Learning Python for PL/SQL Developers: Part 1

by Arup Nanda

Part 1 of a five-part series that presents an easier way to learn Python by comparing and contrasting it to PL/SQL.

Learning a new language can be challenging as is. Working a full-time job and using precious downtime to learn makes this only tougher. But you have to learn to keep pace with technological advances. The Python language is worth learning due to its use in growing areas such as machine learning and data analytics. If you are a PL/SQL developer, you might be very interested in learning it but probably struggle with carving out time to actually do that. What is the best way to jumpstart learning?

Most books on Python start teaching the language the way they would teach English to a newborn. While that is effective, it takes too much time and it is difficult to jump right into using the language. While teaching myself this language, I found it boring and difficult to commit to learning. But then I had an idea. Years ago I picked up rudimentary Spanish very quickly by learning the Spanish equivalent of English words and constructs, rather than learning Spanish the "right" way, that is, by learning grammar. I was already well versed in English and its grammar. So instead of relearning the grammar of another language, simply substituting English constructs with Spanish ones made the initiation process dramatically faster.

In this series of articles, I follow the same approach. If you are a PL/SQL developer, you already know that language very well and you know how to write a program; You just can't write it in Python, yet. Instead of starting out from the beginning, as with a grammar book, my objective is to teach you Python by showing how it is same as or different from PL/SQL. That guarantees a faster adoption. Once the roots are established, you are free to move onto more intricate details of the language. Happy learning.

How This Series Is Organized

This series has four articles. Each has a corresponding video to show you the concepts presented in the article. I encourage you to check out the videos in addition to the articles. But sometimes finding time can be a challenge, even for the most motivated. To address their needs, each article also has a corresponding podcast that you can listen on the go--while driving, working out, gardening, or whatever. It is a challenge to explain elements of coding in an audio media, and as far I know it has not been attempted. But I believe it can be done, at least to a reasonable extent.

Each article has a summary of important points and finally short quiz to test your understanding. The quizzes are also available to download as PDF documents that you can take with you.

Here are the items you will get for each installment:

- The article
- The video
- The quick summary
- The quiz

In this tutorial, all user inputs will be shown in bold and all the outputs will be in normal code font. Take, for instance, the following code:

```
$ which python
/usr/bin/python
```

Here:

- `$` is the prompt on the computer; so it's in normal weight.
- **`which python`** is in bold because you enter it at the prompt.
- `/usr/bin/python` is also in normal weight because it's displayed as output by the system.

We will begin Part 1 of the introduction here. A video of this presentation is available from the OTN DB Developer Channel on YouTube here: https://www.youtube.com/watch?v=WoAVY7LQbt4

Preparation

First, you have to download the Python environment onto your laptop. That's really simple. Download the software for your specific OS from python.org. Here is the URL at the time of this writing: https://www.python.org/downloads/release/python-2711/. But the URL could change. Visiting python.org is a better choice. When you install the tool, it installs the Python interpreter into a directory chosen by you. I chose C:\Python34. Linux systems might already have Python pre-installed. Here is output from my Linux system:

```
$ which python
/usr/bin/python
```

Getting Started with Python

The easiest approach is to bring up the Python command-line utility--`python.exe` in Windows or `python` in UNIX/Linux-based systems. Make sure the directory is in the `PATH` variable; or just call the executable by its fully qualified name, as shown below.

```
C:\>C:\Python34\python
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC
v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` is the Python prompt, similar to `SQL>` in SQL*Plus. You might want to just type `help()` to see what's available there.

```
>>> help()
Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.4/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

Now you are at the help prompt. Here you can check out the Python help commands. Let's start with a command you will use first: print().

```
help> print
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Quit the help> prompt by typing quit and you will get back to the now-familiar Python prompt (>>>).

```
help> quit


You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>
```

With this out of the way, let's start with our very first Python program, the classic of all beginners, printing "Hello World." Remember, throughout the series, the convention is simple: bold means something you enter; otherwise, it's the computer's output.

```
>>> "Hello World"
'Hello World'
```

That's it. You just printed the words without an explicit `print` command. Python returns the value when you enter it at the command line. Here, enclosing it within double quotes makes it a character string. You could have also enclosed it within single quotes, as shown below:

```
>>> 'Hello World'
```

Here Python merely echoes whatever input was given to it; it's not really performing the exact activity you wanted it to do, that is, printing. In echoing, it preserves the type of the data, that is, string and, therefore, the single quotes in the output are needed. But you don't like them. You want to display the words only; not the quotes around them. To accomplish it, you need to call the `print()` function.

