1 Python Programming Language

Python is a high-level programming language for general purpose developed by Guido van Rossum and first released in 1991. Several packages offer a wide range of functionalities, including graphical user interfaces, text processing, system administration, image processing, scientific computing, machine learning, databases, networking, computer vision, among others.

Several libraries are available to provide a powerful environment for developing code in Python:

- NumPy: optimized library for numerical operations, including support for large, multi-dimensional arrays and matrices.
- SciPy: library for scientific computing, including packages for optimization algorithms, linear algebra routines, numerical integration, interpolation tools, statistical functions, signal processing tools.
- Matplotlib: library for generation of plots, histograms, bar charts, scatter plots, among several other graphical resources.
- OpenCV: open-source library that includes computer vision and machines learning algorithms used for face detection and recognition, object tracking, camera calibration, point cloud generation from stereo cameras, among many others.
- Scikit-Image: open-source library that contains a collection of image processing algorithms.
- Scikit-Learn: open-source library that implements several algorithms for machine learning and tools for data analysis.
- Pandas: library that provides data structures, data analysis tools and operations for manipulating numerical tables and time series.

2 Reference Sources

Python: `https://wiki.python.org/moin/SimplePrograms`
`https://www.tutorialspoint.com/python/`

NumPy: `http://www.numpy.org/`

SciPy: `https://www.scipy.org/`

Matplolib: `http://matplotlib.org/`

OpenCV: `http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html`

Scikit-image: `http://scikit-image.org/`

Scikit-learn: `http://scikit-learn.org/`

Pandas: `http://pandas.pydata.org/`

3 Programming Modes

Python provides two different modes of programming.

3.1 Interactive Mode

Invoke the interpreter without passing a script parameter:

```
$ python
```

Output:

```
Python 3.6.10 (default, May 8 2020, 15:58:13)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Example:

```
>>> print("Hello, World!")
```

Output:

```
Hello, World!
```

3.2 Script Mode

Invoke the interpreter passing a script parameter. Assume that the following code is included into the "prog.py" file:

```
#!/usr/bin/python
print("Hello, World!")
```

Example:

```
$ python prog.py
```

Output:

```
Hello, World!
```

4 Indentation

Blocks of code are denoted by line indentation. Although the number of space in the indentation is variable, all statements within the block must be indented the same amount.

Correct:

```
if True:
   print("True")
else:
   print("False")
```

Incorrect:

```
if True:
   print("Answer")
   print("True")
else:
   print("Answer")
  print("False")
```

5 Quotation

Python accepts single (') and double (") quotes to denote string literals, as long as the same type of quote starts and ends the string.

Examples:

```
word = 'word'
sentence = "this is a sentence"
```

6 Comments

The character '#' that is not inside a string literal begins a comment. All characters after the '#' and up to the end of the line are part of the comment and the Python interpreter ignores them.

Example:

```
#!/usr/bin/python

# first comment
print("Hello, World!") # second comment
```

7 Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration occurs automatically a value is assigned to a variable. The assignment operator is the equal sign '='.

Example:

```
#!/usr/bin/python

counter = 50       # an integer
miles   = 7000.0   # a floating point
name    = "Mary"   # a string

print(counter)
print(miles)
print(name)
```

8 Multiple Assignment

Example:

```
a = b = c = 0
```

9 Standard Data Types

- Numbers
- String
- List
- Tuple
- Dictionary

9.1 Numerical Types

- int: signed integers
- long: long integers (they can also be represented in octal and hexadecimal)
- float: floating point real values
- complex: complex numbers

9.2 Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Pairs of single or double quotes can be used to denote string literals. Subsets of strings can be taken using the slice operator '`[ ]`' and '`[:]`' with indices starting at 0 in the beginning of the string and working their way from -1 at the end.

Example:

```
#!/usr/bin/python

str = 'Hello, World!'

print(str)          # print complete string
print(str[0])       # print first character of the string
print(str[2:5])     # print characters starting from 3rd to 5th
print(str[2:])      # print string starting from 3rd character
print(str * 2)      # print string two times
print(str + "!!")   # print concatenated string
```

```
Hello, World!
H
llo
llo, World!
Hello, World!Hello, World!
Hello, World!!!
```

9.3 Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets '[]'. Lists are similar to arrays in C programming language, however, the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator '[ ]' and '[:]' with indices starting at 0 in the beginning of the list and working their way to end -1. The plus '+' sign is the list concatenation operator, and the asterisk '*' is the repetition operator.

Example:

```
#!/usr/bin/python

list1 = ['mary', 258, 13.67, 'text', 12]
list2 = [158, 'mary']

print(list1)          # print complete list
print(list1[0])       # print first element of the list
print(list1[1:3])     # print elements starting from 2nd to 3rd
print(list1[2:])      # print elements starting from 3rd element
print(list2 * 2)      # print list two times
print(list1 + list2)  # print concatenated lists
```

Output:

```
['mary', 258, 13.67, 'text', 12]
mary
[258, 13.67]
[13.67, 'text', 12]
[158, 'mary', 158, 'mary']
['mary', 258, 13.67, 'text', 12, 158, 'mary']
```

9.4 Tuple

A tuple consists of a number of values separated by commas. Unlike lists, tuples are enclosed within parentheses.

The main differences between lists and tuples are: lists are enclosed in brackets '[ ]' and their elements and size can be changed, while tuples are enclosed in parentheses '( )' and cannot be updated. Tuples can be thought of as read-only lists.

Example:

```
#!/usr/bin/python

tuple1 = ('mary', 258, 13.67, 'text', 12)
tuple2 = (158, 'mary')

print(tuple1)           # print complete list
print(tuple1[0])        # print first element of the list
print(tuple1[1:3])      # print elements starting from 2nd to 3rd
print(tuple1[2:])       # print elements starting from 3rd element
print(tuple2 * 2)       # print list two times
print(tuple1 + tuple2)  # print concatenated lists
```

Output:

```
('mary', 258, 13.67, 'text', 12)
mary
(258, 13.67)
(13.67, 'text', 12)
(158, 'mary', 158, 'mary')
('mary', 258, 13.67, 'text', 12, 158, 'mary')
```

9.5 Dictionary

Dictionaries are known as associative arrays or associative memories, working in a similar way as hash tables. Dictionaries consist of key-value pairs. Unlike arrays, which are indexed by a range of numbers, dictionaries are indexed by unique keys (usually numbers or strings). A value is associated with each key.

Dictionaries are enclosed by curly braces '{ }' and values can be assigned and accessed using square braces '[]'.

Example:

```
#!/usr/bin/python

dict1 = {}
dict1['one'] = "This is one"
dict1[2]     = "This is two"

dict2 = {'name': 'john', 'code': 6734, 'dept': 'sales'}

print(dict1['one'])    # print value for 'one' key
print(dict1[2])        # print value for 2 key
print(dict2)           # print complete dictionary
print(dict2.keys())    # print all the keys
print(dict2.values())  # print all the values
```

Output:

```
This is one
This is two
{'name': 'john', 'code': 6734, 'dept': 'sales'}
['name', 'code', 'dept']
['john', 6734, 'sales']
```

10 Basic Operators

Python language supports a set of operators for manipulating expressions.

10.1 Arithmetic Operators

`+ - * / % ** //`

Example:

```
print(5*7)
```

Output:

```
35
```

Example:

```
print(3**2)
```

Output:

```
9
```

Example:

```
print(6.0 // 3.5)
```

Output:

```
1.0
```

10.2 Relational Operators

`==  !=  >  <  >=  <=`

Example:

```
print(7 > 10)
```

Output:

```
False
```

Example:

```
print(4 == 4)
```

Output:

```
True
```

10.3 Assignment Operators

`=  +=  -=  *=  /=  %=  **=  //=`

Example:

```
a = 5
a *= 3
print(a)
```

Output:

```
15
```

10.4 Logical Operators

`and  or  not`

Example:

```
age = 32
salary = 35000
print((age > 25) and (salary > 40000))
```

Output:

```
False
```

10.5 Bitwise Operators

`&  |  ^  ~  <<  >>`

Example:

```
a = 00111100   # decimal 60
b = 00001101   # decimal 13

print(a & b)
print(a | b)
print(a ^ b)
print(~a)
```

Output:

```
00001100   # decimal 12
00111101   # decimal 61
00110001   # decimal 49
11000011   # decimal -61
```

Example: Test if number 'x' is even or odd

```
x = int(input('Enter a number: '))
print('x is %s.' % ('even', 'odd')[x&1])
```

10.6 Memberships Operators

Membership operators test for membership in a sequence, such as strings, lists or tuples.

```
in   not in
```

Example:

```
s = 'This is a string'
for c in s:
    print(c, end=" ")
print('p' in s)
print(' ' in s)
print('y' in s)
```

Output:

```
T h i s   i s   a   s t r i n g
False
True
False
```

Example:

```
#!/usr/bin/python

a = 10
b = 20
list = [1, 2, 3, 4, 5];

if (a in list):
   print("a is available in the given list")
else:
   print("a is not available in the given list")

if (b not in list):
   print("b is not available in the given list")
else:
   print("b is available in the given list")
```

Output:

```
a is not available in the given list
b is not available in the given list
```

10.7 Identity Operators

Identity operators compare the memory locations of two objects.

```
is   is not
```

Example:

```
#!/usr/bin/python

a = 20
b = 20

if a is b:
   print("a and b have same identity")
else:
   print("a and b do not have same identity")

if id(a) == id(b):
   print("a and b have same identity")
else:
   print("a and b do not have same identity")

b = 30
if a is b:
   print("a and b have same identity")
else:
   print("a and b do not have same identity")
```

Output:

```
a and b have same identity
a and b have same identity
a and b do not have same identity
```

11 Conditional Statements

```
score = float(input("Enter score: "))
if score >= 50:
   print('Pass')
else:
   print('Fail')
```

```
score = float(input("Enter score: "))
if score >= 90:
   grade = 'A'
else:
   if score >= 80:
      grade = 'B'
   else:
      if score >= 70:
         grade = 'C'
      else:
         if score >= 60:
            grade = 'D'
         else:
            grade = 'F'
print('\nGrade is: %s' % grade)
```

```
score = float(input("Enter score: "))
if score >= 90:
   grade = 'A'
elif score >= 80:
   grade = 'B'
elif score >= 70:
   grade = 'C'
elif score >= 60:
   grade = 'D'
else:
   grade = 'F'
print('\nGrade is: %s' % grade)
```

12 Loop Statements

Python provides two loop control statements, which allow us to execute a command or block of blocks multiple times.

12.1 While Loop

It repeats a statement or block of statements while a given condition is true. It tests the condition before executing the loop body.

Example:

```
total = 0.0
count = 0
while count < 10:
    number = float(input("Enter a number: "))
    count = count + 1
    total = total + number
average = total / count
print("The average is " + str(average))
```

Example:

```
count = 0
while count < 9:
   print('The count is:', count)
   count = count + 1
print("Goodbye!")
```

12.2 For Loop

It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Example:

```
for x in range(0, 5):
    print("Index %d" % (x))
```

Example:

```
for x in range(1, 11):
    for y in range(1, 11):
        print("%d * %d = %d" % (x, y, x*y))
```

Example:

```
for x in range(3):
    print(x)
    if x == 1:
       break
```

Example:

```
string = "Hello, World!"
for x in string:
    print(x)
```

Example:

```
fruits = ['strawberry', 'banana', 'apple',  'mango']
for index in range(len(fruits)):
    print('Current fruit :', fruits[index])
print("Goodbye!")
```

13 Functions

Functions are a convenient way to divide a code into smaller portions in order to provide better modularity, maintainability, reusability, understandability.

13.1 Definition

A function is a block of organized, reusable code that is used to perform a single, related action.

Example:

```
def print_result(score):
  if score >= 50:
     print('Pass')
  else:
     print('Fail')
  return
```

Example:

```
def result(score):
  if score >= 50:
     return 'Pass'
  else:
     return 'Fail'
```

13.2 Function Call

Example:

```
print_result(51)
result(92)
```

Output:

```
Pass
'Pass'
```

13.3 Passing by reference versus value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Example:

```
#!/usr/bin/python

# function definition
def change_list(mylist):
   mylist.append([1, 2, 3, 4]);
   print("Values inside the function: ", mylist)
   return

# function call
mylist = [10, 20, 30];
change_list(mylist);
print("Values outside the function: ", mylist)
```

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

13.4 Scope of Variables

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python: local and global variables.

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope. This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions.

```
#!/usr/bin/python

total = 0  # this is global variable.
# function definition
def sum(arg1, arg2):
   total = arg1 + arg2  # total is local variable here
   print("Inside the function local total: ", total)
   return total;

# function call
sum(10, 20);
print("Outside the function global total: ", total)
```

Output:

```
Inside the function local total:  30
Outside the function global total:  0
```

14 Modules

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

A module is basically a file consisting of Python code that can define functions, classes and variables.

Example: module myFunctions.py

```
def mycube(y):
    return y * y * y

def mydouble(z):
 return 2 * z
```

14.1 The import Statement

A Python source file can be used as a module by executing an import statement in some other Python source file. The import has the following syntax:

```
import module1, module2, ..., moduleN
```

Example:

```
#!/usr/bin/python

import myFunctions

print("1 to 5 cubed")
for x in range(1, 6):
    print(myFunctions.mycube(x), end=" "),
print()
print()

print("1 to 5 doubled")
for x in range(1, 6):
    print(myFunctions.mydouble(x), end=" ")
```

Output:

```
1 to 5 cubed
1 8 27 64 125

1 to 5 doubled
2 4 6 8 10
```

14.2 The from...import Statement

Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax:

```
from modname import name1, name2, ..., nameN
```

Example: to import the function for the Fibonacci sequence from the module fib:

```
from fib import fibonacci
```

14.3 The from...import * Statement

It is also possible to import all names from a module into the current namespace:

```
from modname import *
```

Although this provides an easy way to import all the items from a module into the current namespace, this statement should be used sparingly.

15 Input and Output

There are several ways to read information from the keyboard or a file, as well as present the output of a program on the screen or write information to another file.

15.1 Reading Keyboard Input

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard:

- raw_input: reads one line from standard input and returns it as a string (removing the trailing newline)
- input: equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

Example:

```
#!/usr/bin/python

str = raw_input("Enter your input: ");
print("Received input is : ", str)
```

Output: when “Hello, World!” is typed

```
Enter your input: Hello, World!
Received input is :  Hello, World!
```

Example:

```
#!/usr/bin/python

str = input("Enter your input: ");
print("Received input is : ", str)
```

Output:

```
Enter your input: [x*5 for x in range(2,10,2)]
Received input is :  [10, 20, 30, 40]
```

15.2 Printing to the Screen

The simplest way to produce output is using the print statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output.

Example:

```
#!/usr/bin/python

print("This is an example of string")
```

Output:

```
This is an example of string
```

15.3 Opening and Closing Files

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object.

`open  file  close  read  write  tell  seek  rename  remove`

Example:

```
#!/usr/bin/python

# open a file
f = open("input.txt", "r+")
str = f.read(10);
print("Read string is : ", str)

# check current position
position = f.tell();
print("Current file position : ", position)

# reposition pointer at the beginning once again
position = f.seek(0, 0);
str = f.read(10);
print("Again read string is : ", str)
# close opened file
f.close()
```

Output: assuming the input file contains string “Python language”

```
Read String is :  Python lan
Current file position :  10
Again read String is :  Python lan
```

16 NumPy Arrays

An array holds and represents regular data in a structured way.

```
import numpy as np
a = np.zeros(shape=(3,2))
a
```

```
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

```
a[0] = [1,2]
a[1] = [2,3]
a
```

```
array([[ 1.,  2.],
       [ 2.,  3.],
       [ 0.,  0.]])
```

Examples:

```
# create an array of ones
np.ones((3,4))

# check shape of an array
x = np.ones((3,4))
print(x.shape)

# create an array of zeros
np.zeros((2,3,4), dtype=np.int16)

# create an array with random values
np.random.random((2,2))

# create an empty array
np.empty((3,2))

# create a full array with a constant to be inserted into the array
np.full((2,2),7)

# create an array of evenly-spaced values
np.arange(10,25,5)

# create an array of evenly-spaced values
np.linspace(0,2,9)
```

Equivalently to the operators `+`, `-`, `*`, `/` or `%`, functions `np.add()`, `np.subtract()`, `np.multiply()`, `np.divide()` and `np.remainder()` can be used to do arithmetic operations with arrays.

Some other useful functions:

```
a.sum()          # array-wise sum
a.min()          # array-wise minimum value
a.max(axis=0)    # maximum value of an array row
a.cumsum(axis=1) # cumulative sum of the elements starting from 0
a.cumprod()      # cumulative product of the elements starting from 1
a.mean()         # mean
a.median()       # median
a.corrcoef()     # correlation coefficient
np.var()         # variance
np.std(a)        # standard deviation
```

Axes are defined for arrays with more than one dimension. In a 2D array, we can apply an operation vertically across rows (axis 0) or horizontally across columns (axis 1).

```
x = np.arange(12).reshape((3,4))
print(x)
print(x.sum(axis=1))
```

```
[[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]]

[ 6, 22, 38]
```

Differences between matrix multiplication and element-wise matrix multiplication:

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
# common matrix product
x = np.dot(a,b)
print(x)
# element-wise matrix multiplication (Hadamard product)
y = np.multiply(a,b)
print(y)
# element-wise matrix multiplication (Hadamard product)
z = a * b
print(z)
```

```
[[19, 22],
 [43, 50]]

[[ 5, 12],
 [21, 32]]

[[ 5, 12],
 [21, 32]]
```

Operations for selecting elements, slicing regions or indexing arrays:

```
array1d = np.arange(10)
print(array1d)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(array1d[5])
```

```
5
```

```
print(array1d[5:8])
```

```
[5, 6, 7]
```

```
array1d[5:8] = 12
print(array1d)
```

```
[ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9]
```

```
array2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
print(array2d[2])
```

```
[7, 8, 9]
```

```
print(array2d[0][2])
```

```
3
```

```
print(array2d[:2])
```

```
[[1, 2, 3],
 [4, 5, 6]]
```

```
print(array2d[0:2,1])
```

```
[2, 5]
```

```
print(array2d[:2, 1:])
```

```
[[2, 3],
 [5, 6]]
```

```
print(array2d[1:3,2])
```

```
[6, 9]
```

```
print(array[1:5:2,::3])
```

```
[[ 4]
 [10]]
```

```
array3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(array3d)
```

```
[[[ 1,  2,  3],
  [ 4,  5,  6]],
 [[ 7,  8,  9],
  [10, 11, 12]]]
```

```
print(array3d[0])
```

```
[[1, 2, 3],
 [4, 5, 6]]
```

```
print(array3d[1, 0])
```

```
[7, 8, 9]
```

```
print(array3d[1,...])
```

```
[[ 7  8  9]
 [10 11 12]]
```

In essence, the following holds:

```
a[start:end]  # items start through the end (but the end is not included)
a[start:]     # items start through the rest of the array
a[:end]       # items from the beginning through the end (but the end is not included)
```

```
# select values from an array that fulfill a certain condition
a = np.random.random((1,7))
print(a[a < 0.6])
```

Two ways to transpose a two-dimensional array:

```
print(np.transpose(a))
print(a.T)
```

Arrays can be reshaped or resized to make them compatible for desired arithmetic operations.

To resize an array, its new dimensions can be passed to the `np.resize()` function. If the new array is larger than the original one, the new array will be filled with copies of the original array that are repeated as many times as necessary.

Example:

```
a = np.array([[0,1],[2,3]])
np.resize(a,(2,3))
```

```
array([[0, 1, 2],
       [3, 0, 1]])
```

```
np.resize(a,(1,4))
```

```
array([[0, 1, 2, 3]])
```

```
np.resize(a,(2,4))
```

```
array([[0, 1, 2, 3],
       [0, 1, 2, 3]])
```

It is also possible to reshape an array, that is, a new shape to an array is given without changing its data. The key to reshaping is to make sure that the total size of the new array is unchanged.

```
x = np.ones((3,4))

# print the size of x
print(x.size)

# reshape x to (2,6)
y = x.reshape((2,6))
```

Another operation to change the shape of arrays is `ravel()`, which is used to flatten arrays. This means that n-D arrays are converted to a 1-D array.

```
# flatten x
z = x.ravel()

# print z
print(z)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

It is possible to append arrays to an original array, where they are attached to the end of that original array.

```
# append a 1D array to array 'array1d'
new_array1d = np.append(array1d, [7, 8, 9, 10])

# print new array
print(new_array1d)

# append an extra column to array 'array2d'
# axis 1 indicates the columns, while axis 0 indicates the rows in 2D arrays
new_array2d = np.append(array2d, [[7], [8]], axis=1)

# print new array
print(new_array2d)
```

There are also functions for inserting of deleting array elements.

```
# insert '5' at index 1
np.insert(array1d, 1, 5)

# delete the value at index 1
np.delete(array1d, [1])
```

There are several functions to merge or join arrays. Some are listed below:

```
# concatenate 'array1d' and 'x'
print(np.concatenate((array1d, x)))

# stack arrays row-wise
print(np.vstack((array1d, array2d)))

# stack arrays row-wise
print(np.r_[new_array1d, array2d])

# stack arrays horizontally
print(np.hstack((new_array1d, array2d)))

# stack arrays column-wise
print(np.column_stack((new_array1d, array2d)))

# stack arrays column-wise
print(np.c_[new_array1d, array2d])

# split 'my_stacked_array' horizontally at the 2nd index
print(np.hsplit(my_stacked_array, 2))

# split 'my_stacked_array' vertically at the 2nd index
print(np.vsplit(my_stacked_array, 2))
```

17 Vectorized Computation

Vectorized array computation provides a fast way to compute operations for manipulating data, such as filtering, transformation, summarizing, aggregation.

Example: mathematical operations performed on entire blocks of data using similar syntax to the equivalent operations between scalar elements:

```
A = np.array([[0.84, -0.26,  0.92],
              [0.42,  0.31, -0.73]])

B = A * 10
print(B)

C = A + A
print(C)
```

```
[[ 8.4 -2.6  9.2]
 [ 4.2  3.1 -7.3]]

[[ 1.68 -0.52  1.84]
 [ 0.84  0.62 -1.46]]
```

To set all of the negative values in the array to 0:

```
A[A < 0] = 0
```

```
[[ 0.84 0.0  0.92]
 [ 0.42 0.31 0.  ]]
```

Example: inner matrix product $Y = X^T X$

```
X = np.random.randn(6, 3)
Y = np.dot(X.T, X)
print(Y)
```

```
[[ 4.27527809 -0.24339761 -2.65985356]
 [-0.24339761  4.76032665 -1.07444714]
 [-2.65985356 -1.07444714  5.46999821]]
```

For higher dimensional arrays, the transpose statement accepts a tuple of axis numbers to permute the axes:

```
X = np.arange(16).reshape((2, 2, 4))
print(X)
Y = X.transpose((1, 0, 2))
print(Y)
```

```
[[[ 0,  1,  2,  3],
  [ 4,  5,  6,  7]],
 [[ 8,  9, 10, 11],
  [12, 13, 14, 15]]]

[[[ 0,  1,  2,  3],
  [ 8,  9, 10, 11]],
 [[ 4,  5,  6,  7],
  [12, 13, 14, 15]]]
```

The `numpy.where` statement is a vectorized version of the ternary expression `x if condition else y`.

Example: take a value from `xarray` whenever the corresponding value in `cond` is True, otherwise take the value from `yarray`. A possible solution might be:

```
xarray = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarray = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])

result = [(x if c else y)
          for x, y, c in zip(xarray, yarray, cond)]
print(result)
```

```
[1.1, 2.2, 1.3, 1.4, 2.5]
```

This solution has some problems. It will not be fast for large arrays. Moreover, it will not work with multidimensional arrays. A more concise with `numpy.where` would be:

```
result = np.where(cond, xarray, yarray)
print(result)
```

```
[ 1.1,  2.2,  1.3,  1.4,  2.5]
```

The `where` function can be cleverly used to express more complicated logic. Consider this example where we have two boolean arrays, `cond1` and `cond2`, and wish to assign a different value for each of the 4 possible pairs of boolean values:

```
result = []
for i in range(n):
    if cond1[i] and cond2[i]:
        result.append(0)
    elif cond1[i]:
        result.append(1)
    elif cond2[i]:
        result.append(2)
    else:
        result.append(3)
```

The `for` loop can be converted into a nested `where` expression:

```
np.where (cond1 & cond2, 0,
     np.where (cond1, 1,
          np.where (cond2, 2, 3)))
```

Mathematical and statistical functions can be calculated over an entire array or along an axis.

Examples:

```
x = np.random.randn(5, 4)  # normally-distributed data

print(x.mean())
print(np.mean(x))
print(x.sum())
print(x.mean(axis=1))
print(x.sum(0))
```

```
0.158657175642
0.158657175642
3.17314351285
[ 0.0936983   0.61586662 -0.5950235   0.28796628  0.39077817]
[-0.41126513  2.74393207 -2.46900462  3.30948119]
```

NumPy arrays can be sorted in-place using the `sort` method:

```
x = np.random.randn(6)
print(x)

x.sort()
print(x)
```

```
[-2.0282963   0.13228264 -0.19936979 -0.06106798 -0.13287158 -1.33947695]

[-2.0282963  -1.33947695 -0.19936979 -0.13287158 -0.06106798  0.13228264]
```

NumPy has some basic set operations for one-dimensional arrays. A commonly used one is `np.unique`, which returns the sorted unique values in an array:

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
print(np.unique(names))

ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
print(np.unique(ints))
```

```
['Bob' 'Joe' 'Will']
[1 2 3 4]
```

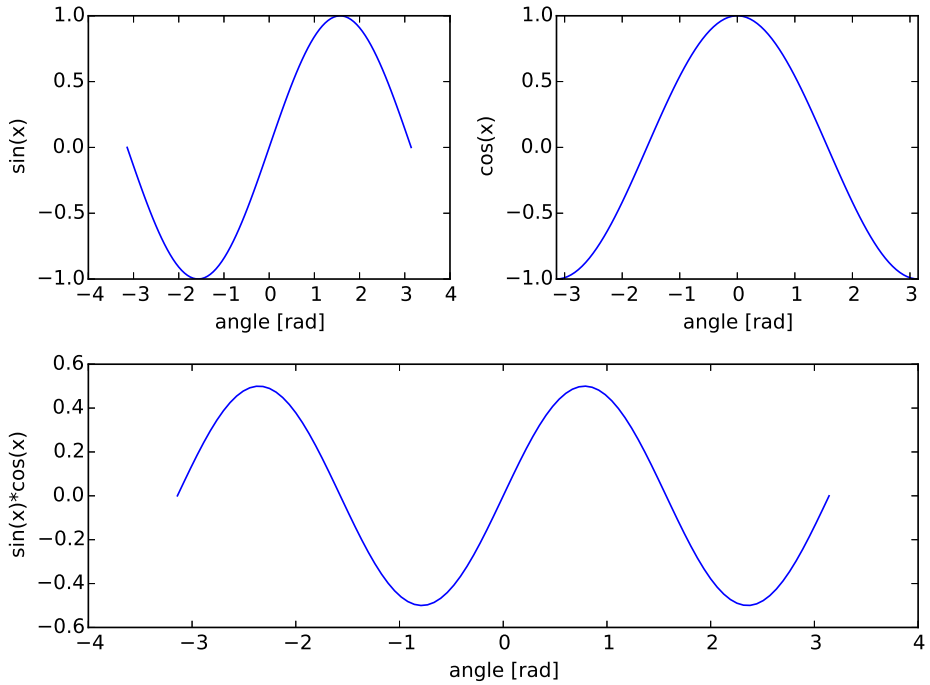
The function `np.in1d` tests membership of the values in one array in another, returning a boolean array:

```
values = np.array([6, 0, 0, 3, 2, 5, 6])
print(np.in1d(values, [2, 3, 6]))
```

```
[ True False False  True  True False  True]
```

To calculate and plot sine and cosine values of an array of angles given in radians:

```
import matplotlib.pylab as plt
import numpy as np
x = np.linspace(-np.pi, np.pi, 100)
plt.subplot(2,2,1)
plt.plot(x, np.sin(x))
plt.xlabel('angle [rad]')
plt.ylabel('sin(x)')
plt.subplot(2,2,2)
plt.plot(x, np.cos(x))
plt.xlabel('angle [rad]')
plt.ylabel('cos(x)')
plt.axis('tight')
plt.subplot(2,1,2)
plt.plot(x, np.sin(x) * np.cos(x))
plt.title('sin(x)*cos(x)')
plt.tight_layout()
plt.show()
```



18 Images using NumPy and SciPy

```
from scipy import misc
from scipy import ndimage
import numpy as np
import matplotlib.pyplot as plt

# open image file and stores it in a numpy array
img = misc.imread('iris.png')

# print image dimensions and type
print(img.shape, img.dtype)

# show image
plt.imshow(img, cmap='gray')
plt.show()

#  save image in PNG format
misc.imsave('iris.png', img)

# calculate some statistical information
print(img.min(), img.mean(), img.max())

# apply rotation transformation
f = np.flipud(img)
plt.imshow(f)
plt.show()

# smooth image with Gaussian filter
g = ndimage.gaussian_filter(img, sigma=7)
h = ndimage.gaussian_filter(img, sigma=11)
plt.imshow(g)
plt.show()
plt.imshow(h)
plt.show()
```

19 Exercises

1. Initialize a 2D array $M_{5\times10}$ with value 7.
   Non-vectorized version:
   ```
   M = np.empty([5, 10])
   for i in range(5):
       for j in range(10):
           M[i][j] = 7
   ```
   Vectorized version 1:
   ```
   M = np.full((5, 10), 7)
   ```
   Vectorized version 2:
   ```
   M = np.ones((5, 10)) * 7
   ```
2. Vectorize the following code:
   ```
   for i in range(200):
       for j in range(400):
           if data[i][j] > 0:
              data[i][j] = 0
   ```
   Vectorized version:
   ```
   data[data > 0] = 0
   ```
3. Reverse an array.
   Non-vectorized version:
   ```
   a = np.arange(10)
   b = np.copy(a)
   n = len(a)
   for i in range(n):
       b[i] = a[n-i-1]
   ```
   Vectorized version 1:
   ```
   b = a[::-1]
   ```
   Vectorized version 2:
   ```
   b = np.flipud(a)
   ```
4. Obtain the mean of each column of a 2D array x. Ignore elements that are less than or equal to 0.
   ```
   a = np.array([[2.0, 4.0, 5.0], [-3.0, 8.0, -7.0], [0.0, 6.0, 4.0]])
   mask = a > 0
   col_sums = np.sum(a * mask, axis=0)
   counts = np.sum(mask, axis=0)
   means = col_sums / counts
   ```
5. Remove from a vector all the elements that are equal to the biggest element.
   ```
   x = np.array([6, 0, 0, 3, 2, 5, 6])
   max_value = np.max(x)
   indices = np.where(x == max_value)
   new_x = np.delete(x, indices)
   ```
6. A vector x contains some 0s. Create y such that $y[i]$ is 0 if $x[i]$ is 0, otherwise $y[i]$ is $log(x[i])$.
   ```
   x = np.array([0, 2, 3, 5, 0, 7, 9, 0, 4])
   indices = x > 0
   y = np.zeros(len(x))
   y[indices] = np.log(x[indices])
   ```
7. Calculate the sum of the elements along the main diagonal of a square matrix `M`.
   Non-vectorized version:
   ```
   def diagonal_sum(M):
       count = 0.0
       for i in xrange(0, len(M)):
           count += M[i][i]
       print(count)
   ```
   Vectorized version:
   ```
   print(M.trace())
   ```
8. Check if a matrix `M` is a lower, upper or diagonal triangular matrix.

```
# check if lower triangular matrix
np.allclose(M, np.tril(M))

# check if upper triangular matrix
np.allclose(M, np.triu(M))

# check if diagonal triangular matrix
np.allclose(M, np.diag(np.diag(M)))
```

9. Vectorize the following code:

```
def threshold(M, max_value, min_value):
    n = M.shape[0]
    m = M.shape[1]
    for i in range(n):
        for j in range(m):
            c = M[i][j]
            if c > max_value:
               M[i][j] = max_value
            elif c < min_value:
               M[i][j] = min_value
    return M
```

Vectorized version:

```
def threshold(M, max_value, min_value):
    M[M > max_value] = max_value
    M[M < min_value] = min_value
    return M
```

10. Vectorize the following code:

```
x = np.linspace(-2., 2., 5)
y = np.zeros(x.shape)

for i in range(len(x)):
    if x[i] <= 0.5:
       y[i] = x[i]**2
    else:
       y[i] = -x[i]
```

Vectorized version:

```
y = np.where(x <= 0.5, x**2, -x)
```

11. Computation of $\sin(x)$.
Non-vectorized version:

```
n = 1000000
x = np.linspace(0, 1, n+1)

def sin_func(x):
    res = np.zeros_like(x)
    for i in range(len(x)):
        res[i] = np.sin(x[i])
    return res

y = sin_func(x)
```

Vectorized version:

```
y = np.sin(x)
```

12. Problem: sum up two lists of integers.
One simple way using pure Python is:

```
def add_python(Z1,Z2):
    return [z1+z2 for (z1,z2) in zip(Z1,Z2)]
```

This first naive solution can be vectorized using numpy:

```
def add_numpy(Z1,Z2):
    return np.add(Z1,Z2)
```

The second method is the fastest with one order of magnitude. Not only is the second approach faster,

but it also naturally adapts to the shape of `Z1` and `Z2`. This is the reason why we did not write `Z1 + Z2` because it would not work if `Z1` and `Z2` were both lists. In the first Python method, the inner `+` is interpreted differently depending on the nature of the two objects such that if we consider two nested lists, we get the following outputs:

```
>>> Z1 = [[1, 2], [3, 4]]
>>> Z2 = [[5, 6], [7, 8]]
>>> Z1 + Z2
[[1, 2], [3, 4], [5, 6], [7, 8]]
>>> add_python(Z1, Z2)
[[1, 2, 5, 6], [3, 4, 7, 8]]
>>> add_numpy(Z1, Z2)
[[ 6  8]
 [10 12]]
```

The first method concatenates the two lists together, the second method concatenates the internal lists together and the last one computes what is (numerically) expected.

13. Given two vectors `X` and `Y`, we want to compute the sum of `X[i]*Y[j]` for all pairs of indices `i`, `j`. One simple and obvious solution is to write:

```
def compute_python(X, Y):
    result = 0
    for i in range(len(X)):
        for j in range(len(Y)):
            result += X[i] * Y[j]
    return result
```

However, this first and naive implementation requires two loops and we already know it will be slow. A way to vectorize the problem is to identify from linear algebra concepts that the expression `X[i]*Y[j]` to be very similar to a matrix product expression. Therefore, we could benefit from some numpy speedup. One wrong solution would be:

```
def compute_numpy_wrong(X, Y):
    return (X*Y).sum()
```

This is wrong because the `X*Y` expression will actually compute a new vector `Z` such that `Z[i] = X[i]*Y[i]` and this is not what we want. Instead, we can exploit numpy broadcasting by first reshaping the two vectors and then multiply them:

```
def compute_numpy(X, Y):
    Z = X.reshape(len(X),1) * Y.reshape(1,len(Y))
    return Z.sum()
```

Here we have `Z[i,j] == X[i,0]*Y[0,j]` and if we take the sum over each elements of `Z`, we get the expected result. This is better, we gained a factor of approximately 150 (assuming an array of 1000 elements). However, we can do much better. If we look again and more closely at the pure Python version, we can see that the inner loop is using `X[i]` that does not depend on the `j` index, meaning it can be removed from the inner loop. Code can be rewritten as:

```
def compute_numpy_better_1(X, Y):
    result = 0
    for i in range(len(X)):
        Ysum = 0
        for j in range(len(Y)):
            Ysum += Y[j]
        result += X[i] * Ysum
    return result
```

Since the inner loop does not depend on the `i` index, we might as well compute it only once:

```
def compute_numpy_better_2(X, Y):
    result = 0
    Ysum = 0
    for j in range(len(Y)):
        Ysum += Y[j]
    for i in range(len(X)):
        result += X[i] * Ysum
    return result
```

We have removed the inner loop, meaning a $O(n^2)$ complexity into $O(n)$ complexity with transform. Using the same approach, we can now write:

```
def compute_numpy_better_3(x, y):
    Ysum = 0
    for j in range(len(y)):
        Ysum += y[j]
    Xsum = 0
    for i in range(len(y)):
        Xsum += y[i]
    return Xsum * Ysum
```

Finally, having realized we only need the product of the sum over `X` and `Y` respectively, we can benefit from the `np.sum` function and write:

```
def compute_numpy_better(x, y):
    return np.sum(y) * np.sum(x)
```

It is shorter, clearer and much, much faster.
We have indeed reformulated our problem, taking advantage of the fact that $\sum_{i,j} X_i Y_j = \sum_i X_i \sum_j Y_j$.
In summary, there are types of vectorization: code vectorization and problem vectorization. The latter is the most difficult but the most important because this is where we can expect huge gains in speed. In this simple example, we gain a factor of 150 with code vectorization but we gained a factor of 70,000 with problem vectorization (assuming an array of 1000 elements), just by writing our problem differently (even though we cannot expect such a huge speedup in all situations). However, code vectorization remains an important factor, and if we rewrite the last solution the Python way, the improvement is good but not as much as in the numpy version:

```
def compute_python_better(x, y):
    return sum(x) * sum(y)
```

This new Python version is much faster than the previous Python version, but still, it is 50 times slower than the numpy version.