Learning Python for PL/SQL Developers: Part 4

by Arup Nanda

Part 4 of a five-part series that presents an easier way to learn Python by comparing and contrasting it to PL/SQL.

You may follow along with the Online Presentation: Learning Python for PL/SQL Developers - Part 4

Functions, Modules and File Operations

As is the case in most languages, Python provides repeatable code segments, similar to procedures and functions in PL/SQL. As you already know, in PL/SQL, a procedure does not return anything (although it can have an OUT parameter; but that's not a return, so it's not the same thing) and a function returns a single value. In Python, the equivalents of both PL/SQL procedures and functions is called simply a function. A Python function may or may not return anything. In this article, we will cover how to write functions and use them in your programs. As in the previous article in this series, we will see how to do something in PL/SQL and then do the equivalent in Python.

A function definition in PL/SQL has this general syntax format:

```
function FunctionName (
  Parameter1Name in DataType,
  Parameter2Name in DataType,
  ...
)
return ReturnDatatype
is
   localVariable1 datatype;
   localVariable2 datatype;
begin
   ... function code ...
    return ReturnVariable;
```

```
end;
```

A procedure definition in PL/SQL has this general syntax:

```
procedure ProcedureName(
  Parameter1Name in DataType,
  Parameter2Name in DataType,
  ...
)
is
   localVariable1 datatype;
   localVariable2 datatype;
begin
   ... procedure code ...
end;
```

A function definition in Python follows this simple syntax:

```
def FunctionName (Parameter1Name,Parameter2Name, ...):
        ... function code ...
  return ReturnVariable
```

Note some important properties of the Python function definition compared to the PL/SQL equivalent:

- The definition starts with `def` followed by the name of the function.
- The parameters come inside the parentheses.
- Only the parameter names are listed. The datatypes are not mentioned, unlike with PL/SQL. Parameters are optional; a function doesn't need to have parameters.
- There is a colon (:) at the end of the function name to mark the end of the name and beginning of the definition. Unlike PL/SQL, there is no "BEGIN ... END" construct.
- There is no mention of what the function returns at the definition level. If you notice there is no mention of whether the function even returns anything at the definition level.
- The indentation after the colon shows the code of the function. This is the same style followed in Python to mark "IF ... THEN ... ELSE" blocks or loops. Indentations mark the beginning and end of function code, not "begin ... end," as in PL/SQL.
- The function may optionally return something at the final line. The syntax is the same as in PL/SQL: `return` *ReturnVariable*.

Now that you've got the basic idea about the syntax vis-a-vis PL/SQL, let's start with a very simple procedure in PL/SQL that accepts a principal amount and interest rate, computes the interest amount and the new principal after the interest is added, and displays the new principal.

Here is how we do it in PL/SQL. Note that I deliberately chose to use the Python naming convention, for example, `pPrincipal`, not a PL/SQL-style variable name such as `p_principal`.

PL/SQL

```
-- pl1.sql
declare
  procedure calcInt (
   pPrincipal     number,
   pIntRate       number
  ) is
    newPrincipal   number;
  begin
    newPrincipal := pPrincipal * (1+(pIntRate/100));
    dbms_output.put_line ('New Principal is '||newPrincipal);
  end;
begin
  calcInt(100,10);
end;
/
```

Here is the output:

```
New Principal is 110

PL/SQL procedure successfully completed.
```

Python

```
#py1.txt
def calcInt(pPrincipal, pIntRate):
    newPrincipal = pPrincipal * (1+(pIntRate/100))
    print ("New Principal is " + str(newPrincipal))

calcInt(100,10)
```

Executing it produces this:

```
C:\>python py1.txt
New Principal is 110.00000000000001
```

This shows an example of a very basic Python function that does not return anything, similar to a PL/SQL procedure. Now that you know the basics, let's explore some more intricate details of the syntax.

Default Value of Parameters

Sometimes you need to pass a default value to a parameter. This value is in effect if the user does not explicitly pass the parameter. Building on the previous procedure, suppose we want to make the parameter `pIntRate` optional, that is, make it a certain value (such as 5, when the user does not explicitly mention it). In PL/SQL, you mention the parameter this way:

ParameterName DataType := DefaultValue

In Python, it's exactly the same, but since the assignment operator in Python is "`=`" (not "`:=`"), that's what you need to use. Besides, remember, you don't mention the datatype for parameters. Here is the general syntax:

ParameterName = DefaultValue

You can write the PL/SQL function this way (the changes are in bold):

PL/SQL

```
--pl2.sql
declare
  procedure calcInt (
    pPrincipal      number,
    pIntRate        number := 5
  ) is
   newPrincipal    number;
  begin
    newPrincipal := pPrincipal *(1+(pIntRate/100));
    dbms_output.put_line('New Principal is '||newPrincipal);
  end;
begin
  -- don't mention the pIntRate parameter.
  -- defaults to 5
```

```
  calcInt(100);
end;
/
```

The output is:

```
New Principal is 105

PL/SQL procedure successfully completed.
```

Python

```
#py2.txt
def calcInt(pPrincipal, pIntRate = 5):
   newPrincipal = pPrincipal * (1+(pIntRate/100))
   print ("New Principal is " + str(newPrincipal))

# Don't pass the 2nd parameter. Defaults to 5
calcInt(100)
```

Executing it produces this:

```
C:\>python py2.txt
New Principal is 105.0
```

If you need to have a string as a default variable, use double quotes:

```
# py2a.txt
def calcInt(pPrincipal, pIntRate = 5, pAccType = "Checking"):
   vIntRate = pIntRate
   if (pAccType == 'Savings'):
       # eligible for bonus interest
       vIntRate = pIntRate + 5
    newPrincipal = pPrincipal * (1+(vIntRate/100))
    print ("New Principal is " + str(round(newPrincipal)))

calcInt(100,10,'Savings')
```

Executing it produces this:

```
C:\>python py2a.txt
```

```
New Principal is 115
```

One important property of functions in Python is that the default values can be variables as well. This is not possible in PL/SQL. For instance, in PL/SQL the following will be illegal:

```
-- pl4.sql
declare
   defIntRate     number := 5;
   procedure calcInt (
          pPrincipal      number,
          pIntRate        number := defIntRate;
   ) is

...
```

But it's perfectly valid in Python. Let's see how:

```
# py4.txt
defIntRate = 5
def calcInt(pPrincipal, pIntRate = defIntRate):
  newPrincipal = pPrincipal * (1+(pIntRate/100))
  print ("New Principal is " + str(round(newPrincipal)))

calcInt(100)
```

Another important property of this variable assignment is that the assignment occurs only at the time of the declaration of the function, and is not affected afterwards. Take for instance the following:

```
# py4a.txt
# Assign the value to the variable
defIntRate = 5

# define the function
def calcInt(pPrincipal, pIntRate = defIntRate):
   newPrincipal = pPrincipal * (1+(pIntRate/100))
   print ("New Principal is " + str(round(newPrincipal)))

# change the variable value
defIntRate = 10

# call the function
calcInt(100)
```

What value will be printed? Will the function take the value of `pIntRate` as 5 or 10?

The answer is it will take the value of `pIntRate` as 5, not 10. Why? It's because when the function was defined, the value was 5. When the function was called, the value of `pIntRate` was 10; but that will not be considered. This is a very important property you need to keep in mind when learning Python as a PL/SQL developer. It's a major source of bugs if not taken into consideration.

Positional Parameters

You already know that in PL/SQL you do not have to provide the parameter values in the order in which they were defined in the procedure. You can pass values by specifying the parameter by name. For instance, if a procedure `F1` assumes the parameters `P1` and `P2`--in that *order*--you can call the procedure this way with the parameter values `Val1` and `Val2` respectively:

```
F1 (Val1, Val2);
```

But you can also call them with explicit parameter name assignments:

```
F1 (P2 => Val2, P1 => Val1);
```

This explicit naming allows you to order the parameters any way you want while calling the procedure. It also allows you to skip some non-mandatory parameters. In Python the equivalent syntax is this:

```
F1 (P2=Val2, P1=Val1)
```

So, just the operator "`=>`" is changed to "`=`." Let's see examples in both PL/SQL and Python.

