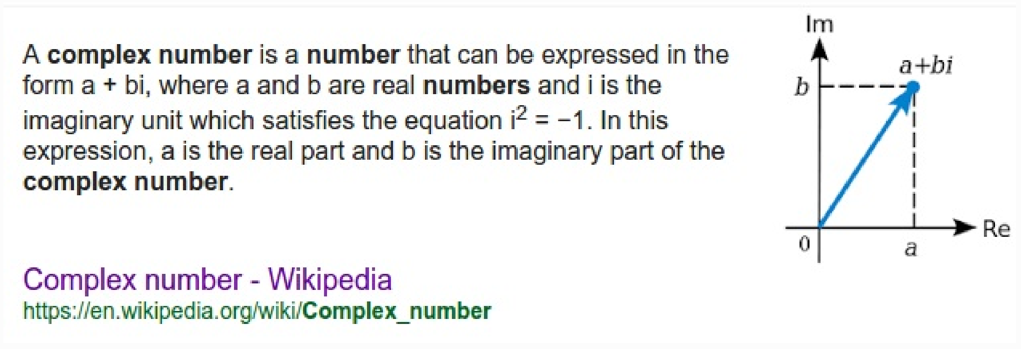
Chapter 2.1: Data Structures - ComplexNumber

1. Abstract Data Types (ADT) Theory

1.1. Intro

- Theory from the slides: 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` included)
- Object Oriented programming on the the book (http://interactivepython.org/runestone/static/pythonds/Introduction/ObjectOrientedProgramminginPythonDefin (In particular, Fraction class (http://interactivepython.org/runestone/static/pythonds/Introduction/ObjectOrientedProgramminginPythonDefin fraction-class), in this course we won't focus on inheritance)

1.2. Complex number theory


A **complex number** is a **number** that can be expressed in the form a + bi, where a and b are real **numbers** and i is the imaginary unit which satisfies the equation $i^2 = -1$. In this expression, a is the real part and b is the imaginary part of the **complex number**.

Complex number - Wikipedia
https://en.wikipedia.org/wiki/**Complex_number**

1.3. Datatypes the old way

From the definition we see that to identify a complex number **we need two float values** . One number is for the ***real* part**, and another number is for the ***imaginary* part**.

How can we represent this in Python? So far, you saw there are many ways to put two numbers together. One way could be to put as items in a list of two elements, and implicitly assume the first one is the *real* and the second the *imaginary* part:

In [2]:

```
c = [3.0, 5.0]
```

Or we could use a tuple:

In [3]:

```
c = (3.0, 5.0)
```

A problem with the previous representations is that a casual observer might not know exactly the meaning of the two numbers. We could be more explicit and store the values into a dictionary, using keys to identify the two parts:

In [4]:

```
c = {'real': 3.0, 'imaginary': 5.0}
```

In [5]:

```
print c
```

```
{'real': 3.0, 'imaginary': 5.0}
```

In [6]:

```
print c['real']
```

```
3.0
```

In [7]:

```
print c['imaginary']
```

```
5.0
```

Now, writing the whole record `{'real': 3.0, 'imaginary': 5.0}` each time we want to create a complex number might be annoying and error prone. To help us, we can create a little shortcut function named `complex_number` that creates and returns the dictionary:

In [8]:

```
def complex_number(real, imaginary):
    d = {}
    d['real'] = real
    d['imaginary'] = imaginary
    return d
```

In [9]:

```
c = complex_number(3.0, 5.0)
```

In [10]:

```
print c
```

```
{'real': 3.0, 'imaginary': 5.0}
```

To do something with our dictionary, we would then define functions, like for example `complex_str` to show them nicely:

In [11]:

```
def complex_str(cn):
    return str(cn['real']) + " + " + str(cn['imaginary']) + "i"
```

In [12]:

```
c = complex_number(3.0, 5.0)
print complex_str(c)
```

```
3.0 + 5.0i
```