```
>>> print ("Hello World")
Hello World
```

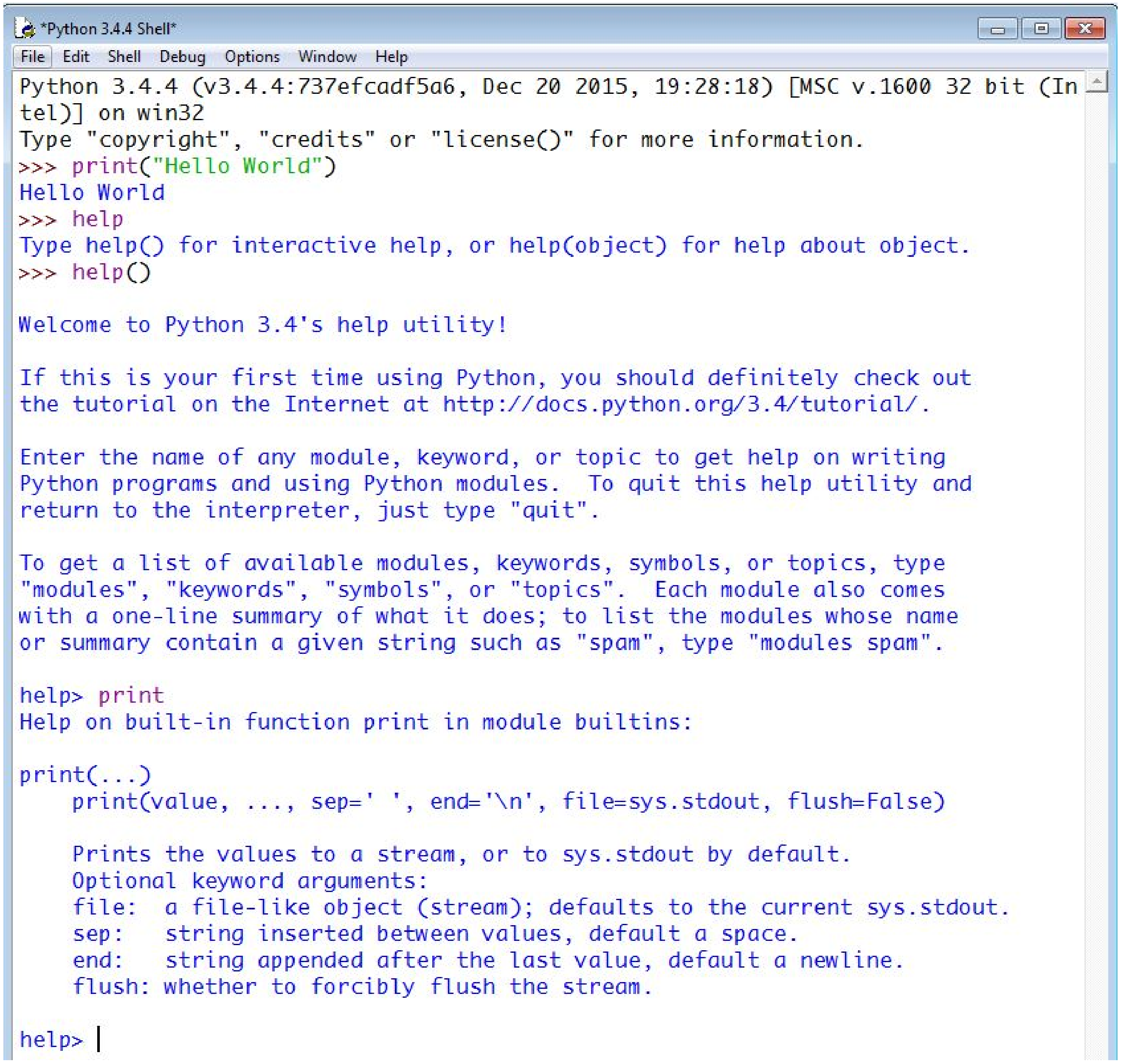
There you go; it's displayed without the quotes. We will learn more about `print` later. We just saw the interactive nature of Python commands, similar to entering commands at the SQL*Plus prompt or the SQL Developer window. We can also create Python programs. Typically Python programs have an extension of `.py`, similar to `.sql` for SQL and PL/SQL scripts. Also, similar to SQL scripts, the extensions are merely a convenience. You don't have to use that; you can use `.anything` or even no extension at all. For the sake of convention, let's use `.py` as an extension. Open up a text editor and create a file with this line:

```
print("hello world")
```

Save the file as `myfirst.py`. Now you can execute this Python file from the Python command line by using the filename as the argument to the Python application.

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

Let's pause for a little and explore one more environment to interact with Python. It comes with its own Interactive Development Environment (IDE) called **IDLE**, which gets installed as well. It's not necessary to use this GUI tool; you can use Python from the command line, very similar to the SQL*Plus command line. You can write Python program code in any text editor and execute it at the Python command line, as you can see later. However, the IDLE environment offers some interesting helper functionality that enables you to code faster and debug easier. Think of it as using SQL Developer. You don't need SQL Developer to write PL/SQL code; any text editor will work just fine. Having SQL Developer just makes it easier by highlighting keywords, allowing debugging stops, automatically formatting the code, and so on. IDLE allows you similar functionality for Python development. We will explore IDLE in detail in a later article. For now, let's just see what the IDLE environment looks like.



