1 Introduction to Python

1.1 General Information

Quick Overview

This chapter is not a comprehensive manual of Python. Its sole aim is to provide sufficient information to give you a good start if you are unfamiliar with Python. If you know another computer language, and presumably you do, it is not difficult to pick up the rest as you go.

Python is an object-oriented language that was developed in late 1980s as a scripting language (the name is derived from the British television show Monty Python's Flying Circus). Although Python is not as well known in engineering circles as some other languages, it has a considerable following in the programming community—in fact, Python is considerably more widespread than Fortran. Python may be viewed as an emerging language, since it is still being developed and refined. In the current state, it is an excellent language for developing engineering applications—it possesses a simple elegance that other programming languages cannot match.

Python programs are not compiled into machine code, but are run by an *interpreter*[1]. The great advantage of an interpreted language is that programs can be tested and debugged quickly, allowing the user to concentrate more on the principles behind the program and less on programming itself. Since there is no need to compile, link and execute after each correction, Python programs can be developed in a much shorter time than equivalent Fortran or C programs. On the negative side, interpreted programs do not produce stand-alone applications. Thus a Python program can be run only on computers that have the Python interpreter installed.

[1] The Python interpreter also compiles *byte code*, which helps to speed up execution somewhat.

Python has other advantages over mainstream languages that are important in a learning environment:

- Python is open-source software, which means that it is *free*; it is included in most Linux distributions.
- Python is available for all major operating systems (Linux, Unix, Windows, Mac OS etc.). A program written on one system runs without modification on all systems.
- Python is easier to learn and produces more readable code than other languages.
- Python and its extensions are easy to install.

Development of Python was clearly influenced by Java and C++, but there is also a remarkable similarity to MATLAB® (another interpreted language, very popular in scientific computing). Python implements the usual concepts of object-oriented languages such as classes, methods, inheritance etc. We will forego these concepts and use Python strictly as a procedural language.

To get an idea of the similarities between MATLAB and Python, let us look at the codes written in the two languages for solution of simultaneous equations $\mathbf{Ax} = \mathbf{b}$ by Gauss elimination. Here is the function written in MATLAB:

```
function [x,det] = gaussElimin(a,b)
n = length(b);
for k = 1:n-1
    for i = k+1:n
        if a(i,k) ~= 0
            lam = a(i,k)/a(k,k);
            a(i,k+1:n) = a(i,k+1:n) - lam*a(k,k+1:n);
            b(i)= b(i) - lam*b(k);
        end
    end
end
det = prod(diag(a));
for k = n:-1:1
    b(k) = (b(k) - a(k,k+1:n)*b(k+1:n))/a(k,k);
end
x = b;
```

The equivalent Python function is:

```
from numarray import dot
def gaussElimin(a,b):
    n = len(b)
```

```
for k in range(0,n-1):
    for i in range(k+1,n):
       if a[i,k] != 0.0:
           lam = a [i,k]/a[k,k]
           a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
           b[i] = b[i] - lam*b[k]
for k in range(n-1,-1,-1):
    b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
return b
```

The command `from numarray import dot` instructs the interpreter to load the function `dot` (which computes the dot product of two vectors) from the module `numarray`. The colon (`:`) operator, known as the *slicing operator* in Python, works the same way it does in MATLAB and Fortran90—it defines a section of an array.

The statement `for k = 1:n-1` in MATLAB creates a loop that is executed with $k = 1, 2, \ldots, n-1$. The same loop appears in Python as `for k in range(n-1)`. Here the function `range(n-1)` creates the list $[0, 1, \ldots, n-2]$; k then loops over the elements of the list. The differences in the ranges of k reflect the native offsets used for arrays. In Python all sequences have *zero offset*, meaning that the index of the first element of the sequence is always 0. In contrast, the native offset in MATLAB is 1.

Also note that Python has no `end` statements to terminate blocks of code (loops, conditionals, subroutines etc.). The body of a block is defined by its *indentation*; hence indentation is an integral part of Python syntax.

Like MATLAB, Python is *case sensitive*. Thus the names n and N would represent different objects.

