Week 5

Sun Jun

with slides from Hans Petter Langtangen
A class packs together data (a collection of variables) and functions as *one single unit*

As a programmer you can create a new class and thereby a new object type (like `float`, `list`, `file`, ...)  

A class is much like a module: a collection of ”global” variables and functions that belong together

There is only one instance of a module while a class can have many instances (copies)

Modern programming applies classes to a large extent

It will take some time to master the class concept

Let’s learn by doing!
Consider a function of \( t \) with a parameter \( v_0 \):

\[
y(t; v_0) = v_0 t - \frac{1}{2}gt^2
\]

We need both \( v_0 \) and \( t \) to evaluate \( y \) (and \( g = 9.81 \))

How should we implement this?

```python
def y(t, v0):
g = 9.81
return v0*t - 0.5*g*t**2
```

# or \( v_0 \) global?

```python
def y(t):
g = 9.81
return v0*t - 0.5*g*t**2
```

It is best to have \( y \) as function of \( t \) only (\( y(t) \), see the book for a thorough discussion)

Two possibilities for \( y(t) \): \( v_0 \) as global variable (bad solution!) or \( y \) as a class (good solution!)
A class has variables and functions

Here: class $\mathcal{Y}$ for $y(t; v_0)$ has variables $v_0$ and $g$ and a function $\text{value}(t)$ for computing $y(t; v_0)$

Any class should also have a function $\_\text{init}\_\$ for initialization of the variables

A UML diagram of the class:

```
\begin{array}{|l|}
\hline
Y \\
\_\text{init}\_
\text{value} \\
g \\
v0 \\
\hline
\end{array}
```
Representing a function by a class; the code

The code:

class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2

Usage:

    y = Y(v0=3)  # create instance
    v = y.value(0.1)  # compute function value
When we write
\[ y = Y(v0=3) \]
we create a new variable (instance) \( y \) of type \( Y \)

\( Y(3) \) is a call to the constructor:
```python
def __init__(self, v0):
    self.v0 = v0
    self.g = 9.81
```

Think of \( self \) as \( y \), i.e., the new variable to be created –
\( self.v0 \) means that we attach a variable \( v0 \) to \( self \) (\( y \))
\( Y.__init__(y, 3) \)  # logic behind \( Y(3) \)

\( self \) is always first parameter in a function, but never inserted in the call

After \( y = Y(3) \), \( y \) has two variables \( v0 \) and \( g \), and we can do
```python
print y.v0
print y.g
```
Functions in classes are called **methods**

Variables in classes are called **attributes**

The `value` method:

```python
def value(self, t):
    return self.v0*t - 0.5*self.g*t**2
```

Example on a call:

```python
v = y.value(t=0.1)
```

`self` is left out in the call, but Python automatically inserts `y` as the `self` argument inside the `value` method.

Inside `value` things ”appear” as

```python
return y.v0*t - 0.5*y.g*t**2
```

The method `value` has, through `self` (here `y`), access to the attributes – attributes are like ”global variables” in the class, and any method gets a `self` parameter as first argument and can then access the attributes through `self`
Class \( Y \) collects the attributes \( v_0 \) and \( g \) and the method \( \text{value} \) as one unit

\( \text{value}(t) \) is function of \( t \) only, but has automatically access to the parameters \( v_0 \) and \( g \)

The great advantage: we can send \( y\.\text{value} \) as an ordinary function of \( t \) to any other function that expects a function \( f(t) \),

```python
def table(f, tstop, n):
    """Make a table of \( t \), \( f(t) \) values."""
    for t in linspace(0, tstop, n):
        print t, f(t)

def g(t):
    return sin(t)*exp(-t)

table(g, 2*pi, 101)  # send ordinary function

y = Y(6.5)
table(y.value, 2*pi, 101)  # send class method
```
Given a function with \( n + 1 \) parameters and one independent variable, 

\[
f(x; p_0, \ldots, p_n)
\]

it is smart to represent \( f \) by a class where \( p_0, \ldots, p_n \) are attributes and where there is a method, say `value(self, x)`, for computing \( f(x) \)

```python
class MyFunc:
    def __init__(self, p0, p1, p2, ..., pn):
        self.p0 = p0
        self.p1 = p1
        ...
        self.pn = pn

    def value(self, x):
        return ...
```
Representing a function by a class; another example

A function with four parameters:

\[ v(r; \beta, \mu_0, n, R) = \left( \frac{\beta}{2\mu_0} \right)^{\frac{1}{n}} \frac{n}{n+1} \left( R^{\frac{1}{n}} - r^{\frac{1}{n}} \right) \]

```python
class VelocityProfile:
    def __init__(self, beta, mu0, n, R):
        self.beta, self.mu0, self.n, self.R = beta, mu0, n, R

    def value(self, r):
        beta, mu0, n, R = self.beta, self.mu0, self.n, self.R
        n = float(n) # ensure float divisions
        v = (beta/(2.0*mu0))**(1/n)*(n/(n+1))*
        (R**(1+1/n) - r**(1+1/n))
        return v

v = VelocityProfile(R=1, beta=0.06, mu0=0.02, n=0.1)
print v.value(r=0.1)
```
class MyClass:
    def __init__(self, p1, p2):
        self.attr1 = p1
        self.attr2 = p2

    def method1(self, arg):
        # can init new attribute outside constructor:
        self.attr3 = arg
        return self.attr1 + self.attr2 + self.attr3

    def method2(self):
        print 'Hello!'

m = MyClass(4, 10)
print m.method1(-2)
m.method2()
Another class example: a bank account

- Attributes: name of owner, account number, balance
- Methods: deposit, withdraw, pretty print

```python
class Account:
    def __init__(self, name, account_number, initial_amount):
        self.name = name
        self.no = account_number
        self.balance = initial_amount

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def dump(self):
        s = '%s, %s, balance: %s' % 
            (self.name, self.no, self.balance)
        print s
```
### UML diagram of class Account

<table>
<thead>
<tr>
<th>Account</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td></td>
</tr>
<tr>
<td>deposit</td>
<td></td>
</tr>
<tr>
<td>withdraw</td>
<td></td>
</tr>
<tr>
<td>dump</td>
<td></td>
</tr>
<tr>
<td>balance</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
Example on using class Account

```python
>>> a1 = Account('John Olsson', '19371554951', 20000)
>>> a2 = Account('Liz Olsson', '19371564761', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a2.withdraw(10500)
>>> a1.withdraw(3500)
>>> print "a1’s balance: ", a1.balance
a1’s balance: 13500
>>> a1.dump()
John Olsson, 19371554951, balance: 13500
>>> a2.dump()
Liz Olsson, 19371564761, balance: 9500
```
Protected names for avoiding misuse

Possible, but not intended:

```python
>>> a1.name = 'Some other name'
>>> a1.balance = 100000
>>> a1.no = '19371564768'
```

The assumptions on correct usage:

- The attributes should *not* be changed!
- The `balance` attribute can be viewed
- Changing `balance` is done through `withdraw` or `deposit`

Remedy:

Attributes and methods not intended for use outside the class can be marked as *protected* by prefixing the name with an underscore (e.g., `_name`). This is just a convention – and no technical way of avoiding attributes and methods to be accessed.
Protected names for avoiding misuse

Possible, but not intended:

```python
>>> a1.name = 'Some other name'
>>> a1.balance = 100000
>>> a1.no = '19371564768'
```

The assumptions on correct usage:

- The attributes should not be changed!
- The `balance` attribute can be viewed
- Changing `balance` is done through `withdraw` or `deposit`

Remedy:

Attributes and methods not intended for use outside the class can be marked as `protected` by prefixing the name with an underscore (e.g., `_name`). This is just a convention – and no technical way of avoiding attributes and methods to be accessed.
### Protected names for avoiding misuse

#### Possible, but not intended:

```python
>>> a1.name = 'Some other name'
>>> a1.balance = 100000
>>> a1.no = '19371564768'
```

#### The assumptions on correct usage:

- The attributes should *not* be changed!
- The `balance` attribute can be viewed
- Changing `balance` is done through `withdraw` or `deposit`

#### Remedy:

Attributes and methods not intended for use outside the class can be marked as *protected* by prefixing the name with an underscore (e.g., `_name`). This is just a convention – and no technical way of avoiding attributes and methods to be accessed.
Possible, but not intended:

```python
>>> a1.name = 'Some other name'
>>> a1.balance = 100000
>>> a1.no = '19371564768'
```

The assumptions on correct usage:

- The attributes should *not* be changed!
- The `balance` attribute can be viewed
- Changing `balance` is done through `withdraw` or `deposit`

Remedy:

Attributes and methods not intended for use outside the class can be marked as *protected* by prefixing the name with an underscore (e.g., `_name`). This is just a convention – and no technical way of avoiding attributes and methods to be accessed.
class AccountP:
    def __init__(self, name, account_number, initial_amount):
        self._name = name
        self._no = account_number
        self._balance = initial_amount

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount

    def get_balance(self):  # NEW - read balance value
        return self._balance

    def dump(self):
        s = f'{self._name}, {self._no}, balance: {self._balance}'
        print(s)
Usage of improved class AccountP

```python
a1 = AccountP('John Olsson', '19371554951', 20000)
a1.withdraw(4000)

print a1._balance  # it works, but a convention is broken
print a1.get_balance()  # correct way of viewing the balance
a1._no = '19371554955'  # this is a "serious crime"!!!
```
Another example: a phone book

- Phone book: list of data about persons
- Data about a person: name, mobile phone, office phone, private phone, email
- Data about a person can be collected in a class as attributes
- Methods:
  - Constructor for initializing name, plus one or more other data
  - Add new mobile number
  - Add new office number
  - Add new private number
  - Add new email
  - Write out person data
### UML diagram of class Person

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
</tr>
<tr>
<td>add_mobile_phone</td>
</tr>
<tr>
<td>add_office_phone</td>
</tr>
<tr>
<td>add_private_phone</td>
</tr>
<tr>
<td>add_email</td>
</tr>
<tr>
<td>dump</td>
</tr>
<tr>
<td>email</td>
</tr>
<tr>
<td>mobile</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>office</td>
</tr>
<tr>
<td>private</td>
</tr>
</tbody>
</table>
class Person:
    def __init__(self, name,
                 mobile_phone=None, office_phone=None,
                 private_phone=None, email=None):
        self.name = name
        self.mobile = mobile_phone
        self.office = office_phone
        self.private = private_phone
        self.email = email

    def add_mobile_phone(self, number):
        self.mobile = number

    def add_office_phone(self, number):
        self.office = number

    def add_private_phone(self, number):
        self.private = number

    def add_email(self, address):
        self.email = address
Code of class Person

def dump(self):
    s = self.name + '\n'
    if self.mobile is not None:
        s += 'mobile phone: %s\n' % self.mobile
    if self.office is not None:
        s += 'office phone: %s\n' % self.office
    if self.private is not None:
        s += 'private phone: %s\n' % self.private
    if self.email is not None:
        s += 'email address: %s\n' % self.email
    print s

Usage:

p1 = Person('Hans Petter Langtangen', email='hpl@simula.no')
p1.add_office_phone('67828283'),
p2 = Person('Aslak Tveito', office_phone='67828282')
p2.add_email('aslak@simula.no')
phone_book = [p1, p2]  # list
phone_book = {'Langtangen': p1, 'Tveito': p2}  # better
for p in phone_book:
    p.dump()
Another example: a circle

- A circle is defined by its center point $x_0$, $y_0$ and its radius $R$
- These data can be attributes in a class
- Possible methods in the class: area, circumference
- The constructor initializes $x_0$, $y_0$ and $R$

```python
class Circle:
    def __init__(self, x0, y0, R):
        self.x0, self.y0, self.R = x0, y0, R

    def area(self):
        return pi*self.R**2

    def circumference(self):
        return 2*pi*self.R
```

```python
>>> c = Circle(2, -1, 5)
>>> print 'A circle with radius %g at (%g, %g) has area %g'
      ... (c.R, c.x0, c.y0, c.area())
A circle with radius 5 at (2, -1) has area 78.5398
```
Some class methods have leading and trailing double underscores

```python
def __init__(self, ...)
def __call__(self, ...)
def __add__(self, other)
```

These are called *special methods* and allow for special syntax

Recall the constructor, we write

```python
y = Y(4)
```

and not (the more logical)

```python
Y.__init__(y, 4)
```

With the `__call__` special method we can make the class instance behave and look as a function

With the `__add__` special method we can add two class instances (with our own tailored rule for addition)

Many forthcoming examples illustrate various special methods
Let us replace the `value` method in class `Y` by a `call` special method:

```python
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

Now we can write

```python
y = Y(3)
v = y(0.1)  # same as v = y.__call__(0.1)
```

The instance `y` behaves/looks as a function!

The `value(t)` method does the same, but the special method `(call)` allows nicer syntax for computing function values.
Given a function with \( n + 1 \) parameters and one independent variable,

\[ f(x; p_0, \ldots, p_n) \]

it is wise to represent \( f \) by a class where \( p_0, \ldots, p_n \) are attributes and where there is a call special method for computing \( f(x) \)

```python
class MyFunc:
    def __init__(self, p0, p1, p2, ..., pn):
        self.p0 = p0
        self.p1 = p1
        ...
        self.pn = pn

    def __call__(self, x):
        return ...
```
Given some mathematical function in Python, say

```python
def f(x):
    return x**3
```

can we make a class `Derivative` and write

```python
dfdx = Derivative(f)
```

so that `dfdx` will behave as a function that computes the derivative of `f`?

```python
print dfdx(2)  # computes 3*x**2 for x=2
```

Yes – this is possible using classes
We use numerical differentiation "behind the curtain":

\[
f'(x) \approx \frac{f(x + h) - f(x)}{h}
\]

for a small \( h \), say \( h = 10^{-9} \)

Class `Derivative` stores \( f \) and \( h \) as attributes and applies the differentiation formula in the call special method:

```python
class Derivative:
    def __init__(self, f, h=1E-9):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h  # make short forms
        return (f(x+h) - f(x))/h
```
```python
>>> from math import *
>>> df = Derivative(sin)
>>> x = pi
>>> df(x)
-1.000000082740371
>>> cos(x)  # exact
-1.0
>>> def g(t):
...     return t**3
...     return t**3
>>> dg = Derivative(g)
>>> t = 1
>>> dg(t)  # compare with 3 (exact)
3.000000248221113
```
Newton’s method solves nonlinear equations $f(x) = 0$, but the method requires $f'(x)$

Suppose we have an implementation

```python
def Newton(f, xstart, dfdx, epsilon=1E-9):
    ...
    return x, no_of_iterations, f(x)
```

Suppose we have a function $f(x)$ that we do not want to differentiate – then class `Derivative` is handy:

```python
>>> def f(x):
...     return 100000*(x - 0.9)**2 * (x - 1.1)**3
...
>>> df = Derivative(f)
>>> xstart = 1.01
>>> Newton(f, xstart, df, epsilon=1E-5)
(1.0987610068093443, 8, -7.5139644257961411e-06)```
Given a function $f(x)$, we want to compute

$$F(x; a) = \int_{a}^{x} f(t)dt$$

**Technique: Trapezoidal rule**

$$\int_{a}^{x} f(t)dt = h \left( \frac{1}{2} f(a) + \sum_{i=1}^{n-1} f(a + ih) + \frac{1}{2} f(x) \right)$$

This is the application code we want:

```python
def f(x):
    return exp(-x**2)*sin(10*x)

a = 0; n = 200
F = Integral(f, a, n)
x = 1.2
print F(x)
```
Let us first make a separate integration function:

```
from scitools.std import iseq

def trapezoidal(f, a, x, n):
    h = (x-a)/float(n)
    I = 0.5*f(a)
    for i in iseq(1, n-1):
        I += f(a + i*h)
    I += 0.5*f(x)
    I *= h
    return I
```

Class `Integral` holds $f$, $a$ and $n$ as attributes and has a special method call for computing the integral:

```
class Integral:
    def __init__(self, f, a, n=100):
        self.f, self.a, self.n = f, a, n

    def __call__(self, x):
        return trapezoidal(self.f, self.a, x, self.n)
```
In Python, we can usually print an object `a` by `print a`.

This works for built-in types (strings, lists, floats, ...)

If we have made a new type through a class, Python does not know how to print objects of this type.

However, if the class has defined a method `__str__`, Python will use this method to convert the object to a string.

```python
class Y:
    ...
    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2

    def __str__(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

Demo:

```python
>>> y = Y(1.5)
>>> y(0.2)
0.1038
>>> print y
v0*t - 0.5*g*t**2; v0=1.5
```
A polynomial can be specified by a list of its coefficients.

Example: \([1,0,-1,2]\) corresponds to

\[
1 + 0 \cdot x - 1 \cdot x^2 + 2 \cdot x^3 = 1 - x^2 + 2x^3
\]

i.e., elem. no. \(i\) in the list is the coefficient for the \(x^i\) term.

We want the following application code:

```python
>>> p1 = Polynomial([1, -1])
>>> print p1
1 - x

>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = p1 + p2
>>> print p3.coeff
[1, 0, 0, 0, -6, -1]
>>> print p3
1 - 6x^4 - x^5

>>> p2.differentiate()
>>> print p2
1 - 24x^3 - 5x^4
```

How can we make class Polynomial?
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # return self + other
        # start with the longest list and add in the other:
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:]  # copy!
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]
        else:
            coeffsum = other.coeff[:]  # copy!
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]
        return Polynomial(coeffsum)
Mathematics:

Multiplication of two general polynomials:

\[
\left( \sum_{i=0}^{M} c_i x^i \right) \left( \sum_{j=0}^{N} d_j x^j \right) = \sum_{i=0}^{M} \sum_{j=0}^{N} c_i d_j x^{i+j}
\]

The coeff. corresponding to power \( i + j \) is \( c_i \cdot d_j \). If \( r \) is the list representation of the result, we then have \( r[i+j] = c[i] \cdot d[j] \) (\( i \) and \( j \) running from 0 to \( M \) and \( N \), resp.)

Implementation:

```python
class Polynomial:
    ...
    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0] * (M + N + 1)  # or zeros(M+N+1)
        for i in range(0, M + 1):
            for j in range(0, N + 1):
                coeff[i+j] += self.coeff[i] * other.coeff[j]
        return Polynomial(coeff)
```
Mathematics:

Rule for differentiating a general polynomial:

\[
\frac{d}{dx} \sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} ic_i x^{i-1}
\]

If \( c \) is the list of coefficients, the derivative has a list of coefficients, \( dc \), where \( dc[i-1] = i*c[i] \) for \( i \) running from 1 to the largest index in \( c \). Note that \( dc \) has one element less than \( c \).

Implementation:

```python
class Polynomial:
    ...
    def differentiate(self):  # change self
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i*self.coeff[i]
        del self.coeff[-1]

    def derivative(self):  # return new polynomial
        dpdx = Polynomial(self.coeff[:])  # copy
        dpdx.differentiate()
        return dpdx
```
class Polynomial:
...

def __str__(self):
    s = ''
    for i in range(0, len(self.coeff)):
        if self.coeff[i] != 0:
            s += ' + %g*x^%d' % (self.coeff[i], i)
    # fix layout:
    s = s.replace('+ -', ' - ')
    s = s.replace(' 1*', ' ')
    s = s.replace('x^0', '1')
    s = s.replace('x^1 ', 'x ')
    s = s.replace('x^1', 'x')
    if s[0:3] == ' + ':  # remove initial +
        s = s[3:]
    if s[0:3] == ' - ':  # fix spaces for initial -
        s = ' - ' + s[3:]
    return s
Consider

\[ p_1(x) = 1 - x, \quad p_2(x) = x - 6x^4 - x^5 \]

and their sum

\[ p_3(x) = p_1(x) + p_2(x) = 1 - 6x^4 - x^5 \]

```python
>>> p1 = Polynomial([1, -1])
>>> print p1
1 - x
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = p1 + p2
>>> print p3.coeff
[1, 0, 0, 0, -6, -1]
>>> p2.differentiate()
>>> print p2
1 - 24*x^3 - 5*x^4
```
Special methods for arithmetic operations

\[ c = a + b \quad \# \quad c = a.__add__(b) \]
\[ c = a - b \quad \# \quad c = a.__sub__(b) \]
\[ c = a \times b \quad \# \quad c = a.__mul__(b) \]
\[ c = a/b \quad \# \quad c = a.__div__(b) \]
\[ c = a**e \quad \# \quad c = a.__pow__(e) \]
Special methods for comparisons

\[
\begin{align*}
  a &= b & \# & a.__eq__(b) \\
  a &=& b & \# & a.__ne__(b) \\
  a &<& b & \# & a.__lt__(b) \\
  a &<= b & \# & a.__le__(b) \\
  a &> b & \# & a.__gt__(b) \\
  a &>= b & \# & a.__ge__(b)
\end{align*}
\]
Some mathematical operations for vectors in the plane:

\[(a, b) + (c, d) = (a + c, b + d)\]
\[(a, b) - (c, d) = (a - c, b - d)\]
\[(a, b) \cdot (c, d) = ac + bd\]
\[(a, b) = (c, d) \text{ if } a = c \text{ and } b = d\]

Desired application code:

```python
>>> u = Vec2D(0,1)
>>> v = Vec2D(1,0)
>>> print u + v
(1, 1)
>>> a = u + v
>>> w = Vec2D(1,1)
>>> a == w
True
>>> print u - v
(-1, 1)
>>> print u*v
0
```
class Vec2D:
    def __init__(self, x, y):
        self.x = x; self.y = y

    def __add__(self, other):
        return Vec2D(self.x+other.x, self.y+other.y)

    def __sub__(self, other):
        return Vec2D(self.x-other.x, self.y-other.y)

    def __mul__(self, other):
        return self.x*other.x + self.y*other.y

    def __abs__(self):
        return math.sqrt(self.x**2 + self.y**2)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)

    def __ne__(self, other):
        return not self.__eq__(other) # reuse __eq__
Two special methods for turning an object into a string:

```python
class MyClass:
    def __init__(self, a, b):
        self.a, self.b = a, b

    def __str__(self):
        """Return string with pretty print."""
        return 'a=%s, b=%s' % (self.a, self.b)

    def __repr__(self):
        """Return s such that eval(s) recreates self."""
        return 'MyClass(%s, %s)' % (self.a, self.b)

Demo:

```python
>>> m = MyClass(1, 5)
>>> print m  # calls m.__str__()
a=1, b=5
>>> str(m)   # calls m.__str__()
'a=1, b=5'
>>> s = repr(m)  # calls m.__repr__()
'MyClass(1, 5)'
>>> m2 = eval(s)
>>> m2      # calls m.__repr__()
'MyClass(1, 5)'
```
class Y:
    """Class for function y(t; v0, g) = v0*t - 0.5*g*t**2."""

    def __init__(self, v0):
        """Store parameters."""
        self.v0 = v0
        self.g = 9.81

    def __call__(self, t):
        """Evaluate function."""
        return self.v0*t - 0.5*self.g*t**2

    def __str__(self):
        """Pretty print."""
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0

    def __repr__(self):
        """Print code for regenerating this instance."""
        return 'Y(\%s)' % self.v0
Python already has a class `complex` for complex numbers, but implementing such a class is a good pedagogical example on class programming (especially with special methods).

Here is what we would like to do:

```python
>>> u = Complex(2,-1)
>>> v = Complex(1)  # zero imaginary part
>>> w = u + v
>>> print w
(3, -1)
>>> w != u
True
>>> u*v
Complex(2, -1)
>>> u < v
illegal operation "<" for complex numbers
>>> print w + 4
(7, -1)
>>> print 4 - w
(1, 1)
```
class Complex:
    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.imag)

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

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

    def __div__(self, other):
        ar, ai, br, bi = self.real, self.imag, other.real, other.imag,
        r = float(br**2 + bi**2)
        return Complex((ar*br+ai*bi)/r, (ai*br-ar*bi)/r)
def __abs__(self):
    return sqrt(self.real**2 + self.imag**2)

def __neg__(self):    # defines -c (c is Complex)
    return Complex(-self.real, -self.imag)

def __eq__(self, other):
    return self.real == other.real and \
    self.imag == other.imag

def __ne__(self, other):
    return not self.__eq__(other)

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

def __repr__(self):
    return 'Complex' + str(self)

def __pow__(self, power):
    raise NotImplementedError\
    ('self**power is not yet impl. for Complex')
Can we add a real number to a complex number?

>>> u = Complex(1, 2)
>>> w = u + 4.5
...
AttributeError: 'float' object has no attribute 'real'

Problem: we have assumed that other is Complex

Remedy:

```python
def __add__(self, other):
    if isinstance(other, (float, int)):
        other = Complex(other)
    return Complex(self.real + other.real,
                   self.imag + other.imag)
```

# or

```python
def __add__(self, other):
    if isinstance(other, (float, int)):
        return Complex(self.real + other, self.imag)
    else:
        return Complex(self.real + other.real,
                        self.imag + other.imag)
```
What if we try this:

```python
>>> u = Complex(1, 2)
>>> w = 4.5 + u
...
TypeError: unsupported operand type(s) for +: 'float' and 'instance'
```

Problem: Python’s `float` objects cannot add a `Complex`

Remedy: if a class has a `__radd__(self, other)` special method, Python applies this for the expression `other + self`

```python
def __radd__(self, other):
    """Defines other + self."""
    # other + self = self + other:
    return self.__add__(other)
```
Right operands for subtraction is a bit more complicated since $a - b \neq b - a$:

```python
def __sub__(self, other):
    if isinstance(other, (float, int)):
        other = Complex(other)
    return Complex(self.real - other.real, self.imag - other.imag)

def __rsub__(self, other):
    if isinstance(other, (float, int)):
        other = Complex(other)
    return other.__sub__(self)
```
What’s in a class?

- A demo class:

  ```python
  class A:
      """A class for demo purposes.""
      def __init__(self, value):
          self.v = value
  ```

- Any instance holds its attributes in the `self.__dict__` dictionary (Python automatically creates this dict)

  ```python
  >>> a = A([1,2])
  >>> print a.__dict__  # all attributes
  {'v': [1, 2]}
  >>> dir(a)        # what's in object a?
  '__doc__', '__init__', '__module__', 'dump', 'v']
  >>> a.__doc__    # programmer's documentation of A
  'A class for demo purposes.'
  ```
>>> a.myvar = 10               # add new attribute (!)
>>> a.__dict__
{'myvar': 10, 'v': [1, 2]}
>>> dir(a)
['__doc__', '__init__', '__module__', 'dump', 'myvar', 'v']

>>> b = A(-1)
>>> b.__dict__                # b has no myvar attribute
{'v': -1}
>>> dir(b)
['__doc__', '__init__', '__module__', 'dump', 'v']
Example on defining a class with attributes and methods:

class Gravity:
    """Gravity force between two objects."""
    def __init__(self, m, M):
        self.m = m
        self.M = M
        self.G = 6.67428E-11 # gravity constant

    def force(self, r):
        G, m, M = self.G, self.m, self.M
        return G*m*M/r**2

    def visualize(self, r_start, r_stop, n=100):
        from scitools.std import plot, linspace
        r = linspace(r_start, r_stop, n)
        g = self.force(r)
        title='m=%g, M=%g' % (self.m, self.M)
        plot(r, g, title=title)
Example on using the class:

```python
mass_moon = 7.35E+22
mass_earth = 5.97E+24

# make instance of class Gravity:
gravity = Gravity(mass_moon, mass_earth)

r = 3.85E+8  # earth-moon distance in meters
Fg = gravity.force(r)  # call class method
```
c = a + b implies
   c = a.__add__(b)

There are special methods for a+b, a−b, a*b, a/b, a**b, −a, if
   a:, len(a), str(a) (pretty print), repr(a) (recreate a with
   eval), etc.

With special methods we can create new mathematical
objects like vectors, polynomials and complex numbers and
write ”mathematical code” (arithmetics)

The call special method is particularly handy:
   c = C()
   v = c(5) # means v = c.__call__(5)

Functions with parameters should be represented by a class
with the parameters as attributes and with a call special
method for evaluating the function
Before we start: define the heading of this chapter

Object-oriented programming (OO) means different things to different people:
- programming with classes (better: object-based programming)
- programming with class hierarchies (class families)

The 2nd def. is most widely accepted and used here.
What is a class hierarchy?
A family of closely related classes
A key concept is *inheritance*: child classes can inherit attributes and methods from parent class(es) – this saves much typing and code duplication
As usual, we shall learn through examples

OO is a Norwegian invention – one of the most important inventions in computer science, because OO is used in all big computer systems today
Warnings: OO is difficult and takes time to master

- The OO concept might be difficult to understand
- Let ideas mature with time and try to work with it
- OO is less important in Python than in C++, Java and C#, so the benefits of OO are less obvious in Python
- Our examples here on OO employ numerical methods for differentiation, integration og ODEs – make sure you understand the simplest of these numerical methods before you study the combination of OO and numerics
- Ambitions: write simple OO code and understand how to make use of ready-made OO modules
Let us make a class for evaluating lines $y = c_0 + c_1x$

class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ''
        for x in linspace(L, R, n):
            y = self(x)
            s += '%12g %12g
' % (x, y)
        return s
Let us make a class for evaluating parabolas \( y = c_0 + c_1 x + c_2 x^2 \)

class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Return a table with n points for L \leq x \leq R."""
        s = ''
        for x in linspace(L, R, n):
            y = self(x)
            s += '%12g %12g
' % (x, y)
        return s

This is almost the same code as class Line, except for the things with \( c_2 \)
Parabola code = Line code + a little extra with the \( c_2 \) term

Can we utilize class `Line` code in class `Parabola`?

This is what inheritance is about!

Writing

```python
class Parabola(Line):
    pass
```

makes `Parabola` inherit all methods and attributes from `Line`, so `Parabola` has attributes \( c_0 \) and \( c_1 \) and three methods

`Line` is a `superclass`, `Parabola` is a `subclass`

(parent class, base class; child class, derived class)

Class `Parabola` must add code to `Line`'s constructor (an extra \( c_2 \) attribute), `__call__` (an extra term), but `table` can be used unaltered

The principle is to reuse as much code in `Line` as possible and avoid duplicating code
Parabola code = Line code + a little extra with the $c_2$ term

Can we utilize class Line code in class Parabola?

This is what inheritance is about!

Writing

```python
class Parabola(Line):
    pass
```

makes Parabola inherit all methods and attributes from Line, so Parabola has attributes $c_0$ and $c_1$ and three methods

Line is a superclass, Parabola is a subclass
(parent class, base class; child class, derived class)

Class Parabola must add code to Line's constructor (an extra $c_2$ attribute), `__call__` (an extra term), but `table` can be used unaltered

The principle is to reuse as much code in Line as possible and avoid duplicating code
Parabola code = Line code + a little extra with the $c_2$ term

Can we utilize class Line code in class Parabola?

This is what inheritance is about!

Writing

```python
class Parabola(Line):
    pass
```

makes Parabola inherit all methods and attributes from Line, so Parabola has attributes $c_0$ and $c_1$ and three methods.

Line is a superclass, Parabola is a subclass (parent class, base class; child class, derived class)

Class Parabola must add code to Line's constructor (an extra $c_2$ attribute), `__call__` (an extra term), but `table` can be used unaltered.

The principle is to reuse as much code in Line as possible and avoid duplicating code.
Parabola code = Line code + a little extra with the $c_2$ term

Can we utilize class Line code in class Parabola?

This is what inheritance is about!

Writing

```python
class Parabola(Line):
    pass
```

makes Parabola inherit all methods and attributes from Line, so Parabola has attributes $c_0$ and $c_1$ and three methods

Line is a superclass, Parabola is a subclass (parent class, base class; child class, derived class)

Class Parabola must add code to Line’s constructor (an extra $c_2$ attribute), `__call__` (an extra term), but `table` can be used unaltered

The principle is to reuse as much code in Line as possible and avoid duplicating code
Class Parabola as a subclass of Line; principles

- Parabola code = Line code + a little extra with the $c_2$ term
- Can we utilize class Line code in class Parabola?
- This is what inheritance is about!
- Writing
  ```python
class Parabola(Line):
    pass
  ```
  makes Parabola inherit all methods and attributes from Line, so Parabola has attributes $c_0$ and $c_1$ and three methods

- Line is a superclass, Parabola is a subclass (parent class, base class; child class, derived class)
- Class Parabola must add code to Line’s constructor (an extra $c_2$ attribute), `__call__` (an extra term), but `table` can be used unaltered
- The principle is to reuse as much code in Line as possible and avoid duplicating code
Parabola code = Line code + a little extra with the $c_2$ term

Can we utilize class Line code in class Parabola?

This is what inheritance is about!

Writing

```python
class Parabola(Line):
    pass
```

makes Parabola inherit all methods and attributes from Line, so Parabola has attributes $c_0$ and $c_1$ and three methods

Line is a superclass, Parabola is a subclass

(parent class, base class; child class, derived class)

Class Parabola must add code to Line’s constructor (an extra $c_2$ attribute), `__call__` (an extra term), but table can be used unaltered

The principle is to reuse as much code in Line as possible and avoid duplicating code
Parabola code = Line code + a little extra with the $c_2$ term

Can we utilize class Line code in class Parabola?

This is what inheritance is about!

Writing

```python
class Parabola(Line):
    pass
```

makes Parabola inherit all methods and attributes from Line, so Parabola has attributes $c_0$ and $c_1$ and three methods.

Line is a superclass, Parabola is a subclass (parent class, base class; child class, derived class)

Class Parabola must add code to Line’s constructor (an extra $c_2$ attribute), `__call__` (an extra term), but `table` can be used unaltered

The principle is to reuse as much code in Line as possible and avoid duplicating code.
Parabola code = Line code + a little extra with the $c_2$ term

Can we utilize class Line code in class Parabola?

This is what inheritance is about!

Writing

```python
class Parabola(Line):
    pass
```

makes Parabola inherit all methods and attributes from Line, so Parabola has attributes $c_0$ and $c_1$ and three methods

- Line is a *superclass*, Parabola is a *subclass* (parent class, base class; child class, derived class)
- Class Parabola must add code to Line’s constructor (an extra $c_2$ attribute), `__call__` (an extra term), but `table` can be used unaltered
- The principle is to reuse as much code in Line as possible and avoid duplicating code
A subclass method can call a superclass method in this way:

```python
superclass_name.method(self, arg1, arg2, ...)
```

**Class Parabola as a subclass of Line:**

```python
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)  # Line stores c0, c1
        self.c2 = c2

    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2
```

**What is gained?** Class `Parabola` just adds code to the already existing code in class `Line` — no duplication of storing `c0` and `c1`, and computing `c0 + c1 * x`

**Class Parabola also has a `table` method — it is inherited**

**The constructor and `__call__` are overridden**, meaning that the subclass redefines these methods (here: redefined method = call superclass method + something extra)
Demo of class Parabola

Main program:
```python
p = Parabola(1, -2, 2)
p1 = p(2.5)
print p1
print p.table(0, 1, 3)
```

Output:
```
8.5
0 1
0.5 0.5
1 1
```

Follow the program flow of p(2.5)!
Checking class types and class relations

```python
>>> l = Line(-1, 1)
>>> isinstance(l, Line)
True
>>> isinstance(l, Parabola)
False

>>> p = Parabola(-1, 0, 10)
>>> isinstance(p, Parabola)
True
>>> isinstance(p, Line)
True

>>> issubclass(Parabola, Line)
True
>>> issubclass(Line, Parabola)
False

>>> p.__class__ == Parabola
True
>>> p.__class__.__name__ # string version of the class name
'Parabola'
```
Suppose we want to implement mathematical functions $f(x; p_1, \ldots, p_n)$ with parameters as classes. These classes should also support computation of $df/dx$ and $d^2f/dx^2$.

Example on such a class:

```python
class SomeFunc:
    def __init__(self, parameter1, parameter2, ...):
        # store parameters
    def __call__(self, x):
        # evaluate function
    def df(self, x):
        # evaluate the first derivative
    def ddf(self, x):
        # evaluate the second derivative
```

If we do not bother to derive the analytical derivatives, we can use numerical differentiation formulas in $df$ and $ddf$. 

Observation: with numerical differentiation, all such classes contain the same $df$ and $ddf$ methods

Such duplication of code is considered a bad thing

Better: collect $df$ and $ddf$ in a superclass and let subclasses automatically inherit these methods
class FuncWithDerivatives:
    def __init__(self, h=1.0E-5):
        self.h = h  # spacing for numerical derivatives
    def __call__(self, x):
        raise NotImplementedError('___call__ missing')
    def df(self, x):
        # compute 1st derivative by a finite difference:
        h = float(self.h)  # short form
        return (self(x+h) - self(x-h))/(2*h)
    def ddf(self, x):
        # compute 2nd derivative by a finite difference:
        h = float(self.h)  # short form
        return (self(x+h) - 2*self(x) + self(x-h))/h**2

The superclass is not useful in itself – a subclass is needed to implement a specific mathematical function
How to implement a specific function as a subclass

- Inherit from superclass `FuncWithDerivatives`
- Store parameters in constructor
- Implement function formula in `__call__`
- Rely on inherited `df` and `ddf` for numerical derivatives, or reimplement `df` and `ddf` with exact expressions
Say we want to implement \( f(x) = \cos(ax) + x^3 \)

We do this in a subclass:

```python
class MyFunc(FuncWithDerivatives):
    def __init__(self, a):
        self.a = a

    def __call__(self, x):
        return cos(self.a*x) + x**3

    def df(self, x):
        a = self.a
        return -a*sin(a*x) + 3*x**2

    def ddf(self, x):
        a = self.a
        return -a*a*cos(a*x) + 6*x
```

This subclass inherits from the superclass, but does not make use of anything from the superclass – no practical use of the superclass in this example.
Say we want to implement \( f(x) = \ln |p \tanh(qx \cos rx)| \)

We are lazy and want to avoid hand derivation of long expressions for \( f'(x) \) and \( f''(x) \) – use finite differences instead

Implementation as a subclass:

```python
class MyComplicatedFunc(FuncWithDerivatives):
    def __init__(self, p, q, r, h=1.0E-9):
        FuncWithDerivatives.__init__(self, h)
        self.p, self.q, self.r = p, q, r

def __call__(self, x):
    p, q, r = self.p, self.q, self.r
    return log(abs(p*tanh(q*x*con(r*x))))
```

This time we inherit \( df \) and \( ddf \)

Note also that we must pass \( h \) on to the superclass constructor

It is always a good habit to call the superclass constructor
Interactive example

\[ f(x) = \ln |p \tanh(qx \cos rx)| \]
compute \( f(x) \), \( f'(x) \), \( f''(x) \) for \( x = \pi/2 \)

```python
>>> from math import *
>>> f = MyComplicatedFunc(1, 1, 1)
>>> x = pi/2
>>> f(x)
-36.880306514638988
>>> f.df(x)
-60.593693618216086
>>> f.ddf(x)
3.3217246931444789e+19
```
Recall the class for numerical differentiation (Ch. 9)

Simplest numerical differentiation formula:

\[
f'(x) \approx \frac{f(x + h) - f(x)}{h}
\]

for a small \( h \), say \( h = 10^{-9} \)

Class `Derivative` stores \( f \) and \( h \) as attributes and applies the differentiation formula in the call special method

```python
class Derivative:
    def __init__(self, f, h=1E-5):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h  # make short forms
        return (f(x+h) - f(x))/h
```
There are numerous formulas for numerical differentiation:

\[
f'(x) = \frac{f(x + h) - f(x)}{h} + O(h)
\]

\[
f'(x) = \frac{f(x) - f(x - h)}{h} + O(h)
\]

\[
f'(x) = \frac{f(x + h) - f(x - h)}{2h} + O(h^2)
\]

\[
f'(x) = \frac{4}{3} \frac{f(x + h) - f(x - h)}{2h} - \frac{1}{3} \frac{f(x + 2h) - f(x - 2h)}{4h} + O(h^4)
\]

\[
f'(x) = \frac{3}{2} \frac{f(x + h) - f(x - h)}{2h} - \frac{3}{5} \frac{f(x + 2h) - f(x - 2h)}{4h} +
\]

\[
\begin{align*}
&\frac{1}{10} \frac{f(x + 3h) - f(x - 3h)}{6h} + O(h^6) \\
&f'(x) = \frac{1}{h} \left( -\frac{1}{6} f(x + 2h) + f(x + h) - \frac{1}{2} f(x) - \frac{1}{3} f(x - h) \right) + O(h^3)
\end{align*}
\]
Why should these formulas be implemented in a class hierarchy?

- A numerical differentiation formula can be implemented as a class: $h$ and $f$ are attributes and a call special method evaluates the formula.
- The `Derivative` class appears as a plain function in use.
- All classes for different differentiation formulas are similar: the constructor is the same, only `__call__` differs.
- Code duplication (of the constructors) is a bad thing (=rule!)
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we let a superclass implement the common constructor.
- Subclasses extend the superclass with code specific to a math formula.
Why should these formulas be implemented in a class hierarchy?

- A numerical differentiation formula can be implemented as a class: \( h \) and \( f \) are attributes and a call special method evaluates the formula
- The \texttt{Derivative} class appears as a plain function in use
- All classes for different differentiation formulas are similar: the constructor is the same, only \texttt{__call__} differs
- Code duplication (of the constructors) is a bad thing (=rule!)
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we let a superclass implement the common constructor
- Subclasses extend the superclass with code specific to a math formula
Why should these formulas be implemented in a class hierarchy?

- A numerical differentiation formula can be implemented as a class: $h$ and $f$ are attributes and a call special method evaluates the formula.
- The Derivative class appears as a plain function in use.
- All classes for different differentiation formulas are similar: the constructor is the same, only `__call__` differs.
- Code duplication (of the constructors) is a bad thing (=rule!).
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we let a superclass implement the common constructor.
- Subclasses extend the superclass with code specific to a math formula.
Why should these formulas be implemented in a class hierarchy?

- A numerical differentiation formula can be implemented as a class: \( h \) and \( f \) are attributes and a call special method evaluates the formula.
- The Derivative class appears as a plain function in use.
- All classes for different differentiation formulas are similar: the constructor is the same, only \_\_call\_\_ differs.
- Code duplication (of the constructors) is a bad thing (=rule!)
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we let a superclass implement the common constructor.
- Subclasses extend the superclass with code specific to a math formula.
Why should these formulas be implemented in a class hierarchy?

- A numerical differentiation formula can be implemented as a class: $h$ and $f$ are attributes and a call special method evaluates the formula.
- The `Derivative` class appears as a plain function in use.
- All classes for different differentiation formulas are similar: the constructor is the same, only `__call__` differs.
- Code duplication (of the constructors) is a bad thing (=rule!)

  - The general OO idea: place code common to many classes in a superclass and inherit that code – here we let a superclass implement the common constructor.
  - Subclasses extend the superclass with code specific to a math formula.
A numerical differentiation formula can be implemented as a class: \( h \) and \( f \) are attributes and a call special method evaluates the formula.

The `Derivative` class appears as a plain function in use.

All classes for different differentiation formulas are similar: the constructor is the same, only `__call__` differs.

Code duplication (of the constructors) is a bad thing (=rule!)

The general OO idea: place code common to many classes in a superclass and inherit that code – here we let a superclass implement the common constructor.

Subclasses extend the superclass with code specific to a math formula.
Why should these formulas be implemented in a class hierarchy?

- A numerical differentiation formula can be implemented as a class: $h$ and $f$ are attributes and a call special method evaluates the formula.
- The `Derivative` class appears as a plain function in use.
- All classes for different differentiation formulas are similar: the constructor is the same, only `__call__` differs.
- Code duplication (of the constructors) is a bad thing (=rule!)
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we let a superclass implement the common constructor.
- Subclasses extend the superclass with code specific to a math formula.
Superclass:

```python
class Diff:
    def __init__(self, f, h=1E-5):
        self.f = f
        self.h = float(h)
```

Subclass for simple forward formula:

```python
class Forward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
```

Subclass for 4-th order central formula:

```python
class Central4(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (4./3)*(f(x+h) - f(x-h)) / (2*h) -
Interactive example: \( f(x) = \sin x \), compute \( f'(x) \) for \( x = \pi \)

```python
>>> from Diff import *
>>> from math import sin
>>> mycos = Central4(sin)
>>> # compute \( \sin'(\pi) \):
... mycos(pi)
-1.000000082740371
```

Note: `Central4(sin)` calls inherited constructor in superclass, while `mycos(pi)` calls `__call__` in the subclass `Central4`
Suppose we want to differentiate function expressions from the command line:

Unix/DOS> python df.py 'exp(sin(x))' Central 2 3.1 -1.04155573055

Unix/DOS> python df.py f(x) difftype difforder x f’(x)

With `eval` and the `Diff` class hierarchy this main program can be realized in a few lines (many lines in C# and Java!):

```python
import sys
from Diff import *
from math import *
from scitools.StringFunction import StringFunction

f = StringFunction(sys.argv[1])
difftype = sys.argv[2]
difforder = sys.argv[3]
classname = difftype + difforder
df = eval(classname + '(f)')
x = float(sys.argv[4])
print df(x)
```
Suppose we want to differentiate function expressions from the command line:

Unix/DOS> python df.py 'exp(sin(x))' Central 2 3.1 -1.04155573055

Unix/DOS> python df.py f(x) difftype difforder x f’(x)

With `eval` and the `Diff` class hierarchy this main program can be realized in a few lines (many lines in C# and Java!):

```python
import sys
from Diff import *
from math import *
from scitools.StringFunction import StringFunction

f = StringFunction(sys.argv[1])
difftype = sys.argv[2]
difforder = sys.argv[3]
classname = difftype + difforder
df = eval(classname + '(f)')
x = float(sys.argv[4])
print df(x)
```
Suppose we want to differentiate function expressions from the command line:

Unix/DOS> python df.py 'exp(sin(x))' Central 2 3.1 -1.04155573055

Unix/DOS> python df.py f(x) difftype difforder x f'(x)

With `eval` and the `Diff` class hierarchy this main program can be realized in a few lines (many lines in C# and Java!):

```python
import sys
from Diff import *
from math import *
from scitools.StringFunction import StringFunction
import string

f = StringFunction(sys.argv[1])
difftype = sys.argv[2]
diffforder = sys.argv[3]
classname = difftype + diffforder
df = eval(classname + '(f)')
x = float(sys.argv[4])
print df(x)
```
We can empirically investigate the accuracy of our family of 6 numerical differentiation formulas.

Sample function: \( f(x) = \exp(-10x) \)

See the book for a little program that computes the errors:

<table>
<thead>
<tr>
<th>h</th>
<th>Forward1</th>
<th>Central2</th>
<th>Central4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.25E-02</td>
<td>-2.56418286E+00</td>
<td>6.63876231E-01</td>
<td>-5.32825724E-02</td>
</tr>
<tr>
<td>3.12E-02</td>
<td>-1.41170013E+00</td>
<td>1.63556996E-01</td>
<td>-3.21608292E-03</td>
</tr>
<tr>
<td>1.56E-02</td>
<td>-7.42100948E-01</td>
<td>4.07398036E-02</td>
<td>-1.99260429E-04</td>
</tr>
<tr>
<td>7.81E-03</td>
<td>-3.80648092E-01</td>
<td>1.01756309E-01</td>
<td>-1.24266603E-05</td>
</tr>
<tr>
<td>3.91E-03</td>
<td>-1.92794011E-01</td>
<td>2.54332554E-03</td>
<td>-7.76243120E-07</td>
</tr>
<tr>
<td>1.95E-03</td>
<td>-9.70235594E-02</td>
<td>6.35795004E-04</td>
<td>-4.85085874E-08</td>
</tr>
</tbody>
</table>

Halving \( h \) from row to row reduces the errors by a factor of 2, 4 and 16, i.e., the errors go like \( h \), \( h^2 \), and \( h^4 \).

Observe the superior accuracy of Central4 compared with the simple

\[
\frac{f(x + h) - f(x)}{h} \quad \text{Forward1}
\]
Alternative implementations (in the book)

- **Pure Python functions**
  downside: more arguments to transfer, cannot apply formulas twice to get 2nd-order derivatives etc.

- **Functional programming**
  gives the same flexibility as the OO solution

- **One class and one common math formula**
  applies math notation instead of programming techniques to generalize code

These techniques are beyond scope in the course, but place OO into a bigger perspective
There are numerous formulas for numerical integration and all of them can be put into a common notation:

$$\int_a^b f(x) dx \approx \sum_{i=0}^{n-1} w_i f(x_i)$$

$w_i$: weights, $x_i$: points (specific to a certain formula)

The Trapezoidal rule has $h = (b - a)/(n - 1)$ and

$$x_i = a + ih, \quad w_0 = w_{n-1} = \frac{h}{2}, \quad w_i = h \ (i \neq 0, n - 1)$$

The Midpoint rule has $h = (b - a)/n$ and

$$x_i = a + \frac{h}{2} + ih, \quad w_i = h$$
Simpson’s rule has

\[ x_i = a + ih, \quad h = \frac{b - a}{n - 1} \]

\[ w_0 = w_{n-1} = \frac{h}{6} \]

\[ w_i = \frac{h}{3} \text{ for } i \text{ even}, \quad w_i = \frac{2h}{3} \text{ for } i \text{ odd} \]

Other rules have more complicated formulas for \( w_i \) and \( x_i \).
Why should these formulas be implemented in a class hierarchy?

- A numerical integration formula can be implemented as a class: \(a, b\) and \(n\) are attributes and an `integrate` method evaluates the formula.
- We argued in the class chapter that this is smart.
- All such classes are quite similar: the evaluation of \(\sum_j w_j f(x_j)\) is the same, only the definition of the points and weights differ among the classes.
- Recall: code duplication is a bad thing!
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we put \(\sum_j w_j f(x_j)\) in a superclass (method `integrate`).
- Subclasses extend the superclass with code specific to a math formula, i.e., definition of \(w_j\) and \(x_j\) in this case, in a method `construct_rule`. 
Why should these formulas be implemented in a class hierarchy?

- A numerical integration formula can be implemented as a class: \( a, b \) and \( n \) are attributes and an `integrate` method evaluates the formula.

- We argued in the class chapter that this is smart.

- All such classes are quite similar: the evaluation of \( \sum_j w_j f(x_j) \) is the same, only the definition of the points and weights differ among the classes.

- Recall: code duplication is a bad thing!

- The general OO idea: place code common to many classes in a superclass and inherit that code — here we put \( \sum_j w_j f(x_j) \) in a superclass (method `integrate`).

- Subclasses extend the superclass with code specific to a math formula, i.e., definition of \( w_j \) and \( x_j \) in this case, in a method `construct_rule`. 
Why should these formulas be implemented in a class hierarchy?

- A numerical integration formula can be implemented as a class: \( a, b \) and \( n \) are attributes and an `integrate` method evaluates the formula.
- We argued in the class chapter that this is smart.
- All such classes are quite similar: the evaluation of \( \sum_j w_j f(x_j) \) is the same, only the definition of the points and weights differ among the classes.
- Recall: code duplication is a bad thing!
- The general OO idea: place code common to many classes in a superclass and inherit that code — here we put \( \sum_j w_j f(x_j) \) in a superclass (method `integrate`).
- Subclasses extend the superclass with code specific to a math formula, i.e., definition of \( w_j \) and \( x_j \) in this case, in a method `construct_rule`.
Why should these formulas be implemented in a class hierarchy?

- A numerical integration formula can be implemented as a class: $a$, $b$ and $n$ are attributes and an `integrate` method evaluates the formula.
- We argued in the class chapter that this is smart.
- All such classes are quite similar: the evaluation of $\sum_j w_j f(x_j)$ is the same, only the definition of the points and weights differ among the classes.
- Recall: code duplication is a bad thing!
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we put $\sum_j w_j f(x_j)$ in a superclass (method `integrate`).
- Subclasses extend the superclass with code specific to a math formula, i.e., definition of $w_j$ and $x_j$ in this case, in a method `construct_rule`. 
Why should these formulas be implemented in a class hierarchy?

- A numerical integration formula can be implemented as a class: $a$, $b$ and $n$ are attributes and an `integrate` method evaluates the formula.
- We argued in the class chapter that this is smart.
- All such classes are quite similar: the evaluation of $\sum_j w_j f(x_j)$ is the same, only the definition of the points and weights differ among the classes.
- Recall: code duplication is a bad thing!
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we put $\sum_j w_j f(x_j)$ in a superclass (method `integrate`).
- Subclasses extend the superclass with code specific to a math formula, i.e., definition of $w_i$ and $x_i$ in this case, in a method `construct_rule`
Why should these formulas be implemented in a class hierarchy?

- A numerical integration formula can be implemented as a class: $a$, $b$ and $n$ are attributes and an `integrate` method evaluates the formula.
- We argued in the class chapter that this is smart.
- All such classes are quite similar: the evaluation of $\sum_j w_j f(x_j)$ is the same, only the definition of the points and weights differ among the classes.
- Recall: code duplication is a bad thing!
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we put $\sum_j w_j f(x_j)$ in a superclass (method `integrate`).
- Subclasses extend the superclass with code specific to a math formula, i.e., definition of $w_i$ and $x_i$ in this case, in a method `construct_rule`.
Why should these formulas be implemented in a class hierarchy?

- A numerical integration formula can be implemented as a class: a, b and \( n \) are attributes and an `integrate` method evaluates the formula.
- We argued in the class chapter that this is smart.
- All such classes are quite similar: the evaluation of \( \sum_j w_j f(x_j) \) is the same, only the definition of the points and weights differ among the classes.
- Recall: code duplication is a bad thing!
- The general OO idea: place code common to many classes in a superclass and inherit that code – here we put \( \sum_j w_j f(x_j) \) in a superclass (method `integrate`).
- Subclasses extend the superclass with code specific to a math formula, i.e., definition of \( w_i \) and \( x_i \) in this case, in a method `construct_rule`.
class Integrator:

    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.points, self.weights = self.construct_method()

    def construct_method(self):
        raise NotImplementedError('no rule in class %s' %

    def integrate(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s

    def vectorized_integrate(self, f):
        # f must be vectorized
        return dot(self.weights, f(self.points))
class Trapezoidal(Integrator):
    def construct_method(self):
        h = (self.b - self.a)/float(self.n - 1)
        x = linspace(self.a, self.b, self.n)
        w = zeros(len(x))
        w[1:-1] += h
        w[0] = h/2; w[-1] = h/2
        return x, w
Simpson’s rule is more tricky to implement because of different formulas for odd and even points.

Don’t bother with the details of \( w_i \) and \( x_i \) in Simpson’s rule now – focus on the class design!

class Simpson(Integrator):
    def construct_method(self):
        if self.n % 2 != 1:
            print 'n=%d must be odd, 1 is added' % self.n
            self.n += 1

        <code for computing x and w>
        return x, w
Let us integrate $\int_0^2 x^2 \, dx$ using 101 points:

```python
def f(x):
    return x*x

m = Simpson(0, 2, 101)
print m.integrate(f)
```

- `m = Simpson(...)`: this invokes the superclass constructor, which calls `construct_method`
- `m.integrate(f)`: invokes the inherited `integrate` method
We can empirically test out the accuracy of different integration methods Midpoint, Trapezoidal, Simpson, GaussLegendre2, ...

Sample integral:

$$\int_{0}^{1} \left(1 + \frac{1}{m}\right) t^{\frac{1}{m}} dt = 1$$

This integral is "difficult" numerically for $m > 1$

Key problem: the error in numerical integration formulas is of the form $Cn^{-r}$, mathematical theory can predict $r$ (the "order"), but we can estimate $r$ empirically too

See the book for computational details

Here we focus on the conclusions
Summary of object-orientation principles

- A subclass inherits everything from the superclass
- When to use a subclass/superclass?
  - if code common to several classes can be placed in a superclass
  - if the problem has a natural child-parent concept
- The program flow jumps between super- and sub-classes
- It takes time to master *when* and *how* to use OO
- Study examples!