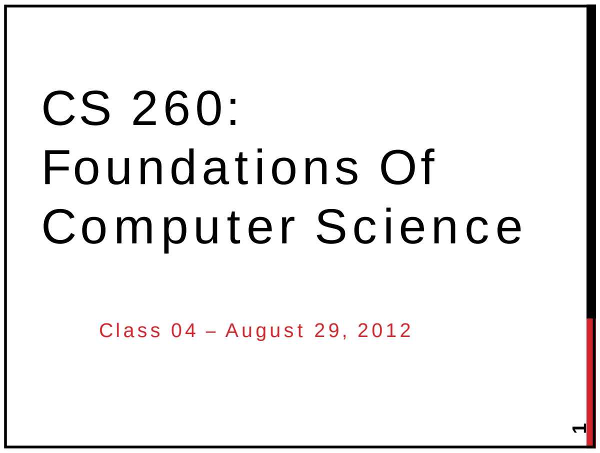
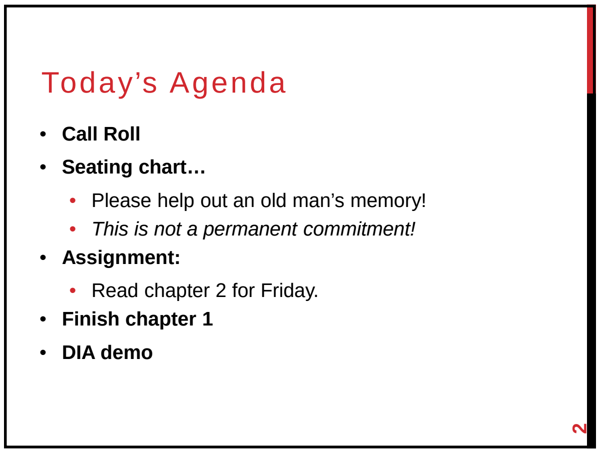


CS 260: Foundations Of Computer Science

Class 04 – August 29, 2012

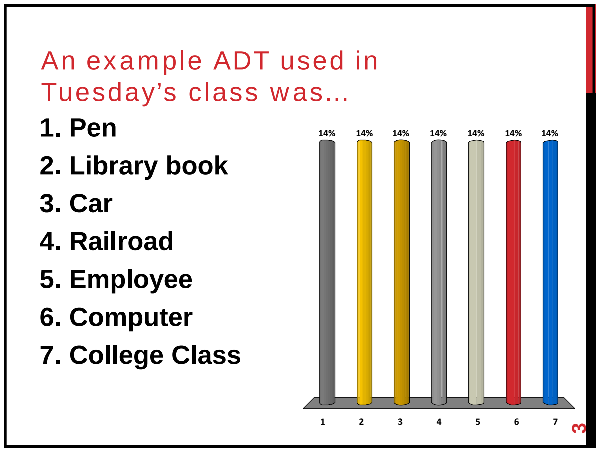
1



Today's Agenda

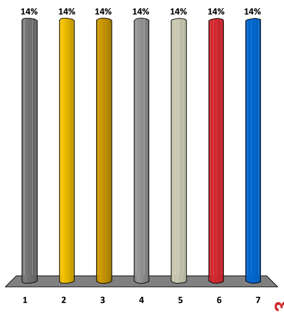
- **Call Roll**
- **Seating chart…**
  - Please help out an old man's memory!
  - *This is not a permanent commitment!*
- **Assignment:**
  - Read chapter 2 for Friday.
- **Finish chapter 1**
- **DIA demo**

2

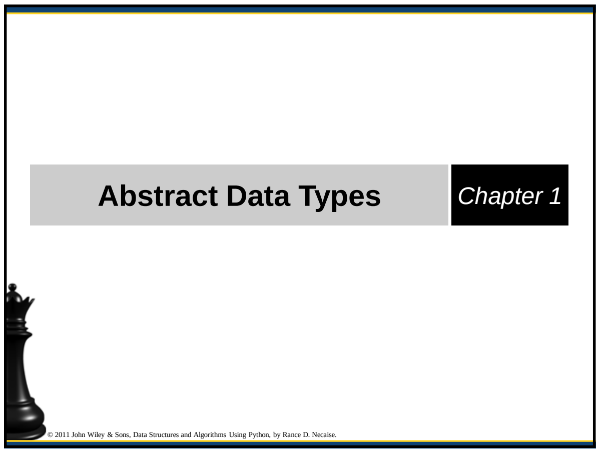


An example ADT used in Tuesday's class was…

1. **Pen**
2. **Library book**
3. **Car**
4. **Railroad**
5. **Employee**
6. **Computer**
7. **College Class**

