Classes and Objects

Reading material: [tutorialspoint (http://www.tutorialspoint.com/python/python_classes_objects.htm)](http://www.tutorialspoint.com/python/python_classes_objects.htm)

A `class` is a user-defined variable type that groups functions and data, which can be access with the `.` (dot) operator. A `class` serves as a blueprint for objects.

In [ ]:
```
class Complex :
    '''class representing complex numbers. supports basic complex a
rithmetic'''
    def __init__(self, real, imag=0.0):
        self.real = real    # instance variable
        self.imag = imag    # instance variable

    def add(self, other):
        return Complex(self.real + other.real, self.imag + other.im
ag)

    def sub(self, other):
        return Complex(self.real - other.real, self.imag - other.im
ag)

    def mul(self, other):
        return Complex(self.real*other.real - self.imag*other.imag,
                       self.imag*other.real + self.real*other.imag)

    def display(self):
        print('{:.2f}+{:.2f}i'.format(self.real, self.imag))

c1 = Complex(1.1,-0.3)   #directly create Complex object/instance
c2 = Complex(5.5,2)      #directly create Complex object/instance
c3 = c1.mul(c2)          #indirectly create Complex object/instance
c3.display()
```

```
6.65+0.55i
```

We write the `class Complex` once and create multiple `Complex` objects. In this sense, a `class` is a blueprint.

Notes:

- **Instance variables** are not listed outside the methods. You initialize them inside methods.
- `self` refers to the current object. `self` must be the first parameter methods. You must use `self` to refer to instance variables.
- The constructor or initialization method `__init__` is called when you create a new instance of the class.

Magic methods and overloading operators

Magic methods are special methods that add "magic" to your classes. They are surrounded by double underscores (e.g. `__init__` or `__add__`). We read `__` as "dunder" which is short for "double under". Overview of all of Python's magic methods: [http://minhhh.github.io/posts/a-guide-to-pythons-magic-methods (http://minhhh.github.io/posts/a-guide-to-pythons-magic-methods)](http://minhhh.github.io/posts/a-guide-to-pythons-magic-methods)

The following example implements `__add__` , `__sub__` , and `__mul__` so we can use the arithmetic operators. It also implements `__str__` so we can `print` the object meaningfully.

In [ ]:
```
class Complex:
    '''this is a class demo'''
    def __init__(self, real, imag=0.0):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return Complex(self.real + other.real, self.imag + other.im
ag)

    def __sub__(self, other):
        return Complex(self.real - other.real, self.imag - other.im
ag)

    def __mul__(self, other):
        return Complex(self.real*other.real - self.imag*other.imag,
                       self.imag*other.real + self.real*other.imag)

    def __str__(self):
        return '{:.2f}+{:.2f}i'.format(self.real, self.imag)


c1 = Complex(2.3,10)
c2 = Complex(5.2,-2.9)
print(c1 * c2)
```

```
40.96+45.33i
```

Imagine performing complex arithmetic without a `class` . You would have to carry around pairs of real numbers, and performing arithmetic would be much more error-prone.

Having `class Complex` hold two real numbers and provide methods operating on the data is convenient.

Private Variables

Variables and method beginning with `__` (dunder) dunder are by convention understood to be private. **Private** variables and methods should only be accessed within the `class` .

Example: Polynomial class

The following `class` implements a univariate polynomial real numbers.

In [ ]:
```
class Polynomial :
    '''
        This class implements a univariate polynomial.
        Arithmetic operations such as + - are supported. (* is an e
xercise)
    '''

    def __init__(self, init = 0) :
        self.__poly_coeff = []      # list storing coefficients (pri
vate instance variable)

        # Creates constant polynomial p(x) = init
        if isinstance(init, int) or isinstance(init, float) :
            self.__poly_coeff = [init]

        # Copy the coefficients from given list
        # init[n] = 'n-th coefficient'
        elif isinstance(init, list) :
            self.__poly_coeff = init.copy()

        # Copy the given Polynomial instance
        elif isinstance(init, Polynomial) :
            for n in range(init.degree()+1) :
                self.set_coeff(n, init.get_coeff(n))


    # Returns the degree of Polynomial
    def degree(self) :
        return max([0]+[n for n,c in enumerate(self.__poly_coeff) i
f c != 0.0])

    # Sets the coefficient of given degree term
    def set_coeff(self, deg, new_coeff) :
        if len(self.__poly_coeff) <= deg :
            self.__poly_coeff += [0.0 for _ in range(deg + 1 - len
(self.__poly_coeff))]
        self.__poly_coeff[deg] = new_coeff

    # Returns the coefficient of given degree term
    def get_coeff(self, deg) :
        return 0 if self.degree() < deg else self.__poly_coeff[deg]


    # -self
    def __neg__(self) :
        result = Polynomial()
        for n in range(self.degree() + 1) :
            result.set_coeff(n, -self.__poly_coeff[n])
        return result

    # self + poly2
    def __add__(self, poly2) :
        result = Polynomial(self)
        result += poly2
        return result

    # self - poly2
    def __sub__(self, poly2) :
        result = Polynomial(self)
```

```
        result -= poly2
        return result

    # Overload += (self += poly2)
    def __iadd__(self, poly2) :
        poly2 = Polynomial(poly2)
        for n in range(max(self.degree(),poly2.degree()) + 1) :
            self.set_coeff(n, self.get_coeff(n) + poly2.get_coeff
(n))
        return self

    # Overload -=
    def __isub__(self, poly2) :
        return (self.__iadd__(-poly2))

    # Operators with Polynomial instance on the right
    __radd__ = __add__      # other + self

    # poly2 - self
    def __rsub__(self, poly2) :
        return -Polynomial(self) + poly

    # Evaluation of polynomial at x : p(x)
    def __call__(self,x):
        return sum([self.get_coeff(n)*(x**n) for n in range(self.de
gree() + 1)])

    #returns algebraic formula of polynomial as a string
    def __str__(self):
        coeff_list = [self.get_coeff(n) for n in range(self.degree
() + 1) ]

        expr = ''
        # Generate polynomial expression
        for n in range(self.degree(), 0, -1) :
            if coeff_list[n] == 0 :
                pass
            elif coeff_list[n] == 1 :
                expr += '+ x^{0} '.format(n)
            elif coeff_list[n] == -1 :
                expr += '- x^{0} '.format(n)
            elif coeff_list[n] < 0 :
                expr += '- {0:.2f}x^{1} '.format(- coeff_list[n],
n)
                pass
            else :
                expr += '+ {0:.2f}x^{1} '.format(coeff_list[n], n)

        if coeff_list[0] < 0 :
            expr += '- ' + '{:.2f}'.format(- coeff_list[0])
        elif coeff_list[0] > 0 :
            expr += '+ ' + '{:.2f}'.format(coeff_list[0])

        if expr[:2] == "+ ":
            return expr[2:]
        elif expr[:2] == "- ":
            return "-" + expr[2:]
```

```
# Test code
p1 = Polynomial()
p1.set_coeff(0, 1.2)
p1.set_coeff(3, 2.2)
p1.set_coeff(7, -9.0)
p1.set_coeff(7, 0.0)
# # degree of polynomial is now 3
print(p1)
print(-p1)  #call negation operator

print(p1.degree())

p2 = Polynomial([1, 1.3])
# print(p2.get_coeff(0))
# print(p2.get_coeff(1))
# print(p2.get_coeff(2))  #should be 0
# print(p2.get_coeff(3))  #should be 0
# print(p2.get_coeff(4))  #should be 0
# print(p2.get_coeff(5))  #should be 0

print(p2 + p1)
```

```
2.20x^3 + 1.20
-2.20x^3 - 1.20
3
2.20x^3 + 1.30x^1 + 2.20
```

Access the **docstring** of a class by accessing the `__doc__` attribute of the `class`. By convention, the **docstring** provides a brief description of the `class`.

In [ ]:
```
print(Polynomial.__doc__)
```

```
        This class implements a univariate polynomial.
        Arithmetic operations such as + - are supported. (* is an e
xercise)
```

Use `dir` or access the `__dict__` attribute to see the functionality a `class` provides.

In [ ]:
```
print(Polynomial.__dict__)
print(dir(Polynomial))
```

```
{'__module__': '__main__', '__doc__': '\n        This class impleme
nts a univariate polynomial.\n        Arithmetic operations such as
+ - are supported. (* is an exercise)\n    ', '__init__': <function
Polynomial.__init__ at 0x7fbd002c9ca0>, 'degree': <function Polynom
ial.degree at 0x7fbce0f33160>, 'set_coeff': <function Polynomial.se
t_coeff at 0x7fbce0f33940>, 'get_coeff': <function Polynomial.get_c
oeff at 0x7fbce0f339d0>, '__neg__': <function Polynomial.__neg__ at
0x7fbce0f33a60>, '__add__': <function Polynomial.__add__ at 0x7fbce
0f33af0>, '__sub__': <function Polynomial.__sub__ at 0x7fbce0f33b80
>, '__iadd__': <function Polynomial.__iadd__ at 0x7fbce0f33c10>, '_
_isub__': <function Polynomial.__isub__ at 0x7fbce0f33ca0>, '__radd
__': <function Polynomial.__add__ at 0x7fbce0f33af0>, '__rsub__': <
function Polynomial.__rsub__ at 0x7fbce0f33d30>, '__call__': <funct
ion Polynomial.__call__ at 0x7fbce0f33dc0>, '__str__': <function Po
lynomial.__str__ at 0x7fbce0f33e50>, '__dict__': <attribute '__dict
__' of 'Polynomial' objects>, '__weakref__': <attribute '__weakref_
_' of 'Polynomial' objects>}
['__add__', '__call__', '__class__', '__delattr__', '__dict__', '__
dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribut
e__', '__gt__', '__hash__', '__iadd__', '__init__', '__init_subclas
s__', '__isub__', '__le__', '__lt__', '__module__', '__ne__', '__ne
g__', '__new__', '__radd__', '__reduce__', '__reduce_ex__', '__repr
__', '__rsub__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__weakref__', 'degree', 'get_coeff', 'set_coef
f']
```

Duck typing

The following function `sum_all` sums numbers of a list.

