Python Basics

This notebook is provided in Numerical Comuting Course of CS-2008 for FAST-NUCES Karachi campus students for making the students self sufficient with the desired knowledge of Python to use in upcoming labs. All the material in this lab is taken from 'Numerical Computing with Python 3' by Jaan Kiusalaas published by Cambridge. Compiled By Ms.Amber Shaikh

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:

In [5]:
```
string1 = 'Welcome to learn Python'
string2 = 'for Numerical Computing'
print(string1 + ' ' + string2) # Concatenation

print(string2[4:23])           # Slicing

```

```
Welcome to learn Python for Numerical Computing
Numerical Computing
```

In [6]:
```
#A string can be split into its component parts using the split command. The components appear as elements in a list. For ex
s = '3 9 81'
print(s.split())   # Delimiter is white space

```

```
['3', '9', '81']
```

In [7]:
```
#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 immutabilitywill result in TypeError, as follows:
s = 'Press return to exit'
s[0] = 'p'

```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-b8d331975183> in <module>
      2 #and it has a fixed length. An attempt to violate immutabilitywill result in TypeError, as follows:
      3 s = 'Press return to exit'
----> 4 s[0] = 'p'

TypeError: 'str' object does not support item assignment
```

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.

In [11]:
```
b = 2
# b is integer type
print(b)
b = b*2.0 # Now b is float type
print(b)

```

```
2
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.

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):

In [12]:
```
rec = ('Smith','John',(6,23,68))
# This is a tuple
lastName,firstName,birthdate = rec # Unpacking the tuple
print(firstName)
birthYear = birthdate[2]
print(birthYear)
name = rec[1] + ' ' + rec[0]
print(name)

print(rec[0:2])

```

```
John
68
John Smith
('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:

In [13]:
```
a = [1.0, 2.0, 3.0]  # Create a list
a.append(4.0)        # Append 4.0 to list
print(a)

a.insert(0,0.0)      # Insert 0.0 in position 0
print(a)

print(len(a))        # Determine length of list

a[2:4] = [1.0, 1.0, 1.0] # Modify selected elements
print(a)

```

```
[1.0, 2.0, 3.0, 4.0]
[0.0, 1.0, 2.0, 3.0, 4.0]
5
[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:

In [14]:
```
a = [1.0, 2.0, 3.0]
b = a
# 'b' is an alias of 'a'
b[0] = 5.0
# Change 'b'
print(a)
# The change is reflected in 'a'
c = a[:]
# 'c' is an independent copy of 'a'
c[0] = 1.0
# Change 'c'
print(a)
# 'a' is not affected by the change
```

```
[5.0, 2.0, 3.0]
[5.0, 2.0, 3.0]
```

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

print(a[1][2])  # Print third element of second row
```

```
[4, 5, 6]
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 moreconvenient to employ array objects provided by the numpy module. Array objects are discussed later.

Arithmetic Operators

Python supports the usual arithmetic operators:

Airthmetic	Operations
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	**
Modular division	%

Some of these operators are also defined for strings and sequences as follows:

In [15]:
```
s = 'Hello'
t = 'to you'
a = [1, 2, 3]
print(3*s)       # Repetition

print(3*a)       # Repetition
[1, 2, 3, 1, 2, 3, 1, 2, 3]

print(a + [4, 5]) # Append elements

print(s + t)     # Concatenation

print(3 + s)    # This addition makes no sense

```

```
HelloHelloHello
[1, 2, 3, 1, 2, 3, 1, 2, 3]
[1, 2, 3, 4, 5]
Helloto you

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-c1db11289351> in <module>
     11 print(s + t)     # Concatenation
     12
---> 13 print(3 + s)    # This addition makes no sense

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.

Augmented	Assignment Operator
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

Comparison	Operators
<	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:

In [16]:
```
a = 2    # Integer
b = 1.99 # Floating point
c = '2'  # String
print(a > b)

print(a == c)

print((a > b) and (a != c))

print((a > b) or (a == b))

```

```
True
False
True
True
```

Conditionals

The if construct

```
if condition:
   block
```

executes a block of statements (which must be indented) if the condition returnsTrue. 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.

In [17]:
```
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))
```