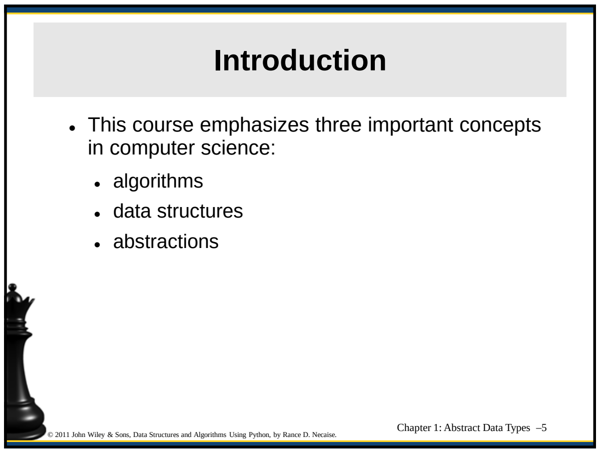


3



Abstract Data Types

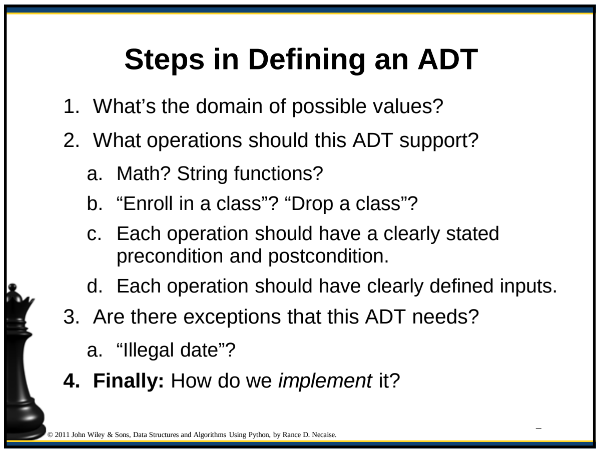
Chapter 1



Introduction

- This course emphasizes three important concepts in computer science:
  - algorithms
  - data structures
  - abstractions

Chapter 1: Abstract Data Types –5



Steps in Defining an ADT

1. What's the domain of possible values?
2. What operations should this ADT support?
   a. Math? String functions?
   b. "Enroll in a class"? "Drop a class"?
   c. Each operation should have a clearly stated precondition and postcondition.
   d. Each operation should have clearly defined inputs.
3. Are there exceptions that this ADT needs?
   a. "Illegal date"?
4. **Finally:** How do we *implement* it?

–

The Date ADT

- Example of a simple ADT.
- Represents a single day in the proleptic Gregorian calendar:
  - First date of the Gregorian calendar
    - Friday, October 15, 1582
  - What about earlier dates?
    - use proleptic Gregorian calendar
    - extension for accommodating earlier dates
    - first date: November 24, 4713 BC

Defining Operations

- The ADT definition should specify:
  - required inputs and resulting outputs.
  - state of the ADT instance before and after the operation is performed.

Precondition

- Condition or state of the ADT instance and data inputs before the operation is performed.
  - Assumed to be true.
  - Error occurs if the condition is not satisfied.
    - ex: index out of range
  - Implied conditions
    - the ADT instance has been created and initialized.
    - valid input types.

Postcondition

- Result or state of the ADT instance after the operation is performed.
  - Will be true if the preconditions are met.
    - given: `x.pop(i)`
    - the ith item will be removed if `i` is a valid index.

