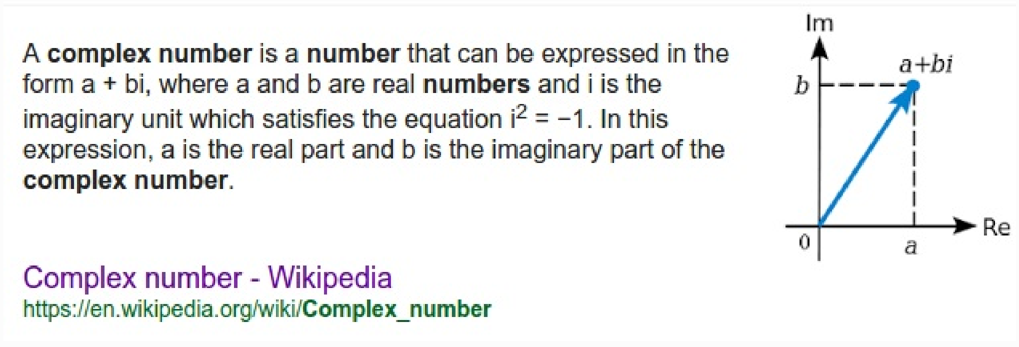
Chapter 2: Data Structures

Abstract Data Types (ADT) theory

- 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/ObjectOrientedProgramminginPythonDefinin (In particular, Fraction class (http://interactivepython.org/runestone/static/pythonds/Introduction/ObjectOrientedProgramminginPythonDefinin fraction-class), in this course we won't focus on inheritance)

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**

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 co
mplex number

            See definition: https://en.wikipedia.org/wiki/Complex_number#Absolute_va
lue_and_argument
        """
        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 nu
mber

            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
```

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.

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*.

ComplexNumber class

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.**

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 two 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 a 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 - Inside the previous `complex_number` function 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 0xb468688c>
```

What happend here?

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.

2) Python created a new special dictionary for the instance

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__`.**

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, I give one important word of wisdom, that can determine the result of your exam:

> **!!!!!! 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 p
assed by Python
self = 6        # self is only supposed to be a sort of dictionary which is p
assed by Python</p>
```

5) Python automatically returns from `__init__` the special dictionary `self`.

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

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

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

```
7) You can then statst using c as any variable
```

In [32]:

```
print c
```

```
<__main__.ComplexNumber instance at 0xb468688c>
```

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
```

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 co
mplex 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.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 co
mplex 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.atan2(self.imaginary, self.real)

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

            This method is something we introduce by ourselves, according to the def
inition:
            (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() / ma
th.log(base))
```

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)
```

%%HTML

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 [44]:

```
print logarithm
```

```
<__main__.ComplexNumber instance at 0xb4686d0c>
```

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

In [45]:

```
print logarithm.real
```

```
1.09861228867
```

In [46]:

```
print logarithm.imaginary
```

```
1.03037682652
```

A better print with str

As we said, printing is not so informative:

In [47]:

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

```
<__main__.ComplexNumber instance at 0xb46866cc>
```

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 [48]:

```python
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 co
mplex 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.atan2(self.imaginary, self.real)

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

            This method is something we introduce by ourselves, according to the def
inition:
            (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() / ma
th.log(base))

    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 [50]:

```
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 [51]:

```
print c
```

```
3.0 + 5.0i
```

In [52]:

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

```
1.58496250072 + 1.48651953787i
```

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 [53]:

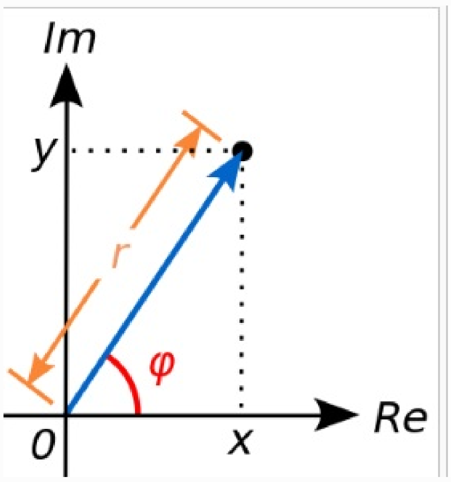
```
import unittest
import math

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 co
mplex 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.atan2(self.imaginary, self.real)

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

            This method is something we introduce by ourselves, according to the def
inition:
            (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() / ma
th.log(base))

    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.assert
AlmostEqual

            NOTE: assertEqual might still work on your machine but just DO NOT use i
t
            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):
```

```
    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)
```

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:

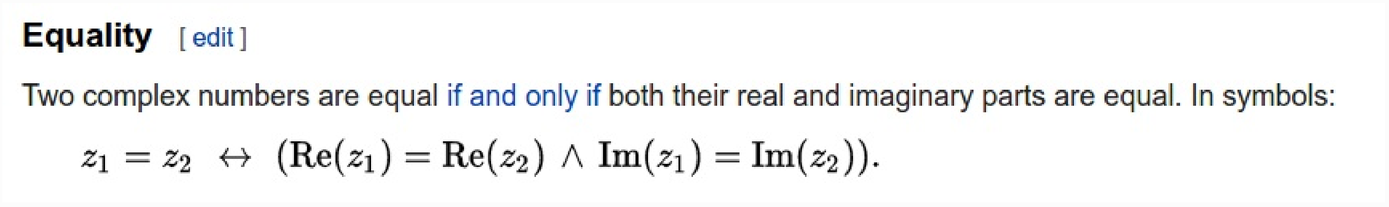
```
def magnitude(self):
        """ Returns a float which is the magnitude (that is, the absolute val
ue) of the complex number

            This method is something we introduce by ourselves, according to
the definition:
            https://en.wikipedia.org/wiki/Complex_number#Absolute_value_and_a
rgument
        """
        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)
```

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:

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

- 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:

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 han
d
        """
        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/)

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:

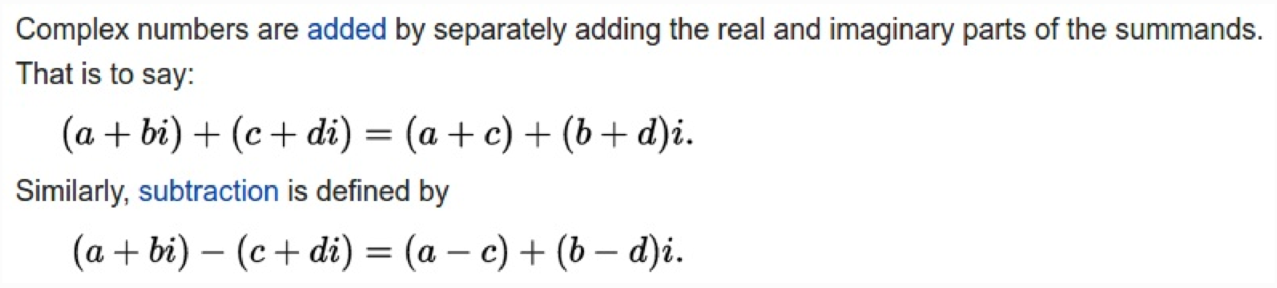
```python
def isclose(self, c, delta):
        """ Returns True if the complex number is within a delta distance from complex number c.
        """
        raise Exception("TODO Implement me!")
```

and add this test case to `ComplexNumberTest` class:

```python
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))
```