```
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, theblock is executed again. This process is continued until the condition becomes False.Theelse 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, ...]:

In [18]:
```
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)

```

```
[1.0, 0.5, 0.3333333333333333, 0.25]
```

We met the for statement in Section 1.1. This statement requires a target and a sequence over which the target loops. The form of the construct is

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

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

In [19]:
```
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, ..., nMax1](created by calling the range function) is the sequence.
#Any loop can be terminated by the break statement.
```

```
[1.0, 0.5, 0.3333333333333333, 0.25, 1.0, 0.5, 0.3333333333333333, 0.25]
```

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:

In [20]:
```
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')

# While running code don't forget to give 'Tim' rather then Tim
```

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

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.

In [21]:
```
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)

```

```
[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:

Type	Conversion
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:

In [22]:
```
a = 5
b = -3.6
d = '4.0'
print(a + b)
1.4
print(int(b))


```

```
1.4
-3
```

In [23]:
```
print(complex(a,b))

```

```
(5-3.6j)
```

In [24]:
```
print(float(d))
```

```
4.0
```

In [25]:
```
print(int(d)) # This fails: d is a string
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-25-8855b2389f81> in <module>
----> 1 print(int(d)) # This fails: d is a string

ValueError: invalid literal for int() with base 10: '4.0'
```

In [26]:
```
print(int(d))
print(float(d))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-cca78df32116> in <module>
----> 1 print(int(d))
      2 print(float(d))

ValueError: invalid literal for int() with base 10: '4.0'
```

Mathematical Functions

Core Python supports only the following mathematical functions:

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)	1 if a<b returns, 0 if a=b 1 if a>b

#The majority of mathematical functions are available in the math module. In above table last three rows are showing different kind of returns of cmp(a,b)

Reading Input

The intrinsic function for accepting user input is 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:

In [27]:
```
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:23
23 <class 'str'>
23 <class 'int'>
```

-> A convenient way to input a number and assign it to the variable a is a = eval(input(prompt))

Printing Output

Output can be displayed with the print function

print(object1, object2, ...)

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 for newline. Forexample,

In [28]:
```
a = 1234.56789
b = [2,4,6,8]
print(a,b)

```

```
1234.56789 [2, 4, 6, 8]
```

In [29]:
```
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.Forexample,

```
print(object1, object2, ...,end=' ')
```

replaces \n with a space.

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

```
'{:fmt1}{:fmt2}...'.format(arg1,arg2,...)
```

where fmt1, fmt2,...are the format specifications for arg1, arg2,..., respectively. Typically used format specifications are

FormatSpecification	
wd	Integer
w.df	Floating point notation
w.de	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:

In [30]:
```
a = 1234.56789
print('{:7.2f}'.format(a))

```

```
1234.57
```

In [31]:
```
n = 9876
print('n = {:6d}'.format(n)) # Pad with spaces


```

```
n =   9876
```

In [32]:
```
print('n = {:06d}'.format(n)) # Pad with zeros


```

```
n = 009876
```

In [33]:
```
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

```
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:

Strings for Action	
'r'	Read from an existing file.
'w'	Write to a file. If filename doesnotexist,itiscreated.
'a'	Append to the end of the file.
'r+'	Read to and write from an existing file.
'w+'	Same as 'r+',butfilename 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. 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.
```

First make a file with above name and try to save it in current working directory then use the below code to read it.

In [ ]:
```
x =[]
data = open('sunspots.txt','r')
print(data.read())
```

Writing Data to a File

The method

```
file object.write(string)
```

writes a string to a file, whereas

```
file object.writelines(list of strings)
```

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:

In [35]:
```
f = open('testfile','w')
for k in range(101,111):
  f.write('{:4d} {:6d}'.format(k,k**2))
  f.write('\n')
f.close()


```

Type *Markdown* and LaTeX: α^2

In [ ]:
```
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:

In [37]:
```
c=12.0/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-37-0a9ae6d7debc> in <module>
----> 1 c=12.0/0

ZeroDivisionError: float division by zero
```

In [38]:
```
try:
  c = 12.0/0.0
except ZeroDivisionError:
  print('Division by zero')
```

```
Division by zero
```

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:

In [39]:
```
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 determinethe twoderivatives 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)
```

```
First derivative= 0.7999999995730867
Second derivative = -0.6399999918915711
```

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:

```
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. An example follows:

In [40]:
```
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'

```

```
[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:

In [41]:
```
c = lambda x,y : x**2 + y**2
print(c(3,4))

```

```
25
```

Modules

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.