PL/SQL

```
--pl3.sql
declare
   procedure calcInt (
     pPrincipal      number,
     pIntRate        number := 5
   ) is
     newPrincipal   number;
   begin
     newPrincipal := pPrincipal *(1+(pIntRate/100));
     dbms_output.put_line('New Principal is '||newPrincipal);
```

```
  end;
begin
  calcInt(pIntRate=>10, pPrincipal=>100);
end;
/
```

The output is this:

```
New Principal is 110

PL/SQL procedure successfully completed.
```

Python

```
#py3.txt
def calcInt(pPrincipal, pIntRate = 5, pAccType = "Checking"):
  vIntRate = pIntRate
  if (pAccType == 'Savings'):
    # eligible for bonus interest
    vIntRate = pIntRate + 5
  newPrincipal = pPrincipal * (1+(vIntRate/100))
  print ("New Principal is " + str(round(newPrincipal)))

calcInt(pAccType="Savings", pIntRate=10, pPrincipal=100)
```

One of the useful cases in PL/SQL is to define a default value only when the value is not explicitly provided. Take for instance, when the user didn't specify anything for the interest rate, and you want the default values to be based on something else, for example, the account type. If the account type is Savings (the default), the interest rate should should be 10 percent; otherwise, it should be 5 percent. Here is how you will need to write the function:

```
-- pl3b.sql
declare
  procedure calcInt (
    pPrincipal      number,
    pIntRate        number := null,
    pAccType        varchar2 := 'Savings'
  ) is
    newPrincipal    number;
    vIntRate        number;
  begin
```

```
    if (pAccType = 'Savings') then
      if (pIntRate is null) then
     vIntRate := 10;
      else
     vIntRate := pIntRate;
   end if;
    else
      if (pIntRate is null) then
        vIntRate := 5;
      else
        vIntRate := pIntRate;
      end if;
    end if;
    newPrincipal := pPrincipal * (1+(vIntRate/100));
    dbms_output.put_line('New Principal is '|| newPrincipal);
  end;
begin
  calcInt(100);
  calcInt(100, pAccType => 'Checking');
end;
/
```

The equivalent of this:

```
pIntRate       number := null,
```

in Python is this:

```
pIntRate = None
```

(Note the capitalization of `None`). Here is the complete Python code:

```
# py3b.txt
def calcInt(pPrincipal, pIntRate = None, pAccType = "Savings"):
  vIntRate = pIntRate
  if (pAccType == 'Savings'):
    if (pIntRate == None):
      vIntRate = 10
    else:
      vIntRate = pIntRate
  else:
    if (pIntRate == None):
      vIntRate = 5
```

```
    else:
      vIntRate = pIntRate
  newPrincipal = pPrincipal * (1+(vIntRate/100))
  print ("New Principal is " + str(round(newPrincipal)))

calcInt(100)
calcInt(100, 20)
calcInt(100, pAccType="Checking")
```

Here is the output when we execute the code:

```
C:\>python py3b.txt
New Principal is 110
New Principal is 120
New Principal is 105
```

Returning Values

So far we have talked about the equivalent procedures in PL/SQL, which do not return anything. In contrast, functions in PL/SQL return a value. Here is a simple example of a function that returns the interest rate for the account type, which is the parameter passed to it:

```
--pl5
declare
  function getIntRate
  (
    pAccType   in varchar2
  )
  return number
  is
    vRate    number;
  begin
    case pAccType
        when 'Savings' then vRate := 10;
        when 'Checking' then vRate := 5;
        when 'MoneyMarket' then vRate := 15;
     end case;
     return vRate;
  end;
begin
```

```
   dbms_output.put_line('Int Rate = '||getIntRate('Savings'));
   dbms_output.put_line('Int Rate = '||getIntRate('Checking'));
   dbms_output.put_line('Int Rate = '||getIntRate('MoneyMarket'));
end;
/
```

The equivalent of this:

```
return vRate;
```

in Python, fortunately, is exactly the same:

```
return vRate
```

Let's see how we can write the Python code. Remember, there is no CASE equivalent in Python; so we have to resort to the `if ... elif ...else` construct.

```
# py5.txt
def getIntRate (pAccType):
   if (pAccType == 'Savings'):
      vIntRate = 10
   elif (pAccType == 'Checking'):
      vIntRate = 5
   elif (pAccType == 'MoneyMarket'):
      vIntRate = 15
   return vIntRate

print('Int Rate = ', getIntRate ('Savings'))
print('Int Rate = ', getIntRate ('Checking'))
print('Int Rate = ', getIntRate ('MoneyMarket'))
```

Executing it produces this:

```
C:\>python py5.txt
Int Rate =  10
Int Rate =  5
Int Rate =  15
```

Documentation

When you write a lot of functions, you might lose track of what each one does. Others may be even more confused. You can add documentation for the function to show what the function does. In PL/SQL, you use this syntax:

```
/*  ... documentation ... */
```

In Python, the equivalent is to put lines between a pair of three double quotes. Anything inside the pair of three double quotes will be considered documentation.

```
# py6.txt
def myFunc (pParam1, pParam2):
   """Version : 2.0
   Purpose      : The purpose comes here
   Created      : mm/dd/yyyy
   Author       : XXX
   Last Changed : mm/dd/yyyy
   Change History:
   Date    Ver Made By Description
   ------- --- ------- -----------------------------
   mm/dd/yy 1.0 XXX    Created
   mm/dd/yy 2.0 yyy    Changed something
   """
   vInt = pParam1 + pParam2
   return vInt

print(myFunc(1,2))
```

But, unlike PL/SQL, which interprets the text between `/*` and `*/` as comments, Python interprets this correctly as function documentation and not as mere comments. There is a special attribute called `__doc__` of any function that holds this documentation, and if it is defined it can be called later. Here is how you can reference this attribute:

```
# py7.txt
def myFunc (pParam1, pParam2):
   """Version : 2.0
   Purpose      : The purpose comes here
   Created      : mm/dd/yyyy
   Author       : XXX
   Last Changed : mm/dd/yyyy
   Change History:
   Date    Ver Made By Description
   ------- --- ------- -----------------------------
   mm/dd/yy 1.0 XXX    Created
```

```
  mm/dd/yy 2.0 yyy    Changed something
  """
  vInt = pParam1 + pParam2
  return vInt

print(myFunc.__doc__)
```

If we execute it, we get this:

```
C:\>python py7.txt
Version : 2.0
  Purpose      : The purpose comes here
  Created      : mm/dd/yyyy
  Author       : XXX
  Last Changed : mm/dd/yyyy
  Change History:
  Date    Ver Made By Description
  ------- --- ------- -----------------------------
  mm/dd/yy 1.0 XXX    Created
  mm/dd/yy 2.0 yyy    Changed something
```

The documentation is helpful in many ways for documenting your programs and checking them later. The documentation comes in very handy when you write modules and classes (explained later in this installment).

