Lecture 11

- Set Class (page 234, exercise 6.15)
- BinaryNumber Class (page 235, exercise 6.18)

In previous class we designed and developed **Fraction** class – class for fractions.

In this class we'll design and develop two more classes: **Set** and **BinaryNumber**

Set Class
(page 234, exercise 6.15)

Define a class *Set* that keeps track of a collection of *distinct* objects. Internally, you should use a ***list*** as an instance variable, but you should make sure not to allow duplicates into your set.

Set Class
(page 234, exercise 6.15)

book recommends the following design:

Set	
_contents	
__ init__()	add(value)
__ contains__(value)	discard(value)

add - inserts the value into the set,
if its' not already there.

discard - removes the value from the set if it is found ,
and does nothing otherwise.

__contains__ is used to support the syntax *value* **in** *s*,
returning True or False appropriately.

Set Class
(page 234, exercise 6.15)

defining the *constructor*:

```
def __init__(self, l):
    ''' constructor of class Set '''

    n=len(l)

    # generating list with unique elem.
    # (see List Comprehension lecture)

self._contents=
      [l[i] for i in range(n) if
                          l[i] not in l[i+1:n]]
```

Set Class
(page 234, exercise 6.15)

defining/customizing the __ *str* __ method:

- in order to be able to display elements of the Set objects we need to customize the __str__ method.

```
def __str__(self):
         ''' print method '''

         return str(self._contents)
```

Set Class
(page 234, exercise 6.15)

defining/customizing the __ *contains* __ method:

- in order to be able to use <value> in <s> we need to customize the __contains__ method.

```
def __contains__(self,value):
   ''' checks presence of value in the Set
       object '''

   return value in self._contents
```

using the same syntax, but here it is applied to lists

Set Class
(page 234, exercise 6.15)

defining *add* method:

- if a value is not already in the Set object, we will add it.
- if it is already there, we'll do nothing

```
def add(self,value):
    ''' adds element to the Set object '''

    if value not in self._contents:
        self._contents.append(value)
```

Set Class
(page 234, exercise 6.15)

defining *discard* method:

- if a value is an element of the the Set object, remove
- if it is not an element of the Set object, do nothing

```
def discard(self,value):
    '''Discards an element from the Set.
       If the value is an element of the Set
       object, it removes it, otherwise does
       nothing.'''

    if value in self._contents:
       self._contents.remove(value)
```

Set Class
(page 234, exercise 6.15)

Note!

```
if value in self._contents:
    self._contents.remove(value)
```

We have to check for the presence of the value in the Set object, because if the element is not present in the list, and we will try to remove it, the function remove will give an error message, saying that we are trying to remove something which is not there. Since we don't want to see any error messages, we'll check beforehand for the presence, and remove it only if it is present.

Set Class
(page 234, exercise 6.15)

see the program set-class.py

Set Class
(page 234, exercise 6.15)

Did we build an immutable or mutable class?

BinaryNumber Class
(page 235, exercise 6.18)

Design an immutable class *BinaryNumber*

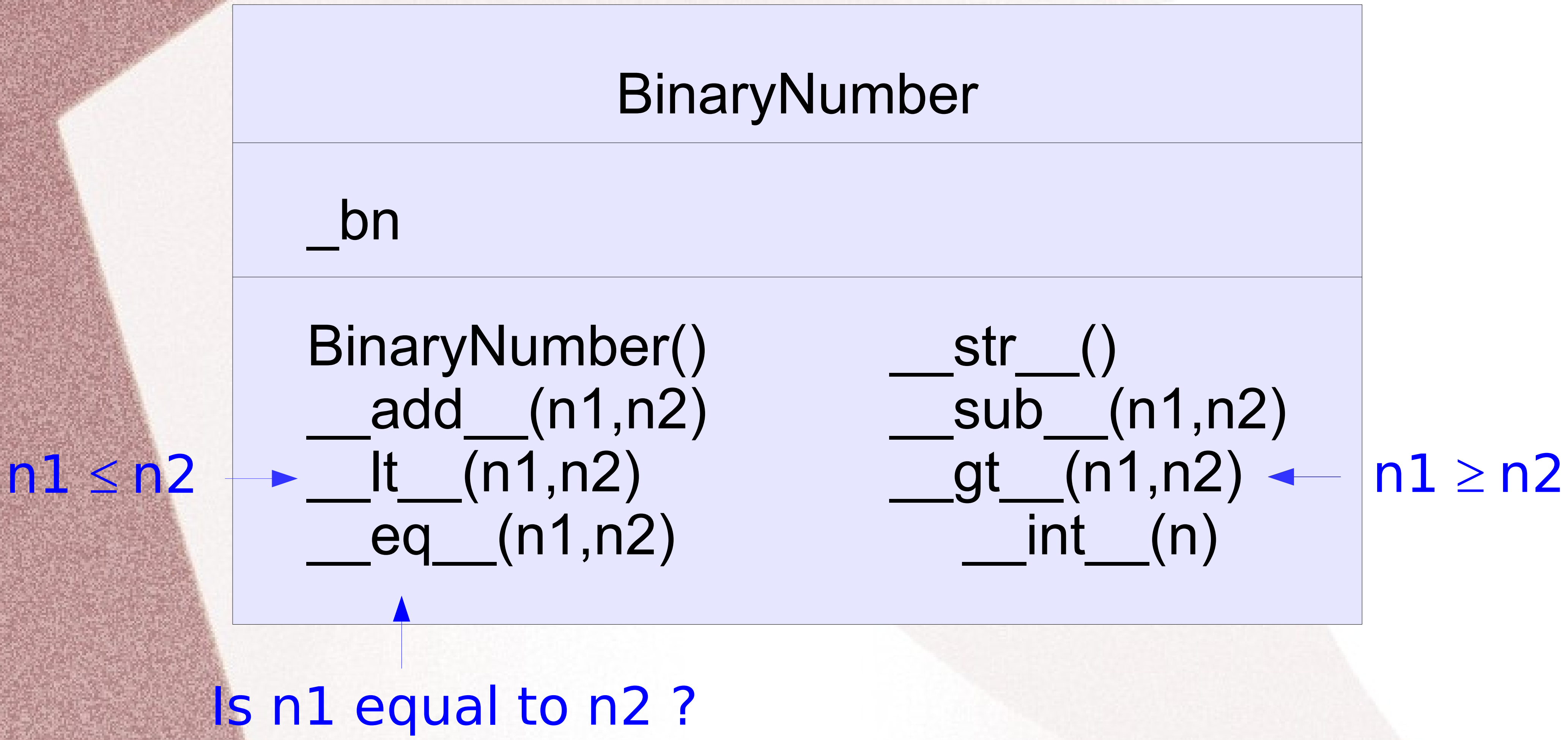
Internally, the only instance variable should be a text string that represents the binary value, of the form “11110100”.

