Lecture 4 – User defined types

- Design guidelines – How to organize your source code
- Exceptions
- User Defined types

How to organize your source code

Responsibilities

Responsibility is a reason to change

- function: do a computation
- module: all the functions responsibilities

Single responsibility principle (SRP)

A function/module should have one, and only one, reason to change.

```
#Function with multiple responsibilities
#implement user interaction (read/print)
#implement a computation (filter)
def filterScore():
    st = input("Start score:")
    end = input("End score:")
    for c in l:
        if c[1]>st and c[1]<end:
            print c
```

Multiple responsibilities

- Harder to understand and use
- Unable to test
- Unable to reuse
- Difficult to maintain and evolve

Separation of concerns

Separation of concerns (**SoC**) is the process of separating a computer program into distinct features that overlap in functionality as little as possible

```
def filterScoreUI():
   st = input("Start sc:")
   end = input("End sc:")
   rez = filtrareScore(l,st, end)
   for e in rez:
        print e


def filterScore(l,st, end):
    """
    filter participants
    l - list of participants
    st, end - integers -scores
    return list of participants
        filtered by st end score
    """
    rez = []
    for p in l:
        if p[1]>st and p[1]<end:
            rez.append(p)
    return rez
```

```
def testScore():
  l = [["Ana", 100]]
  assert filterScore(l,10,30)==[]
  assert filterScore(l,1,30)==l
  l = [["Ana", 100],["Ion", 40],["P", 60]]
 assert filterScore(l,3,50)==[["Ion", 40]]
```

Dependency

- function: a function invokes another function
- module: any function from the module invoke a function from a another module

In order to increase re usability we need to manage dependency

Coupling

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements

The more the connections between one module and the rest, the harder to understand that module, the harder to re-use that module in another situation, the harder it is to isolate failures caused by faults in the module ⇒The lower the coupling the better

Low coupling facilitate the development of programs that can handle change because they minimize the interdependency between functions/modules

Cohesion

Cohesion is a measure of how strongly related and focused the responsibilities of an element are.
A module may have:

- **High Cohesion**: it is designed around a set of related functions
- Low Cohesion: it is designed around a set of unrelated functions

A cohesive module performs a single task within a software, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing

The less tightly bound the internal elements, the more disparate the parts to the module, the harder it is to understand ⇒The higher the cohesion the better

Cohesion is a more general concept than the SRP, modules that are follow the SRP tend to have high cohesion.

Layered Architecture

Structure the application such that:

- Minimizes coupling between modules (modules don't need to know much about one another to interact, makes future change easier)
- Maximizes the cohesion of each module (the contents of each module are strongly inter-related)

Layered Architecture – is an architectural pattern that allows you to design flexible systems using components (The components are as independent of each other as possible)

- Each layer communicates only with the layer immediately below it.
- Each layer has a well-defined interface used by the layer immediately above. (implementation details are hidden)

Common layers in an information system logical architecture

- User interface / Presentation (User interface related functions/modules/classes)
- Domain / Application Logic (provide the application functions determined by the use-cases)
- Infrastructure (general/utility functions/modules/classes)
- Application coordinator

