Section 5 Class and Modules

A possibly overlooked point: Modules and Class in Python share many similaries at the basic level. They both contain some data (names, attributes) and codes (functions, methods) for the convenience of users -- and the codes to call them are also similar. Of course, Class also serves as the blue prints to generate instances, and supports more advanced functions such as Inheritance.

Class and Instance

Intuitively speaking, **classes** (or understood as types) are the "factories" to produce **instances** (concrete objects). For example, you can image that in the class of "list" in python, it defines the behavior of lists (methods) such as `append` , `copy` , and you can create concrete list objects (each with different values) from the list class, and directly uses the methods defined.

Programming with the idea of creating classes is the key to Object-Oriented Programming(OOP), (often%20known%20as%20methods)). See the concrete example of circle here.

Simple Example of Vector

Let's first define the simplest class in Python

In [1]:
```
class VectorV0:
    '''The simplest class in python'''  # this is the document string

    pass
```

and create two instances `v1` and `v2`

In [2]:
```
v1 = VectorV0()  # note the parentheses here, they are the grammar to create instance f
print(id(v1))
v2 = VectorV0()
id(v2)
```

```
140603288047376
```

Out[2]:
```
140603288047440
```

Now `v1` and `v2` are the objects in Python

In [3]:
```
type(v1)
```

Out[3]:
```
__main__.VectorV0
```

In [4]:
```
dir(v1)
```

Out[4]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
```

```
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

In [6]:

```
help(v1)
```

```
Help on VectorV0 in module __main__ object:

class VectorV0(builtins.object)
 |  The simplest class in python
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

We can manually assign the attributes to instance `v1` and `v2`

In [7]:

```
v1.x = 1.0 # this is called instance attributes
v1.y = 2.0
v2.x = 2.0
v2.y = 3.0
```

In [8]:

```
dir(v1)
```

Out[8]:

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
```

```
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'x',
 'y']
```

We don't want to create the instance or define the coordinates seperately. Can we do these in one step, when initializing the instance?

In [9]:

```python
class VectorV1:
    '''define the vector'''  # this is the document string
    dim = 2   # this is the attribute in class -- class attributes
    def __init__(self, x=0.0, y=0.0):  # any method in Class requires the first paramet
        self.x = x # instance attributes defined by self.attr
        self.y = y
```

In [10]:

```python
v1 = VectorV1(1.0,2.0) # you can pass the value directly, because you defined the __ini
```

In [11]:

```python
dir(v1)
```

Out[11]:

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'dim',
 'x',
 'y']
```

In [12]:
```python
print(v1.dim)
print(v1.x)
print(v1.y)
```

```
2
1.0
2.0
```

Btw, there is nothing mysterious about the `__init__` : you can just assume it is a function (method) stored in v1, and you can always call it if you like!

When you write `v1.__init__()` , you can equivalently think that you are calling a function with "ugly function name" `__init__` , and the parameter is `v1` (self), i.e. you are writing `__init__(v1)` . It is just a function updating the attributes of instance objects!

More generally, for the method `method(self, params)` you can call it by `self.method(params)` .

In [13]:
```python
print(v1.x)
print(id(v1))
y = v1.__init__() #reinitializes the values of our vector
print(v1.x)
print(id(v1))
print(y)
```

```
1.0
140603287795920
0.0
140603287795920
None
```

`v1` is just like a mutable object, and the "function" `__init__( )` just change `v1` in place!

Now we move on to update our vector class by defining more functions. Since you may not like ugly names here with dunder (a.k.a **d**ouble **under**score), let's just begin with normal function names.

In [1]:
```python
import math

class VectorV2:
    '''define the vector'''  # this is the document string
    dim = 2   # this is the class attribute

    def __init__(self, x=0.0, y=0.0):  # any method in Class requires the first paramet
        '''initialize the vector by providing x and y coordinate'''
        self.x = x
        self.y = y

    def norm(self):
        '''calculate the norm of vector'''
        return math.sqrt(self.x**2+self.y**2)

    def vector_sum(self, other):
        '''calculate the vector sum of two vectors'''
        return VectorV2(self.x + other.x, self.y + other.y)

    def show_coordinate(self):
        '''display the coordinates of the vector'''
        return 'Vector(%r, %r)' % (self.x, self.y)
```

In [15]:

```
help(VectorV2)
```

```
Help on class VectorV2 in module __main__:

class VectorV2(builtins.object)
 |  VectorV2(x=0.0, y=0.0)
 |  
 |  define the vector
 |  
 |  Methods defined here:
 |  
 |  __init__(self, x=0.0, y=0.0)
 |      initialize the vector by providing x and y coordinate
 |  
 |  norm(self)
 |      calculate the norm of vector
 |  
 |  show_coordinate(self)
 |      display the coordinates of the vector
 |  
 |  vector_sum(self, other)
 |      calculate the vector sum of two vectors
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |  
 |  dim = 2
```

In [3]:

```
v1 = VectorV2(1.0,2.0)
v2 = VectorV2(2.0,3.0)
```

In [4]:

```
v1_length = v1.norm()
print(v1_length)
```

```
2.23606797749979
```

Equivalent way to call this method is (although not used often):

In [19]:

```
VectorV2.norm(v1)
```

Out[19]:

```
2.23606797749979
```

Even for built-in types, we have something similiar

In [20]:

```
a = [1,2,3]
list.append(a,4) # equivalent to a.append(4), note that list is the class name
print(a)
```

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

despite that we don't have any reason not to use `a.append()` directly.

In [21]:
```
v3 = v1.vector_sum(v2)
v3.show_coordinate()
```

Out[21]:
```
'Vector(3.0, 5.0)'
```

In [22]:
```
v1+v2 # will it work?
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-3b50698d06d4> in <module>
----> 1 v1+v2 # will it work?

TypeError: unsupported operand type(s) for +: 'VectorV2' and 'VectorV2'
```

In [25]:
```
print(v3)
v3
```

```
<__main__.VectorV2 object at 0x7fe0c1103a90>
```

Out[25]:
```
<__main__.VectorV2 at 0x7fe0c1103a90>
```

Something that we are still not satisfied:

- By typing v3 or using `print()` in the code, we cannot show its coordinates directly
- We cannot use the `+` operator to calculate the vector sum

Special (Magic) Methods

Here's the magic: by merely changing the function name, we can realize our goal!

In [26]:
```
class VectorV3:
    '''define the vector'''  # this is the document string
    dim = 2   # this is the attribute

    def __init__(self, x=0.0, y=0.0):  # any method in Class requires the first paramet
        '''initialize the vector by providing x and y coordinate'''
        self.x = x
        self.y = y

    def norm(self):
        '''calculate the norm of vector'''
        return math.sqrt(self.x**2+self.y**2)

    def __add__ (self, other):
        '''calculate the vector sum of two vectors'''
        return VectorV3(self.x + other.x, self.y + other.y)

    def __repr__(self):   #special method of string representation
        '''display the coordinates of the vector'''
        return 'Vector(%r, %r)' % (self.x, self.y)
```

In [27]:
```python
help(VectorV3)
```

```
Help on class VectorV3 in module __main__:

class VectorV3(builtins.object)
 |  VectorV3(x=0.0, y=0.0)
 |  
 |  define the vector
 |  
 |  Methods defined here:
 |  
 |  __add__(self, other)
 |      calculate the vector sum of two vectors
 |  
 |  __init__(self, x=0.0, y=0.0)
 |      initialize the vector by providing x and y coordinate
 |  
 |  __repr__(self)
 |      display the coordinates of the vector
 |  
 |  norm(self)
 |      calculate the norm of vector
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |  
 |  dim = 2
```

In [28]:
```python
v1 = VectorV3(1.0,2.0)
v2 = VectorV3(2.0,3.0)
```

In [29]:
```python
v3 = v1.__add__(v2) # just call special methods as ordinary methods
v3.__repr__()
```

Out[29]:
```
'Vector(3.0, 5.0)'
```

In [35]:
```python
v3 = v1 +v2 # here is the point of using special methods!
```

In [36]:
```python
print(v3)
```

