Chapter 2: Data Abstraction

- Abstract Data Types
- ADTs and Objects
- ADTs and Object-oriented programming
- Incremental Development and Unit Testing
- Unit Testing Card’s methods
- Homework assignment
Values

A value is a unit of information used in a program. It can be associated with a constant or variable (a name) by an assignment statement:

```
person = 'George'
n = 4
```

All primitive type values (integer, float, string) are represented internally in the computer program’s memory as a series of zeros and ones.

More complex types have values which are combinations of more primitive types.
A **Data Type** is the set of possible values which all represent the same type of information and share the same behavior. In Python, most data types are classes, and a value of some data type is an object in that class.

- `int`
- `str`
- `float`
- `list`
- `dict`
- `file`
Defining an ADT = Specification

Data Abstraction

The data is represented using abstract attributes.

Example: a card has attributes suit and rank. The behavior is given by specifying functions, with the signature, preconditions, and postconditions of procedural abstraction.

Data Abstraction is the hiding of the primitive components comprising the values of some type, and hiding the implementation of the operations using that type.

Abstract Data Type (ADT) is described by providing a specification for the data type, independent of any actual implementation, i.e. describes what operations are supported by the ADTs.
Defining an ADT = Specification

Example: playing card  Pages 41-42

ADT Card:
A simple playing card, characterized by:

- rank: integer, in the range 1-13
- suit: a character in ‘‘cdhs’’ for clubs, diamonds, hearts, and spades.

Operations:
- create(rank, suit): create a new card; pre ...
- suit(): card suit; pre, post ...
- rank(): card rank; pre, post ...
- suitName(): card suit name; pre, post ...
- rankName(): card rank name; pre, post ...
- toString(): string representation of card
Implementation of an ADT

From Description to Implementation

Given a description of ADT we can implement it.

see code in cardADT.py and test in test-cardADT.py

Concrete Representation

The abstract attributes are represented using types from the programming language or previously defined classes.

Example: a suit is now a member of the Python string class str, and is allowed to have the values ‘c’, ‘d’, ‘h’ or ‘s’.
Class Specification

Python Classes
ADTs become **Python classes**, and their behaviors become **methods** for those classes. See `Card_spec.py`

Data Abstraction
In the class definition, a comment will tell how the concrete representation corresponds to the abstract attributes (for example, the letter ’c’ corresponds to the suit ’clubs’).

Functional Abstraction
Each method specification includes a comment listing all preconditions and postconditions.
Class Specification

Python Classes
ADTs become Python classes, and their behaviors become methods for those classes. See Card_spec.py.

Data Abstraction
In the class definition, a comment will tell how the concrete representation corresponds to the abstract attributes (for example, the letter 'c' corresponds to the suit 'clubs').

Functional Abstraction
Each method specification includes a comment listing all preconditions and postconditions.
Class Specification

Python Classes
ADTs become Python classes, and their behaviors become methods for those classes. See Card_spec.py

Data Abstraction
In the class definition, a comment will tell how the concrete representation corresponds to the abstract attributes (for example, the letter 'c' corresponds to the suit 'clubs').

Functional Abstraction
Each method specification includes a comment listing all preconditions and postconditions.
Class Implementation

Data Representation
In the class definition, the constructor __init__ will take parameters to set the attributes of new objects (or use default values).

Method Implementation
The actual Python code to implement the behavior of each method follows the comments listing preconditions and postconditions. Typical methods are mutators, which change attribute values, and accessors which return attribute values without changing them.
**Class Implementation**

**Data Representation**

In the class definition, the constructor `__init__` will take parameters to set the attributes of new objects (or use default values).