In [ ]:
```
def sum_all(lst):
    ret = None
    for elem in lst:
        if ret is None:
            ret = elem
        else:
            ret = ret + elem
    return ret


print(sum_all([1,2,3]))
```

```
6
```

But wait, `lst` need not be a list and the elements of `lst` need not be numbers. "Sums numbers of a list" does not fully describe the capability of `sum_all`.

Really, you can use `sum_all(lst)` if you can iterate through the elements of `lst` with a for loop (i.e., `lst` is an "iterable" as we define later) and you can use `+` with the elements of `lst` (i.e., the elements of `lst` are objects with the `__add__` method).

In [ ]:
```
lst1 = ['Python was named after ', 'the British TV series "Monty Python." ']
lst2 = ['The Dutch creator of Python, Guido van Rossum, seems to have a British sense of humor.']

# print(sum_all((lst1,lst2)))  # list of strings

c1 = Complex(1,2)
c2 = Complex(3,4)
c3 = Complex(-5,0)

print(sum_all({c1,c2,c3}))   # tuple of Complex
```

```
-1.00+6.00i
```

In the context of logic (논리학), the following saying describes a form of abductive reasoning:

> "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

In the context of programming, **duck typing** refers to the practice of caring about what the object can do, rather than what it is.

Inheritance

Because Python is not a strongly-typed language, inheritance is not used to provide type-safety. Rather, inheritance is used to re-use certain features of another class and to build on top of it.

In [ ]:
```
class Matrix:
    def __init__(self, dim, arr):
        self.h = dim[0]  # height
        self.w = dim[1]  # width
        self.elem_list = arr[:] # make copy

    def __add__(self, RHS):
        return Matrix((self.h,self.w), [self.elem_list[i] + RHS.ele
m_list[i] for i in range(self.h*self.w)])

    def __mul__(self, RHS):
        e_list = [0] * self.h * RHS.w
        for i in range(self.h):
            for k in range(self.w):
                for j in range(RHS.w):
                    e_list[i*RHS.w+j] += self.elem_list[i*self.w+k]
* RHS.elem_list[k*RHS.w+j]
        return Matrix((self.h,RHS.w), e_list)

    def __str__(self):
        s = "["
        for i in range(self.h):
            for j in range(self.w):
                s += str(self.elem_list[i*self.w+j]) + " "
            s += "\n"
        s = s[:-2] + "]"
        return s

class SquareMatrix(Matrix):
    def det(self):
        #some formula for computing the determinant
        pass
    def inverse(self):
        #some formula for computing the inverse
        pass

m1 = Matrix((3,2),[1,6,2,6,3,5])
m2 = Matrix((2,3),[1,2,2,1,1,2])
print(m1)
print(m2)
print(str(m1*m2))
```

```
[1 6
2 6
3 5]
[1 2 2
1 1 2]
[7 8 14
8 10 16
8 11 16]
```

For loop and iterables

Container objects can be looped over using a for loop, but how?

In [ ]:
```
for element in [1, 2, 3]:
    print(element)

for element in (1, 2, 3):
    print(element)

for element in {1, 2, 3}:
    print(element)

for key in {'one':1, 'two':2}:
    print(key)  # iterate over keys but not values

for char in "ABC":
    print(char)
```

```
1
2
3
1
2
3
1
2
3
one
two
A
B
C
```

Also, what is `range(n)` ?

In [ ]:
```
for ind in range(5):
    print(ind)
print(range(5))
print(type(range(5)))
print(dir(range(5)))
```

```
0
1
2
3
4
range(0, 5)
<class 'range'>
['__bool__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '_
_getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__
reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__setattr_
_', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index',
'start', 'step', 'stop']
```

Generally, you can use for loops with **iterables**, which are objects that provide an **iterator** through the method `__iter()__` .

In [ ]:
```
print(range(5).__iter__())
```

```
<range_iterator object at 0x7fbcd07e5fc0>
```

An **iterator** provides access to the elements with the method `__next__()` .

The following loop manually iterates through `range(5)` , an iterable.

In [ ]:
```
itr = range(5).__iter__()
while True:
    print(itr.__next__())
```

```
0
1
2
3
4
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
/Users/movie/workspace/assignment-01-movie112/01_3.ipynb 셀 30 line 3
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X41sZmlsZQ%3D%3D?line=0'>1</a> itr = range(5).__iter__()
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X41sZmlsZQ%3D%3D?line=1'>2</a> while True:
----> <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X41sZmlsZQ%3D%3D?line=2'>3</a>     print(itr.__next__())

StopIteration:
```

Usually, there is no need to directly call `__iter__` ; it is better to use a `for` loop. The example above is for learning purposes.

The end of the iterator is signaled using an exception.

In [ ]:
```
itr = range(5).__iter__()
while True:
    try:
        print(itr.__next__())
    except StopIteration:
        break
```

```
0
1
2
3
4
```

We won't spend time on exceptions and exception handling with try-except in this class, so don't worry if the above example doesn't make sense.

In [ ]:
```
itr = iter("Hello")
while True:
    try:
        print(next(itr))
    except StopIteration:
        break
```

```
H
e
l
l
o
```

Custom iterable example.

In [ ]:
```
class Sentence:
    def __init__(self, sentence):
        self.sentence = sentence

    def __iter__(self):
        return SentenceIter(self.sentence)

class SentenceIter:
    def __init__(self, sentence):
        self.words = sentence.split()  # returns a list of words se
parated by spaces
        self.index = 0

    def __next__(self):
        if self.index >= len(self.words):
            raise StopIteration  # StopIteration exception signals
end of iterator
        index = self.index
        self.index += 1
        return self.words[index]



my_sentence = Sentence('This is a test')
# for word in my_sentence:
#     print(word)

stIter = iter(my_sentence)

print(next(stIter))
print(next(stIter))
print(next(stIter))
print(next(stIter))
print(next(stIter))  # out of elements
```

```
This
is
a
test

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
/Users/movie/workspace/assignment-01-movie112/01_3.ipynb 셀 36 line 3
     <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X50sZmlsZQ%3D%3D?line=29'>30</a> print(next(stIter))
     <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X50sZmlsZQ%3D%3D?line=30'>31</a> print(next(stIter))
---> <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X50sZmlsZQ%3D%3D?line=31'>32</a> print(next(stIter))  # out of elements

/Users/movie/workspace/assignment-01-movie112/01_3.ipynb 셀 36 line 1
     <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X50sZmlsZQ%3D%3D?line=12'>13</a> def __next__(self):
     <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X50sZmlsZQ%3D%3D?line=13'>14</a>     if self.index >= len(self.words):
---> <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X50sZmlsZQ%3D%3D?line=14'>15</a>         raise StopIteration  # StopIteration exception signals end of iterator
     <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X50sZmlsZQ%3D%3D?line=15'>16</a>     index = self.index
     <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#X50sZmlsZQ%3D%3D?line=16'>17</a>     self.index += 1

StopIteration: 
```

Iterators are do not have to end. The following is an example with the Fibonacci sequence.

In [ ]:
```
class Fibo:
    def __init__(self):
        pass

    def __iter__(self_):
        return FiboIter()

class FiboIter:
    def __init__(self):
        self.index = -1

    def __next__(self):
        self.index += 1
        if self.index == 0:
            return 0
        elif self.index == 1:
            self.prev, self.curr = 0, 1
            return 1
        else:
            nxt = self.prev + self.curr
            self.prev, self.curr = self.curr, nxt
            return self.curr

for num in Fibo():
    if num > 100:
        break
    print(num)
```

```
0
1
1
2
3
5
8
13
21
34
55
89
```

It is actually common practice to have one single class represent both the iterable and its iterator.

The first of the following two examples was inspired and copied from [Corey Schafer (https://www.youtube.com/channel/UCCezIgC97PvUuR4_gbFUs5g)](https://www.youtube.com/channel/UCCezIgC97PvUuR4_gbFUs5g)'s Youtube channel.

In [ ]:
```
class Sentence:
    def __init__(self, sentence):
        self.sentence = sentence
        self.words = sentence.split()
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.words):
            raise StopIteration  # StopIteration exception signals
end of iterator
        index = self.index
        self.index += 1
        return self.words[index]


my_sentence = Sentence('This is a test')
# for word in my_sentence:
#     print(word)


print(next(my_sentence))
print(next(my_sentence))
print(next(my_sentence))
print(next(my_sentence))
# print(next(my_sentence))  # out of elements



class Fibo:
    def __init__(self):
        self.index = -1

    def __iter__(self):
        return self

    def __next__(self):
        self.index += 1
        if self.index == 0:
            return 0
        elif self.index == 1:
            self.prev, self.curr = 0, 1
            return 1
        else:
            next = self.prev + self.curr
            self.prev, self.curr = self.curr, next
            return self.curr


for num in Fibo():
    if num > 100:
        break
    print(num)
```

```
This
is
a
test
0
1
1
2
3
5
8
13
21
34
55
89
```

Context manager and with

A **context manager** is an object that defines the runtime context to be established when executing a `with` statement. It provides `__enter__` and `__exit__` methods. You use context manager with `with` statements.