```
Vector(3.0, 5.0)
```

Special methods are just like VIP admissions to take full use of the built-in operators in Python. With other special methods, you can even get elements by index `v3[0]`, or iterate through the object you created. For more advanced usage, you can see here.

(Optional) More Comments about `__repr__()` and `__str__()`

These are all the methods to display some strings about the object. An obvious difference is that when you directly **run** (evaluate) the object in code cell, it will execute `__repr__` , and when you **print** the object, it will first execute `__str__` . If `__str__` is not defined, then when calling `print` , the `__repr__` will be executed, but not vice versa. For more information, see the discussion here.

In [37]:
```
class VectorV3_1:
    '''define the vector'''  # this is the document string
    dim = 2   # this is the attribute

    def __init__(self, x=0.0, y=0.0):  # any method in Class requires the first paramet
        '''initialize the vector by providing x and y coordinate'''
        self.x = x
        self.y = y

    def __repr__(self):   #special method of string representation
        '''display the coordinates of the vector'''
        return 'repr: Vector(%r, %r)' % (self.x, self.y)

    def __str__(self):   #special method of string representation
        '''display the coordinates of the vector'''
        return 'str: vector[%r, %r]' % (self.x, self.y)
```

In [38]:
```
v1 = VectorV3_1(1.0,2.0)
```

In [39]:
```
v1 # directly call in cell code, or from repr() function
```

Out[39]:
```
repr: Vector(1.0, 2.0)
```

In [40]:
```
print(v1)
```

```
str: vector[1.0, 2.0]
```

Inheritance

Now we want to add another scalar production method to Vector, but we're tired of rewriting all the other methods. A good way is to create new Class VectorV4 (Child Class) by inheriting from VectorV3 (Parent Class) that we have already defined.

In [41]:
```
class VectorV4(VectorV3): # Note the class VectorV3 in parentheses here
    '''define the vector'''  # this is the document string
    def __mul__(self, scalar):
        '''calculate the scalar product'''
        return VectorV4(self.x * scalar, self.y * scalar)
```

In [42]:
```
help(VectorV4)
```

```
Help on class VectorV4 in module __main__:

class VectorV4(VectorV3)
 |  VectorV4(x=0.0, y=0.0)
```

```
 |
 |  define the vector
 |
 |  Method resolution order:
 |      VectorV4
 |      VectorV3
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __mul__(self, scalar)
 |      calculate the scalar product
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from VectorV3:
 |
 |  __add__(self, other)
 |      calculate the vector sum of two vectors
 |
 |  __init__(self, x=0.0, y=0.0)
 |      initialize the vector by providing x and y coordinate
 |
 |  __repr__(self)
 |      display the coordinates of the vector
 |
 |  norm(self)
 |      calculate the norm of vector
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from VectorV3:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from VectorV3:
 |
 |  dim = 2
```

In [43]:

```
v1 = VectorV4(1.0,2.0)
v2 = VectorV4(2.0,3.0)
```

In [44]:

```
v1+v2
```

Out[44]:

```
Vector(3.0, 5.0)
```

In [45]:

```
v1*2
```

Out[45]:

```
Vector(2.0, 4.0)
```

Modules and Packages

In Python, Functions (plus Classes, Variables) are contained in Modules, and Modules are organized in directories of Packages. In fact, Modules are also objects in Python!

Now we have the `Vector.py` file in the folder. When we import the module, the interpreter will create a name `Vector` pointing to the module object. The functions/classes/variables defined in the module can be called with `Vector.XXX`, i.e. they are in the **namespace** of `Vector` (can be seen through `dir`).

Of course, the (annoying) rules of object assignment (be careful about changing mutable objects even in modules) in Python still applies, but we won't go deep in this course.

In [9]:
```
import Vector
print(type(Vector))
dir(Vector) # 'attributes' (namespace) in the module Vector -- note the variables/funct
```

```
<class 'module'>
```

Out[9]:
```
['VectorV5',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'print_hello',
 'string']
```

In [10]:
```
Vector.string
```

Out[10]:
```
'Python'
```

In [11]:
```
Vector.print_hello()
```

```
Hello
```

In [12]:
```
v5 = Vector.VectorV5(1.0,2.0)
v5
```

Out[12]:
```
Vector(1.0, 2.0)
```

Other different ways to import module:

In [13]:
```
import Vector as vc # create a name vc point to the module Vector.py -- good practice,
vc.string
```

Out[13]:
```
'Python'
```

In [14]:
```
from Vector import print_hello # may cause some name conflicts if write larger programs
print_hello() # Where does this print_hello come from ? It may take some time to figure
```

```
Hello
```

It's totally possible that different modules (packages) contain same names. Some problems may happen if we try the from...import way. That's why the first way (import or import as) is always recommended.

In [15]:
```
import math
import numpy as np
print(math.cos(math.pi))# eveything is clear -- there won't be any confusions
print(np.cos(np.pi))# eveything is clear -- there won't be any confusions
```

```
-1.0
-1.0
```

In [55]:
```
from Vector import * # Be careful about import everything -- may cause serious name con
string
```

Out[55]:
```
'Python'
```

To import the modules, you must ensure that they are in your system paths.

In [16]:
```
import sys
sys.path
```

Out[16]:
```
['C:\\Users\\Luke\\Math_10_SS1',
 'E:\\ProgramData\\Anaconda3\\python38.zip',
 'E:\\ProgramData\\Anaconda3\\DLLs',
 'E:\\ProgramData\\Anaconda3\\lib',
 'E:\\ProgramData\\Anaconda3',
 '',
 'E:\\ProgramData\\Anaconda3\\lib\\site-packages',
 'E:\\ProgramData\\Anaconda3\\lib\\site-packages\\locket-0.2.1-py3.8.egg',
 'E:\\ProgramData\\Anaconda3\\lib\\site-packages\\win32',
 'E:\\ProgramData\\Anaconda3\\lib\\site-packages\\win32\\lib',
 'E:\\ProgramData\\Anaconda3\\lib\\site-packages\\Pythonwin',
 'E:\\ProgramData\\Anaconda3\\lib\\site-packages\\IPython\\extensions',
 'C:\\Users\\Luke\\.ipython']
```

In [ ]:
```
sys.modules.keys() # check all the modules are currently imported in the kernel
```

We can import the `inspect` module and use `getsource` function to see the source codes of imported modules.

In [8]:
```
import inspect # this inspect itself is a module!
lines = inspect.getsource(Vector.VectorV5)
print(lines)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-dc5d0cbb68df> in <module>
      1 import inspect # this inspect itself is a module!
----> 2 lines = inspect.getsource(Vector.VectorV5)
      3 print(lines)

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

Note that this does not work for some Python modules/functions (Because they are written in C language).

You can view all the source codes of Python here. Here is the complete documentation for reference about standard Python libary -- the .py files that are now in your computer when you install python!

In [17]:

```
import math # this won't work, because math is the built-in function -- written in C la
lines = inspect.getsource(math.sqrt) # will print the error
print(lines)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-c1fae04687db> in <module>
      1 import math # this won't work, because math is the built-in function -- written
 in C language!
----> 2 lines = inspect.getsource(math.sqrt) # will print the error
      3 print(lines)

E:\ProgramData\Anaconda3\lib\inspect.py in getsource(object)
    995     or code object.  The source code is returned as a single string.  An
    996     OSError is raised if the source code cannot be retrieved."""
--> 997     lines, lnum = getsourcelines(object)
    998     return ''.join(lines)
    999 

E:\ProgramData\Anaconda3\lib\inspect.py in getsourcelines(object)
    977     raised if the source code cannot be retrieved."""
    978     object = unwrap(object)
--> 979     lines, lnum = findsource(object)
    980 
    981     if istraceback(object):

E:\ProgramData\Anaconda3\lib\inspect.py in findsource(object)
    778     is raised if the source code cannot be retrieved."""
    779 
--> 780     file = getsourcefile(object)
    781     if file:
    782         # Invalidate cache if needed.

E:\ProgramData\Anaconda3\lib\inspect.py in getsourcefile(object)
    694     Return None if no way can be identified to get the source.
    695     """
--> 696     filename = getfile(object)
    697     all_bytecode_suffixes = importlib.machinery.DEBUG_BYTECODE_SUFFIXES[:]
    698     all_bytecode_suffixes += importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES[:]

E:\ProgramData\Anaconda3\lib\inspect.py in getfile(object)
    674     if iscode(object):
    675         return object.co_filename
--> 676     raise TypeError('module, class, method, function, traceback, frame, or '
    677                     'code object was expected, got {}'.format(
    678                     type(object).__name__))

TypeError: module, class, method, function, traceback, frame, or code object was expecte
d, got builtin_function_or_method
```

In [18]:

```
import copy # this can work, because copy.py is the "lib" folder and is written in Pyth
lines = inspect.getsource(copy.deepcopy) # no problem
print(lines)
```

```
def deepcopy(x, memo=None, _nil=[]):
    """Deep copy operation on arbitrary Python objects.

    See the module's __doc__ string for more info.
```

```
    """

    if memo is None:
        memo = {}

    d = id(x)
    y = memo.get(d, _nil)
    if y is not _nil:
        return y

    cls = type(x)

    copier = _deepcopy_dispatch.get(cls)
    if copier is not None:
        y = copier(x, memo)
    else:
        if issubclass(cls, type):
            y = _deepcopy_atomic(x, memo)
        else:
            copier = getattr(x, "__deepcopy__", None)
            if copier is not None:
                y = copier(memo)
            else:
                reductor = dispatch_table.get(cls)
                if reductor:
                    rv = reductor(x)
                else:
                    reductor = getattr(x, "__reduce_ex__", None)
                    if reductor is not None:
                        rv = reductor(4)
                    else:
                        reductor = getattr(x, "__reduce__", None)
                        if reductor:
                            rv = reductor()
                        else:
                            raise Error(
                                "un(deep)copyable object of type %s" % cls)
                if isinstance(rv, str):
                    y = x
                else:
                    y = _reconstruct(x, memo, *rv)

    # If is its own copy, don't memoize.
    if y is not x:
        memo[d] = y
        _keep_alive(x, memo) # Make sure x lives at least as long as d
    return y
```

In [19]:

```
inspect.getsourcefile(copy) # see? the copy.py is in our local computer
```

Out[19]:

```
'E:\\ProgramData\\Anaconda3\\lib\\copy.py'
```

Notes on Numpy Package

If we are interested in `numpy` that we're going to talk about in details soon -- in fact `numpy` is a package rather than modules. Package can contain many modules (some are also called subpackages, their difference is not important for our course) -- for example, the module (or subpackage, which is in the sub-directory of numpy) of linalg.

In [ ]:

```python
import numpy as np # import the package numpy, and assign the "nickname" np to it
[name for name in sys.modules.keys() if name.startswith('numpy')] # check what modules
```

In [ ]:

```python
print(np)
dir(np) # namespace of numpy package -- it also includes the functions in np.core
```

Something special about numpy: The namespace of numpy contains both modules (e.g. `linalg` module) and functions (e.g. `sum` function). In fact, thesse functions are imported from the modules (subpackages) `numpy.core` or `numpy.lib` -- they are loaded only for the convenience of users, because of their high frequency in usage. For a more complete understanding, we can go to see the structure of numpy package in GitHub.

In [22]:

```python
type(np.linalg)
```

Out[22]:

```
module
```

In [23]:

```python
type(np.sum)
```

Out[23]:

```
function
```

In [24]:

```python
print(id(np.core.sum))
print(id(np.sum))# see? np.sum is the same function with np.core.sum. In your usage, pl
np.core.sum is np.sum
```

```
2022960148688
2022960148688
```

Out[24]:

```
True
```

In [ ]:

```python
print(inspect.getsource(np.sum))# let's see the source code of sum function
```

In [26]:

```python
'eig' in dir(np) # where is the eigen value/vector function?
```

Out[26]:

```
False
```

In [27]:

```python
np.eig # Won't work! Because eig is not defined in numpy (core) module!
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-27-a5400bd55fe7> in <module>
----> 1 np.eig # Won't work! Because eig is not defined in numpy (core) module!

E:\ProgramData\Anaconda3\lib\site-packages\numpy\__init__.py in __getattr__(attr)
    301                 return Tester
    302
--> 303             raise AttributeError("module {!r} has no attribute "
    304                                  "{!r}".format(__name__, attr))
    305

AttributeError: module 'numpy' has no attribute 'eig'
```

In [28]:
```
print(np.linalg) # np.linalg is a module(subpackage) -- its namespace containing many f
dir(np.linalg) # let's check the names (functions) in linalg
```

```
<module 'numpy.linalg' from 'E:\\ProgramData\\Anaconda3\\lib\\site-packages\\numpy\\lina
lg\\__init__.py'>
```

Out[28]:
```
['LinAlgError',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_umath_linalg',
 'cholesky',
 'cond',
 'det',
 'eig',
 'eigh',
 'eigvals',
 'eigvalsh',
 'inv',
 'lapack_lite',
 'linalg',
 'lstsq',
 'matrix_power',
 'matrix_rank',
 'multi_dot',
 'norm',
 'pinv',
 'qr',
 'slogdet',
 'solve',
 'svd',
 'tensorinv',
 'tensorsolve',
 'test']
```

In [ ]:
```
help(np.linalg.eig) # eig function is here! Don't forget to import numpy as np first
```

In [30]:
```
from numpy import linalg # another way to import linalg module(subpackage) from numpy p
linalg.eig # now we create a name linalg to point to the linalg.py module, and can get
```

Out[30]:
```
<function numpy.linalg.eig(a)>
```

In [31]:
```
import numpy.linalg as LA # another way to import the linalg
LA.eig
```

Out[31]:
```
<function numpy.linalg.eig(a)>
```

In [32]:
```
import numpy.linalg # another way to import the linalg
numpy.linalg.eig
```

Out[32]:
```
<function numpy.linalg.eig(a)>
```

In [33]:
```python
from numpy.linalg import eig #import the eig function directly
eig
```

Out[33]:
```
<function numpy.linalg.eig(a)>
```

Take-home message (Basic requirements)

- Understand the concept of Python modules (.py files storing objects)
- Know different ways to import modules and objects in the modules (`import` , `import ... as` , `from ... import`)
- Understand the basic concept of package, and know how to import modules and functions within it (use `numpy` , `linalg` and `eig` as example)

Beyond Basic Python: What's next? -- Some Suggestions

- Knowledge and wisdom
- What we have not covered in basic python: other data types (dictionary, set, tuple), input/output, exceptions, -- consult a byte of python, or programiz
- The systematic book (for example,Python Cookbook) or course in computer science department (ICS-31,33)
- Practice!Practice!Practive! Useful websites such as Leetcode
- These cheetsheets from datacamp websites might also be helpful throughout this course.