Learning Python for PL/SQL Developers: Part 2

by Arup Nanda

Part 2 of a five-part series that presents an easier way to learn Python by comparing and contrasting it to PL/SQL.

Conditions and Loops

I hope you enjoyed Part 1 of this series where you were introduced to the basics of Python vis-a-vis the PL/SQL language. In this article, we will explore slightly more-advanced topics such as decisions and loops. As in the case of the Part 1, you also have the ability to watch the video of the article as well as download a podcast. Check out the Quick Summary on page 22 at the end and go through the quiz to make sure you understand all concepts well.

Remember the typographical convention used for code fonts. Bold means it's your input. The rest is output or just plain code.

Like the earlier installment in the series, all Python code shown here is in a Python file. Unless explicitly mentioned otherwise, I name this file `myfirst.py`. Execute this in Python by passing it as an argument to the Python command-line interpreter as shown below.

```
C:\>python myfirst.py
```

What IF?

Let's start with the simplest of conditions, the `IF` statement. In PL/SQL, the general structure of the `IF` condition is:

```
IF <conditional expression> THEN
  <some statement>
ELSE
  <some statement>
END IF;
```

In Python, the syntax is still `IF` and `ELSE`; but there are two huge differences:

- The blocks between `IF`, `ELSE`, and `END IF` are not controlled by the syntactical elements but by positions or indentations (or spaces).
- The end of the syntactical elements is indicated by a colon (`:`).

Here is a general syntax. Remember, it's `if`; not `IF`; Python is case-sensitive. I have added line numbers to help in explanation.

```
1. if <conditional expression>:
2.    <some statement>
3.    <more statements>
4. else:
5.    <some statement>
```

Note the colon (`:`) at the end of the `if` and `else` lines (lines 1 and 4). That is needed. Also note the indentation on lines 2, 3, and 5. That is needed to tell Python what statements fall under the block. If you miss the indentation, Python will interpret those as outside the block. Let's examine an example. We want to compare two numbers `n1` and `n2` and display whether `n1` is less than `n2`.

PL/SQL

```
declare
  n1      number;
  n2      number;
begin
  n1 := 2;
  n2 := 3;
  if (n1<n2) then
     dbms_output.put_line('n1 is less than n2');
  else
     dbms_output.put_line('n1 is not less than n2');

  end if;
end;
/
```

The output is:

```
n1 is less than n2
```

Python

```
n1 = 2
n2 = 3
if (n1<n2):
    print('n1 is less than n2')
else:
    print('n1 is not less than n2')
print('This is the end of program')
```

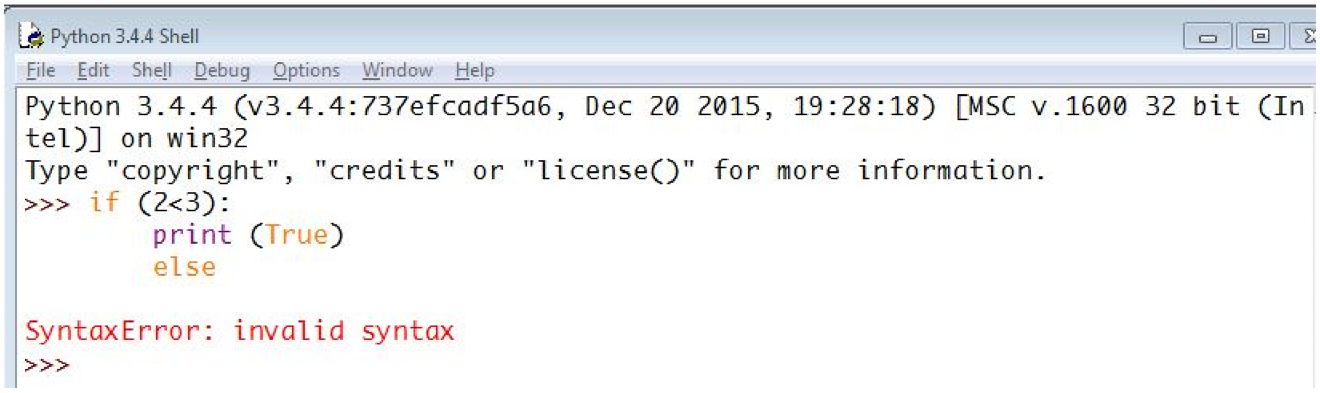
Here is the output:

```
C:\>python myfirst.py
n1 is less than n2
This is the end of program
```

I deliberately added the last line, `print('This is the end of program')`, to show you how Python interprets the indentation. Since that line was not indented, Python interpreted it as outside the `else` block and, hence, executed it. Had you wanted it to be part of the `else` block, you would have indented it along the previous line. This is a very important point to remember, and forgetting it could be a source of many bugs.