Annotations

PL/SQL functions are strongly and unmistakably typed; that is, you have given the datatype of input parameters and return values explicitly at the time of defining the function. When you check the function later, all you need to do is to describe the function in SQL*Plus as shown below:

```
SQL> describe myFunc
```

However, there is no such facility in Python. What if you need to know what the datatypes are? There is a special attribute called annotations for functions. This allows you to document the datatypes of the input parameters and return values. This is the generalized syntax:

```
def FunctionName (ParameterName: DataType, ParameterName: DataType) -> ReturnDataType
```

When you want to see the annotations, a special attribute of the function, `__annotations__`, holds the value. Let's see a simple example where a function accepts two parameters of `int` type and returns a value of `int` type as well.

```
# py8.txt
def myFunc (pParam1 : int, pParam2 : int) -> int:
   vInt = pParam1 + pParam2
   return vInt

print(myFunc(1,2))

print("Let's see the annotations")
print(myFunc.__annotations__)
```

When we execute it, we get this:

```
C:\>python py8.txt
3
Let's see the annotations
{'pParam1': <class 'int'>, 'pParam2': <class 'int'>, 'return': <class 'int'>}
```

It shows the `class`, which is analogous (but not exactly the same as) the datatype, of the input parameters (`int` in this case) and the return value (also `int`). However, please note a very important property of annotations. This is merely a suggestion; nothing else. It's not binding. There is nothing that prevents you from changing the datatype of the Python code after declaring the parameter annotations. Here is a modified version of the code shown earlier, but instead of integers we pass strings.

```
# py8a.txt
def myFunc (pParam1 : int, pParam2 : int) -> int:
   vInt = pParam1 + pParam2
   return vInt

print(myFunc("My","World"))

print("Let's see the annotations")
print(myFunc.__annotations__)
```

When you execute it, this is the output:

```
C:\>python py8a.txt
MyWorld
Let's see the annotations
```

```
{'pParam1': <class 'int'>, 'return': <class 'int'>, 'pParam2': <class 'int'>}
```

Not only does it execute just fine; it's grossly misleading. Its annotations show that the input and outputs are `int`; but they are actually strings. So, annotations are merely suggestive, not definitive, and--do not forget--optional.

Global and Local Variables

In PL/SQL packages, or even in procedures, you can define variables where the scope of that variable is important. Take this simple procedure for instance. Here a variable named `v1` is defined in two places--inside and outside the function--and assigned values in two places. Which value is shown where? Let's see:

```
-- pl9.sql
declare
  v1    number;
  function myFunc (
    p1 number,
    p2 number
  )
    return number
    is
    v1 number;
  begin
    v1 := p1+p2;
    dbms_output.put_line('inside the function');
    dbms_output.put_line ('v1='||v1);
    return v1;
  end;
begin
  dbms_output.put_line('outside the function');
  v1 := 10;
  dbms_output.put_line ('v1='||v1);
  dbms_output.put_line ('the output of the function is '||myfunc(20,30));
end;
/
```

Executing it produces this:

```
outside the function
v1=10
inside the function
```

```
v1=50
the output of the function is 50
```

The values are differently assigned and maintained. So, when you reference a variable named `v1`, you have to pay attention to its scope, that is, where it is defined. That is where the value will be changed. In the case above, values set inside the function will be different from the value outside. It's exactly the same way in Python:

```
#py9.txt
def myFunc (p1, p2):
  v1 = p1 + p2
  print('inside the function')
  print ('v1 = ', v1)
  return v1

print('outside the function')
v1 = 10
print('v1=', v1)
print('The output of the function is ',myFunc(20,30))
```

Executing it produces this:

```
C:\>python py9.txt
outside the function
v1= 10
inside the function
v1 =  50
The output of the function is  50
```

In fact, the scope of the local variables is strictly inside the function. It's completely independent; so they can be even different datatypes.

```
#py9a.txt
def myFunc (p1, p2):
  v1 = p1 + p2
  print('inside the function')
  print ('local v1 = ', v1)
  return v1

print('outside the function')
v1 = 10
print('global v1=', v1)
p1 = 'Hello'
```

```
p2 = 'World'
print('The output of the function is ',myFunc(20,30))
print('Global p1 and p2 = ',p1,p2,sep=' ')
```

Executing it produces this:

```
C:\>python py9a.txt
outside the function
global v1= 10
inside the function
local v1 =  50
The output of the function is  50
Global p1 and p2 =  Hello World
inside the function
v1 =  50
The output of the function is  50
```

The scope is something you should pay attention to. The scope of a variable is determined at the time of its first appearance. Since there is no such thing called "declaration of variables" (variables are declared at the time of the assignment of values), whether a variable is local or global depends on when it was first referenced. Consider the following Python code.

```
#py10a.txt
1 v1 = 100
2 def myFunc (p1, p2):
3    print ('v1=',v1)
4    v2 = p1 + p2
5    print('inside the function')
6    print ('v2=', v2)
7    return v2
8
9 print('The output of the function is ',myFunc(20,30))
```

What do you think the output will be? On line 9, we call the function, which in turn goes to line 2 where the function is declared. Immediately inside the function, we print the variable `v1`; but wait, there is no variable `v1` defined inside the function at that point. In fact, there is no variable called `v1` in that function. Will this `print` statement (line 3) work, then? Let's see:

```
C:\>python py10a.txt
v1= 100
inside the function
v2= 50
The output of the function is  50
```

It may defy logic, but it did work. The fact is that there is no variable called `v1` at all in the function; therefore, whenever there is a reference to variable named `v1`, Python will check if there is a global variable of that name. In this case, there is a global variable called `v1` (line 1); therefore, Python assumes that's what you meant on line 3.

However, what happens if we have a local variable called `v1` *as well* in the function?

```
#py10b.txt
1 v1 = 100
2 def myFunc (p1, p2):
3    print ('v1=',v1)
4    v1 = p1 + p2
5    print('inside the function')
6    print ('v1=', v1)
7    return v1
8
9 print('The output of the function is ',myFunc(20,30))
```

When we execute this code, here is the output:

```
C:\>python py10b.txt
Traceback (most recent call last):
File "py10b.txt", line 9, in <module>
  print('The output of the function is ',myFunc(20,30))
File "py10b.txt", line 3, in myFunc
  print ('v1=',v1)
UnboundLocalError: local variable 'v1' referenced before assignment
```

What happened? Why didn't it work this time? The error is pretty clear; it's in line 3.

```
  print ('v1=',v1)
UnboundLocalError: local variable 'v1' referenced before assignment
```

This is because `v1` is a local variable in this function, defined in line 4. However, before we defined it, we referenced it in line 3; hence, the error. Pay particular attention to this behavior. In summary, if you reference a local variable with the same name as a global variable, you must have declared a local variable before referencing it in the function. If you don't have have a local variable of the same name, a reference to that will not fail; it will succeed. The global variable will be referenced instead.

Argument Array

Suppose you are writing a small function to add all the numbers. There is a `sum` function already available; but assume it wasn't and you write your own function. The problem is that you have to accept a series of numbers as arguments and--worse yet--the number of arguments is not known at the declaration time of the function. For instance, the name of function is `mySum`. To get the sum of three numbers--1, 2 and 3--you would call this:

```
mySum (1,2,3)
```

It has three arguments. To get the sum of four numbers--1, 2, 3, and 4--you would call this:

```
mySum (1,2,3,4)
```

This time, there are four arguments, and so on. The number of arguments is not known at the time of the definition of the function. How would you write the function then? This is something for which there is no PL/SQL equivalent. Readers familiar with C language may remember pointers by reference. C accepts an array of numbers or strings. However, you have to use an array, not pure primitive datatypes such as numbers. Python excels in this case. It allows an array of arguments as well, but since it doesn't require a prior declaration, arguments passed will be implicitly be treated as an array. The trick is to prefix the parameter name with an asterisk Let's see an example of the `mySum` function that accepts any amount of numbers and returns the sum:

```
#py11.txt
1 def mySum(*numList):
2    tot = 0
3    for n in numList:
4       print n
5       tot = tot + n
6    return tot
7
8 print ('Total=',mySum(1,2,3))
```

Notice the * before the parameter name, `numList`, which indicates that it is an array. Therefore in line 3, we can extract each element of the array and print them as a demo (line 4). We add each element to a variable called `tot` and finally return the variable. When we execute it, we get this:

```
C:\>python py11.txt
1
2
3
Total= 6
```

You can pass any number of parameters in this case. This is a very powerful feature of Python functions.

