Lecture 6 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) (https://en.wikipedia.org/wiki/Object-oriented_programming#:~:text=Object%2Doriented%20programming%20(OOP), (often%20known%20as%20methods)). See the concrete example of circle here (https://en.wikipedia.org/wiki/Comparison_of_programming_paradigms).

Simple Example of Vector

Let's first define the simplest class in Python

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

    pass
```

and create two instances `v1` and `v2`

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

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

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

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

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

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

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

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 [ ]:
```
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 param
eter to be self!
        self.x = x # instance attributes defined by self.attr
        self.y = y
```

In [ ]:
```
v1 = VectorV1(1.0,2.0) # you can pass the value directly, because you defined the __i
nit__ in class
```

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

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

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 [ ]:
```
print(v1.x)
print(id(v1))
y = v1.__init__()
print(v1.x)
print(id(v1))
print(y)
```

`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, let's just begin with normal function names.

In [ ]:
```
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 param
eter to be self!
        '''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 [ ]:
```
help(VectorV2)
```

In [ ]:
```
v1 = VectorV2(1.0,2.0)
v2 = VectorV2(2.0,3.0)
```

In [ ]:
```
v1.norm()
```

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

In [ ]:
```
VectorV2.norm(v1)
```

Even for built-in types, we have something similiar

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

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

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

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

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

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 [ ]:
```
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 param
eter to be self!
        '''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 [ ]:
```
help(VectorV3)
```

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

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

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

In [ ]:
```
v3
```

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 (https://rszalski.github.io/magicmethods/).

(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 (https://stackoverflow.com/questions/1436703/what-is-the-difference-between-str-and-repr).

In [ ]:
```
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 param
eter to be self!
        '''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 [ ]:
```
v1 = VectorV3_1(1.0,2.0)
```

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

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

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

In [ ]:
```
v1 = VectorV4(1.0,2.0)
v2 = VectorV4(2.0,3.0)
```

In [ ]:
```
v1+v2
```

In [ ]:
```
v1*2
```

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 [ ]:
```
import Vector
print(type(Vector))
dir(Vector) # 'attributes' (namespace) in the module Vector -- note the variables/fun
ctions we have defined in the .py file are here!
```

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

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

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

Other different ways to import module:

In [ ]:
```
import Vector as vc # create a name vc point to the module Vecotr.py -- good practic
e, all the functions will start with vc. -- you know where they are from!
vc.string
```

In [ ]:
```
from Vector import print_hello # may cause some name conflicts if write larger progra
ms
print_hello() # where does this print_hello come from ? it may take some time to figu
re out...
```

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 [ ]:
```
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
```

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

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

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

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

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 (https://github.com/python/cpython). Here is the complete documentation for reference about standard Python libary (https://docs.python.org/3/library/) -- the .py files that are now in your computer when you install python!

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

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

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

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 (https://github.com/numpy/numpy/blob/master/numpy/linalg/linalg.py).

In [ ]:
```
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 module
s in numpy package has been imported
```

In [ ]:
```
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 (https://github.com/numpy/numpy).

In [ ]:
```
type(np.linalg)
```

In [ ]:
```
type(np.sum)
```

In [ ]:
```
print(id(np.core.sum))
print(id(np.sum))# see? np.sum is the same function with np.core.sum. In your usage,
 please use np.sum because it is more convenient
np.core.sum is np.sum
```

In [ ]:
```
print(inspect.getsource(np.sum))# let's see the source code of sum function
```

In [ ]:
```
'eig' in dir(np) # where is the eigen value/vector function?
```

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

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

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

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

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

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

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

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 (https://python.swaroopch.com/), or programiz (https://www.programiz.com/python-programming)
- The systematic book (for example,Python Cookbook (https://www.oreilly.com/library/view/python-cookbook-3rd/9781449357337/)) or course in computer science department (ICS-31,33)
- Practice!Practice!Practive! Useful websites such as Leetcode (https://leetcode.com/)
- These cheetsheets (https://www.datacamp.com/community/data-science-cheatsheets?page=4) from datacamp websites might also be helpful throughout this course.