> **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.**

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), ComplexNum
ber(1,2));

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

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 [55]:

```
    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)
```

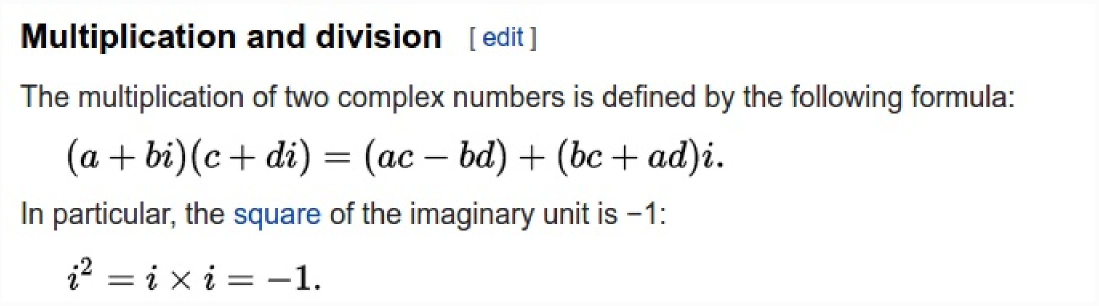
Answer: it won't, Python needs further instructions. Usually Python tries to see if the class of the object on left of the expression defines addition for operands *to the right* of it. In this case on the left we have a `float` number, and float numbers don't define any way to deal to the right with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

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

- 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), 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));
```

Stack

Stack theory

See theory here: [http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf (http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf)] (Slide 46)

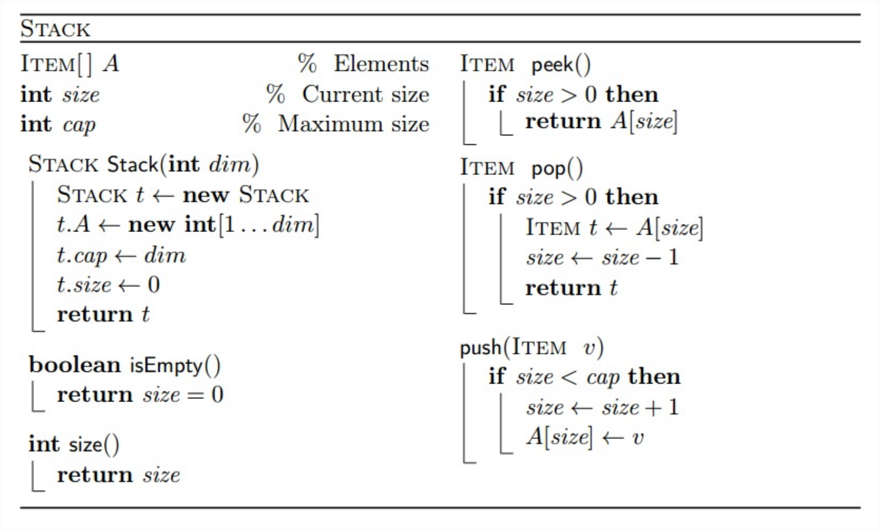
See stack definition on the book (http://interactivepython.org/runestone/static/pythonds/BasicDS/WhatisaStack.html)

and following sections :

- Stack Abstract Data Type (http://interactivepython.org/runestone/static/pythonds/BasicDS/TheStackAbstractDa
- Implementing a Stack in Python (http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaStack
- Simple Balanced Parenthesis (http://interactivepython.org/runestone/static/pythonds/BasicDS/SimpleBalancedPare
- Balanced Symbols - a General Case (http://interactivepython.org/runestone/static/pythonds/BasicDS/BalancedSymbols(AG

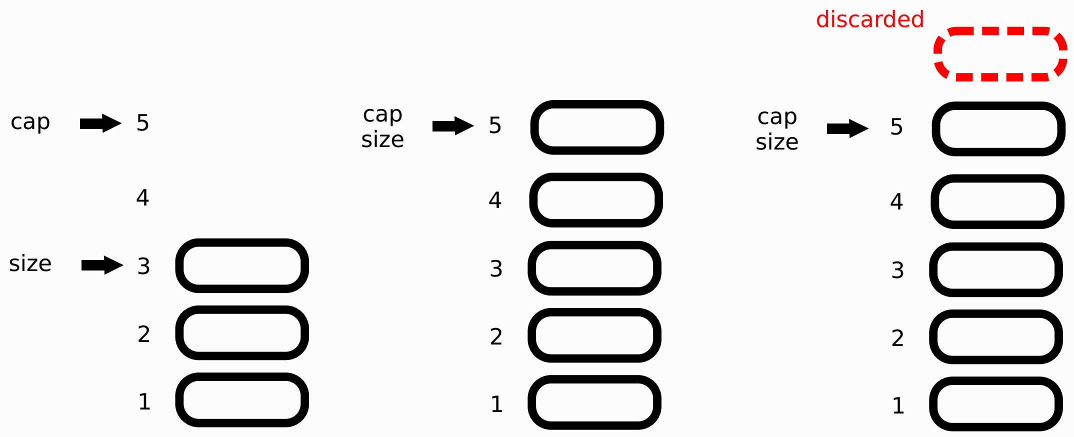
Stack exercises

On slide 46 of theory (http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf (Slide 46) there is the pseudo code for a version of stack we will call CappedStack:



```
STACK
ITEM[] A                 % Elements
int size                 % Current size
int cap                  % Maximum size

STACK Stack(int dim)
    STACK t ← new STACK
    t.A ← new int[1 ... dim]
    t.cap ← dim
    t.size ← 0
    return t

boolean isEmpty()
    return size = 0

int size()
    return size

ITEM peek()
    if size > 0 then
        return A[size]

ITEM pop()
    if size > 0 then
        ITEM t ← A[size]
        size ← size − 1
        return t

push(ITEM v)
    if size < cap then
        size ← size + 1
        A[size] ← v
```

A capped stack has a limit called *cap* over which elements are discarded:



- Copy the following skeleton and unit tests, and then implement the pseudo code
- Name internal variables that you don't want to expose to class users by prepending them with one underscore `'_'`, like `_elements` or `_cap`.
    - The underscore is just a convention, class users will still be able to get internal variables by accessing them with field accessors like `mystack._elements`.
    - If users manipulate private fields and complain something is not working, you can tell them it's their fault!

- This time, we will try to write a little more robust code. In general, when implementing pseudocode you might need to think more about boundary cases. In this case, we add the additional constraint that if you pass to the stack a negative or zero cap, your class initalization is expected to fail and raise an `AssertionError`. Such error can be raised by commands like `assert my_condition` where `my_condition` is `False`
- For easier inspection of the stack, implement also an `__str__` method so that calls to print show text like `CappedStack: cap=4 elements=['a', 'b']`

**IMPORTANT: The psudo code uses indexes to keep track the stack size. Since you are providing
an actual implementation in Python, you can exploit any Python feature you deem correct to implement
the data structure, and even depart a bit from the literal pseudo code. For example, internally
you could represent the data as a list, and use its own methods to grow it.**

**QUESTION: If we already have Python lists that can more or less do the job of the stack, why do we
need to wrap them inside a Stack? Can't we just give our users a Python list?**

QUESTION: When would you *not* use a Python list to hold the data in the stack?

Notice that:

- We tried to use more pythonic names (https://www.python.org/dev/peps/pep-0008/#id45) for methods, so for example isEmpty was renamed to is_empty
- In this case, when this stack reaches cap size, successive push requests silently exit without raising errors. Other implementations might raise an error and stop excecution when trying to push over on already filled stack.
- In this case, when this stack is required to pop or peek, if it is empty the functions will not return anything. During the Python translation, we might not return anything as well and relying on Python implicitly returning None.
- pop will both modify the stack *and* return a value

CappedStack Code Skeleton

In [57]:

```
import unittest

class CappedStack:

    def __init__(self, cap):
        """ Creates a CappedStack capped at cap. Cap must be > 0, otherwise an Asser
tionError is thrown
        """
        raise Exception("TODO Implement me!")

    def size(self):
        raise Exception("TODO Implement me!")

    def is_empty(self):
        raise Exception("TODO Implement me!")

    def pop(self):
        raise Exception("TODO Implement me!")
    def peek(self):
        raise Exception("TODO Implement me!")

    def push(self, item):

        raise Exception("TODO Implement me!")

    def cap(self):
        """ Returns the cap of the stack
        """
        raise Exception("TODO Implement me!")


class CappedStackTest(unittest.TestCase):
    """ Test cases for CappedStackTest

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

    def test_init_wrong_cap(self):
        """
            We use the special construct 'self.assertRaises(AssertionError)' to stat
e
            we are expecting the calls to CappedStack(0) and CappedStack(-1) to rais
e
            an AssertionError.
        """
        with self.assertRaises(AssertionError):
            CappedStack(0)
        with self.assertRaises(AssertionError):
            CappedStack(-1)


    def test_cap(self):
        self.assertEqual(CappedStack(1).cap(), 1)
        self.assertEqual(CappedStack(2).cap(), 2)


    def test_size(self):
        s = CappedStack(5)
        self.assertEqual(s.size(), 0)
        s.push("a")
        self.assertEqual(s.size(), 1)
        s.pop()
        self.assertEqual(s.size(), 0)

    def test_is_empty(self):
        s = CappedStack(5)
        self.assertTrue(s.is_empty())
        s.push("a")
        self.assertFalse(s.is_empty())


    def test_pop(self):
        s = CappedStack(5)
        self.assertEqual(s.pop(), None)
        s.push("a")
        self.assertEqual(s.pop(), "a")
        self.assertEqual(s.pop(), None)

    def test_peek(self):
        s = CappedStack(5)
        s = CappedStack(5)
        self.assertEqual(s.peek(), None)
        s.push("a")
        self.assertEqual(s.peek(), "a")
        self.assertEqual(s.peek(), "a")  # testing peek is not changing the stack
        self.assertEqual(s.size(), 1)

    def test_push(self):
        s = CappedStack(2)
        self.assertEqual(s.size(), 0)
        s.push("a")
        self.assertEqual(s.size(), 1)
        s.push("b")
        self.assertEqual(s.size(), 2)
        self.assertEqual(s.peek(), "b")
        s.push("c")  # capped, pushing should do nothing now!
        self.assertEqual(s.size(), 2)
        self.assertEqual(s.peek(), "b")

    def test_str(self):
        s = CappedStack(4)
        s.push("a")
        s.push("b")
        print s
```

UnorderedList

UnorderedList Theory

An UnorderedList for us is a linked list starting with a pointer called *head* that points to the first Node (if the list is empty the pointer points to None). Think of the list as a chain where each Node can contain some data retriavable with Node.get data() method and you can access one Node at a time by calling the method Node.get_next() on each node.

- See theory slides (http://disi.unitn.it/~montreso/sp/slides/04-strutture.pdf) from slide 25 (Monodirectional list)
- See UnorderedList Abstract Data Type (http://interactivepython.org/runestone/static/pythonds/BasicDS/TheUnorderedListAb on the book
- See Implementing UnorderedListLinkedLists (http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementinganUnor on the book

UnorderedList Exercises

1) Understand the problem

Copy the following skeleton and unit tests, and then implement the missing methods, in the order they are presented in the skeleton.

- This time there is not much pseudo or Python code to find around, you should rely solely on theory from the slides and book, method definitions and your intuition
- Pay close attention to the comments below each method definition, especially for boundary cases

COMMANDMENT: You shall also draw lists on paper, helps a lot avoiding mistakes

WARNING: Do *not* use a Python list
    to hold data inside the data structure. Differently from the `CappedStack` exercise, here you
    can only use `Node` class. Each `Node` in the `_data` field can hold only one element which is provided by the user of the class, and we
    don't care about the type of the value the user gives us (so it can be an `int`,
    a `float`, a `string`, or even a Python list !)

WARNING: NEVER EVER WRITE code like this: `self = something`

Instead, assign values to *fields*
     of `self`, like `self._head = something`

WARNING: if there isn't the word "*Return*" in a method comment, then most probably that
    method does not return anything!

WARNING: Methods of the class `UnorderedList` are supposed to *never* return instances of `Node`. If
 you see them returned in the tests, then you are making some mistake. Users of `UnorderedList` are
    should only be able to get access to items inside the `Node` `data` fields.

Notice that there are a few differences with the book:

- We don't assume the list has all different values
- We used more pythonic names (https://www.python.org/dev/peps/pep-0008/#id45) for properties and methods, so for example private attribute `Node.data` was renamed to `Node._data` and accessor method `Node.getData()` was renamed to `Node.get_data()`. There are nicer ways to handle these kind of getters/setters pairs called 'properties' but we won't address them here.
- In boundary cases like removing a non-existing element we prefer to raise an exception with the command

```
raise Exception("Some error occurred!")
```

In general, this is the behaviour you also find in regular Python lists.

2) Save a copy of your work

You already wrote a lot of code, and you don't want to lose it, right? Since we are going to make many modifications, when you reach a point when the code does something useful, it is good practice to save a copy of what you have done somewhere, so if you later screw up something, you can always restore the copy of the work

- Add also somewhere in the copy (in a comment at the top of the `.py` file or in a separate `README.txt` file) the version (like `1.0`), the date, and a description of the main features (for example "Simple linked list, not particularly performant").
- Backing up the work is a form of the so-called *versioning* : there are much better ways to do it (like using git (https://git-scm.com)) but we don't address them here.

3) Faster size()

Implement a `size()` method that works in `O(1)`. To make this work without going through the whole list each time, we will need a new `_size` field that keeps track of the size. When the list is mutated with methods like `add`, `append`, etc you will also need to update the `_size` field accordingly. Proceed like this:

3.1) add a new field `_size` in the class constructor and initialize it to zero

3.2) modify the `size()` method to just return the `_size` field.

3.3) The data structure starts to be complex, and we need better testing. If you look at the tests, very often there are lines of code like `self.assertEquals(ul.to_python(), ['a', 'b'])` in the `test_add` method:

```
def test_add(self):
        ul = UnorderedList()
        self.myAssert(ul, [])
        ul.add('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.add('a')
        self.assertEquals(ul.to_python(), ['a', 'b'])
```

Last line checks our unordered list `ul` contains a sequence of linked nodes that once transformed to a python list actually equals `['a', 'b']`. Since in the new implementation we are going to mutate `_size` field a lot, it could be smart to also check that `ul.size()` equals `len(["a", "b"])`. Repeating this check in every test method could be quite verbose. Instead, we can do a smarter thing, and develop in the `UnorderedListTest` class a new assertion method on our own:

3.3.1) Add this method to `UnorderedListTest` class:

```
def myAssert(self, unordered_list, python_list):
        """ Checks provided unordered_list can be represented as the given py
thon_list. Since v2.
        """
        self.assertEquals(unordered_list.to_python(), python_list)
        # check this new invariant about the size
        self.assertEquals(unordered_list.size(), len(python_list))
```

> **WARNING: method `myAssert` must *not* being with `test_`, otherwise `unittest`
> will run it as a test!**

3.3.2) Now, how to use this powerful new `myAssert` method? In the test class, just replace every occurence of

```
self.assertEquals(ul.to_python(), ['a', 'b'])
```

into calls like this:

```
self.myAssert(ul, ['a', 'b'])
```

WARNING: Notice the `.to_python()` after `ul` is gone.

3.4) Actually update `_size` in the various methods where data is mutated, like `add`, `insert`, etc.

3.5) Run the tests and hope for the best ;-)

4) Faster append()

We are now better equipped to make further improvements. Once you're done implementing the above and made sure everything works, you can implement an `append` method that works in $O(1)$ by adding an additional pointer in the data structure that always point at the last node. To further exploit the pointer, you can also add a fast `last(self)` method that returns the last value in the list. Proceed like this:

4.1) Save a copy of your work somewhere, putting version (2.0), date, and comments on the improvements.

4.2) Add an additional pointer called `_last` in the constructor.

4.3) Copy this method into the class. Just copy it, don't implement it for now.

```
    def last(self):
            """ Returns the last element in the list, in O(1).

                If list is empty, raises an Exception. Since v2.
            """
            raise Exception("TODO implement me!")
```

4.4) Let's do some so-called *test driven development*, that is, first we write the tests, then we write the implementation.

> **WARNING: During the exam you *will* be asked to write tests, so don't skip writing them now !!**

4.4.1) Implement a test for `last()` method, by adding this to `UnorderedListTest` class:

```
    def test_last(self):
            raise Exception("TODO IMPLEMENT ME !")
```

In the method, create a list and add elements using only calls to `add` method and checks using the `myAssert` method. When done, ask your instructor if the test is correct (or look at the proposed solution at the bottom of the worksheet), it is important you get it right otherwise you won't be able to properly test your code.

4.4.2) You already have a test for the `append()` method, but, how can you be sure the `_last` pointer is updated correctly throughout the code? When you implemented the fast `size()` method you wrote some invariant in the myAssert method. We can do the same this time, too. Find the invariant and add the corresponding check to the `myAssert` method. When done, ask your instructor if the invariant is correct (or look at the proposed solution at the bottom of the worksheet): it is important you get it right otherwise you won't be able to properly test your code.

4.5) Update the methods that mutate the data structure (`add`, `insert`, `remove` ...) so they keep `_last` pointed to last element. If the list is empty, `_last` will point to None. Taking particular care of corner cases such as empty list and one element list.

4.6) Cross your fingers and run the tests!

5) Go bidirectional

Our list so far has links that allow us to traverse it fast in one direction. But what if we want fast traversal in the reverse direction, from last to first element? What if we want a `pop()` that works in $O(1)$? To speed up these operations we could add backward links to each `Node`. Proceed in the following way:

5.1) Save your work

Once you're done with previous points, save the version you have somewhere adding comments about the improvements done so far, the version number (like 2.0) and the date. Then start working on a new copy.

5.2) Node backlinks

In `Node` class, add backlinks by adding the attribute `_prev` and methods `get_prev(self)` and `set_prev(self, pointer)`.

5.3) Better `__str__`

Improve `__str__` method so it shows presence or absence of links, along with the size of the list.
`next` pointers presence must be represented with `>` character , absence with `*` character. They must be put after the item representation.
`prev` pointers presence must be represented with `<` character , absence with `*` character. They must be put befor the item representation.

For example, for the list ['a','b','c'], you would have the following representation:

```
UnorderedList(size=3):*a><b><c*
```

As a special case for empty list you should print the following:

```
UnorderedList(size=0):[]
```

Other examples of proper lists, with 3, 2, and 1 element can be:

```
UnorderedList(size=3):*a><b><c*
UnorderedList(size=2):*a><b*
UnorderedList(size=1):*a*
```

This new `__str__` method should help you to spot broken lists like the following, were some pointers are not correct:

```
Broken list, all prev pointers are missing:
UnorderedList(size=3):*a>*b>*c*

Broken list, size = 3 but shows only one element with next pointer set to Non
e:
UnorderedList(size=3):*a*

Broken list, first backward pointer points to something other than None
UnorderedList(size=3):<a><b><c*
```

5.4) Modify add()

Update the UnorderedList add method to take into account you now have backlinks. Take particular care for the boundary cases when the list is empty, has one element, or for nodes at the head and at the tail of the list.

5.5) Add to_python_reversed()

Implement to_python_reversed method with a linear scan by using the newly added backlinks:

```
def to_python_reversed(self):
        """ Returns a regular Python list with the elements in reverse order,
            from last to first. Since v3. """
        raise Exception("TODO implement me")
```

Add also this test, and make sure it pass:

```
def test_to_python_reversed(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('b')
        ul.add('a')
        pr = ul.to_python()
        pr.reverse()  # we are reversing pr with Python's 'reverse()' method
        self.assertEquals(pr, ul.to_python_reversed())
```

5.6) Add invariant

By using the method to_python_reversed(), add a new invariant to the myAssert method. If implemented correctly, this will surely spot a lot of possible errors in the code.

5.7) Modify other methods

Modify all other methods that mutate the data structure (insert, remove, etc) so that they update the backward links properly.

5.8) Run the tests

If you wrote meaningful tests and all pass, congrats!

UnorderedList Code Skeleton

In [62]:

```
import unittest

class Node:
    """ A Node of an UnorderedList. Holds data provided by the user. """

    def __init__(self,initdata):
        self._data = initdata
        self._next = None

    def get_data(self):
        return self._data

    def get_next(self):
        return self._next

    def set_data(self,newdata):
        self._data = newdata

    def set_next(self,newnext):
        self._next = newnext


class UnorderedList:
    """
        This class is slightly different from the one present in the book:
            - has more pythonic names
            - tries to mimic more closely the behaviour of default Python list, rais
ing exceptions on
              boundary conditions like removing non exisiting elements.
    """

    def __init__(self):
        self._head = None

    def to_python(self):
        """ Returns this UnorderedList as a regular Python list. This method is very
 handy for testing.
        """
        python_list = []
        current = self._head

        while (current != None):
            python_list.append(current.get_data())
            current = current.get_next()
        return python_list

    def __str__(self):
        """ For potentially complex data structures like this one, having a __str__ method is essential to
            quickly inspect the data by printing it.
        """
        current = self._head
        strings = []

        while (current != None):
            strings.append(str(current.get_data()))
            current = current.get_next()

        return "UnorderedList: " + ",".join(strings)


    def is_empty(self):
        """ Returns True if the list has no nodes, True otherwise """
        raise Exception("TODO implement me!")


    def add(self,item):
        """ Adds item at the beginning of the list """
        raise Exception("TODO implement me!")


    def size(self):
        """ Returns the size of the list """
        raise Exception("TODO implement me!")


    def search(self,item):
        """ Returns True if item is present in list, False otherwise
        """
        raise Exception("TODO implement me!")

    def remove(self, item):
        """ Removes first occurrence of item from the list

            If item is not found, raises an Exception.
        """
        raise Exception("TODO implement me!")

    def append(self, e):
        """ Appends element e to the end of the list.