Obtaining Python

Python interpreter can be downloaded from the Python Language Website `www.python.org`. It normally comes with a nice code editor called *Idle* that allows you to run programs directly from the editor. For scientific programming we also need the *Numarray* module which contains various tools for array operations. It is obtainable from the Numarray Home Page `http://www.stsci.edu/resources/software_hardware/numarray`. Both sites also provide documentation for downloading. If you use Linux or Mac OS, it is very likely that Python is already installed on your machine (but you must still download Numarray).

You should acquire other printed material to supplement the on-line documentation. A commendable teaching guide is *Python* by Chris Fehly, Peachpit Press, CA (2002). As a reference, *Python Essential Reference* by David M. Beazley, New Riders

Publishing (2001) is recommended. By the time you read this, newer editions may be available.

1.2 Core Python

Variables

In most computer languages the name of a variable represents a value of a given type stored in a fixed memory location. The value may be changed, but not the type. This it not so in Python, where variables are *typed dynamically*. The following interactive session with the Python interpreter illustrates this (>>> is the Python prompt):

```
>>> b = 2         # b is integer type
>>> print b
2
>>> b = b * 2.0  # Now b is float type
>>> print b
4.0
```

The assignment `b = 2` creates an association between the name `b` and the *integer* value 2. The next statement evaluates the expression `b * 2.0` and associates the result with `b`; the original association with the integer 2 is destroyed. Now `b` refers to the *floating* point value 4.0.

The pound sign (#) denotes the beginning of a *comment*—all characters between # and the end of the line are ignored by the interpreter.

Strings

A string is a sequence of characters enclosed in single or double quotes. Strings are *concatenated* with the plus (+) operator, whereas *slicing* (`:`) is used to extract a portion of the string. Here is an example:

```
>>> string1 = 'Press return to exit'
>>> string2 = 'the program'
>>> print string1 + ' ' + string2  # Concatenation
Press return to exit the program
>>> print string1[0:12]            # Slicing
Press return
```

A string is an *immutable* object—its individual characters cannot be modified with an assignment statement and it has a fixed length. An attempt to violate immutability will result in `TypeError`, as shown below.

```
>>> s = 'Press return to exit'
>>> s[0] = 'p'
Traceback (most recent call last):
  File ''<pyshell#1>'', line 1, in ?
    s[0] = 'p'
TypeError: object doesn't support item assignment
```

Tuples

A *tuple* is a sequence of *arbitrary objects* separated by commas and enclosed in parentheses. If the tuple contains a single object, the parentheses may be omitted. Tuples support the same operations as strings; they are also immutable. Here is an example where the tuple `rec` contains another tuple `(6,23,68)`:

```
>>> rec = ('Smith','John',(6,23,68))    # This is a tuple
>>> lastName,firstName,birthdate = rec  # Unpacking the tuple
>>> print firstName
John
>>> birthYear = birthdate[2]
>>> print birthYear
68
>>> name = rec[1] + ' ' + rec[0]
>>> print name
John Smith
>>> print rec[0:2]
('Smith', 'John')
```

Lists

A list is similar to a tuple, but it is *mutable*, so that its elements and length can be changed. A list is identified by enclosing it in brackets. Here is a sampling of operations that can be performed on lists:

```
>>> a = [1.0, 2.0, 3.0]      # Create a list
>>> a.append(4.0)            # Append 4.0 to list
>>> print a
[1.0, 2.0, 3.0, 4.0]
```

```
>>> a.insert(0,0.0)          # Insert 0.0 in position 0
>>> print a
[0.0, 1.0, 2.0, 3.0, 4.0]
>>> print len(a)             # Determine length of list
5
>>> a[2:4] = [1.0, 1.0] # Modify selected elements
>>> print a
[0.0, 1.0, 1.0, 1.0, 1.0, 4.0]
```

If *a* is a mutable object, such as a list, the assignment statement `b = a` does not result in a new object *b*, but simply creates a new reference to *a*. Thus any changes made to *b* will be reflected in *a*. To create an independent copy of a list *a*, use the statement `c = a[:]`, as illustrated below.

```
>>> a = [1.0, 2.0, 3.0]
>>> b = a                # 'b' is an alias of 'a'
>>> b[0] = 5.0           # Change 'b'
>>> print a
[5.0, 2.0, 3.0]          # The change is reflected in 'a'
>>> c = a[:]             # 'c' is an independent copy of 'a'
>>> c[0] = 1.0           # Change 'c'
>>> print a
[5.0, 2.0, 3.0]          # 'a' is not affected by the change
```