We could do something more complex, like defining the `phase` of the complex number which returns a `float`:

> **IMPORTANT: In these exercises, we care about programming, not complex numbers theory. There's no need to break your head over formulas!**

In [14]:

```
import math
def phase(cn):
        """ Returns a float which is the phase (that is, the vector angle) of the complex numb
er

            See definition: https://en.wikipedia.org/wiki/Complex_number#Absolute_value_and_ar
gument
        """
        return math.atan2(cn['imaginary'], cn['real'])
```

In [15]:

```
c = complex_number(3.0, 5.0)
print phase(c)
```

```
1.03037682652
```

We could even define functions that that take the complex number and some other parameter, for example we could define the `log` of complex numbers, which return another complex number (mathematically it would be infinitely many, but we just pick the first one in the series):

In [16]:

```
import math
def log(cn, base):
        """ Returns another complex number which is the logarithm of this complex number

            See definition (accomodated for generic base b):
            https://en.wikipedia.org/wiki/Complex_number#Natural_logarithm
        """
        return {'real':math.log(cn['real']) / math.log(base),
                'imaginary' : phase(cn) / math.log(base)}
```

In [17]:

```
print log(c,2)
```

```
{'real': 1.5849625007211563, 'imaginary': 1.4865195378735334}
```

You see we got our dictionary representing a complex number. If we want a nicer display we can call on it the `complex_str` we defined:

In [18]:

```
print complex_str(log(c,2))
```

```
1.58496250072 + 1.48651953787i
```

1.4. Finding the pattern

So, what have we done so far?

1) Decided a data format for the complex number, saw that the dictionary is quite convenient

2) Defined a function to quickly create the dictionary:

```
def complex_number(real, imaginary):
```

3) Defined some function like `phase` and `log` to do stuff on the complex number

```
def phase(cn):
def log(cn, base):
```

4) Defined a function `complex_str` to express the complex number as a readable string:

```
def complex_str(cn):
```

Notice that:

- all functions above take a `cn` complex number dictionary as first parameter
- the functions `phase` and `log` are quite peculiar to complex number, and to know what they do you need to have deep knowledge of what a complex number is.
- the function `complex_str` is more intuitive, because it covers the common need of giving a nice string representation to the data format we just defined. Also, we used the word `str` as part of the name to give a hint to the reader that probably the function behaves in a way similar to the Python function `str()`.

When we encounter a new datatype in our programs, we often follow the procedure of thinking listed above. Such procedure is so common that software engineering people though convenient to provide a specific programming paradigm to represent it, called *Object Oriented* programming. We are now going to rewrite the complex number example using such paradigm.

1.5. Object Oriented programming

In object oriented programming, we usually

1. Introduce new datatypes by declaring a *class*, named for example `ComplexNumber`
2. Are given a dictionary and define how data is stored in the dictionary (i.e. in fields `real` and `imaginary`)
3. Define a way to *construct* specific *instances* , like `3 + 2i`, `5 + 6i` (instances are also called *objects*)
4. Define some *methods* to operate on the *instances* (like `phase`)
5. Define some special *methods* to customize how Python treats *instances* (for example for displaying them as strings when printing)

Let's now create our first *class*.

2. ComplexNumber class

2.1. Class declaration

A minimal class declaration will at least declare the class name and the `__init__` method:

In [19]:

```
class ComplexNumber:

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

Here we declare to Python that we are starting defining a template for a new *class* called `ComplexNumber`. This template will hold a collection of functions (called methods) that manipulate *instances* of complex numbers (instances are `1.0 + 2.0i`, `3.0 + 4.0i`, ...).

IMPORTANT: Although classes can have any name (i.e. `complex_number`, `complexNumber`, ...), by convention you *SHOULD* use a camel cased name like `ComplexNumber`, with capital letters as initials and no underscores.

2.2. Constructor `__init__`

With the dictonary model, to create complex numbers remember we defined that small utility function `complex_number`, where inside we were creating the dictionary:

```
def complex_number(real, imaginary):
    d = {}
    d['real'] = real
    d['imaginary'] = imaginary
    return d
```

With classes, to create objects we have instead to define a so-called *constructor method* called `__init__`:

In [21]:

```
class ComplexNumber:

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

`__init__` is a very special method, that has the job to initialize an *instance* of a complex number. It has three important features:

a) it is defined like a function, inside the `ComplexNumber` declaration (as usual, indentation matters!)

b) it always takes as first parameter `self`, which is an instance of a special kind of dictionary that will hold the fields of the complex number. Inside the previous `complex_number` function, we were creating a dictionary d. In `__init__` method, the dictionary instead is automatically created by Python and given to us in the form of parameter `self`

c) `__init__` does not return anything: this is different from the previous `complex_number` function where instead we were returning the dictionary `d`.

Later we will explain better these properties. For now, let's just concentrate on the names of things we see in the declaration.

WARNING: There can be only one constructor method per class, and MUST be named `__init__`

WARNING: `__init__` MUST take at least one parameter, by convention it is usually named `self`

IMPORTANT: `self` is just a name we give to the first parameter. It could be any name our fantasy suggest and the program would behave exactly the same!

If the editor you are using will evidence it in some special color, it is because it is aware of the convention but *not* because `self` is some special Python keyword.

IMPORTANT: In general, any of the `__init__` parameters can have completely arbitrary names, so for example the following code snippet would work exactly the same as the initial definition:

In [23]:

```
class ComplexNumber:

    def __init__(donald_duck, mickey_mouse, goofy):
        donald_duck.real = mickey_mouse
        donald_duck.imaginary = goofy
```

Once the `__init__` method is defined, we can create a specific `ComplexNumber` *instance* with a call like this:

In [24]:

```
c = ComplexNumber(3.0,5.0)
print c
```

```
<__main__.ComplexNumber instance at 0x7f30c951a5f0>
```

What happend here?

init 2.2.1) We told Python we want to create a new particular *instance* of the template defined by *class* `ComplexNumber`. As parameters for the instance we indicated `3.0` and `5.0`.

WARNING: to create the instance, we used the name of the class `ComplexNumber` following it by an open round parenthesis and parameters like a function call: `c=ComplexNumber(3.0,5.0)`

Writing just: `c = ComplexNumber` would *NOT* instantiate anything and we would end up messing with the *template `ComplexNumber`*, which is a collection of functions for complex numbers.

init 2.2.2) Python created a new special dictionary for the instance

init 2.2.3) Python passed the special dictionary as first parameter of the method `__init__`, so it will be bound to parameter `self`. As second and third arguments passed *3.0* and *5.0*, which will be bound respectively to parameters `real` and `imaginary`

> **WARNING: When instantiating an object with a call like `c=ComplexNumber(3.0,5.0)` you *don't* need to pass a dictionary as first parameter! Python will implicitly create it and pass it as first parameter to `__init__`**

init 2.2.4) In the `__init__` method, the instructions

```
self.real = real
self.imaginary = imaginary
```

first create a key in the dictionary called `real` associating to the key the value of the parameter `real` (in the call is *3.0*). Then the value *5.0* is bound to the key `imaginary`.

> **IMPORTANT: we said Python provides `__init__` with a special kind of dictionary as first parameter. One of the reason it is special is that you can access keys using the dot like `self.my_key`. With ordinary dictionaries you would have to write the brackets like `self["my_key"]`**

> **IMPORTANT: like with dictionaries, we can arbitrarily choose the name of the keys, and which values to associate to them.**

> **IMPORTANT: In the following, we will often refer to keys of the `self` dictionary with the terms *field*, and/or *attribute*.**

Now one important word of wisdom:

> **!!!!!! COMMANDMENT: NEVER EVER REASSIGN `self` !!!!!!! :**

Since self is a kind of dictionary, you might be tempted to do like this:

In [29]:

```
class EvilComplexNumber:
    def __init__(self, real, imaginary):
        self = {'real':real, 'imaginary':imaginary}
```

but to the outside world this will bring no effect. For example, let's say somebody from outside makes a call like this:

In [30]:

```
ce = EvilComplexNumber(3.0, 5.0)
```

At the first attempt of accessing any field, you would get an error because after the initalization `c` will point to the yet untouched `self` created by Python, and not to your dictionary (which at this point will be simply lost):

```
print ce.real
```

AttributeError: EvilComplexNumber instance has no attribute 'real'

In general, you *DO NOT* reassign `self` to anything. Here are other example *DON'Ts*:

```
self = ['666']  # self is only supposed to be a sort of dictionary which is passed by Py
thon
self = 6        # self is only supposed to be a sort of dictionary which is passed by Py
thon</p>
```

init 2.2.5) Python automatically returns from `__init__` the special dictionary `self`

> **WARNING: `__init__` must **NOT** have a `return` statement ! Python will implicitly return `self` !**

init 2.2.6) The result of the call (so the special dictionary) is bound to external variable 'c`:

```
c = ComplexNumber(3.0, 5.0)
```

init 2.2.7) You can then start using `c` as any variable

In [32]:

```
print c
```

```
<__main__.ComplexNumber instance at 0x7f30c951a5f0>
```

From the output, you see we have indeed an *instance* of the *class* `ComplexNumber`. To see the difference between *instance* and *class*, you can try printing the *class* instead:

In [33]:

```
print ComplexNumber
```

```
__main__.ComplexNumber
```

> **IMPORTANT: You can create an infinite number of different *instances* (i.e. `ComplexNumber(1.0, 1.0)`, `ComplexNumber(2.0, 2.0)`, `ComplexNumber(3.0, 3.0)`, ...), but you will have only one *class* definition for them (`ComplexNumber`).**

We can now access the fields of the special dictionary by using the dot notation as we were doing with the 'self`:

In [35]:

```
print c.real
```

```
3.0
```

In [36]:

```
print c.imaginary
```

```
5.0
```

If we want, we can also change them:

In [37]:

```
c.real = 6.0
print c.real
```

```
6.0
```

2.3. Defining methods

Let's make our class more interesting by adding the method `phase(self)` to operate on the complex number:

In [38]:

```
import unittest
import math

class ComplexNumber:

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

    def phase(self):
        """ Returns a float which is the phase (that is, the vector angle) of the complex numb
er

            This method is something we introduce by ourselves, according to the definition:
            https://en.wikipedia.org/wiki/Complex_number#Absolute_value_and_argument
        """
        return math.atan2(self.imaginary, self.real)
```

The method takes as first parameter `self` which again is a special dictionary. We expect the dictionary to have already been initialized with some values for `real` and `imaginary` fields. We can access them with the dot notation as we did before:

```
return math.atan2(self.imaginary, self.real)
```

How can we call the method on instances of complex numbers? We can access the method name from an instance using the dot notation as we did with other keys:

In [39]:

```
c = ComplexNumber(3.0,5.0)
print c.phase()
```

```
1.03037682652
```

What happens here?

By writing `c.phase()` , we call the method `phase(self)` which we just defined. The method expects as first parameter `self` a class instance, but in the call `c.phase()` apparently we don't provide any parameter. Here some magic is going on, and Python implicitly is passing as first parameter the special dictionary bound to `c`. Then it executes the method and returns the desired float.

