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 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 used by more programmers than Fortran. Python may be viewed as an emerging language, because it is still being developed and refined. In the current state, it is an excellent language for developing engineering applications – Python's facilities for numerical computation are as good as those of Fortran or MATLAB®.

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. Because 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.

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.

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

- 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 do most 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, and inheritance. We will not use object-oriented programming in this text. The only object that we need is the N-dimensional *array* available in the NumPy module (the NumPy module is discussed later in this chapter).

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 = 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 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 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, 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

The 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 *NumPy* module, which contains various tools for array operations. It is obtainable from the NumPy home page `http://numpy.scipy.org/`. Both sites also provide documentation for downloading. If you use Linux, it is very likely that Python is already installed on your machine (but you must still download NumPy).

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. A useful guide to NumPy is found at `http://www.scipy.org/Numpy_Example_List`.

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 here:

```
>>> 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 here:

```
>>> 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, and so forth. 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 illustrated here:

```
>>> 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 $a+=b$, 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 is 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 that are to be executed if none of the `if-elif` clauses is 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 before in Section 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 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 list $[1, 2, \ldots, nMax - 1]$, 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(raw_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 the statements in an iterative loop. If the interpreter encounters the `continue` statement, it immediately returns to the beginning of the loop without executing the statements below `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. Type conversions can also be achieved by the following functions:

`int(a)`	Converts a to integer
`long(a)`	Converts a to long integer
`float(a)`	Converts a to floating point
`complex(a)`	Converts to complex $a + 0j$
`complex(a,b)`	Converts to complex $a + bj$

The foregoing functions also work for converting strings to numbers as long as the literal in the string represents a valid number. Conversion from a float to an integer is carried out by truncation, not by rounding off. Here are a few examples:

```
>>> a = 5
>>> b = -3.6
```

```
>>> d = '4.0'
>>> print a + b
1.4
>>> print int(b)
-3
>>> print complex(a,b)
(5-3.6j)
>>> print float(d)
4.0
>>> print int(d)  # This fails: d is not Int type
Traceback (most recent call last):
  File ''<pyshell#7>'', line 1, in ?
    print int(d)
ValueError: invalid literal for int(): 4.0
```

Mathematical Functions

Core Python supports only a few mathematical functions:

`abs(a)`	Absolute value of `a`
`max(`*sequence*`)`	Largest element of *sequence*
`min(`*sequence*`)`	Smallest element of *sequence*
`round(a,n)`	Round `a` to `n` decimal places
`cmp(a,b)`	Returns $\begin{cases} -1 \text{ if } a < b \\ 0 \text{ if } a = b \\ 1 \text{ if } a > b \end{cases}$

The majority of mathematical functions are available in the `math` module.

Reading Input

The intrinsic function for accepting user input is

```
raw_input(prompt)
```

It displays the prompt and then reads a line of input that is converted to a *string*. To convert the string into a numerical value, use the function

```
eval(string)
```

The following program illustrates the use of these functions:

```
a = raw_input('Input a: ')
print a, type(a)           # Print a and its type
b = eval(a)
print b,type(b)            # Print b and its type
```

The function `type(a)` returns the type of the object *a*; it is a very useful tool in debugging. The program was run twice with the following results:

```
Input a: 10.0
10.0 <type 'str'>
10.0 <type 'float'>

Input a: 11**2
11**2 <type 'str'>
121 <type 'int'>
```

A convenient way to input a number and assign it to the variable *a* is

```
a = eval(raw_input(prompt))
```

Printing Output

Output can be displayed with the print statement:

```
printobject1, object2, ...
```

which converts *object1*, *object2*, and so on to strings and prints them on the same line, separated by spaces. The *newline* character `'\n'` can be used to force a new line. For example,

```
>>> a = 1234.56789
>>> b = [2, 4, 6, 8]
>>> print a,b
1234.56789 [2, 4, 6, 8]
>>> print 'a =',a, '\nb =',b
a = 1234.56789
b = [2, 4, 6, 8]
```

The *modulo operator* (%) can be used to format a tuple. The form of the conversion statement is

