Python Crash Course

Ole Nielsen*
School of Mathematical Sciences
Australian National University, Canberra
THE AUSTRALIAN NATIONAL UNIVERSITY
Ole.Nielsen@anu.edu.au

April 29, 2002

Introduction

This text provides an informal introduction to selected parts of the programming language Python for use in the ANU, SMS Summer school 2002. It should take about 40 minutes to complete most of it.

A more comprehensive tutorial is available at

`http://www.python.org/doc/current/tut/tut.html (Chapter 3)`

The site `www.python.org` itself contains everything worth knowing about Python. You won't need to access any of the WEB resources given here to complete the demo — they are included for your reference only.

*A big thanks to Agnes Boskovitz, RSISE, ANU for suggestions greatly improving this tutorial.

1 A first glance

The first thing to try is to invoke the Python interpreter by typing the command

```
python
```

You should see the following message and the python prompt (>>>):

```
Python 2.1 (#1, Nov 20 2001, 09:58:43) [C] on osf1V5
Type "copyright", "credits" or "license" for more information.
>>>
```

Python is now ready to take your order.

Python can do calculations like in most other languages. For example to compute the interest earned in half a year on 5000 dollars with a rate of 4.8% per year, type:

```
>>> principal = 5000
>>> rate = 4.8
>>> years = 0.5
>>> interest = principal * (1+rate/100)**years - principal
>>> interest
118.59355682789101
```

The last command causes Python to print the result to the screen.

Python is often used to manipulate texts. Let us try print a statement from the bank. For example

```
>>> 'Initial deposit: $%d' %(principal)
'Initial deposit: $5000'
```

Notice how the value of the decimal number `principal` is inserted in the text instead of the *placeholder* %d. This is an example of *string interpolation*. Other placeholders are %f for floating point numbers and %s for strings[1]. A complete list of placeholders is available at `http://www.python.org/doc/current/lib/typesseq-strings.html`. A key facet of Python's string interpolation is that it can occur anywhere that a string can.

Back to the example: Let us type

[1] These codes are very similar to those used in the programming language C

```
>>> s = 'Initial deposit: $%d' %(principal)
>>> s = s + '\n'
>>> s = s + 'Interest in %d months: $%.2f.' %(years*12, interest)
>>> print s
Initial deposit: $5000
Interest in 6 months: $118.59.
```

Note that text strings can be assigned to variables and concatenated using '+'. The special character '`\n`' is turned into a newline when the string is printed. The first number (years*12) is inserted in the placeholder %d the second in %.2f. The rounded to 2 decimals.

Strings can be indexed to extract any characters or substrings. The first character in string has index 0 and the last has index -1. Substrings can be specified with the *slice notation*: two indices separated by a colon.

```
>>> s[0]
'I'
>>> s[0:3]
'Ini'
>>> s[3:5]
'ti'
>>> s[-1]
'.'
>>> s[-4:-1]
'.59'
```

Exercise: (optional) *Suppose you wish to create a string like*

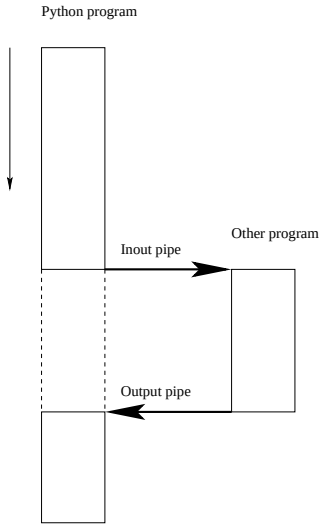
```
'primes := select(isprime, [seq(i,i=1..100)]):'
```

This happens to be valid command in the programming language 'Maple' which we will encounter later. For now just think of it as an arbitrary text string. Now suppose that we wish the first number to be taken from a variable called `lower` *and the second from a variable* `upper`.