Named Argument

Remember passing options in a UNIX shell script? You can use something as simple as `argument1=value1 argument2=value2`, and so on. Sometimes you may need to write a similar program in Python as well. Not only will you not know the number of arguments in advance; you may not know even the argument names while declaring the parameters as well. Once again, there is no equivalency in PL/SQL. Fortunately it's a breeze in Python. You can prefix a double asterisk (**) before the name of the parameter, which indicates a dictionary collection in Python. Remember the dictionary datatype? If you don't, go back to Part 1 of this series to refresh your memory. In summary, a dictionary in Python is a set of key-value pairs.

Here is a small program where we accept a list of key-value pairs and check if they are twins, that is, whether the value is the same as the key.

```
#py12.txt
1 def checkTwins (**wordList):
2    words = sorted(wordList.keys())
3    for w in words:
4       print (w,':',wordList[w])
5       if (w==wordList[w]):
6          print('Twins Found')
7
8 checkTwins (
9        firstWord="first",
10        secondWord="second",
11        thirdWord="thirdWord"
12 )
```

We put the entire dictionary in a variable called `words` (line 2) after sorting it (sorting is not necessary; but I did it to refresh your memory on the `sorted` function). Then we iterate over each pair in that variable (line 3) and print each pair (line 4). In line 5, we compare the value and the key and if they are the same, we print the fact that they are twins. When we execute the program, we get the following output:

```
C:\>python py12.txt
firstWord : first
secondWord : second
thirdWord : thirdWord
Twins Found
```

Notice how the keys and their corresponding values are printed.

Stored Functions

So far, we have talked about defining functions right there in the Python script, analogous to declaring inline PL/SQL functions and procedures. However, if you use specific code a lot, you may want to create a stored procedure (or function) that is stored in the database and called as often as needed without redefining it. In Python, you can also store functions; but since there is no database, these are stored in files in groups called *modules*. A module is analogous to a stored package in PL/SQL. Just like packages have functions and procedures, modules contain functions.

Let's see an example of a simple module called `intModule` that has two functions: `getIntRate()` and `calcInt()`. To create this module, simply create a file named `intModule.py` with the following content:

```
#intModule.py
def getIntRate (pAccType):
  if (pAccType in ('Savings','S')):
    vIntRate = 10
  elif (pAccType in ('Checking','C')):
    vIntRate = 5
  elif (pAccType in ('MoneyMarket','M')):
    vIntRate = 15
  else:
    vIntRate = 0
  return vIntRate

def calcInt (pPrincipal, pAccType):
  vIntRate = getIntRate (pAccType)
  vInt = pPrincipal * vIntRate / 100
  return vInt

vAllowed = True
```

This is the "module," the equivalent of a package in PL/SQL. The package name equivalent is the module name, which is `intModule` (note the case). There is no such thing called "creating" the package. Python parses it at run time. The only requirement is that you should have this file at the same location as the program calling it and some other predefined locations. With this module set, let's write Python programs that will call functions defined in this module.

The last line of the module shows how we can also define variables that can be used by the callers.

```
#py13.txt
```

```
1 import intModule
2
3 def getNewBal (pAccType, pOldBal):
4    vInt = intModule.calcInt(pOldBal, pAccType)
5    return vInt + pOldBal
6
7 if (intModule.vAllowed == True):
8    print('New balance = ', getNewBal('Savings', 1000))
9 else:
10    print('Not allowed to see')
```

Note the very first line where we import the module. It's similar to the `include` lines of a C program. It means look for functions defined in a file called `intModule.py`, which we placed in that directory. To reference specific functions of this module, we have to prefix them by the module name, as we have done in line 4. We have used the `calcInt` function in the `intModule` module. We also saw how we can reference a variable defined in the module named `vAllowed` (line 7). This variable is the equivalent of the PL/SQL `package` variable.

You can also import multiple modules, separated by a comma:

```
import module1, module2, ....
```

Where should these modules (which are just plain Python code text files) be located? They can be in the present directory; but that is not practical in the typical development environment. You may want to place all your modules files in a single `tools` directory. In addition, Python comes with many built-in modules. These may be located in the `/usr/bin/python` directory. Python also looks for a special environment variable, `PYTHONPATH`, which is searched for the modules as well. You can mention the search path at the beginning of the code.

```
#!/usr/bin/python
import help
```

In code development, you may want to put all your utilities in a single module; but you may not need all the functions defined in that module. Instead of loading the entire module (and, therefore, consuming memory), you can choose to import only specific functions. To import only the function `getIntRate` from `intModule` module, write the following:

```
from intModule import getIntRate
```

If you want to import everything from that module, use a wildcard character, which is an asterisk (*).

```
from intModule import *
```

One of the best parts of using this approach is that you don't need to have name of the module prefixed. So, the earlier program, in which you had to compute the new balance, will look like the following. Note how we didn't prefix the module name in line 4.

```
# py14a.txt
1 from intModule import *
2
3 def getNewBal (pAccType, pOldBal):
4    vInt = calcInt(pOldBal, pAccType)
5    return vInt + pOldBal
6
7 if (vAllowed == True):
8    print('New balance = ', getNewBal('Savings', 1000))
9 else:
10    print('Not allowed to see')
```

Now you might ask a question: what if there is a conflict? For example, you have a function in the same name locally defined as well:

```
#py14b.txt
1 from intModule import *
2
3 def calcInt (pOldBal, pAccType):
4    return -1
5
6 def getNewBal (pAccType, pOldBal):
7    vInt = calcInt(pOldBal, pAccType)
8    return vInt + pOldBal
9
10 if (vAllowed == True):
11    print('New balance = ', getNewBal('Savings', 1000))
12 else:
13    print('Not allowed to see')
```

Here, at line 3 we defined a function also named `calcInt`, which returns -1. When we call the function `calcInt` in line 7, which function will be called: the local one or the one in the module? The answer follows what would logically happen in a PL/SQL program as well. The local one will take precedence. Let's execute the code and find out:

```
C:\>python py14b.txt
New balance = 999
```

The new balance was 999, since the local `calcInt` returned -1, which, when added to the balance of 1000, produced 999.

Listing Module Contents

Listing module contents is the equivalent of using the `describe` command in a PL/SQL package, which shows the different functions, procedures, and variables defined in the package. The equivalent in Python is the `dir` function.

Suppose after importing the `intModule` module, we want to see the various functions and variables in that module.

```
>>> import intModule
>>> dir(intModule)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'calcInt', 'getIntRate', 'vAllowed']
```

In addition to the functions we defined--`calcInt` and `getIntRate`--and the variable--`vAllowed`--there are other predefined functions in the module as well, which offer valuable information on the module. Say, for instance, that you want to find out the exact file name (actually the location) of the module. Simply execute the `__file__` function:

```
>>> intModule.__file__
'C:\\intModule.py'
```

There are many predefined modules in Python to help you do your job. Let's take a very common one: `math`, for mathematical functions.

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
```

You will see the various functions available in the `math` package. Note that there is no `__file__` function, because it was in our user-defined function. `math` is a built-in function that comes with Python instead of being loaded from a file.

By the way, there is also something called **package** in Python, which is similar to what directories are to subdirectories; but for the sake of keeping the content simple, I prefer to skip discussing that.

Classes

Those familiar with or even exposed to the concepts of object-oriented programming will "get" this immediately. Suppose you want to define a type of data such as an employee, which has some attributes such as first name, last name, dept. number, salary, and so on. You can define a PL/SQL "type" for this. Once a type is created, you can declare variables of that type. You can define functions or procedures of that type as well, which can be called by PL/SQL programs to manipulate data. Here is an example of a type created for employees. Remember, this is created in the database; you can't just define a type in PL/SQL.