```
'%format1 %format2 ...' % tuple
```

where *format1*, *format2* ··· are the format specifications for each object in the tuple. Typically used format specifications are:

*w*d	Integer
w.*d*f	Floating point notation
w.*d*e	Exponential notation

where *w* is the width of the field and *d* is the number of digits after the decimal point. The output is right-justified in the specified field and padded with blank spaces

(there are provisions for changing the justification and padding). Here are a couple of examples:

```
>>> a = 1234.56789
>>> n = 9876
>>> print '%7.2f' % a
1234.57
>>> print 'n = %6d' % n  # Pad with spaces
n =   9876
>>> print 'n = %06d' % n # Pad with zeroes
n = 009876
>>> print '%12.4e %6d' % (a,n)
 1.2346e+003   9876
```

Opening and Closing a File

Before a data file can be accessed, you must create a *file object* with the command

file_object = `open`(*filename*, *action*)

where *filename* is a string that specifies the file to be opened (including its path if necessary) and *action* is one of the following strings:

`'r'`	Read from an existing file.
`'w'`	Write to a file. If *filename* does not exist, it is created.
`'a'`	Append to the end of the file.
`'r+'`	Read to and write from an existing file.
`'w+'`	Same as `'r+'`, but *filename* is created if it does not exist.
`'a+'`	Same as `'w+'`, but data is appended to the end of the file.

It is good programming practice to close a file when access to it is no longer required. This can be done with the method

file_object`.close()`

Reading Data from a File

There are three methods for reading data from a file. The method

file_object`.read(`*n*`)`

reads *n* characters and returns them as a string. If *n* is omitted, all the characters in the file are read.

If only the current line is to be read, use

file_object`.readline(`*n*`)`

which reads *n* characters from the line. The characters are returned in a string that terminates in the newline character `\n`. Omission of *n* causes the entire line to be read.

All the lines in a file can be read using

file_object`.readlines()`

This returns a list of strings, each string being a line from the file ending with the newline character.

Writing Data to a File

The method

file_object`.write()`

writes a string to a file, whereas

file_object`.writelines()`

is used to write a list of strings. Neither method appends a newline character to the end of a line.

The `print` statement can also be used to write to a file by redirecting the output to a file object:

`print >>` *file_object*, *object1*, *object2*, ...

Apart from the redirection, this statement works just like the regular `print` command.

Error Control

When an error occurs during execution of a program, an exception is raised and the program stops. Exceptions can be caught with `try` and `except` statements:

```
try:
    do something
except error:
    do something else
```

where *error* is the name of a built-in Python exception. If the exception *error* is not raised, the `try` block is executed; otherwise, the execution passes to the `except` block. All exceptions can be caught by omitting *error* from the `except` statement.

Here is a statement that raises the exception `ZeroDivisionError`:

```
>>> c = 12.0/0.0
Traceback (most recent call last):
  File ''<pyshell#0>'', line 1, in ?
    c = 12.0/0.0
ZeroDivisionError: float division
```

This error can be caught by

```
try:
    c = 12.0/0.0
except ZeroDivisionError:
    print 'Division by zero'
```

1.3 Functions and Modules

Functions

The structure of a Python function is

```
def func_name(param1, param2,...):
    statements
    return return_values
```

where *param1, param2,...* are the parameters. A parameter can be any Python object, including a function. Parameters may be given default values, in which case the parameter in the function call is optional. If the `return` statement or *return_values* are omitted, the function returns the null object.

The following example computes the first two derivatives of $\arctan(x)$ by finite differences:

```
from math import atan
def  finite_diff(f,x,h=0.0001):   # h has a default value
    df =(f(x+h) - f(x-h))/(2.0*h)
    ddf =(f(x+h) - 2.0*f(x) + f(x-h))/h**2
    return df,ddf

x = 0.5
df,ddf = finite_diff(atan,x)   # Uses default value of h
print 'First derivative  =',df
print 'Second derivative =',ddf
```

Note that `atan` is passed to `finite_diff` as a parameter. The output from the program is

