Algolab (index.html#Chapters)

Out[2]:

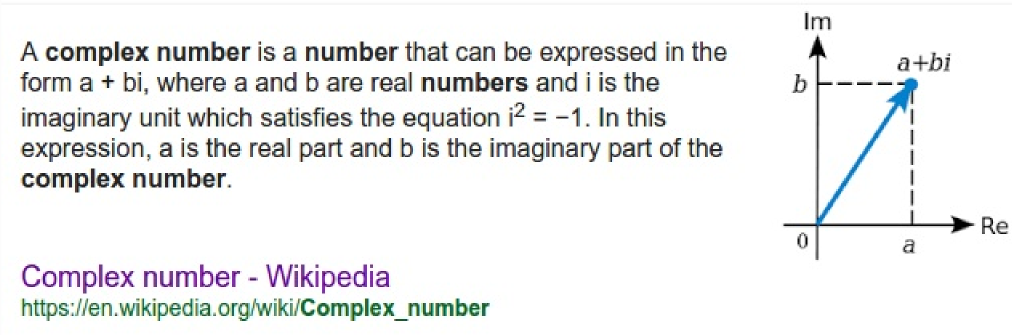
Chapter 3: Data Structures

Chapter 3: Data Structures

class ComplexNumber

See theory here: [http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf (http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf)](http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf) (First slides until `class Fraction`)

Let's try to define a complex number:



As the `Fraction` class, the `ComplexNumber` holds two values, in this case one for the *real* part and one for the *imaginary* one.

- Note each method takes as first import `self` argument. `self` will always be a reference to the object itself, and allows accessing its fields and methods
- `self` is not a keyword of Python, you could use any name you want for the first parameter, but it is much better to follow conventions and stick using `self` !
- Methods beginning and ending with double underscore '__' have often special meaning in Python: if you see such a method around, it means it is overriding some default behaviour of Python

In [3]:

```
import unittest
import math

class ComplexNumber:

    def __init__(self, real, imaginary):
        """ Initializes the complex number with real and imaginary part """
        self.real = real
        self.imaginary = imaginary

    def magnitude(self):
        """ Returns a float which is the magnitude (that is, the absolute value) of
the complex number

            This method is something we introduce by ourselves, according to the def
inition:
            https://en.wikipedia.org/wiki/Complex_number#Absolute_value_and_argument
        """
        return math.sqrt(self.real**2 + self.imaginary**2)

    def __str__(self):
        """ Returns a string representation of the object, overriding default Python
 behaviour. """
        return str(self.real) + " + " + str(self.imaginary) + "i"
```

```python
class ComplexNumberTest(unittest.TestCase):
    """ Test cases for ComplexNumber

          Note this is a *completely* separated class from ComplexNumber and
          we declare it here just for testing purposes!
          The 'self' you see here have nothing to do with the selfs from the
          ComplexNumber methods!
    """

    def test_init(self):
        self.assertEqual(ComplexNumber(1,2).real, 1)
        self.assertEqual(ComplexNumber(1,2).imaginary, 2)

    def test_magnitude(self):
        """
            Notice we can't use assertEqual, as the result of magnitude() is a
            float number which may have floating point rounding errors. So it's
            necessary to use assertAlmostEqual
            As an option with the delta you can declare the precision you require.
            For more info see Python docs:
            https://docs.python.org/2/library/unittest.html#unittest.TestCase.assert
AlmostEqual

            NOTE: assertEqual might still work on your machine but just DO NOT use i
t
            for float numbers!!!
        """
        # self.assertEqual(ComplexNumber(3.0,4.0).magnitude(),5)
        self.assertAlmostEqual(ComplexNumber(3.0,4.0).magnitude(),5, delta=0.001)


    def test_str(self):
        self.assertEqual(str(ComplexNumber(1,2)), "1 + 2i")
        #self.assertEqual(str(ComplexNumber(1,0)), "1")
        #self.assertEqual(str(ComplexNumber(1.0,0)), "1.0")
        #self.assertEqual(str(ComplexNumber(0,1)), "i")
        #self.assertEqual(str(ComplexNumber(0,0)), "0")
```

In [4]:

```python
algolab.run(ComplexNumberTest)
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.007s

OK
```

Once the `__init__` method is defined, we can create a ComplexNumber with a call like 'ComplexNumber(3,5)'

Notice in the constructor call we *do not* pass anything as `self` parameter (after all, we are creating the object)

In [5]:

```python
my_complex = ComplexNumber(3,5)
```

We can now try to use one of the methods we defined:

In [6]:

```
mag = my_complex.magnitude()
print mag
```

```
5.83095189485
```

We can also pretty print the whole complex number. Internally, `print` function will look if the `ComplexNumber` has defined an `__str__` method. If so, it will pass to the method the instance `my_complex` as the first argument , which in our methods will end up in the `self` parameter:

In [7]:

```
print my_complex
```

```
3 + 5i
```

Ok, now we are ready to define our own stuff.

