Operator Overloading

1. OOP to count Flops
   - a flop = a floating-point operation
   - overloading arithmetical operators
   - counting number of flops in a sum
2. Quaternions
   - hypercomplex numbers
   - application in computer graphics
3. Operator Overloading
   - the class `Quaternion`
   - defining functional attributes
4. Summary + Assignments

MCS 260 Lecture 25
Introduction to Computer Science
Jan Verschelde, 9 March 2016

Object-Oriented Programming

A definition from Grady Booch et al.:

Object-oriented programming is a method of implementation in which

1. programs are organized as cooperative collections of ***objects***,
2. each of which represents an instance of some ***class***,
3. and whose classes are all members of a ***hierarchy*** of classes united via inheritance relationships.

Objects — not algorithms — are the building blocks.

Algorithms are central in procedure-oriented programming.

Operator Overloading

1. OOP to count Flops
   - a flop = a floating-point operation
   - overloading arithmetical operators
   - counting number of flops in a sum
2. Quaternions
   - hypercomplex numbers
   - application in computer graphics
3. Operator Overloading
   - the class `Quaternion`
   - defining functional attributes
4. Summary + Assignments

Counting Flops

floating-point operations

A flop is short for floating-point operation.

In scientific computation, the cost analysis is often measured in flops.

Note: before version 6, MATLAB had a `flops` command.

Using Object Oriented Programming:

1. we define a class `FlopFloat`,
2. every object stores its #flops:
   these are the flops used to compute the number,
3. the overloaded arithmetical operators
   count also the flops for each result.

the class `FlopFloat`

```
class FlopFloat(object):
    """
    An object of the class FlopFloat records
    the number of floating-point operations
    executed to compute the float.
    """

    def __init__(self, f=0.0, n=0):
        """
        Construct a FlopFloat from a number f
        and an operational cost n, by default 0.
        """
        self.float = float(f)
        self.flops = n
```

interactive usage

If we save the class definition in `flopfloats.py`,
then we can import the definition as follows:

```
>>> from flopfloats import FlopFloat
>>> x = FlopFloat(3.14159625)
>>> x
3.1416e+00
>>> str(x)
'3.1416e+00'
>>> print(x)
3.1416e+00
>>> x.__str__('%.8f')
'3.14159625'
```

The representation of a `FlopFloat` is defined by `__repr__()`.
The string representation is defined by `__str__()`.

representations of `FlopFloat`

```
def __str__(self, form='%.4e'):
    """
    Uses the formatting string in the %
    to return a string representation.
    """
    return form % self.float

def __repr__(self):
    """
    Returns the default string representation.
    """
    return str(self)
```

Because of the definition of `__repr__()`, we have:

```
>>> x
3.1416e+00
```

Operator Overloading

1. OOP to count Flops
   - a flop = a floating-point operation
   - overloading arithmetical operators
   - counting number of flops in a sum
2. Quaternions
   - hypercomplex numbers
   - application in computer graphics
3. Operator Overloading
   - the class `Quaternion`
   - defining functional attributes
4. Summary + Assignments

overloading operators

Recall the addition of strings:

```
>>> "ab" + "bc"
'abbc'
```

and the addition of lists:

```
>>> [1,2] + [2,3]
[1, 2, 2, 3]
```

The + operator is defined via the `__add__` method:

```
>>> L = [1,2]
>>> L.__add__([3,4])
[1, 2, 3, 4]
```

the class `list`

Typing `help(list)` in a Python session shows:

```
Help on class list in module __builtin__:

class list(object)
 |  list() -> new list
 |  list(sequence) -> new list initialized
 |                    from sequence's items
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
...
```

comparison operators and methods

comparison operation	operator	method
equality	==	__eq__
inequality	!=	__ne__
less than	<	__lt__
greater than	>	__gt__
less or equal	<=	__le__
greater or equal	>=	__ge__

motivation for an equality test

Without having defined the equality in `FlopFloat`:

```
>>> x = FlopFloat(3)
>>> x == 3
False
```

But we have:

```
>>> x.float == 3
True
```

And consider:

```
>>> y = FlopFloat(3)
>>> x == y
False
```

We get `False` because `x` and `y` are different objects.

