Outline

1. **OOP to count Flops**
   a flop = a floating-point operation
   overloading arithmetical operators

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, 22 October 2008
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.
attributes of classes
operator overloading

1. **OOP to count Flops**
   a flop = a floating-point operation
   overloading arithmetical operators

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. class FlopFloat inherits from float;
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.
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. `class FlopFloat inherits from float;`
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.
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. class `FlopFloat` inherits from `float`;
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.
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. `class FlopFloat inherits from float;`
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(float):
    """
    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):
        """constructor for a flopfloat"""
        self.float = f
        self.flops = n
```
# L-25 MCS 260 Wed 22 Oct 2008 : flopfloats
#
# To analyze the cost of an algorithm, we count the number of operations.

class FlopFloat(float):
    
    """
    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):
    """constructor for a flopfloat"
    self.float = f
    self.flops = n
attributes of classes
operator overloading

1. OOP to count Flops
   a flop = a floating-point operation
   overloading arithmetical operators

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:

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

and the addition of lists:

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

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

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

Recall the addition of strings:

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

and the addition of lists:

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

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

```python
>>> 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__`:

```python
class list(object)
    list() -> new list
    list(sequence) -> new list initialized from sequence’s items

    Methods defined here:

    __add__(...)  # x.__add__(y) <=> x+y
```

...
Operators and Methods

<table>
<thead>
<tr>
<th>comparison operation</th>
<th>operator</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>equality</td>
<td>==</td>
<td><strong>eq</strong></td>
</tr>
<tr>
<td>inequality</td>
<td>!=</td>
<td><strong>ne</strong></td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td><strong>lt</strong></td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td><strong>gt</strong></td>
</tr>
<tr>
<td>less or equal</td>
<td>&lt;=</td>
<td><strong>le</strong></td>
</tr>
<tr>
<td>greater or equal</td>
<td>&gt;=</td>
<td><strong>ge</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>arithmetical operation</th>
<th>operator</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>-</td>
<td><strong>neg</strong></td>
</tr>
<tr>
<td>addition</td>
<td>+</td>
<td><strong>add</strong></td>
</tr>
<tr>
<td>inplace addition</td>
<td>+=</td>
<td><strong>iadd</strong></td>
</tr>
<tr>
<td>subtraction</td>
<td>-</td>
<td><strong>sub</strong></td>
</tr>
<tr>
<td>inplace subtraction</td>
<td>-=</td>
<td><strong>isub</strong></td>
</tr>
<tr>
<td>multiplication</td>
<td>*</td>
<td><strong>mul</strong></td>
</tr>
<tr>
<td>inplace multiplication</td>
<td>*=</td>
<td><strong>imul</strong></td>
</tr>
<tr>
<td>division</td>
<td>/</td>
<td><strong>div</strong></td>
</tr>
<tr>
<td>inplace division</td>
<td>/=</td>
<td><strong>idiv</strong></td>
</tr>
<tr>
<td>invert</td>
<td>~</td>
<td><strong>invert</strong></td>
</tr>
</tbody>
</table>
## Operators and Methods

<table>
<thead>
<tr>
<th>comparison operation</th>
<th>operator</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>equality</td>
<td>==</td>
<td><strong>eq</strong></td>
</tr>
<tr>
<td>inequality</td>
<td>!=</td>
<td><strong>ne</strong></td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td><strong>lt</strong></td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td><strong>gt</strong></td>
</tr>
<tr>
<td>less or equal</td>
<td>&lt;=</td>
<td><strong>le</strong></td>
</tr>
<tr>
<td>greater or equal</td>
<td>&gt;=</td>
<td><strong>ge</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>arithmetical operation</th>
<th>operator</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>-</td>
<td><strong>neg</strong></td>
</tr>
<tr>
<td>addition</td>
<td>+</td>
<td><strong>add</strong></td>
</tr>
<tr>
<td>inplace addition</td>
<td>+=</td>
<td><strong>iadd</strong></td>
</tr>
<tr>
<td>subtraction</td>
<td>-</td>
<td><strong>sub</strong></td>
</tr>
<tr>
<td>inplace subtraction</td>
<td>-=</td>
<td><strong>isub</strong></td>
</tr>
<tr>
<td>multiplication</td>
<td>*</td>
<td><strong>mul</strong></td>
</tr>
<tr>
<td>inplace multiplication</td>
<td>*=</td>
<td><strong>imul</strong></td>
</tr>
<tr>
<td>division</td>
<td>/</td>
<td><strong>div</strong></td>
</tr>
<tr>
<td>inplace division</td>
<td>/=</td>
<td><strong>idiv</strong></td>
</tr>
<tr>
<td>invert</td>
<td>~</td>
<td><strong>invert</strong></td>
</tr>
</tbody>
</table>
Overloading Operators

