# Learning Python for PL/SQL Developers: Part 3

*by Arup Nanda*

### Part 3 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 3

Collections, or arrays, are important parts of a program, especially for data analysis. Since our ultimate objective is to do exactly that, understanding arrays is an important part of the learning Python. We saw some examples of collections in part 1 of this series. In this module, we will see more details including advanced use cases.

The overriding objective of the series is to jumpstart learning Python for PL/SQL developers. So we will start with how things are done in PL/SQL and how the same functionality is achieved in Python. Then we will explore further by looking at advanced collection operations that might not be available, at least not directly, in PL/SQL.

In PL/SQL, there are basically three types of collections:

- Associative arrays
- Nested tables
- VARRAYs

Let's see each one in detail and its Python counterpart.

## Associative Arrays

These are also called PL/SQL tables. An associative array is an arbitrary collection of keys and values. The two important properties of associative arrays are

- They are empty (but not null) until you populate them.
- They can hold an any number of elements. You don't need to specify the number while creating the array.

- You can access the elements of the array without knowing their positions.
- You have to reference the elements by an index, not by position. Therefore associative arrays are also called unordered lists.

To demonstrate a use case, let's see the example where we need to hold some book titles and their authors using a key-value pair structure. We build an associative array of `varchar2(30)` values (the "value" part of key-value pair) indexed by another `varchar2` (the "key" part of the key-value pair). The book is the key and the author is the value. We will initially populate this array with four books as shown below:

| **Book** | **Author** |
|---|---|
| Pride and Prejudice | Jane Austen |
| 1984 | George Orwell |
| Anna Karenina | Leo Tolstoy |
| Adventures of Tom Sawyer | Mark Twain |

In the programs below, we will populate the array and then select from the array. This type of datatype is called a *dictionary* object in Python. Remember from the part 1 that dictionary objects are lists of key-value pairs that need to referenced by the index and not by the position, since position can be arbitrary.

# PL/SQL

```
--pl1.sql
declare
  type ty_tabtype is table of varchar2(30)
       index by varchar2(30);
 l_books       ty_tabtype;
 i             varchar2(30);
begin
  l_books ('Pride and Prejudice') := 'Jane Austen';
  l_books ('1984') := 'George Orwell';
  l_books ('Anna Karenina') := 'Leo Tolstoy';
  l_books ('Adventures of Tom Sawyer') := 'Mark Twain';
  --
  -- now let's display the books and their authors
  --
  i := l_books.first;
  while i is not null loop
    dbms_output.put_line('Author of '||i||' = '||
                l_books (i));
```

```
      i := l_books.next(i);
   end loop;
end;
```

Here is the output:

```
Author of 1984 = George Orwell
Author of Adventures of Tom Sawyer = Mark Twain
Author of Anna Karenina = Leo Tolstoy
Author of Pride and Prejudice = Jane Austen
```

# Python

```
#py1.txt
l_books = {'Pride and Prejudice':'Jane Austen',\
    '1984':'George Orwell',\
    'Anna Karenina':'Leo Tolstoy',\
    'Adventures of Tom Sawyer':'Mark Twain'}

for i in l_books.keys():
       print ('Author of '+i+' = '+ l_books[i])
```

Output:

```
C:\>python py1.txt
Author of Anna Karenina = Leo Tolstoy
Author of 1984 = George Orwell
Author of Pride and Prejudice = Jane Austen
Author of Adventures of Tom Sawyer = Mark Twain
```

The biggest difference is perhaps how you can assign values to an associative array. In PL/SQL, you have to assign elements one by one. In Python, you can assign all the values at once. In PL/SQL, you can assign some types of associative arrays (those that are indexed by PLS_INTEGER or a number) by selecting from a table and using the BULK COLLECT INTO clause to store the retrieved column values in the array; but there is no way to assign values directly. Other PL/SQL collections can do that, which we will see later.