```
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.
>>> print("Hello World")
Hello World
>>> help
Type help() for interactive help, or help(object) for help about object.
>>> help()

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.4/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> print
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

help> |
```

Figure 1

IDLE displays the keys words in a color-coded manner. Note how the `print()` function is in red--indicating it as a keyword--and how the words `"Hello World"` are in green, which indicates a literal value. Color coding could be useful in some cases, especially for a beginner. Again, IDLE is not mandatory; you can use the Python command-line tool as well. For this article series, we will call the Python command line tool from the Windows (or other OS) prompt when we execute a Python code file. You can use IDLE to create the file for color coding, and so on, or just use the plain notepad text editor. You might prefer to use vi, if it is available on you platform, for its familiar interface and powerful editing facilities. For interactive Python commands, you are free to use either option.

Python's echoing does not stop at just inputs. You can also use arithmetic operations directly at the command prompt; you will get the result in the output, like a calculator. Here is an example:

```
>>> 1+2
3
>>>
```

In PL/SQL, you would have to do this:

```
begin
    dbms_output.put_line(1+2);
end;
```

If you pass a non-numeric character without quotes, it will not produce the same result. Let's see that by using a value `x` at the Python prompt:

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

`x` will be interpreted as a variable and because you haven't defined it, it will result in a syntax error. That brings us to our first lesson, *variables*.

Variables

Variables are fundamental building blocks of any program. Let's see how you use variables in PL/SQL and Python:

PL/SQL

You define variables in the DECLARE section.

```
DECLARE
   numvar1    number;
```

Python

There is no need to define variables beforehand. Variables are defined when you assign values to them. Variables can be integer, floating point number, long integer, string, boolean, and more.

Variable Assignments

PL/SQL

```
BEGIN
   numvar1 := 1;
   charvar1 := 'a char var';
```

Python

```
numvar1 = 1
charvar1 = 'a char var'
```

Note two important changes in Python compared to PL/SQL:

- Assignments are `=`, not `:=`.
- There is no terminator character like `;`.

There are many types of datatypes in Python: integer, floating point, string, boolean, and so on. For floating point values, use a decimal point:

```
floatvar2 = 1.0
```

For character variables, just assign a character value:

```
charvar1 = 'a char var'
```

OR

```
charvar1 = "a char var"
```

Remember, PL/SQL expects literals to be enclosed in single quotes only. Literals enclosed in double quotes are interpreted as variables. Here is an example in PL/SQL that produces an error.

```
SQL> declare
  2    charvar1 varchar2(4000);
  3 begin
  4    charvar1 := "a char var";
  5 end;
  6 /
   charvar1 := "a char var";

ERROR at line 4:
ORA-06550: line 4, column 17:
PLS-00201: identifier 'a char var' must be declared
ORA-06550: line 4, column 4:
PL/SQL: Statement ignored
```

Dissimilarities

Unlike PL/SQL, Python variables can also be assigned in a chain, like the following:

```
var1 = var2 = var3 = 10
```

In this case, `var3` is assigned the value 10, which is then assigned to `var2`, which in turn is assigned to `var1`.

One of the most interesting variable assignments in Python can be seen in the positional value assignment, shown below.

```
var1, var2, var3 = 1, 2, 3
```

This assigns values 1, 2 and 3 to variables `var1`, `var2`, and `var3`, respectively.

But the powerful functionality doesn't stop there. You can assign any datatype in this string of values. Here we assign a number, a boolean type, and a character to the variables in the same string of assignments.

```
var1, var2, var3 = 1, True, "Three"
```

We will cover more on other types of variables such as arrays and collections later in this article and in much more detail later in the series.

Since Python does not need you to define the variables beforehand and the datatype of the variable is determined by the assigned value, you might lose track of what datatype a specific variable is. To identify the datatype of the variable, use the `type()` function:

```
>>> n = 1
>>> type n
<class 'int'>

>>> nf = 1.0
>>> type(nf)
<class 'float'>
```

Printing

Let's see how to display something--a variable or just a literal value.

PL/SQL

```
dbms_output.put_line('here is a text');
```

OR, to display a number, you may omit the single quotes.

```
dbms_output.put_line(1);
```

Python

At the Python command line, as you saw earlier, there is no need to give an explicit command for display. Whatever you put in the command line is returned back to you in output. Here is an example where we first pass a string and then an integer:

```
>>> 'here is a text'
'here is a text'
>>> 1
1
```

But notice how the displayed value is inside single quotes for the string input. Also this is available only for interactive commands. For Python scripts, you need to use the specific `print()` function to display the value. Let's create the script called `myfirst.py`, as shown below. Note that we used the same filename earlier. We will continue using the same filename for all the tutorials, unless there is a need to preserve the filename. We will just overwrite the file each time.

```
numvar1 = 1
floatvar1 = 1.000000
charvar1 = 'a char var'
print(numvar1)
print(floatvar1)
print(charvar1)
```

You can execute it:

```
C:\>python myfirst.py
1
1.0
a char var
```

Like PL/SQL, Python supports boolean variables as well, which are assigned either "True" and "False."