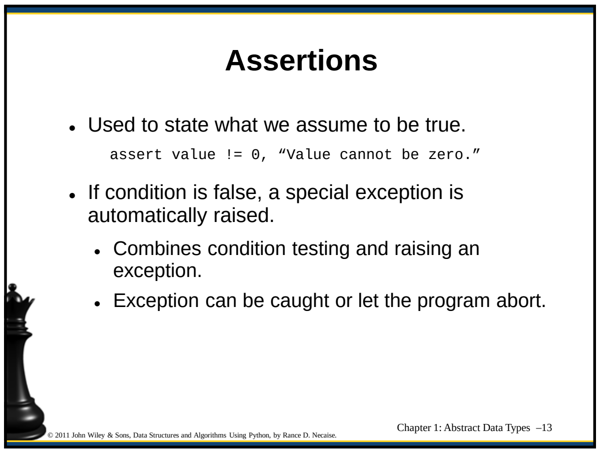
Postcondition

- The specific postcondition depends on the type of operation:
  - Access methods and iterators
    - no postcondition.
  - Constructors
    - create and initialize ADT instances.
  - Mutators
    - the ADT instance is modified in a specific way.

Exceptions

- OOP languages raise exceptions when errors occur.
  - An event that can be triggered by the program.
  - Optionally handled during execution.
- Example:

```
myList = [ 12, 50, 5, 17 ]
print( myList[4] )

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

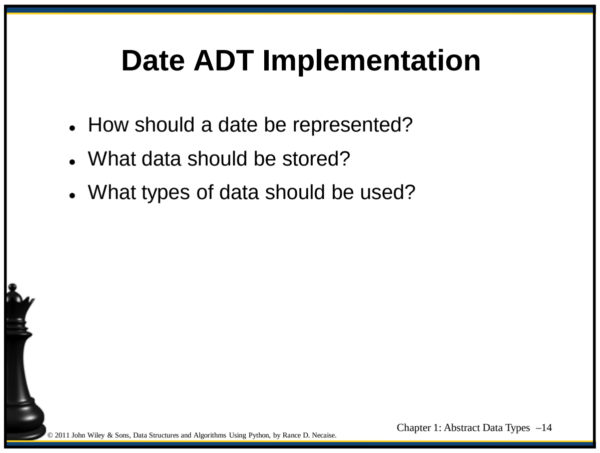


Assertions

- Used to state what we assume to be true.

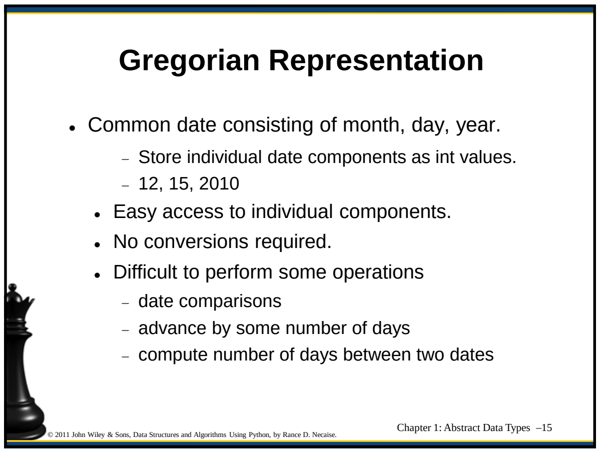
```
assert value != 0, “Value cannot be zero.”
```

- If condition is false, a special exception is automatically raised.
  - Combines condition testing and raising an exception.
  - Exception can be caught or let the program abort.



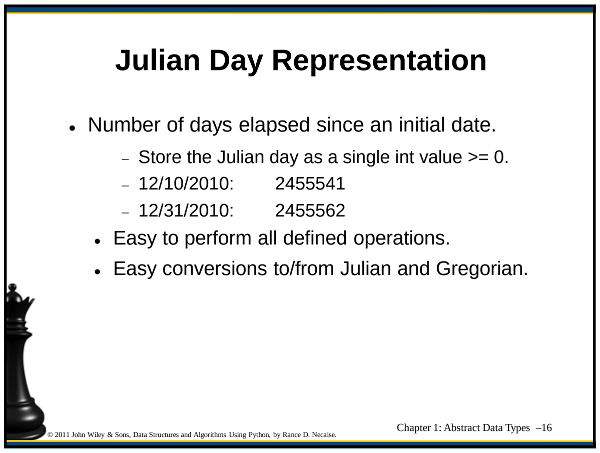
Date ADT Implementation

- How should a date be represented?
- What data should be stored?
- What types of data should be used?



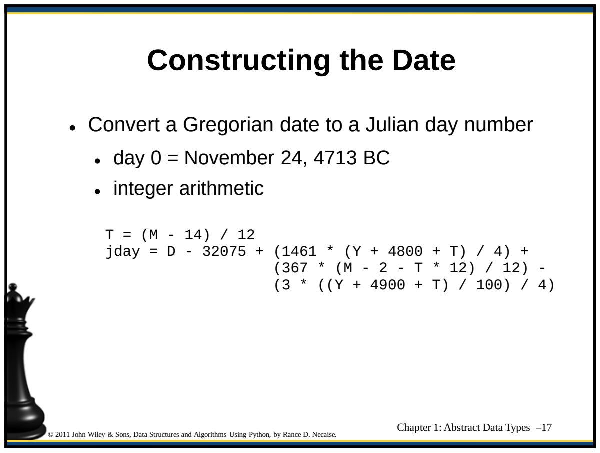
Gregorian Representation

- Common date consisting of month, day, year.
  - Store individual date components as int values.
  - 12, 15, 2010
  - Easy access to individual components.
  - No conversions required.
  - Difficult to perform some operations
    - date comparisons
    - advance by some number of days
    - compute number of days between two dates



Julian Day Representation

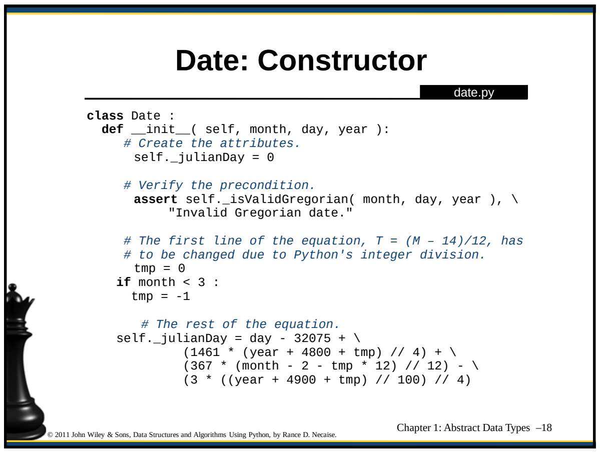
- Number of days elapsed since an initial date.
  - Store the Julian day as a single int value >= 0.
  - 12/10/2010: 2455541
  - 12/31/2010: 2455562
  - Easy to perform all defined operations.
  - Easy conversions to/from Julian and Gregorian.



Constructing the Date

- Convert a Gregorian date to a Julian day number
  - day 0 = November 24, 4713 BC
  - integer arithmetic