In [ ]:
```python
class c_manager :
    def __init__(self):
        print("Manager constructred")
    def __enter__(self):
        print("Context begins")
        print("------------------------------------------------")
    def __exit__(self, exc_type, value, traceback):
        print("------------------------------------------------")
        print("Context ends")

with c_manager():
    print("hello")
    print("Let's do some stuff here.")
```

```
Manager constructred
Context begins
------------------------------------------------
hello
Let's do some stuff here.
------------------------------------------------
Context ends
```

Example: Using a context manager to measure runtime of a code block

In [ ]:
```
from time import time

class Timer :
    def __init__(self, description):
        self.description = description
    def __enter__(self):
        self.start = time()
    def __exit__(self, exc_type, value, traceback):
        self.end = time()
        print(f"{self.description}: {self.end - self.start:.2f}s")


with Timer("List Comprehension Example"):
    print("We do stuff here")
    s = [x for x in range(10000000)]
    print("We did stuff here")
```

```
We do stuff here
We did stuff here
List Comprehension Example: 0.40s
```

NumPy

NumPy is the numerical computation library of Python. When performing numerical computation, `numpy` arrays are far superior than raw Python `list` s.

numpy arrays

`numpy.array(...)` creates a `numpy` array from a Python list.

In [ ]:
```
import numpy as np

a = np.array([1,2,3], dtype='int32')  #dtype specifies data type

b = np.array([1,2,3], dtype='float64')

c = np.array([[9.0,8.0,7.0],[6.0,5.0,4.0]])
```

In [ ]:
```
# dimension of np array
# print(a.ndim)

# shape of np array
# print(a.shape)

# number of elements in np array
# print(c.size)

# type of elements
# print(c.dtype)

# size of elements in bytes
# print(c.itemsize)

# total size of np array in bytes
# print(c.nbytes)
```

In this lecture, an "array" can have 1, 2, 3, or more dimensions, while a "matrix" specifically is 2-dimensional.

Creating basic arrays

In [ ]:
```
# A = np.zeros((2,3))  # all 0 array
# A = np.ones((4,2,2))  # all 1 array

# b = np.ones(5)     # ndim = 1
# b = np.ones((5,))  # same 1D array
# print(b)

# # np.random uses different notation for specifying dimensions
# A = np.random.rand(4,2)                # random numbers between 0 
and 1
# A = np.random.randn(5)                  # random standard normal
# A = np.random.randint(-4,8, size=(3,3)) # random integers

# A = np.identity(5) # identity matrix

# np.arange(...) returns numpy array; range(...) returns iterable
# arange is short for array-range; unrelated to verb arrange
x = np.arange(1,8,1)
```

Reorganizing arrays

In [ ]:
```
A = np.array([[1,2,3,4],[5,6,7,8]])
# print(A.reshape((4,2)))
# print(A.reshape((4,-1))) # as many columns as needed to fit eleme
nts

v1 = np.array([1,2,3,4])
v2 = np.array([5,6,7,8])
print(np.vstack([v1,v2])) # vertical stack

h1 = np.ones((2,4))
h2 = np.zeros((2,2))
print(np.hstack((h1,h2))) # horizontal stack
```

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

Vectorizing

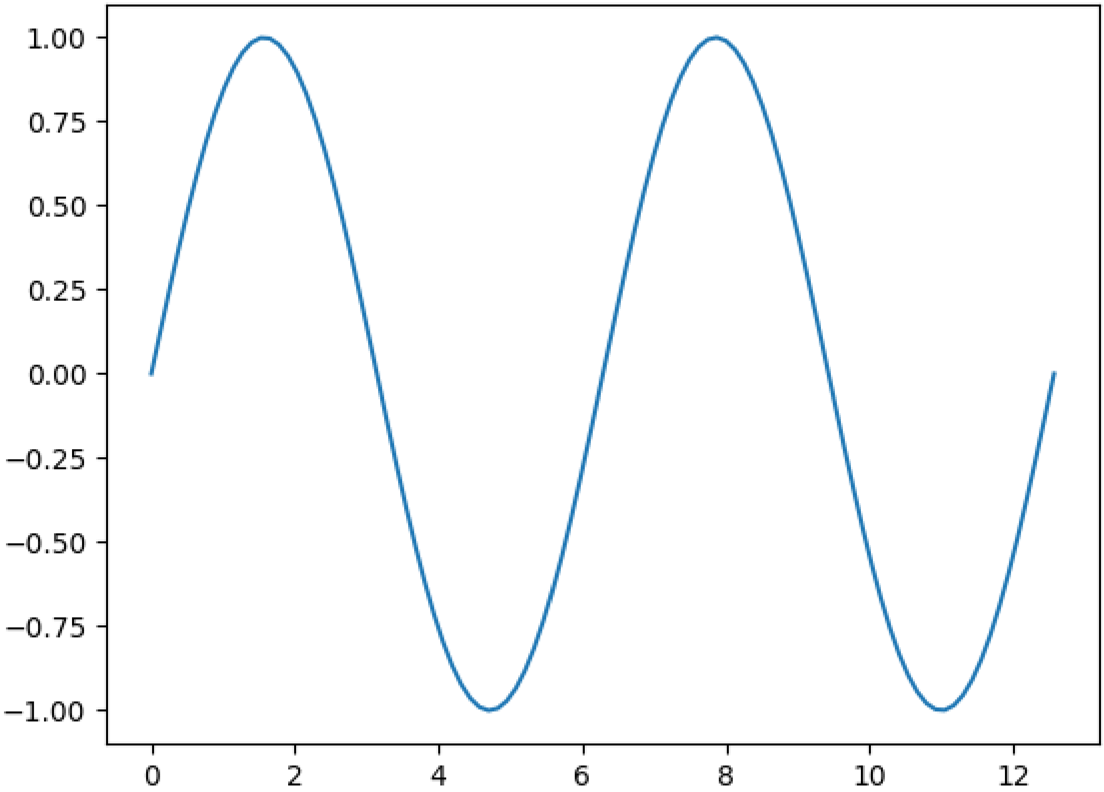
The following is a reasonably Pythonic way of plotting the $\sin(x)$ without using `numpy`. (But this is bad.)

In [ ]:
```
import math
import matplotlib.pyplot as plt

x = [i*(4*math.pi/(N-1)) for i in range(100)]
y = [math.sin(x_i) for x_i in x]
plt.plot(x, y)
plt.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/Users/movie/workspace/assignment-01-movie112/01_3.ipynb 셀 55 line 4
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=0'>1</a> import math
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=1'>2</a> import matplotlib.pyplot as plt
----> <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=3'>4</a> x = [i*(4*math.pi/(N-1)) for i in range(100)]
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=4'>5</a> y = [math.sin(x_i) for x_i in x]
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=5'>6</a> plt.plot(x, y)

/Users/movie/workspace/assignment-01-movie112/01_3.ipynb 셀 55 line 4
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=0'>1</a> import math
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=1'>2</a> import matplotlib.pyplot as plt
----> <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=3'>4</a> x = [i*(4*math.pi/(N-1)) for i in range(100)]
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=4'>5</a> y = [math.sin(x_i) for x_i in x]
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignment-01-movie112/01_3.ipynb#Y105sZmlsZQ%3D%3D?line=5'>6</a> plt.plot(x, y)

NameError: name 'N' is not defined
```

It is better to use `numpy` and avoid the use of loops or list comprehensions. With `numpy`, **vectorize** operations as much as possible.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 4*np.pi, 100)
plt.plot(x, np.sin(x))  # math.sin(x) doesn't support vector eval
plt.show()
```



If you are iterating through a `numpy` array (with a for loop or list comprehension) there is a good chance you are doing something wrong. Vectorized code is shorter, faster, and usually more readible, so always look for ways to vectorize.

(The principle of vectorization applies to `numpy` arrays, but the name **arrayrize** doesn't roll off one's toungue.)

Broadcasting

Arithmetic operations on arrays of same size are performed elementwise.

In [ ]:
```
x = np.arange(7)
print(x * x)  #not the inner product
```

```
[ 0  1  4  9 16 25 36]
```

When we have arrays of different sizes, the smaller array is **broadcast** across the larger array and then the arithmetic operations are carried out. (In some sense, broadcast generalizes the outer product of vectors.)

In [ ]:
```
x, y = np.arange(5), np.arange(6)
# print(x + y)  # fail! dimension mismatch
# print(x.reshape(-1,1) + y.reshape(1,-1))  # broadcasting

# print(x.reshape(-1,1) * y.reshape(1,-1))  # outer product with br
oadcasting
# print(np.outer(x,y))                      # outer product with ou
ter
```

Scalar-array operations is the most common instance of broadcasting.

In [ ]:
```
# print(5.5 + np.arange(5))

print(3.5 * np.ones((3,3)))
```

```
[[3.5 3.5 3.5]
 [3.5 3.5 3.5]
 [3.5 3.5 3.5]]
```

Indexing

You can access elements of `numpy` arrays with **direct indexing** and **slicing**, similar to how you access elements of lists. You also have **advanced indexing** and **Boolean masks**.

In [ ]:
```
A = np.random.randn(10,10)
print(A[4,5])     # direct indexing
print(A[1:8:2,5:7]) # slicing
```