The need for indentation could be a major source of issues for PL/SQL developers new to Python because there is no such requirement in PL/SQL. This is one case where the Python GUI environment--the IDLE editor about which you learned in Part 1--can help. When you type the following, the cursor automatically moves to the proper indentation; so your chances of making a mistake are reduced. Similarly if you don't remove the indentation while entering `else`, IDLE throws an error, shown in Figure 1.

```
if (n1<n2):
```



```
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> if (2<3):
        print (True)
        else

SyntaxError: invalid syntax
>>>
```

Figure 1

In Figure 1, note how I entered `else` at the same indent as the `print()` statement, which wasn't legal; so IDLE threw a syntax error. So, apart from color coding, IDLE also helps you master Python a bit faster by watching out for the little nuances you are not familiar with.

The **`ELSIF`** statement in PL/SQL is **`elif`** in Python.

PL/SQL

```
declare
  n1      number;
  n2      number;
begin
  n1 := 5;
  n2 := 3;
  if (n1<n2) then
    dbms_output.put_line('n1 is less than n2');
  elsif (n1=n2) then

    dbms_output.put_line('n1 is equal to n2');
  else
    dbms_output.put_line('n1 is greater than n2');
  end if;
end;
```

The output:

```
n1 is greater than n2
```

Python

```
n1 = 5
n2 = 3
if (n1<n2):
   print('n1 is less than n2')
elif (n1==n2):
   print('n1 is equal to n2')
else:
   print('n1 is greater than n2')
print('This is the end of program')
```

If you want to introduce multiple levels of `IF`, you can, but with *indentation*. Suppose you want to modify the above program to display a message that the difference is less than 2. Here is how you would do it. Note the indentations for the nested `if` statement.

```
n1 = 5
n2 = 3
if (n1==n2):
    print('n1 is equal to n2')
elif (n1<n2):
    print('n1 is less than n2')
    if ((n2-n1)<=2):
        print('but it\'s not that much less')
else:
    print('n1 is greater than n2')
    if ((n1-n2)<=2):
        print('but it\'s not that much more')
print('This is the end of program')
```

Let's Make a Case

Unlike PL/SQL, there is no `CASE` statement in Python. You have to use `if...elif...else` statement blocks to accomplish the equivalent functionality. Here is a sample of PL/SQL with the `case` statement followed by the equivalent in Python.

```
declare
   n1      number;
begin
   n1 := 5;
   case
      when (n1<=25) then
        dbms_output.put_line('n1 is within 25');
      when (n1<=50) then
             dbms_output.put_line('n1 is within 50');
      when (n1<=75) then
        dbms_output.put_line('n1 is within 75');
      else
        dbms_output.put_line('n1 is greater than 75');
   end case;
end;
```

The same program can be written in Python as:

```
n1 = 5
if (n1<=25):
   print('n1 is within 25')
elif (n1<=25):
   print('n1 is within 50')
elif (n1<=75):
   print('n1 is within 75')
else:
   print('n1 is greater than 75')
```

Again, pay attention to the indentations for the blocks of statements and the colon (`:`) character.

For the Love of Loop

Consider the classic looping in PL/SQL using the `FOR ... LOOP ... END LOOP` construct. The general structure is:

```
FOR i in StartingNumber .. EndingNumber LOOP
  ...
END LOOP;
```

The Python equivalent is also called `for`; but there is no `LOOP` keyword and, consequently, no `END LOOP` either. As in the case of the `if` statement, the end of the conditional expression is marked by the colon (`:`) character. Also, like the `if` statement, the block of program statements to be repeated is identified by its indentation.

In Python, you can specify the starting and ending numbers using a function called `range()`. This function produces a range of values. Let's see an example. We want to produce a range of values from 1 to 10 and display them.