```
T = (M - 14) / 12
jday = D - 32075 + (1461 * (Y + 4800 + T) / 4) +
                   (367 * (M - 2 - T * 12) / 12) -
                   (3 * ((Y + 4900 + T) / 100) / 4)
```



Date: Constructor

date.py

```
class Date :
  def __init__( self, month, day, year ):
      # Create the attributes.
        self._julianDay = 0

      # Verify the precondition.
        assert self._isValidGregorian( month, day, year ), \
             "Invalid Gregorian date."

      # The first line of the equation, T = (M – 14)/12, has
      # to be changed due to Python's integer division.
        tmp = 0
     if month < 3 :
       tmp = -1

        # The rest of the equation.
     self._julianDay = day - 32075 + \
              (1461 * (year + 4800 + tmp) // 4) + \
              (367 * (month - 2 - tmp * 12) // 12) - \
              (3 * ((year + 4900 + tmp) // 100) // 4)
```

Protected Members

- Python does not provide for a technique to protect attributes and methods from direct access.
  - We use identifiers beginning with an underscore.
  - Rely on the user to not attempt direct access.

```
self._julianDay = 0
```

Helper Methods

- Methods used internally to implement the class.
  - Allow for the subdivision of larger methods.
  - Help to reduce code repetition.
- Not meant to be accessed from the outside.

```
self._isValidGregorian( month, day, year )
```

Julian to Gregorian

- To access the Gregorian components, convert Julian day back to Gregorian.

date.py

```
class Date :
# ...
  def _toGregorian( self ):
        A = self._julianDay + 68569
        B = 4 * A // 146097
        A = A - (146097 * B + 3) // 4
        year = 4000 * (A + 1) // 1461001
        A = A - (1461 * year // 4) + 31
        month = 80 * A // 2447
        day = A - (2447 * month // 80)
        A = month // 11
        month = month + 2 - (12 * A)
        year = 100 * (B - 49) + year + A
        return month, day, year
```

Date: Date Components

date.py