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, andnumpy. 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:

In [42]:
```
from math import log,sin
print(log(sin(0.5)))

```

```
-0.7351666863853142
```

In [43]:
```
#alternate way to do above task
import math
print(math.log(math.sin(0.5)))
```

```
-0.7351666863853142
```

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:

In [44]:
```
import math as m
print(m.log(m.sin(0.5)))

```

```
-0.7351666863853142
```

In [45]:
```
#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 mo
import math
dir(math)
#Most of these functions are familiar to programmers. Note that the module includes two constants: π and e.
```

Out[45]:
```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'comb',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'dist',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'isqrt',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'perm',
 'pi',
 'pow',
 'prod',
 'radians',
 'remainder',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc']
```

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:

In [46]:
```
from cmath import sin
x = 3.0 -4.5j
y = 1.2 + 0.8j
z = 0.8
print(x/y)

print(sin(x))

print(sin(z))

```

```
(-2.562053133750361e-16-3.75j)
(6.352392998168388+44.55264336489803j)
(0.7173560908995228+0j)
```

NumpyModule

General Information

The numpy module 2 is not a part of the standard Python release. As pointed out ear- lier, 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 turnalistintoanarray:

```
array(list,type)
```

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

In [47]:
```
from numpy import array
a = array([[2.0, -1.0],[-1.0, 3.0]])
print(a)
b = array([[2, -1],[-1, 3]],float)
print(b)

```

```
[[ 2. -1.]
 [-1.  3.]]
[[ 2. -1.]
 [-1.  3.]]
```

Other available functions are

```
zeros((dim1,dim2),type)
```

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

```
ones((dim1,dim2),type)
```

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 sequence. Here are examples of creating arrays:

In [48]:
```
from numpy import*
print(arange(2,10,2))

```

```
[2 4 6 8]
```

In [49]:
```
print(arange(2.0,10.0,2.0))

```

```
[2. 4. 6. 8.]
```

In [50]:
```
print(zeros(3))

```

```
[0. 0. 0.]
```

In [51]:
```
print(zeros((3),int))


```

```
[0 0 0]
```

In [52]:
```
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:

In [53]:
```
from numpy import *
a = zeros((3,3),int)
print(a)
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)

```

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
[[ 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:

In [54]:
```
from numpy import array
a = array([0.0, 4.0, 9.0, 16.0])
```

In [56]:
```
print(a/16.0)

print(a - 4.0)

#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))

print(sin(a))
[ 0.84147098 -0.7568025 0.41211849 -0.28790332]


```

```
  File "<ipython-input-56-e14fe9875fd0>", line 11
    [ 0.84147098 -0.7568025 0.41211849 -0.28790332]
                             ^
SyntaxError: invalid syntax
```

In [57]:
```
#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       #observe the difference here.
a = array([1.0, 4.0, 9.0, 16.0])
print(sqrt(a[1]))

print(sqrt(a))
```

```
2.0

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-6581d8c51e03> in <module>
      6 print(sqrt(a[1]))
      7
----> 8 print(sqrt(a))

TypeError: only size-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:

In [58]:
```
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

print(diagonal(A,1)) # First subdiagonal

print(trace(A))      # Sum of diagonal elements

print(argmax(b))     # Index of largest element

print(argmin(A,axis=0)) # Indices of smallest col. elements

print(identity(3)) # Identity matrix