PL/SQL

```
begin
  for i in 1..10 loop
        dbms_output.put_line('i= '||i);
  end loop;
end;
```

Python

```
for i in range(1,11):
    print('i=',i)
```

Executing it produces this:

```
C:\>python myfirst.py
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
i= 7
i= 8
i= 9
i= 10
```

Note we passed `1,11` to the `range()` function, not `1,10`. This is important because `range()` creates numbers up to, but not including, the upper limit. If we want to step backwards, that is, start with 10 and then count backwards to 9, 8, and so on to 1, we can use the programs shown below. In Python the third parameter in the `range()` function identifies the step. A `-1` starts the count backwards.

PL/SQL

```
begin
   for i in reverse 1..10 loop
           dbms_output.put_line('i= '||i);
   end loop;
end;
```

Python

```
for i in range(10,0,-1):
    print('i=',i)
```

But the `range()` function is more powerful. If we had asked it to skip every other number, that is, to print odd numbers, it could have by using the following:

```
for i in range(1,11,2):
    print('i=',i)
```

Here is the output:

```
C:\>python myfirst.py
i= 1
i= 3
i= 5
i= 7
i= 9
```

There is no corresponding command in PL/SQL. We would have to use something like this:

```
begin
   for i in 1..10 loop
           if (mod(i,2) = 1) then
                   dbms_output.put_line('i= '||i);
           end if;
   end loop;
end;
```

Looping Through Arrays

In the previous case, we learned how to loop through a series of numbers using the `range()` function. But remember, the `range()` function simply returns an array of numbers which you iterate through. The loop is not limited to numbers, though. Remember you learned about arrays in Part 1. You can iterate through any array.

Here is an example. Suppose we have an array of week days and we want to iterate through them and print each one with its position in the array.

Python

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
```

Here is the output:

```
0 Mary
1 had
2 a
3 little
4 lamb
```

Looping While

The second type of loop we will cover is a variant of `FOR` but without a start and end--the `WHILE` loop. It allows you to loop as long as a condition is met (the condition could always be manipulated to be always true for a loop-forever loop). Here is an example of printing the 10 number.

PL/SQL

```
declare
   i number := 0;
begin
   while (i<11) loop
     dbms_output.put_line('i= '||i);
     i := i+1;
   end loop;
end;
/
```

```
i= 0
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
i= 7
i= 8
i= 9
i= 10
```

Python

```
i = 0
while (i<11):
    print ('i=',i)
    i = i+1
```

Note the super important differences from PL/SQL.

- There is a colon (`:`) after the condition in the line containing the `while` statement.
- The indentation of the `print` function is important. This indentation tells the program that it is a part of that `while` loop. All the statements with the same indentation will be executed as a part of that `while` statement.

Executing this program produces this:

```
C:\>python myfirst.py
i= 0
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
i= 7
i= 8
i= 9
i= 10
```

Breaking from the Loop

Suppose you want to put a condition in the loop that will make the program break away from the loop when the condition is satisfied. For instance, in the previous program, you want to break form the loop when the variable `i` is a multiple of 5. In PL/SQL, you can do that in two different ways:

- `exit when` *ConditionIsSatisfied*
- `if (`*ConditionIsSatisfied*`) then exit`

Functionally they are the same. In Python, the keyword `break` breaks the loop from executing and jumps out to the first line *after* the loop. We will see the approaches in both these languages.

PL/SQL

Approach 1

```
declare
   i       number := 1;
begin
   while (i<11) loop
           exit when mod (i,5) = 0;
           dbms_output.put_line('i= '||i);
           i := i+1;
   end loop;
end;
```

Approach 2

```
declare
   i       number := 1;
begin
   while (i<11) loop
           dbms_output.put_line('i= '||i);
           i := i+1;
           if mod (i,5) = 0 then
                   exit;
           end if;
   end loop;
end;
```

In either approach the output is the same:

```
i= 1
i= 2
i= 3
i= 4
```

```
PL/SQL procedure successfully completed.
```

While the output is same, the approaches are different and might behave differently. In the first approach, the condition for breaking is checked immediately at the start of the loop. In the second approach, it's evaluated after the counter is incremented. So you have to be careful in coding for each approach. The change in logic might be subtle, but it is important and can introduce bugs in the program.

Python

```
i = 1
while (i<11):
    print ('i=',i)
    i = i+1
    if (i%5 == 0):
        print ('Condition satisfied. Breaking')
        break
    print ('This is inside the loop but after break')
print ('This is after the loop')
```

Executing it produces this:

```
C:\>python myfirst.py
i= 0
This is inside the loop but after break
i= 1
This is inside the loop but after break
i= 2
This is inside the loop but after break
i= 3
This is inside the loop but after break
i= 4
Condition satisfied. Breaking
This is after the loop
```

Note that the following line is executed for each iteration of the loop.