```
First derivative  = 0.799999999573
Second derivative = -0.639999991892
```

The number of input parameters in a function definition may be left arbitrary. For example, in the function definition

```
def func(x1,x2,*x3)
```

`x1` and `x2` are the usual parameters, also called *positional parameters*, whereas `x3` is a tuple of arbitrary length containing the *excess parameters*. Calling this function with

```
func(a,b,c,d,e)
```

results in the following correspondence between the parameters:

`a` ⟷ `x1`, `b` ⟷ `x2`, `(c,d,e)` ⟷ `x3`

The positional parameters must always be listed before the excess parameters.

If a mutable object, such as a list, is passed to a function where it is modified, the changes will also appear in the calling program. Here is an example:

```
def squares(a):
    for i in range(len(a)):
        a[i] = a[i]**2

a = [1, 2, 3, 4]
squares(a)
print a
```

The output is

```
[1, 4, 9, 16]
```

Lambda Statement

If the function has the form of an expression, it can be defined with the lambda statement

func_name = `lambda` *param1, param2,. . .* : *expression*

Multiple statements are not allowed.

Here is an example:

```
>>> c = lambda x,y : x**2 + y**2
>>> print c(3,4)
25
```

Modules

It is sound practice to store useful functions in modules. A module is simply a file where the functions reside; the name of the module is the name of the file. A module can be loaded into a program by the statement

`from` *module_name* `import *`

Python comes with a large number of modules containing functions and methods for various tasks. Some of the modules are described briefly in the following section.

Additional modules, including graphics packages, are available for downloading on the Web.

1.4 Mathematics Modules

`math` Module

Most mathematical functions are not built into core Python, but are available by loading the `math` module. There are three ways of accessing the functions in a module. The statement

```
from math import *
```

loads *all* the function definitions in the `math` module into the current function or module. The use of this method is discouraged because it not only is wasteful, but can also lead to conflicts with definitions loaded from other modules.

You can load selected definitions by

```
from math import func1, func2, ...
```

as illustrated here:

```
>>> from math import log,sin
>>> print log(sin(0.5))
-0.735166686385
```

The third method, which is used by the majority of programmers, is to make the module available by

```
import math
```

The module can then be accessed by using the module name as a prefix:

```
>>> import math
>>> print math.log(math.sin(0.5))
-0.735166686385
```

The contents of a module can be printed by calling `dir(`*module*`)`. Here is how to obtain a list of the functions in the `math` module:

```
>>> import math
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan',
 'atan2', 'ceil', 'cos', 'cosh', 'e', 'exp', 'fabs',
 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
 'log10', 'modf', 'pi', 'pow', 'sin', 'sinh', 'sqrt',
 'tan', 'tanh']
```

Most of these functions are familiar to programmers. Note that the module includes two constants: π and e.

`cmath` Module

The `cmath` module provides many of the functions found in the `math` module, but these accept complex numbers. The functions in the module are:

```
['__doc__', '__name__', 'acos', 'acosh', 'asin', 'asinh',
 'atan', 'atanh', 'cos', 'cosh', 'e', 'exp', 'log',
 'log10', 'pi', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

Here are examples of complex arithmetic:

```
>>> from cmath import sin
>>> x = 3.0 -4.5j
>>> y = 1.2 + 0.8j
>>> z = 0.8
>>> print x/y
(-2.56205313375e-016-3.75j)
>>> print sin(x)
(6.35239299817+44.5526433649j)
>>> print sin(z)
(0.7173560909+0j)
```

1.5 `numpy` Module

General Information

The NumPy module[2] is not a part of the standard Python release. As pointed out before, it must be obtained separately and installed (the installation is very easy). The module introduces *array objects* that are similar to lists, but can be manipulated by numerous functions contained in the module. The size of an array is immutable, and no empty elements are allowed.