But an array is not just for storing and selecting all the elements at once. We have to be able to manipulate the individual elements and records. Suppose we want to modify the name of the author of the book *Adventures of Tom Sawyer* from "Mark Twain" to his real name: Samuel Langhorne Clemens.

# PL/SQL

```
-- pl2.sql
declare
  type ty_tabtype is table of varchar2(30)
        index by varchar2(30);
  l_books     ty_tabtype;
  i              varchar2(30);
begin
  l_books ('Pride and Prejudice') := 'Jane Austen';
  l_books ('1984') := 'George Orwell';
  l_books ('Anna Karenina') := 'Leo Tolstoy';
  l_books ('Adventures of Tom Sawyer') :='Mark Twain';
  --
  -- let's make a change in the author's name
  --
  l_books ('Adventures of Tom Sawyer') :='Samuel Langhorne Clemens';
  --
  -- let's display the values now
  --
  i := l_books.first;
  while i is not null loop
    dbms_output.put_line('Author of '||i||' = '||
                l_books (i));
     i := l_books.next(i);
  end loop;
end;
```

Output:

```
Author of 1984 = George Orwell
Author of Adventures of Tom Sawyer = Samuel Langhorne Clemens
Author of Anna Karenina = Leo Tolstoy
Author of Pride and Prejudice = Jane Austen
```

The name has been successfully changed for the book *Adventures of Tom Sawyer*.

# Python

```
# py2.txt
```

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

# Let's change the element

l_books["Adventures of Tom Sawyer"] = 'Samuel Langhorne Clemens'

#Check the elements

for i in l_books.keys():
      print ('Author of '+i+' = '+ l_books[i])
```

Running it, we get this:

```
Author of Anna Karenina is Leo Tolstoy
Author of 1984 is George Orwell
Author of Pride and Prejudice is Jane Austen
Author of Adventures of Tom Sawyer is Samuel Langhorne Clemens
```

The name has been changed. Just like PL/SQL, the values can be directly assigned by referencing the key. Similarly if you want to add another key-value pair, you can do it by merely adding to the collection.

```
# py3.txt
l_books = {'Pride and Prejudice':'Jane Austen',\
    '1984':'George Orwell',\
    'Anna Karenina':'Leo Tolstoy',\
    'Adventures of Tom Sawyer':'Mark Twain'}

for i in l_books.keys():
      print ('Author of '+i+' is '+ l_books[i])

print ('Let\'s add a new book')
l_books["Adventures of Huckleberry Finn"] = 'Samuel Langhorne Clemens'
for i in l_books.keys():
      print ('Author of '+i+' is '+ l_books[i])
```

Here is the output:

```
Author of Anna Karenina is Leo Tolstoy
Author of 1984 is George Orwell
Author of Pride and Prejudice is Jane Austen
```

```
Author of Adventures of Tom Sawyer is Mark Twain
Let's add a new book
Author of Anna Karenina is Leo Tolstoy
Author of 1984 is George Orwell
Author of Pride and Prejudice is Jane Austen
Author of Adventures of Huckleberry Finn is Samuel Langhorne Clemens
Author of Adventures of Tom Sawyer is Mark Twain
```

The last part of the output shows the new record.

# Nested Tables

Unlike associative arrays, nested tables in PL/SQL are ordered and the elements can be addressed by their position in the array. There is no "index" to reference them by. Therefore, this type of data can't be used for key-value pairs. Let's see an example where we will store the days of the week to a variable.

## PL/SQL

```
-- pl4.sql
declare
  type ty_tabtype is table of varchar2(30);
  l_days_of_the_week ty_tabtype;
begin
  l_days_of_the_week := ty_tabtype (

        'Sun',
        'Mon',
        'Tue',
        'Wed',
        'Thu',
        'Fri',
        'Sat'
  );
  -- let's print the values
  for d in l_days_of_the_week.FIRST .. l_days_of_the_week.LAST loop
        dbms_output.put_line ('Day '||d||' = '||
              l_days_of_the_week (d)
         );
  end loop;
```

```
end;
```

Output:

```
Day 1 = Sun
Day 2 = Mon
Day 3 = Tue
Day 4 = Wed
Day 5 = Thu
Day 6 = Fri
Day 7 = Sat
```

# Python

In Python, two datatypes are equivalents of these two PL/SQL constructs:

- **list**: A list of values
- **tuple**: Same as a list of values but the data can't be altered

In both cases, the variables are referenced by their position in the collection and are otherwise identical. Let's see the list datatype first. Here is how you assign the days of the week:

```
l_days_of_week = [
   'Sun',
   'Mon',
   'Tue',
   'Wed',
   'Thu',
   'Fri',
   'Sat'
   ]
```

Note the square brackets ("[" and "]"). If you want to just print the values, you can merely call `print` with the variable as an argument. Here is the complete program and its output:

```
# py4.txt
l_days_of_week = [
   'Sun',
   'Mon',
   'Tue',
   'Wed',
```

```
    'Thu',
    'Fri',
    'Sat'
    ]
print (l_days_of_week)
```

Output:

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

It's not quite as we wanted it. It should be like it was in the case of PL/SQL. To do that, we need to display the position of the elements in the list.

```
#py4a.txt
l_days_of_week = [
    'Sun',
    'Mon',
    'Tue',
    'Wed',
    'Thu',
    'Fri',
    'Sat'
    ]
for i in range(len(l_days_of_week)):
   print ('Day '+ str(i) + ' = '+l_days_of_week[i])
```

Output:

```
Day 0 = Sun
Day 1 = Mon
Day 2 = Tue
Day 3 = Wed
Day 4 = Thu
Day 5 = Fri
Day 6 = Sat
```

Let's dissect the Python program a bit:

- `len(l_days_of_week)` returns the length of the list, that is, the number of elements in it, equivalent to the PL/SQL `l_days_of_week.count`
- `l_days_of_week[i]` returns the element at the i-th position (remember, unlike PL/SQL, the first position is 0, not 1).
- The counter `i` starts with 0 and goes up to the total number of elements in the list.

## More Operation on Lists

Elements in the lists are referenced by the position, as shown in part 1. You can create a list from another. Here is an example where we create a variable to hold the first three days of the week.

```
>>> first_three_days_of_week = l_days_of_week [0:3]
>>> first_three_days_of_week
['Mon', 'Tue', 'Wed']
```

And we create another list--the last four days of the week:

```
>>> last_four_days_of_week = l_days_of_week [3:]
>>> last_four_days_of_week
['Thu', 'Fri', 'Sat', 'Sun']
```

We can combine these two lists:

```
>>> week_days = first_three_days_of_week + last_four_days_of_week
>>> week_days
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

We can delete specific elements, again referencing the position. To delete the first three elements, we do this:

```
>>>  del l_days_of_week [0:3]
```

To delete the entire list, call the same statement without any parameters.

```
>>>  del l_days_of_week
```

After doing that, if you reference the list, you will get an error:

```
>>> l_days_of_week

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    l_days_of_week
NameError: name 'l_days_of_week' is not defined
```

You can add values to the list using the `append` method.

```
>>> l_days_of_week.append('Sun')
```

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

Sometimes you might want to append a list to another. Suppose we have two lists: work_week and weekend.

```
>>> work_week = ['Mon','Tue','Wed','Thu','Fri']
>>> weekend = ['Sat','Sun']
```

We define another list called `week_days` from the list `work_week`.

```
>>> week_days = work_week
>>> week_days
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
```

Later, we want to add weekend days to this list as well. Instead of appending one day at a time from the list `weekend`, we can add the entire list to this list `week_days`.

```
>>> week_days.extend(weekend)
>>> week_days
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

Remember, you could also join two lists using the "+" operator, shown earlier. The difference between this approach and merely joining the two lists by concatenation ("+") shown earlier is that this approach extends the *current* array, instead of having to store the results of the concatenation in *another variable*.

But you can even avoid that using another trick you learned earlier.