**Method Implementation**

The actual Python code to implement the behavior of each method follows the comments listing preconditions and postconditions. Typical methods are mutators, which change attribute values, and accessors which return attribute values without changing them.
Changing the Representation

The Abstraction Barrier

By keeping the specification (see Card-spec.py) and implementation (see Card.py) separate, it is possible to change the implementation—say, to use a more efficient algorithm—without having to change any program that uses the ADT respectful of its specification.
The Abstraction Barrier

The program which uses the ADT only has access to it through its methods, which are written to obey the specification—what types of parameters are passed and what types of values are returned. The concrete representation can change, but as long as the methods have the same signatures, and honors the same contracts (preconditions and postconditions), the program which calls them will still work.
Object Oriented Design (OOD) and Object Oriented Programming (OOP)

OOD and ADTs

Data Abstraction is only one of several ideas that have helped to advance software engineering.

OO design and programming uses ADTs as well as other principles.

Most OO gurus talk about three features that together make development truly object-oriented: encapsulation, polymorphism, and inheritance.
**Encapsulation**

**Objects** *know stuff* (data) and *do stuff* (operations). The process of packaging some data along with the set of operations that can be performed on the data is called **encapsulation**.

- also known as *information hiding*
- separates the issues of “what to do” from issues of “how to do” it.
- gives us implementation independence

Encapsulation alone makes the system *object-based* only.
Polymorphism

The word *polymorphism* means “many forms”.

This is the principle that sending the same message (that is, calling the same method) to objects in different classes or different types should make the objects behave the same.

Example: recall *cs1graphics* library, where we can draw different shapes. Rectangle, circle, polygon, ... can be all drawn into a window. The behavior(result) of the *draw* operation is similar for all the drawable objects, but how it is performed is different for each of them.
**Object Oriented Design (OOD) and Object Oriented Programming (OOP)**

**Inheritance**

Classes which share behaviors should not have to re-implement these behaviors if they can be *inherited* from a base class which implements that same behavior.

This principle promotes reuse of code, which in turn makes software more reliable, since bugs are more localized.

Terminology:
- parent class - child class
- superclass - subclass
Unit testing

Once the development is broken into separate classes, it is nice to be able to test each class once it’s developed.

Moreover, it is very convenient to test the class as it’s being developed!

Recall the tests that we have for cardADT.py and for Card.py.

How about testing just one of the behaviors/operations? Like rank, or rankName, ...

Testing a component in isolation is known as unit testing.
Benefits of writing unit tests

- tests can be run again when we go back and modify the code
- running a modified program against the previously successful tests is called regression testing
- writing unit tests while writing the class helps to work out the design of a class
- test-driven-development is when tests are written before any actual production code is added to the system. This way as each function/method is added, it is immediately testable.
Test-driven Development

- write the original class with each method containing just pass statement
- write the test code for a method, and then
- implement enough of the class to get the test to pass
- keep repeating the process of writing a test and modifying the class until the class is complete and passes all the tests.
Let's test the `rank()` method of our Card class.

```python
import sys
import unittest  # a framework for unit testing
from Card import *

class RankTest(unittest.TestCase):
    
    """ Tests Rank methods: rank() and rankName() ""
    def testRanks(self):
        """ creates cards of rank 1 through 13 of clubs and verifies that the created card’s rank is equal to the rank it was created with ""
```
UNIT TESTING CARD’S rank() METHOD

TestCase
t

**TestCase** class defines a number of useful methods for unit tests. Two commonly used:

* `assertEqual` (also known as `failUnlessEqual`
* `assertNotEqual` (also known as `failIfEqual`

Each method takes two parameters that are tested for equality.

Within our class `RankTest`, every method that starts with `test` will be called automatically by the `unittest` framework.
def testRanks(self):
    """creates cards of rank 1 through 13 of clubs and verifies ... """

    for i in range(1,14):
        myCard = Card(i,'c')  # create i of clubs
        self.assertEqual(myCard.rank(),i)

def main(argv):
    unittest.main()

if __name__ == '__main__':
    main(sys.argv)
Augment our `unitTestingCard.py` by writing unit tests to test the methods `suit`, `suitName` and `rankName` for `Card.py`.

see the supplied `Chapter2-unitTestingCardHW.py`