Python

Python is a high-level, dynamically typed multiparadigm programming language

In [2]:
```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

In [3]:
```
print(quicksort([10,8,8,1,9,7,9]))
```

```
[1, 7, 8, 8, 9, 9, 10]
```

Python versions

> python 2 version no longer available
>
> currently we are using python 3

- Python is very simple to pick up;
- Packages useful for ML are available in Python;
- Jupyter Notebooks for interactive programming;
- Extensively used in the industry;
- Python is much more general purpose programming language;

Keywords and identifiers

- Keywords are the reserved words in python;
- We can’t use a keyword as variable name, function name or any other identifier; Keywords are case sensitive;
- [Import keyword]
  - Example: False, None, True, class, if, else, return, def, try, while, for, etc Total number of keywords: 36
- Identifiers:
- Name given to entities like class, functions and variables
- Can be a combination of letters, digits and underscores, cannot start with a digit Keywords cannot be used as - identifiers, special characters cannot be used
- Python has straight forward Error indications;

In [4]:
```
import keyword
```

In [6]:
```
print(len(keyword.kwlist))
```

```
36
```

- Name given to entities like class, functions and variables
- Can be a combination of letters, digits and underscores, cannot start with a digit
- Keywords cannot be used as identifiers, special characters cannot be used
- Python has straight forward Error indications;

Comments, indentations and statements

- Start a line with a # or use triple quotes, ‘” ‘”
- Indentations are used (4 spaces preferred) to make blocks of code, a for loop
- Rather than writing code in a single line try to write in multiple lines (can use) to make code readable
- The written instructions are called statements

Variables and data types in python

- Variable is a location in memory used to store some data;
- Variable declaration is not needed `a, b = 10, 'Hi'`
- id(a) prints location of a Data types:
- Everything in python is an object; **Number: Integers, float and complex**
- Boolean: True and False
- Strings: Sequence of Unicode characters, defined with quotes, indexable, slicable
- List: An ordered sequence of items, like an array, can have multiple data type elements, defined with square brackets; Lists are mutable
- Tuple: Defined with parenthesis, can have multiple data type elements, tuple is immutable, can be indexable
- Set: Defined with Curly braces, Set is an unordered collection of unique items; behaves as a set in mathematics; does not support indexing
- Dictionary: an unordered collection of key-value pairs, defined with curly braces and a colon, value accessible with key
- Data types can be converted provided the value is valid in both data types; `List('Hello') = ['H','e','l','l','o']`

In [7]:
```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
x += 1
print(x)  # Prints "4"
x *= 2
print(x)  # Prints "8"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

```
<class 'int'>
3
4
2
6
9
4
8
<class 'float'>
2.5 3.5 5.0 6.25
```

Booleans:

Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (&&, ||, etc.):

In [8]:
```
t = True
f = False
print(type(t))
print(t and f)
print(t or f)
print(not t)
print(t != f)
```

```
<class 'bool'>
False
True
False
True
```

Strings: Python has great support for strings:

- Sequence of characters: (Unicode (default) or ASCII) S = “kl” or = str(1)
- Access characters of a string as a list;
- Strings are immutable;
- Operations: str1+str2; for i in string: __; lower(), upper(), join(), split(), find(), replace() `“Bad Morning”.replace(“Bad”, “Good”)`

In [9]:
```
hello = 'hello'
world = "world"
print(hello)
print(len(hello))
hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string form
atting
print(hw12)  # prints "hello world 12"
```

```
hello
5
hello world
hello world 12
```

In [10]:
```
s = "hello"
print(s.capitalize())
print(s.upper())
print(s.rjust(7))
print(s.center(7))
print(s.replace('l', '(ell)'))

print('  world '.strip())
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

Containers

- Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Lists

- A list is the Python equivalent of an array, but is resizeable and can contain elements of different types:

- Data Structures: collection of data elements
- List: Sequence data structures, these are indexable, mutable, defined by square brackets and elements are comma separated **Operations on list:**