```
-- pl19.sql
create or replace type employee as object
(
  firstName  varchar2(30),
  lastName   varchar2(30),
  deptNo     number(10),
  salary     number(10),
  member procedure displayEmp
);
/
```

Notice how we defined a "member procedure" of the type to display the values. Once the type is defined, we can create the "body" of the type where the actual code of the member procedure is created.

```
create or replace type body employee as
  member procedure displayEmp is
  begin
     dbms_output.put_line('Name :'||
        firstName||' '||lastName);
     dbms_output.put_line('DeptNo :'||deptNo);
     dbms_output.put_line('Salary :'||salary);
  end;
end;
/
```

Now that the type and body are created, we can define variables of that type in our PL/SQL code. In the following example, we define two variables--`employee1` and `employee2`--of this type in the `declare` section. This is also known as "instantiating" the type. In the main block, we assign values to various attributes of the variables and then call the `displayEmp` procedure to display the attributes' values for each "instance" of the object.

```
declare
   employee1 employee;
   employee2 employee;
begin
  employee1 := employee ('Martin','King',1000,10);
  employee2 := employee ('Scott','Tiger',2000,20);
  -- display the first employee
  employee1.displayEmp;
  -- display the second employee
  employee2.displayEmp;
end;
/
```

When we execute the code, we get the following output:

```
Type created.

Type body created.

Name :Martin King
DeptNo :10
Salary :1000
Name :Scott Tiger
DeptNo :20
Salary :2000

PL/SQL procedure successfully completed.
```

The equivalent of an object in Python is called a *class*. The general syntax of defining a class is as follows:

```
class ClassName:
  def MemberFunction:
```

Note the indentation and the colon (:) that defines the blocks in Python. There is no `begin...end` block as in PL/SQL. Also, there is no creation of classes beforehand. You define the class as it is required. Of course, you can put the class definition in a module using the technique shown earlier, which allows you to call the class without defining it every time.

In addition, Python also has a function called `__init__`, which is the "constructor" function of the class, that is, the function that gets executed when the class is instantiated. The `self` attribute refers to the specific instance of the class. It's the same as the `self` attribute in PL/SQL. The example above didn't have `self` attribute because it was not needed. In Python, it is somewhat mandatory, as you can see later.. Here is the same example in Python.

```
#py17.txt
1 class employee:
2     def __init__ (self, firstName, lastName, salary, deptNo):
3       self.firstName = firstName
4       self.lastName = lastName
5       self.salary = salary
6       self.deptNo = deptNo
7     def displayEmp(self):
8       print("Name  : ", self.firstName+" "+self.lastName)
9       print("DeptNo: ", self.deptNo)
10       print("Salary: ", self.salary)
11
12 employee1 = employee("Martin","King",1000,10)
13 employee2 = employee("Scott","Tiger",2000,20)
14
15 # display the first employee
16 employee1.displayEmp()
17
18 # display the second employee
19 employee2.displayEmp()
```

Let's dissect the code line by line. In line 1, we define a class. In line 2, we define a special type of function that is part of all classes. This is called a constructor function. I will explain it later. The first parameter is a special one called `self`, which means a pointer to an object of that class. Since this is a function, it has other parameters. The parameters are all the attributes of the class--first name, last name, salary, and department number--in that order. In lines 3 through 6, we assign values to the attributes. In line 7, we declare a function we want to use to display the values. This completes the definition of the class.

Once the class is defined, we create an object of that class in line 12. We call this object `employee1`, which is also called an "instance" of the class. In this instance, we assign various values: Martin as the first name, King as the last name, 1000 as salary, and 10 as department number. Similarly we create another object--`employee2`--which is also called another instance of the same class but with different values, as shown in line 13. Imagine these two are two variables where the datatype is the class called `employee`. In fact datatypes, are nothing but predefined classes.

Once the values are assigned, we can call the function `displayEmp`. However, this is where the object seems to vary from a variable. An object is an "instance" or a copy of the class. Therefore, it has the functions defined in the class. These functions are called methods. One method we defined is `displayEmp()`, which is what we call for the first object, `employee1`, as shown in line 16. Going back to line 7, you can see that `displayEmp` displays the values of variables `self.firstName`, `self.lastName`, and so on. Remember, "self" means the instance of the class itself. In this case, it is `employee1`. So, `employee1.displayEmp()` will display whatever was assigned to the `employee1` object. Line 19 shows the same display for the other object we defined: `employee2`. Let's see the output when we execute this code:

```
C:\> python py17.txt
Name  :  Martin King
DeptNo:  10
Salary:  1000

Name  :  Scott Tiger
DeptNo:  20
Salary:  2000
```

The output shows the values we assigned when we created objects of the class. We will see how to use the objects; but first let's revisit the special method called `self`. As I mentioned, it's called a constructor function. There is a reason for that. This function is called every time an object of that class is instantiated. In plain English, every time you create an object of that class, the constructor function is executed. You defined two objects of this class: `employee1` and `employee2`; so the constructor function is executed twice. How can we prove that? Simple. Let's define a variable and increment it by 1 in the function. Here is some simple code to do that:

```
#py18.txt
1  class employee:
2     vEmpCount = 0
3     def __init__ (self, firstName, lastName, salary, deptNo):
4        self.firstName = firstName
5        self.lastName = lastName
6        self.salary = salary
7        self.deptNo = deptNo
8
9        employee.vEmpCount = employee.vEmpCount + 1
10
11  employee1 = employee("Martin","King",1000,10)
12  employee2 = employee("Scott","Tiger",2000,20)
13
14  # display the total number of employees
15  print("Total Employees: ", employee.vEmpCount)
```

In line 2, we defined a variable and set it to 0. We increment that variable by 1 inside the `__init__` method. In lines 11 and 12, we defined two objects of that class, and in line 15 we display the counts. When we execute the code, here is the output:

```
C:\> python py18.txt
Total Employees:  2
```

The total came out as 2, as expected, since we defined two objects and, therefore, the method was executed two times. This is a powerful property of the constructor function.

One common question people ask is why we need to use "self" in `py17.txt`. We have already used it in the `__init__` function; so why use it in `displayEmp()`? Well, consider this line:

```
print("Name  : ", self.firstName+" "+self.lastName)
```

If you omit `self`, it becomes the following:

```
print("Name  : ", firstName+" "+lastName)
```

In this case, what does `firstName` actually mean? Is it an attribute of a class called `employee`? Or is a variable? It's not clear; so Python will not be able to parse it. Therefore, you have to use "self" as a prefix to make the meaning crystal clear to Python.

Methods are ways to manipulate the data in instances of the object. Take a small example: we want to give a raise. So, we will define a method called `giveRaise` in the class. This method will accept only one parameter--the raise percentage--and will return the new salary.