The complete set of functions in `numpy` is far too long to be printed in its entirety. The following list is limited to the most commonly used functions:

```
['complex', 'float', 'abs', 'append', arccos',
'arccosh', 'arcsin', 'arcsinh', 'arctan', 'arctan2',
'arctanh', 'argmax', 'argmin', 'cos', 'cosh', 'diag',
'diagonal', 'dot', 'e', 'exp', 'floor', 'identity',
'inner, 'inv', 'log', 'log10', 'max', 'min',
'ones','outer', 'pi', 'prod' 'sin', 'sinh', 'size',
'solve','sqrt', 'sum', 'tan', 'tanh', 'trace',
'transpose', 'zeros', 'vectorize']
```

[2] *NumPy* is the successor of older Python modules called *Numeric* and *NumArray*. Their interfaces and capabilities are very similar. Although *Numeric* and *NumArray* are still available, they are no longer supported.

Creating an Array

Arrays can be created in several ways. One of them is to use the `array` function to turn a list into an array:

`array(`*list*`,dtype =` *type_specification*`)`

Here are two examples of creating a 2×2 array with floating-point elements:

```
>>> from numpy import array,float
>>> a = array([[2.0, -1.0],[-1.0, 3.0]])
>>> print a
[[ 2. -1.]
 [-1.  3.]]
>>> b = array([[2, -1],[-1, 3]],dtype = float)
>>> print b
[[ 2. -1.]
 [-1.  3.]]
```

Other available functions are

`zeros((`*dim1*`,`*dim2*`),dtype =` *type_specification*`)`

which creates a *dim1* $\times$ *dim2* array and fills it with zeroes, and

`ones((`*dim1*`,`*dim2*`),dtype =` *type_specification*`)`

which fills the array with ones. The default type in both cases is `float`.

Finally, there is the function

`arange(`*from*`,`*to*`,`*increment*`)`

which works just like the `range` function, but returns an array rather than a list. Here are examples of creating arrays:

```
>>> from numpy import *
>>> print arange(2,10,2)
[2 4 6 8]
>>> print arange(2.0,10.0,2.0)
[ 2.  4.  6.  8.]
>>> print zeros(3)
[ 0.  0.  0.]
>>> print zeros((3),dtype=int)
[0 0 0]
>>> print ones((2,2))
[[ 1.  1.]
 [ 1.  1.]]
```

Accessing and Changing Array Elements

If *a* is a rank-2 array, then `a[i,j]` accesses the element in row *i* and column *j*, whereas `a[i]` refers to row *i*. The elements of an array can be changed by assignment:

```
>>> from numpy import *
>>> a = zeros((3,3),dtype=int)
>>> print a
[[0 0 0]
 [0 0 0]
 [0 0 0]]
>>> a[0] = [2,3,2]     # Change a row
>>> a[1,1] = 5         # Change an element
>>> a[2,0:2] = [8,-3]  # Change part of a row
>>> print a
[[ 2  3  2]
 [ 0  5  0]
 [ 8 -3  0]]
```

Operations on Arrays

Arithmetic operators work differently on arrays than they do on tuples and lists – the operation is *broadcast* to all the elements of the array; that is, the operation is applied to each element in the array. Here are examples:

```
>>> from numpy import array
>>> a = array([0.0, 4.0, 9.0, 16.0])
>>> print a/16.0
[ 0.      0.25    0.5625  1.    ]
>>> print a - 4.0
[ -4.   0.   5.  12.]
```

The mathematical functions available in NumPy are also broadcast:

```
>>> from numpy import array,sqrt,sin
>>> a = array([1.0, 4.0, 9.0, 16.0])
>>> print sqrt(a)
[ 1.  2.  3.  4.]
>>> print sin(a)
[ 0.84147098 -0.7568025   0.41211849 -0.28790332]
```

Functions imported from the `math` module will work on the individual elements, of course, but not on the array itself. Here is an example:

```
>>> from numpy import array
>>> from math import sqrt
```

```
>>> a = array([1.0, 4.0, 9.0, 16.0])
>>> print sqrt(a[1])
2.0
>>> print sqrt(a)
Traceback (most recent call last):
   ⋮
TypeError: only length-1 arrays can be converted to Python scalars
```

