Introduction in Python

Python

- uses new line to complete a command, as opposed to other programming languages which often use semicolons or parentheses.

- relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

- Primitive data types
  - **Numbers: int** and **float** works as you would expect from other languages.

```
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"
```

    * Python does not have unary increment (x++) or decrement (x--) operators.

  - **Boolean:** Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols.

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```

  - **Strings:** Python has great support for strings.

```
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print(hello)       # Prints "hello"
print(len(hello))  # String length; prints "5"
```

```
hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style
                                        # string formatting
print(hw12)  # prints "hello world 12"
```

- String objects have a bunch of useful methods; for example:

```
s = "hello"
print(s.capitalize())  # Capitalize a string;
                       # prints "Hello"
print(s.upper())       # Convert a string to uppercase;
                       # prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding
                       # with spaces; prints "  hello"
print(s.center(7))     # Center a string, padding
                       # with spaces; prints " hello "
print(s.replace('l', '(ell)'))  #Replace all instances of one
                                #substring  with another;
                                #prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing
                           # whitespace; prints "world"
```

- Collections
  - 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**:**

```
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])  # Prints "[3, 1, 2] 2"
print(xs[-1])     # Negative indices
                  # count from the end
                  # of the list;
                  # prints "2"
xs[2] = 'foo'     # Lists can contain elements
                  # of different types
print(xs)         # Prints "[3, 1, 'foo']"
xs.append('bar')  # Add a new element to
                  # the end of the list
print(xs)         # Prints "[3, 1, 'foo', 'bar']"
```

```
x = xs.pop()      # Remove and return the
                  # last element of the list'
y = xs.pop(-1)    # Remove and return the
                  # i-th element of the list'
print(x, xs, y)   # Prints "foobar [3, 1, 'foo'] bar"
```

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

```
nums = list(range(5)) # range is a built-in function
                      # that creates a list of integers
print(nums)           # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])      # Get a slice from index 2 to 4
                      # (exclusive); prints "[2, 3]"
print(nums[2:])       # Get a slice from index 2 to the end;
                      # prints "[2, 3, 4]"
print(nums[:2])       # Get a slice from the start to
                      # index 2 (exclusive); prints "[0, 1]"
print(nums[:])        # Get a slice of the whole list;
                      # prints "[0, 1, 2, 3, 4]"
print(nums[:-1])      # Slice indices can be negative;
                      # prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]    # Assign a new sublist to a slice
print(nums)           # Prints "[0, 1, 8, 9, 4]"

numbers = list(range(5, 10)) # creates a list of integers
                             # starting from 5 to 9 (inclusive)
nums_reverse = numbers[::-1] # iterate through list from the last element
                             # to the first one
print(nums_reverse)          # prints "[9, 8, 7, 6, 5]"
```

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

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.
```

- If you want access to the index of each element within the body of a loop, use the built-in **enumerate** function:

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

- **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:

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

- You can make this code simpler using a **list comprehension**:

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

- List comprehensions can also contain conditions:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)  # Prints "[0, 4, 16]"
```

- You can use **list comprehension** to transform boolean variables in int:

```
bool_list = [True, False, True, True, False]
num_list = [x * 1 for x in bool_list]
print(num_list) # Prints [1, 0, 1, 1, 0]
```

- **Dictionaries:** A dictionary stores (key, value) pairs, similar to a Map in Java. You can use it like this**:**

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new
                                     # dictionary with some data
print(d['cat'])       # Get an entry from
                      # a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary
                      # has a given key; prints "True"
d['fish'] = 'water'   # Set an entry in a dictionary
print(d['fish'])      # Prints "water"
# print(d['monkey'])  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))  # Get an element with a default;
                               # prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default;
                               # prints "water"
del d['fish']         # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key;
```

```
                                # prints "N/A"
```

- Loops through a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

- If you want access to keys and their corresponding values, use the items method:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

- **Sets:** A set is an unordered collection of distinct elements. As a simple example, consider the following: A set is an unordered collection of distinct elements. As a simple example, consider the following:

```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints
"True"
print('fish' in animals)  # prints "False"
animals.add('fish')       # Add an element to a set
print('fish' in animals)  # Prints "True"
print(len(animals))       # Number of elements in a set; prints "3"
animals.add('cat')        # Adding an element that is already in the set
does nothing
print(len(animals))       # Prints "3"
animals.remove('cat')     # Remove an element from a set
print(len(animals))       # Prints "2"

# operations
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
# union
print(A | B) # Prints {1, 2, 3, 4, 5, 6, 7, 8}
print(A.union(B))
# intersection
```

```
print(A & B) # Prints {4, 5}
print(A.intersection(B))
# difference
print(A - B) # Prints {1, 2, 3}
print(A.difference(B))
# symmetric difference - the elements that are in union but not in
intersection
print(A ^ B) # Prints {1, 2, 3, 6, 7, 8}
A.symmetric_difference(B)
```

- Loops through a set: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

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

- **Set comprehensions:** Like lists and dictionaries, we can easily construct sets using set comprehensions:

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

- **Tuples**: A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple
keys
t = (5, 6)        # Create a tuple
print(type(t))    # Prints "<class 'tuple'>"
print(d[t])       # Prints "5"
print(d[(1, 2)])  # Prints "1"
```

Functions

- Python functions are defined using the def keyword. For example:

```
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"
```

- We will often define functions to take optional keyword arguments, like this:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True)  # Prints "HELLO, FRED!"
```

- We will define a function that takes 2 lists and returns True if **the length** of the lists is **different** and the minimum element of the first list **is not** in the second list **or** the minimum element of the second list **is in** the first list, otherwise it returns False.