```
booVar1 = True
print(booVar1)
```

Note two important differences with PL/SQL:

- The value "True" is case sensitive. It's not "true." Same goes for "False."
- You can print the boolean variable, which is not possible in PL/SQL.

There is another important difference with PL/SQL. The boolean values are not just True and False. Later in this series, in the installment on conditions, you will learn that Python considers any numerical result of zero as false and any non-zero value as true.

To print multiple values in a single print function, in PL/SQL you concatenate the values as shown below:

```
SQL> declare
  2    x       number;
  3    y       number;
4  begin
  5     x := 10;
  6     y := 9;
  7     dbms_output.put_line('x='||x||' y='||y);
  8  end;
  9  /
```

```
x=10 y=9

PL/SQL procedure successfully completed.
```

In Python you simply pass multiple values separated by comma. The following are the contents of a Python file. For all the Python code from this point onwards, we will use a Python file and execute it, unless we explicitly specify as an interactive command.

```
x = 10
y = 9
print('x=',x,'y=',y)
```

Here is the output:

```
C:\>python myfirst.py
x= 10 y= 9
```

It's not quite as expected. There is a gap before the values right after the equality sign. To remove that gap, we have to pass another parameter to the `print()` function: `sep=''`, as shown below:

```
x = 10
y = 9
print('x=',x,' y=',y, sep='')
```

Here is the output:

```
C:\>python myfirst.py
x=10 y=9
```

In fact, the `sep` parameter can take any value. Here is another example where we assign `...` as the separator:

```
x = 10
y = 9
print('x=',x,' y=',y, sep='....')
```

Here is the output:

```
C:\>python myfirst.py
x=....10.... y=....9
```

Like the PL/SQL `||` concatenation operator, Python has the explicit `+` operator. Here is an example on the interactive command prompt, which also applies to files.

```
>>> print ("This is " + "a string")
This is a string
```

Just as the PL/SQL operator `||` can be used concatenate multiple strings to a single one, `+` in Python can be used to string together many characters to a single string.

```
stmt = '   Welcome to learning Python'
stmt = stmt + '\n     for PL/SQL Developers!'
print(stmt)
```

But the power of Python doesn't just stop there. You can achieve the above effect by writing the code slightly differently, as shown in the following

```
stmt = (
'   Welcome to learning Python'
'\n    for PL/SQL Developers!'
)
print(stmt)
```

Note how we split up the `stmt` variable into multiple lines for easier readability. There was no need to concatenate. Putting all the fragments in parenthesis was the key.

The multiplication operator (`*`) works in the case of strings in Python. This repeats the strings that many times.

```
>>> print (2 * ("This is " + "a string "))
This is a string This is a string
```

There is no equivalent in PL/SQL. You can loosely refer to this functionality as overloading of the `*` operator.

To print multiple lines, use the following structure. Note the `\` at the end of the first line only. The set of two triple-quotes (single or double) mark multi-line displays.

```
print("""\
   Welcome to learning Python
       for PL/SQL Developers!
""")
```

Conversions

PL/SQL

To convert a number to the character format, you use the `to_char()` function.

```
v1 := to_char(n1);
```

Or, to convert a character to number:

```
n1 := to_number(v1);
```

Python

The corresponding function is `str()` in Python. It returns the internal printable representation of the number, which is practically the same as the `to_char()` functionality. Here is an example where you print the value of a floating point variable:

```
n1 = 1.123456789
print ('The value of n1 = ' + str(n1))
```

Similarly, to convert from character to a number, use the `float()` or `int()` functions:

```
v1 = "1.123456789"
n1 = float(v1)
print (n1)
```

Comments

PL/SQL

Comments are marked in two different manners:

- With `--` marks at the beginning of each line
- With anything that is between `/*` and `*/` marks

Python

Comment lines starts with `#`, just like UNIX shell scripts, as shown below.

```
# print('this line is a comment')
print('this line is not a comment')
```

When you execute this file, you get this output:

```
C:\>python myfirst.py
this line is not a comment
```

The comment line is not printed, as expected.

Multi-line comments are enclosed in `"""` marks, similar to `/* */` in PL/SQL.

```
""" this is comment
    comment still going
    and going ....
"""
```

The text between the `"""` marks is all comment.

User Input for Interactive Programs

PL/SQL

There is no user input in PL/SQL, since it's just a database-resident tool. You can use the SQL*Plus command ACCEPT. Here is an example script--`myplsql.sql`.

```
accept x number prompt 'Enter the value of x: '
begin
    dbms_output.put_line('x='||&x);
end;
```

Executing it:

```
SQL> @myplsql
```

```
Enter the value of x: 1
old  2:       dbms_output.put_line('x='||&x);
new  2:       dbms_output.put_line('x='||        1);
x=1

PL/SQL procedure successfully completed.
```

Python

To accept the user input, you use the `input` function. It accepts one parameter, the prompt shown to the user.

```
x = input('Enter the value of x: ')
print ('x=', x, sep='')
```

Executing it:

```
C:\>python myfirst.py
Enter the value of x: 1
x=1
```

You can pass any type of data, for example, a string, as shown below:

```
C:\>python myfirst.py
Enter the value of x: 'aaa'
x=aaa
```

Operators

Many operators in Python are the same as in PL/SQL. Here is a list of the operators that have the same meaning in PL/SQL.