Write a Python statement creating such a string by replacing appropriate placeholders with the values of `lower` *and* `upper` *respectively.*

2 Pipes

Python can call upon other programs to perform tasks for it through the use of pipes. A pipe is essentially a channel in which one program can communicate with another by sending and receiving text strings.



A simple example is to let Python call up the standard UNIX command `wc -w` (word count), supply some input through the input pipe and retrieve the result through the output pipe. The command is used to count the number of words in the input string[2]. First we need to open the pipes:

```
>>> from popen2 import popen2
>>> output, input = popen2('wc -w')
```

[2]If you don't know the command wc -w, you can try to run it from the UNIX command line, then type some words, end by typing CTRL-d, and wc will tell you how many words you typed.

The command `popen2` will start the supplied program ('wc -w'), associate the two pipes with its output and input, return them to two variables on the left hand side of =, and then wait. `popen2` is not available to the basic Python interpreter so it needs to be imported from the module of the same name. This happens in the first line. Python comes with a very large number of such modules - they are listed at `http://www.python.org/doc/current/lib/modindex.html`.

Next thing is to is to *write* some text to the input pipe using the command `input.write()` and then close it to let the external program know that we are finished. This is accomplished by the command `input.close()`. To retrieve the result through the output pipe we use as you guessed `output.read()`. Here is the code:

```
>>> input.write('Now to something completely different')
>>> input.close()
>>> count = output.read()
>>> print count
         5
```

2.1 Another — not quite as useless — example

Now suppose we wish to compute all prime numbers in some interval for our Python code. Python is a general purpose language and does not have primality tests. On the other hand we know that Maple which is specialised for mathematical problems can do it by using the code from the exercise above. We can get the best of both worlds — Pythons power and generality combined with Maple's specialised functions — through the use of pipes. First we build the Maple command as a Python string using values from the variables lower and upper denoting the desired interval.

```
>>> lower = 1
>>> upper = 50
>>> s = 'primes := select(isprime, [seq(i,i=%d..%d)]):' %(lower, upper)
>>> s = s + '\nprint(primes):'
>>> print s
primes := select(isprime, [seq(i,i=1..50)]):
print(primes):
```

Now we have our Maple command. Next thing is to open pipes to Maple, issue the command and retrieve the result. Here we go.

```
>>> maple_out, maple_in = popen2("maple -q")
>>> maple_in.write(s)
```

```
>>> maple_in.close()
>>> primes = maple_out.read()
>>> print primes.strip()
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Pretty simple really - except for that .strip() thing. All strings (as any other type) have functions associated with them which allow us to modify them - they are known as *methods*. The call `primes.strip()` gets rid of leading and trailing spaces in strings. Try printing without it. To find out what other methods are available in the string write `dir(primes)` (or any string in place of primes - e.g. dir('asdf')). Documentation of each of them is available at

```
http://www.python.org/doc/current/lib/string-methods.html#string-methods
```

Now leave the python prompt by pressing `CTRL-d`.

3 Python scripts

Writing commands directly to Python as we have done quickly becomes tedious for complex tasks. Python commands can therefore be collected in a file to form a *Python program*.

Your directory on the Alpha Server `Python_demos` contains a file called pipe_demo.py which is python program that calls up Maple to get all primes in a given interval using pipes as we just did. It then proceeds to call another programming language called R which is specialised in statistics to get the mean value and the median of the primes - again using a set of pipes. To execute, go to the directory and write

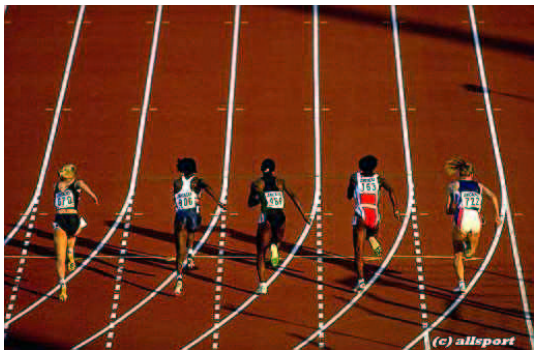
```
python pipe_demo.py
```

at the unix prompt. You can look or edit the file with an editor, for example