```
 len(list), append(element), insert(index, element), remove(element)
(removes first occurrence only), list.append(element), list.extend(list),
pop(index)
['one', 'two'].append(['one', 'two']) = ['one', 'two', ['one', 'two']]
['one', 'two'].extend(['one', 'two']) = ['one', 'two', 'one', 'two']
del lst[1]
list reverse: list.reverse()
sorted(list), list.sort()
lst = [1, 2, 3, 4, 5]; abc = lst; abc.append(6); print(lst)[1, 2, 3, 4,
5, 6]
lst and abc are pointers;
string.split(' ')
lst[index]
lst[slice_index_start: slice_index_end]
lst1 + lst2
lst.count()
for ele in lst: print(ele)
List comprehensions: [i**2 for i in range(10) if i%2 ==0]
[[row[i] for row in matrix] for i in range(4)]
```

In [11]:
```
xs = [3, 1, 2]
print(xs, xs[2])
print(xs[-1])
xs[2] = 'foo'
print(xs)
xs.append('bar')
print(xs)
x = xs.pop()
print(x, xs)
```

```
[3, 1, 2] 2
2
[3, 1, 'foo']
[3, 1, 'foo', 'bar']
bar [3, 1, 'foo']
```

Slicing:

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing:

In [12]:
```
nums = list(range(5))
print(nums)
print(nums[2:4])
print(nums[2:])
print(nums[:2])
print(nums[:])
print(nums[:-1])
nums[2:4] = [8, 9]
print(nums)
```

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

Loops:

You can loop over the elements of a list like this:

```
 Control flow: while loop
While loop: block of code runs until a test expression is true; lst = [10, 20,
30, 40, 50]
index = 0
while index < len(lst):
product * = lst[index]
index += 1 # increment statement is important 0: 1*10, 1: 10*20, 2: 200*30,...
We can use an else block when the test condition fails;
```

```
 Control flow: for loop
Used to iterate over a sequence; for element in sequence:
statement(s) for ele in lst:
product *=ele
range(): range(10) will generate a list of 10 numbers from 0 to 9;
```

In [13]:
```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
```

```
cat
dog
monkey
```

In [14]:
```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
```

```
#1: cat
#2: dog
#3: monkey
```

List comprehensions:

When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

In [15]:
```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)   # Prints [0, 1, 4, 9, 16]
```

```
[0, 1, 4, 9, 16]
```

We can simplified by above code

In [16]:
```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]
```

```
[0, 1, 4, 9, 16]
```

Dictionary

- An unordered collection of key value pairs; O(1) for time complexity for search tasks; Dictionary is mutable; `Operations: dict.pop(key), dict.clear(), dict.fromkeys(list, values), dict.items(), .keys(), .values(), .copy()`
  `Dictionary Comprehension:`
  `for pair in d.items(): print(pair)`
  `{k:v for k,v in d.items() if v>2}`
  `{k:v for k+'c',v*2 in d.items() if v>2}`

In [17]:
```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])
print('cat' in d)
d['fish'] = 'wet'
print(d['fish'])
# print(d['monkey'])
print(d.get('monkey', 'N/A'))
print(d.get('fish', 'N/A'))
del d['fish']
print(d.get('fish', 'N/A'))
```

```
cute
True
wet
N/A
wet
N/A
```

In [19]:
```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A {0} has {1} legs'.format(animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

In [20]:
```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)  # Prints "{0: 0, 2: 4, 4: 16}"
```

```
{0: 0, 2: 4, 4: 16}
```

Sets

- Sets are unordered collection of unique items; Mutable, non-indexable; `S = {1, 2, 3}`
  `Sets does not allow duplicate numbers;`
  `set([1,2,3,1]) = (1, 2, 3)`
  `Operations: set.update(elements or sets), set.discard(element),`
  `set.remove(element), set.pop(), s.clear()`
  `Set1 | Set2: Union; Set1.union(Set2)`
  `Set1 & Set2, Set1.intersection(Set2)`
  `Set1 – Set2, Set1.difference(Set2)`
  `Set1^Set2, Set1.symmetric_difference(Set2)): Union - Intersection`
  `Frozenset: immutable sets: Set1 = frozenset([1,2,3,4])`