after extending `FlopFloat` with `__eq__`

```
>>> from flopfloats import FlopFloat
>>> x = FlopFloat(3)
>>> x == 3
True
>>> y = FlopFloat(3)
>>> x == y
True
>>> z = x + 0
>>> z
3.0000e+00
>>> z == x
False
>>> z.flops
1
>>> x.flops
0
```

defining the equality operator

```
def __eq__(self, other):
    """
    Two flopfloats are equal if both
    their float and flops match.
    """
    if isinstance(other, FlopFloat):
        return (other.float == self.float) \
            and (other.flops == self.flops)
    else:
        return self.float == other
```

arithmetical operators and methods

arithmetical operation	operator	method
negation	-	__neg__
addition	+	__add__
inplace addition	+=	__iadd__
reflected addition	+	__radd__
subtraction	-	__sub__
inplace subtraction	-=	__isub__
reflected subtraction	-=	__rsub__
multiplication	*	__mul__
inplace multiplication	*=	__imul__
reflected multiplication	*	__rmul__
division	/	__div__
inplace division	/=	__idiv__
reflected division	/=	__rdiv__
invert	~	__invert__
power	**	__pow__

Overloading Operators, e.g.: addition

The `other` may be an ordinary `float`:

```
def __add__(self, other):
    """
    Returns the result of the addition,
    the other may be an ordinary number.
    """
    if isinstance(other, FlopFloat):
        result = FlopFloat(self.float + other.float)
        result.flops = self.flops + other.flops + 1
    else:  # treat other just as ordinary number
        result = FlopFloat(self.float + other)
        result.flops = self.flops + 1
    return result
```

Other arithmetical operations are defined similarly.

We store the class definition in the file `flopfloats.py` and import it as a module.

reflected addition

The reflected (or swapped) addition happens when the first operand in `+` is not a `FlopFloat` but an ordinary number.

```
def __radd__(self, other):
    """
    Addition when operand is not a FlopFloat,
    as in 2.7 + x or 3 + x (reflected operand).
    """
    result = self + other
    return result
```

When `x` is a `FlopFloat`, then `x+y` is executed as `x.__add__(y)`, where `x` is `self` and `y` is other.

For `x + y` when `x` is not a `FlopFloat`, but `y` is a `FlopFloat`, then `y.__radd__(x)` is executed.

inplace addition and negation

The inplace operator allows for shorter notation.

```
def __iadd__(self, other):
    """
    Defines the inplace addition, x += y.
    """
    result = self + other
    return result
```

Flipping sign is a unary operator (no `other`):

```
def __neg__(self):
    """
    Returns -x.
    """
    result = self
    result.float = -result.float
    # flipping sign does not count as a flop
    return result
```

Operator Overloading

1. OOP to count Flops
   - a flop = a floating-point operation
   - overloading arithmetical operators
   - counting number of flops in a sum
2. Quaternions
   - hypercomplex numbers
   - application in computer graphics
3. Operator Overloading
   - the class `Quaternion`
   - defining functional attributes
4. Summary + Assignments

A Simple Test of Use

summing numbers in `flopsum.py`

```
"""
We use FlopFloats to count the number
of operations when summing n floats.
"""

from flopfloats import FlopFloat
from random import gauss

print('counting flops in a sum' + \
      ' of n floating-point numbers')
N = int(input('Give n : '))
RANDSUM = FlopFloat()
for i in range(0, N):
    RANDSUM = RANDSUM + gauss(0, 1)
print('sum = ' + str(RANDSUM) + \
    ' #flops is %d' % RANDSUM.flops)
```

running `flopsum`

At the command prompt `$`:

```
$ python flopsum.py
counting flops in a sum of n floats
Give n : 100
sum = -2.8354e+00 #flops is 100
```

It works!

A less trivial application:
Use `FlopFloats` to count #operations to evaluate

$$p(x) = 6x^4 - 3.23x^3 - x^2 + 0.3453x - 9.23.$$

Operator Overloading

1. OOP to count Flops
   - a flop = a floating-point operation
   - overloading arithmetical operators
   - counting number of flops in a sum
2. Quaternions
   - hypercomplex numbers
   - application in computer graphics
3. Operator Overloading
   - the class `Quaternion`
   - defining functional attributes
4. Summary + Assignments

Complex Numbers

Typing `help(complex)` in a Python session shows

```
Help on class complex in module __builtin__:

class complex(object)
 |  complex(real[, imag]) -> complex number
 |
 |  Create a complex number from a real part
    and an optional imaginary part.
 |  This is equivalent to (real + imag*1j)
    where imag defaults to 0.
 |
 |  Methods defined here:
 |
 |  __abs__(...)
 |       x.__abs__() <==> abs(x)
...
```

Quaternion

hypercomplex numbers

A quaternion q (or a hypercomplex number) is

$$q = a_0 + a_1 i + a_2 j + a_3 k,$$

where the tuple (a_0, a_1, a_2, a_3) is the coefficient vector of q

and the symbols i, j, and k satisfy

$$\begin{array}{lll} i^2 = -1 & j^2 = -1, & k^2 = -1, \\ ij = k, & jk = i, & ki = j, \\ ji = -k, & kj = -i, & ik = -j, \end{array}$$

defining the multiplication of two quaternions.