Array Functions

There are numerous functions in NumPy that perform array operations and other useful tasks. Here are a few examples:

```
>>> from numpy import *
>>> A = array([[4,-2,1],[-2,4,-2],[1,-2,3]],dtype=float)
>>> b = array([1,4,3],dtype=float)
>>> print diagonal(A)       # Principal diagonal
[ 4.  4.  3.]
>>> print diagonal(A,1)     # First subdiagonal
[-2. -2.]
>>> print trace(A)          # Sum of diagonal elements
11.0
>>> print argmax(b)         # Index of largest element
1
>>> print argmin(A,axis=0)  # Indecies of smallest col. elements
[1 0 1]
>>> print identity(3)       # Identity matrix
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

There are three functions in NumPy that compute array products. They are illustrated by the program listed below For more details, see Appendix A2.

```
from numpy import *
x = array([7,3])
y = array([2,1])
A = array([[1,2],[3,2]])
B = array([[1,1],[2,2]])

# Dot product
print "dot(x,y) =\n",dot(x,y)      # {x}.{y}
print "dot(A,x) =\n",dot(A,x)      # [A]{x}
print "dot(A,B) =\n",dot(A,B)      # [A][B]
```

```
# Inner product
print "inner(x,y) =\n",inner(x,y)  # {x}.{y}
print "inner(A,x) =\n",inner(A,x)  # [A]{x}
print "inner(A,B) =\n",inner(A,B)  # [A][B_transpose]

# Outer product
print "outer(x,y) =\n",outer(x,y)
print "outer(A,x) =\n",outer(A,x)
print "Outer(A,B) =\n",outer(A,B)
```

The output of the program is

```
dot(x,y) =
17
dot(A,x) =
[13 27]
dot(A,B) =
[[5 5]
 [7 7]]
inner(x,y) =
17
inner(A,x) =
[13 27]
inner(A,B) =
[[ 3  6]
 [ 5 10]]
outer(x,y) =
[[14  7]
 [ 6  3]]
outer(A,x) =
[[ 7  3]
 [14  6]
 [21  9]
 [14  6]]
Outer(A,B) =
[[1 1 2 2]
 [2 2 4 4]
 [3 3 6 6]
 [2 2 4 4]]
```

Linear Algebra Module

NumPy comes with a linear algebra module called *linalg* that contains routine tasks such as matrix inversion and solution of simultaneous equations. For example:

```
>>> from numpy import array
>>> from numpy.linalg import inv,solve
>>> A = array([[ 4.0, -2.0,  1.0], \
               [-2.0,  4.0, -2.0], \
               [ 1.0, -2.0,  3.0]])
>>> b = array([1.0, 4.0, 2.0])
>>> print inv(A)                          # Matrix inverse
[[ 0.33333333  0.16666667  0.        ]
 [ 0.16666667  0.45833333  0.25      ]
 [ 0.          0.25        0.5       ]]
>>> print solve(A,b)                      # Solve [A]{x} = {b}
[ 1. ,  2.5,  2. ]
```

Copying Arrays

We explained before that 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, called a *deep copy*. This also applies to arrays. To make an independent copy of an array a, use the `copy` method in the NumPy module:

```
b = a.copy()
```

Vectorizing Algorithms

Sometimes the broadcasting properties of the mathematical functions in the NumPy module can be utilized to replace loops in the code. This procedure is known as *vectorization*. Consider, for example, the expression

$$s = \sum_{i=0}^{100} \sqrt{\frac{i\pi}{100}} \sin\frac{i\pi}{100}$$

The direct approach is to evaluate the sum in a loop, resulting in the following "scalar" code:

```
from math import sqrt,sin,pi
x=0.0; sum = 0.0
for i in range(0,101):
    sum = sum + sqrt(x)*sin(x)
    x = x + 0.01*pi
print sum
```

The vectorized version of algorithm is