            For this exercise you can write the O(n) version
        """

        raise Exception("TODO implement me!")

    def insert(self, i, e):
        """ Insert an item at a given position.

            The first argument is the index of the element before which to insert, so list.insert(0, e)
            inserts at the front of the list, and list.insert(list.size(), e) is equivalent to list.append(e).
            When i > list.size(), raises an Exception (default Python list appends instead to the end :-/ )

        """
        raise Exception("TODO implement me!")

    def index(self, e):
        """ Return the index in the list of the first item whose value is x.
            If there is no such item, an exception is raised.
        """

        raise Exception("TODO implement me!")


    def pop(self):
        """ Remove the last item of the list, and return it.

            If the list is empty, an exception is raised.
        """
        raise Exception("TODO implement me!")

class UnorderedListTest(unittest.TestCase):
    """ Test cases for UnorderedList

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

    def test_init(self):
        ul = UnorderedList()

    def test_str(self):
        ul = UnorderedList()
        self.assertTrue('UnorderedList' in str(ul))
        ul.add('z')
        self.assertTrue('z' in str(ul))
        ul.add('w')
        self.assertTrue('z' in str(ul))
        self.assertTrue('w' in str(ul))


    def test_is_empty(self):
        ul = UnorderedList()
        self.assertTrue(ul.is_empty())
        ul.add('a')
        self.assertFalse(ul.is_empty())

    def test_add(self):
        """ Remember 'add' adds stuff at the beginning of the list ! """

        ul = UnorderedList()
        self.assertEquals(ul.to_python(), [])
        ul.add('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.add('a')
        self.assertEquals(ul.to_python(), ['a', 'b'])

    def test_size(self):
        ul = UnorderedList()
        self.assertEquals(ul.size(), 0)
        ul.add("a")
        self.assertEquals(ul.size(), 1)
        ul.add("b")
        self.assertEquals(ul.size(), 2)

    def test_search(self):
        ul = UnorderedList()
        self.assertFalse(ul.search("a"))
        ul.add("a")
        self.assertTrue(ul.search("a"))
        self.assertFalse(ul.search("b"))
        ul.add("b")
        self.assertTrue(ul.search("a"))
        self.assertTrue(ul.search("b"))

    def test_remove_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.remove('a')

    def test_remove_one_element(self):
        ul = UnorderedList()
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('b')
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])

    def test_remove_two_element(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['a'])
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])


    def test_remove_first_occurrence(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('b')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.remove('b')
        self.assertEquals(ul.to_python(), [])

    def test_append(self):
        ul = UnorderedList()
        ul.append('a')
        self.assertEquals(ul.to_python(),['a'])
        ul.append('b')
        self.assertEquals(ul.to_python(),['a', 'b'])

    def test_insert_empty_list_zero(self):
        ul = UnorderedList()
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a'])

    def test_insert_empty_list_out_of_bounds(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.insert(1, 'a')
        with self.assertRaises(Exception):
            ul.insert(-1, 'a')

    def test_insert_one_element_list_before(self):
        ul = UnorderedList()
        ul.add('b')
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a','b'])


    def test_insert_one_element_list_after(self):
        ul = UnorderedList()
        ul.add('a')
        ul.insert(1, 'b')
        self.assertEquals(ul.to_python(), ['a','b'])

    def test_insert_two_element_list_insert_before(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('b')
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a','b','c'])

    def test_insert_two_element_list_insert_middle(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('a')
        ul.insert(1, 'b')
        self.assertEquals(ul.to_python(), ['a','b', 'c'])

    def test_insert_two_element_list_insert_after(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        ul.insert(2, 'c')
        self.assertEquals(ul.to_python(), ['a','b', 'c'])


    def test_index_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.index('a')

    def test_index(self):
        ul = UnorderedList()
        ul.add('b')
        self.assertEquals(ul.index('b'),  0)
        with self.assertRaises(Exception):
            ul.index('a')
        ul.add('a')
        self.assertEquals(ul.index('a'),  0)
        self.assertEquals(ul.index('b'),  1)

    def test_pop_empty(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.pop()

    def test_pop_one(self):
        ul = UnorderedList()
        ul.add('a')
        x = ul.pop()
        self.assertEquals('a', x)

    def test_pop_two(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        x = ul.pop()
        self.assertEquals('b', x)
        self.assertEquals('b', x)
        self.assertEquals(ul.to_python(), ['a'])
        y = ul.pop()
        self.assertEquals('a', y)
        self.assertEquals(ul.to_python(), [])
```

Solutions

ComplexNumber Solution

In [63]:

```
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 co
mplex 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.atan2(self.imaginary, self.real)

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


            This method is something we introduce by ourselves, according to the def
inition:
            (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() / ma
th.log(base))


    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 __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 compl
ex 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.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):

    """ 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.assert
AlmostEqual

            NOTE: assertEqual might still work on your machine but just DO NOT use i
t
            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.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 `boo
l` 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 [64]:

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

```
.................
----------------------------------------------------------------------
Ran 17 tests in 0.012s

OK
```