> **WARNING: when *calling* a method, you MUST put the round parenthesis after the method name like in `c.phase()` !**
>
> **If you just write `c.phase` without parenthesis you will get back an address to the physical location of the method code:**
>
> ```
>     >>> c.phase
>
>     <bound method ComplexNumber.phase of <__main__.ComplexNumber instance at
> 0xb465a4cc>>
> ```

We can also define methods that take more than one parameter, and also that create and return `ComplexNumber` instances, like for example the method `log(self, base)`:

In [41]:

```
import math

class ComplexNumber:

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

    def phase(self):
        """ Returns a float which is the phase (that is, the vector angle) of the complex numb
er

            This method is something we introduce by ourselves, according to the definition:
            https://en.wikipedia.org/wiki/Complex_number#Absolute_value_and_argument
        """
        return math.atan2(self.imaginary, self.real)

    def log(self, base):
        """ Returns another ComplexNumber which is the logarithm of this complex number

            This method is something we introduce by ourselves, according to the definition:
            (accomodated for generic base b)
            https://en.wikipedia.org/wiki/Complex_number#Natural_logarithm
        """
        return ComplexNumber(math.log(self.real) / math.log(base), self.phase() / math.log(bas
e))
```

> **WARNING: *ALL* METHODS MUST HAVE AT LEAST ONE PARAMETER, WHICH BY CONVENTION IS NAMED `self` !**

To call `log`, you can do as with `phase` but this time you will need also to pass one parameter for the `base` parameter, in this case we use the exponential `math.e`:

In [43]:

```
c = ComplexNumber(3.0, 5.0)
logarithm = c.log(math.e)
```

> **WARNING: As before for `phase`, notice we didn't pass any dictionary as first parameter! Python will implicitly pass as first argument the instance `c` as `self`, and `math.e` as `base`**

In [45]:

```
print logarithm
```

```
<__main__.ComplexNumber instance at 0x7f30c951a050>
```

To see if the method worked and we got back we got back a different complex number, we can print the single fields:

In [46]:

```
print logarithm.real
```

```
1.09861228867
```

In [47]:

```
print logarithm.imaginary
```

```
1.03037682652
```

2.4. A better print with `__str__`

As we said, printing is not so informative:

In [48]:

```
print ComplexNumber(3.0, 5.0)
```

```
<__main__.ComplexNumber instance at 0x7f30c951de60>
```

It would be nice to instruct Python to express the number like "*3.0 + 5.0i*" whenever we want to see the `ComplexNumber` represented as a string. How can we do it? Luckily for us, defining the `__str__(self) method` will do the magic (see bottom of class definition):

In [49]:

```
import math

class ComplexNumber:

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

    def phase(self):
        """ Returns a float which is the phase (that is, the vector angle) of the complex numb
er

            This method is something we introduce by ourselves, according to the definition:
            https://en.wikipedia.org/wiki/Complex_number#Absolute_value_and_argument
        """
        return math.atan2(self.imaginary, self.real)

    def log(self, base):
        """ Returns another ComplexNumber which is the logarithm of this complex number

            This method is something we introduce by ourselves, according to the definition:
            (accomodated for generic base b)
            https://en.wikipedia.org/wiki/Complex_number#Natural_logarithm
        """
        return ComplexNumber(math.log(self.real) / math.log(base), self.phase() / math.log(bas
e))

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

IMPORTANT: all methods starting and ending with a double underscore `__` have a special meaning in Python: depending on their name, they override some default behaviour. In this case, with `__str__` we are overriding how Python represents a `ComplexNumber` instance into a string.

WARNING: Since we are overriding Python default behaviour, it is very important that we follow the specs of the method we are overriding *to the letter*. In our case, the specs for __str__ (https://docs.python.org/2/reference/datamodel.html#object.__str__) obviously state you MUST return a string.

In [51]:

```
c = ComplexNumber(3.0, 5.0)
```

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

In [52]:

```
print c
```

```
3.0 + 5.0i
```

In [53]:

```
print c.log(2)
```

```
1.58496250072 + 1.48651953787i
```

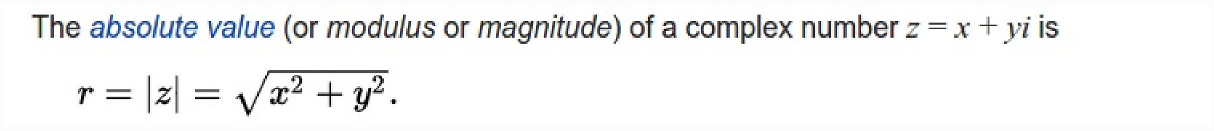
2.5. ComplexNumber code skeleton

We are now ready to write methods on our own. Create a new file and copy paste the following skeleton, including the tests, then proceed doing the exercises.

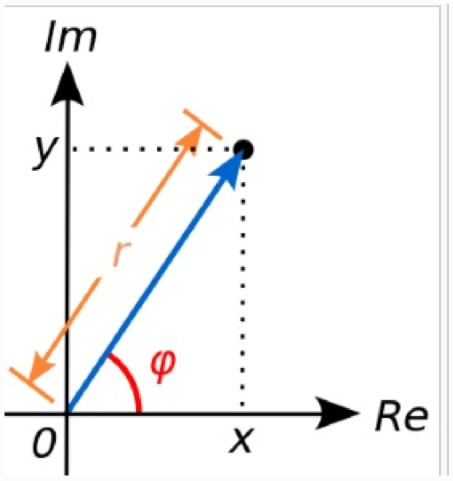
In [54]:

```
import unittest
import math

class ComplexNumber:

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

    def phase(self):
        """ Returns a float which is the phase (that is, the vector angle) of the complex numb
er

            This method is something we introduce by ourselves, according to the definition:
            https://en.wikipedia.org/wiki/Complex_number#Absolute_value_and_argument
        """
        return math.atan2(self.imaginary, self.real)

    def log(self, base):
        """ Returns another ComplexNumber which is the logarithm of this complex number

            This method is something we introduce by ourselves, according to the definition:
            (accomodated for generic base b)
            https://en.wikipedia.org/wiki/Complex_number#Natural_logarithm
        """
        return ComplexNumber(math.log(self.real) / math.log(base), self.phase() / math.log(bas
e))


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


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_phase(self):
        """
            NOTE: we can't use assertEqual, as the result of phase() 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.assertAlmostEqua
l

            NOTE: assertEqual might still work on your machine but just DO NOT use it
            for float numbers!!!
        """
        self.assertAlmostEqual(ComplexNumber(0.0,1.0).phase(), math.pi / 2, 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")


    def test_log(self):
        c = ComplexNumber(1.0,1.0)
        l = c.log(math.e)
        self.assertAlmostEqual(l.real, 0.0, delta=0.001)
        self.assertAlmostEqual(l.imaginary, c.phase(), delta=0.001)
```

2.6. Complex numbers magnitude


The *absolute value* (or *modulus* or *magnitude*) of a complex number $z = x + yi$ is

$$r = |z| = \sqrt{x^2 + y^2}.$$



Implement the `magnitude` method, using this signature:

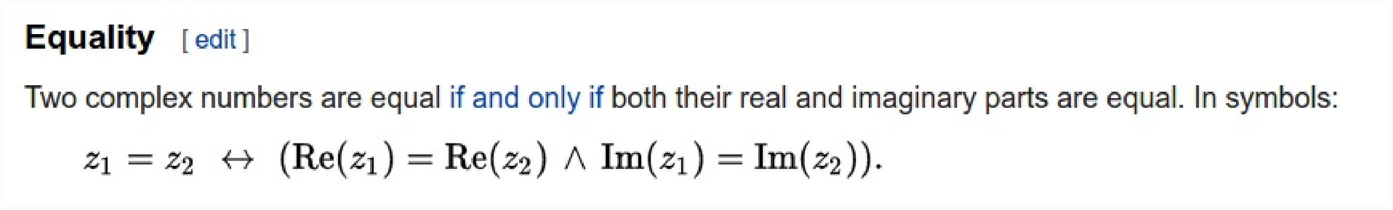
```
def magnitude(self):
        """ Returns a float which is the magnitude (that is, the absolute value) of the
complex number