```
0.7226067575226982
[[-0.56641435  0.384961  ]
 [ 1.29098696 -0.40092616]
 [ 0.8158452   0.25912553]
 [-1.42081449  1.03335441]]
```

Be careful when copying numpy arrays!!!

For the sake of efficiency, `numpy` operations often avoid copying data and rather provides different **views** of the underlying data.

In [ ]:
```
x = np.arange(5)
y = x[:]  # creates a different view, not a copy of x
for i in range(5):
    y[i] = 0

# z = x[2:4]  # creates a different view, not a copy of x
# z[:] = 7 # write broadcasted
# print(x)
```

This behavior contrasts with that of lists.

In [ ]:
```
x = [0, 1, 2, 3, 4]
y = x[:]  # creates a copy of x
for i in range(5):
    y[i] = 0
print(x)

# x[:] = 7  # no broadcasting for lists
```

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

If you really need to copy the data, be explicit by using `copy()` .

In [ ]:
```
x = np.arange(5)
y = x.copy()  # creates a copy of x
y[:] = 0
print(x)
```

```
[0 1 2 3 4]
```

With **advanced indexing**, you pass in a list or `numpy` array of indices to access elements. (Advanced indexing doesn't work on Python lists.)

In [ ]:
```
# x = np.arange(5)
# print(x)
# print(x[[1,4]])
# print(x[1,4])     # doesn't work. Why?

A = np.arange(24).reshape((4,-1))
# print(A)
perm = np.random.permutation(np.arange(A.shape[1]))
print(perm)
print(A[:,perm])  #randomly permute columns of x
```

```
[4 2 0 3 1 5]
[[ 4  2  0  3  1  5]
 [10  8  6  9  7 11]
 [16 14 12 15 13 17]
 [22 20 18 21 19 23]]
```

With **Boolean masks**, you pass in a list or `numpy` array of booleans of the same shape to access elements. (Boolean masks don't work on Python lists.)

In [ ]:
```
np.set_printoptions(formatter={'float': lambda x: "{0:0.2f}".format(x)})
np.random.seed(1)

x = np.random.randn(5)
print(x)

mask = (x >= 0)
print(mask)
print(x[mask])

x[mask] = 0
print(x)

# x[x>=0] = 0
# print(x)
```

```
[1.62 -0.61 -0.53 -1.07 0.87]
[ True False False False  True]
[1.62 0.87]
[0.00 -0.61 -0.53 -1.07 0.00]
```

We cannot directly use logical operators on Boolean masks. You must explicitly use NumPy's versions of the boolean operators.

In [ ]:
```
np.random.seed(1)
x = 5*np.random.randn(5)

mask1 = x <= 6
mask2 = x >= 3
print(mask1)
print(mask2)
print(np.logical_and(mask1, mask2))
print(np.logical_xor(mask1, mask2))
```

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

Linear Algebra

Perform matrix multiplication with `@` rather than `*`.

In [ ]:
```
import numpy as np

n = 7
A = np.ones((n,n))
b = np.arange(n)
# print(A*b)  # broadcasted product. *Not* matrix-vector product.
# print(A@b)  # matrix-vector product

# print(b*b)  # element-wise product
print(b@b)  # dot product
```

```
91
```

Transpose a matrix with `.transpose()` or `.T`.

In [ ]:
```
A = np.ones((4,7))
b = np.random.randn(7)

print(A.T@A@b)
```

```
[-7.39 -7.39 -7.39 -7.39 -7.39 -7.39 -7.39]
```

The `np.linalg` module provides linear algebraic functions.

In [ ]:
```
A = np.identity(3)
print(np.linalg.det(A))      # determinant
print(np.linalg.eigvals(A))  # eigenvalues
```

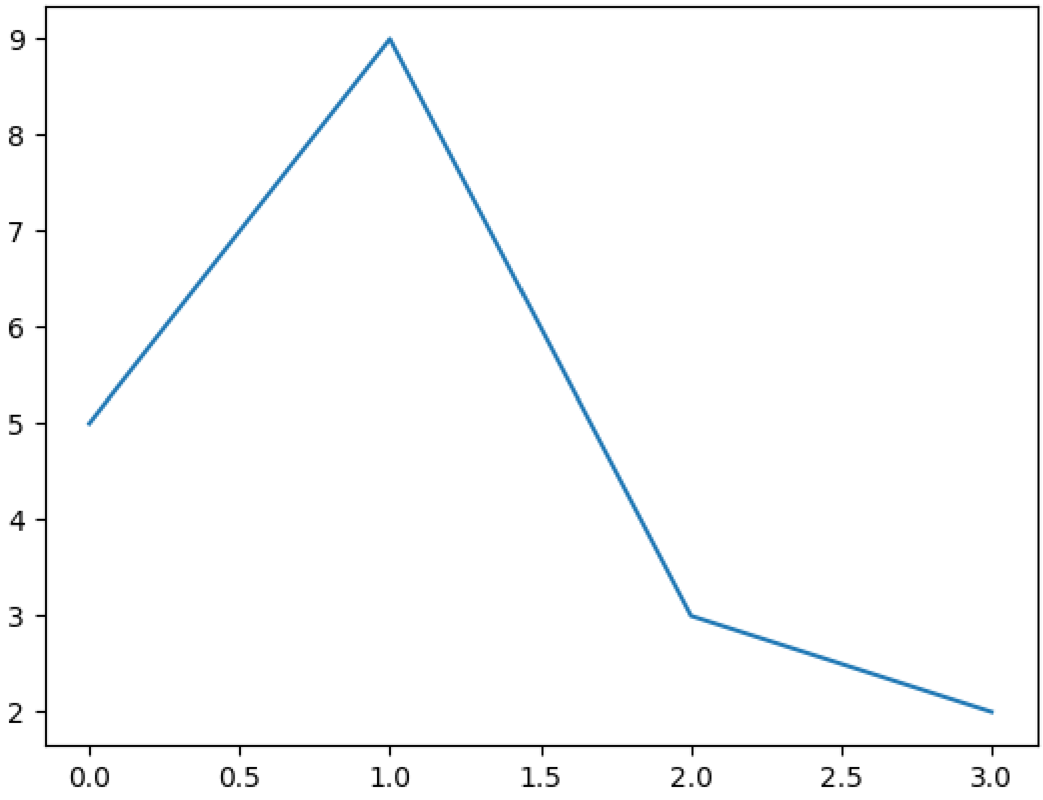
```
1.0
[1.00 1.00 1.00]
```

Matplotlib and pyplot

`matplotlib` and `pyplot` plot data contained in raw Python lists and `numpy` arrays. In its most basic form, a plot is a line sequentially connecting points in the 2D plane.

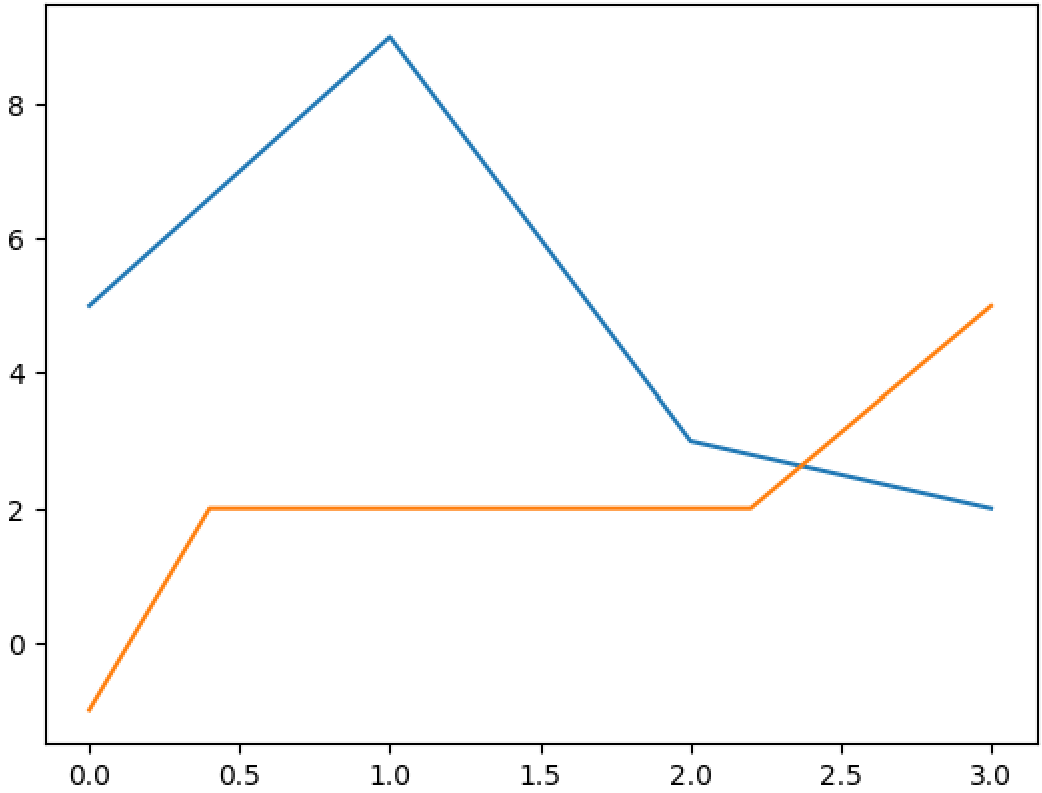
To display plots on Jupyter notebooks, use the "magic" `%matplotlib inline`

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot([0,1,2,3],[5,9,3,2])
plt.show()
```