CappedStack Solution

In [65]:

```
import unittest

class CappedStack:

    def __init__(self, cap):
        """ Creates a CappedStack capped at cap.

            Cap must be > 0, otherwise an AssertionError is thrown
        """
        assert cap > 0
        assert cap > 0
        # notice we assign to variables with underscore to respect Python convention
s
        self._cap = cap
        # notice with use _elements instead of the A in the pseudocode, because it i
s
        # clearer, starts with underscore, and capital letters are usual reserved
        # for classes or constants
        self._elements = []

    def size(self):
        return len(self._elements)

    def is_empty(self):
        return len(self._elements) == 0

    def pop(self):
        if (len(self._elements) > 0):
            return self._elements.pop()
        # else: implicitly, Python will return None

    def peek(self):
        if (len(self._elements) > 0):
            return self._elements[-1]
        # else: implicitly, Python will return None

    def push(self, item):
        if (len(self._elements) < self._cap):
            self._elements.append(item)
        # else fail silently

    def cap(self):
        """ Returns the cap of the stack
        """
        return self._cap

    def __str__(self):
        return "CappedStack: cap=" + str(self._cap) + " elements=" + str(self._eleme
nts)


class CappedStackTest(unittest.TestCase):

    """ Test cases for CappedStackTest

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

    def test_init_wrong_cap(self):
        """
            We use the special construct 'self.assertRaises(AssertionError)' to stat
e
            we are expecting the calls to CappedStack(0) and CappedStack(-1) to rais
e
            an AssertionError.
        """
        with self.assertRaises(AssertionError):
            CappedStack(0)
        with self.assertRaises(AssertionError):
            CappedStack(-1)


    def test_cap(self):
```

```
    def test_cap(self):
        self.assertEqual(CappedStack(1).cap(), 1)
        self.assertEqual(CappedStack(2).cap(), 2)


    def test_size(self):
        s = CappedStack(5)
        self.assertEqual(s.size(), 0)
        s.push("a")
        self.assertEqual(s.size(), 1)
        s.pop()
        self.assertEqual(s.size(), 0)

    def test_is_empty(self):
        s = CappedStack(5)
        self.assertTrue(s.is_empty())
        s.push("a")
        self.assertFalse(s.is_empty())


    def test_pop(self):
        s = CappedStack(5)
        self.assertEqual(s.pop(), None)
        s.push("a")
        self.assertEqual(s.pop(), "a")
        self.assertEqual(s.pop(), None)

    def test_peek(self):
        s = CappedStack(5)
        self.assertEqual(s.peek(), None)
        s.push("a")
        self.assertEqual(s.peek(), "a")
        self.assertEqual(s.peek(), "a")  # testing peek is not changing the stack
        self.assertEqual(s.size(), 1)

    def test_push(self):
        s = CappedStack(2)
        self.assertEqual(s.size(), 0)
        s.push("a")
        self.assertEqual(s.size(), 1)
        s.push("b")
        self.assertEqual(s.size(), 2)
        self.assertEqual(s.peek(), "b")
        s.push("c")  # capped, pushing should do nothing now!
        self.assertEqual(s.size(), 2)
        self.assertEqual(s.peek(), "b")

    def test_str(self):
        s = CappedStack(4)
        s.push("a")
        s.push("b")
        print s
```

In [66]:

```
algolab.run(CappedStackTest)
```

```
........

CappedStack: cap=4 elements=['a', 'b']

----------------------------------------------------------------------
Ran 8 tests in 0.017s

OK
```

UnorderedList v1 Solution

In [67]:

```
import unittest

class Node:
    """ A Node of an UnorderedList. Holds data provided by the user. """

    def __init__(self,initdata):
        self._data = initdata
        self._next = None

    def get_data(self):
        return self._data

    def get_next(self):
        return self._next

    def set_data(self,newdata):
        self._data = newdata

    def set_next(self,newnext):
        self._next = newnext


class UnorderedList:
    """
        UnorderedList v1

        This class is slightly different from the one present in the book:
            - has more pythonic names
            - tries to mimic more closely the behaviour of default Python list, rais
ing exceptions on
              boundary conditions like removing non exisiting elements.
    """

    def __init__(self):
        self._head = None

    def to_python(self):
        """ Returns this UnorderedList as a regular Python list. This method is very
 handy for testing.
        """
        python_list = []
        current = self._head

        while (current != None):
            python_list.append(current.get_data())
            current = current.get_next()
        return python_list

    def __str__(self):
        """ For potentially complex data structures like this one, having a __str__
method is essential to
            quickly inspect the data by printing it.
        """
        current = self._head
        strings = []

        while (current != None):
            strings.append(str(current.get_data()))
            current = current.get_next()

        return "UnorderedList: " + ", ".join(strings)
        return "UnorderedList: " + ", ".join(strings)


    def is_empty(self):
        return self._head == None

    def add(self,item):
        """ Adds item at the beginning of the list """
        new_head = Node(item)
        new_head.set_next(self._head)
        self._head = new_head

    def size(self):
        """ Returns the size of the list """
        current = self._head
        count = 0

        while (current != None):
            current = current.get_next()
            count += 1

        return count

    def search(self,item):
        """ Returns True if item is present in list, False otherwise
        """
        current = self._head

        while (current != None):
            if (current.get_data() == item):
                return True
            else:
                current = current.get_next()

        return False

    def remove(self, item):
        """ Removes first occurrence of item from the list

            If item is not found, raises an Exception.
        """
        current = self._head
        prev = None

        while (current != None):
            if (current.get_data() == item):
                if prev == None:  # we need to remove the head
                    self._head = current.get_next()
                else:
                    prev.set_next(current.get_next())
                    current = current.get_next()
                return  # Found, exits the function
            else:
                prev = current
                current = current.get_next()

        raise Exception("Tried to remove a non existing item! Item was: " + str(item
))

    def append(self, e):
        """ Appends element e to the end of the list.

            For this exercise you can write the O(n) version
        """

        if self._head == None:
```

```
        if self._head == None:
            self.add(e)
        else:
            current = self._head