```
+ - * /  < > <= >= != <>
```

Here is a simple Python program to demonstrate the various operators. Remember, in Python you can print the outcome of a boolean value directly.

```
x = input('Enter the value of x: ')
y = input('Enter the value of y: ')
print ('x=', x, ' y=', y, sep='')
print (x>y)
```

Executing this program:

```
C:\>python myfirst.py
Enter the value of x: 10
Enter the value of y: 3
x=10 y=3
True
```

If you execute it with different values:

```
C:\>python myfirst.py
Enter the value of x: 10
Enter the value of y: 11
x=10 y=11
False
```

But the other operators are different. The most notable is the equality matching operator, `==`, which is similar in functionality to `=` in PL/SQL. Here is an example:

PL/SQL

```
declare
  v1      number := 1;
  v2      number := 2;
begin
  if (v1=v2) then
        dbms_output.put_line('Same');
  else
        dbms_output.put_line ('Different');
  end if;
end;
```

Python

You can use merely the following line in the Python command-line interface. The `print()` function can print booleans.

```
>>> print (1==2)
```

```
False
```

There are many operators available in Python which are either not available in PL/SQL or available as functions only. Take, for instance, the POWER function in PL/SQL. To get the 5 powered 2, you use:

```
declare
   n1      number;
begin
   n1 := power(5,2);
   dbms_output.put_line(n1);
end;

25

PL/SQL procedure successfully completed.
```

In Python, it's an operator (**).

```
>>> 5**2
25
```

Note that there is a `pow()` function in Python as well:

```
>>> pow(2,4)
16
```

Another example is the function which returns the remainder.

```
SQL> declare
  2    n1      number;
  3  begin
  4     n1 := remainder(5,2);
  5    dbms_output.put_line(n1);
  6  end;
  7  /

1

PL/SQL procedure successfully completed.
```

In Python, you use the `%` operator.

```
>>> 5%2
```

```
1
```

A huge caveat here might be in order. If you pass a negative number, the PL/SQL `remainder()` function returns negative but Python's `%` operator will not. It will return the classic remainder operation.

Unlike PL/SQL, Python follows the C-style combination assignment-operators. For instance, `+=` adds the value to the current value of the variable. Here is an example:

```
>>> v1 = 3
>>> v1 += 3
>>> v1
6
```

The value of `v1` is 6 now. So:

```
v1 += 3
```

is the same as:

```
v1 = v1 + 3
```

Other operators in the same category are `-=`, `*=`, `/=`, `%=`, `**=`, and `//=`.

Some of the operators in Python are not even available in PL/SQL. Here are some examples:

The floor division operator (`//`) returns the floor of the value after the division. Here is an example. Let's see a simple division:

```
>>> 15.0/2.0
7.5
```

Now let's use the same input values but with floor division:

```
>>> 15.0//2.0
7.0
```

We will cover arithmetic operations in detail later.

String Operations

Let's explore some common string operations such as converting to upper case, locating a pattern, and so on. Rather than going through each operation in detail, it's probably easier to see the PL/SQL analogous commands in Python. Suppose I have a variable called `s1` with a value `'This is a String'`.

```
>>> s1 = 'This is a String'
```

Operation	**PL/SQL**	**Python**
Converting to uppercase	upper(s1)	s1.upper()
Converting to lowercase	lower(s1)	s1.lower()
Truncating, that is, removing characters (`'String'`) at the end of the string	rtrim(s1,'String') Omitting the second parameter means removing white spaces	s1.strip('String') Omitting the second parameter means removing white spaces
Finding the pattern `'is'` in the string	`loc := instr(s1,'is')`	`loc = s1.find('is')`

Replacing a pattern in the string	replace(s1,'String','Character String')
	
	Example:
	declare
	  s1 varchar2(40);
	  s2 varchar2(40);
	begin
	  s1 := 'This is a String';
	  s2 := replace (s1,'String','Character String');
	  dbms_output.put_line(s2);
	end;
		s1.replace('String','Character String')
		
		Example:
		s1 = 'This is a String'
		s2 = s1.replace('String','Character String')
		print(s2)

There are more string manipulation operations as well. In particular, you might be wondering about the SUBSTR() function in SQL, which allows you to choose a specific set of characters from a string. You will learn about this operation in the next section, "Arrays on page 21." But right now, let's conclude the string operations with one specific operation that is valuable and unique to Python. There is no direct operational analog in PL/SQL. It's `split()`. You can split a string into multiple parts on a delimiter. Take for instance a case where you want to break up a sentence into multiple words. You would do this:

```
>>> s1 = 'This is a String'
>>> s2 = s1.split()
>>> s2
['This', 'is', 'a', 'String']
```

Now `s2` contains these words. In this case, `split()` splits the string on whitespaces, which is the default. You can put anything else as a parameter, for example, to split on the "-" character, you can use the following.

```
>>> s1.split('-')
['This is', 'just', 'a string']
```

The opposite of `split()` is `join()`, which joins the words.