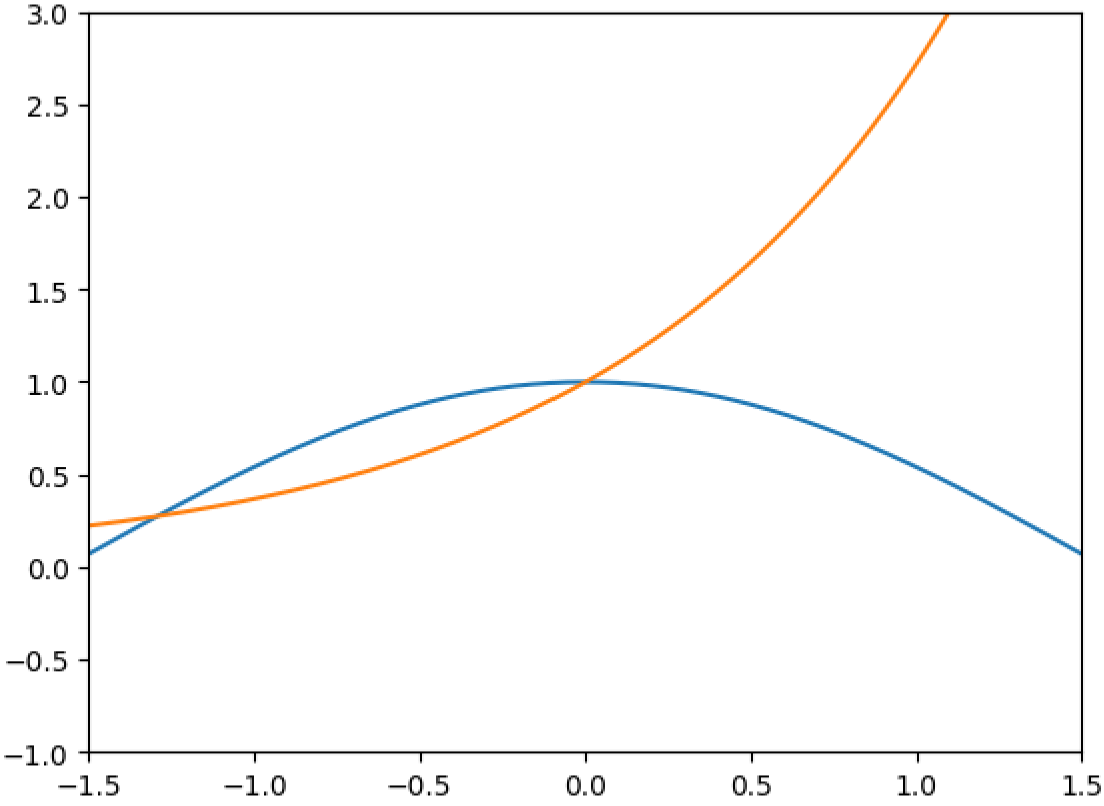
You can have multiple cuves on the same plot

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot([0,1,2,3],[5,9,3,2])
plt.plot([0,0.4,2.2,3],[-1,2,2,5])
plt.show()
```



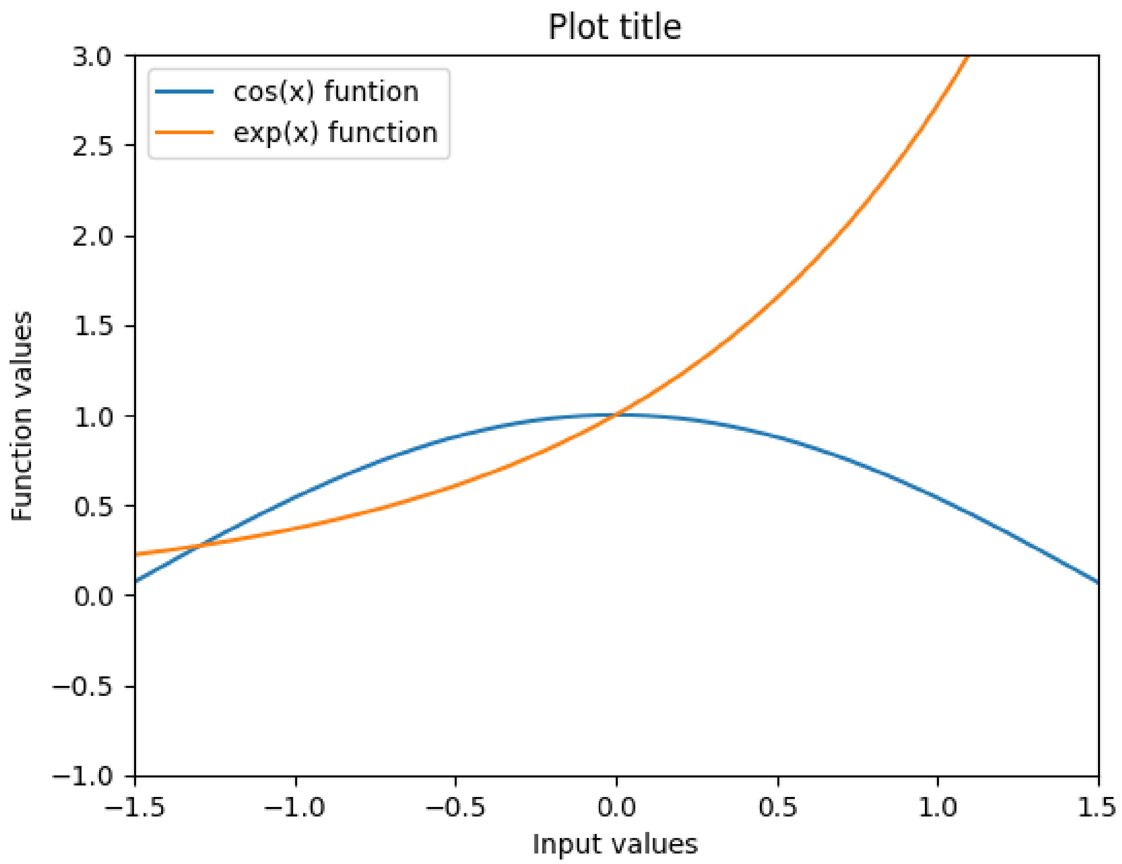
Manually choose the axis limits with `axis([xmin,xmax,ymin,ymax])`

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

xx = np.linspace(-2,2,1024)
plt.plot(xx,np.cos(xx))
plt.plot(xx,np.exp(xx))

plt.axis([-1.5,1.5,-1,3])

plt.show()
```



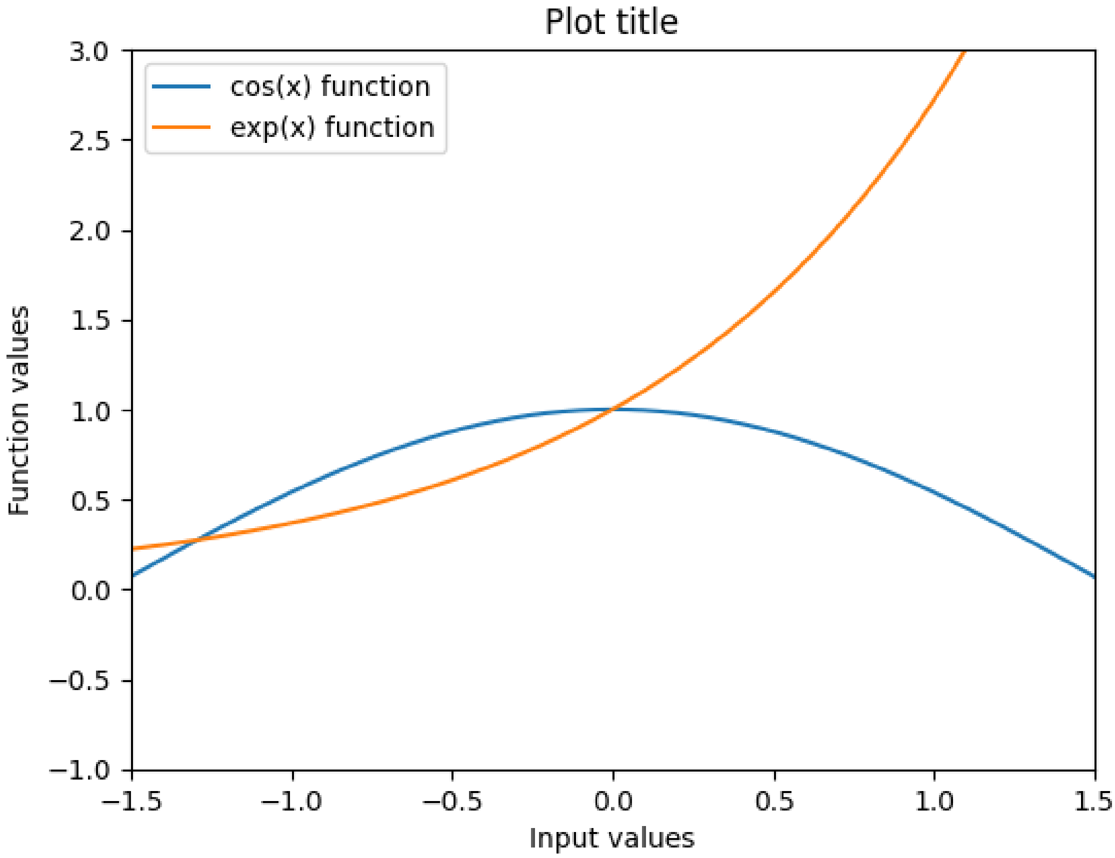
Label your plots as follows.