```
print ('This is inside the loop but after break')
```

Why? It is *after* the `break` statement; so it should not have been executed until after the break condition, right?

The answer lies in the indentation. The indentation is less, that is, higher than, the `break` statement; so it is not part of the same code block as the `break` statement. It's executed inside the loop for each iteration. We also have this statement:

```
print ('This is after the loop')
```

This statement executes at the end, *after* the loop ends. Let's see a functionally same program in PL/SQL:

```
declare
   i       number := 1;
begin
   while (i<11) loop
          dbms_output.put_line('i= '||i);
           i := i+1;
           if mod (i,5) = 0 then
                exit;
                dbms_output.put_line('This is inside the loop but after exit');
           end if;
   end loop;
   dbms_output.put_line('This is after the loop');
end;
```

Executing it produces this output:

```
i= 1
i= 2
i= 3
i= 4
This is after the loop
```

Note how this line was *not* executed:

```
dbms_output.put_line('This is inside the loop but after exit');
```

Why was that, when the same line in Python was executed? It's simple.

```
if mod (i,5) = 0 then
   exit;
   dbms_output.put_line('This is inside the loop but after exit');
end if;
```

This statement was after the `exit` call, and the loop exits. It does not go to any other line after that. In contrast, the Python program has this:

```
    if (i%5 == 0):
        print ('Condition satisfied. Breaking')
        break
    print ('This is inside the loop but after break')
```

Note the indentation of the `print` function. It's not the same as the `break`. Therefore, it was not executed after the `break` statement. If you change the indentation as shown below, the `print()` statement is now at the same indentation as the `break` statement and, therefore, it will execute right after that, which is something you want:

```
    if (i%5 == 0):
        print ('Condition satisfied. Breaking')
        break
        print ('This is inside the loop but after break')
```

Here is the new Python program in its entirety.

```
i = 1
while (i<11):
    print ('i=',i)
    i = i+1
    if (i%5 == 0):
        print ('Condition satisfied. Breaking')
        break
        print ('This is inside the loop but after break')
print ('This is after the loop')
```

Here is the output:

```
C:\>python myfirst.py
i= 1
i= 2
i= 3
i= 4
Condition satisfied. Breaking
This is after the loop
```

Note how the annoying undesirable statement is not executing anymore. I hope that explains how important the indentation in Python is. It's more than just a tool to make the code more readable; it's actually functional.

The `break` statement applies to `FOR` loops as well. Let's see an example in both languages.

PL/SQL

```
begin
  for i in 1..10 loop
        dbms_output.put_line('i= '||i);
        if mod (i,5) = 0 then
              exit;
        end if;
  end loop;
end;
```

Python

```
for i in range(1,11):
   print ('i=',i)
   if (i%5 == 0):
      print ('Condition satisfied. Breaking')
      break
```

`Else` in `For` Loops

Python has another loop construct not available in PL/SQL. In a `FOR` loop, you can specify an `ELSE` statement block. This `ELSE` block is executed only if the loop is broken or completes without a match. It's easier to show that via an example. In this following program, I will ask the user to enter a number. Inside the program I will check if the number is a multiple of any numbers from 1 to 10. If a multiple is found, I will print "multiple is found" along with the number. If no multiple is found, then I will merely print "no multiples found."

