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^R (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 $k = 1, 2, \dots, n - 1$. The same loop appears in Python as `for k in range(n-1)`. Here the function `range(n-1)` creates the sequence $[0, 1, \dots, n - 2]$; 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

$$\boxed{\texttt{while}\ \textit{condition}\texttt{:}\quad \textit{block}}$$

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

$$\boxed{\texttt{else:}\quad \textit{block}}$$

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, ...]:

```
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

$$\boxed{\texttt{for}\ \textit{target}\ \texttt{in}\ \textit{sequence}\texttt{:}\quad \textit{block}}$$

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

$$\boxed{\texttt{break}}$$

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.

Type conversions can also achieved by the following functions:

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

These 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 a string
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    print(int(d))
ValueError: invalid literal for int() with base 10: '4.0'
```

Mathematical Functions

Core Python supports only the following 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

$$\boxed{\texttt{input(}\textit{prompt}\texttt{)}}$$

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

$$\boxed{\texttt{eval(}\textit{string}\texttt{)}}$$

The following program illustrates the use of these functions:

```
a = 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 <class 'str'>
10.0 <class 'float'>

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

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

$$\boxed{\texttt{a = eval(input(}\textit{prompt}\texttt{))}}$$

Printing Output

Output can be displayed with the *print function*

$$\boxed{\texttt{print(}\textit{object1, object2, \ldots}\texttt{)}}$$

that 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 `print` function always appends the newline character to the end of a line. We can replace this character with something else by using the keyword argument `end`. For example,

$$\boxed{\texttt{print(}\textit{object1, object2, \ldots}\texttt{,end=' ')}}$$

replaces \n with a space.

Output can be formatted with the *format method*. The simplest form of the conversion statement is

$\boxed{\texttt{'}\{:\textit{fmt1}\}\{:\textit{fmt2}\}\texttt{...'.format(}\textit{arg1,arg2,}\texttt{...)}}$

where *fmt1, fmt2,...* are the format specifications for *arg1, arg2,...*, respectively. 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 several examples:

```
>>> a = 1234.56789
>>> n = 9876
>>> print('{:7.2f}'.format(a))
1234.57
>>> print('n = {:6d}'.format(n))   # Pad with spaces
n =  9876
>>> print('n = {:06d}'.format(n))  # Pad with zeros
n =009876
>>> print('{:12.4e} {:6d}'.format(a,n))
  1.2346e+03   9876
```

Opening and Closing a File

Before a data file on a storage device (e.g., a disk) can be accessed, you must create a *file object* with the command

$\boxed{\textit{file\_object} = \texttt{open(}\textit{filename, action}\texttt{)}}$

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

$\boxed{\textit{file\_object}\texttt{.close()}}$

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.

A convenient method of extracting all the lines one by one is to use the loop

```
for line in file_object:
    do something with line
```

As an example, let us assume that we have a file named `sunspots.txt` in the working directory. This file contains daily data of sunspot intensity, each line having the format (year/month/date/intensity), as follows:

```
1896 05 26 40.94
1896 05 27 40.58
1896 05 28 40.20
      etc.
```

Our task is to read the file and create a list `x` that contains only the intensity. Since each line in the file is a string, we first split the line into its pieces using the `split` command. This produces a list of strings, such as `['1896','05','26','40.94']`. Then we extract the intensity (element `[3]` of the list), evaluate it, and append the result to `x`. Here is the algorithm:

```
x = []
data = open('sunspots.txt','r')
for line in data:
   x.append(eval(line.split()[3]))
data.close()
```

Writing Data to a File

The method

$$\boxed{\textit{file\_object}\texttt{.write(}\textit{string}\texttt{)}}$$

writes a string to a file, whereas

$$\boxed{\textit{file\_object}\texttt{.writelines(}\textit{list\_of\_strings}\texttt{)}}$$

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

As an example, let us write a formatted table of k and k^2 from $k = 101$ to 110 to the file `testfile`. Here is the program that does the writing:

```
f = open('testfile','w')
for k in range(101,111):
    f.write('{:4d} {:6d}'.format(k,k**2))
    f.write('\n')
f.close()
```

The contents of `testfile` are

```
 101   10201
 102   10404
 103   10609
 104   10816
 105   11025
 106   11236
 107   11449
 108   11664
 109   11881
 110   12100
```

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

`print(`*object1, object2, . . .*`,file = `*file_object*`)`

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

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.

The following statement raises the exception `ZeroDivisionError`:

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

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 function computes the first two derivatives of $f(x)$ by finite differences:

```
def derivatives(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
```

Let us now use this function to determine the two derivatives of $\arctan(x)$ at $x = 0.5$:

```
from math import atan
df,ddf = derivatives(atan,0.5)    # Uses default value of h
print('First derivative  =',df)
print('Second derivative =',ddf)
```

Note that `atan` is passed to `derivatives` 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 following 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:

$$\texttt{a} \longleftrightarrow \texttt{x1}, \quad \texttt{b} \longleftrightarrow \texttt{x2}, \quad \texttt{(c,d,e)} \longleftrightarrow \texttt{x3}$$

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. An example follows:

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

a = [1, 2, 3, 4]
squares(a)
print(a)  # 'a' now contains 'a**2'
```

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

$$\boxed{\textit{func\_name} = \texttt{lambda}\ \textit{param1, param2,}\ldots :\ \textit{expression}}$$

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 next two sections. 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. For example, there are three different definitions of the *sine* function in the Python modules `math`, `cmath`, and `numpy`. If you have loaded two or more of these modules, it is unclear which definition will be used in the function call `sin(x)`(it is the definition in the module that was loaded last).

A safer but by no means foolproof method is to load selected definitions with the statement

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

as illustrated as follows:

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

Conflicts can be avoided altogether by first making the module accessible with the statement

```
import math
```

and then accessing the definitions in the module by using the module name as a prefix. Here is an example:

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

A module can also be made accessible under an *alias*. For example, the `math` module can be made available under the alias `m` with the command

```
import math as m
```

Now the prefix to be used is `m` rather than `math`:

```
>>> import math as m
>>> print(m.log(m.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', sign', 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 functions 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 earlier, it must be installed separately (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', 'vectorize','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:

$$\boxed{\texttt{array(}\mathit{list}\texttt{,}\ \mathit{type}\texttt{)}}$$

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

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

Other available functions are

$$\boxed{\texttt{zeros((}\mathit{dim1}\texttt{,}\ \mathit{dim2}\texttt{),}\ \mathit{type}\texttt{)}}$$

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

$$\boxed{\texttt{ones((}\mathit{dim1}\texttt{,}\ \mathit{dim2}\texttt{),}\ \mathit{type}\texttt{)}}$$

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

[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.

Finally, there is the function

$$\boxed{\texttt{arange(}\mathit{from}\texttt{,}\ \mathit{to}\texttt{,}\ \mathit{increment}\texttt{)}}$$

which works just like the `range` function, but returns an array rather than a sequence. 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),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 as follows:

```
>>> from numpy import *
>>> a = zeros((3,3),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, as follows:

```
>>> 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. An example follows:

```
>>> 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]],float)
>>> b = array([1,4,3],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))  # Indices 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 following program. 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

The `numpy` module 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 earlier 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 used 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; s = 0.0
for i in range(101):
    s = s + sqrt(x)*sin(x)
    x = x + 0.01*pi
print(s)
```

The vectorized version of the algorithm is

```
from numpy import sqrt,sin,arange
from math import pi
x = arange(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 `numpy`. The vectorized algorithm executes much faster, but uses more memory.