```
class Date :
# ...
  def month( self ):
       return (self._toGregorian())[0]

  def day( self ):
    return (self._toGregorian())[1]

  def year( self ):
    return (self._toGregorian())[2]

  def __str__( self ):
    month, day, year = self._toGregorian()
    return "%02d/%02d/%04d" % (month, day, year)
```

Date: Day of Week

- Can be determined from the Julian day.

date.py

```
class Date :
# ...
  def dayOfWeek( self ):
    month, day, year = self._toGregorian()
    if month < 3 :
      month = month + 12
      year = year – 1

     # Returns 0...6 for Monday...Sunday.
    return ((13 * month + 3) // 5 + day + \
           year + year // 4 - year // 100 + year // 400) % 7
```

Overloading Operators

- We can implement methods to define many of Python's standard operators.
  - Allows for more natural use of objects.
  - Limit use of operator methods for meaningful purposes.

Date: Comparable

- Need only implement 3 of the 6 comparable operators.

date.py

```
class Date :
# ...
  def __eq__( self, otherDate ):
    return self._julianDay == otherDate._julianDay

  def __lt__( self, otherDate ):
    return self._julianDay < otherDate._julianDay

  def __le__( self, otherDate ):
    return self._julianDay <= otherDate._julianDay
```

Bags

- A *bag* is a basic container like a shopping bag that can be used to store collections.
- There are several variations:
  - simple bag
  - grab bag
  - counting bag

Bag ADT

- A *simple bag* is a container that stores a collection with duplicate values allowed. The elements
  - are stored individually
  - have no particular order
  - must be comparable

- Bag()
- *length*()
- *contains*(item)
- add(item)
- remove(item)
- *iterator*()

Bag: Example 1

```
# Creates a bag and fills it with values. The user is then
# prompted to guess a value contained in the bag.

myBag = Bag()
myBag.add( 19 )
myBag.add( 74 )
myBag.add( 23 )
myBag.add( 19 )
myBag.add( 12 )

value = int( input("Guess a value contained in the bag.") )
if value in myBag:
  print( "The bag contains the value", value )
else :
  print( "The bag does not contain the value", value )
```

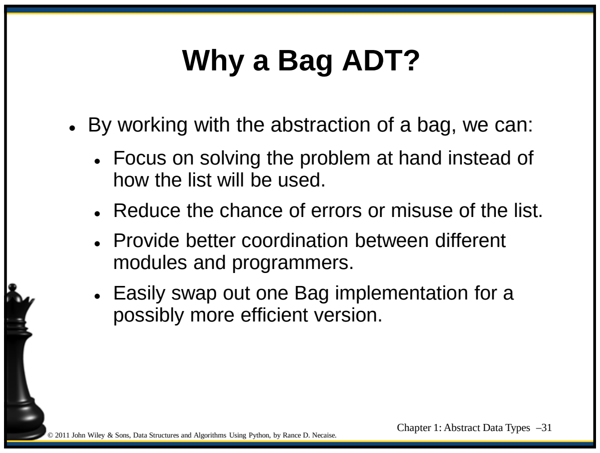
Bag: Example 2

checkdates2.py

```
# Modified version of the checkdates.py program which first
# stores the birth dates and then processes them.

from linearbag import Bag
from date import Date

def main():
  bornBefore = Date( 6, 1, 1988 )
  bag = Bag()

   # Extract dates from the user and place them in the bag.
  date = promptAndExtractDate()
  while date is not None :
    bag.add( date )
    date = promptAndExtractDate()