In [21]:
```
animals = {'cat', 'dog'}
print('cat' in animals)
print('fish' in animals)
animals.add('fish')
print('fish' in animals)
print(len(animals))
animals.add('cat')
print(len(animals))
animals.remove('cat')
print(len(animals))
```

```
True
False
True
3
3
2
```

In [22]:
```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

```
#1: fish
#2: dog
#3: cat
```

In [23]:
```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)  # Prints "{0, 1, 2, 3, 4, 5}"
```

```
{0, 1, 2, 3, 4, 5}
```

Tuple

- A tuple is similar to list; Tuple is immutable, its elements cannot be altered;
- T = “abcd”, # comma is important to create a tuple
- Tuple access: T[1]
- Changing a tuple: a list in a tuple is mutable;
- Concat tuples using +

- Deletion: whole tuple will be deleted
- Tuple.count(), tuple.index(element), element in tuple, element not in tuple, len(tuple), sorted(tuple), min(tuple), max(tuple), sum(tuple)

In [24]:
```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with
tuple keys
t = (5, 6)
print(type(t))
print(d[t])
print(d[(1, 2)])
```

```
<class 'tuple'>
5
1
```

Functions

- Functions: a group of related statements that perform a specific task;
- Converts a program into smaller chunk which makes management easy

def function():

- Doc string ‘” statements return
- Doc strings is written to explain the working of the function (function.**doc**)
- Scope and Life Time of Variables: Portion of the code where the variable is recognized and Lifetime is the period throughout
- which the variable exists in memory
- Variable inside a function are local variables which are destroyed once the function finishes execution; - - - Global variables are not destroyed unless deleted;
- Program to print highest common factor (HCF):
- def computeHCF(a, b): “””
- Computing HCF of two numbers “””
- Smaller =b if a>b else a
- hcf = 1

`for i in range(1, smaller + 1):

if (a%i==0) and (b%i==0): hcf = i return hcf `

In [25]:
```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"
```

```
negative
zero
positive
```

Classes

- The syntax for defining classes in Python is straightforward:

In [26]:
```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred')
g.greet()
g.greet(loud=True)
```

```
Hello, Fred
HELLO, FRED!
```

Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

In [27]:
```
import numpy as np

a = np.array([1, 2, 3])
print(type(a))
print(a.shape)
print(a[0], a[1], a[2])
a[0] = 5
print(a)

b = np.array([[1,2,3],[4,5,6]])
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
(2, 3)
1 2 4
```

In [28]:
```
import numpy as np

a = np.zeros((2,2))
print(a)


b = np.ones((1,2))
print(b)

c = np.full((2,2), 7)
print(c)


d = np.eye(2)
print(d)


e = np.random.random((2,2))
print(e)
```

```
[[0. 0.]
 [0. 0.]]
[[1. 1.]]
[[7 7]
 [7 7]]
[[1. 0.]
 [0. 1.]]
[[0.89032768 0.43775097]
 [0.65058856 0.74750872]]
```

Array indexing

- Numpy offers several ways to index into arrays.
- **Slicing:** Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

In [29]:
```
import numpy as np
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a[:2, 1:3]
print(a[0, 1])
b[0, 0] = 77
print(a[0, 1])
```

```
2
77
```

In [30]:
```
import numpy as np
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
row_r1 = a[1, :]
row_r2 = a[1:2, :]
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
print(col_r2, col_r2.shape)
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[ 2  6 10] (3,)
[[ 2]
 [ 6]
 [10]] (3, 1)
```

- Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

In [31]:
```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

```
[1 4 5]
[1 4 5]
[2 2]
[2 2]
```