```
#py19.txt
class employee:
  def __init__ (self, firstName, lastName, salary, deptNo):
    self.firstName = firstName
    self.lastName = lastName
    self.salary = salary
    self.deptNo = deptNo
  def displayEmp(self):
    print("Name  : ", self.firstName+" "+self.lastName)
    print("DeptNo: ", self.deptNo)
    print("Salary: ", self.salary)
  def giveRaise (self, raisePercent):
    if (raisePercent > 100):
      raise RuntimeError('Cannot give more than 100% raise')
    self.salary = self.salary * (1+(raisePercent/100))
    return self.salary
employee1 = employee("Martin","King",1000,10)
employee2 = employee("Scott","Tiger",2000,20)

# display the first employee before raise
```

```
print('Before the raise')
employee1.displayEmp()

# give a raise of 10%
employee1.giveRaise(10)

# display the first employee again
print('After the raise')
employee1.displayEmp()
```

Executing it produces this:

```
C:\>python py19.txt
Before the raise
Name  :  Martin King
DeptNo:  10
Salary:  1000
After the raise
Name  :  Martin King
DeptNo:  10
Salary:  1100.0
```

Note the new salary: it became 1100 from 1000. Had you given more than 100 in the raise, for example, 101 percent, the code would have thrown an error:

```
RuntimeError: Cannot give more than 100% raise
```

Suppose we have a slightly more-complicated requirement, that is, the actual raise is not just whatever the user entered but is based on the department. The programmer of this code does not like folks from department 10; so she wants to cut down the raise percent entered by 50 percent only for department 10. She can code this little login in the method:

```
#py19a.txt
class employee:
  def __init__ (self, firstName, lastName, salary, deptNo):
    self.firstName = firstName
    self.lastName = lastName
    self.salary = salary
    self.deptNo = deptNo
  def displayEmp(self):
    print("Name  : ", self.firstName+" "+self.lastName)
    print("DeptNo: ", self.deptNo)
    print("Salary: ", self.salary)
```

```
    def giveRaise (self, raisePercent):
        if (raisePercent > 100):
            raise RuntimeError('Cannot give more than 100% raise')
        actualRaisePercent = raisePercent
        if (self.deptNo == 10):
            actualRaisePercent = raisePercent * 0.5
        self.salary = self.salary * (1+(actualRaisePercent/100))

        return self.salary
employee1 = employee("Martin","King",1000,10)
employee2 = employee("Scott","Tiger",2000,20)

# display the first employee before raise
print('Before the raise')
employee1.displayEmp()

# give a raise of 10%
employee1.giveRaise(10)

# display the first employee again
print('After the raise')
employee1.displayEmp()

# display the second employee before raise (dept 20)
print('Before the raise')
employee2.displayEmp()

# give a raise of 10%
employee2.giveRaise(10)

# display the first employee again
print('After the raise')
employee2.displayEmp()
```

When you execute it, you get this:

```
C:\> python py19a.txt
Before the raise
Name  :  Martin King
DeptNo:  10
Salary:  1000
After the raise
Name  :  Martin King
```

DeptNo: 10
Salary: 1050.0
Before the raise
Name : Scott Tiger
DeptNo: 20
Salary: 2000
After the raise
Name : Scott Tiger
DeptNo: 20
Salary: 2200.0

Note how the raise was only 5 percent for department 10, even if the user entered 10 percent; but the raise was the full 10 percent for department 20.

I haven't seen widespread usage of object types in PL/SQL, but classes are quite common in Python. We will explore classes much more in a subsequent article of this series. We will close this discuss with a special piece of information. The word *method* is not unique to a class in Python. Other types of objects may have a method as well. For instance the "list" datatype has methods. Do you remember from the earlier articles of this series that you saw the `append()` operation on a list? Well, `append()` is nothing but a method of the list datatype.

File I/O

File I/O is easier to show in the Python interactive command line interpreter. You first need to define a variable of class `TextIOWrapper`. All you do is open a file. Suppose we want to create a file called `employee.txt`. This is how we open it:

```
>>>  filevar = open ("employees.txt","w+")
```

The second parameter is the mode. `"w+"` indicates that the file is opened in read/write mode and if it doesn't exist, then Python should create a file of that name. Then write something to it:

```
>>>  filevar.write("Martin 1000 10\n")
```

After writing a few lines, you need to close the file:

```
>>>  filevar.close()
```

Later, if you want to add more lines, you need to open the file once again but in append mode by specifying `"a"` instead of `"w+"`, add the lines, and then close the file.

>>> filevar = open ("employees.txt","a")

```
>>> filevar.write("Scott 2000 20\n")
>>> filevar.close()
```

Later, when you want to read the file, just open the it using the `"r"` mode, as shown below:

```
>>>  filevar = open ("employees.txt","r")
```

At this point, you can read the entire file in one shot:

```
>>> filevar.read()
'Martin 1000 10\nScott 2000 20\n'
```

But that might not be very practical, especially when reading data files record by record. You can read the files line by line by using the `readline()` method of that class.

```
>>>  filevar.readline()''
```

But as you can see, Python didn't read anything. Why? It's because the file pointer is already at the end, since you read the file entirely earlier. To move the file pointer to the beginning, use the following:

```
>>>  filevar.seek(0,0)
```

This moves the pointer to the 0-th byte counting from the 0-th byte. Now read a line:

```
>>> filevar.readline()
'Martin 1000 10\n'
```

It got the first line. Give the same command once more:

```
>>> filevar.readline()
'Scott 2000 20\n'
```

And it read the second line and so on. Note that the method is called `readline()` not `readlines()`; `readlines()` presents all the lines in one shot with `"\n"` separating the lines. But it's unlikely that you will read files interactively. You will probably read all the lines of the file. You can use the file variable, `filevar`, as a collection and read from it. However, first move the pointer back to the beginning.

```
>>>  filevar.seek(0,0)
```

Then read the lines one by one:

```
>>> for vLine in filevar:
...     print(vLine)
```

...
Martin 1000 10
Scott 2000 20

That's it; file handling is as simple as that.

One of the practical use cases of file handling is reading and writing JSON documents, which are used in data interchanges. We can manage JSON files directly. First we have to import the `json` module.

```
>>> import json
```

Then we open the file as usual for writing:

```
>>> filevar = open ("employees.txt","w+")
```

Let's define a simple collection:

```
>>> vJson1 = [1,'A','z']
```

The function `dumps()` in the module converts it to a JSON representation.

```
>>> vJson1 = json.dumps(vJson1)
```

A slightly differently named method, `dump()`, puts the data in the file. Note the difference: `dump` as opposed to `dumps`.

```
>>> json.dump (vJson1, filevar)
```

Let's close the file and open it for reading:

```
>>> filevar.close()
>>> filevar = open ("employees.txt","r")
```

When it is open, we can use a predefined method called `load()` to load the data from the file to a variable named `vJson2`.

```
>>> vJson2 = json.load(filevar)
```

Let's check the value of this variable.

```
>>> vJson2
'[1, "A", "z"]'
```

And, let's confirm that these two variables are the same:

```
>>> vJson1 == vJson2
True
```

They are the same. We loaded the value from a variable to a file and loaded the data from the file to another variable. It's the same data.

Let's consider another possibility: what if the file doesn't exist? Rather than failing the program with some error, you can use the `try-except` clause to let the program execute successfully but not have a value in the `vLine` variable.

```
try:
   fileVar = open('employees.txt', 'r')
   vLine = fileVar.read()
   fileVar.close()
except IOError:
   vLine = ''
```

If the file doesn't exist, `vLine` will be null. We can also catch if a string is of zero length. While we are at it, why don't we also build some intelligence into it so that we can add lines as comments. Suppose we have these requirements:

- When the line is of zero length, we should stop reading the file.
- When the first character is "#," the line should be considered a comment and, hence, should be skipped from being read.

Here is how we can code these requirements:

```
fileVar = open('employees.txt', 'r')
while True:
   vLine = fileVar.readline()
   if len(vLine) == 0:
      break
   if vLine[0] == '#':
      continue

   # put any more processing logic here
   print(vLine)
```

Putting It All Together

Let's now put together a real application using all the concepts we learned so far. The US Census Department publishes many types of data, one of which is the percent of households with high-speed internet access in the 50 US states and the District of Columbia. Here is how the raw text file of the data looks with a tab character between the values and a newline character ending each line. The file name is `high_speed_internet_data_by_state.txt`.

Alabama 65.8

Alaska 81.4

Arizona 75.5

Arkansas 63.5

California 80.0

Colorado 81.2

Connecticut 80.5

Delaware 75.5

District of Columbia 73.4

Florida 75.8

Georgia 73.4

Hawaii 80.6

Idaho 73.6

Illinois 75.5

Indiana 71.4

Iowa 74.2

Kansas 74.5

Kentucky 68.9

Louisiana 66.6

Maine 74.9

Maryland 80.1

Massachusetts 80.5

Michigan 72.9

Minnesota 78.3

Mississippi 59.1

Missouri 71.6

Montana 72.9

Nebraska 74.8

Nevada 76.3

New Hampshire 82.1

New Jersey 80.9

New Mexico 67.5

New York 76.5

North Carolina 72.4

North Dakota 74.7

Ohio 73.9

Oklahoma 69.2

Oregon 78.9

Pennsylvania 73.9

Rhode Island 76.5

South Carolina 68.1

South Dakota 71.6

Tennessee 68.2

Texas 73.0

Utah 81.7

Vermont 76.3

Virginia 77.2

Washington 81.9

West Virginia 66.2

Wisconsin 75.3

Wyoming 76.1

We want to use the file and do some computations to showcase what we learned so far. We will limit the computations to some simple ones such as the mean usage, the maximum and minimum usage, and the states who have the maximum usage. One thing we will have to pay attention to is the possibility of many states with the same number for internet usage; so we may have to print an array of names rather than just one name. We will read all the values into a variable in the program for faster manipulation. What type of variable do you think will be the best? This is a simple key-value pair; so a Python dictionary datatype will be perfect. Here is the program with inline documentation explaining each step.