```
>>> week_days += work_week
```

So, in effect, `week_days.extend(work_week)` is functionally the same as `week_days += work_week`.

What if you want to add the new value at a specific point in the list instead of appending at the end? For instance, suppose you have left out the "Thu" value from `work_week` and want to add it after the "Wed" value. You can use `insert` in that case to insert a value at a position. Since the positions start at 0, "Fri" is at position 3, where the new value should go.

```
>>> week_days = ['Mon', 'Tue', 'Wed', 'Fri', 'Sat','Sun']
>>> week_days
['Mon', 'Tue', 'Wed', 'Fri', 'Sat', 'Sun']
>>> week_days.insert (3,'Thu')
>>> week_days
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

Sometimes you want to remove the first occurrence of an item in a list. For instance, suppose you want to remove "Sun" from the list. If there were multiple occurrences of "Sun," the first one would have been removed.

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

Now if you check the variable, the item is gone:

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

If you attempt to remove the same element again by doing this:

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

You will get an error:

```
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    l_days_of_week.remove('Sun')
ValueError: list.remove(x): x not in list
```

Let's quickly scan through some of the other important methods of the list class.

## Pop

It removes the last element from the collection and returns it.

```
>>> l_days_of_week.pop()
'Sat'
>>> l_days_of_week.pop()
'Fri'
```

And so on. It might not be worth simply displaying the element. You want to put the element in some variable.

```
>>> d1 = l_days_of_week.pop()
>>> d1
'Thu'
```

After issuing these operations, this is what you'll see if you take a look in the original collection:

```
>>> l_days_of_week
['Sun', 'Mon', 'Tue', 'Wed']
```

You can also pop an element at a specific position, for example, position 2, by passing it as an argument.

```
>>> l_days_of_week.pop(2)
'Tue'
```

The methods `pop()` and `append()` can be used to build a stack data structure.

## Clear

Use this to remove the list altogether.

## Index

Use this to find the position of a specific item in the list. Remember, the first position is 0.

```
>>> l_days_of_week.index('Tue')
2
```

## Count

Use this to determine how many times certain data appears in the list.

```
>>> l_days_of_week.count('Tue')
1
```

## Sort

Use this to sort or arrange the items in an order. The default is to sort in ascending order.

```
>>> l_days_of_week.sort(key=None)
>>> l_days_of_week
['Fri', 'Mon', 'Sat', 'Sun', 'Thu', 'Tue', 'Wed']
```

Do this to do a reverse sort:

```
>>> l_days_of_week.sort(key=None,reverse=True)
```

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

### Reverse

Here's a second way to re-sort the values in the list in a reverse order:

```
>>> l_days_of_week.reverse()
>>> l_days_of_week
['Wed', 'Tue', 'Thu', 'Sun', 'Sat', 'Mon', 'Fri']
```

## Variable-Style Behavior

Just like PL/SQL, we can treat the Python collection variables as any other scalar variables. For instance, we can compare them directly.

```
>>> week_days
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> days_of_week
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> days_of_week == week_days
True
```

The code above returns "True," which means these two lists are the same. We can also use other comparisons in similar manner.

```
>>> work_week
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
>>> week_days
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> work_week < week_days
True
```

To illustrate the concept of collection variables, let's build a small program to check if a particular day is a working day or not. Save the following code in a file named `myfirst.py`.

```
# py5.txt
workWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
dayTocheck = input ('Enter the Day to check: ')
```

```
if (dayTocheck in workWeek):
   print (str(dayTocheck) + ' is a working day')
else:
   print (str(dayTocheck) + ' is a not working day')
print ('Thank you for using my app')
```

Executing it produces this:

```
C:\>python py5.txt
Enter the Day to check: Sat
Sat is a not working day
Thank you for using my app

C:\>python py5.txt
Enter the Day to check: Thu
Thu is a working day
Thank you for using my app
```

## Set Operations on Collections

Since collections are sets, we can use direct set options such as UNION, INTERSECT, and so on with them as well. Let's see some set operations in PL/SQL and explore the same in Python.

