03_22_classes_numerical_differentiation

Unknown Author

April 1, 2014

1 Numerical differentiation : Classes

In [1]:
```
# Imports
import math
import cmath
```

Classes

Classes are a tool to make the programs modular and hence reusable. Also they lend us a bit of abstraction which makes coding a bit easier. To give a simple example, one can define a class `Quadratic` which codes quadratic polynomials. Calling something like `q = Quadratic(a, b, c)` would define a function `q` such that `q(x) = ax^x + bx + c`. Classes can be used for more complicated things like coding and drawing geometric objects and so on.

Classes have their own set of variables, which we call *attribute*s and functions called *method*s. We have already encountered *method*s in our course, figure out which.

All the material in this lecture are in the 7th chapter of Langtangen's *A Primer on Scientific Programming with Python*.

First let us code the `Quadratic` class to gain some experience. Note that, though it is not technically necessary, all the classes are conventionally named using a capitalized word.

In [2]:
```
class Quadratic :
    """This class creates quadratic equations.
    Attributes :
        a, b, c : Coefficients of ax^2 + bx + c
    Methods :
        valueat(x) : Value of the quadrating at x.
        roots() : Returns a tuple containing the two roots
            of ax^2 + bx + c = 0
        display() : Print the quadratic equation.
    """
    # The doc string is similar to that of functions.
    # In a class, we need a function to initialize the class.
    def __init__(self, a, b, c) :
        """Constructor function for the class Quadratic."""
        # I'll explain the variable self below.
        self.a = a
        self.b = b
        self.c = c

    def valueat(self, x) :
        """This computes the value of the quadratic at x."""
```

```
        # In the following line note how we refer to the elements
        # a, b,c of the quadratic as self.a, self.b etc. self
        # is the instance of the current class.
        return (self.a)*x*x + (self.b)*x + (self.c)

    def roots(self) :
        """Returns the roots of the quadratic."""
        a = self.a
        b = self.b
        c = self.c
        if a !=  0 :
            disc = b*b - 4*a*c
            if disc >= 0 :
                r1 = float(-b + math.sqrt(disc))/2*a
                r2 = float(-b - math.sqrt(disc))/2*a
            else :
                r1 = (-b + cmath.sqrt(disc))/2*a
                r2 = (-b - cmath.sqrt(disc))/2*a
            retval = (r1, r2)
        else :
            if b != 0 :
                retval = - float(c)/b
            else :
                print "No solutions."
                retval = None
        return retval

    def display(self) :
        """Returns a string printing the quadratic."""
        str = "%g x^2 + %g x + %g" % (self.a, self.b, self.c)
        return str
```

Now we use the class to do some computations. For fun we define another function to do the computation.

In [3]:
```
def solve_quads(a, b, c) :
    q = Quadratic(a, b, c)
    # We call q an instance of the class Quadratic.
    print "q = ", q.display(), "has", q.roots(), "as roots. q(",
    print math.pi,") = ", q.valueat(math.pi)
    return None

solve_quads(1, 2, 1)
solve_quads(1, 0, 1)
solve_quads(1, 1, 0)
```

```
q =  1 x^2 + 2 x + 1 has (-1.0, -1.0) as roots. q( 3.14159265359 ) =
17.1527897083
q =  1 x^2 + 0 x + 1 has (1j, -1j) as roots. q( 3.14159265359 ) =
10.8696044011
q =  1 x^2 + 1 x + 0 has (0.0, -1.0) as roots. q( 3.14159265359 ) =
13.0111970547
```

Making classes callable.

We eventually want to diffentiate and integrate functions. We'll implement the as classes. What we want to do is to make the instances behave like functions. We do that using the `__call__` method. As an example, we just copy the `Quadratic` class to `Quadratic2` renaming the `valueat` method to be `__call__`.

In [4]:
```
class Quadratic2 :
    """This class creates quadratic equations.
    Attributes :
        a, b, c : Coefficients of ax^2 + bx + c
    Methods :
        __call__(x) : Value of the quadrating at x.
```

```
        roots() : Returns a tuple containing the two roots
            of ax^2 + bx + c = 0
        display() : Print the quadratic equation.
    """
    # The doc string is similar to that of functions.
    # In a class, we need a function to initialize the class.
    def __init__(self, a, b, c) :
        """Constructor function for the class Quadratic."""
        # I'll explain the variable self below.
        self.a = a
        self.b = b
        self.c = c

    def __call__(self, x) :
        """This computes the value of the quadratic at x."""
        # In the following line note how we refer to the elements
        # a, b,c of the quadratic as self.a, self.b etc. self
        # is the instance of the current class.
        return (self.a)*x*x + (self.b)*x + (self.c)

    def roots(self) :
        """Returns the roots of the quadratic."""
        a = self.a
        b = self.b
        c = self.c
        if a !=  0 :
            disc = b*b - 4*a*c
            if disc >= 0 :
                r1 = float(-b + math.sqrt(disc))/2*a
                r2 = float(-b - math.sqrt(disc))/2*a
            else :
                r1 = (-b + cmath.sqrt(disc))/2*a
                r2 = (-b - cmath.sqrt(disc))/2*a
            retval = (r1, r2)
        else :
            if b != 0 :
                retval = - float(c)/b
            else :
                print "No solutions."
                retval = None
        return retval

    def display(self) :
        """Returns a string printing the quadratic."""
        str = "%g x^2 + %g x + %g" % (self.a, self.b, self.c)
        return str
```

In [5]:
```
q = Quadratic2(1, -3 , 2)
print q(10)
```

```
72
```

Numeric Differentiation

We know that the mathematical definition of the derivative f' of a function f at x is

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1}$$

Instead of finding limit, we can code the derivative as just the fraction $(f(x+h) - f(x))/h$ for small h. *Assuming* that as we reduce h, the value of the function will be closer and closer to f', we can also write a check if reducing h, say by half, doesn't change the value of the fraction beyond some acceptible error. Then we can use that value as an approximate value for the derivative.

In [6]:
```
class SimpleDerivative :
    """Implements a callable class for a naive numerical
    derivative."""

    def __init__(self, f, h = 1.0E-3) :
        self.f = f
        self.h = float(h)

    def __call__(self, x) :
        """Return the value of the derivative at x."""
        f = self.f
        h = self.h
        der = (f(x + h) - f(x)) / h
        return der

    def set_deviation(self, val) :
        """Set the deviation."""
        self.h = val
```

In [7]:
```
def f(x) :
    return x*x

df = SimpleDerivative(f,.01)
print df(2)
df.set_deviation(df.h/100)
print df(2)
```

```
4.01
4.00010000001
```

Now to end, we modify the class so that when we call, it automatically keeps on halving the deviation till two consecutive derivatives differ by a very small amount. Again as in Newton's method we have to keep track of how many iterations we are doing.

In [8]:
```
class NaiveDerivative :
    """This uses a loop to compute derivatives with smaller
    smaller deviations till the difference between successive
    computations is less than a predetermined error

    Attributes :
        f : function whose derivative we seek,
        h : deviaion
        err : error
        N : max number of interations allowed

    Methods :
        single_der(x) : returns (f(x+h) - f(x)) / h
        set_dev(d) : sets the deviation to d
        set_err(e) : sets the error allowed to e
        allow_iter(M) : allow M iterations
        __init__ : Constructor
        __call__ : the actual code
    """

    def __init__(self, f, h=0.01, err=1E-5, N=10000) :
        self.f = f
        self.h = float(h)
        self.err = err
        self.N = N

    def set_dev(self, d) :
        self.h = d
        return None

    def set_err(self, e) :
        self.err = e
        return None
```

```
    def allow_iter(self, M) :
        self.N = M
        return None

    def single_der(self, x) :
        f = self.f
        h = self.h
        return (f(x + h) - f(x))/h

    def __call__(self, x) :
        N = self.N
        err = self.err

        dfold = self.single_der(x)
        self.set_dev(self.h / 2.0)
        dfnew = self.single_der(x)

        no_iter = 0
        while no_iter < N and abs(dfold - dfnew) >= err :
            dfold = dfnew
            self.set_dev(self.h / 2.0)
            dfnew = self.single_der(x)
            no_iter += 1

        if abs(dfold - dfnew) >= err :
            print "Could not converge after %d iterations." % no_iter
            retval = None
        else :
            retval = dfnew
        return retval
```

In [9]:

```
def myfn(x) :
    return x*x*x + x

dmyfn = NaiveDerivative(myfn)
dmyfn.set_dev(1)
dmyfn.set_err(.1)
print dmyfn(1)
dmyfn.set_err(1E-5)
print dmyfn(1)
```

```
4.0947265625
4.00000572205
```