for example, the addition

The other may be an ordinary float:

def __add__(self,*other):
    "returns the result of the addition"
    if isinstance(other[0],FlopFloat):
        sum = FlopFloat(self.float + other[0].float)
        sum.flops = self.flops + other[0].flops + 1
    else:  # treat other just as ordinary number
        sum = FlopFloat(self.float + other[0])
        sum.flops = self.flops + 1
    return sum

Other arithmetical operations are defined similarly.

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

for example, the addition

The other may be an ordinary float:

```python
def __add__(self,*other):
    "returns the result of the addition"
    if isinstance(other[0],FlopFloat):
        sum = FlopFloat(self.float + other[0].float)
        sum.flops = self.flops + other[0].flops + 1
    else:  # treat other just as ordinary number
        sum = FlopFloat(self.float + other[0])
        sum.flops = self.flops + 1
    return sum
```

Other arithmetical operations are defined similarly.

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

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

from flopfloats import *
from random import gauss

print 'counting flops in a sum' + \\
    ' of n floats'

n = input('Give n : ')

sum = FlopFloat()
for i in range(0,n):
    r = FlopFloat(gauss(0,1))
    sum = sum + r

print 'sum = ' + str(sum) + \\
    ' #flops is %d' % sum.flops
A Simple Test of Use
summing numbers in flopsum.py

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

from flopfloats import *
from random import gauss
print 'counting flops in a sum' + \
    ' of n floats'
n = input('Give n : ')
sum = FlopFloat()
for i in range(0,n):
    r = FlopFloat(gauss(0,1))
    sum = sum + r
print 'sum = ' + str(sum) + \
    ' #flops is %d' % sum.flops
```
#
# We use FlopFloats to count the number
# of operations when summing n floats.

from flopfloats import *
from random import gauss
print 'counting flops in a sum' + \
    ' of n floats'
n = input('Give n : ')
sum = FlopFloat()
for i in range(0,n):
    r = FlopFloat(gauss(0,1))
    sum = sum + r
print 'sum = ' + str(sum) + \
    '#flops is %d' % sum.flops
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 *
from random import gauss
print 'counting flops in a sum' + \  
    ' of n floats'

n = input('Give n : ')
sum = FlopFloat()
for i in range(0,n):
    r = FlopFloat(gauss(0,1))
    sum = sum + r
print 'sum = ' + str(sum) + \  
    '#flops is %d' % sum.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

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

\[ p(x) = 6x^4 - 3.23x^3 - x^2 + 0.3453x - 9.23. \]
attributes of classes
operator overloading

1. OOP to count Flops
   a flop = a floating-point operation
   overloading arithmetical operators

2. Quaternions
   hypercomplex numbers
   application in computer graphics

3. Operator Overloading
   the class Quaternion
   defining functional attributes

4. Summary + Assignments
Typing \texttt{help(complex)} in a Python session shows

Help on class complex in module \texttt{__builtin__}:

```python
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 \((\text{real} + \text{imag} \times 1j)\)
    where imag defaults to 0.