Complex numbers equality

Equality [edit]

Two complex numbers are equal if and only if both their real and imaginary parts are equal. In symbols:

$$z_1 = z_2 \;\leftrightarrow\; (\mathrm{Re}(z_1) = \mathrm{Re}(z_2) \wedge \mathrm{Im}(z_1) = \mathrm{Im}(z_2)).$$

Here we will try to give you a glimpse of some aspectes related to Python equality, and trying to respect interfaces when overriding methods. Equality can be a nasty subject, here we will treat it in a simplified form.

- Can you try to implement equality for ComplexNumber more or less as it was done for Fraction in theory slides?

  use

  ```
  def __eq__(self, other):
  ```

  and write some tests as well!
- Beware 'equality' is tricky in Python for float numbers! Rule of thumb: when overriding __eq__, use 'dumb' equality, two things are the same only if their parts are literally equal
- If instead you need to determine if two objects are similar, define other 'closeness' functions.
- (Non mandatory read) if you are interested in the gory details of equality, see
  - How to Override comparison operators in Python (http://jcalderone.livejournal.com/32837.html)
  - Messing with hashing (http://www.asmeurer.com/blog/posts/what-happens-when-you-mess-with-hashing-in-python/)

Use this simple test case to check for equality:

```
def test_integer_equality(self):
        """
            Note all other tests depend on this test !

            We want also to test the constructor, so in c we set stuff by han
d
        """
        c = ComplexNumber(0,0)
        c.real = 1
        c.imaginary = 2
        self.assertEquals(c, ComplexNumber(1,2))
```

Complex numbers addition

Complex numbers are added by separately adding the real and imaginary parts of the summands. That is to say:

$$(a+bi)+(c+di)=(a+c)+(b+d)i.$$

Similarly, subtraction is defined by

$$(a+bi)-(c+di)=(a-c)+(b-d)i.$$

- Can you try to implement addition for ComplexNumber more or less as it was done for Fraction in theory slides? Write some tests as well!
- a and c correspond to `real`, b and d correspond to `imaginary`

use

```
def __add__(self, other):
```

In [8]:

```
import unittest

class ComplexNumberTest(unittest.TestCase):

    def test_add(self):
        assertEquals(ComplexNumber(1,2) + ComplexNumber(3,4), ComplexNumber(3,6));
```

Adding a scalar

We defined addition among ComplexNumbers, but what about addition among a ComplexNumber and an `int` or a `float`?

Will this work?

```
ComplexNumber(3,4) + 5
```

What about this?

```
ComplexNumber(3,4) + 5.0
```

Try to add the following method to your class, and check if it does work with the scalar:

In [9]:

```
    def __add__(self, other):
         # checks other object is instance of the class ComplexNumber
        if isinstance(other, ComplexNumber):
            return ComplexNumber(self.real + other.real,self.imaginary + other.imagi
nary)

        # else checks the basic type of other is int or float
        elif type(other) is int or type(other) is float:
            return ComplexNumber(self.real + other, self.imaginary)

        # other is of some type we don't know how to process.
        # In this case the Python specs say we MUST return 'NotImplemented'
        else:
            return NotImplemented
```

Hopefully now you have a better add. But what about this? Will this work?

```
5 + ComplexNumber(3,4)
```

Answer: it won't, Python needs further instructions. Usually Python tries to see if the class of the object on left of the expression defines addition for operands *to the right* of it. In this case on the left we have a `float` number, and float numbers don't define any way to deal to the right with your very own `ComplexNumber` class. So as a last resort Python tries to see if your `ComplexNumber` class has defined also a way to deal with operands *to the left* of the `ComplexNumber`, by looking for the method `__radd__` , which means *reverse addition* . Here we implement it :

```
def __radd__(self, other):
        """ Returns the result of expressions like    other + self      """
        if (type(other) is int or type(other) is float):
            return ComplexNumber(self.real + other, self.imaginary)
        else:
            return NotImplemented
```

To check it is working and everything is in order for addition, add these test cases:

```
def test_add_zero(self):
        self.assertEquals(ComplexNumber(1,2) + ComplexNumber(0,0), ComplexNum
ber(1,2));

    def test_add_numbers(self):
        self.assertEquals(ComplexNumber(1,2) + ComplexNumber(3,4), ComplexNum
ber(4,6));

    def test_add_scalar_right(self):
        self.assertEquals(ComplexNumber(1,2) + 3, ComplexNumber(4,2));


    def test_add_scalar_left(self):
        self.assertEquals(3 + ComplexNumber(1,2), ComplexNumber(4,2));


    def test_add_negative(self):
        self.assertEquals(ComplexNumber(-1,0) + ComplexNumber(0,-1), ComplexN
umber(-1,-1));
```