            This method is something we introduce by ourselves, according to the definit
ion:
            https://en.wikipedia.org/wiki/Complex_number#Absolute_value_and_argument
        """
        raise Exception("TODO implement me!")
```

To test it, add this test case to `ComplexNumberTest` class (notice the `almost` in `assertAlmostEquals` !!!):

```
def test_magnitude(self):
        self.assertAlmostEqual(ComplexNumber(3.0,4.0).magnitude(),5, delta=0.001)
```

2.7. Complex numbers equality

Here we will try to give you a glimpse of some aspects 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.



Equality [edit]

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

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

- Implement equality for `ComplexNumber` more or less as it was done for `Fraction`

  Use this method signature:

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

and 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 hand
        """
        c = ComplexNumber(0,0)
        c.real = 1
        c.imaginary = 2
        self.assertEquals(c, ComplexNumber(1,2))
```

- 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/)

2.8. Complex numbers isclose

Complex numbers can be represented as vectors, so intuitively we can determine if a complex number is close to another by checking that the distance between its vector tip and the the other tip is less than a given delta. There are more precise ways to calculate it, but here we prefer keeping the example simple.

Given two complex numbers

$$z_1 = a + bi$$

and

$$z_2 = c + di$$

We can consider them as close if they satisfy this condition:

$$\sqrt{(a-c)^2 + (b-d)^2} < delta$$

- Implement the method, adding it to `ComplexNumber` class:

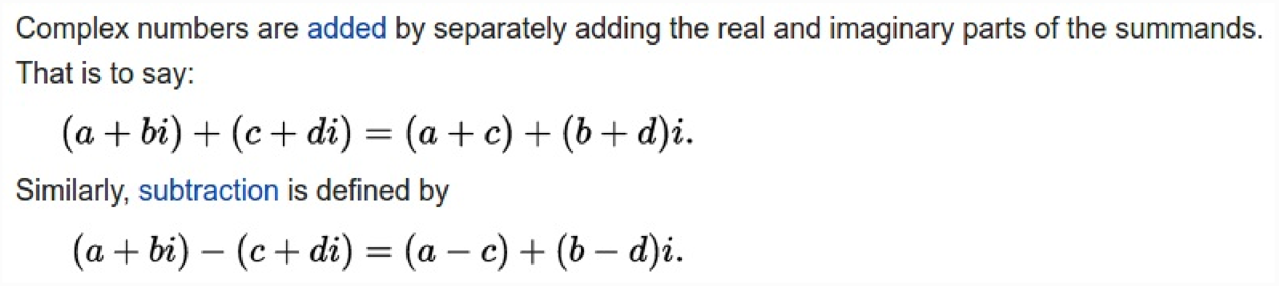
```
def isclose(self, c, delta):
        """ Returns True if the complex number is within a delta distance from complex n
umber c.
        """
        raise Exception("TODO Implement me!")
```

and add this test case to `ComplexNumberTest` class:

```
def test_isclose(self):
        """  Notice we use `assertTrue` because we expect `isclose` to return a `bool` v
alue, and
             we also test a case where we expect `False`
        """
        self.assertTrue(ComplexNumber(1.0,1.0).isclose(ComplexNumber(1.0,1.1), 0.2))

        self.assertFalse(ComplexNumber(1.0,1.0).isclose(ComplexNumber(10.0,10.0), 0.2))
```

> **REMEMBER: Equality with __eq__ and closeness functions like `isclose` are very different things. Equality should check if two objects have the same memory address or, alternatively, if they contain the same things, while closeness functions should check if two objects are similar. You should never use functions like `isclose` inside __eq__ methods, unless you really know what you're doing.**

2.9. 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

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

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

Use this definition:

```
def __add__(self, other):
    raise Exception("TODO implement me!")
```

And add this to the `ComplexNumberTest` class:

```
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));
```

2.10. 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 [56]:

```
    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.imaginary)

        # 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)
```

