Python for Oracle Geeks

Even the most focused Oracle professional can't live at the *SQL>* prompt forever; every database career sometimes involves working outside the database. Learning a general-purpose programming language adds greatly to an Oracle professional's effectiveness. The Python programming language is a very attractive option to meet this need. This article outlines Python's advantages, demonstrates the essentials of writing Python, and supplies details on using Python against an Oracle database.

Why Python?

PL/SQL is a great programming language for use within the database, but all Oracle professionals run up against its limitations at some point. Manipulating files on disk, calling batchfiles or shell scripts, complex text parsing, heavily object-oriented work, and many other tasks can be difficult or even impossible in PL/SQL.

Of course, most of us are too busy to invest large amounts of time in another language. For us, an ideal second language should be very easy to learn and use and should help us write highly readable, easily maintainable code, so that we can get value out of it without losing our focus on the database. On the other hand, it should be powerful and versatile, so that it will meet all our needs and won't force us to go looking for yet more languages to cover its gaps. Finally, it would be nice if it were free, to spare us the hassle of getting permission for a new purchase.

Python meets all these requirements. It is an open-source, object-oriented, high-level, general-purpose interpreted language available without cost on all common computing platforms. Its elegant, straightforward syntax helps programmers write clean, readable code.

Rather than try to introduce Python thoroughly or teach it rigorously, this article will jump right into demonstrating its use in some Oracle-related tasks. If it convinces you that Python is worth checking out further, the links listed at the end of the article can guide you.

Getting started

If you're running Linux, you may already have Python installed. Otherwise, download it from www.python.org; convenient Windows and RPM installers are available. After installing Python, make sure that the Python directory is in your PATH, then go to a command prompt and type 'python' to start the Python interpreter. You should see something like:

```
c:\>python
Python 2.4 (#60, Nov 30 2004, 11:49:19) [MSC v.1310 32 bigt (Intel)] on
win32
Type "help", "copyright", "credits", or "license" for more information.
>>>
```

Python commands are typically combined into text files (called modules or scripts) and run together, but they can also be issued ad-hoc at the interpreter prompt. Let's get some instant gratification by issuing our commands interactively.

In obedience to tradition, type at the triple-arrow Python prompt:

```
>>> print 'Hello, World'
```

The interpreter will respond with

```
Hello, World
```

…unless you typed *Print* or *PRINT*. If you did, you have learned that everything in Python - commands, variable names, etc. - is case-sensitive.

Working with a text file

Suppose that you have a standard init.ora file defining default parameters for all your databases. You wish to compare its contents with the init.ora for a particular database instance.

Listing 1. init_default.ora

```
DB_BLOCK_SIZE=4096
COMPATIBLE=9.2.0.0.0
SGA_MAX_SIZE=104857600
SHARED_POOL_SIZE=50331648
```

Listing 2. init_orcl.ora

```
DB_BLOCK_SIZE=8192
COMPATIBLE=9.2.0.0.0
UNDO_MANAGEMENT=AUTO
SGA_MAX_SIZE=135339844
SHARED_POOL_SIZE=50331648
FAST_START_MTTR_TARGET=300
```

Begin by opening init_orcl.ora for reading.

```
>>> initFile = open('init_orcl.ora', 'r')
```

You have now opened the file and assigned a variable, initFile, to refer to it. Note that we didn't have to declare initFile or specify in advance what type of data initFile would hold; Python is a "dynamically typed" language, unlike PL/SQL, Java, and C.

Let's see what we have.

```
>>> firstLine = initFile.readline()
>>> firstLine
'DB_BLOCK_SIZE=8192\n'
```

Here, readline() is a method defined on the object initFile. If you are unfamiliar with object-oriented programming, this will be a new concept to you, but the Python language provides a great place to get familiar with it.