Complex numbers multiplication

Multiplication and division [edit]

The multiplication of two complex numbers is defined by the following formula:

$$(a+bi)(c+di) = (ac-bd) + (bc+ad)i.$$

In particular, the square of the imaginary unit is −1:

$$i^2 = i \times i = -1.$$

- Can you try to implement multiplication for ComplexNumber more or less as it was done for Fraction in theory slides?
- Can you extend multiplication to wor with scalars (both left and right) as well?

use

```
def __mul__(self, other):
```

Use these test cases:

```
def test_mul_by_zero(self):
        self.assertEquals(ComplexNumber(0,0) * ComplexNumber(1,2), ComplexNum
ber(0,0));

    def test_mul_just_real(self):
        self.assertEquals(ComplexNumber(1,0) * ComplexNumber(2,0), ComplexNum
ber(2,0));

    def test_mul_just_imaginary(self):
        self.assertEquals(ComplexNumber(0,1) * ComplexNumber(0,2), ComplexNum
ber(-2,0));

    def test_mul_scalar_right(self):
        self.assertEquals(ComplexNumber(1,2) * 3, ComplexNumber(3,6));

    def test_mul_scalar_left(self):
        self.assertEquals(3 * ComplexNumber(1,2), ComplexNumber(3,6));
```

Solutions

ComplexNumber Solution

In [10]:

```
import unittest

class ComplexNumber:

    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary

    def __str__(self):
        return str(self.real) + " + " + str(self.imaginary) + "i"

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

    def __add__(self, other):
        if isinstance(other, ComplexNumber):
            return ComplexNumber(self.real + other.real,self.imaginary + other.imagi
```

```
            return ComplexNumber(self.real + other.real,self.imaginary + other.imagi
nary)
        elif type(other) is int or type(other) is float:
            return ComplexNumber(self.real + other, self.imaginary)
        else:
            return NotImplemented


    def __radd__(self, other):
        if (type(other) is int or type(other) is float):
            return ComplexNumber(self.real + other, self.imaginary)
        else:
            return NotImplemented


    def __mul__(self, other):

        if isinstance(other, ComplexNumber):
            return ComplexNumber(self.real * other.real - self.imaginary * other.ima
ginary,
                                 self.imaginary * other.real + self.real * other.ima
ginary)
        elif type(other) is int or type(other) is float:
            return ComplexNumber(self.real * other, self.imaginary * other)
        else:
            return NotImplemented

    def __rmul__(self, other):
        if (type(other) is int or type(other) is float):
            return ComplexNumber(self.real * other, self.imaginary * other)
        else:
            return NotImplemented


class ComplexNumberTest(unittest.TestCase):


    def test_integer_equality(self):
        """
            Note all other tests depend on this test !

            We want also to test the constructor, so in c we set stuff by hand
        """
        c = ComplexNumber(0,0)
        c.real = 1
        c.imaginary = 2
        self.assertEquals(c, ComplexNumber(1,2))

    def test_add_zero(self):
        self.assertEquals(ComplexNumber(1,2) + ComplexNumber(0,0), ComplexNumber(1,2
));

    def test_add_numbers(self):
        self.assertEquals(ComplexNumber(1,2) + ComplexNumber(3,4), ComplexNumber(4,6
));

    def test_add_scalar_right(self):
        self.assertEquals(ComplexNumber(1,2) + 3, ComplexNumber(4,2));

    def test_add_scalar_left(self):
        self.assertEquals(3 + ComplexNumber(1,2), ComplexNumber(4,2));

    def test_add_negative(self):
        self.assertEquals(ComplexNumber(-1,0) + ComplexNumber(0,-1), ComplexNumber(-
1,-1));
```

```
    def test_mul_by_zero(self):
        self.assertEquals(ComplexNumber(0,0) * ComplexNumber(1,2), ComplexNumber(0,0
));

    def test_mul_just_real(self):
        self.assertEquals(ComplexNumber(1,0) * ComplexNumber(2,0), ComplexNumber(2,0
));

    def test_mul_just_imaginary(self):
        self.assertEquals(ComplexNumber(0,1) * ComplexNumber(0,2), ComplexNumber(-2,
0));

    def test_mul_scalar_right(self):
        self.assertEquals(ComplexNumber(1,2) * 3, ComplexNumber(3,6));

    def test_mul_scalar_left(self):
        self.assertEquals(3 * ComplexNumber(1,2), ComplexNumber(3,6));
```

In [11]:

```
algolab.run(ComplexNumberTest)
```

```
...........
----------------------------------------------------------------------
Ran 11 tests in 0.018s

OK
```