### PL/SQL

```
-- pl6.sql
declare
   l_string    varchar2(32000);
   type ty_tabtype is table of varchar2(30);
   week_days ty_tabtype;
   work_week ty_tabtype;
   first_half ty_tabtype;
   second_half ty_tabtype;
   middle_week ty_tabtype;
   temp_days   ty_tabtype;
begin
   week_days := ty_tabtype ('Sun','Mon','Tue','Wed','Thu','Fri','Sat');
   work_week := ty_tabtype ('Mon','Tue','Wed','Thu','Fri');
   first_half := ty_tabtype ('Sun','Mon','Tue');
```

```
  second_half := ty_tabtype ('Wed','Thu','Fri','Sat');
  middle_week := ty_tabtype ('Tue','Wed','Thu');
  -- let's print the values
  dbms_output.put_line('All Week Days');
  l_string := null;
  for i in week_days.FIRST .. week_days.LAST loop
     l_string := l_string||' '||week_days(i);
  end loop;
    dbms_output.put_line ('...'||l_string);
  dbms_output.put_line('Work Week');
  l_string := null;
  for i in work_week.FIRST .. work_week.LAST loop
     l_string := l_string||' '||work_week(i);
  end loop;
  dbms_output.put_line ('...'||l_string);
  dbms_output.put_line('First Half');
  l_string := null;
  for i in first_half.FIRST .. first_half.LAST loop
     l_string := l_string||' '||first_half(i);
  end loop;
  dbms_output.put_line ('...'||l_string);
  dbms_output.put_line('Second Half');
  l_string := null;
  for i in second_half.FIRST .. second_half.LAST loop
     l_string := l_string||' '||second_half(i);
  end loop;
  dbms_output.put_line ('...'||l_string);
  dbms_output.put_line('Middle Week');
  l_string := null;
  for i in middle_week.FIRST .. middle_week.LAST loop
     l_string := l_string||' '||middle_week(i);
  end loop;
  dbms_output.put_line ('...'||l_string);
  --
  -- Let's perform some operations now:
  --
  dbms_output.put_line('Union');
  l_string := null;
  temp_days := first_half multiset union second_half;
  for i in temp_days.FIRST .. temp_days.LAST loop
     l_string := l_string||' '||temp_days(i);
  end loop;
  dbms_output.put_line ('...'||l_string);
```

```
    dbms_output.put_line('Union of three');
    l_string := null;

    temp_days := first_half multiset union second_half multiset union middle_week;
    for i in temp_days.FIRST .. temp_days.LAST loop
        l_string := l_string||' '||temp_days(i);
    end loop;
    dbms_output.put_line ('...'||l_string);
    dbms_output.put_line('Union distinct of three');
    l_string := null;
    temp_days := first_half multiset union distinct second_half multiset union distinct middle_week;
    for i in temp_days.FIRST .. temp_days.LAST loop
        l_string := l_string||' '||temp_days(i);
    end loop;
    dbms_output.put_line ('...'||l_string);
end;
/
```

Output:

```
All Week Days
... Sun Mon Tue Wed Thu Fri Sat
Work Week
... Mon Tue Wed Thu Fri
First Half
... Sun Mon Tue
Second Half
... Wed Thu Fri Sat
Middle Week
... Tue Wed Thu
Union
... Sun Mon Tue Wed Thu Fri Sat
Union of three
... Sun Mon Tue Wed Thu Fri Sat Tue Wed Thu
Union distinct of three
... Sun Mon Tue Wed Thu Fri Sat
```

## Python

Set arithmetic is not allowed on the `list` datatype; only variables of `set` datatype can be subject to that. Remember from our earlier discussion that `list` can contain duplicates; `set` can't.