```
def my_function(first_list, second_list):
 if(len(first_list) == 0 or len(second_list) == 0):
   raise ValueError('Lists must not be empty!')

 min_first = min(first_list)
 min_second = min(second_list)
 if (min_first not in second_list or min_second in first_list)
and len(first_list) != len(second_list):
   return True
 else:
   return False

first_list = [1, 2, 3, 4]
second_list = [0, 1, 3]
```

```
print(my_function(first_list, second_list)) # Prints False
first_list = [-1, 1, 2, 3, 4]
second_list = [0, 1, 3]
print(my_function(first_list, second_list)) # Prints True
first_list = [1, 2, 3, 4]
second_list = [0, 1, 3, 9]
print(my_function(first_list, second_list)) # Prints False
first_list = [1, 2, 3, 4]
second_list = []
print(my_function(first_list, second_list)) # ValueError: Lists
must not be empty!
```

- **Classes**
  - The syntax for defining classes in Python is straightforward:

```
class Greeter:

    # 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')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

Introduction in Numpy

- Numpy:
  - is the core library for scientific computing in Python
  - provides a high-performance multidimensional array object, and tools for working with these arrays
- How to use numpy:

```
import numpy as np
```

- Arrays
  - Initialize an array using Python lists:

```
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

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

  - Initialize an array using Numpy functions:

```
a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

random_array = np.random.random((2,2))  # Create an array filled with
random values drawn from a uniform distribution
```

```
print(random_array)

random_array_gauss = np.random.normal(mu, sigma, (3,2))  # Create an
array filled with values drawn from a gaussian distribution, where mu is
the mean and sigma the standard deviation
print(random_array_gauss)
```

- Array indexing
  - Slicing

```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"

# If you don't want to modify the original array, you can copy the
subarray.
slice_copy = np.copy(array_to_slice[:, 0:3])
slice_copy[0][0] = 100
print(slice_copy[0][0]) # => 100
print(array_to_slice[0][0]) # => 1
```

  - You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array.

```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

```
# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 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.

```
a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

- **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

```
a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.
```

```
print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

- Datatypes

```
x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

- Array math
  - Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Element-wise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Element-wise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

```
# Element-wise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Element-wise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Element-wise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

- Inner product. * is element-wise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

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

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

- Matrix operations

```
# the transpose of a matrix
my_array = np.array([[1, 2, 3], [4, 5, 6]]) # [[1, 2, 3],
 # [4, 5, 6]]
print(my_array.T) # => [[1, 4],
 # [2, 5],
# [3, 6]]
# the inverse of a matrix
my_array = np.array([[1., 2.], [3., 4.]])
print(np.linalg.inv(my_array)) # => [[-2. , 1. ],
 [ 1.5, -0.5]]
```

- Operations along a specific dimension (axis):

```
x = np.array([[1, 2],[3, 4]])
# sum along axis
print(np.sum(x)) # sum all numbers => 10
print(np.sum(x, axis=0)) # sum along columns => [4 6]
print(np.sum(x, axis=1)) # sum along rows => [3 7]

# If axis is a tuple of ints, a sum is performed on all of the axes
specified in the tuple instead of a single axis or all the axes as
before.
print(np.sum(x, axis=(0, 1))) # sum all numbers => 10

# mean along axis
y = np.array([[[1, 2, 3, 4], [5, 6, 7, 8]], [[1, 2, 3, 4], [5, 6, 7, 8]],
[[1, 2, 3,
4], [5, 6, 7, 8]]])
print(y.shape) # => (3, 2, 4)
print(y) # => [[[1 2 3 4]
 # [5 6 7 8]]
 # [[1 2 3 4]
 # [5 6 7 8]]
 # [[1 2 3 4]
 # [5 6 7 8]]]
print(np.mean(y, axis=0)) # => [[1. 2. 3. 4.]
 # [5. 6. 7. 8.]]
print(np.mean(y, axis=1)) # => [[3. 4. 5. 6.]
 # [3. 4. 5. 6.]
# [3. 4. 5. 6.]]

# argmax on rows
z = np.array([[10, 12, 5], [17, 11 ,19]])
print(np.argmax(z, axis=1)) # => [1 2]
```

- 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.

