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 we assume that you do, it is not difficult to pick up the rest as you go.

Python is an object-oriented language that was developed in the late 1980s as a scripting language (the name is derived from the British television series, *Monty Python's Flying Circus*). Although Python is not as well known in engineering circles as are some other languages, it has a considerable following in the programming community. Python may be viewed as an emerging language, because it is still being developed and refined. In its current state, it is an excellent language for developing engineering applications.

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 the programming itself. Because there is no need to compile, link, and execute after each correction, Python programs can be developed in 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.

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

- Python is an 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, and so on). A program written on one system runs without modification on all systems.

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

- Python is easier to learn and produces more readable code than most languages.
- Python and its extensions are easy to install.

Development of Python has been 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 do not use object-oriented programming in this text. The only object that we need is the N-dimensional *array* available in the module `numpy` (this module is discussed later in this chapter).

To get an idea of the similarities and differences 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. Do not worry about the algorithm itself (it is explained later in the text), but concentrate on the semantics. Here is the function written in MATLAB:

```
function x = 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
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 numpy 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 numpy import dot` instructs the interpreter to load the function `dot` (which computes the dot product of two vectors) from the module `numpy`. The colon `(:)` operator, known as the *slicing operator* in Python, works the same way as it does in MATLAB and Fortran90—it defines a slice 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 sequence $[0, 1, \ldots, n-2]$; k then loops over the elements of the sequence. 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, subroutines, and so on). 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

The *Python interpreter* can be downloaded from

```
http://www.python.org/getit
```

It normally comes with a nice code editor called *Idle* that allows you to run programs directly from the editor. If you use Linux, it is very likely that Python is already installed on your machine. The download includes two extension modules that we use in our programs: the `numpy` module that contains various tools for array operations, and the `matplotlib` graphics module utilized in plotting.

The Python language is well documented in numerous publications. A commendable teaching guide is *Python* by Chris Fehly (Peachpit Press, CA, 2nd ed.). As a reference, *Python Essential Reference* by David M. Beazley (Addison-Wesley, 4th ed.) is highly recommended. Printed documentation of the extension modules is scant. However, tutorials and examples can be found on various websites. Our favorite reference for `numpy` is

```
http://www.scipy.org/Numpy_Example_List
```

For `matplotlib` we rely on

```
http://matplotlib.sourceforge.net/contents.html
```

If you intend to become a serious Python programmer, you may want to acquire *A Primer on Scientific Programming with Python* by Hans P. Langtangen (Springer-Verlag, 2009).

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 is not so in Python, where variables are *typed dynamically*. The following interactive session with the Python interpreter illustrates this feature (`>>>` 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 can be split into its component parts using the `split` command. The components appear as elements in a list. For example,

```
>>> s = '3 9 81'
>>> print(s.split())    # Delimiter is white space
['3', '9', '81']
```

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 follows:

```
>>> 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, a final comma is required; for example, `x = (2,)`. 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, 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 shown in the following example:

```
>>> 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 be 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 `numpy` module. Array objects are 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 follows:

```
>>> 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#13>", line 1, in <module>
    print(3 + s)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python also has *augmented assignment operators*, such as $a+=b$, that are familiar to the users of C. The augmented operators and the equivalent arithmetic expressions are shown in 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 `True` or `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, and so on) 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)
True
```

```
>>> print(a == c)
False
>>> print((a > b) and (a != c))
True
>>> print((a > b) or (a == b))
True
```

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 is skipped. The `if` conditional can be followed by any number of `elif` (short for "else if") constructs

```
elif condition:
    block
```

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

```
else:
    block
```

can be used to define the block of statements that are to be executed if none of the `if-elif` clauses are true. The function `sign_of_a` 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 that are to be executed if the 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 in Section 1.1. This statement requires a target and a sequence over which the target loops. The form of the construct is

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

You may add an `else` clause that is executed after the `for` loop has finished.

The previous program could be written with the `for` construct as

```
nMax = 5
a = []
for n in range(1,nMax):
    a.append(1.0/n)
print(a)
```

Here n is the target, and the *range object* $[1, 2, \ldots, nMax - 1]$ (created by calling the `range` function) is the sequence.

Any loop can be terminated by the

```
break
```

statement. If there is an `else` cause associated with the loop, it is not executed. The following program, which searches for a name in a list, illustrates the use of `break` and `else` in conjunction with a `for` loop:

```
list = ['Jack', 'Jill', 'Tim', 'Dave']
name = eval(input('Type a name: '))  # Python input prompt
for i in range(len(list)):
    if list[i] == name:
        print(name,'is number',i + 1,'on the list')
        break
else:
    print(name,'is not on the list')
```

Here are the results of two searches:

```
Type a name: 'Tim'
Tim is number 3 on the list

Type a name: 'June'
June is not on the list
```

The

`continue`

statement allows us to skip a portion of an iterative loop. If the interpreter encounters the `continue` statement, it immediately returns to the beginning of the loop without executing the statements that follow `continue`. The following example compiles a list of all numbers between 1 and 99 that are divisible by 7.

```
x = []                    # Create an empty list
for i in range(1,100):
  if i%7 != 0: continue   # If not divisible by 7, skip rest of loop
  x.append(i)             # Append i to the list
print(x)
```

The printout from the program is

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
```

Type Conversion

If an arithmetic operation involves numbers of mixed types, the numbers are automatically converted to a common type before the operation is carried out.