Computational Programming with Python
Unit 8: Object oriented programming with classes

Numerical Analysis, Lund University
Lecturer: Claus Führer

2016
A datatype, e.g. a list binds data and related methods together:

```python
my_list = [1, 2, 3, 4, -1]
my_list.<tab> # show a list of all related methods (in ipython)
my_list.sort() # is such a method
my_list.reverse() # .. another method
```

In this unit we show how own datatypes and related methods can be created. For example

- polynomials
- triangles
- special problems (find a zero)
- etc.
A minimalistic example

class Nix:
    pass

An object with a certain datatype is said to be an *instance* of that type:

```python
a = Nix()
```

Check:

```python
if isinstance(a, Nix):
    print('Indeed it belongs to the new class Nix')
```
A more complete example

Let us define a new datatype for rational numbers:

```python
class RationalNumber:
    def __init__(self, numerator, denominator):
        if not isinstance(numerator, int):
            raise TypeError('numerator should be of type int')
        if not isinstance(denominator, int):
            raise TypeError('denominator should be of type int')
        self.numerator = numerator
        self.denominator = denominator
```

It got a method which initializes an instance with two attributes:

```python
q = RationalNumber(10, 20)  # Defines a new object
q.numerator               # returns 10
q.denominator             # returns 20
p = RationalNumber(10, 20.0)  # raises a TypeError
```
__init__ and self

What happens, when

```python
q = RationalNumber(10, 20)  # Defines a new object
```

is executed?

- a new object with name q is created
- the command `q.__init__(10, 20)` is executed.

self is a placeholder for the name of the newly created instance (here: q)
Adding methods

Methods are functions bound to the class:

```python
class RationalNumber:
    ...
    def convert2float(self):
        return float(self.numerator)/float(self.denominator)
```

and its use ...

```python
q=RationalNumber(10,20)  # Defines a new object
q.convert2float()  # returns 0.5
```

Note, again the special role of `self`!
Adding methods (Cont.)

Note:
Both commands are equivalent:

```python
RationalNumber.convert2float(q)
q.convert2float()
```
Adding method (Cont.)

A second method ....

```python
class RationalNumber:
    ...
    def add(self, other):
        p1, q1 = self.numerator, self.denominator
        if isinstance(other, RationalNumber):
            p2, q2 = other.numerator, other.denominator
        elif isinstance(other, int):
            p2, q2 = other, 1
        else:
            raise TypeError('...')
        return RationalNumber(p1 * q2 + p2 * q1, q1 * q2)
```

A call to this method takes the following form

```python
q = RationalNumber(1, 2)
p = RationalNumber(1, 3)
q.add(p)  # returns the RationalNumber for 5/6
```
We would like to add RationalNumber number instances just by \( p+q \).

Renaming the method

\[
\text{RationalNumber.add}
\]

\[
\text{to}
\]

\[
\text{RationalNumber.__add__}
\]

makes this possible.
## Special methods: Operators (Cont.)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Method</th>
<th>Operator</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>__add__</td>
<td>+=</td>
<td>__iadd__</td>
</tr>
<tr>
<td>*</td>
<td>__mul__</td>
<td>*=</td>
<td>__imul__</td>
</tr>
<tr>
<td>-</td>
<td>__sub__</td>
<td>-=</td>
<td>__isub__</td>
</tr>
<tr>
<td>/</td>
<td>__div__</td>
<td>/=</td>
<td>__idiv__</td>
</tr>
<tr>
<td>**</td>
<td>__pow__</td>
<td></td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>__eq__</td>
<td>!=</td>
<td>__ne__</td>
</tr>
<tr>
<td>&lt;=</td>
<td>__le__</td>
<td>&lt;</td>
<td>__lt__</td>
</tr>
<tr>
<td>&gt;=</td>
<td>__ge__</td>
<td>&gt;</td>
<td>__gt__</td>
</tr>
<tr>
<td>()</td>
<td>__call__</td>
<td>[]</td>
<td>__getitem__</td>
</tr>
</tbody>
</table>
Special methods: Representation

__repr__ defines how the object is represented, when just typing its name:

```python
class RationalNumber:
    ...
    def __repr__(self):
        return '{}/{}'.format(self.numerator, self.denominator)
```

Now: q returns 10/20.
Reverse operations

So far we can use the class for operations like

$$\frac{1}{5} + \frac{5}{6} \text{ or } \frac{1}{5} + 5$$

but $5 + \frac{1}{5}$ requires that the integer’s method `__add__` knows about `RationalNumber`.

Instead of extending the methods of `int` we use reverse operations:

```python
class RationalNumber(object):
    ....
    def __radd__(self, other):
        return self + other
```

`__radd__` reverses the role of self and other.