```

```
[4. 4. 3.]
[-2. -2.]
11.0
1
[1 0 1]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

There are three functions in numpy that compute array products. They are illus- trated by the following program. For more details, see Appendix A2.

In [60]:
```
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)

dot(A,x)

dot(A,B)

inner(x,y)

inner(A,x)

inner(A,B)

outer(x,y)

outer(A,x)



```

```
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]]
```

Out[60]:
```
array([[ 7,  3],
       [14,  6],
       [21,  9],
       [14,  6]])
```

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,

In [61]:
```
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


print(solve(A,b)) # Solve [A]{x} = {b}

```

```
[[0.33333333 0.16666667 0.        ]
 [0.16666667 0.45833333 0.25      ]
 [0.         0.25       0.5       ]]
[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,usethecopy method in the numpy module:

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

Vectorizing Algorithms

Sometimes the broadcasting properties of the mathematical functions in the numpy modulecan be usedtoreplace loops in thecode. This procedureisknown as vectorization. Consider, for example, the expression

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

In [63]:

```
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)))
```

```
0.0
0.005567412088789389
0.02130666125188408
0.05019772638947366
0.09462720088497198
0.156627299443006
0.2379809120320106 3
0.3402785764616709
0.46495319701811455
0.6133028550984008
0.7865066090173275
0.9856358762849615
1.2116628882528127
1.465467128065556
1.7478403369791122
2.0594904801403984
2.4010449420801847
2.7730531439606216
3.1759887223404
3.610251373290963
4.076168440399162
4.573996306995954
5.103921639613108
5.666062519725874
6.260469493310951
6.887126561968701
7.545952134863047
8.23679995719429
8.959460028104342
9.713659518649584
10.499063698637746
11.315276879618091
12.161843380067646
13.038248517776777
13.943919633563134
14.87822714970225
15.84048566583038
16.829955094531
17.84584183834496
18.887300009533142
19.953432693559613
21.04329325694438
22.155886699851774
23.29017105352728
24.445058822468535
25.619418471010903
26.812075953822262
28.021816289632216
29.247385177366347
30.487490653714246
31.740804791029746
33.00596543434154
34.28157797614137
35.56621716751434
36.85842896408068
38.156732405130235
39.459621524249094
40.76556728966208
42.07301957244456
43.38040914069208
44.68614967767645
45.98863982196134
47.28626522740016
48.57740064089195
49.86041199572946
51.1336585183346
52.395494846142796
53.64427315436682
54.87834528934404
56.0960649061479
57.29578960812473
58.47588308600086
59.63471725419251
60.77067438194127
61.882149216892344
62.96755109872974
64.02530606048323
65.05385891512552
66.0516753250849
67.01724385230862
67.94907798652503
68.84571814936862
69.70573367205115
70.52772474428349
71.31032433217798
72.05220006288847
```

```
72.7520560737754
73.40863482391671
74.0207188658207
74.58713257523576
75.10674383699254
75.57846568485782
76.00125789342519
76.37412852011609
76.6961353954155
76.96638755951953
77.18404664362748
77.3483281941681
77.45850293830978
77.5138979891652
77.51389798916522
77.51389798916512
```

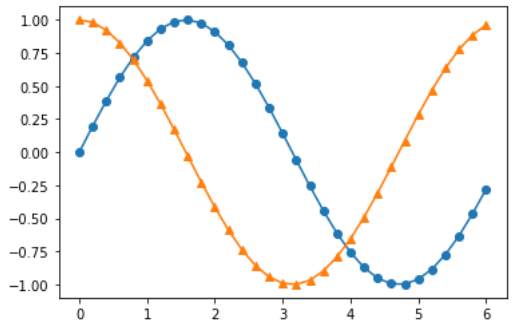
Note that the first algorithm uses the scalar versions of sqrt and sin functions in the math module, whereas the second algorithm imports these functions from numpy. The vectorized algorithm executes much faster, but uses more memory.

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.

