Abstract Data Types

CS 234, Fall 2017

• Types, Data Types
• Abstraction
• Abstract Data Types
• Preconditions, Postconditions
• ADT Examples
Data Types

• Data is stored in a computer as a sequence of binary digits: 1’s and 0’s
  • Number 8 : 00001000
  • Number 5 : 00000101

• What exactly does the following sequence represent?
  00000001 01011100 10111101 01011100 01110001 01110000 10011111 11111100

  Answer: it depends
  • Could be some characters, an integer, a float
  • You (or the computer) cannot tell just by looking at it
  • Need something to tell us how to interpret the data
  • That “something” is ...
Data Types

• **Type**
  • a collection of values
  • ex. numeric

• **Data Type**
  • a given type along with a collection of operations
  • how to interpret the data
  • ex. integers

Programming languages commonly provide data types as part of the language itself.
Data Types

- Data types can be divided into two groups:
  - **simple**
    - consists of single values
    - ex: integers, floating-points
  - **complex**
    - multiple components
    - ex: lists, tuples, strings
Data Types

• Data types can also be characterized by their definition:
  • **primitive types**
    • provided by the language itself.
    • ex: int, float, list, string
  • **user-defined types**
    • defined by the programmer as needed.
    • class definitions create new data types.
    • ex: social security number, student record
How do humans cope with complexity in everyday life?

• Humans deal with complexity by abstracting details away
  • e.g., surfing the internet, users don’t require knowledge of internal 
    processors registers; sufficient to think of a computer as simple visualization 
    tool.

• To be useful, an abstraction (model) must be smaller than what it 
  represents
  • e.g., road map vs photographs of terrain vs physical model
Example

• Q: How does the Python function `math.cos(x)` actually calculate the cosine of the value x?

The following example shows the usage of `cos()` method.

```python
#!/usr/bin/python
import math
print "cos(3) : ", math.cos(3)
print "cos(-3) : ", math.cos(-3)
print "cos(0) : ", math.cos(0)
```

When we run above program, it produces following result:

- `cos(3) : -0.9899924966`
- `cos(-3) : -0.9899924966`
- `cos(0) : 1.0`

• Almost all Python programmers just need to know what these functions do, not how they are implemented.
Abstractions

- Used by computer scientists to help manage complex problems.
  - **abstraction**
    - a mechanism for *separating* the properties of an object, and
    - restricting the focus to those *relevant* in the current context.
  - Focus on the “*what*” not the “*how*”.

- *Abstraction* means *capture the relevant (or important) aspects*
Abstractions

- Common types in computer science:
  - **functional abstraction**
    - use of a function/method knowing what it does but not how it's done.
    - ex: \( y = \sqrt{x} \)
  - **data abstraction**
    - separate the values/operations from the implementation of a data type.
    - ex: big integers in Python
Abstract Data Type

- A **programmer-defined** data type specified by:
  - a domain or set of **data values**
  - a collection of well-defined **operations**

- Defined independent of its implementation.
Information Hiding

User programs interact with ADTs through their interface or set of operations.

The implementation details are hidden as if inside a black box.

string ADT
- str()
- upper()
- lower()
Types of Operations

- ADT operations can be grouped into four categories:
  - constructors
  - accessors
  - mutators
  - iterators
Advantages

Several advantages of working with ADTs:

- Focus on solving the problem at hand. Focus on functionality, instead of implementation.
- Help to reduce logical errors from misuse of the data type by limiting controlling the use of implementation details.
- Can easily change the implementation w/o affecting the use of the ADT.
- Easier to manage and divide larger problems into smaller parts.
ADT Implementation

- Abstractions make problem solving easier.
- Programs require concrete implementations in order to execute.
  - language library ADTs
    - implemented by the library maintainers.
  - your own ADTs
    - implemented by you.
Data Structure

- Complex abstract data types are implemented using a **data structure**.
  - Physical representation of how data is stored, organized, and manipulated.
  - Store a collection of values.
Data Structure

• Many common data structures
  • arrays, linked lists, stacks, queues, trees

• Differ in:
  • data organization
  • available operations

• Choice of data structure depends on:
  • the specific ADT
  • the problem being solved.
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

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bag()</td>
</tr>
<tr>
<td>length()</td>
</tr>
<tr>
<td>contains( item )</td>
</tr>
<tr>
<td>add( item )</td>
</tr>
<tr>
<td>remove( item )</td>
</tr>
<tr>
<td>iterator()</td>
</tr>
</tbody>
</table>
Bag: Example

# 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 )
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.
Using the ADT

- Given the definition, we can use the ADT without knowing how it's implemented.
- Reinforces the use of abstraction:
  - by focusing on what functionality is provided
  - instead of how that functionality is implemented.
Creating an ADT in Python

• Create a new class
• Describe the data values (types) as a comment
• Describe the operations (the functions of the class). For each function, 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
- Usually focus on the **inputs** to a function
  - 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.
Precondition

• Example
  • Integer division ( a // b )
  • math.sqrt(x)
• Typical preconditions involve checking ..
  • the type (is it a float) and raising a TypeError exception if it is not
  • the value (is it non-zero) and raising a ValueError exception if it is not
• Users prefer few preconditions, implementers prefer lots of them.
Postcondition

- Result or state of the ADT instance after the operation is performed.
- Usually focus on the **output** of a function and any **side effect** (such as print which has the side effect of printing to the screen or a file)
  - Will be true if the preconditions are met.
    - given: \( x . \text{pop}(i) \)
    - the \( i^{\text{th}} \) item will be removed if \( i \) is a valid index.
- Users prefer lost of postconditions, implementers prefer few of them.
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

- Raise exceptions when errors occur.
  - An event that can be triggered by the program.
  - Optionally handled during execution.

- Example:

```python
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.
  
  ```python
  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.
What is Missing from an ADT

• Can't run this code
• Need an implementation of the ADT
  • Follow function descriptions,
  • Assume or check preconditions
  • Guarantee postconditions
  • Include needed fields
• Add helper functions as needed (functions not mentioned in interface but are helpful in implementing the ADT)
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 DS

- 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)
    - cannot 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

```python
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)
```

linearbag.py
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.

    ```python
    self._theItems = list()
    ```
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._isValidType(self, item)
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.

```python
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.

```python
# Iterate over the bag and check the items.
for num in bag:
    if num <= upperbound:
        print(num, "is not the largest number")
```
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.

```python
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.

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

- An instance of the iterator is automatically created.

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