A union is done in one of the following two ways:

- The "|" (the pipe character) operator
- The `union` method

Here is an example from the Python command prompt.

```
week_days = {'Sun','Mon','Tue','Wed','Thu','Fri','Sat'}
work_week = {'Mon','Tue','Wed','Thu','Fri'}
first_half = {'Sun','Mon','Tue'}
second_half = {'Wed','Thu','Fri','Sat'}
middle_week = {'Tue','Wed','Thu'}
print(first_half|second_half)
print(first_half.union(second_half))
```

Output:

```
{'Sun', 'Wed', 'Tue', 'Mon', 'Thu', 'Fri', 'Sat'}
```

You can create a union of all the three sets as well.

```
print(first_half|middle_week|second_half)
```

Output:

```
{'Sun', 'Wed', 'Tue', 'Thu', 'Sat', 'Mon', 'Fri'}
```

Since sets can't contain duplicates, Python removes the duplicates automatically to return a valid set. Since a set is also ordered, Python sorts the data into an ordered list as well. Look at the following:

```
print(second_half.union(first_half))
```

Output:

```
{'Sun', 'Wed', 'Tue', 'Mon', 'Thu', 'Fri', 'Sat'}
```

It produces the same result, in the same order. So Set 1 UNION Set 2 is the same as Set 2 UNION Set 1. Notice the most important difference here. UNION in PL/SQL does not remove the duplicates and does not

sort them; Python does. If you wanted the Python-like functionality in PL/SQL, you should have used UNION DISTINCT instead.

Like performing a union, getting the intersection is possible in sets using the "&" operator or the INTERSECTION method. Here is an example.

```
>>> first_half&middle_week
{'Tue'}
>>> first_half.intersection(middle_week)
{'Tue'}
```

Other set operations are also dramatically simple in Python. Here is the DIFFERENCE functionality using two different approaches:

```
>>> first_half - middle_week
{'Sun', 'Mon'}
>>> first_half.difference(middle_week)
{'Sun', 'Mon'}
```

Another type of difference is symmetric difference, which is the items present in either Set 1 or Set 2, but *not both*. You can accomplish that using the "^" operator or the `symmetric_difference` method.

```
>>> first_half ^ middle_week
{'Thu', 'Sun', 'Wed', 'Mon'}
>>> first_half.symmetric_difference(middle_week)
{'Thu', 'Sun', 'Wed', 'Mon'}
```

It's not practical to list all the set operations in this short article. Refer to the documentation for a full list. Sets are not just for the set operations; they can be used as regular data items as well. For instance, if you want to examine whether a set is a superset of another, you can use ">=":

```
>>> first_half >= middle_week
False
>>> week_days >= middle_week
True
```

Remember ">" is the test for a "proper" superset, that is, S1 > S2 and S1 != S2.

Likewise, the subset operator is "<=":

```
>>> middle_week <= week_days
True
```

How about checking whether two sets are the same?

```
>>> days_of_the_week = week_days # assigned the same value to a new variable
>>> days_of_the_week == week_days
True
```

# VARRAYs

The third type of collections in PL/SQL is called VARRAY. It's similar to nested tables, except that the maximum number of elements is fixed. The actual number of elements vary and depend on the elements added or subtracted at runtime. Here is an example:

```
-- pl7.sql
declare
  type ty_weekdays_list is varray(7) of varchar2(3);
  v_week_days ty_weekdays_list;
  v_count number;
begin
  v_week_days := ty_weekdays_list('Sun','Mon','Tue','Wed','Thu','Fri','Sat');
  v_count := v_week_days.count;
  for i in 1..v_count loop
       dbms_output.put_line(i||'='||v_week_days(i));
  end loop;
end;
/
```

Output:

```
1=Sun
2=Mon
3=Tue
4=Wed
5=Thu
6=Fri
7=Sat
```

This is along the same lines as the list in Python, although there is no limit for the number of elements. Here is the Python equivalent:

```
>>> v_week_days = ['Sun','Mon','Tue','Wed','Thu','Fri','Sat']
>>> for i in range(len(v_week_days)):
```

```
...     print(i,v_week_days[i])
...
0 Sun
1 Mon
2 Tue
3 Wed
4 Thu
5 Fri
6 Sat
```

Note that the indexing starts at 0, unlike PL/SQL which starts at 1. You rarely use VARRAYs in real life; most collections can be easily done with associative arrays and those are more powerful anyway. So, there is no reason to extend the Python topic on this PL/SQL component.