Implement a constructor that takes a string parameter that specifies the original binary value.

Provide natural implementations for the special methods __add__, __sub__, __lt__, __gt__, __eq__, __str__, and __int__

Note: with the exception of __int__, we are not to convert the internal values back to integers when implementing the operations; perform the arithmetic directly upon the binary representations.

BinaryNumber Class
(page 235, exercise 6.18)



BinaryNumber Class
(page 235, exercise 6.18)

Here are some examples of arithmetic (before we proceed to implementation):

```
   11     1             1
   1111  1010          1010  0101
  +1010  1110        + 0010  1010
   ----------          ----------
1  1010  1000          1100  1111
```

BinaryNumber Class
(page 235, exercise 6.18)

Here are some examples of arithmetic (before we proceed to implementation):

```
   1111 11
   1111 1010
  +1010 1110
   ---------
 1 1010 1000
```

```
     1
     1010 0101
  +  0010 1010
     ---------
     1100 1111
```

```
    1010 0111
 -  0010 0010
    ---------
    1000 0101
```

```
        1
      0 2  2
    1110 0111
 -  0011 1111
    ---------
       0 1000
```

In the last example, the 2 above the third bit and the 1 beneath it in 1110 are crossed out.

BinaryNumber Class

(page 235, exercise 6.18)

Here are some examples of arithmetic (before we proceed to implementation):

```
   1111 11
   1111 1010
  +1010 1110
   ---------
1  1010 1000
```

```
   1
   1010 0101
+  0010 1010
   ---------
   1100 1111
```

```
   1010 0111
-  0010 0010
   ---------
   1000 0101
```

```
      2
   1110 0111
-  0011 1111
   ---------
   1010 1000
```

BinaryNumber Class
(page 235, exercise 6.18)

How to convert a binary number into a decimal number?

Example: 1110 1010

$0*2^0+1*2^1+0*2^2+1*2^3+0*2^4+1*2^5+1*2^6+1*2^7 =$
$0 + 2 + 0 + 8 + 0 + 32 + 64 + 128 =$
234

BinaryNumber Class
(page 235, exercise 6.18)

here is a definition of the *constructor*

```
def __init__(self, s):
  ''' constructor of class BinaryNumber '''

   if binary_number_check(s) == False:
     print("Warning: ",s,"is not a binary
string")

   self._bn = s
```

BinaryNumber Class (page 235, exercise 6.18)

here is a customization of the print method *__str__*:

```
def __str__(self):
   ''' print method '''

    if binary_number_check(self._bn) == True:
      return self._bn
    else:
       return "not a binary string"
```

BinaryNumber Class
(page 235, exercise 6.18)

See the finished BinaryNumber Class
in binaryNumbers-class.py

check the __lt__ and __gt__ methods

Homework assignment

Improve our Binary Numbers class:

- fix __lt__ method or update the constructor,
- add __gt__ method,
- add XOR, AND, OR methods (work on the strings of the same length)

In-Class Work – part 1

1. Define a class with *one attribute* of type `list`.
   Initially (during instantiation) it should be an empty list.

2. Then define the following methods of this class:
   - `__str__` to print the value of the *attribute* (i.e. of the list)
   - `add(item)` to add item of type `string` to the list
   - `get(position)` to get an element of the list from the `position`.
   - `remove(item)` to remove an `item` from the list.

3. Finally, define the `main()` method: write a code that will
   - create one instance (object) of the class defined above;
   - inhabit it with the following strings (words): 'money', 'logarithm', 'sum', 'sun';
   - print the element of the object's attribute (list) on the third position;
   - and remove the string (word) 'sun' from the object's attribute (list)

In-Class Work – part 2

Let's try to build something like **Dictionary**.

1) Modify your class from part 1, so that its only attribute is a list that has tuples as elements (first element of the tuple is the word, and the second element of the list is the meaning/description of this word)

for example (the value of the attribute of the class) :
[(“lake”,“a body of (usually fresh) water surrounded by land ”),
(“sand”,”a loose material consisting of grains of rock or coral ”]

2) Think on how to add the following functionality to your class:

• if an item is added to the object's attribute (i.e. to our dictionary get an entry), the list is re-ordered (alphabetically) by the first elements of tuples.

• define a method that allows to pull up the words staring with a pattern of letters, i.e.
get(“la”) will return tuple (“lake”,“a body of (usually fresh) water surrounded by land ”)