```
mynum = int(input('Enter a number '))
for i in range(1,11):
   if (i%mynum) == 0:
      print ('Multiple found for ', mynum)
      break
else:
   print ('No multiple found for ', mynum)
```

When I execute the code, I get this:

```
C:\>python myfirst.py
```

```
Enter a number 3
multiple found for  3
```

Entering 3 satisfies the `if (i%mynum) == 0` line in the program. There is a `break` statement; so the loop is broken and no further matching is made. If I pass a number that will not be matched, for example, 13, this happens:

```
C:\>python myfirst.py
Enter a number 13
No multiple found for 13
```

The loop executes completely without matching and then it comes to the `else` block and prints the statement there. Remember, this `else` block statements are executed only if the `for` loop completes without being broken. If the `for` loop is broken, the `else` block is not executed.

You might be tempted to argue back with a similar construct in PL/SQL, as follows:

```
declare
   mynum number := 13;
begin
   for i in 1..10 loop
       if mod (i,mynum) = 0 then
          dbms_output.put_line('multiple found for '||i);
           exit;
       end if;
   end loop;
   dbms_output.put_line ('No multiple found');
end;
```

When you execute it, you will see:

```
No multiple found
```

This is as expected. So, why are we saying there is no PL/SQL equivalent of the Python program? Let's see by changing the input value of `mynum` from 13 to 3 as it as in the Python program. Here is the output:

```
multiple found for 3
No multiple found
```

Well, we got a mixed output. The first line in the output is actually correct; but the second one is not. The problem is PL/SQL executes the line `dbms_output.put_line ('No multiple found')` anyway, after the loop either completes or is broken. We don't want that. Rather, we want to execute this line only when the loop ends without a match.

There is a solution, however, through this PL/SQL code:

```
declare
   mynum number := 3;
   found boolean := false;
begin
   for i in 1..10 loop
      if mod (i,mynum) = 0 then
         dbms_output.put_line('multiple found for '||i);
         found := true;
         exit;
      end if;
   end loop;
   if (not found) then
      dbms_output.put_line ('No multiple found');
   end if;
end;
```

This is functionally equivalent to the Python program, but with extra lines of code. Keep in mind the behavior of the Python program when `break` and `else` are used. In summary, the `else` block is executed only when the `break` is not executed.

The `ELSE` block in a loop is not just limited to `FOR` loops; you can use it in `WHILE` loops as well. Here is an example Python program where `else` is used in a `while` loop:

```
mynum = int(input('Enter a number '))
i = 0;
while i<11:
    i = i+1
    if (i%mynum) == 0:
        print ('multiple found for ', mynum)
        break
else:
    print ('No multiple found')
```

In summary, the `else` in a loop works as described in the following pseudo code:

- Go through the loop.
- If the loop completes and exits normally, execute the `else` block.
- If the loop breaks, do not execute the `else` block.

Let's Continue

The `continue` statement is used inside a loop to instruct the program to jump to the end of the loop and continue with the rest of the loop iterations as usual. The syntax is the same in Python.

PL/SQL

```
declare
   mynum number := 3;
begin
   for i in 1..10 loop
         if mod (i,mynum) = 0 then
               dbms_output.put_line('multiple found as '||i);
               continue;
               dbms_output.put_line('we are continuing');
         end if;
         dbms_output.put_line ('No multiple found as '||i);
  end loop;
end;
```

The output:

```
No multiple found as 1
No multiple found as 2
multiple found as 3
No multiple found as 4
No multiple found as 5
multiple found as 6
No multiple found as 7
No multiple found as 8
multiple found as 9
No multiple found as 10
```

As you can see, the program prints when a multiple is found; but that does not stop the loop because a loop breaker is not provided. However, the following line is never executed because it comes after the `CONTINUE` statement:

```
dbms_output.put_line('we are continuing');
```

Python

```
mynum = int(input('Enter a number '))
i = 0;
while i<10:
    i = i+1
    if (i%mynum) == 0:
        print ('multiple found as ',i)
        continue
        print ('we are continuing')
    print ('No multiple found as ',i)
```

```
C:\>python myfirst.py
Enter a number 3
No multiple found for  1
No multiple found for  2
multiple found for  3
No multiple found for  4
No multiple found for  5
multiple found for  6
No multiple found for  7
No multiple found for  8
multiple found for  9
No multiple found for  10
```

Note how the following line was not executed.