```
# add a vector (v) to every row of a matrix (m)
m = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = m + v
print(y) # => [[ 2 2 4]
 # [ 5 5 7]
 # [ 8 8 10]
 # [11 11 13]]
```

  - Broadcasting two arrays together follows these rules:
    1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
    2. The two arrays are said to be *compatible* in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
    3. The arrays can be broadcast together if they are compatible in all dimensions.
    4. After broadcasting, each array behaves as if it had shape equal to the elemen-twise maximum of shapes of the two input arrays.
    5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

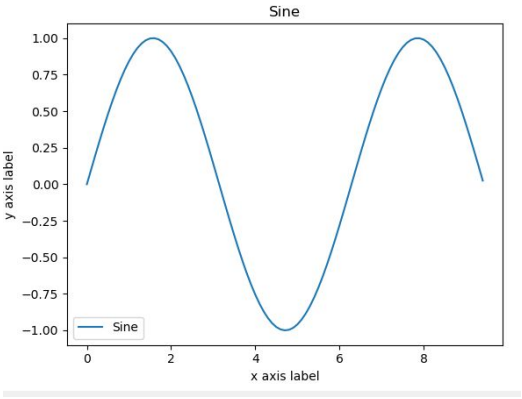
- Matplotlib
  - is a plotting library.
- How to use:

```
import numpy as np
import matplotlib.pyplot as plt
```

- Plotting

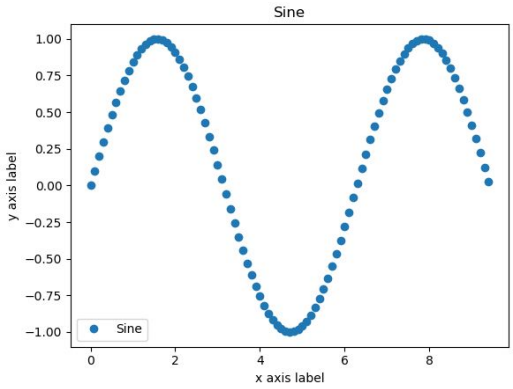
```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()  # You must call plt.show() to make graphics appear.
```



  - To plot the points independently without interpolating the values, we have to set the third parameters of function ‘plot’:

```
plt.plot(x, y, 'o')
```

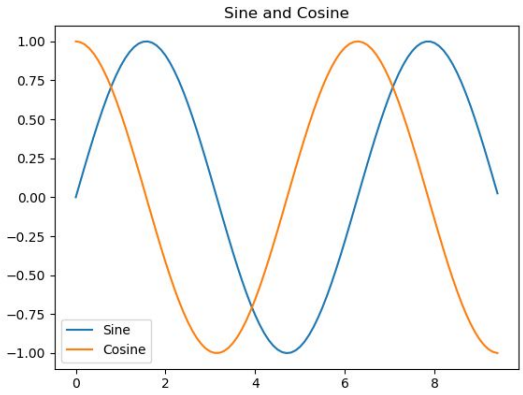


- Plotting multiple lines in the same figure

```
# Compute the x and y coordinates for points on sine and cosine curves
```

```
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
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

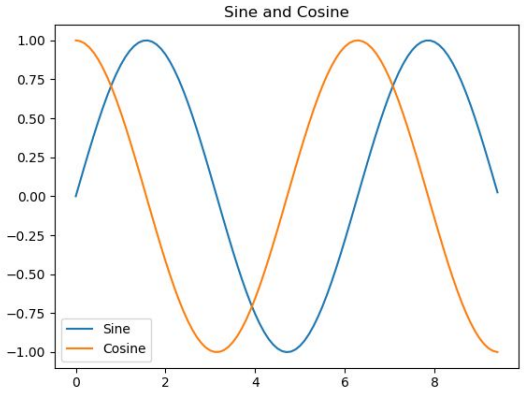
```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')
```

```
# Show the figure.
plt.show()
```



- Display images using matplotlib

```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt

img = imread('assets/cat.jpg', mode='RGB')
img_tinted = img * [1, 0.95, 0.9]

# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)

# Show the tinted image
plt.subplot(1, 2, 2)

# A slight gotcha with imshow is that it might give strange results
# if presented with data that is not uint8. To work around this, we
# explicitly cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```

Exercises

1. Here, we have 8 images, read the images, flatten them, then store them in a numpy array. Before storing the images divide them by 255.
2. Compute the mean of the images.
3. Normalize the images by subtracting the mean from each image.
4. Here, we have the weights and the bias for a perceptron that classifies the images in 4 classes. Use the weights (*use np.load('path') to load the weights*) and the images (after normalization) to compute (y_hat) the predictions of the perceptron.
    a. y_hat = softmax(X * W + b)
    b. X - input data, W - weights, b - bias
    c. $Softmax(z_i) = \frac{e^{z}i}{\sum_{j=1}^{j=K} e^{z}j}$
5. Let the ground-truth labels be y = [0, 0, 1, 1, 2, 2, 3, 3]. Compute the accuracy of the classifier (define a new function *accuracy_score*).
    a. Accuracy = $\frac{\sum_{i=1}^{n} y\_pred_i == y\_true_i}{n}$
6. The labels of the classes are: (Cat - 0, Dog - 1, Frog - 2, Horse - 3), print the label for each prediction (use a dictionary).
7. Display the mean image (use imshow from matplotlib, don't forget to multiply the image by 255).