In [32]:
```
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
b = np.array([0, 2, 0, 1])
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
a[np.arange(4), b] += 10
print(a)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[ 1  6  7 11]
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing:

- Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

In [33]:
```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2)
print(bool_idx)
print(a[bool_idx])
print(a[a > 2])     # Prints "[3 4 5 6]"
```

```
[[False False]
 [ True  True]
 [ True  True]]
[3 4 5 6]
[3 4 5 6]
```

Datatypes

- Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

In [34]:
```
import numpy as np

x = np.array([1, 2])
print(x.dtype)

x = np.array([1.0, 2.0])
print(x.dtype)

x = np.array([1, 2], dtype=np.int64)
print(x.dtype)
```

```
int64
float64
int64
```

numpy math

In [35]:
```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
print(x + y)
print(np.add(x, y))
print(x - y)
print(np.subtract(x, y))
print(x * y)
print(np.multiply(x, y))
print(x / y)
print(np.divide(x, y))
print(np.sqrt(x))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

In [36]:
```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])
print(v.dot(w))
print(np.dot(v, w))
print(x.dot(v))
print(np.dot(x, v))
print(x.dot(y))
print(np.dot(x, y))
```

```
219
219
[29 67]
[29 67]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

In [37]:
```
import numpy as np

x = np.array([[1,2],[3,4]])

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

```
10
[4 6]
[3 7]
```

In [38]:
```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)

print(x.T)
v = np.array([1,2,3])
print(v)
print(v.T)
```

```
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
[1 2 3]
[1 2 3]
```

Broadcasting

- Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

In [39]:
```
import numpy as np
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)
for i in range(4):
    y[i, :] = x[i, :] + v
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

In [40]:
```
import numpy as np
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))
print(vv)
y = x + vv
print(y)
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

In [41]:
```
import numpy as np
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

In [42]:
```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
print(np.reshape(v, (3, 1)) * w)
x = np.array([[1,2,3], [4,5,6]])
print(x + v)
print((x.T + w).T)
print(x + np.reshape(w, (2, 1)))
print(x * 2)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
[[2 4 6]
 [5 7 9]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 2  4  6]
 [ 8 10 12]]
```

SciPy

Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays. SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications.

In [6]:
```
import numpy as np
from scipy.misc import imread, imsave, imresize
import matplotlib.pyplot as plt
img = imread('cat.jpeg')
print(img.dtype, img.shape)  # Prints "uint8 (400, 248, 3)"
img_tinted = img * [1, 0.95, 0.9]
img_tinted = imresize(img_tinted, (300, 300))
imsave('cat_tinted.jpeg', img_tinted)
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.subplot(1, 2, 2)
plt.imshow(np.uint8(img_tinted))
plt.show()
```