Introspection

The \n at the end of firstLine is a newline character, and we don't want it. How can we get rid of it? In Python, a string like firstLine is an object. As an object, it has methods defined for it, and it can be inspected with Python's introspection capabilities. For example, the *dir* function returns a list of the attributes and methods defined for an object.

```
>>> dir(firstLine)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__str__', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rstrip',
```

```
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Glancing through this list, we see *strip* - that looks right. Let's find out by printing strip's *__doc__* attribute, where a brief documentation string is stored by convention for every method. (Names like *__doc__* that begin and end with two underscores are used for special system-defined methods and attributes.)

```
>>> print firstLine.strip.__doc__
S.strip([chars]) -> string or unicode

Return a copy of the string S with leading and trailing whitespace
removed.
If chars is given and not None, remove characters in chars instead.
If chars is unicode, S will be converted to unicode before stripping

>>> firstLine = firstLine.strip()
>>> firstLine
'DB_BLOCK_SIZE=8192'
```

Next, we want to distinguish the parameter in firstLine from its value. It's tempting to show off Python's powerful regular expression handling abilities, but for now let's stick with a simpler way: the string method split().

```
>>> firstLine.split('=')
['DB_BLOCK_SIZE', '8192']
```

Variables and assignment

Calling split() produced a list of the strings preceding and following the dividing character, "=". Unfortunately, we didn't assign this result to any variables, so the result was dumped on the screen and then forgotten. This time, we'll use multiple assignment to capture both results at once. You may want to use the up-arrow key to save yourself some typing.

```
>>> param, val = firstLine.split('=')
>>> param
'DB_BLOCK_SIZE'
>>> val
'8192'
```

Actually, we'll want to store the values for each of several parameters. This is a great place to use a powerful Python variable type called a *dictionary*. A dictionary is an unordered set of *key: value* pairs. The key can be any string, number, or user-defined object; the value can be virtually anything. We'll create an empty dictionary and populate it with what we've extracted so far.

```
>>> initParams = {}
>>> initParams[param] = val
>>> initParams
{'DB_BLOCK_SIZE': '8192'}
```

Now grab another line from the open file and store it in the dictionary as well. This time we'll chain strip() directly onto the end of readline(), much as if we were using a Unix pipe.

```
>>> nextLine = initFile.readline().strip()
>>> param, val = nextLine.split('=')
>>> initParams[param] = val
>>> initParams
{'DB_BLOCK_SIZE': '8192', 'COMPATIBLE': '9.2.0.0.0'}
```

Writing scripts

Now that we've practiced interactively with the interpreter, we're ready to write a Python script to handle the whole file. Use Ctrl-Z to exit the Python interpreter, and create a text file - call it *readInitOra.py*.

Listing 3. readInitOra.py

```
initFile = open('init_orcl.ora', 'r')
initParams = {}
rawTextLine = initFile.readline()
while rawTextLine:
  param, val = rawTextLine.strip().split('=')
  initParams[param] = val
  rawTextLine = initFile.readline()
print initParams
```

As you read this code, you are most likely thinking, "Where are the semicolons? Where are the curly brackets?" They aren't used in Python. When you read code, you expect to see one command per line, and you expect blocks (like the *while* loop above) to be indented. Python reads your code exactly the same way your eye does!

This may seem startling, but it can be a real advantage. In PL/SQL and many other languages, you would use curly brackets to denote blocks of code to the compiler, while using indentations to help you read the code with your own eye. Unfortunately, it's easy to mismatch curly brackets and indentations, and when you do, you're telling one story to the compiler and a different story to yourself. This can create cruelly elusive errors.

Likewise, although the PL/SQL compiler expects a command to end with a semicolon, humans reading code expect to see one command per line. The Python interpreter, like a human reader, expects commands to be separated by line breaks.

Let's see the code work. At the operating system command prompt (not the Python interpreter prompt), type

```
c:\> python compareInitOra.py
{'UNDO_MANAGEMENT': 'AUTO', 'COMPATIBLE': '9.2.0.0.0', 'DB_BLOCK_SIZE':
'8192', 'FAST_START_MTTR_TARGET': '300', 'SGA_MAX_SIZE': 157286400,
'SHARED_POOL_SIZE': '50331648'}
```

If, out of habit, you indented all the lines in compareInitOra.py a few spaces, you confused Python and received a syntax error. Go back and make sure that each line begins in column 1, unless you specifically mean to indent it as part of a block.

We'll actually want to use this code in a couple different places, so let's change it from a simple script to the definition of a function that accepts a parameter.

Listing 4. readInitOra.py (with function definition)

```
def read(fileName):
  initFile = open(fileName, 'r')
  initParams = {}
  rawTextLine = initFile.readline()
  while rawTextLine:
    param, val = rawTextLine.strip().split('=')
    initParams[param] = val
    rawTextLine = initFile.readline()
  return initParams
```

Nesting

Next we need to create a similar dictionary containing our default parameters from init_default.ora. We could read them into brand-new variables, of course, but instead let's show off how nicely objects nest in Python. We'll create a single parent directory *initParams* and nest a directory within it for each init.ora file. We'll also *import* the file we just wrote so that we can call its read() function. Create a new text file called compareInitOra.py.

Listing 5. compareInitOra.py

```
import readInitOra
initParams = {}
# define a list of filenames to loop through
fileNames = ['init_orcl.ora', 'init_default.ora']
for fileName in fileNames:
  initParams[fileName] = readInitOra.read(fileName)
print initParams
```

```
c:\> python compareInitOra.py
{'init_orcl.ora':
  {'UNDO_MANAGEMENT': 'AUTO', 'COMPATIBLE': '9.2.0.0.0',
'DB_BLOCK_SIZE': '8192', 'FAST_START_MTTR_TARGET': '300',
'SGA_MAX_SIZE': '157286400, 'SHARED_POOL_SIZE': '50331648'}
'init_default.ora':
  {'COMPATIBLE': '9.2.0.0.0', 'DB_BLOCK_SIZE': '4096',
'FAST_START_MTTR_TARGET': '300', 'SGA_MAX_SIZE': '100663296',
'SHARED_POOL_SIZE': '50331648'}}
```

I've added some whitespace to the output this time, to help you see the nested structure. We could easily write Python code to print it prettily, of course, but we're database people - so let's get this data into an Oracle database instead.

Issuing SQL through Python

To access a database, your Python interpreter needs to have a database module installed. You have many choices, all of which conform to a standardized API specification and will look very familiar to anyone experienced with using ODBC or JDBC programatically. We'll use cx_Oracle for its ease of installation. Just go to http://www.computronix.com/utilities.shtml and download a Windows installer or a Linux RPM file matching your versions of Python and Oracle.

After cx_Oracle is installed, let's go back to the Python command-line interpreter to try it out.

Because cx_Oracle is a module separate from the core Python language, we must import it before using it in any given session or script.

```
>>> import cx_Oracle
```

Remember to watch your capitalization! Now let's create a table to store our results in.

```
>>> orcl = cx_Oracle.connect('scott/tiger@orcl')
>>> curs = orcl.cursor()
>>> sql = """CREATE TABLE INIT_PARAMS
... ( fileName VARCHAR2(30),
...   param VARCHAR2(64),
...   value VARCHAR2(512) )"""
```

The triple-quote (""") is a handy syntax for entering strings that include line breaks. The Python interpreter changes its prompt from >>> to ... to remind you that you're continuing input begun on an earlier line.

```
>>> curs.execute(sql)
>>> curs.close()
```

Now that our table is ready, let's modify recordInitOra.py to populate it.

Listing 6. recordInitOra.py