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

xx = np.linspace(-2,2,1024)
plt.plot(xx,np.cos(xx))
plt.plot(xx,np.exp(xx))

plt.axis([-1.5,1.5,-1,3])

plt.xlabel("Input values")
plt.ylabel("Function values")
plt.title("Plot title")

plt.legend(["cos(x) funtion", "exp(x) function"])

plt.show()
```



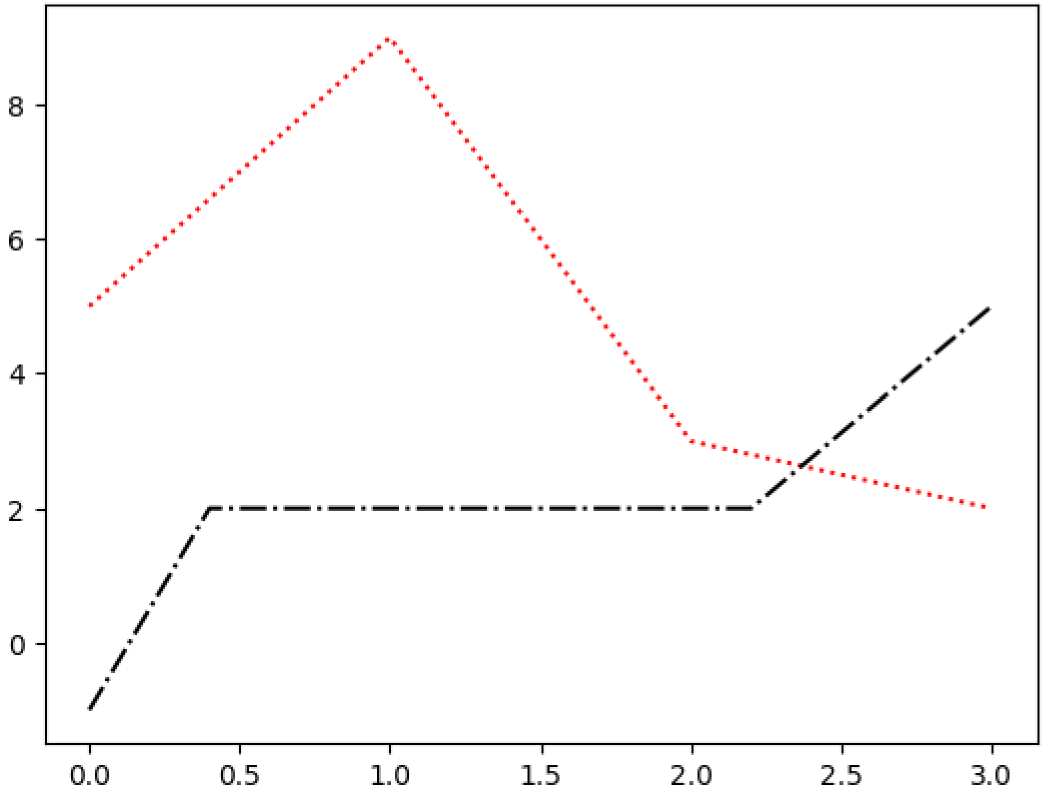
It is better to specify the legends via keyword arguments.

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

xx = np.linspace(-2,2,1024)
plt.plot(xx,np.cos(xx), label="cos(x) function")
plt.plot(xx,np.exp(xx), label="exp(x) function")

plt.axis([-1.5,1.5,-1,3])

plt.xlabel("Input values")
plt.ylabel("Function values")
plt.title("Plot title")

plt.legend()

plt.show()
```



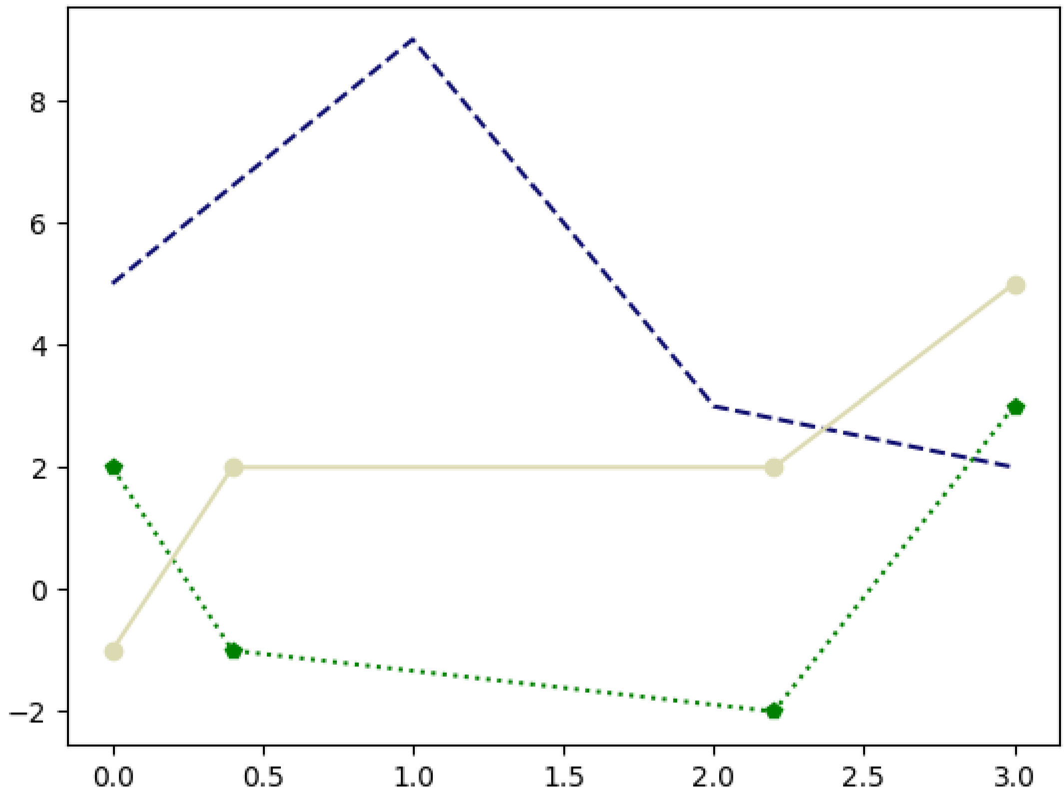
You can specify line styles with ["format strings" (https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.plot.html)](https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.plot.html).

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# plt.plot([0,1,2,3],[5,9,3,2], 'r+')   #red, no line, cross marker
# plt.plot([0,0.4,2.2,3],[-1,2,2,5], 'b--o')  #blue, -- line, circl
e marker


plt.plot([0,1,2,3],[5,9,3,2],'r:')
plt.plot([0,0.4,2.2,3],[-1,2,2,5],'k-.')

plt.show()
```



While format strings are concise and "standard", I don't think they are very readable. I prefer using keyword arguments. You can specify colors with their names or their RGB hex code.

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot([0,1,2,3],[5,9,3,2], color='#0F0F70', linestyle='--')
plt.plot([0,0.4,2.2,3],[2,-1,-2,3], color="green", linestyle=':', m
arker='p')
plt.plot([0,0.4,2.2,3],[-1,2,2,5], color="#dcdab2", linestyle='-',
marker='o')

plt.show()
```

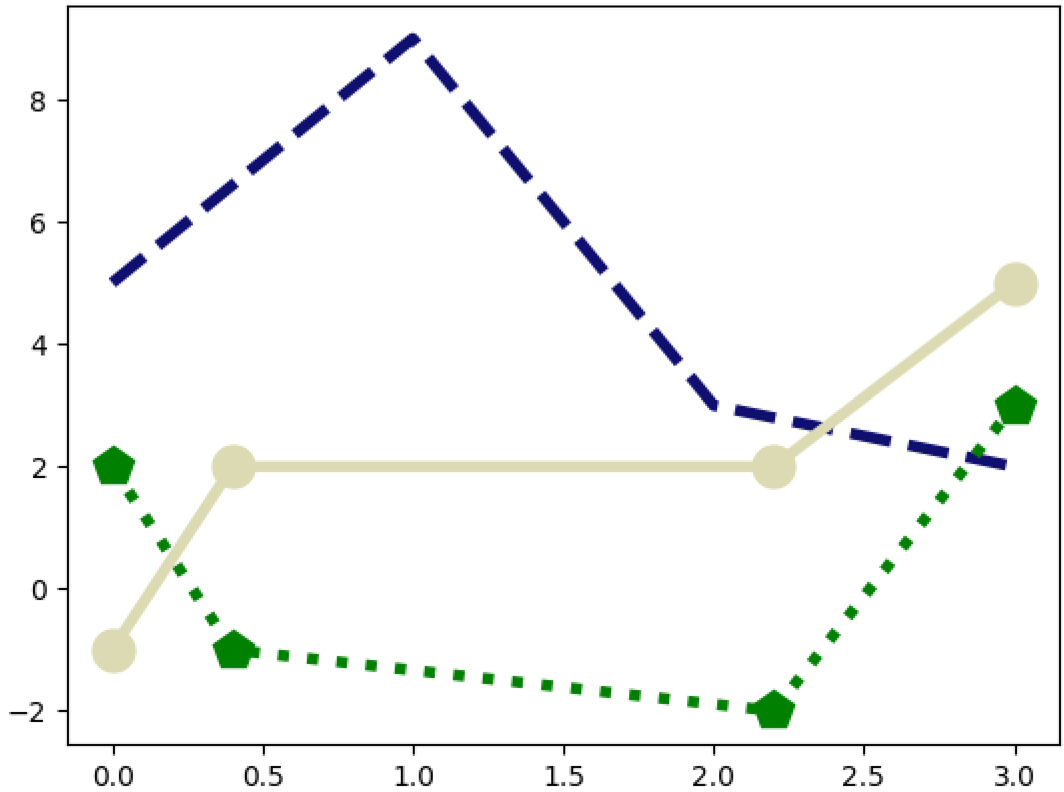


You can specify other plot properties with keyword arguments.