   # Iterate over the bag and check the age.
  for date in bag :
    if date <= bornBefore :
      print( "Is at least 21 years of age: ", date )
```

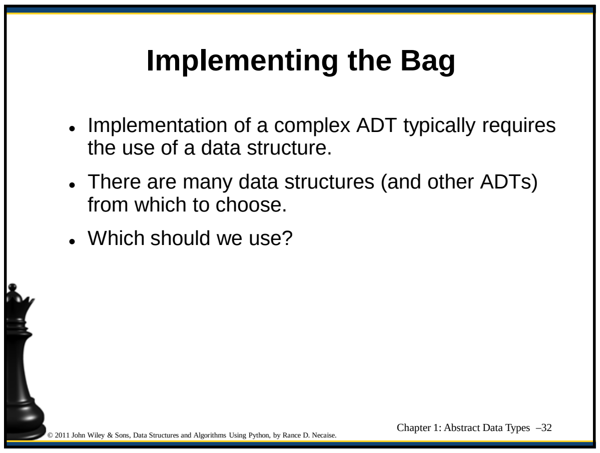
Why a Bag ADT?

- Python's list can accomplish the same thing as a Bag ADT.
- So, why do we need a new ADT?
  - For a small program, the use of a list may be appropriate.
  - For large programs the use of new ADTs provide several advantages.



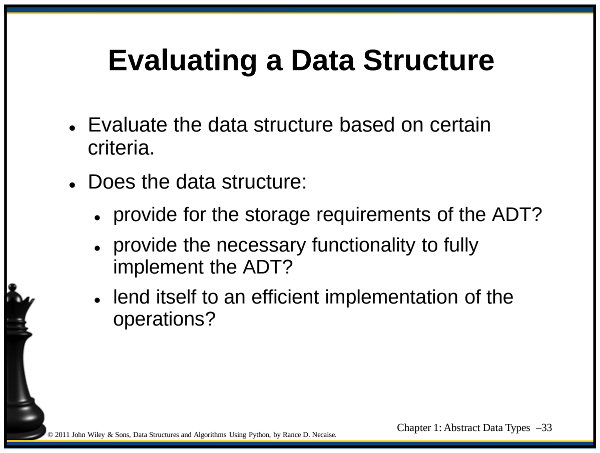
Why a Bag ADT?

- By working with the abstraction of a bag, we can:
  - Focus on solving the problem at hand instead of how the list will be used.
  - Reduce the chance of errors or misuse of the list.
  - Provide better coordination between different modules and programmers.
  - Easily swap out one Bag implementation for a possibly more efficient version.



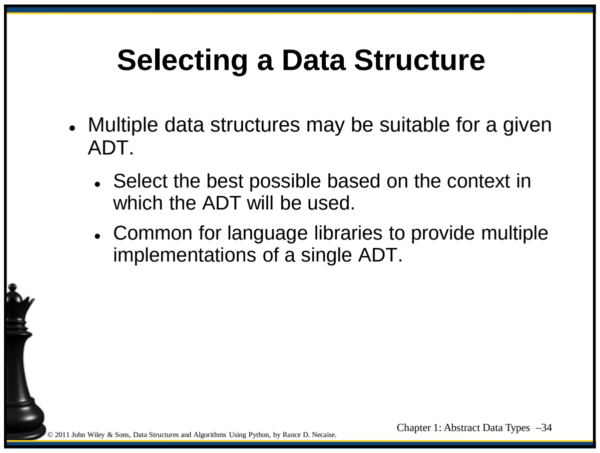
Implementing the Bag

- Implementation of a complex ADT typically requires the use of a data structure.
- There are many data structures (and other ADTs) from which to choose.
- Which should we use?



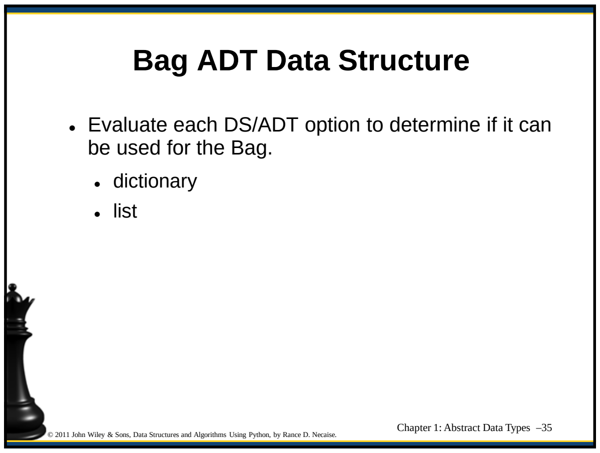
Evaluating a Data Structure

- Evaluate the data structure based on certain criteria.
- Does the data structure:
  - provide for the storage requirements of the ADT?
  - provide the necessary functionality to fully implement the ADT?
  - lend itself to an efficient implementation of the operations?



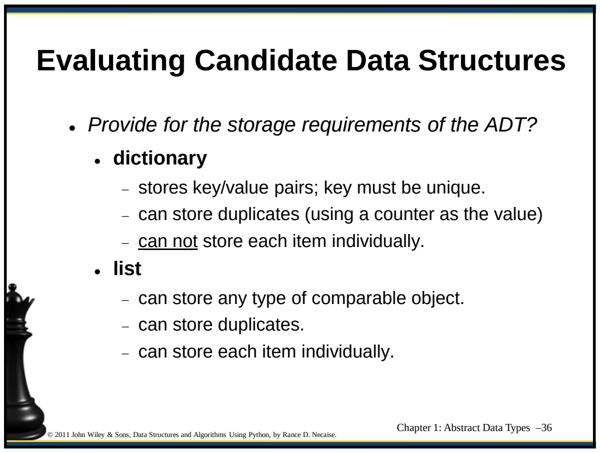
Selecting a Data Structure

- Multiple data structures may be suitable for a given ADT.
  - Select the best possible based on the context in which the ADT will be used.
  - Common for language libraries to provide multiple implementations of a single ADT.



Bag ADT Data Structure

- Evaluate each DS/ADT option to determine if it can be used for the Bag.
  - dictionary
  - list



Evaluating Candidate Data Structures

- *Provide for the storage requirements of the ADT?*
  - **dictionary**
    - stores key/value pairs; key must be unique.
    - can store duplicates (using a counter as the value)
    - <u>can not</u> store each item individually.
  - **list**
    - can store any type of comparable object.
    - can store duplicates.
    - can store each item individually.

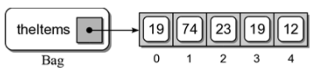
Evaluating Candidate DS

- *Does the list provide the necessary functionality to fully implement the ADT?*
  - Empty bag – empty list
  - Bag size – list size
  - Contains item – use in operator on list.
  - Add item – append() to the list.
  - Remove item – remove() from the list.
  - Traverse items – can access each list elements.

Selecting the List

- The Python list can be used to implement the bag:
  - provides for the storage requirements.
  - provides the necessary functionality.

Sample Bag Instance



Bag: List Implementation

linearbag.py