# Summary

In summary, the most used collections in PL/SQL are the following two types. The Python equivalents are give next to them.

| **PL/SQL** | **Python** |
|---|---|
| Nested table, where the elements are ordered and can be referenced by their position in the collection using an index, which starts at 1.<br><br>Example:<br><br>declare<br>  type ty_weekdays_list is table of varchar2(3);<br>  v_week_days ty_weekdays_list;<br>  v_count number;<br>begin<br>  v_week_days := ty_weekdays_list('Sun','Mon','Tue','Wed','Thu','Fri','Sat');<br>  v_count := v_week_days.count;<br>  for i in 1..v_count loop<br>  dbms_output.put_line(i\|\|'='\|\|v_week_days(i));<br>  end loop;<br>end;<br><br>Here each element (the three-letter shortened day names) can be referenced only by the position, that is, `v_week_days(i)`. | Three types:<br><br>• **list**: The elements can be manipulated.<br>• **tuple**: The elements are read-only.<br>• **set**: The elements have no duplicate and are sorted.<br><br>The index starts at 0. The datatypes are not required.<br><br>Example:<br><br>v_week_days = ['Sun','Mon','Tue','Wed','Thu','Fri','Sat']<br>for i in range(len(v_week_days)):<br>  print(i,v_week_days[i])<br><br>In the case of tuple, just replace this:<br><br>`v_week_days = ['Sun','Mon','Tue','Wed','Thu','Fri','Sat']`<br><br>With this: |

| | |
|---|---|
| | `v_week_days = ('Sun','Mon','Tue','Wed','Thu','Fri','Sat')`<br><br>There is no PL/SQL equivalent for **set**. |
| Associative array, also called PL/SQL Tables, where the elements are not ordered. The index can be any arbitrary number or string. This is very similar to a key-value pair. Here is an example:<br><br>declare<br>  type ty_weekdays_list is table of varchar2(9)<br>  index by varchar2(3);<br>  v_week_days ty_weekdays_list;<br>  v_count number;<br>  v_abbr  varchar2(3);<br>begin<br>  v_week_days('Sun') := 'Sunday';<br>  v_week_days('Mon') := 'Monday';<br>  v_week_days('Tue') := 'Tuesday';<br>  v_week_days('Wed') := 'Wednesday';<br>  v_week_days('Thu') := 'Thursday';<br>  v_week_days('Fri') := 'Friday';<br>  v_week_days('Sat') := 'Saturday';<br>  -- Now list them:<br>  v_abbr := v_week_days.first;<br>  while v_abbr is not null loop<br>  dbms_output.put_line(v_abbr\|\|':'\|\|<br>  v_week_days(v_abbr));<br>  v_abbr := v_week_days.next (v_abbr);<br>  end loop;<br>end;<br><br>Note the differences in indexing. PL/SQL indexes are references inside normal parentheses: "(" and ")." Python's are square brackets: "[" and "]." | Called a dictionary, where the key and value pairs are clearly mentioned. It's not ordered.<br><br>>>> v_week_days = {'Sun':'Sunday',<br>...  'Mon':'Monday',<br>...  'Tue':'Tuesday',<br>...  'Wed':'Wednesday',<br>...  'Thu':'Thursday',<br>...  'Fri':'Friday',<br>...  'Sat':'Saturday'<br>... }<br><br>To display the variable, you can simply type the name of the variable at the Python command prompt:<br><br>`>>> v_week_days`<br><br>Or, you can iterate inside a Python program:<br><br>for v_abbr in v_week_days.keys():<br>  print(v_abbr+':'+v_week_days[v_abbr]) |

# 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.