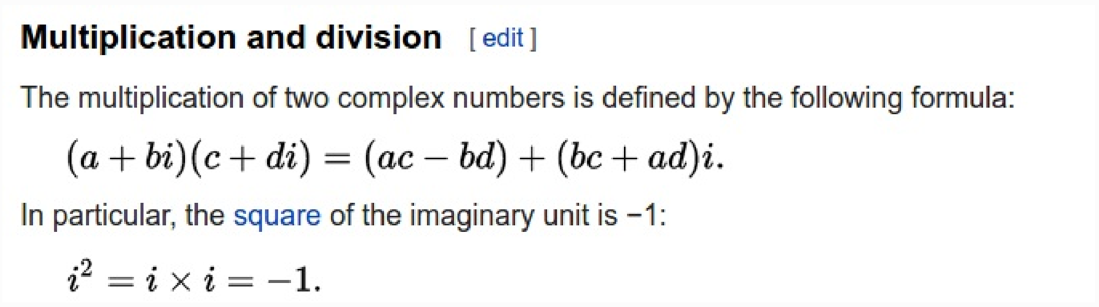
Answer: it won't, Python needs further instructions. Usually Python tries to see if the class of the object on left of the expression defines addition for operands *to the right* of it. In this case on the left we have a `float` number, and float numbers don't define any way to deal to the right with 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), 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
));
```

2.11. 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:

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

- Implement multiplication for `ComplexNumber`, taking inspiration from previous __add__ implementation
- Can you extend multiplication to work with scalars (both left and right) as well?

To implement `__mul__`, copy this definition into `ComplexNumber` class:

```
    def __mul__(self, other):
        raise Exception("TODO Implement me!")
```

and add test cases to `ComplexNumberTest` class:

```
    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));
```

3. Solutions

3.1. ComplexNumber Solution

In [57]:

```
import unittest
import math

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 phase(self):
        """ Returns a float which is the phase (that is, the vector angle) of the complex numb
er

            This method is something we introduce by ourselves, according to the definition:
            https://en.wikipedia.org/wiki/Complex_number#Absolute_value_and_argument
        """
        return math.atan2(self.imaginary, self.real)

    def log(self, base):
```

```
    def log(self, base):
        """ Returns another ComplexNumber which is the logarithm of this complex number


            This method is something we introduce by ourselves, according to the definition:
            (accomodated for generic base b)
            https://en.wikipedia.org/wiki/Complex_number#Natural_logarithm
        """
        return ComplexNumber(math.log(self.real) / math.log(base), self.phase() / math.log(bas
e))


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

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


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

    def isclose(self, c, delta):
        """ Returns True if the complex number is within a delta distance from complex number
c.
        """
        return math.sqrt((self.real-c.real)**2 + (self.imaginary-c.imaginary)**2) < delta

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

    """ 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_phase(self):
        """
            NOTE: we can't use assertEqual, as the result of phase() 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.assertAlmostEqua
l

            NOTE: assertEqual might still work on your machine but just DO NOT use it
            for float numbers!!!
        """
        self.assertAlmostEqual(ComplexNumber(0.0,1.0).phase(), math.pi / 2, 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")


    def test_log(self):
        c = ComplexNumber(1.0,1.0)
        l = c.log(math.e)
        self.assertAlmostEqual(l.real, 0.0, delta=0.001)
        self.assertAlmostEqual(l.imaginary, c.phase(), delta=0.001)


    def test_magnitude(self):
        self.assertAlmostEqual(ComplexNumber(3.0,4.0).magnitude(),5, delta=0.001)


    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_isclose(self):
        """  Notice we use `assertTrue` because we expect `isclose` to return a `bool` value,
and
             we also test a case where we expect `False`
        """
        self.assertTrue(ComplexNumber(1.0,1.0).isclose(ComplexNumber(1.0,1.1), 0.2))
        self.assertFalse(ComplexNumber(1.0,1.0).isclose(ComplexNumber(10.0,10.0), 0.2))

    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 [59]:

```
```