```
import readInitOra, cx_Oracle
initParams = {}
fileNames = ['init_orcl.ora', 'init_default.ora']
for fileName in fileNames:
  initParams[fileName] = readInitOra.read(fileName)
orcl = cx_Oracle.connect('scott/tiger@orcl')
curs = orcl.cursor()
for fileName in initParams.keys():
  for param in initParams[fileName].keys():
    value = initParams[fileName][param]
    sql = """INSERT INTO INIT_PARAMS VALUES
          (:fileName, :param, :value)"""
    bindVars = {'fileName': fileName,
                'param': param, 'value': value}
    curs.execute(sql, bindVars)
curs.close()
orcl.commit()
```

That’s all it takes! Note that, this time, we used bind variables in our SQL string and supplied values for them in a separate dictionary. Using bind variables helps keep us out of trouble with the SPCSP (Society for the Prevention of Cruelty to the Shared Pool).

Getting results from a query is just a little more complicated. After calling execute() on a cursor object, we can either fetchone() to get one row at a time or fetchall() to get a list of all rows. In either case, each row takes the form of a *tuple* – an ordered series of values that can be accessed by a numerical index. For example, this script will print out *init.ora* parameters that conflict with the current values in V$PARAMETER.

Listing 7. compareLiveParams.py

```
import readInitOra, cx_Oracle
def readLiveParams():
  liveParams = {}
  orcl = cx_Oracle.connect('scott/tiger@orcl')
  curs = orcl.cursor()
  curs.execute('SELECT name, value FROM V$PARAMETER')
  row = curs.fetchone()
  while row:
    (param, val) = (row[0], row[1])
    liveParams[param.upper()] = val
    row = curs.fetchone()
  return liveParams
liveParams = readLiveParams()
fileName = 'init_orcl.ora'
fileParams = readInitOra.read(fileName)
for (param, val) in fileParams.items():
  liveVal = liveParams.get(param)
  if liveVal != val:
    print """For %s, V$PARAMETER shows %s,
             but the file %s shows %s""" % \
          (param, liveVal, fileName, val)
```

This script introduces a few tricks you haven’t seen yet.

- Calling items() on the dictionary *fileParams* returns a list of (key, value) pairs. These can be looped through together by specifying two loop variables in the *for* statement.
- Calling liveParams.get(param) works like liveParams[param], except that liveParams[param] returns an error if param is not found in liveParams; liveParams.get(param) returns 'None' in this case.
- Python can use the *%* operator for string substitution. As in C's *printf*, *%s* indicates that a value will be inserted in string form at that point. The values are taken, in order, from the tuple that follows the *%*.
- The last line of code runs longer than we want to type without a line break, so we use a backslash to make an exception to Python's usual rule of interpreting a line break as the end of a command.

Conclusion

I hope you've become intrigued by Python's ease, elegance, and readability. What you haven't seen yet is Python's power. Its capabilities include elegant exception handling, unit testing, object orientation, functional programming, GUI toolkits, web frameworks, XML, web services... virtually everything programmers do. You won't need to "graduate" to a different language as your work gets more advanced.

References

http://www.python.org/ - The official central Python site. Downloads, tutorials, documentation, etc.
http://www.python.org/topics/database/ - Central point for Python database work.
http://www.amk.ca/python/writing/DB-API.html - A tutorial on querying databases with Python.
http://www.python.org/sigs/db-sig/ - A special interest group for Python – database work.
http://www.ibm.com/developerworks/ - the "Charming Python" series of articles here provides excellent, accessible introductions to many specialty Python topics.
http://www.pythonware.com/daily/ - aggregated Python newsblogs. This is a good place to learn of useful new techniques and modules.

Oracle database modules

http://www.zope.org/Members/matt/dco2
http://www.computronix.com/utilities.shtml

O/R Mapping Tools

You may be interested in object-relational mapping tools, which can take over the writing of SQL and provide an object-oriented interface for the programmer. Oracle's TopLink is an example of an object-relational mapper for Java. Some Oracle-compatible ORM tools for Python are at

http://modeling.sourceforge.net/
http://opensource.theopalgroup.com/
http://soiland.no/software/forgetsql
http://skunkweb.sourceforge.net/pydo.html
http://www.livinglogic.de/Python/orasql/

Catherine Devlin is an Oracle Certified Professional with six years of experience as an all-purpose DBA, PL/SQL developer, distributed systems architect, and web application developer for several small-scale Oracle OLTP systems. She was introduced to Python less than two years ago. She works for IntelliTech Systems in Dayton, Ohio and can be reached at catherine.devlin@gmail.com.