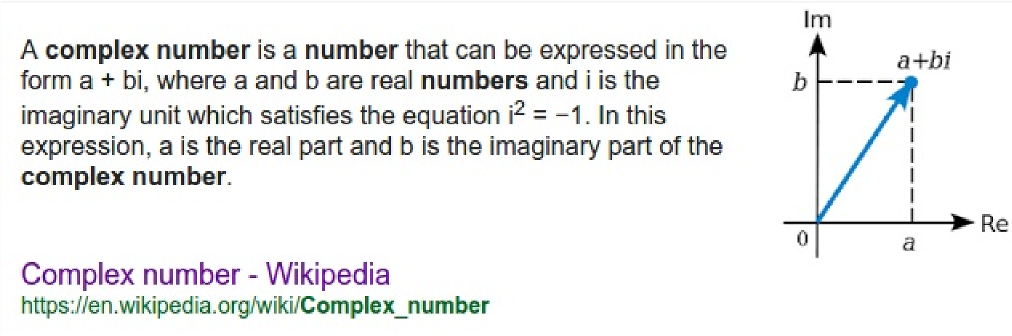
Chapter 2: Data Structures

class ComplexNumber

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

Let's try to define a complex number:



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

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

In [2]:

```
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):
        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)
```

In [3]:

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

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.009s

OK
```

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

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

In [4]:

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

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

In [5]:

```
phase = my_complex.phase()
print phase
```

```
1.03037682652
```

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

In [6]:

```
print my_complex
```

```
3 + 5i
```

We can also call methods that require a parameter like `log(base)` . Notice that `log` function returns a `ComplexNumber`, and Python will automatically pretty print it for us.

In [7]:

```
logarithm = my_complex.log(math.e)
print logarithm
```

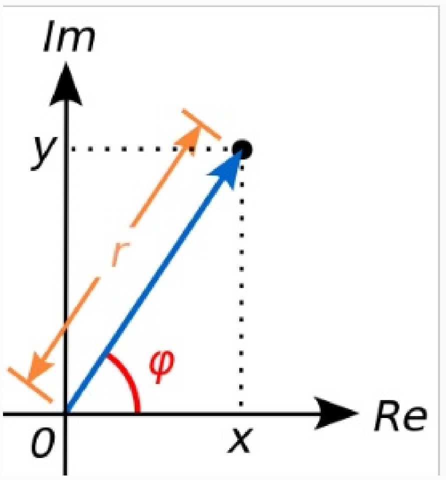
```
1.09861228867 + 1.03037682652i
```

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

Complex numbers magnitude

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

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



Implement the `magnitude` method, using this signature:

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

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

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

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

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:

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

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

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

Complex numbers isclose

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

Given two complex numbers

$$z_1 = a + bi$$

and

$$z_2 = c + di$$

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

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

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

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

and add this test case to `ComplexNumberTest` class:

```
def test_isclose(self):
        """  Notice we use `assertTrue` because we expect `isclose` to return
 a `bool` 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 [9]:

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

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

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

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

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

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

```python
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:

```python
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, usintaking 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/TheStackAbstractDataType.html)
- Implementing a Stack in Python (http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaStackinPython.html)
- Simple Balanced Parenthesis (http://interactivepython.org/runestone/static/pythonds/BasicDS/SimpleBalancedParentheses.html)
- Balanced Symbols - a General Case (http://interactivepython.org/runestone/static/pythonds/BasicDS/BalancedSymbols(AGeneralCase).html)