In [ ]:
```
!pip install matplotlib
```

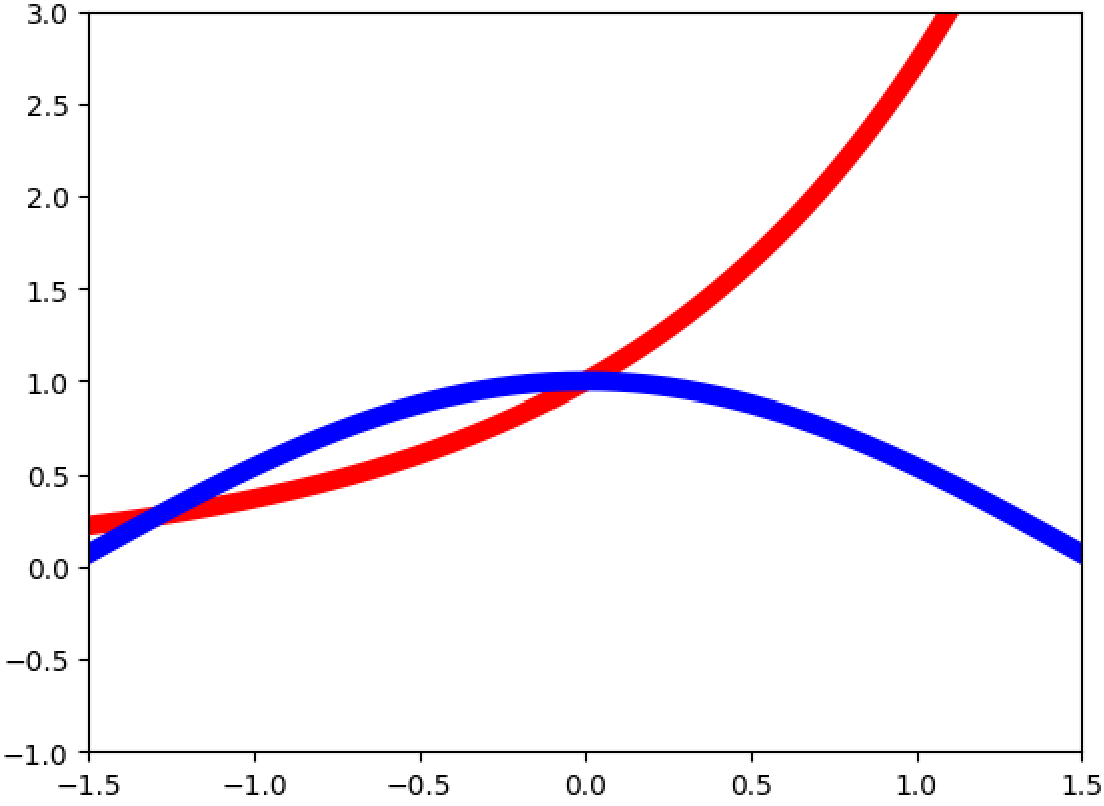
```
Requirement already satisfied: matplotlib in /opt/anaconda3/envs/ml
_homework/lib/python3.8/site-packages (3.7.5)
Requirement already satisfied: contourpy>=1.0.1 in /opt/anaconda3/e
nvs/ml_homework/lib/python3.8/site-packages (from matplotlib) (1.1.
1)
Requirement already satisfied: cycler>=0.10 in /opt/anaconda3/envs/
ml_homework/lib/python3.8/site-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /opt/anaconda3/
envs/ml_homework/lib/python3.8/site-packages (from matplotlib) (4.4
9.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /opt/anaconda3/
envs/ml_homework/lib/python3.8/site-packages (from matplotlib) (1.
4.5)
Requirement already satisfied: numpy<2,>=1.20 in /opt/anaconda3/env
s/ml_homework/lib/python3.8/site-packages (from matplotlib) (1.24.
4)
Requirement already satisfied: packaging>=20.0 in /opt/anaconda3/en
vs/ml_homework/lib/python3.8/site-packages (from matplotlib) (24.0)
Requirement already satisfied: pillow>=6.2.0 in /opt/anaconda3/env
s/ml_homework/lib/python3.8/site-packages (from matplotlib) (10.2.
0)
Requirement already satisfied: pyparsing>=2.3.1 in /opt/anaconda3/e
nvs/ml_homework/lib/python3.8/site-packages (from matplotlib) (3.1.
2)
Requirement already satisfied: python-dateutil>=2.7 in /opt/anacond
a3/envs/ml_homework/lib/python3.8/site-packages (from matplotlib)
(2.9.0)
Requirement already satisfied: importlib-resources>=3.2.0 in /opt/a
naconda3/envs/ml_homework/lib/python3.8/site-packages (from matplot
lib) (6.3.0)
Requirement already satisfied: zipp>=3.1.0 in /opt/anaconda3/envs/m
l_homework/lib/python3.8/site-packages (from importlib-resources>=
3.2.0->matplotlib) (3.17.0)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/envs/ml_h
omework/lib/python3.8/site-packages (from python-dateutil>=2.7->mat
plotlib) (1.16.0)
```

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot([0,1,2,3],[5,9,3,2], color='#0F0F70', linestyle='--',\
         linewidth=4)
plt.plot([0,0.4,2.2,3],[2,-1,-2,3], color="green", linestyle=':', m
arker='p',\
         linewidth=4, markersize=15)
plt.plot([0,0.4,2.2,3],[-1,2,2,5], color="#dcdab2", linestyle='-',
marker='o',\
         linewidth=4, markersize=15)

plt.show()
```



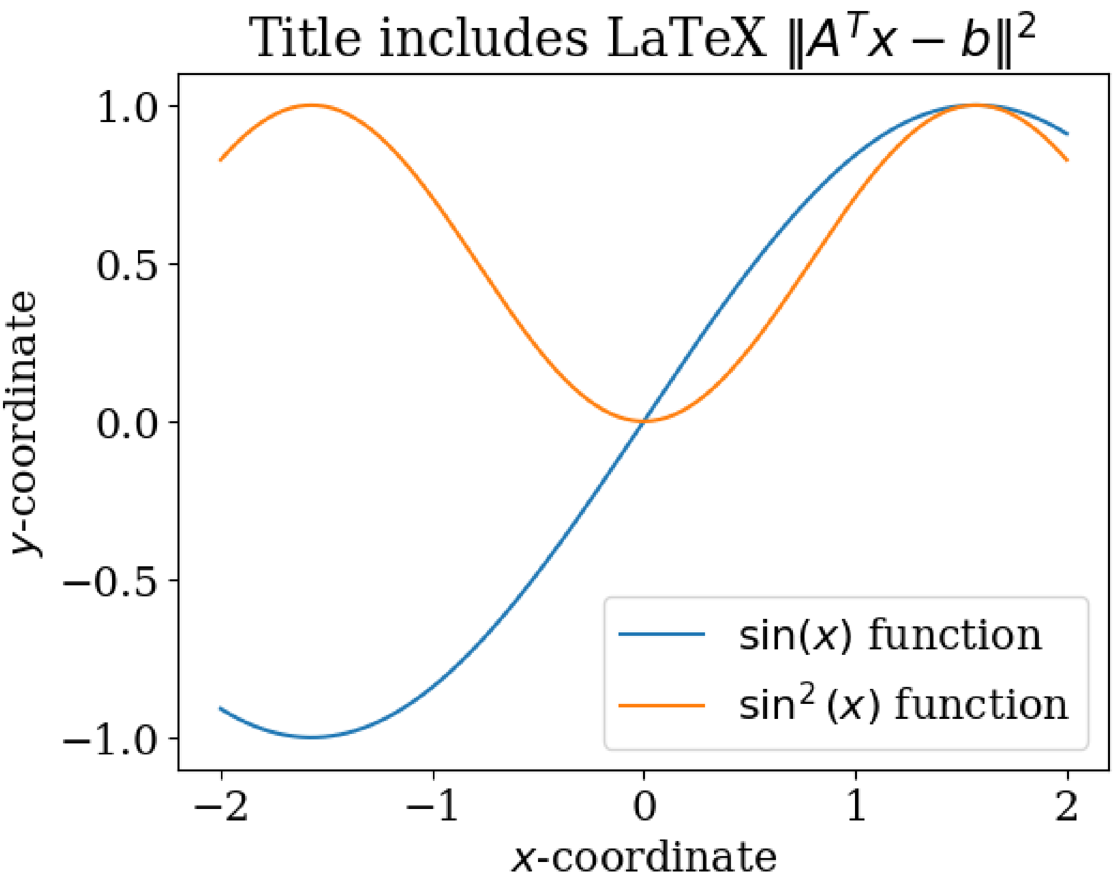
Lines are layered in the order they are added.

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

xx = np.linspace(-2,2,1024)
plt.plot(xx,np.exp(xx), color='red', linewidth=7)     #order matter
s
plt.plot(xx,np.cos(xx), color='blue', linewidth=7)    #order matter
s

plt.axis([-1.5,1.5,-1,3])


plt.show()
```



You can change font settings with `plt.rc`. ("rc" is a standard abbreviation in programming for "runtime configuration".)