```
emacs pipe_demo.py &
```

The contents of pipe_demo.py is given in Section 5 for your reference.

4 Parallel Python



...and now to something completely different

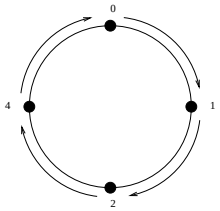
If you have more time you can have a look at the program `para_demo.py` available on the computer and in Section 6 which demonstrates how to run Python programs in parallel using *message passing* - a paradigm where processors communicate by explicitly sending and receiving data to and from each other. The Python module `pypar` developed here at the ANU implements message passing parallelism for Python.

The processors are numbered from 0 to $P-1$ where P is the number of processors in use. The program is started on all P processors using the command

```
prun -n 4 para_demo.py
```

where $P = 4$ in this example (you can try this!).

Each program must find out which number it is and how many processors are being used. This is done in the beginning of the program in the statements `numproc = pypar.size()` and `myid = pypar.rank()`. The program then prints a statement about who it is to the screen. In this demo a message is passed in a ring from processor 0 to processor 1 then on to 2 and so forth until the last processor sends a message back to processor 0. Not in itself a very useful parallel program but often used to measure communication speed on parallel machines.



The program running on processor 0 initiates this communication as follows:

```
if myid == 0:
  msg = "Anybody out there?"
  pypar.send(msg, 1)
```

It makes a message and sends it to processor 1 using the command `send` for the module `pypar`. Here you also see an example of an *if-statement* in Python; the indented code following `if` is only executed in the program running on processor 0. After having sent its message processor 0 issues the command

```
  msg = pypar.receive(numproc-1)
```

which puts it to sleep until the last processor sends it something.

All other processors must receive a message from the one preceding them and pass it on to their successor (or processor 0 in case of the last one). The code is

```
else:
  source = myid-1
  destination = (myid+1)%numproc

  msg = pypar.receive(source)
  msg = msg + ' P' + str(myid) + '_OK'
  print 'Processor %d sending msg "%s" to %d' %(myid, msg, destination)
  pypar.send(msg, destination)
```

The `else:` statement indicates that the following indented code is only executed if myid $\neq 0$. It first identifies the preceding processor (source) and the following processor. The expression `(myid+1)%numproc` computes myid+1 *modulo* numproc i.e. it evaluates to myid+1 except in case of the last processor (myid = numproc-1) where it will evaluate to 0 as desired. Then it issues the receive command and waits. When the message arrives it adds its own id to it (as an example) and passes it on to destination using the send command. And that is really all there is to it.

The hard part in parallel programming, however, is to arrange communications and computations to make the parallel program more efficient than the sequential counterpart. But that is a different story.

Hope you enjoyed the demo !

5 pipe_demo.py

Here is the Python (+Maple and R) source code for the pipe demo. Lines starting with # are comments not executed by Python.

```
#!/bin/env python
############################################################
#
#  ANU Summer School in Computational Mathematics 2002
#
#  Example: Using Python to invoke Maple and R using pipes.
#
#  Author:  Ole Nielsen,  SMS, ANU, Jan. 2002.
#
############################################################
#
# The purpose of this code is to demonstrate how Python can be
# used to combine functionality from foreign packages into
# one single program by using bi-directional Unix pipes.
# Understanding pipes is crucial for appreciating the
# Master-Slave program presented on Thursday 17 January 2002.
#
#
# The code invokes Maple to generate a vector of all primes
# in a given interval [lower, upper]
#
# The result is then transformed into a set of R commands
# form computing the mean and the median of those primes.
# R is invoked and the results are written to the screen


from popen2 import popen2   # The pipe function

# Interval in which prime numbers are generated.
#
lower = 1
upper = 50
```

```
# Get primes in [lower, upper] from Maple
#
maple_cmd = """
  primes := select(isprime, [seq(i,i=%d..%d)]):
  print(primes):
  """ %(lower, upper)

print 'MAPLE COMMANDS:'
print maple_cmd

maple_out, maple_in = popen2("maple -q")
maple_in.write(maple_cmd)
maple_in.close()
primes = maple_out.read()

print 'MAPLE OUTPUT:'
print primes

primes = primes.strip()
print


# Get median and mean of primes from R
#
#
R_cmd = """
  primes <- c(%s)
  median(primes)
  mean(primes)
  """ %primes[1:-1]

print 'R COMMANDS:'
print R_cmd

print 'R OUTPUT:'
R_out, R_in = popen2("R --vanilla --quiet")
R_in.write(R_cmd)
R_in.close()
print R_out.read()
```

5.1 Output

Example output from the pipe demo is:

```
> python pipe_demo.py
MAPLE COMMANDS:

  primes := select(isprime, [seq(i,i=1..50)]):
  print(primes):

MAPLE OUTPUT:
            [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]



R COMMANDS:

  primes <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47)
  median(primes)
  mean(primes)

R OUTPUT:
>
>   primes <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47)
>   median(primes)
[1] 19
>   mean(primes)
[1] 21.86667
```

6 para_demo.py

Here is the Python source code for the parallel demo.

```
#!/bin/env python
#############################################################
#
#  ANU Summer School in Computational Mathematics 2002
#
#  Example: Running Python in parallel using MPI.
#
#  Author:  Ole Nielsen,  SMS, ANU, Jan. 2002.
#
#############################################################
#
# The purpose of this code is to demonstrate how Python can be
# used in parallel using MPI.
# Understanding parallel Python is desirable for appreciating the
# Master-Slave program presented on Thursday 17 January 2002.
#
# This demo passes messages on in a ring from processor n-1 to n starting
# and ending with processor 0.
# Each processor adds some text to the message before passing it on
# and writes a log statement to the screen.
#
# To execute on Alpha server:
#
#   prun -n 4 MPI_demo.py


import pypar  # The Python-MPI interface

numproc = pypar.size()
myid =    pypar.rank()
node =    pypar.Get_processor_name()

if numproc < 2:
  raise "MPI_demo must run on at least 2 processors"

print "I am proc %d of %d on node %s" %(myid, numproc, node)
```

```
if myid == 0:
  msg = "Anybody out there?"
  print 'Processor 0 sending message "%s" to processor %d' %(msg, 1)
  pypar.send(msg, 1)
  msg = pypar.receive(numproc-1)
  print 'Processor 0 received message "%s" from processor %d' %(msg, numproc-

else:
  source = myid-1
  destination = (myid+1)%numproc

  msg = pypar.receive(source)
  print 'Processor %d received message "%s" from processor %d'\
        %(myid, msg, source)

  msg = msg + ' P' + str(myid) + '_OK'
  print 'Processor %d sending msg "%s" to %d' %(myid, msg, destination)
  pypar.send(msg, destination)
```

6.1 Output

The output of when executed on four processors as `prun -n 4 para_demo.py` is

```
I am proc 1 of 4 on node sc0
I am proc 2 of 4 on node sc0
I am proc 0 of 4 on node sc0
Processor 0 sending message "Anybody out there?" to processor 1
I am proc 3 of 4 on node sc0
Processor 1 received message "Anybody out there?" from processor 0
Processor 1 sending msg "Anybody out there? P1_OK" to 2
Processor 0 received message "Anybody out there? P1_OK P2_OK P3_OK" from processor 3
Processor 2 received message "Anybody out there? P1_OK" from processor 1
Processor 2 sending msg "Anybody out there? P1_OK P2_OK" to 3
Processor 3 received message "Anybody out there? P1_OK P2_OK" from processor 2
Processor 3 sending msg "Anybody out there? P1_OK P2_OK P3_OK" to 0
```