```
class Bag :
  def __init__( self ):
    self._theItems = list()

  def __len__( self ):
    return len( self._theItems )

  def __contains__( self, item ):
    return item in self._theItems

  def add( self, item ):
    self._theItems.append( item )

  def remove( self, item ):
    assert item in self._theItems, "The item must be in the bag."
    ndx = self._theItems.index( item )
    return self._theItems.pop( ndx )
```

Traversals and Iterators

- Traversals are very common operations performed on containers.
  - Iterates over the entire collection.
  - Provides access to each individual element.
  - Examples:
    - find an item.
    - print entire collection.

Traversals

- We could define specific traversal operations as part of the ADT.

```
class Bag :
# ...
  def saveToFile( self, filename ):
    ......

  def findSmallest( self ):
    ......
```

Generic Traversals

- What about other traversals?
  - Find the largest instead of the smallest?
  - Save to a file in a different format?
- We can not possibly add all traversals to a general container.
  - Need to provide generic traversal.
  - Without requiring access to the implementation.

Python Iterators

- Python provides a built-in iterator mechanism.
  - Create an iterator object.
  - Used with the `for` loop construct .
  - Works for both built-in and user-defined containers.

```
 # Iterate over the bag and check the ages.
for date in bag :
  if date <= bornBefore :
    print( "Is at least 21 years of age: ", date )
```

Designing an Iterator

- **Step 1**: define and implement an iterator class.
  - Class with two special methods.
  - Defined in the same module as the container class.

```
class _BagIterator :
  def __init__( self, theList ):
    self._bagItems = theList
    self._curItem = 0

  def __iter__( self ):
    return self

  def __next__( self ):
    if self._curItem < len( self._bagItems ) :
      item = self._bagItems[ self._curItem ]
      self._curItem += 1
      return item
    else :
      raise StopIteration
```

Designing an Iterator

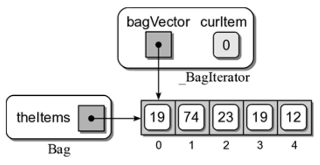
- **Step 2**: define the iterator operator as part of the container class.
  - Creates an instance of the iterator object.
  - Called at the beginning of the `for` loop.

```
class Bag :
# ...
  def __iter__( self ):
    return _BagIterator( self._theItems )
```

Bag Iterator

- An instance of the iterator is automatically created.