```
uint8 (400, 248, 3)
```

MATLAB files

The functions `scipy.io.loadmat` and `scipy.io.savemat` allow you to read and write MATLAB files.

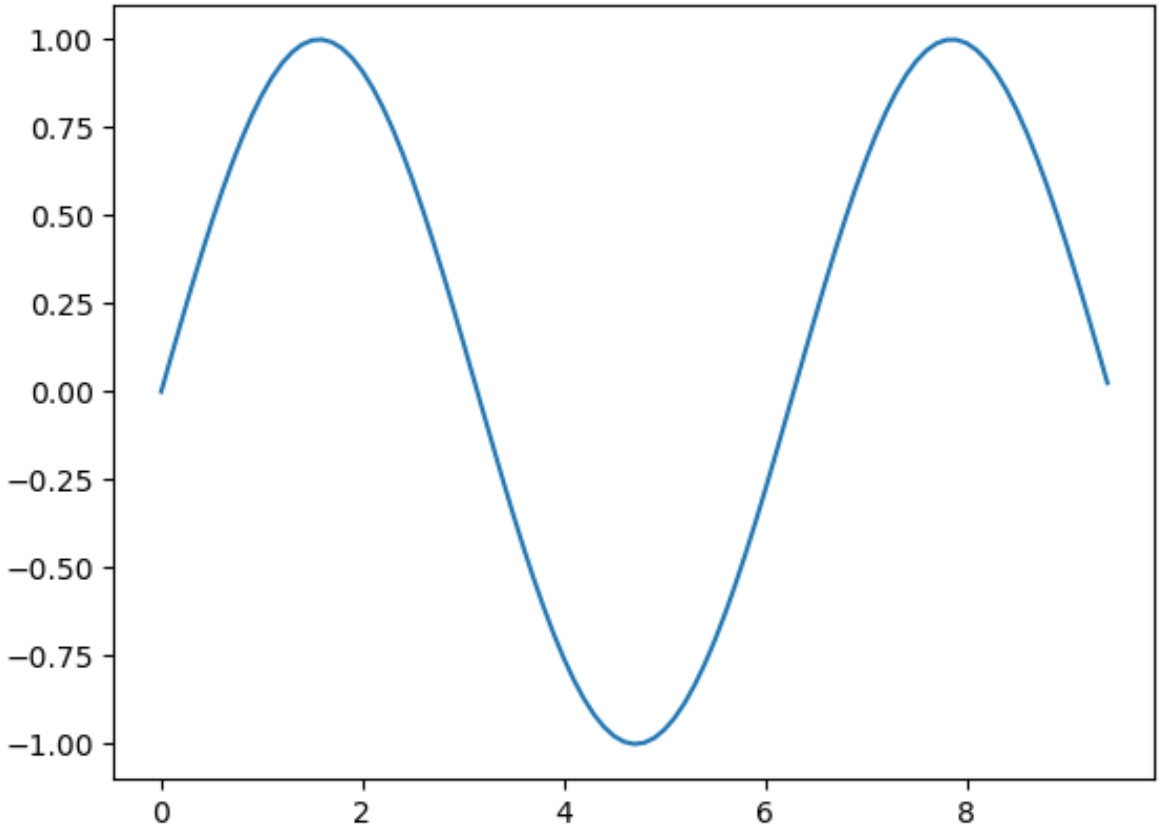
In [7]:
```
import numpy as np
from scipy.spatial.distance import pdist, squareform
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)
d = squareform(pdist(x, 'euclidean'))
print(d)
```