```
# py20.txt
# open the file for reading
fileVar = open ("high_speed_internet_data_by_state.txt","r")

# define an empty dictionary variable to hold the values
keyVal1 = {}

# Now read the fle line by line
for vLine in fileVar:
  # the values in the line are delimited by tab; so split it on that character
  (key,val) = vLine.split('\t')
  # assign the value to that key
  keyVal1[key] = val

# We don't need to kep the file open anymore. Let's close it.
fileVar.close()

# For some reason, we are left with a \n at the end of the values. And, we
# want it to be float; not a string. So, we strip the \n and convert to float.
for i in keyVal1.keys():
  keyVal1[i] = float(keyVal1[i].strip('\n'))

# At this point we have a dictionary variable, keVal1, with all the data,
# we can confirm that by iterating through the values
for vState in keyVal1.keys():
  print('State ', vState, ' Highspeed Internet Use = ', keyVal1[vState])

# Let's do something with this data
# We will do mostly statistical analysis; so let's import the stats module
import statistics

# Next we want to do lookup of a bit different type. We saw how to get the value
# given a specific "key"; but how about given a specific value and we want to get
# the corresponding key? To do that we will create a function to make it easy
# for calling multiple times. This function accepts the value as parameter and
# searches and returns the corresponding key. Since there may be multiple keys
# with that value, we will create a list rather than a string
def getKey(vValue):
  tempArr = []
  for (k,v) in keyVal1.items():
    if (v == vValue):
      tempArr.append(k)
```

```
    return tempArr

# print a separator line
print ('-' * 40)
# Let's find the various statistical patterns
print('Mean usage = ', statistics.mean(keyVal1.values()))
print('Standard Deviation of usage = ', statistics.stdev(keyVal1.values()))
vVal = max(keyVal1.values())
print('Maximum usage = ', vVal, 'States = ', getKey(vVal))
vVal = min(keyVal1.values())
print('Minimum usage = ', vVal, 'States = ', getKey(vVal))
vVal = statistics.mode(keyVal1.values())
print('Most common usage = ', vVal, 'States = ', getKey(vVal))
```

When we execute this program, we will get this:

```
C:\>python py20.txt
State  Kentucky  Highspeed Internet Use =  68.9
State  Wisconsin  Highspeed Internet Use =  75.3
State  Florida  Highspeed Internet Use =  75.8
State  Texas  Highspeed Internet Use =  73.0
State  Idaho  Highspeed Internet Use =  73.6
State  South Dakota  Highspeed Internet Use =  71.6
State  Connecticut  Highspeed Internet Use =  80.5
State  Utah  Highspeed Internet Use =  81.7
State  Louisiana  Highspeed Internet Use =  66.6
State  Arkansas  Highspeed Internet Use =  63.5
State  Illinois  Highspeed Internet Use =  75.5
State  Georgia  Highspeed Internet Use =  73.4
State  New Jersey  Highspeed Internet Use =  80.9
State  Hawaii  Highspeed Internet Use =  80.6
State  Montana  Highspeed Internet Use =  72.9
State  Tennessee  Highspeed Internet Use =  68.2
State  Wyoming  Highspeed Internet Use =  76.1
State  New Hampshire  Highspeed Internet Use =  82.1
State  West Virginia  Highspeed Internet Use =  66.2
State  Oklahoma  Highspeed Internet Use =  69.2
State  Indiana  Highspeed Internet Use =  71.4
State  New York  Highspeed Internet Use =  76.5
State  Delaware  Highspeed Internet Use =  75.5
State  Nevada  Highspeed Internet Use =  76.3
State  Maryland  Highspeed Internet Use =  80.1
State  Alaska  Highspeed Internet Use =  81.4
```

```
State  California  Highspeed Internet Use =  80.0
State  Ohio  Highspeed Internet Use =  73.9
State  Massachusetts  Highspeed Internet Use =  80.5
State  Maine  Highspeed Internet Use =  74.9
State  Pennsylvania  Highspeed Internet Use =  73.9
State  District of Columbia  Highspeed Internet Use =  73.4
State  Colorado  Highspeed Internet Use =  81.2
State  Nebraska  Highspeed Internet Use =  74.8
State  New Mexico  Highspeed Internet Use =  67.5
State  Mississippi  Highspeed Internet Use =  59.1
State  Vermont  Highspeed Internet Use =  76.3
State  Alabama  Highspeed Internet Use =  65.8
State  North Carolina  Highspeed Internet Use =  72.4
State  Arizona  Highspeed Internet Use =  75.5
State  Missouri  Highspeed Internet Use =  71.6
State  Kansas  Highspeed Internet Use =  74.5
State  Iowa  Highspeed Internet Use =  74.2
State  Michigan  Highspeed Internet Use =  72.9
State  Washington  Highspeed Internet Use =  81.9
State  South Carolina  Highspeed Internet Use =  68.1
State  Virginia  Highspeed Internet Use =  77.2
State  North Dakota  Highspeed Internet Use =  74.7
State  Rhode Island  Highspeed Internet Use =  76.5
State  Oregon  Highspeed Internet Use =  78.9
State  Minnesota  Highspeed Internet Use =  78.3
----------------------------------------
Mean usage =  74.4078431372549
Standard Deviation of usage =  5.154525900109724
Maximum usage =  82.1 States =  ['New Hampshire']
Minimum usage =  59.1 States =  ['Mississippi']
Most common usage =  75.5 States =  ['Illinois', 'Delaware', 'Arizona']
```

We got the data as planned. Note how the last line shows multiple states. This is why we defined the function to return a list and not a string. Hopefully this program gives you an idea of using Python for data analysis. In the next article, we will go over more about data manipulations--using Python to directly access Oracle Database.

Summary

Here is a quick summary of PL/SQL language elements and analogous elements in Python.

Element	**PL/SQL**	**Python**

Procedures and functions	`function` *FunctionName*
	`(`
	Parameter1Name `in` *DataType*`,`
	Parameter2Name `in` *DataType*`,`
	`)`
	`return` *ReturnDatatype*
	`is`
	localVariable1 datatype`;`
	localVariable2 datatype`;`
	`begin`
	... function code ...
	`return` *ReturnVariable*`;`
	`end;`
		Function definition in Python follows this simple syntax:
		`def` *FunctionName* `(`*Parameter1Name*`,` *Parameter2Name*`, ...):`
		`...`
		... function code ...
		`return` *SomeValue*
		• Unlike PL/SQL, there is no difference between procedures and functions. Both types are called functions. A function may

A procedure definition in PL/SQL has this general syntax:

```
procedure ProcedureName(
    Parameter1Name in DataType,
    Parameter2Name in DataType,
    ...
) is
    localVariable1 datatype;
    localVariable2 datatype;
begin
    ... procedure code ...
end;
```

or may not return anything. If it does return something, you do not specify what datatype it returns at the time of definition.

- Function definitions start with the keyword `def` compared to `function` or `procedure` in PL/SQL.
- Like PL/SQL, the parameters shown above are optional; not all functions need parameters. The parameters

do not have datatypes listed at definition. Therefore, you can pass any type of data at runtime.

- Unlike PL/SQL, if you don't have any parameters, you still need to have the parentheses, for example, `def myFunc():`
- Unlike PL/SQL, there is no `begin ... end` block to designate the code for a function. The

indentation designates the code for the function.

- Like PL/SQL, the `return` statement is the last statement of the function code.

ParameterName DataType := DefaultValue

default value

Unlike PL/SQL, default values of parameters can be variables as well. Example:

```
defIntRate = 5
def calcInt(pPrincipal, pIntRate = defIntRate):
```

However, watch out for

a very important property. When the function is defined--not when the function is called--whatever the value of `defIntRate` was will be set as the default value of `pIntRate`. For instance, suppose this is the sequence:

```
defIntRate = 5

# define the function
def calcInt(pPrincipal, pIntRate = defIntRate):
...
```

```
# change the variable value
defIntRate = 10

# call the function
calcInt(100)
```

What will be taken as the value of `pIntRate`: 5 or 10? It will be 5, which was the value of `defIntRate` at the time of the function definition. At the time of the function call, the value of `defIntRate` was 10, but that value would not be taken.

You can also do the same in Python, as follows:

```
P1 (P2 = Val2, P1 = Val1)
```

Positional parameter specification in functions and procedures

If a PL/SQL procedure named `P1` has two parameters, `P1` and `P2` (in that order), you have to pass the parameter values at runtime in the same order, as shown below, where `Val1` and `Val2` are values for parameters `P1` and

`P2`, respectively:

`F1 (Val1, Val2)`

However, you can also use positional parameters by specifying the parameter names and their values in any other order by using the => operator, as shown below:

`F1 (P2 => Val2, P1 => Val1);`

`pIntRate =umber := None,` values of parameters to null at design time so that in the code you can check if the parameter value was passed or not.

Returning The function

`return` *SomeValue*; for a big difference is that you do not need to specify whether the function has to return something, and if it does, you do not need to specify the datatype of the return value at design time.

Documenting functions	`procedure MyFunc is`
	`/`
	`/*`
	`Documentation`
	`...`
	`*/`
	`begin`
	... function code here ...
	`end;`
		`def MyFunc:`
		`"""Documentation`
		`...`
		`"""`
		... function code here ...
		One huge difference is that this documentation becomes a property of the function and can be displayed by `print(MyFunc.__doc__)`.

Annotations	Not needed since you have to specify the datatypes of input and output	You have to mention the datatypes of parameters explicitly.
		`def`
		FunctionName
		`(`

values of functions at design time. You can see the datatype later using the `MyFunc` command in SQL*Plus.

ParameterName : *DataType* , *ParameterName* : *DataType*) -> *ReturnDataType*

This way, the Python interpreter knows about the datatypes of input and output values. You can see the datatypes by doing this:

```
print(MyFunc.__annotations__)
```

However, that's merely suggestive; you are allowed to change datatypes in any way in the design. For example:

```
def myFunc (pParam1 : int, pParam2 : int) -> int:
```

The code line above indicates the inputs and outputs are all of type int. However, you can call the function as follows:

```
myFunc("My","World"))
```

The inputs are of datatype string and, therefore, the output will also be string, not int, unlike what we specified as annotations. This is perfectly allowed. If we print the annotations, the output will still show the input and output as type int. Beware of this behavior.

The behavior. However, two important caveats are particularly important for PL/SQL developers.

Global variables defined outside the functions will be different from variables defined with the same name inside the functions.

- Since there is no such

> thing as declaration of variables, the variables come into existence when they are first referenced. If they are referenced inside a function first, they are local; otherwise, they are global. If you must be 100 percent clear about the scope, simply assign a value such as "None" to the variable first wherever you want

the scope to be.

- If a variable is not referenced at all in a function, but a variable of the same name exists in the program outside the function, then the variable is valid inside the function as well.

Unknown number of arguments -- for example, `average(n1, n2, n3, ...)`, the arguments are	Not possible in PL/SQL. Design time, you must know the exact number of the arguments and	Easy to do. `def mySum(*numList):` The code above expects the arguments

[illegible] datatypes [illegible] datatypes [illegible] arguments [illegible] design [illegible] example, `mySum(1,2,3,...)`. [illegible] [illegible] a function [illegible] code, [illegible] you [illegible] arguments [illegible] [illegible] [illegible] variable `numList` [illegible] default; [illegible] [illegible] cumbersome. the following construct:

```
for i in numList:
```

[illegible] [illegible] [illegible] arguments, [illegible] example, `def myFunc(**argumentList):` `argument2=Value2`, [illegible] The parameters variable [illegible] argumentList [illegible] passed a Python [illegible] dictionary datatype. [illegible] You designed, call [illegible] the [illegible] function pass as parameters follows: in any

```
myFunc(
```

order;

but the name of the parameters must be known.

```
key1=Val1,
key2=Val2
key3=Val3
)
```

Stored procedures stored functions the database for reuse later. Packages are good examples.

```
create or replace package pkg1 as
procedure p1;
procedure p2;
end;
/
create or replace package body pkg1 as
  ...
```

The analogous solution is to create a module. However, you don't actually create a module. Just place all the function definitions in a file called `myModule.py`, as shown below:

```
def p1:
  ...
  code ...
def p2:
  ...
```

```
procedure p1 is
begin
  ...
end;
procedure p2 is
begin
  ...
end;
end;
```

You can call the procedure this way:

```
begin
pkg1.p1;
end;
```

```
function code ...
```

In the Python code, import this module using this:

```
import myModule
```

Call the function by prefixing the module name:

```
myModule.p1
```

You can reference the variable defined in the module by prefixing the module name.

You can import multiple modules:

```
import module1, module2, ...
```

You can also import specific functions from a module rather than importing all functions.

```
from myModule import p1, p2
```

To import all functions, do this:

```
from myModule import *
```

In this case, the functions `p1` and `p2` do not need to be prefixed by module name. If you have a local function called `p1`, then the local function will take precedence.

Describing the package contents

`SQL> describe packageName`

Listing the contents (variables and functions in the module):

```
>>> dir(packageName)
```

Showing the filename of the module:

```
>>> myModule.__file__
```

Objects

`class typeName:`

`replace type` *`typeName ...`* `as object` *`(attributes of the object ...`*

```
attribute1 Datatype,
...
member procedure ProcName
);
/
```

Note the ":" and the indentation of the attributes.

```
class typeName:
... attributes ...
def memberFunc:
... code of the function ...
```

```
create or replace type body typeName as
member procedure ProcName is
begin
...
```

Instantiation:

```
varName = className (... attribute list ...)
```

Declaration:

```
varName typeName;
begin
varName := typeName (... attribute list ...);
```

Calling a method:

```
varName.methodName();
```

varName.methodName`();`

Various functions and procedures with `utl_file`

File operations file of the name `file_name`:

```
fileVar = open('file_name','w')
```

If `w` is replaced by `w +`, Python opens a new file if it doesn't exist. `r` opens the file in read-only mode. Examples:

`fileVar.write("...text to write ...")` to write

`fileVar.close()` to close the

open file

`fileVar.seek(lineNo, position)` to move the cursor to the specified position

`fileVar.readline()` to read the entire file

```
for vLine in fileVar:
    print(vLine)
```

In the last example, `vLine` is an array; so `vLine[0]` will show the first character of the line.

Quiz

Now let's have a small quiz to make sure you understand the concepts well. The answers are given at the end of the quiz.

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.