```
from numpy import sqrt,sin,arange
from math import pi
x = arrange(0.0,1.001*pi,0.01*pi)
print sum(sqrt(x)*sin(x))
```

Note that the first algorithm uses the scalar versions of `sqrt` and `sin` functions in the `math` module, whereas the second algorithm imports these functions from the `numpy`. The vectorized algorithm is faster, but uses more memory.

1.6 Scoping of Variables

Namespace is a dictionary that contains the names of the variables and their values. The namespaces are automatically created and updated as a program runs. There are three levels of namespaces in Python:

- Local namespace, which is created when a function is called. It contains the variables passed to the function as arguments and the variables created within the function. The namespace is deleted when the function terminates. If a variable is created inside a function, its scope is the function's local namespace. It is not visible outside the function.
- A global namespace is created when a module is loaded. Each module has its own namespace. Variables assigned in a global namespace are visible to any function within the module.
- Built-in namespace is created when the interpreter starts. It contains the functions that come with the Python interpreter. These functions can be accessed by any program unit.

When a name is encountered during execution of a function, the interpreter tries to resolve it by searching the following in the order shown: (1) local namespace, (2) global namespace, and (3) built-in namespace. If the name cannot be resolved, Python raises a `NameError` exception.

Because the variables residing in a global namespace are visible to functions within the module, it is not necessary to pass them to the functions as arguments (although is good programming practice to do so), as the following program illustrates:

```
def divide():
    c = a/b
    print 'a/b =',c

a = 100.0
b = 5.0
divide()

>>>
a/b = 20.0
```

Note that the variable `c` is created inside the function `divide` and is thus not accessible to statements outside the function. Hence an attempt to move the print statement out of the function fails:

```
def divide():
    c = a/b

a = 100.0
b = 5.0
divide()
print 'a/b =',c

>>>
Traceback (most recent call last):
  File ''C:\Python22\scope.py'', line 8, in ?
    print c
NameError: name 'c' is not defined
```

1.7 Writing and Running Programs

When the Python editor *Idle* is opened, the user is faced with the prompt >>>, indicating that the editor is in interactive mode. Any statement typed into the editor is immediately processed upon pressing the enter key. The interactive mode is a good way to learn the language by experimentation and to try out new programming ideas.

Opening a new window places Idle in the batch mode, which allows typing and saving of programs. One can also use a text editor to enter program lines, but Idle has Python-specific features, such as color coding of keywords and automatic indentation, that make work easier. Before a program can be run, it must be saved as a Python file with the `.py` extension, for example, `myprog.py`. The program can then be executed by typing `python myprog.py`; in Windows, double-clicking on the program icon will also work. But beware: the program window closes immediately after execution, before you get a chance to read the output. To prevent this from happening, conclude the program with the line

```
raw_input('press return')
```

Double-clicking the program icon also works in Unix and Linux if the first line of the program specifies the path to the Python interpreter (or a shell script that provides a link to Python). The path name must be preceded by the symbols `#!`. On my computer the path is `/usr/bin/python`, so that all my programs start with the line `#!/usr/bin/python`. On multiuser systems the path is usually `/usr/local/bin/python`.

When a module is loaded into a program for the first time with the `import` statement, it is compiled into bytecode and written in a file with the extension `.pyc`. The next time the program is run, the interpreter loads the bytecode rather than the original Python file. If in the meantime changes have been made to the module, the

module is automatically recompiled. A program can also be run from Idle using the *Run/Run Module* menu.

It is a good idea to document your modules by adding a *docstring* at the beginning of each module. The docstring, which is enclosed in triple quotes, should explain what the module does. Here is an example that documents the module `error` (we use this module in several of our programs):

```
## module error
''' err(string).
    Prints 'string' and terminates program.
'''
import sys
def err(string):
    print string
    raw_input('Press return to exit')
    sys.exit()
```

The docstring of a module can be printed with the statement

`print` *module_name*`.__doc__`

For example, the docstring of `error` is displayed by

```
>>> import error
>>> print error.__doc__
 err(string).
    Prints 'string' and terminates program.
```