```
[[0 1]
 [1 0]
 [2 0]]
[[0.         1.41421356 2.23606798]
 [1.41421356 0.         1.        ]
 [2.23606798 1.         0.        ]]
```

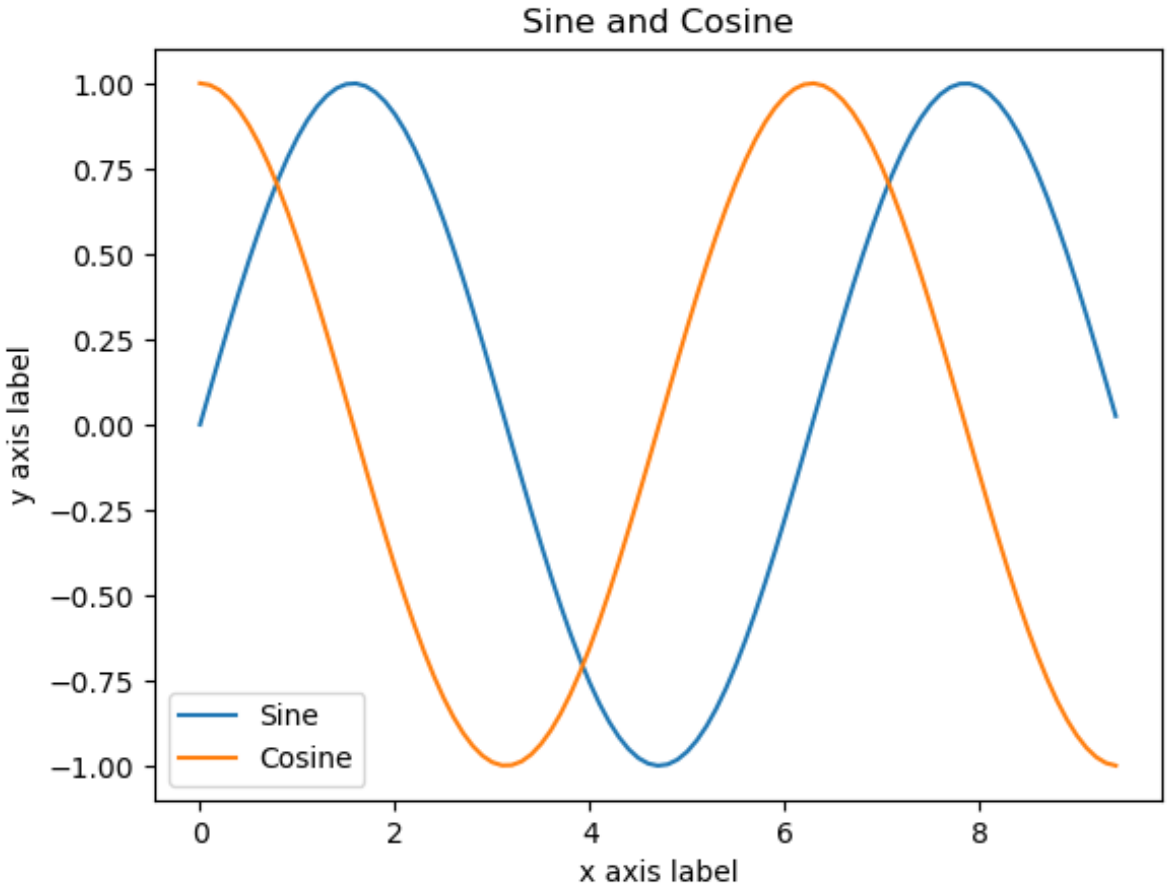
Plotting

- The most important function in matplotlib is `plot` , which allows you to plot 2D data. Here is a simple example:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```



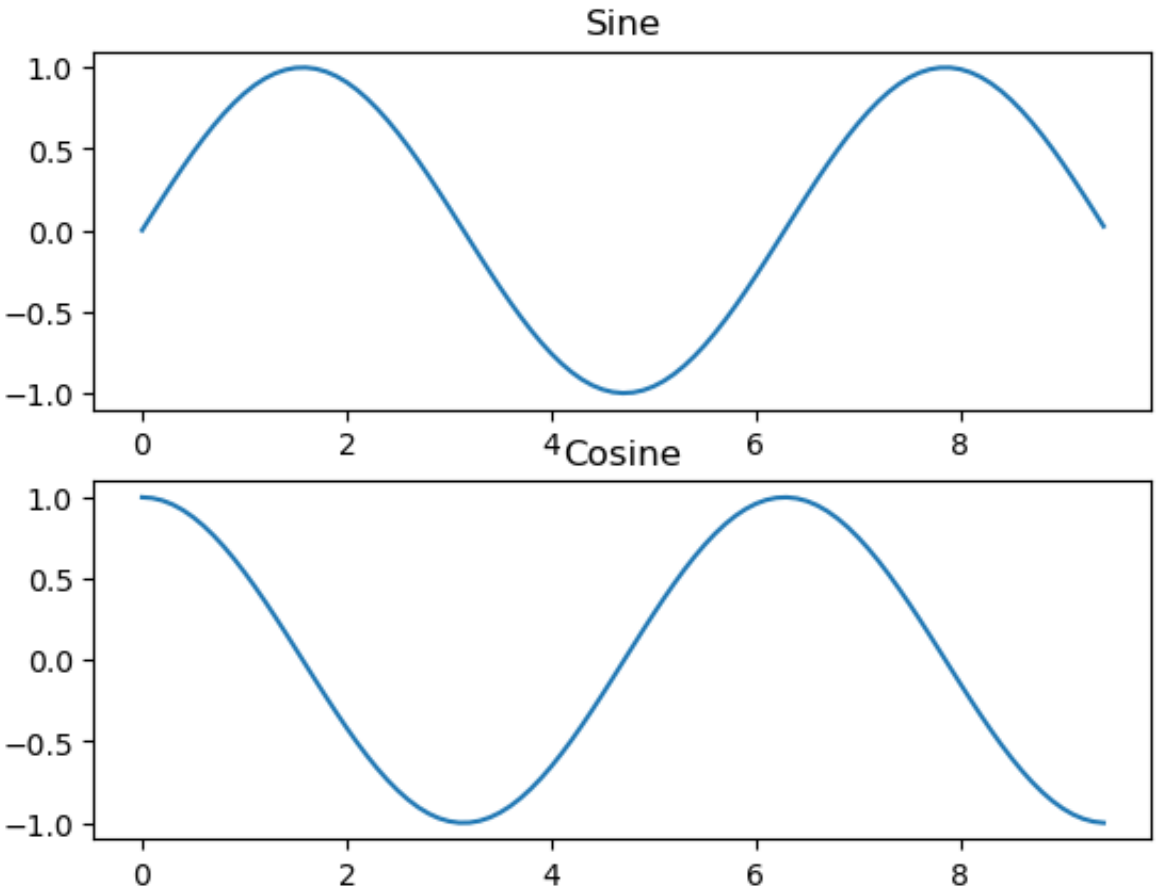
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



Subplots

- You can plot different things in the same figure using the subplot function. Here is an example:

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)
plt.subplot(2, 1, 1)
plt.plot(x, y_sin)
plt.title('Sine')
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')
plt.show()
```