            while (current.get_next() != None):
                current = current.get_next()
            current.set_next(Node(e))

    def insert(self, i, e):
        """ Insert an item at a given position.

            The first argument is the index of the element before which to insert, s
o list.insert(0, e)
            inserts at the front of the list, and list.insert(list.size(), e) is equ
ivalent to list.append(e).
            When i > list.size(), raises an Exception (default Python list appends i
nstead to the end :-/ )

        """
        if (i < 0):
            raise Exception("Tried to insert at a negative index! Index was:" + str(
i))

        count = 0
        current = self._head
        prev = None

        while (count < i and current != None):
            prev = current
            current = current.get_next()
            count += 1

        if (current == None):
            if (count == i):
                self.append(e)
            else:
                raise Exception("Tried to insert outside the list ! "
                                + "List size=" + str(count) + "  insert position=" +
 str(i))
        else:
            #0 1
            #  i
            if (prev == None):
                self.add(e)
            else:
                new_node = Node(e)
                prev.set_next(new_node)
                new_node.set_next(current)

    def index(self, e):
        """ Return the index in the list of the first item whose value is x.

            If item is not found, raises an Exception.
        """

        current = self._head
        count = 0

        while (current != None):
            if (current.get_data() == e):
                return count
            else:
                current = current.get_next()
                count += 1

        raise Exception("Couldn't find element " + str(e) )
        raise Exception("Couldn't find element " + str(e) )



    def pop(self):
        """ Remove the last item of the list, and return it.

            If the list is empty, an exception is raised.
        """
        if (self._head == None):
            raise Exception("Tried to pop an empty list!")
        else:

            current = self._head

            if (current.get_next() == None): # one element list
                last_item = self._head.get_data()
                self._head = None
            else:    # we have more than one element
                prev = None
                while current.get_next() != None:  # current will reach last element
                    prev = current
                    current = current.get_next()

                last_item = current.get_data()
                prev.set_next(None)

            return last_item

class UnorderedListTest(unittest.TestCase):
    """ Test cases for UnorderedList

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

    def test_init(self):
        ul = UnorderedList()

    def test_str(self):
        ul = UnorderedList()
        self.assertTrue('UnorderedList' in str(ul))
        ul.add('z')
        self.assertTrue('z' in str(ul))
        ul.add('w')
        self.assertTrue('z' in str(ul))
        self.assertTrue('w' in str(ul))


    def test_is_empty(self):
        ul = UnorderedList()
        self.assertTrue(ul.is_empty())
        ul.add('a')
        self.assertFalse(ul.is_empty())

    def test_add(self):
        """ Remember 'add' adds stuff at the beginning of the list ! """

        ul = UnorderedList()
        self.assertEquals(ul.to_python(), [])
        ul.add('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.add('a')
        self.assertEquals(ul.to_python(), ['a', 'b'])
        self.assertEquals(ul.to_python(), ['a', 'b'])

    def test_size(self):
        ul = UnorderedList()
        self.assertEquals(ul.size(), 0)
        ul.add("a")
        self.assertEquals(ul.size(), 1)
        ul.add("b")
        self.assertEquals(ul.size(), 2)

    def test_search(self):
        ul = UnorderedList()
        self.assertFalse(ul.search("a"))
        ul.add("a")
        self.assertTrue(ul.search("a"))
        self.assertFalse(ul.search("b"))
        ul.add("b")
        self.assertTrue(ul.search("a"))
        self.assertTrue(ul.search("b"))

    def test_remove_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.remove('a')

    def test_remove_one_element(self):
        ul = UnorderedList()
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('b')
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])

    def test_remove_two_element(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['a'])
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])


    def test_remove_first_occurrence(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('b')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.remove('b')
        self.assertEquals(ul.to_python(), [])


    def test_append(self):
        ul = UnorderedList()
        ul.append('a')
        self.assertEquals(ul.to_python(),['a'])
        ul.append('b')
        self.assertEquals(ul.to_python(),['a', 'b'])

    def test_insert_empty_list_zero(self):
        ul = UnorderedList()
```

```python
        ul = UnorderedList()
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a'])

    def test_insert_empty_list_out_of_bounds(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.insert(1, 'a')
        with self.assertRaises(Exception):
            ul.insert(-1, 'a')

    def test_insert_one_element_list_before(self):
        ul = UnorderedList()
        ul.add('b')
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a','b'])


    def test_insert_one_element_list_after(self):
        ul = UnorderedList()
        ul.add('a')
        ul.insert(1, 'b')
        self.assertEquals(ul.to_python(), ['a','b'])

    def test_insert_two_element_list_insert_before(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('b')
        ul.insert(0, 'a')
        self.assertEquals(ul.to_python(), ['a','b','c'])

    def test_insert_two_element_list_insert_middle(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('a')
        ul.insert(1, 'b')
        self.assertEquals(ul.to_python(), ['a','b', 'c'])

    def test_insert_two_element_list_insert_after(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        ul.insert(2, 'c')
        self.assertEquals(ul.to_python(), ['a','b', 'c'])


    def test_index_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.index('a')

    def test_index(self):
        ul = UnorderedList()
        ul.add('b')
        self.assertEquals(ul.index('b'),  0)
        with self.assertRaises(Exception):
            ul.index('a')
        ul.add('a')
        self.assertEquals(ul.index('a'),  0)
        self.assertEquals(ul.index('b'),  1)

    def test_pop_empty(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.pop()
    def test_pop_one(self):
        ul = UnorderedList()
        ul.add('a')
        x = ul.pop()
        self.assertEquals('a', x)

    def test_pop_two(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        x = ul.pop()
        self.assertEquals('b', x)
        self.assertEquals(ul.to_python(), ['a'])
        y = ul.pop()
        self.assertEquals('a', y)
        self.assertEquals(ul.to_python(), [])
```

