my objective for this article was not to provide a comprehensive review of all the modules in Python, but merely to provide enough to give a jumpstart and whet your appetite for exploring them further. There are many other modules in Python for performing more-specialized tasks. For instance, SciPy is a module for scientific analysis.

This brings us to the conclusion of this five-part series on learning Python for PL/SQL developers, starting with the basics of the language, going through successively complex topics such as loops, conditions, collections, functions, modules, file handling, and, finally, specialized data analysis modules. I hope you enjoyed this series and acquired enough Python skills to build upon and become an expert in the language and the modules to turn into a successful data scientist or solve complex data problems. Happy learning.

About the Author

Arup Nanda (arup@proligence.com) has been an Oracle DBA since 1993, handling all aspects of database administration, from performance tuning to security and disaster recovery. He was *Oracle Magazine's* DBA of the Year in 2003 and received an Oracle Excellence Award for Technologist of the Year in 2012.