Images

- You can use the imshow function to show images. Here is an example:

In [4]:
```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt
img = imread('cat.jpeg')
img_tinted = img * [1, 0.95, 0.9]
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.subplot(1, 2, 2)
plt.imshow(np.uint8(img_tinted))
plt.show()
```

In [ ]:

Object-Oriented Programming (OOP)'s concepts in Python

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual objects.

> Defining a Class in Python

Class Definition

`class ClassName:

```
    # Statement`
```

Object Definition

```
 obj = ClassName()
print(obj.atrr)
```

> Syntax

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

example

`class Dog:

```
    # Class attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
        `
```

In [3]:
```
# Python3 program to
# demonstrate instantiating
# a class


class Dog:

    # A simple class
    # attribute
    attr1 = "mammal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)


# Driver code
# Object instantiation
Rodger = Dog()

# Accessing class attributes
# and method through objects
print(Rodger.attr1)
Rodger.fun()
```

```
mammal
I'm a mammal
I'm a dog
```

In [5]:
```
# Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)


p = Person('Shiva')
p.say_hi()
```

```
Hello, my name is Shiva
```

Class and Instance Variables

> Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Defining instance variables using a constructor.

In [6]:
```
# Python3 program to show that the variables with a value
# assigned in the class declaration, are class variables and
# variables inside methods and constructors are instance
# variables.

# Class for Dog

class Dog:

    # Class Variable
    animal = 'dog'

    # The init method or constructor
    def __init__(self, breed, color):

        # Instance Variable
        self.breed = breed
        self.color = color


# Objects of Dog class
Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")

print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)

# Class variables can be accessed using class
# name also
print("\nAccessing class variable using class name")
print(Dog.animal)
```

```
Rodger details:
Rodger is a dog
Breed:  Pug
Color:  brown

Buzo details:
Buzo is a dog
Breed:  Bulldog
Color:  black

Accessing class variable using class name
dog
```

Defining instance variables using the normal method.