The set of all quaternions is often denoted by $\mathbb{H}$,
in honor of Sir William Rowan Hamilton who introduced them in 1843
before vector algebra was known.

Operator Overloading

1. OOP to count Flops
   - a flop = a floating-point operation
   - overloading arithmetical operators
   - counting number of flops in a sum
2. Quaternions
   - hypercomplex numbers
   - application in computer graphics
3. Operator Overloading
   - the class `Quaternion`
   - defining functional attributes
4. Summary + Assignments

Applications of Quaternions

computer graphics

Quaternions represent coordinate transformations in 3-space more compactly than matrices:

$$q = (a_0, \mathbf{a}), \quad \mathbf{a} = (a_1, a_2, a_3).$$

Also composition of coordinate transformations goes faster with quaternions.

Quaternion multiplication $\otimes$
with scalar, dot ($\cdot$), and cross product ($\times$):

$$(a_0, \mathbf{a}) \otimes (b_0, \mathbf{b}) = a_0 b_0 - \mathbf{a} \cdot \mathbf{b} + a_0 \mathbf{a} + b_0 \mathbf{b} + \mathbf{a} \times \mathbf{b}.$$

Rotations in Space

Rotation about a unit vector $\overrightarrow{\mathbf{u}}$ by angle θ:

$$q = (s, \overrightarrow{\mathbf{v}}) \quad \text{where} \quad \begin{cases} s = \cos(\theta/2), \\ \overrightarrow{\mathbf{v}} = \sin(\theta/2)\overrightarrow{\mathbf{u}}. \end{cases}$$

Applying the rotation to a point $\mathbf{p} = (x, y, z)$:

1. represent $\mathbf{p}$ by the quaternion $P = (0, \mathbf{p})$,
2. compute $q \otimes P \otimes q^{-1}$.

For $q = (q_0, q_1, q_2, q_3)$, its inverse is $q^{-1} = \dfrac{q^*}{||q||^2}$,

where

- the conjugate of q is $q^* = (q_0, -q_1, -q_2, -q_3)$, and
- the magnitude of q satisfies $||q||^2 = q_0^2 + q_1^2 + q_2^2 + q_3^2$.

Operator Overloading

1. OOP to count Flops
   - a flop = a floating-point operation
   - overloading arithmetical operators
   - counting number of flops in a sum
2. Quaternions
   - hypercomplex numbers
   - application in computer graphics
3. Operator Overloading
   - the class `Quaternion`
   - defining functional attributes
4. Summary + Assignments

the class `Quaternion` in the file `quaternion.py`

```
"""
Another illustration of operator overloading.
"""
class Quaternion(object):
    """
    Quaternions are hypercomplex numbers.
    """
```

With the class definition in a file:

```
>>> import quaternion
>>> quaternion.__doc__
'\nAnother illustration of operator overloading.\n'
>>> from quaternion import Quaternion
>>> Quaternion.__doc__
'\n    Quaternions are hypercomplex numbers.\n
```

The `__doc__` gives the documentation strings
of the module `quaternion` and the class `Quaternion`.

data attributes for classes and objects

```
class Quaternion(object): # documentation string omitted

    count = 0   # class data attributes
    who = []

    def __init__(self, a=0, b=0, c=0, d=0):
        """
        Constructs the Quaternion with
        coefficients a,b,c,d.
        """
        self.cre = a  # real part
        self.cim = b  # imaginary part
        self.cjj = c  # j-part
        self.ckk = d  # k-part
        Quaternion.count += 1
        Quaternion.who.append(self)
```

`count` and `who` are *class* data attributes

`a`, `b`, `c`, and `d` are *object* data attributes

instances of data attributes for classes and objects

Consider

```
>>> from quaternion import Quaternion
>>> x = Quaternion(2,3,0,1)
>>> y = Quaternion(0,1,0,0)
>>> Quaternion.count
2
```

The numbers 2,3,0,1 are attributes for the object `x`, just as 0,1,0,0 are for `y`.

The value 2 counts the number of quaternions, stored by the class data attribute `count`.

The other class data attribute, `Quaternion.who` keeps the list of all quaternions.