```
>>> s3 = ' '.join(s2)
>>> s3
'This is a String'
```

Note how we used `' '` as the delimiter, which was used in the beginning. If you want, you can put any other delimiter inside these single quotes.

Arrays

So far we covered only unary datatypes. In a typical application, especially in data-intensive ones, you often need to pull data into arrays to manipulate it easily. Let's see some rudimentary arrays, called *sequences* in Python. We will cover sequences in detail in a separate article later in the series.

PL/SQL

There are different types of collections in PL/SQL. Here is a simple type, called PL/SQL Tables (also called Index-by Tables or Associative Arrays) where you can reference the data by its position in the array.

```
declare
   type ty_tabtype is table of varchar2(3)
          index by binary_integer;
   l_weekdays     ty_tabtype;
begin
   l_weekdays (1) := 'Sun';
   l_weekdays (2) := 'Mon';
   l_weekdays (3) := 'Tue';
   l_weekdays (4) := 'Wed';
```

```
  l_weekdays (5) := 'Thu';
  l_weekdays (6) := 'Fri';
  l_weekdays (7) := 'Sat';
  for i in l_weekdays.FIRST .. l_weekdays.LAST loop
        dbms_output.put_line('Weekday '||i||' = '||
                l_weekdays (i));
  end loop;
end;
/
```

There are other types of collections as well. We will explore these more in detail later in the series. Here is another, called VARRAY:

```
declare
  type ty_working_days_of_week_arr is varray(7) of varchar2(3);
  l_working_days_of_week_arr ty_working_days_of_week_arr;
begin
  l_working_days_of_week_arr := ty_working_days_of_week_arr('Mon','Tue','Wed','Thu','Fri');
  dbms_output.put_line('Day 1 = '||l_working_days_of_week_arr (1));
end;
/
```

When executed, it will store the working days of the week in an array and produce the first element of the array as the output.

```
Day 1 = Mon

PL/SQL procedure successfully completed.
```

We will explore collections in detail in a separate exclusive installment. Here are some important properties of the PL/SQL collections:

- All the elements are of the same datatype, that is, you can't have one as a number and the other character.
- The array index starts with 1.

Python

Likewise Python has three different types of arrays, which are called sequences. The most common is *list*. It's a set of values separated by commas enclosed inside square brackets. You can print it on the screen without iterating it one by one. Here is the `myfirst.py` file:

```
listVar1 = [1,2,3,4]
print(listVar1)
```

```
C:\>python myfirst.py
[1, 2, 3, 4]
```

But unlike PL/SQL, all the elements in the Python array don't have to be of the same datatype. The elements of the list could be of any datatype.

```
listVar2 = [1,"Two",3.0, 4, True]
print (listVar2)
```

```
C:\>python myfirst.py
[1, 'Two', 3.0, 4, True]
```

The second type of array is called a *tuple*. A tuple is the same as a list; but is *immutable*, that is, it isn't allowed to change once defined. Tuples are written by enclosing the similar values, as in the case of a list with regular parentheses.

```
>>> tVar1 = (1,2,3)
>>> print (tVar1)
(1, 2, 3)
>>> type(tVar1)
<class 'tuple'>
```

A classic example of tuple is days of the week.

```
>>> daysOfWeek = ("Sun","Mon","Tue","Wed","Thu","Fri","Sat")
>>> print(daysOfWeek)
('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat')
```

A *set* is the same as a list with two huge exceptions:

- The items inside the list are unique, that is, no repetitions are allowed.
- It is just one collection of data elements. The exact position of the elements is immaterial. That's why it's also called an unordered list.

Sets are represented as lists of values separated by commas but enclosed within curly braces. Here is an example:

```
>>> sVar1 = {1,2,3,3}
>>> sVar1
```

```
set([1, 2, 3])
```

But note how the variable is represented. It's a string of values, as indicated by the values enclosed within (`[` and `]`). You can convert a list to a set, too, using the `set()` function. Note how the extra elements are removed after the conversion.

```
>>> Var1 = (1,2,3,3,4,4,4)
>>> sVar1 = set(Var1)
>>> sVar1
set([1, 2, 3, 4])
```

You can reference individual elements of the array using a subscript notation, just as in PL/SQL. However unlike PL/SQL, Python subscripts start with 0; not 1. Let's see an example:

```
>>> print(Var1)
(1, 2, 3, 3, 4, 4, 4)
>>> # now choose the first element
>>> print(Var1[0])
1
```

You can reference parts of the array as well; not just one element. Here is an example of selecting from a five-element array:

```
>>> Var1 = [1,2,3,4,5]
>>> Var2 = Var1[1:3]
>>> Var2
[2, 3]
```

Remember, an array index starts at 0, not 1; so `Var1[0]` is 1, `Var1[1]` is 2, `Var1[2]` is 3, `Var1[3]` is 4, and so on. Therefore, the elements 1 through 3 should have been 2, 3 and 4, right? How come we got 2 and 3 only? 4 was omitted. Why?

This is yet another of the ways Python's array indexing differs from PL/SQL's. In the Python expression `array[i:j]`, `j` does *not* mean the j-th element; it means the element up to but *not including it*. So the range in `Var1[1:3]` means the selection is up to--but *not including*--the third element; hence, the value 4 (the third element) was not selected. Not understanding this could lead to bugs; so please pay attention to it.

The numbers on either side of the `:` mark in the array indexing are optional. If you omit them, they default to the fullest range on that side. For instance, `[:3]` means all elements starting from the beginning up to, but not including, the third element. It's the same as `[0:3]`.

