CSCI 204: Data Structures & Algorithms

Abstract Data Type

Consider some examples …

- The Date class we saw in CSCI 203
  - d = Date()
  - tomorrow = d.next_day() gives a new date
  - d.is_week_day() gives True of False
- A GradeBook class
  - b = GradeBook(student_list, course_list)
  - print(b.get_grade('Sam Brown', 'CSCI 204'))

What's in common?

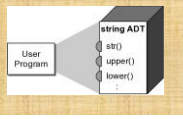
- In both cases, we (the application programs) just wanted to use the pre-built class (Date or GradeBook).
- We don't care how Date or GradeBook was implemented.
- We are not supposed to visit or change the implementation of Date or GradeBook classes.
- Using ADT makes our lives much easier!
- We don't need to re-invent the wheels!

Abstract Data Type

- An abstract data type (ADT) is a collection of data and a set of operations on the data.
- An ADT has the following features.
  - **Information Hiding**: It hides implementation details from the users. That is, it presents what the ADT does, not how it does.
  - It provides an **interface** that other programs can use to access the functionality of the ADT.

Information Hiding

- ADTs can be viewed as black boxes:
  - functionality is provided through an **interface**.
    - Matrix in the coming lab!
  - implementation details are hidden inside the box.



Types of Operations

- ADT operations can be grouped into four categories:
  - **constructors** – creates the ADT
  - **accessors** – gets information
  - **mutators** – changes information
  - **iterators** – navigates through it

What does information hiding look like?

- Date example (date.py)
- Counter example (test_stop_counter.py)
- Inventory example (test_inventory.py)
- You will be working on Matrix in the coming lab

Using the ADT

- 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.

Defining Operations

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

Preconditions

- 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 i^{th} 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 because the state of the object is not changed.
  - 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:

```
my_list = [ 12, 50, 5, 17 ]
print( my_list[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.

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.

Exercise: Build a Bag

Think of this ADT like a shopping cart. Items can be added to it. Items can also be removed from it. However, there is no specific order to them.

Operations:

- **add**: which adds an item to the bag
- **remove**: which removes an item from the bag
- **contains**: which checks if an item is in the bag
- **iterator**: which traverses over the items in the bag one at a time

```
class Bag:

    def __init__ ( self ):
        """ Create an empty Bag """
        pass

    def add(self , item ):
        """ Adds an item to the bag. What if it does not fit? """
        pass

    def remove (self , item ):
        """ removes an item from the bag and returns it.
        What to do if its not in there ? return None ? Exception ? """
        pass

    def contains (self , item ):
        """ checks if an item is in the bag , returns True or False """
        pass

    def iterator ( self ):
        """ returns an iterator for the bag """
        pass
```

```
from bag_base import Bag

items = ['hello', 'world', 123, True, 'How are you?', 234, 567]

my_bag = Bag()

print('Test bag_base ...')
#==============================
# The followings test bag_base
for x in items:
    my_bag.add(x)
    print('Added .. ', x)

print('Test contains ...')
print( 'Test my_bag.contains("hello") True : ', my_bag.contains('hello'))
print( 'Test my_bag.contains(True) True : ', my_bag.contains(True))
print( 'Test my_bag.contains(False) False : ', my_bag.contains(False))
print( 'Test my_bag.contains(567) True : ', my_bag.contains(567))
print( 'Test my_bag.contains(5678) False : ', my_bag.contains(5678))

for x in items:
    my_bag.remove(x)
    print('Removing ...', x)

print('Count of items in the bag : ', len(my_bag))
```