    Methods defined here:

    __abs__(...)  
    x.__abs__()  \iff  abs(x)

...
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

\[
i^2 = -1, \quad j^2 = -1, \quad k^2 = -1,
\]

\[
ij = k, \quad jk = i, \quad ki = j,
\]

\[
ji = -k, \quad kj = -i, \quad ik = -j,
\]

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.
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

$$i^2 = j^2 = k^2 = -1,$$

$$ij = k, \quad jk = i, \quad ki = j,$$

$$ji = -k, \quad kj = -i, \quad ik = -j,$$

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.
Quatetion
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 $i, j, k$ satisfy

$$i^2 = -1, \quad j^2 = -1, \quad k^2 = -1,$$
$$ij = k, \quad jk = i, \quad ki = j,$$
$$ji = -k, \quad kj = -i, \quad ik = -j,$$

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.
attributes of classes
operator overloading

1. OOP to count Flops
   a flop = a floating-point operation
   overloading arithmetical operators

2. Quaternions
   hypercomplex numbers
   application in computer graphics

3. Operator Overloading
   the class Quaternion
   defining functional attributes

4. Summary + Assignments
Applications of Quaternions

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 with scalar, dot (\(\cdot\)), and cross product (\(\times\)):

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

\[(a_0, \mathbf{a}) \otimes (b_0, \mathbf{b}) = a_0 b_0 - \mathbf{a} \cdot \mathbf{b} + a_0 \mathbf{a} + b_0 \mathbf{b} + \mathbf{a} \times \mathbf{b}.\]
attributes of classes
operator overloading

1. OOP to count Flops
   a flop = a floating-point operation
   overloading arithmetical operators

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:

class Quaternion:
    
    """
    Quaternions are hypercomplex numbers.
    """

With the class definition in a file it is like a module:

>>> from quaternion import *
>>> help(quaternion)
>>> dir(quaternion)
>>> Quaternion.__doc__

\n    Quaternions are hypercomplex numbers.\n
The __doc__ is the documentation string of the class.
The Class Quaternion

In the file `quaternion.py`:

```python
class Quaternion:
    
    """
    Quaternions are hypercomplex numbers.
    """
```

With the class definition in a file it is like a module:

```python
>>> from quaternion import *
>>> help(quaternion)
>>> dir(quaternion)
>>> Quaternion.__doc__
'
    Quaternions are hypercomplex numbers.
```

The `__doc__` is the documentation string of the class.
The Class Quaternion

In the file quaternion.py:

class Quaternion:
    ""
    Quaternions are hypercomplex numbers.
    ""

With the class definition in a file it is like a module:

>>> from quaternion import *
>>> help(quaternion)
>>> dir(quaternion)
>>> Quaternion.__doc__
'
    Quaternions are hypercomplex numbers.

The \_\_doc\_\_ is the documentation string of the class.
Data Attributes
for classes and objects

class Quaternion:
    count = 0
    who = []

    def __init__(self, a, b, c, d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

        Quaternion.count += 1
        Quaternion.who.append(self)

    count and who are class data attributes
    a, b, c, and d are object data attributes
class Quaternion:  # strings removed for space
    count = 0  # class data attributes
    who = []

    def __init__(self, a, b, c, d):
        self.a = a  # object data attributes
        self.b = b
        self.c = c
        self.d = d

        Quaternion.count += 1
        Quaternion.who.append(self)

count and who are *class* data attributes
a, b, c, and d are *object* data attributes
class Quaternion:  # strings removed for space
    count = 0  # class data attributes
    who = []  # class data attributes

    def __init__(self,a,b,c,d):
        self.a = a  # object data attributes
        self.b = b
        self.c = c
        self.d = d

        Quaternion.count += 1
        Quaternion.who.append(self)

count and who are class data attributes
a, b, c, and d are object data attributes
Data Attributes
for classes and objects

class Quaternion:  # strings removed for space
count = 0  # class data attributes
who = []  # class data attributes

def __init__(self,a,b,c,d):
    self.a = a  # object data attributes
    self.b = b
    self.c = c
    self.d = d

    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 *
>>> 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.
Instances of Data Attributes
for classes and objects

Consider

```python
>>> from quaternion import *
>>> 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.
Instances of Data Attributes
for classes and objects

Consider

```python
>>> from quaternion import *
>>> 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 \(\text{count}\).