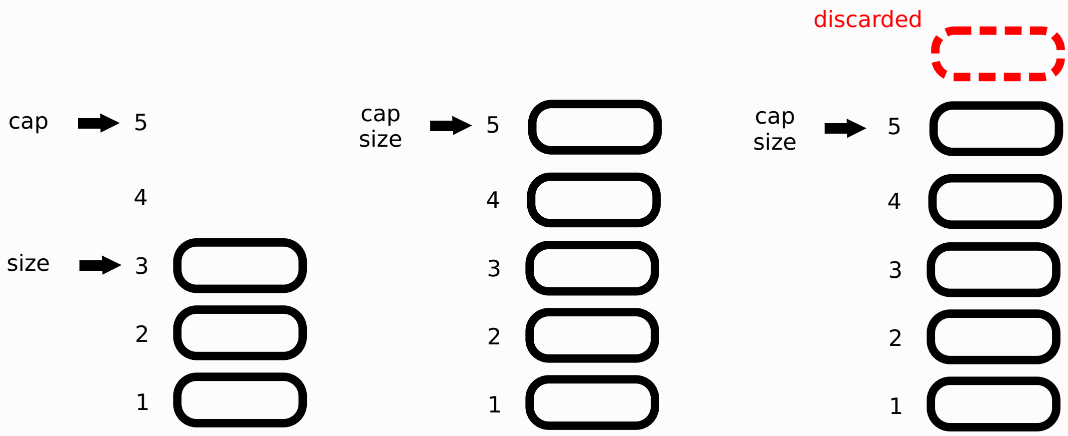
Stack exercises

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          ITEM peek()
int size                % Current size         if size > 0 then
int cap                 % Maximum size            return A[size]

STACK Stack(int dim)                        ITEM pop()
   STACK t ← new STACK                         if size > 0 then
   t.A ← new int[1 ... dim]                       ITEM t ← A[size]
   t.cap ← dim                                    size ← size − 1
   t.size ← 0                                     return t
   return t
                                            push(ITEM v)
boolean isEmpty()                              if size < cap then
   return size = 0                                size ← size + 1
                                                  A[size] ← v
int size()
   return size
```

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 [11]:

```
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
        """
```

```python
        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)
        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")
```

```
        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/TheUnorderedListAbstractDataType.html) on the book
- See Implementing UnorderedListLinkedLists (http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementinganUnorderedListLinkedLists.html) on the book

UnorderedList Exercises

In [12]:

```

```

1) Copy the following skeleton and unit tests, and then implement the missing methods

```
* This time there is no pseudo code, 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 !)**

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) Once you're done implementing the methods, implement an `append` method that works in O(1), by using an additional pointer in the data structure

3) Once you're done with previous points, copy the file you have into a new file, and rename the class into `BidirectionalList`.

3.1) Add to `Node` backlinks by adding the attribute `prev` and methods `get_prev(self)` and `set_prev(self, pointer)`. Then update all `BidirectionalList` methods 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.

3.2) Implement this method in `O(n)` 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 """
        raise Exception("TODO implement me")
```

Make sure this test 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())
```

UnorderedList Code Skeleton

In [13]:

```
class Node:
    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
```

```python
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, raising 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!")
```

```
        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, 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 :-/ )

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

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

        raise Exception("TODO implement me!")


    def pop(self):
        """ Remove the item at the given position in 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())
```

```python
    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_append(self):
        ul = UnorderedList()
        ul.append('a')
        self.assertTrue(ul.to_python(),['a'])
        ul.append('b')
        self.assertTrue(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'])
```

```
        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('a', x)
        self.assertEquals(ul.to_python(), ['b'])
        y = ul.pop()
        self.assertEquals('b', y)
```

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

Solutions

ComplexNumber Solution

In [14]:

```
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
```

```
l
            for float numbers!!!
        """
        self.assertAlmostEqual(ComplexNumber(0.0,1.0).phase(), math.pi / 2, delta=0.
001)

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


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


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


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

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

    def test_isclose(self):
        """  Notice we use `assertTrue` because we expect `isclose` to return a `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
```

```
        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 [15]:

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

```
.................
----------------------------------------------------------------------
Ran 17 tests in 0.010s

OK
```

Stack Solution

In [16]:

```
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
        # 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):
```

```
    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):
        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)
```

```
        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 [17]:

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

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

----------------------------------------------------------------------
Ran 8 tests in 0.013s

OK
```

UnorderedList Solution

In [18]:

```
class Node:
    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):
        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
```

```
        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:
            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.
            It is an error if there is no such item.
        """

        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 item at the given position in 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:
            head_item = self._head.get_data()
            self._head = self._head.get_next()
            return head_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()
```

```python
    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_append(self):
        ul = UnorderedList()
        ul.append('a')
        self.assertTrue(ul.to_python(),['a'])
        ul.append('b')
        self.assertTrue(ul.to_python(),['a', 'b'])

    def test_insert_empty_list_zero(self):
```

```
    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()
```

```
            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('a', x)
        self.assertEquals(ul.to_python(), ['b'])
        y = ul.pop()
        self.assertEquals('b', y)
        self.assertEquals(ul.to_python(), [])
```

In [19]:

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

```
.....................
----------------------------------------------------------------------
Ran 21 tests in 0.027s

OK
```

In [20]:

```

```