Matrices can represented as nested lists with each row being an element of the list. Here is a 3×3 matrix *a* in the form of a list:

```
>>> a = [[1, 2, 3], \
         [4, 5, 6], \
         [7, 8, 9]]
>>> print a[1]           # Print second row (element 1)
[4, 5, 6]
>>> print a[1][2]        # Print third element of second row
6
```

The backslash (\) is Python's *continuation character*. Recall that Python sequences have zero offset, so that `a[0]` represents the first row, `a[1]` the second row, etc. With very few exceptions we do not use lists for numerical arrays. It is much more convenient

to employ *array objects* provided by the `numarray` module, (an extension of Python language). Array objects will be discussed later.

Arithmetic Operators

Python supports the usual arithmetic operators:

+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modular division

Some of these operators are also defined for strings and sequences as illustrated below.

```
>>> s = 'Hello '
>>> t = 'to you'
>>> a = [1, 2, 3]
>>> print 3*s                # Repetition
Hello Hello Hello
>>> print 3*a                # Repetition
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> print a + [4, 5]         # Append elements
[1, 2, 3, 4, 5]
>>> print s + t              # Concatenation
Hello to you
>>> print 3 + s              # This addition makes no sense
Traceback (most recent call last):
  File ''<pyshell#9>'', line 1, in ?
    print n + s
TypeError: unsupported operand types for +: 'int' and 'str'
```

Python 2.0 and later versions also have *augmented assignment operators*, such as $a \mathrel{+}= b$, that are familiar to the users of C. The augmented operators and the equivalent arithmetic expressions are shown in the following table.

`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a*b`
`a /= b`	`a = a/b`
`a **= b`	`a = a**b`
`a %= b`	`a = a%b`

Comparison Operators

The comparison (relational) operators return 1 for true and 0 for false. These operators are

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Numbers of different type (integer, floating point etc.) are converted to a common type before the comparison is made. Otherwise, objects of different type are considered to be unequal. Here are a few examples:

```
>>> a = 2           # Integer
>>> b = 1.99        # Floating point
>>> c = '2'         # String
>>> print a > b
1
>>> print a == c
0
>>> print (a > b) and (a != c)
1
>>> print (a > b) or (a == b)
1
```

Conditionals

The `if` construct

```
if condition:
    block
```

executes a block of statements (which must be indented) if the condition returns true. If the condition returns false, the block skipped. The `if` conditional can be followed by any number of `elif` (short for "else if") constructs

```
elif condition:
    block
```

which work in the same manner. The `else` clause

```
else:
    block
```

can be used to define the block of statements which are to be executed if none of the `if-elif` clauses are true. The function `sign_of_a` below illustrates the use of the conditionals.

```
def sign_of_a(a):
    if a < 0.0:
        sign = 'negative'
    elif a > 0.0:
        sign = 'positive'
    else:
        sign = 'zero'
    return sign

a = 1.5
print 'a is ' + sign_of_a(a)
```

Running the program results in the output

```
a is positive
```

Loops

The `while` construct

```
while condition:
    block
```

executes a block of (indented) statements if the condition is true. After execution of the block, the condition is evaluated again. If it is still true, the block is executed again. This process is continued until the condition becomes false. The `else` clause

```
else:
    block
```

can be used to define the block of statements which are to be executed if condition is false. Here is an example that creates the list $[1, 1/2, 1/3, \ldots]$:

```
nMax = 5
n = 1
a = []                # Create empty list
while n < nMax:
    a.append(1.0/n)   # Append element to list
    n = n + 1
print a
```

The output of the program is

```
[1.0, 0.5, 0.33333333333333331, 0.25]
```

We met the `for` statement before in Art. 1.1. This statement requires a target and a sequence (usually a list) over which the target loops. The form of the construct is

```
for target in sequence:
    block
```

You may add an `else` clause which is executed after the `for` loop has finished. The previous program could be written with the `for` construct as