In [68]:

```python
algolab.run(UnorderedListTest)
```

```
.......................
----------------------------------------------------------------------
Ran 23 tests in 0.040s

OK
```

UnorderedList v2 Solution

In [69]:

```
import unittest

class Node:
    """ A Node of an UnorderedList. Holds data provided by the user.

        Node v2 remains the same as Node v1
    """
    def __init__(self,initdata):
        self._data = initdata
        self._next = None

    def get_data(self):
        return self._data

    def get_next(self):
        return self._next

    def set_data(self,newdata):
        self._data = newdata

    def set_next(self,newnext):
        self._next = newnext


class UnorderedList:
    """
        a linked list implementation, v2.

        Improvements upon UnorderedList v1:

        * calculates size() in O(1)
        * calculates append() in O(1)
        * adds last() method to retrieve last element in O(1)

        This class is slightly different from the one present in the book:
            - has more pythonic names
            - tries to mimic more closely the behaviour of default Python list, rais
ing exceptions on
              boundary conditions like removing non exisiting elements.
    """

    def __init__(self):
        self._head = None
        self._size = 0  # NEW attribute '_size'
        self._last = None # NEW attribute '_last'
    def to_python(self):
        """ Returns this UnorderedList as a regular Python list. This method is very
 handy for testing.
        """

        python_list = []
        current = self._head

        while (current != None):
            python_list.append(current.get_data())
            current = current.get_next()
        return python_list


    def __str__(self):
        """ For potentially complex data structures like this one, having a __str__
method is essential to
            quickly inspect the data by printing it.
        """
        current = self._head
        strings = []

        while (current != None):
            strings.append(str(current.get_data()))
            current = current.get_next()

        return "UnorderedList: " + ",".join(strings)


    def is_empty(self):
        return self._head == None

    def add(self,item):
        """ Adds item at the beginning of the list """
        new_head = Node(item)
        new_head.set_next(self._head)
        if self._head == None: # NEW
            self._last = new_head # NEW
        self._head = new_head
        self._size += 1 # NEW, we just return the field value. This is fast!


    def size(self):
        """ Returns the size of the list in O(1) """
        return self._size  # NEW, we just return the field value. This is fast!

    def search(self,item):
        """ Returns True if item is present in list, False otherwise
        """
        current = self._head

        while (current != None):
            if (current.get_data() == item):
                return True
            else:
                current = current.get_next()

        return False

    def remove(self, item):
        """ Removes first occurrence of item from the list

            If item is not found, raises an Exception.
        """
        current = self._head
        current = self._head
        prev = None

        while (current != None):

            if (current.get_data() == item):
                if (self._last == current):  # NEW
                    self._last = prev        # NEW

                if prev == None:  # we need to remove the head
                    self._head = current.get_next()
                else:
                    prev.set_next(current.get_next())
                    current = current.get_next()
                self._size -= 1  # NEW, need to update _size
                return  # Found, exits the function
            else:
                prev = current
                current = current.get_next()

        raise Exception("Tried to remove a non existing item! Item was: " + str(item
))

    def append(self, e):
        """ Appends element e to the end of the list, in O(1)

        """

        if self._head == None:
            self.add(e)
        else:
            new_node = Node(e)
            self._last.set_next(new_node) # NEW, we directly exploit _last pointer
            self._last = new_node # NEW, need to update _last
            self._size += 1  # NEW, need to update _size

    def insert(self, i, e):
        """ Insert an item at a given position.

            The first argument is the index of the element before which to insert, s
o list.insert(0, e)
            inserts at the front of the list, and list.insert(list.size(), e) is equ
ivalent to list.append(e).
            When i > list.size(), raises an Exception (default Python list appends i
nstead to the end :-/ )

        """
        if (i < 0):
            raise Exception("Tried to insert at a negative index! Index was:" + str(
i))

        count = 0
        current = self._head
        prev = None

        while (count < i and current != None):
            prev = current
            current = current.get_next()
            count += 1

        if (current == None):
            if (count == i):
                self.append(e)
            else:
                raise Exception("Tried to insert outside the list ! "
                                + "List size=" + str(count) + ", insert position=" +
                                + "List size=" + str(count) + "  insert position=" +
 str(i))
        else:
            #0 1
            #  i
            if (prev == None):
                self.add(e)
            else:
                new_node = Node(e)
                prev.set_next(new_node)
                new_node.set_next(current)

                self._size += 1 # NEW, need to update _size

    def index(self, e):
        """ Return the index in the list of the first item whose value is x.

            If item is not found, raises an Exception.
        """

        current = self._head
        count = 0

        while (current != None):
            if (current.get_data() == e):
                return count
            else:
                current = current.get_next()
                count += 1

        raise Exception("Couldn't find element " + str(e) )

    def pop(self):
        """ Remove the last item of the list, and return it.

            If the list is empty, an exception is raised.
        """
        if (self._head == None):
            raise Exception("Tried to pop an empty list!")
        else:

            current = self._head

            if (current.get_next() == None): # one element list
                popped = self._head
                self._head = None
                self._last = None  # NEW
            else:    # we have more than one element
                prev = None
                while current.get_next() != None:  # current will reach last element
                    prev = current
                    current = current.get_next()

                popped = current
                self._last = prev   # NEW
                prev.set_next(None)

            self._size -= 1  # NEW

            return popped.get_data()


    def last(self):
        """ Returns the last element in the list, in O(1).

            If list is empty, raises an Exception. Since v2
```

```
            If list is empty, raises an Exception. Since v2.
        """

        if (self._head == None):
            raise Exception("Tried to get the last element of an empty list!")
        else:
            return self._last.get_data()

class UnorderedListTest(unittest.TestCase):
    """ Test cases for UnorderedList v2

        Test cases are improved by adding a new method myAssert(self, unordered_list
, python_list)

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

    def myAssert(self, unordered_list, python_list):
        """ Checks provided unordered_list can be represented as the given python_li
st. Since v2
        """
        self.assertEquals(unordered_list.to_python(), python_list)
        # we check this new invariant about the size
        self.assertEquals(unordered_list.size(), len(python_list))
        # we check this new invariant about the last element
        if len(python_list) != 0:
            self.assertEquals(unordered_list.last(), python_list[-1])

    def test_init(self):
        ul = UnorderedList()

    def test_str(self):
        ul = UnorderedList()
        self.assertTrue('UnorderedList' in str(ul))
        ul.add('z')
        self.assertTrue('z' in str(ul))
        ul.add('w')
        self.assertTrue('z' in str(ul))
        self.assertTrue('w' in str(ul))

    def test_is_empty(self):
        ul = UnorderedList()
        self.assertTrue(ul.is_empty())
        ul.add('a')
        self.assertFalse(ul.is_empty())

    def test_add(self):
        """ Remember 'add' adds stuff at the beginning of the list ! """

        ul = UnorderedList()
        self.myAssert(ul, [])
        ul.add('b')
        self.myAssert(ul, ['b'])
        ul.add('a')
        self.myAssert(ul, ['a', 'b'])

    def test_size(self):
        ul = UnorderedList()
        self.assertEquals(ul.size(), 0)
        ul.add("a")
        self.assertEquals(ul.size(), 1)
        ul.add("b")
        self.assertEquals(ul.size(), 2)
        self.assertEquals(ul.size(), 2)

    def test_search(self):
        ul = UnorderedList()
        self.assertFalse(ul.search("a"))
        ul.add("a")
        self.assertTrue(ul.search("a"))
        self.assertFalse(ul.search("b"))
        ul.add("b")
        self.assertTrue(ul.search("a"))
        self.assertTrue(ul.search("b"))

    def test_remove_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.remove('a')

    def test_remove_one_element(self):
        ul = UnorderedList()
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('b')
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])

    def test_remove_two_element(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['a'])
        ul.remove('a')
        self.assertEquals(ul.to_python(), [])


    def test_remove_first_occurrence(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('b')
        with self.assertRaises(Exception):
            ul.remove('c')
        ul.remove('b')
        self.assertEquals(ul.to_python(), ['b'])
        ul.remove('b')
        self.assertEquals(ul.to_python(), [])

    def test_append(self):
        ul = UnorderedList()
        ul.append('a')
        self.myAssert(ul,['a'])
        ul.append('b')
        self.myAssert(ul,['a', 'b'])

    def test_insert_empty_list_zero(self):
        ul = UnorderedList()
        ul.insert(0, 'a')
        self.myAssert(ul, ['a'])

    def test_insert_empty_list_out_of_bounds(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.insert(1, 'a')
        with self.assertRaises(Exception):
            ul.insert(-1, 'a')
            ul.insert(-1, 'a')

    def test_insert_one_element_list_before(self):
        ul = UnorderedList()
        ul.add('b')
        ul.insert(0, 'a')
        self.myAssert(ul, ['a','b'])


    def test_insert_one_element_list_after(self):
        ul = UnorderedList()
        ul.add('a')
        ul.insert(1, 'b')
        self.myAssert(ul, ['a','b'])

    def test_insert_two_element_list_insert_before(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('b')
        ul.insert(0, 'a')
        self.myAssert(ul, ['a','b','c'])

    def test_insert_two_element_list_insert_middle(self):
        ul = UnorderedList()
        ul.add('c')
        ul.add('a')
        ul.insert(1, 'b')
        self.myAssert(ul, ['a','b', 'c'])

    def test_insert_two_element_list_insert_after(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        ul.insert(2, 'c')
        self.myAssert(ul, ['a','b', 'c'])


    def test_index_empty_list(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.index('a')

    def test_index(self):
        ul = UnorderedList()
        ul.add('b')
        self.assertEquals(ul.index('b'),  0)
        with self.assertRaises(Exception):
            ul.index('a')
        ul.add('a')
        self.assertEquals(ul.index('a'),  0)
        self.assertEquals(ul.index('b'),  1)

    def test_pop_empty(self):
        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.pop()

    def test_pop_one(self):
        ul = UnorderedList()
        ul.add('a')
        x = ul.pop()
        self.assertEquals('a', x)

    def test_pop_two(self):
        ul = UnorderedList()
        ul.add('b')
        ul.add('a')
        x = ul.pop()
        self.assertEquals('b', x)
        self.myAssert(ul, ['a'])
        y = ul.pop()
        self.assertEquals('a', y)
        self.myAssert(ul, [])

    def test_last(self):
        """ This tests only simple cases. More in-depth testing will be provided by 
calls to myAssert """

        ul = UnorderedList()
        with self.assertRaises(Exception):
            ul.last()
        ul.add('b')
        self.assertEquals(ul.last(), 'b')
        ul.add('a')
        self.assertEquals(ul.last(), 'b')
```

In [70]:

```
algolab.run(UnorderedListTest)
```

```
........................
----------------------------------------------------------------------
Ran 24 tests in 0.045s

OK
```

```
In [71]:
```