The other class data attribute, \(\text{Quaternion.who}\), keeps the list of all quaternions.
Instances of Data Attributes
for classes and objects

Consider

>>> from quaternion import *
>>> 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 \texttt{count}.

The other class data attribute, Quaternion.\texttt{who} keeps the list of all quaternions.
attributes of classes
operator overloading

1. OOP to count Flops
   a flop = a floating-point operation
   overloading arithmetical operators

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:

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

but sometimes we want to supply less:

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

The constructor `__init__`:

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

To define a general quaternion, we supply 4 arguments:

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

but sometimes we want to supply less:

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

The constructor `__init__`:

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

To define a general quaternion, we supply 4 arguments:

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

but sometimes we want to supply less:

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

The constructor `__init__`:

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

The builtin method `str()` converts to a string.

```python
def __str__(self):
    """The string representation of a Quaternion"
    return '%.2f + %.2f i + %.2f j + %.2f k'
    % (self.a, self.b, self.c, self.d)
```

```python
>>> 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()`:

```python
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`.
Representing Quaternions

The builtin method \texttt{str()} converts to a string.

```python
def \_str\_(self):
    "The string representation of a Quaternion"
    return '%.2f + %.2f i + %.2f j + %.2f k' % (self.a, self.b, self.c, self.d)
```

```python
>>> I = Quaternion(b=1)
>>> str(I)
'0.00 + 1.00 i + 0.00 j + 0.00 k'
```

To represent with a string by default, overload \texttt{repr()}:

```python
def \_repr\_(self):
    "Quaternion Representation is a String"
    return str(self)
```

Typing \texttt{I} gives now \(0.00 + 1.00 i + 0.00 j + 0.00 k\).
Representing Quaternions

The builtin method \texttt{str()} converts to a string.

```python
def __str__(self):
    "The string representation of a Quaternion"
    return '%.2f + %.2f i + %.2f j + %.2f k' \
    % (self.a, self.b, self.c, self.d)
```

```python
>>> 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 \texttt{repr()}:

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

Typing \texttt{I} gives now \texttt{0.00 + 1.00 i + 0.00 j + 0.00 k}.
Representing Quaternions

The builtin method \texttt{str()} converts to a string.

def \_\_str\_\_(self):
    "The string representation of a Quaternion"
    return "\%.2f + \%.2f i + \%.2f j + \%.2f k" \n    % (self.a, self.b, self.c, self.d)

>>> 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 \texttt{repr()}:

def \_\_repr\_\_(self):
    "Quaternion Representation is a String"
    return str(self)

Typing \texttt{I} gives now 0.00 + 1.00 i + 0.00 j + 0.00 k.
With the built-in `==` operator:

```python
>>> 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

motivation

With the built-in == operator:

```python
>>> 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

motivation

With the builtin `==` operator:

```python
>>> 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.
With the builtin == operator:

```python
>>> 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

With the builtin \(==\) operator:

```python
>>> 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) \texttt{True} as answer!

It takes two to compare, besides \texttt{self}, we need the other.
Overloading Equality

def __eq__(self, other):
    "Defines Quaternion Equality"
    return self.a == other.a and \
    self.b == other.b and \
    self.c == other.c and \
    self.d == other.d

Now, we will have True:

```python
>>> x = Quaternion(1)
>>> y = Quaternion(1,0)
>>> x == y
True
```
Overloading Equality
defining __eq__

The other is like self, used in binary operators:

```python
def __eq__(self, other):
    "Defines Quaternion Equality"
    return self.a == other.a and \\
    self.b == other.b and \\
    self.c == other.c and \\
    self.d == other.d
```

Now, we will have True:

```python
>>> 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.: $z = -x$

```python
def __neg__(self):
    "Negation of a Quaternion"
    return Quaternion(-self.a,
                       -self.b,
                       -self.c,
                       -self.d)
```

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

---

**Quaternions**

- Hypercomplex numbers
- Application in computer graphics

**Operator Overloading**

- The class Quaternion defining functional attributes

**Summary + Assignments**
Overloading Arithmetical Operators

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

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