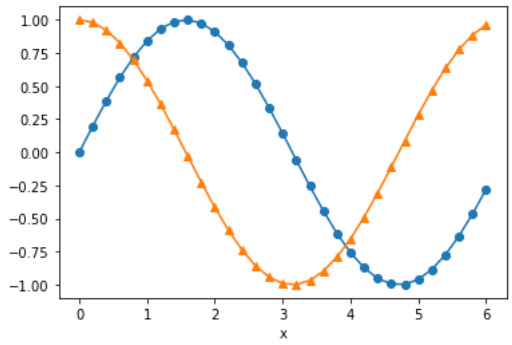
In [64]:
```
import matplotlib.pyplot as plt
from numpy import arange,sin,cos
x = arange(0.0,6.2,0.2)
plt.plot(x,sin(x),'o-',x,cos(x),'^-')    # Plot with specified  # line and marker style

```

Out[64]:
```
[<matplotlib.lines.Line2D at 0x7f7ce6272460>,
 <matplotlib.lines.Line2D at 0x7f7ce6272550>]
```



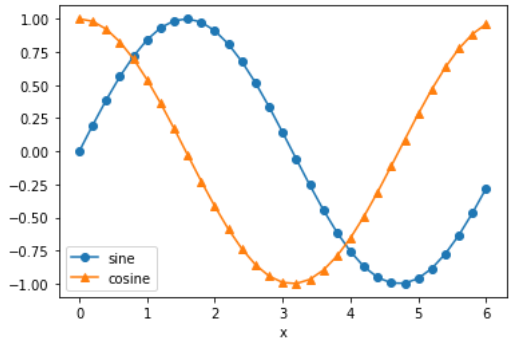
In [65]:
```
x = arange(0.0,6.2,0.2)
plt.plot(x,sin(x),'o-',x,cos(x),'^-')    # Plot with specified  # line and marker style
plt.xlabel('x')                          # Add label to x-axis

```

Out[65]:
```
Text(0.5, 0, 'x')
```

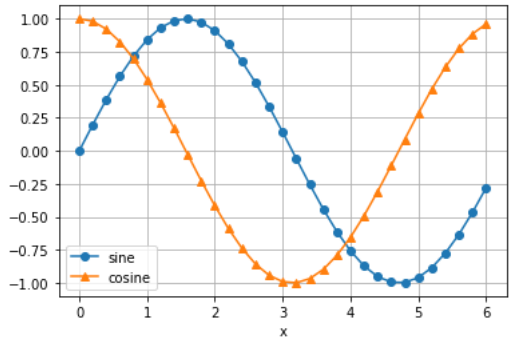


In [66]:
```
x = arange(0.0,6.2,0.2)
plt.plot(x,sin(x),'o-',x,cos(x),'^-')    # Plot with specified  # line and marker style
plt.xlabel('x')                          # Add label to x-axis
plt.legend(('sine','cosine'),loc = 0)    # Add legend in loc. 3

```

Out[66]: <matplotlib.legend.Legend at 0x7f7ce5d27eb0>



In [67]:
```
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

```



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

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

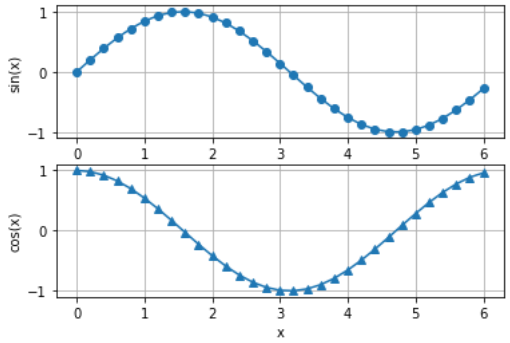
Some of the location (loc) codes for placement of the legend are
0 'Best' location
1 Upper right
2 Upper left
3 Lower left
4 Lower right
Running the program produces the following screen:

In [68]:
```
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")          # Press enter
```

```
<Figure size 432x288 with 0 Axes>


Press return to exit
```

Out[68]: ''

In [69]:
```
#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
#x col grid of subplots (in this case, two rows and one column). The commas between the parameters may be omitted.
```




```
Press return to exit
```

Out[69]: ''

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:

In [70]:
```
def divide():
  c = a/b
  print('a/b =',c)
a = 100.0
b = 5.0
divide()


```

```
a/b = 20.0
```

In [71]:
```
#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)
```

```
a/b= <function <lambda> at 0x7f7ce9373310>
```

In [ ]:
```

```