Check the operation $1 + q$
Classes and generators

Consider the 3-term recursion $u_{i+1} = a_1 u_i + a_0 u_{i-1}$

```python
class Recursion3Term:
    def __init__(self, a0, a1, u0, u1):
        self.coeff = [a1, a0]
        self.initial = [u1, u0]
    def __iter__(self):
        u1, u0 = self.initial
        yield u0
        yield u1
        a1, a0 = self.coeff
        while True:
            u1, u0 = a1*u1 + a0*u0, u1
            yield u1
    def __getitem__(self, k):
        for i, r in enumerate(self):
            if i == k:
                return r
```
Classes and generators (Cont.)

and here its use

```python
r3 = Recursion3Term(-0.35, 1.2, 1, 1)
for i, r in enumerate(r3):
    if i == 7:
        print r  # returns 0.194167
        break
r3[7]  # returns 0.194167
```
Extending the example

We add a method `shorten` to our example:

```python
class RationalNumber:
    ....
    def shorten(self):
        def gcd(a, b):
            # Computes the greatest common divisor
            if b == 0:
                return a
            else:
                return gcd(b, a % b)
        factor = gcd(self.numerator, self.denominator)
        self.numerator = self.numerator // factor
        self.denominator = self.denominator // factor
```

Note, it performs an *inplace* modification of the object.
The complete example

[On the webpage.]
A guiding example

Let us assume the following class to define a triangle

```python
class Triangle:
    def __init__(self, A, B, C):
        self.A = array(A)
        self.B = array(B)
        self.C = array(C)
        self.a = self.C - self.B
        self.b = self.C - self.A
        self.c = self.B - self.A
    def area(self):
        return abs(cross(self.b, self.c))/2
```

An instance of this triangle is created by

```python
tr = Triangle([0.,0.], [1.,0.], [0.,1.])
```

and its area is computed by

```python
tr.area() # returns 0.5
```
Altering an attribute

Altering an attribute,

```
tr.B = array([1.0, 0.1])
```

does not affect the attribute 'area':

```
tr.area() # still 0.5
```

We have to be able to run “internally” some commands when one attribute is changed in order to modify the others!

For this end we introduce attributes only for internal use, e.g _B and ...
Altering an attribute (Cont.)

... and modify our class

class Triangle:
    def __init__(self, A, B, C):
        self._A = array(A)
        self._B = array(B)
        self._C = array(C)
        self._a = self._C - self._B
        self._b = self._C - self._A
        self._c = self._B - self._A
    def area(self):
        return abs(cross(self._c, self._b)) / 2.
    def set_B(self, B):
        self._B = B
        self._a = self._C - self._B
        self._c = self._B - self._A
    def get_B(self):
        return self._B
B = property(fget = get_B, fset = set_B)

What would happen if we change self._B to self.B?
A triangle has three corners

Python allows us to delete an attribute

def del_Pt(self):
    raise Exception('A triangle point cannot be deleted')

B = property(fget = get_B,
             fset = set_B,
             fdel = del_Pt)

del tr.B # now raises an exception
Inheritance - a mathematical concept

Mathematics builds on class inheritance principles:

Example of a mathematical inheritance hierarchy

- **Functions**
  can be evaluated, differentiated, integrated, summed up, ....

- **Polynomials** are functions
  same methods as functions, change representation, truncate, ...

- **Trigonometric polynomials** are polynomials
  same methods as polynomials, ask for real parts, ...

We say polynomials inherit properties and methods from functions.
Inheritance - in Python

We consider a general integrator class performing:

\[ u_{i+1} = u_i + h\Phi(f, u_i, t_i, h). \]

class Onestepmethod:
    def __init__(self, f, x0, t0, te, N):
        self.f = f
        self.x0 = x0
        self.interval = [t0, te]
        self.grid = linspace(t0, te, N)
        self.h = (te - t0)/N
    def step(self):
        ti, ui = self.grid[0], self.x0
        yield ti, ui
        for t in self.grid[1:]:
            ui = ui + self.h*self.phi(self.f, ui, ti)
            ti = t
            yield ti, ui
    def solve(self):
        self.solution = array([tu for tu in self.step()])
... construct from it two subclasses

class ExpEuler(Onestepmethod):
    def phi(self, f, u, t):
        return f(u, t)

class MidPointRule(Onestepmethod):
    def phi(self, f, u, t):
        return f(u + self.h/2*f(u,t), t + self.h/2)
Which we then call

```python
def f(x, t):
    return -0.5*x

euler = ExpEuler(f, 15., 0., 10., 20)
euler.solve()

midpoint = MidPointRule(f, 15., 0., 10., 20)
midpoint.solve()
```