```python
    def __neg__(self):
        "Negation of a Quaternion"
        return Quaternion(-self.a,
                       -self.b,
                       -self.c,
                       -self.d)
```

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

Overloading Arithmetical Operators

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

Consider first the negation, i.e.: \( z = -x \)

```python
def __neg__(self):
    "Negation of a Quaternion"
    return Quaternion(-self.a,
                      -self.b,
                      -self.c,
                      -self.d)
```

Defining \_
\_\_neg\_
\_\_ we overloaded the negation \( - \).
Overloading Addition
the other and inplace addition

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

```python
def __add__(self, other):
    "Adding two Quaternions"
    return Quaternion(self.a + other.a, \
                      self.b + other.b, \
                      self.c + other.c, \
                      self.d + other.d)
```

We also want to update quaternions as `z += x`, this inplace addition is overloaded by

```python
def __iadd__(self, other):
    "Inplace Adder of two Quaternions"
    return Quaternion(self.a + other.a, \
                      self.b + other.b, \
                      self.c + other.c, \
                      self.d + other.d)
```
Overloading Addition
the other and inplace addition

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

```python
def __add__(self, other):
    "Adding two Quaternions"
    return Quaternion(self.a+other.a, \\
                      self.b+other.b, \\
                      self.c+other.c, \\
                      self.d+other.d)
```

We also want to update quaternions as \( z += x \),
this inplace addition is overloaded by

```python
def __iadd__(self, other):
    "Inplace Adder of two Quaternions"
    return Quaternion(self.a+other.a, \\
                      self.b+other.b, \\
                      self.c+other.c, \\
                      self.d+other.d)
```
The Inverse of a Quaternion

def __abs__(self):
    "Returns the Magnitude of a Quaternion"
    from math import sqrt
    return sqrt(self.a**2 + self.b**2 + self.c**2 + self.d**2)

def conjugate(self):
    "Returns the conjugate of a Quaternion"
    return Quaternion(self.a,-self.b,-self.c,-self.d)

def scamul(self,x):
    "Product of Quaternion with Scalar x"
    return Quaternion(self.a*x, self.b*x, self.c*x, self.d*x)

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

def __abs__(self):
    "Returns the Magnitude of a Quaternion"
    from math import sqrt
    return sqrt(self.a**2 + self.b**2 \ + self.c**2 + self.d**2)

def conjugate(self):
    "Returns the conjugate of a Quaternion"
    return Quaternion(self.a, -self.b, -self.c, \ -self.d)

def scamul(self, x):
    "Product of Quaternion with Scalar x"
    return Quaternion(self.a*x, self.b*x, \ self.c*x, self.d*x)

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

```python
def __abs__(self):
    "Returns the Magnitude of a Quaternion"
    from math import sqrt
    return sqrt(self.a**2 + self.b**2 \\
                 + self.c**2 + self.d**2)

def conjugate(self):
    "Returns the conjugate of a Quaternion"
    return Quaternion(self.a,-self.b,-self.c,\
                      -self.d)

def scamul(self,x):
    "Product of Quaternion with Scalar x"
    return Quaternion(self.a*x, self.b*x, \
                       self.c*x, self.d*x)

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

def __abs__(self):
    "Returns the Magnitude of a Quaternion"
    from math import sqrt
    return sqrt(self.a**2 + self.b**2 + self.c**2 + self.d**2)

def conjugate(self):
    "Returns the conjugate of a Quaternion"
    return Quaternion(self.a,-self.b,-self.c,-self.d)

def scamul(self,x):
    "Product of Quaternion with Scalar x"
    return Quaternion(self.a*x, self.b*x, self.c*x, self.d*x)

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

We covered more of chapter 10 in *Python Power!* read §6.5 in *Computer Science, an overview*

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. Use operator overloading in the Python code for the class `Rational` (exercise of Lecture 24).