```
print ('we are continuing')
```

That is because all statements after the `continue` statement were skipped to the end of the loop. Just for the sake of completeness, let's see how indentation plays a vital role in Python. Suppose we make a mistake and do not place the indentation before the above line. Here is the full program:

```
mynum = input('Enter a number ')
i = 0;
while i<10:
    i = i+1
    if (i%mynum) == 0:
        print ('multiple found for ',i)
        continue
    print ('we are continuing')
    print ('No multiple found for ',i)
```

Here is the result:

```
C:\>python myfirst.py
Enter a number 3
we are continuing
No multiple found for  1
we are continuing
No multiple found for  2
multiple found for  3
we are continuing
No multiple found for  4
we are continuing
No multiple found for  5
multiple found for  6
we are continuing
No multiple found for  7
we are continuing
No multiple found for  8
multiple found for  9
we are continuing
No multiple found for  10
```

Note the dramatic change? What a difference the indentation makes! Pay particular attention to indentations in Python. The spacing of lines, not syntactical elements such as `begin` or `end`, define the scope of execution.

Let's Make a Null Pass

When you have to put in a line but nothing needs to be done, you usually use a `NULL` statement in PL/SQL.

PL/SQL

```
declare
   mynum number := 3;
begin
   for i in 1..10 loop
      if mod (i,mynum) = 0 then
         -- dbms_output.put_line('multiple found for '||i);
         null;
      else
         dbms_output.put_line ('No multiple found for '||i);
      end if;
   end loop;
end;
```

Note line number 7. The `null` statement doesn't do anything. But the `IF` statement needs something to be placed there and `NULL` fits the bill.

```
No multiple found for 1
No multiple found for 2
No multiple found for 4
No multiple found for 5
No multiple found for 7
No multiple found for 8
No multiple found for 10
```

Note how the line where the number 3 is encountered is not printed.

Python

In Python, the equivalent statement is called `pass`.

```
mynum = input('Enter a number ')
i = 0;
while i<10:
    i = i+1
    if (i%mynum) == 0:
        # print ('multiple found for ',i)
        pass
    else:
        print ('No multiple found for ',i)
```

Executing it produces this:

```
C:\>python myfirst.py
Enter a number 3
No multiple found for  1
No multiple found for  2
No multiple found for  4
No multiple found for  5
No multiple found for  7
No multiple found for  8
No multiple found for  10
```

Note how the line where the number 3 is encountered is not printed. It's because of the `pass` statement.

Quick Summary

In this article, you learned Python equivalents of some key PL/SQL elements for decision-making and looping.

Functionality	PL/SQL	Python
Basic	IF (condition) then List of statements to be executed; -- list of statements outside of IF block.	`if` (*condition*)`:` *List of statements to be executed* *List of statements outside of if block* Note the dissimilarities: 1. There is no `end if`. 2. The ":" after

the condition.

3. The indentation before the statements in the `if` block. There is no `begin` or `end` statements to mark the beginning and the end of the blocks under the "if" condition. The indentation determines the block.

```
ifSE
(condition):
thenF


List
ofst
sfatements
toatements
be
executed
executed
else:
else
```

List of thet statements inatements else end *block* IF;

list There *of* is *statements* no *outside* end *of* if *IF* and *block*. like `IF`, the blocks of code are determined by indentation.

`elifF`

`for range(`*Start*`,`*End*`):` `in` *Start*`...`*End*

list`op` *of statements*

statement not insidenents the `end` *for* `loop;` *loop*

Note the indentation for the statements inside the `for` loop.

Skipping counters in loop, for example, skipping one number, that is, 1, 3, 5, and so on	for i in Start..End loop if mod(i,2) = 0 then Do something end if; end loop;	`for i in range(Start,End,2):` *Do something*
Counting backwards, for example, 3, 2, 1 and so on	for i in reverse Start..End loop list of statements end loop;	`for i in range(Start, End, -1):` *list of statements*
	`while (condition) loop`	*list of statements*

statement
nottements;
inside
end
the
loop;
while
loop

Note the indentation for the statements inside the `while` loop.

Exiting a loop

`EXIT WHEN` (*Some condition*);

loop

`if` (*Some condition*):

`if` (*Some condition*) `THEN`

`break`

`EXIT;`

Note the indentation.

`while` (*Some condition*):

No PL/SQL equivalent.

`if` (*Some other condition*):

`break`

`else:`

`statements`

`executed if the loop exits or completes without finding a match`

`case` statement when Some condition then List of statements; when Some other condition then List of statements; end case;

There is no `case`; but you can create similar functionality using `if ...` `elif ...`

`continue` in a loop

`pass;` nothing where

a
legal,
syntactically
correct
statement
is
needed

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.