1.6 Plotting with `matplotlib.pyplot`

The module `matplotlib.pyplot` is a collection of 2D plotting functions that provide Python with MATLAB-style functionality. Not being a part of core Python, it requires separate installation. The following program, which plots sine and cosine functions, illustrates the application of the module to simple *xy* plots.

```
import matplotlib.pyplot as plt
from numpy import arange,sin,cos
x = arange(0.0,6.2,0.2)

plt.plot(x,sin(x),'o-',x,cos(x),'^-')    # Plot with specified
                                         # line and marker style
plt.xlabel('x')                          # Add label to x-axis
plt.legend(('sine','cosine'),loc = 0)    # Add legend in loc. 3
plt.grid(True)                           # Add coordinate grid
plt.savefig('testplot.png',format='png') # Save plot in png
                                         # format for future use
plt.show()                               # Show plot on screen
input("\nPress return to exit")
```

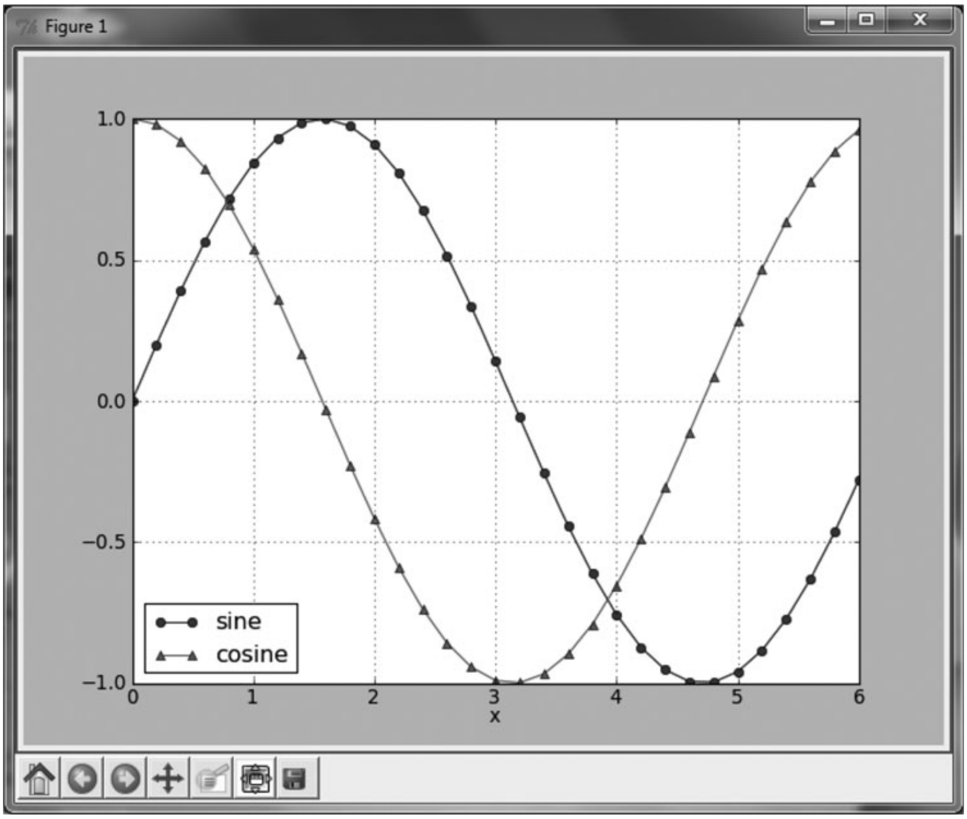
The line and marker styles are specified by the string characters shown in the following table (only some of the available characters are shown).

`'-'`	Solid line
`'--'`	Dashed line
`'-.'`	Dash-dot line
`':'`	Dotted line
`'o'`	Circle marker
`'^'`	Triangle marker
`'s'`	Square marker
`'h'`	Hexagon marker
`'x'`	x marker

Some of the location (`loc`) codes for placement of the legend are

0	"Best" location
1	Upper right
2	Upper left
3	Lower left
4	Lower right

Running the program produces the following screen:

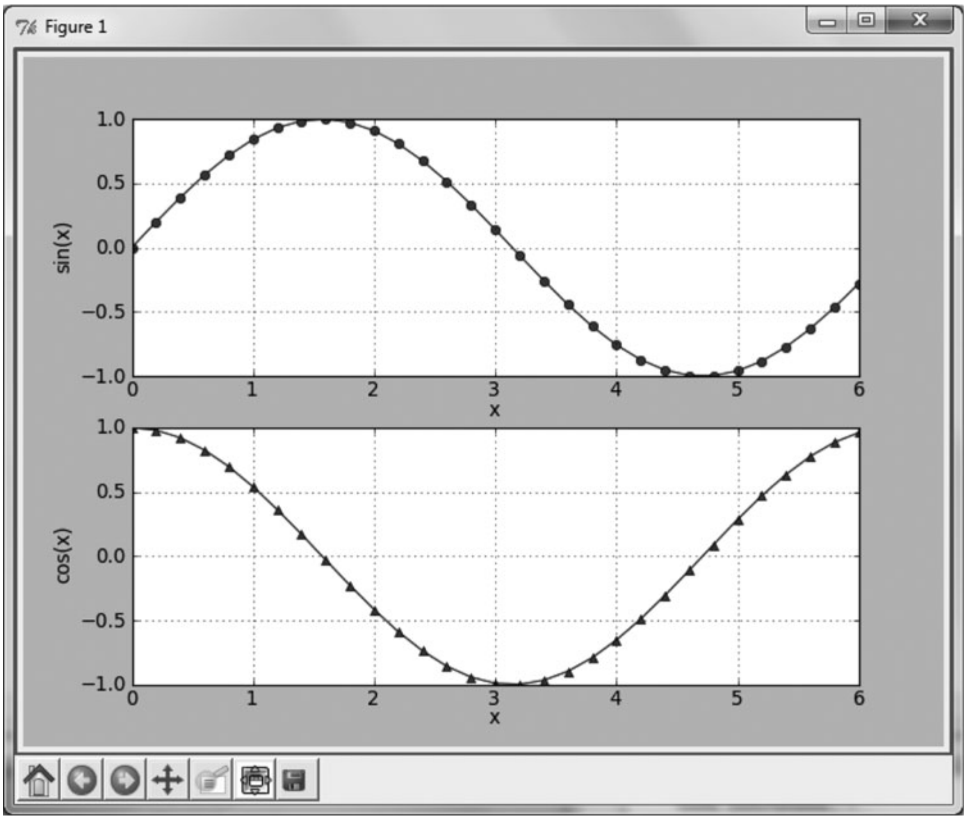


It is possible to have more than one plot in a figure, as demonstrated by the following code:

```
import matplotlib.pyplot as plt
from numpy import arange,sin,cos
x = arange(0.0,6.2,0.2)

plt.subplot(2,1,1)
plt.plot(x,sin(x),'o-')
plt.xlabel('x');plt.ylabel('sin(x)')
plt.grid(True)
plt.subplot(2,1,2)
plt.plot(x,cos(x),'^-')
plt.xlabel('x');plt.ylabel('cos(x)')
plt.grid(True)
plt.show()
input("\nPress return to exit")
```

The command `subplot(`*rows,cols,plot_number*`)` establishes a subplot window within the current figure. The parameters *row* and *col* divide the figure into *row* × *col* grid of subplots (in this case, two rows and one column). The commas between the parameters may be omitted. The output from this above program is



1.7 Scoping of Variables

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

1. Local namespace 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.
2. 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.
3. A 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 it 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:\Python32\test.py", line 6, in <module>
    print('a/b =',c)
NameError: name 'c' is not defined
```

1.8 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 on pressing the enter key. The interactive mode is a good way both 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, which 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

```
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 multi-user 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 *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)
    input('Press return to exit')
    sys.exit()
```

The docstring of a module can be printed with the statement

$\boxed{\texttt{print(}\textit{module\_name}\texttt{.\_\_doc\_\_)}}$

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

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

Avoid backslashes in the docstring because they confuse the Python 3 interpreter.