Operator Overloading

1. OOP to count Flops
   - a flop = a floating-point operation
   - overloading arithmetical operators
   - counting number of flops in a sum
2. Quaternions
   - hypercomplex numbers
   - application in computer graphics
3. Operator Overloading
   - the class `Quaternion`
   - defining functional attributes
4. Summary + Assignments

constructing quaternions

To define a general quaternion, we supply 4 arguments:

```
>>> from quaternion import Quaternion
>>> q = Quaternion(1,2,3,4) # 1 + 2 i + 3 j + 4 k
```

but sometimes we want to supply less:

```
>>> c = Quaternion(5)      # defines a constant
>>> k = Quaternion(d = 1) # defines k
```

The constructor `__init__` has 4 default values:

```
def __init__(self, a=0, b=0, c=0, d=0):
    """
    Constructs the Quaternion with
    coefficients a,b,c,d.
    """
```

representing quaternions

```
    def __str__(self):
        """
        The string representation of a Quaternion
        uses i, j, and k.  Note the + in the format.
        """
        return '%.2f %+.2f i %+.2f j %+.2f k' \
            % (self.cre, self.cim, self.cjj, self.ckk)
>>> I = Quaternion(b=1)
>>> str(I)
'0.00 + 1.00 i + 0.00 j + 0.00 k'
>>> I
<quaternion.Quaternion instance at 0x402d5dcc>
```

To represent with a string by default, overload `__repr__()`:

```
def __repr__(self):
    "Quaternion Representation is a String"
    return str(self)
```

Typing `I` gives now `0.00 + 1.00 i + 0.00 j + 0.00 k`.

overloading equality

With the builtin == operator:

```
>>> x = Quaternion(1)
>>> y = Quaternion(1,0)
>>> x
1.00 + 0.00 i + 0.00 j + 0.00 k
>>> y
1.00 + 0.00 i + 0.00 j + 0.00 k
>>> x == y
False
```

But we wanted (and expected) `True` as answer!

It takes two to compare, besides `self`, we need the `other`.

overloading equality: defining `_ _eq_ _`

The `other` is like `self`, used in binary operators:

```
def __eq__(self, other):
    """
    Defines Quaternion Equality.
    """
    return self.cre == other.cre and \
           self.cim == other.cim and \
           self.cjj == other.cjj and \
           self.ckk == other.ckk
```

Now, we will have `True`:

```
>>> x = Quaternion(1)
>>> y = Quaternion(1,0)
>>> x == y
True
```

overloading arithmetical operators

We want to write `z = x + y` and even `z += x` when adding two quaternions.

Consider first the negation, i.e.: `z = -x`

```
def __neg__(self):
    """
    The negation of a Quaternion.
    """
    return Quaternion(-self.cre, \
                      -self.cim, \
                      -self.cjj, \
                      -self.ckk)
```

Defining `__neg__` we overloaded the negation `-`.

overloading addition the other and inplace addition

It takes two to add, besides `self`, we need the `other`.

```
def __add__(self, other):
    """
    Adding two Quaternions"
    """
    return Quaternion(self.cre + other.cre, \
                      self.cim + other.cim, \
                      self.cjj + other.cjj, \
                      self.ckk + other.ckk)
```

If we also want to update quaternions as `z += x`,
then we overload the inplace addition operator `__iadd__`.

magnitude and conjugate

```
def __abs__(self):
    """
    Returns the magnitude of a Quaternion.
    """
    from math import sqrt
    return sqrt(self.cre**2+self.cim**2 \
                +self.cjj**2+self.ckk**2)

def conjugate(self):
    """
    Returns the conjugate of a Quaternion.
    """
    return Quaternion(self.cre, -self.cim, \
                      -self.cjj, -self.ckk)
```

the inverse of a quaternion

```
def scamul(self, scx):
    """
    Product of Quaternion with scalar scx.
    """
    return Quaternion(self.cre*scx, self.cim*scx, \
                      self.cjj*scx, self.ckk*scx)

def __invert__(self):
    """
    The Inverse of the Quaternion.
    """
    return self.conjugate().scamul(1/abs(self))
```

Summary + Assignments

Assignments:

1. Overload the subtraction operator for quaternions.
   Also do the inplace version.
2. Provide a method `Coefficients` that returns a tuple with the coefficients of the quaternion.
3. Extend `scamul` so that you can compute the multiplication of any quaternion with any scalar multiple of i, j, and k.
4. Write Python code for $\otimes$ and verify whether $q \otimes q^{-1} = 1$, for a random quaternion q.
5. Use operator overloading in the Python code for the class `Rational` to work with rational numbers.