```
>>> Var1[:3]
[1, 2, 3]
```

Similarly, omitting the second parameter means the going to the very end, starting from the position specified. For instance, `[3:]` means starting from after the third element up to the end.

```
>>> Var2 = Var1[3:]
>>> Var2
[4, 5]
```

If the positions in the indexing seems to defy logic, you are not alone. Refer to the diagram in Figure 2 for understanding the starting and ending positions of the indexes in an array. When referring to only one element, use the array index, shown on the top of the diagram.

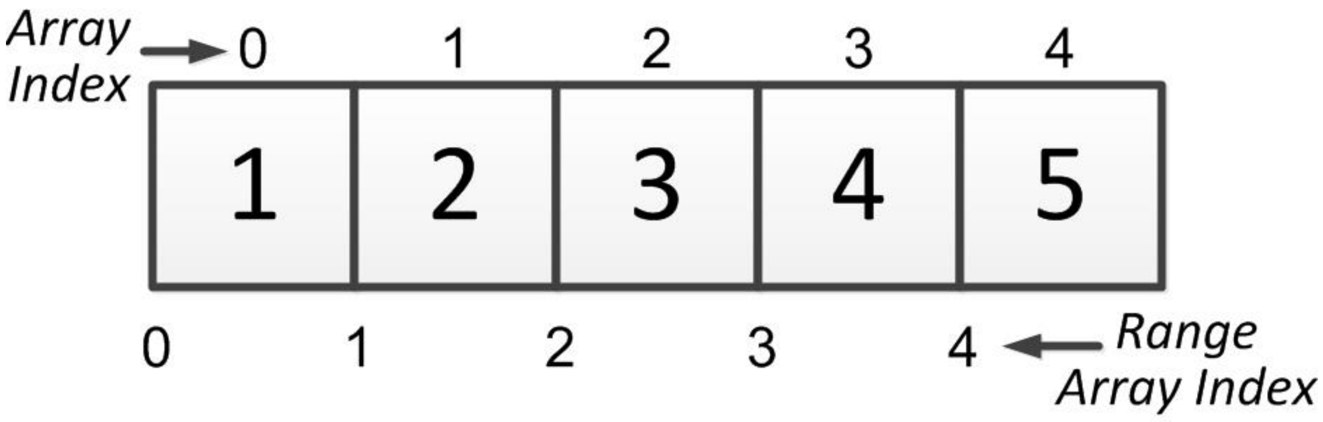


Figure 2

However, when index ranges are referenced, indicating multiple elements, then remember that the *left edge* of the element indicates the position of that element, as shown at the bottom of the Figure 2. So when you write `Var1[0:3]`, it means starting at the left edge of the leftmost element (0-th), with a value of 1, to the left edge that is marked "3." These elements are shaded in Figure 3 below:

Figure 3

Similarly when you write `Var1[2:5]`, the same concept applies. It picks the elements starting at the left edge marked "2" and ending with the left edge "5." But because there is no cell with a left edge of "5," the rightmost cell's right edge is considered to be the left edge marked "5." The chosen elements are shown in Figure 4 shaded.

Figure 4

The indices can be negative as well. Negative indices indicate counting from the end; not the beginning. So `[-2]` means the second element from the end.

```
>>> print Var1[-2]
4
```

Index ranges can also be negative. This could be a bit tricky to grasp. Remember, ranges always point to an edge of the element, the left edge to be precise. In the positive index case, the left edge of the leftmost element was marked "0." In case of negative indexes, the reverse is true; the *rightmost* edge of the element is chosen. So, an expression [-3:-1] will return the values shown in Figure 5.

```
>>> print Var1[-3:-1]
[3, 4]
```

Figure 5

If you omit the rightmost number in the range, it will be assumed to be 0. So `[-3:]` is the same as `[-3:0]`

```
>>> print Var1[-3:]
[3, 4, 5]
```

Similarly, omitting the leftmost side number will default to the leftmost element.

```
>>> print Var1[:-3]
[1, 2]
```

This selection works the same way for tuples.

```
>>> tVar1 = (1,2,3,4,5)
>>> tVar1[1]
2
```

However, it does not for sets. That's because sets are unordered and merely a collection of values. We will see how they work later. But for now, let's explore the ways to manipulate the lists. Start with creating a list variable first.

```
>>> workingDays = ["Mon","Tue","Wed","Thu","Fri"]
```

Let's check the value in this variable:

```
>>> workingDays
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
```

The equivalent of PL/SQL `l_working_days_of_week_arr.count` in Python is `len(workingDays)`. It returns the number of elements you have in that array. Here is an example:

```
>>> len(workingDays)
5
```

Now suppose Saturday becomes a working day.