In [7]:
```
# Python3 program to show that we can create
# instance variables inside methods

# Class for Dog


class Dog:

    # Class Variable
    animal = 'dog'

    # The init method or constructor
    def __init__(self, breed):

        # Instance Variable
        self.breed = breed

    # Adds an instance variable
    def setColor(self, color):
        self.color = color

    # Retrieves instance variable
    def getColor(self):
        return self.color


# Driver Code
Rodger = Dog("pug")
Rodger.setColor("brown")
print(Rodger.getColor())
```

```
brown
```

Constructors are generally used for instantiating an object.

- The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the **init**() method is called the constructor and is always called when an object is created.

> Syntax of constructor declaration :

` def **init**(self):

    # body of the constructor `

In [8]:
```
#default constructor :
class dummy:

    # default constructor
    def __init__(self):
        self.geek = "dummy"

    # a method for printing data members
    def print_Geek(self):
        print(self.geek)


# creating object of the class
obj = dummy()

# calling the instance method using the object obj
obj.print_Geek()
```

```
dummy
```

In [9]:
```
# parameterized constructor :

class Addition:
    first = 0
    second = 0
    answer = 0

    # parameterized constructor
    def __init__(self, f, s):
        self.first = f
        self.second = s

    def display(self):
        print("First number = " + str(self.first))
        print("Second number = " + str(self.second))
        print("Addition of two numbers = " + str(self.answer))

    def calculate(self):
        self.answer = self.first + self.second

# creating object of the class
# this will invoke parameterized constructor
obj = Addition(1000, 2000)

# perform Addition
obj.calculate()

# display result
obj.display()
```

```
First number = 1000
Second number = 2000
Addition of two numbers = 3000
```

Inheritance in py

inheritance is the capability of one class to derive or inherit the properties from another class.

> Benefits of inheritance are:

1. It represents real-world relationships well.
2. It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows 3. us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Inheritance Syntax

```
Class BaseClass:
    {Body}
Class DerivedClass(BaseClass):
    {Body}
```

In [10]:
```
# A Python program to demonstrate inheritance

class Person(object):

  # Constructor
  def __init__(self, name, id):
    self.name = name
    self.id = id

  # To check if this person is an employee
  def Display(self):
    print(self.name, self.id)


# Driver code
emp = Person("Satyam", 102) # An Object of Person
emp.Display()
```

```
Satyam 102
```

In [12]:
```
class Emp(Person):

  def Print(self):
    print("Emp class called")

Emp_details = Emp("shiva", 103)

# calling parent class function
Emp_details.Display()

# Calling child class function
Emp_details.Print()
```

```
shiva 103
Emp class called
```

In [14]:
```
# A Python program to demonstrate inheritance

# Base or Super class. Note object in bracket.
# (Generally, object is made ancestor of all classes)
# In Python 3.x "class Person" is
# equivalent to "class Person(object)"


class Person(object):

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name

    # To check if this person is an employee
    def isEmployee(self):
        return False


# Inherited or Subclass (Note Person in bracket)
class Employee(Person):

    # Here we return true
    def isEmployee(self):
        return True


# Driver code
emp = Person("heyyy")  # An Object of Person
print(emp.getName(), emp.isEmployee())

emp = Employee("heyyy11")  # An Object of Employee
print(emp.getName(), emp.isEmployee())
```

```
heyyy False
heyyy11 True
```

In [16]:
```
# Python code to demonstrate how parent constructors
# are called.

# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

# child class


class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)


# creation of an object variable or an instance
a = Employee('shiva', 886012, 200000, "Intern")

# calling a function of the class Person using its instance
a.display()
```

```
shiva
886012
```

In [18]:
```
class A:
    def __init__(self, n='shiva'):
        self.name = n


class B(A):
    def __init__(self, roll):
        self.roll = roll


object = B(23)
print(object.name)
```

```
---------------------------------------------------------------------
---------
AttributeError                            Traceback (most recent c
all last)
/var/folders/nj/5n17tpm16j1dszf8dzt_qfj00000gn/T/ipykernel_1388/27
42791510.py in <module>
     10
     11 object = B(23)
---> 12 print(object.name)

AttributeError: 'B' object has no attribute 'name'
```

In [ ]: