Branch: master

Find file | Copy path

ma591spring2020 / material / classes.ipynb

asaibab Add files via upload

4110fbe yesterday

1 contributor

Raw | Blame | History

803 lines (802 sloc) 22.1 KB

Classes and Objects

Housekeeping:

```
+ Homework 9 associated with this lecture. This is due April 10
 and will be the final homework
+ Reminder that office hours are Fridays 1:55PM - 3:30 PM via zo
om
+ Project options are due April 4 along with Homework 8
+ Please give feedback on lectures
+ Thank you!
```

Reading: Langtangen's "A primer on scientific programing with Python" Chapters 7 and 9.

A **class** is a (user-defined) data structure that has the following components:

```
+ A constructor: for initialization
+ Attributes: data  associated with the class
+ Methods: functions associated with the class
```

One or more of these can be optional. Some programming languages also require the user to implement a destructor, or garbage collector. However, Python has automatic garbage collection.

An instance of a class is known as an **object**. Some examples:

```
+ `list` is a class. `l = [1,2,3]` is an object.
+ `numpy.ndarray` is a class. `d = numpy.array([1,2,3])` is an o
bject.
```

Similarly dictionaries, sets, ints, floats, strings, figures, subplots are all classes.

Why object oriented?

- Enables reutilizing code through modularity
- Powerful tool to handle complex tasks because of abstraction
- Understanding this is important to use Python libraries

Why not object oriented?

- "Steep" learning curve
- Some loss in perforance due to overhead
- Anything that can be written using object-oriented programming can be written without it

Example: Complex numbers

Although there are already data structures for using complex numbers, let us consider writing our own class. We want the following functionalities

```
z = Complex(1.,1.0)
print('The real part of z is ', z.real)
print('The imaginary part of z is ', z.imag)
print('The absolute value of z is ', z.abs())
print('A string representation of z is ', z)

w = Complex(2.0,3.0)
print(z + w)
print(z-w)
print(z*w)
print(z/w)
```

In this previous example, we want our class to have

- two attributes (real, imag)
- several methods (abs,arg)
- operations (addition, multiplication, division)

Take 0: Without using a class

We can store the real and imaginary parts using a dictionary. In the first attempt we will cover the first few items in the list and handle other operations later on.

In [1]:
```
z = {'real':1.0, 'imag': 1.0}

import math

def zstr(z):
    print('%2.12g+%2.12g *j'%(z['real'],z['imag']))
    return

def cabs(z):
    zr = z['real']
    zi = z['imag']

    return math.sqrt(zr**2. + zi**2.)

def cadd(z,w):
    return {'real':z['real']+w['real'], \
                'imag': z['imag'] + w['imag']}
```

In [2]:
```
zstr(z)
print(cabs(z))
w = {'real':-1.0, 'imag': 10.0}
y = cadd(z,w)
zstr(y)
```

```
 1+ 1 *j
1.4142135623730951
 0+11 *j
```

We can write the complex operations using a dictionary but the resulting operations can look a little messy.

Take 1: simple implementation

Let us implement the same operations using a class structure. Later on, we will handle the operations.

In [3]:
```
import math

class Complex:
    """
    The documentation for the class goes here.
    Notice that a convention is to make the class names capi
talized.


    """

    def __init__(self, real = 0., imag = 0.):
        """
        Constructor for the class
        """
        self.real = real
        self.imag = imag

    def __str__(self):
        return '%2.12g+%2.12g *j'%(self.real,self.imag)

    def abs(self):
        """
        |z| = \sqrt{x^2 + y^2}

        """

        self._abs = math.sqrt(self.real**2. + self.imag**2.)
        return self._abs
```

In [4]:
```
help(Complex)
```

```
Help on class Complex in module __main__:

class Complex(builtins.object)
 |  Complex(real=0.0, imag=0.0)
 |
 |  The documentation for the class goes here.
 |  Notice that a convention is to make the class names capi
talized.
 |
 |  Methods defined here:
 |
```

```
 |
 |  __init__(self, real=0.0, imag=0.0)
 |      Constructor for the class
 |
 |  __str__(self)
 |      Return str(self).
 |
 |  abs(self)
 |      |z| = \sqrt{x^2 + y^2}
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

We test out the various implementations of the class.

In [5]:
```
z = Complex(2.,-1.5)
print('The real part of z is %.4g and the imaginary part is %.4g' %( z.real,z.imag))
print('The absolute value of z is ', z.abs())
print('The complex number is ', z)
```

```
The real part of z is 2 and the imaginary part is -1.5
The absolute value of z is  2.5
The complex number is   2+-1.5 *j
```

In [6]:
```
z = Complex(imag = 1.0)
print(z)
```

```
 0+ 1 *j
```

In [7]:
```
print(type(z)) # The type of z
```

```
<class '__main__.Complex'>
```

Some remarks:

- Classes allow for modular codes by grouping data and functions.
- Here `Complex` is the class, and `z` is an instance of this class.
- Often, codes involving classes can be written without classes. However, classes can be quite powerful.
- If an attribute or a method starts with a _, the convention is that it is "protected" or "private" and should not be used or accessed outside the class by unauthorized users (anyone other than the developer).

The `self` variable

- `self` has to be the first argument of each method (function), including the constructor.
- We can use `self.<variable>` to access the (local) attributes and `self.<method>(<args>)` to call the (local) methods.
- When we call a method, we can drop the `self` argument.
- We can initialize attributes at places other than constructors (e.g. `self._abs`)

In this example, the constructor `__init__` is initializing the variables `real` and `imag`.

```
self.real = real
self.imag = imag
```

These attributes are stored and can be accessed by any method within the class. For example, both the `abs` and `__str__` methods were able to access the attributes.

Take 2: This time with the operations

We are going to overload the + symbol (and other symbols) to handle complex numbers. This is known as **operator overloading**.

In [8]:
```
class Complex:
    def __init__(self, real = 0., imag = 0.):

        self.real = real
        self.imag = imag

    def abs(self):
        self._abs = math.sqrt(self.real**2. + self.imag**2.)
        return self._abs

    def __str__(self):
        return '%2.12g+%2.12gj'%(self.real,self.imag)

    def __add__(self, other):
        # mreal = self.real + other.real
        # mimag = self.imag + other.imag
        # m = Complex(mreal,mimag)
        # return m
        return Complex(self.real + other.real, self.imag + o
ther.imag)

    def __sub__(self, other2):
        return Complex(self.real - other2.real, self.imag -
other2.imag)

    def __mul__(self, other):
        return Complex(self.real*other.real - self.imag*othe
r.imag,\
                      self.imag*other.real + self.real*other
.imag)
    def __eq__(self, other):
        eps = 1.e-14
        return abs(self.real-other.real) < eps and abs(self.
imag - other.imag) < eps
```

```
    def __abs__(self):
        return self.abs()
```

Let's see various ways of using this class.

In [9]:
```
z = Complex(1.0,1.0)
w = Complex(2.0,3.0)
w2 = Complex(3,4)
y1 = z + w + w2
print(y1)
```

```
 6+ 8j
```

In [10]:
```
y2 = z - w
print(y2)
y3 = z*w
print(y3)
print(z.abs(), abs(z))        # Alternative way of calling
 z.abs();  see __abs__()
print(z == w)        # Is z = w?
```

```
-1+-2j
-1+ 5j
1.4142135623730951 1.4142135623730951
False
```

We can also overload other operators. Here is a subset of the operations and the corresponding methods (see Langtangen's Primer book, Section 7.7).

Operation	Method
+	`__add__`
-	`__sub__`
*	`__mul__`
/	`__div__`
**	`__pow__`
<	`__lt__`
<=	`__le__`
==	`__eq__`
!=	`__ne__`

Implementing all these functionalities for the Complex class requires some effort. This doesn't necessarily mean that you should have to do it. Python has two modules: `cmath` and `numpy` to handle complex numbers.

In [11]:
```
help(complex)
```

```
Help on class complex in module builtins:
```

```
class complex(object)
 |  complex(real=0, imag=0)
 |  
 |  Create a complex number from a real part and an optional 
imaginary part.
 |  
 |  This is equivalent to (real + imag*1j) where imag defaul
ts to 0.
 |  
 |  Methods defined here:
 |  
 |  __abs__(self, /)
 |      abs(self)
 |  
 |  __add__(self, value, /)
 |      Return self+value.
 |  
 |  __bool__(self, /)
 |      self != 0
 |  
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |  
 |  __eq__(self, value, /)
 |      Return self==value.
 |  
 |  __float__(self, /)
 |      float(self)
 |  
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |  
 |  __format__(...)
 |      complex.__format__() -> str
 |      
 |      Convert to a string according to format_spec.
 |  
 |  __ge__(self, value, /)
 |      Return self>=value.
 |  
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |  
 |  __getnewargs__(...)
 |  
 |  __gt__(self, value, /)
 |      Return self>value.
 |  
 |  __hash__(self, /)
 |      Return hash(self).
 |  
 |  __int__(self, /)
 |      int(self)
 |  
 |  __le__(self, value, /)
 |      Return self<=value.
 |  
 |  __lt__(self, value, /)
```

```
 |  __lt__(self, value, /)
 |      Return self<value.
 |  
 |  __mod__(self, value, /)
 |      Return self%value.
 |  
 |  __mul__(self, value, /)
 |      Return self*value.
 |  
 |  __ne__(self, value, /)
 |      Return self!=value.
 |  
 |  __neg__(self, /)
 |      -self
 |  
 |  __pos__(self, /)
 |      +self
 |  
 |  __pow__(self, value, mod=None, /)
 |      Return pow(self, value, mod).
 |  
 |  __radd__(self, value, /)
 |      Return value+self.
 |  
 |  __rdivmod__(self, value, /)
 |      Return divmod(value, self).
 |  
 |  __repr__(self, /)
 |      Return repr(self).
 |  
 |  __rfloordiv__(self, value, /)
 |      Return value//self.
 |  
 |  __rmod__(self, value, /)
 |      Return value%self.
 |  
 |  __rmul__(self, value, /)
 |      Return value*self.
 |  
 |  __rpow__(self, value, mod=None, /)
 |      Return pow(value, self, mod).
 |  
 |  __rsub__(self, value, /)
 |      Return value-self.
 |  
 |  __rtruediv__(self, value, /)
 |      Return value/self.
 |  
 |  __str__(self, /)
 |      Return str(self).
 |  
 |  __sub__(self, value, /)
 |      Return self-value.
 |  
 |  __truediv__(self, value, /)
 |      Return self/value.
 |  
 |  conjugate(...)
 |      complex.conjugate() -> complex
```

```
 |      complex.conjugate() -> complex
 |      
 |      Return the complex conjugate of its argument. (3-4
j).conjugate() == 3+4j.
 |  
 |  ----------------------------------------------------------------------
--------------
 |  Static methods defined here:
 |  
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for
accurate signature.
 |  
 |  ----------------------------------------------------------------------
--------------
 |  Data descriptors defined here:
 |  
 |  imag
 |      the imaginary part of a complex number
 |  
 |  real
 |      the real part of a complex number
```

Some useful commands

- `isinstance(<object>, <Class>)`: Is the object an instance of the Class?
- `hasattr(<object>,'<attribute>')`: Does the object have the attribute?

In [12]:
```
print(isinstance(z,Complex))
print(hasattr(z,'real'))
```

```
True
True
```

Exercise

Write a class called `Parabola` to implement the formula:

$$y(x) = a_0 + a_1 x + a_2 x^2$$

- The class constructor should have three attributes `a0, a1, a2`.
- It should have a method `eval` that evaluates the parabola at the point x.

Inheritance and Multiple Inheritance

Classes can be built on top of other classes. These allow us to reutilize functionalities without reimplementing the classes from scratch.

For example, a line is a special case of a parabola. Can we use the Parabola class we wrote to implement lines? Let us first implement the Parabola class.

In [13]:
```
class Parabola:
```

```python
    def __init__(self, a0, a1, a2):
        self.a0 = a0
        self.a1 = a1
        self.a2 = a2

    def eval(self, x):
        return self.a0 + self.a1*x + self.a2*x**2.

    def __call__(self, x):
        return self.eval(x)
```

In [14]:

```python
p = Parabola(1.,-1.,2.)
print(p.eval(2))
print(p(2.))      #Same as eval, because of __call__
```

```
7.0
7.0
```