```
>>> workingDays.append ("Sat")
```

Let's make sure it's added properly:

```
>>> workingDays
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
```

It works. `Sat` is now in the list. Suppose we change our mind and we want to remove the day from the list. `remove` does it for us.

```
>>> workingDays.remove ("Sat")
>>> workingDays
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
```

Suppose by a decree of the government, Wednesday becomes a non-working day (Yoo Hoo!) and now you have to remove it from the list.

```
>>> workingDays.remove ("Wed")
>>> workingDays
['Mon', 'Tue', 'Thu', 'Fri']
```

Good; the day is gone. But moments later, you learned that the removal was a mistake. The government made no such decree and Wednesday is still a working day. You have to put it back on the list. If you use `append` as you saw earlier, the value will go at the end. You want it to go at a specific place--between `Tue` and `Thu`. To insert it in that place, you have to use another approach, using `insert`. But that function takes arguments: the position where it is to be inserted and the value. To find the position of `Thu`, you could count, or you could use another function--`index()`--that shows the position of a specific value:

```
>>> workingDays.index("Thu")
2
```

The position is 2; so we will insert the new element there. Here is how we do it:

```
>>> workingDays.insert(2,"Wed")
```

Let's make sure the list is correct.

```
>>> workingDays
```

```
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
```

Since tuple is *immutable*, that is, changes are not allowed, functions such as `append()` and `remove()` will not work. Let's see an example. A good example of a tuple is the days of the week, which will not change. Let's define and assign values to the tuple variable. Note the regular parentheses, which specify a tuple.

```
>>> daysOfWeek = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri','Sat','Sun')
```

Make sure the variable is indeed that of the tuple datatype.

```
>>> type(daysOfWeek)
<type 'tuple'>
```

Now try to remove an element:

```
>>> daysOfWeek.remove('Sun')
```

It will throw the error shown below:

```
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    daysOfWeek.remove('Sun')
AttributeError: 'tuple' object has no attribute 'remove'
```

The error is self-explanatory.

The final type of array is called a *dictionary*. It allows you to create a key-value pair of data elements. Let's see an example where we will build a key-value pair of books and their corresponding authors.

```
>>> books = {'Pride and Prejudice':'Jane Austen',\
     '1984':'George Orwell',\
     'Anna Karenina':'Leo Tolstoy',\
     'Adventures of Tom Sawyer':'Mark Twain'}
```

If you want to check the contents, you can simply type the name of the variable or print the variable.

```
>>> books
{'Anna Karenina': 'Leo Tolstoy', '1984': 'George Orwell', 'Pride and Prejudice': 'Jane Austen', 'Adventures of
Tom Sawyer': 'Mark Twain'}
```

To get the keys of the pairs:

```
>>> books.keys()
```

```
['Anna Karenina', '1984', 'Pride and Prejudice', 'Adventures of Tom Sawyer']
```

And, to get the values of the pairs:

```
>>> books.values()
['Leo Tolstoy', 'George Orwell', 'Jane Austen', 'Mark Twain']
```

Since it's a key value pair, when you want to get the value, simply call the variable with the key as the index in an array.

```
>>> books["Anna Karenina"]
'Leo Tolstoy'
```

We will discuss the collections in detail in later articles.

Summary

Let's see simple summary of all the Python elements we learned here that we already know work in PL/SQL.

Operation	PL/SQL	Python
Declaration of variables	`DECLARE` `numvar1 NUMBER;`	None required.
Assignment of the values	`numvar1 := 1;`	`numvar1 = 1` `n1 = n2 = n3 = 1` `n1, n2, n3 = 1,2,3`
To know the datatype of a variable	Not required; it's declared explicitly	`type(var1)`
Boolean datatype	TRUE and FALSE (case-insensitive)	True, False (case-sensitive) Any value that evaluates to 0 is False Any non-zero value is True
Concatenation	`'Char1'||'Char2'` `CONCAT('Char1','Char2')`	`'Char1','Char2'` `'Char1' + 'Char2'`
Line termination	`;`	None required
Display	`dbms_output.put_line()`	`print()`
Comments	Lines starting with `--` OR between `/*` and `*/`	Lines starting with `#` Lines between `"""` quotes
Equality operator	`=`	`==`

Conversions	`to_char(n1)` `to_number(c1)`	`str(n1)` `float(c1)`
User inputs	`ACCEPT n1 NUMBER PROMPT 'Enter value of n1: '` (SQL*Plus, actually)	`n1 = input('Enter value of n1: ')`
Same operators	`+ - * /  < > <= >= != <>`	`+ - * /  < > <= >= != <>`
Power function	`POWER(5,2)`	`5**2`
Modulus or remainder	`REMAINDER(n1,n2)`	`n1%n2`
Combination operator	Doesn't exist	`n1 += 3` functionally same as `n1 = n1 + 3`
Array	VARRAY PL/SQL tables Nested tables	list `lVar = [1,2,3,3]` tuple `tVar = (1,2,3,3)` set `sVar = {1,2,3} # duplicates not allowed` dictionary `dVar = {"key1":"val1","key2":"val2"}`
Array index	Starts with 1	Starts with 0

Quiz

Now let's have a small quiz to make sure you understood 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.