```
for item in bag :
  print( item )
```



Student Records

- Suppose we have a collection of student records on external storage.
  - id #
  - first name
  - last name
  - class code
  - GPA
- We need to extract the data and produce a report in a prescribed format.

Sample Report

```
                 LIST OF STUDENTS
ID     NAME                       CLASS       GPA
-----  -------------------------  ----------  ----
10015  Smith, John                Sophomore   3.01
10167  Jones, Wendy               Junior      2.85
10175  Smith, Jane                Senior      3.92
10188  Wales, Sam                 Senior      3.25
10200  Roberts, Sally             Freshman    4.00
10208  Green, Patrick             Freshman    3.95
10226  Nelson, Amy                Sophomore   2.95
10334  Roberts, Jane              Senior      3.81
10387  Taylor, Susan              Sophomore   2.15
10400  Logan, Mark                Junior      3.33
10485  Brown, Jessica             Sophomore   2.91
--------------------------------------------------
Number of students: 11
```

File Format

- We have not been told the type or format of the external storage.
  - What type?
    - plain text file
    - binary file
    - relational database
  - How is the data formatted?

Using Abstraction

- By applying abstraction to this problem, we can begin designing a solution.
- No matter the source, the record extraction will be similar:
  - open the a connection
  - extract the records
  - close the connection

Student File Reader ADT

- A *student file reader* extracts student records from external storage.
  - Data components will be stored in an appropriate storage object.

- StudentFileReader(filename)
- open()
- close()
- fetchRecord()
- fetchAll()

Creating the Report

studentreport..py

```
from studentfile import StudentFileReader

# Name of the file to open.
FILE_NAME = "students.txt"

def main():
   # Extract the student records from the given text file.
  reader = StudentFileReader( FILE_NAME )
  reader.open()
  studentList = reader.fetchAll()
  reader.close()

   # Sort the list by id number and print the report.
  sortTheList( studentList )
  printReport( studentList )

def sortTheList( theList ):
  ......

def printReport( theList ):
  ......

main()
```

Student Records

- Tuples should not be used for structured data.
  - access by subscript.
- Use storage objects created from a storage class.
  - access by named data fields.

```
class StudentRecord :
  def __init__( self ):
    self.idNum = 0
    self.firstName = None
    self.lastName = None
    self.classCode = 0
    self.gpa = 0.0
```

Storage Class

- Regular class with only a constructor.
  - creates and initializes data fields
  - data fields are public
- Internal storage classes should be private.
  - name starts with an underscore.
- `StudentRecord` objects needed outside the ADT.
- Defined within the module where they are used.

Sort The List

studentreport..py

```
def sortTheList( theList ):
  theList.sort( key = lambda rec: rec.idNum )

# Each object is passed to the lambda expression
# which returns the idNum field of the object.
```

Print the Report

studentreport..py

```
def printReport( theList ):
   # The class names associated with the class codes.
  classNames = ( None, "Freshman", "Sophomore",
                 "Junior", "Senior" )
   # Print the header.
  print( "LIST OF STUDENTS".center(50) )
  print( "" )
  print( "%-5s  %-25s  %-10s  %-4s" % \
           ('ID', 'NAME', 'CLASS', 'GPA'))
  print( "%5s  %25s  %10s  %4s" % \
           ('-' * 5, '-' * 25, '-' * 10, '-' * 4))
   # Print the body.
  for record in theList :
    print( "%5d  %-25s  %-10s  %4.2f" % \
                 (record.idNum, \
                  record.lastName + ', ' + record.firstName,
                  classNames[record.classCode], record.gpa) )
   # Add a footer.
  print( "-" * 50 )
  print( "Number of students:", len(theList) )
```

ADT Implementation

- Does not require a data structure.
- Implemented based on the storage type/ format.
- Assume a text file with the format:

```
10015
John
Smith
2
3.01
10334
Jane
Roberts
4
3.81
  :
  :
```

DIA Software

- It's free!
- Download from http://dia-installer.de/
- Use the UML shapes.

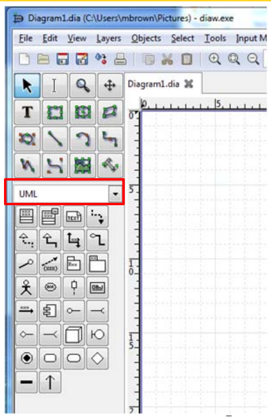


Diagram1.dia (C:\Users\mbrown\Pictures) - diaw.exe
File Edit View Layers Objects Select Tools Input M
Diagram1.dia
UML