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

# plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rc('font', size = 16)

xx = np.linspace(-2,2,1024)
plt.plot(xx,np.sin(xx), label="$\sin(x)$ function")
plt.plot(xx,np.sin(xx)**2, label="$\sin^2(x)$ function")

plt.xlabel("$x$-coordinate")
plt.ylabel("$y$-coordinate")
plt.title("Title includes LaTeX $\|A^Tx-b\|^2$")

plt.legend()

plt.show()
```

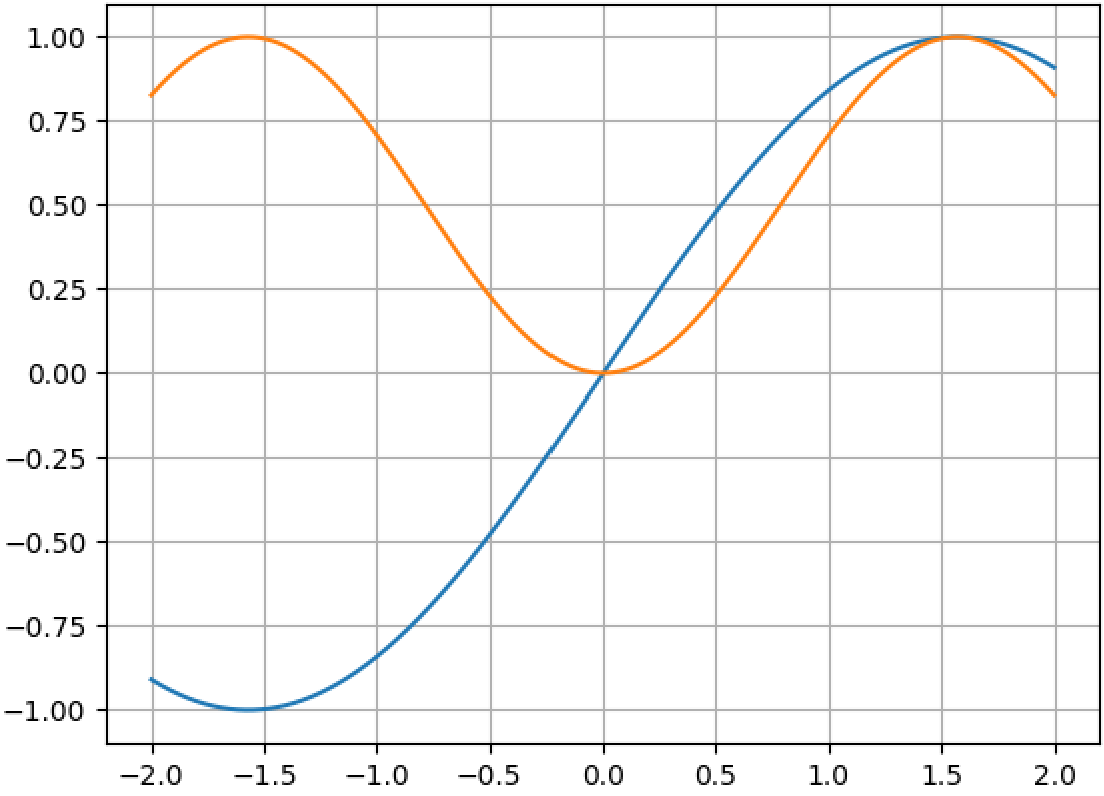


To return all `rc` settings to default, use:

In [ ]:
```
plt.rcdefaults()
```

`plt.grid()` creates a grid in the background.

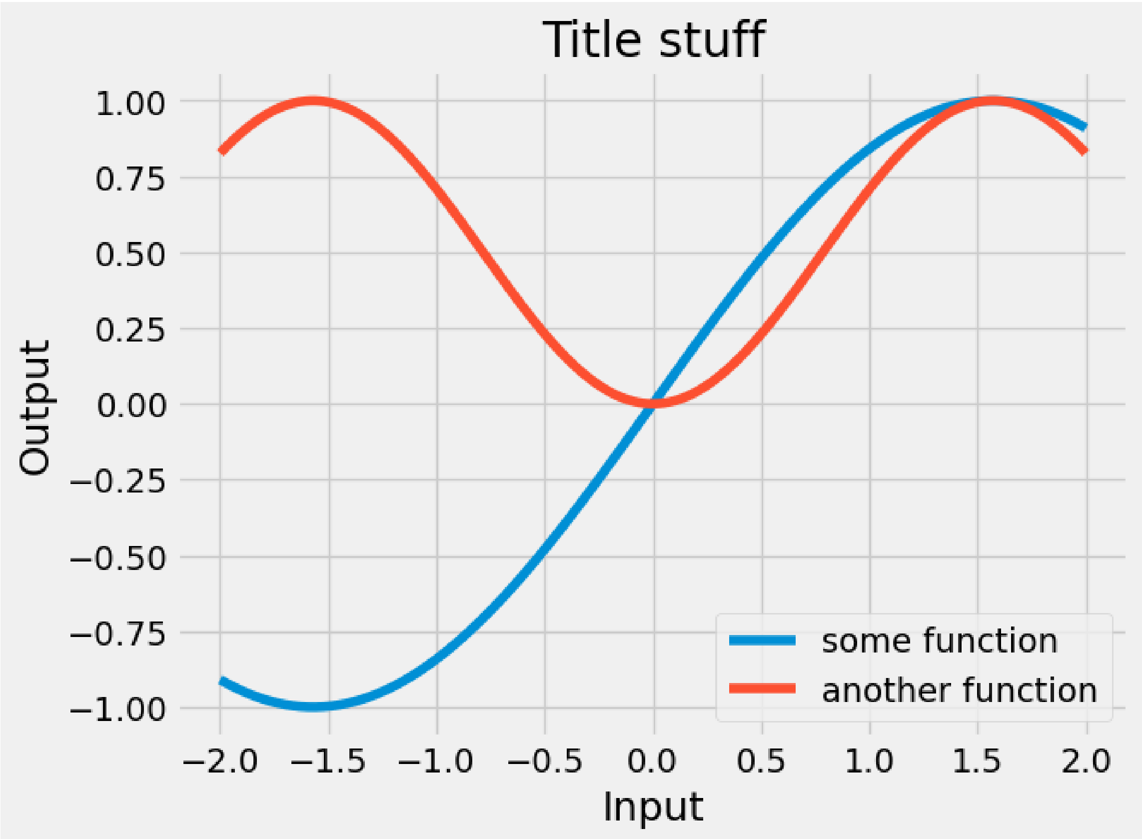
In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

xx = np.linspace(-2,2,1024)
plt.plot(xx,np.sin(xx))
plt.plot(xx,np.sin(xx)**2)


plt.grid()

plt.show()
```



If you are unhappy with the default style but do not want to spend time customizing your plots, use one of the available styles.

```
In [ ]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

print(plt.style.available)  #list of available styles
plt.rcdefaults()
plt.style.use('fivethirtyeight')     #use style ggplot ("gg" stands for Leland Wilkinson's "Grammar of Graphics")

xx = np.linspace(-2,2,1024)
plt.plot(xx,np.sin(xx), label="some function")
plt.plot(xx,np.sin(xx)**2, label="another function")

plt.xlabel("Input")
plt.ylabel("Output")
plt.title("Title stuff")

plt.legend()

plt.show()
```

```
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-nogrid', 'bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn-v0_8', 'seaborn-v0_8-bright', 'seaborn-v0_8-colorblind', 'seaborn-v0_8-dark', 'seaborn-v0_8-dark-palette', 'seaborn-v0_8-darkgrid', 'seaborn-v0_8-deep', 'seaborn-v0_8-muted', 'seaborn-v0_8-notebook', 'seaborn-v0_8-paper', 'seaborn-v0_8-pastel', 'seaborn-v0_8-poster', 'seaborn-v0_8-talk', 'seaborn-v0_8-ticks', 'seaborn-v0_8-white', 'seaborn-v0_8-whitegrid', 'tableau-colorblind10']
```



Save your figure as an image file using `plt.savefig(...)`.

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

plt.rcdefaults()

xx = np.linspace(-2,2,1024)
plt.plot(xx,np.sin(xx), label="some function")
plt.plot(xx,np.sin(xx)**2, label="another function")

plt.xlabel("Input")
plt.ylabel("Output")
plt.title("Title stuff")

plt.legend()

plt.savefig('plot.png')
```