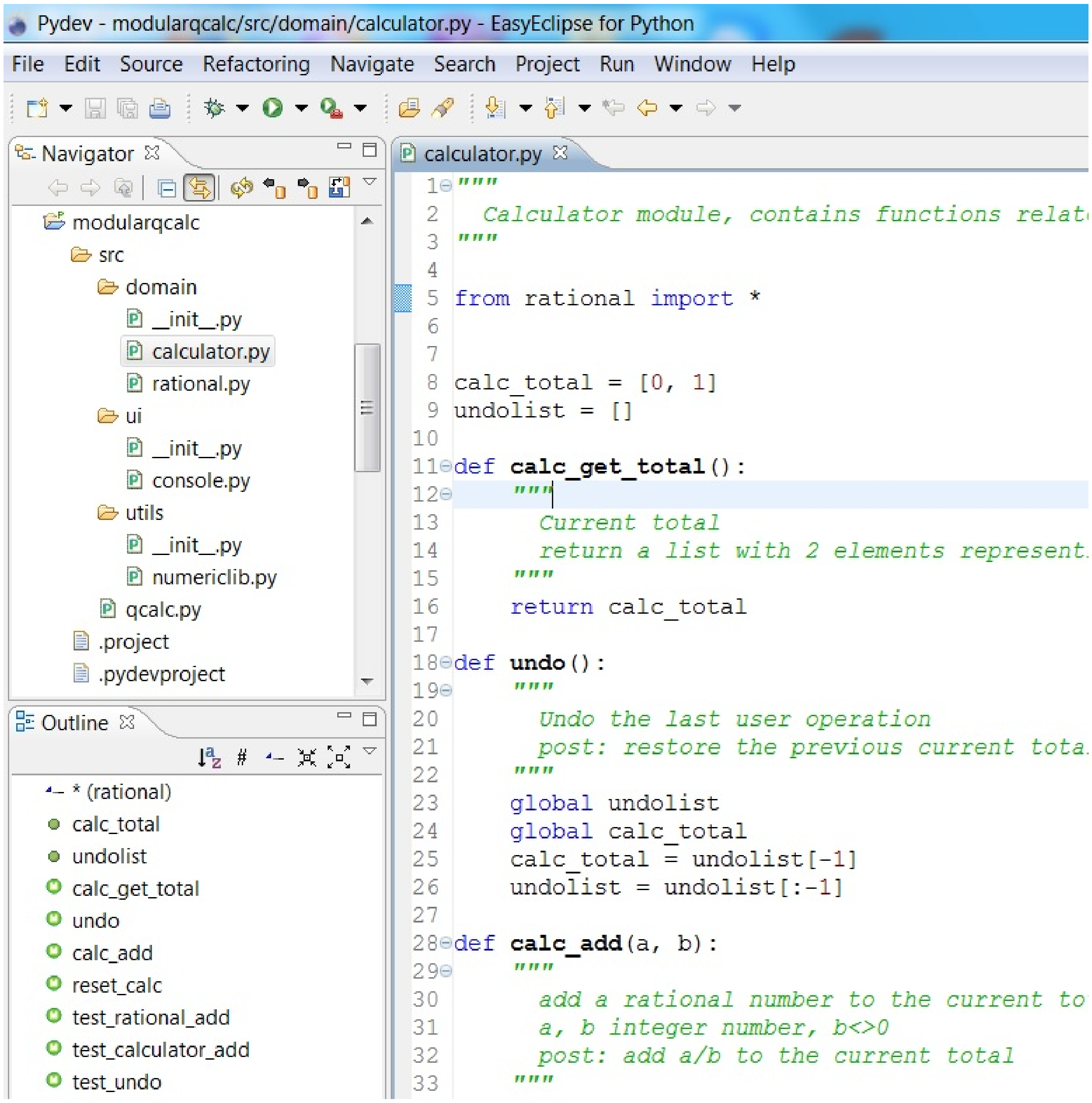
Layered Architecture – simple example

```
#Ui
def filterScoreUI():                               #manage the user interaction
    st = input("Start sc:")
    end = input("End sc:")
    rez = filterScoreDomain(st, end)
    for e in rez:
        print e
```

```
#domain
l = [["Ion",50],["Ana",30],["Pop",100]]
def filterScoreDomain(st, end):                    #filter the score board
    global l
    rez = filterMatrix(l, 1, st, end)
    l = rez
    return rez
```

```
#Utility function - infrastructure
def filterMatrix(matrice, col, st, end):           #filter matrix lines
    linii = []
    for linie in matrice:
        if linie[col]>st and linie[col]<end:
            linii.append(linie)
    return linii
```

How to organize your python projects



```
"""
  Calculator module, contains functions relat
"""

from rational import *


calc_total = [0, 1]
undolist = []

def calc_get_total():
    """
      Current total
      return a list with 2 elements represent
    """
    return calc_total

def undo():
    """
      Undo the last user operation
      post: restore the previous current tota
    """
    global undolist
    global calc_total
    calc_total = undolist[-1]
    undolist = undolist[:-1]

def calc_add(a, b):
    """
      add a rational number to the current to
      a, b integer number, b<>0
      post: add a/b to the current total
    """
```

Exceptions

Errors detected during execution are called **exceptions**.

An exception is *raised* at the point where the error is detected;

- raised by the python interpreter
- raised by the code to signal exceptional situation (broken precondition)

```
>>> x=0
>>> print 10/x

Trace back (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print 10/x
ZeroDivisionError: integer division or modulo by zero
```

```
def rational_add(a1, a2, b1, b2):
    """
    Return the sum of two rational numbers.
    a1,a2,b1,b2 integer numbers, a2<>0 and b2<>0
    return a list with 2 int, representing a rational number a1/b2 + b1/b2
    Raise ValueError if the denominators are zero.
    """
    if a2 == 0 or b2 == 0:
        raise ValueError("0 denominator not allowed")
    c = [a1 * b2 + a2 * b1, a2 * b2]
    d = gcd(c[0], c[1])
    c[0] = c[0] / d
    c[1] = c[1] / d
    return c
```

Execution flow

Exceptions are a means of breaking out of the normal flow of execution in order to handle errors or other exceptional conditions.

```python
def compute(a,b):
    print "compute :start "
    aux = a/b
    print "compute:after division"
    rez = aux*10
    print "compute: return"
    return rez

def main():
    print "main:start"
    a = 40
    b = 1
    c = compute(a, b)
    print "main:after compute"
    print "result:",c*c
    print "main:finish"

main()
```

Exception handling is the process of handling error conditions in a program systematically by taking the necessary action.

```
try:
     #code that may raise exceptions
     pass
except ValueError:
     #code that handle the error
     pass
```

Exceptions need to be *handled* by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred otherwise the program will crash

raise, try-except statements

```
try:
     calc_add (int(m), int(n))
     printCurrent()
except ValueError:
     print ("Enter integers for m, n, with n!=0")
```

Exception handling

- multiple except cases,
- propagate information about the exception

```
def f():
#   x = 1/0
    raise ValueError("Error Message")

try:
    f()
except ValueError as msg:
    print "handle value error:", msg
except KeyError:
    print "handle key error"
except:
    print "handle any other errors"
```

Only use exceptions to:

- signal an exceptional situation – the function is unable to perform the promised situation
- enforce preconditions

Do not use exception just to control the execution flow

Specification

Abstraction (function acting as a black box) will only work if we provide:

- meaningful name for the function
- short description of the function (the problem solved by the function)
- meaning/type of each input parameter
- conditions imposed over the input parameters (precondition)
- meaning/type of each output parameter
- the relation between the input and output parameters (post condition)
- **Exceptions** that the function may signal (raise)

A **precondition** is a condition that must be true just prior to the execution of some section of code.
A **post condition** is a condition that must be true just after the execution of some section of code.

```
def gcd(a, b):
    """
    Return the greatest common divisor of two positive integers.
    a,b integer numbers
    return an integer number, the  greatest common divisor of a and b
    Raise ValueError if a<=0 or b<=0
    """
```

Test case for exceptions

```
def test_rational_add():
    """
      Test function for rational_add
    """
    assert rational_add(1, 2, 1, 3) == [5, 6]
    assert rational_add(1, 2, 1, 2) == [1, 1]
    try:
        rational_add(2, 0, 1, 2)
        assert False
    except ValueError:
        assert True
    try:
        rational_add(2, 3, 1, 0)
        assert False
    except ValueError:
        assert True
```

```
def rational_add(a1, a2, b1, b2):
    """
    Return the sum of two rational numbers.
    a1,a2,b1,b2 integer numbers, a2<>0 and b2<>0
    return a list with 2 ints, representing a rational number a1/b2 + b1/b2
    Raise ValueError if the denominators are zero.
    """
    if a2 == 0 or b2 == 0:
        raise ValueError("0 denominator not allowed")
    c = [a1 * b2 + a2 * b1, a2 * b2]
    d = gcd(c[0], c[1])
    c[0] = c[0] / d
    c[1] = c[1] / d
    return c
```

User defined type

Object oriented programming is a programming paradigm that use objects to design applications.

Objects and Classes

Types classifies values. A type denotes a **domain** (a set of values) and **operations** on those values.

Classes

A **class** is a construct that is used as a template to create instances of itself – referred to as class instances, class objects, instance objects or simply objects. A class defines constituent members which enable these class instances to have *state* and behaviour.

Class definition in python

```
class MyClass:
    <statement 1>
    ….
    <statement n>
```

Class definition is an executable statement.
The statements inside a class definition will usually be function definitions, but other statements are allowed
When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

Objects

Object is a collection of data and functions that operates on that data.
Class instances are of the type of the associated class.
Class objects support two kinds of operations: attribute (data or method) references and instantiation.

Creating instances of a class (__init__)

Class *instantiation* uses function notation.

```
x = MyClass()
```

The instantiation operation ("calling" a class object) creates an empty object. x will be an instance of type MyClass

A class may define a special method named __init__

```
class MyClass:
    def __init__(self):
        self.someData = []
```

__init__ :

- create an instance
- use "self" to refer to that instance

We can have init method with some parameters

```
class RationalNumber:
    """
       Abstract data type for rational numbers
       Domain: {a/b where a and b are integer numbers b!=0}
    """

    def __init__(self, a, b):
        """
          Creates a new instance of RationalNumber
        """
        self.n = a
        self.m = b

   r1 = RationalNumber(1,3)  #create the rational number 1/3
```

self.n = a vs n=a

1 Creates an attribute for the current instance
2 Creates a function local variable