Abstract Data Type

Lesson 1

Textbook

- Data Structures and Algorithms Using Python, by Rance Necaise, Wiley, 2011.
  - https://tinyurl.com/necaisebook

Introduction

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

Data Types

- Data is stored in a computer as a sequence of binary digits: 1's and 0's
  - 0000 1000
  - 0000 0101
- How about this:
  - 00000001 01011100 10111101 01011100 01110001 01111000 10011111 11111100
- It depends:
  - Could be some characters, an integer, a float
  - You (or even the computer) cannot tell just by looking at it
  - Need something to tell us how to interprete the data

Data Types

- Type
  - A collection of values
- Data Type
  - A given type along with a collection of operations
  - Tells us how to interprete the data

Data Types

- Simple
  - Consisting of single values
  - Integers, floating-points
- Complex
  - Multiple components
  - E.g. lists, tuples, strings, dictionaries

Data Types

- Data types can also be characterized by their definition:
- Primitive types
  - Provided by the language itself
  - E.g. int, float, list, string, dictionary
- User-defined types
  - Defined by the programmer as needed
  - Class definitions create new data types
  - E.g. Student record

Managing complexity

- How do humans cope with complexity
  - Abstract away details
    - Surfing the internet – don’t require users to know anything about the internet protocol, how the system interpretes the data packets received, and how the browser renders the page.
  - Abstraction (Computer Science)
    - Separation of the properties of an object, and
    - Restricting the focus to those relevant in the current context
    - Focus on “what" and not the “how”

Abstractions

- Common types in Computer Science
  - Functional abstraction
    - Use of a function/method knowing what it is supposed to do, but not how it is done
      - e.g `y = sqrt(x)`
  - Data abstraction
    - Separate the values/operations from the implementation of a data type
      - e.g. Floating-point numbers in computers

Abstract Data Type = Black-boxes

- ADT operations
  - Constructors: Create and initialize new instances of the ADT.
  - Accessors: Return data contained in an instance without modifying it.
  - Mutators: Modify the contents of an ADT instance.
  - Iterators: Process individual data components sequentially.

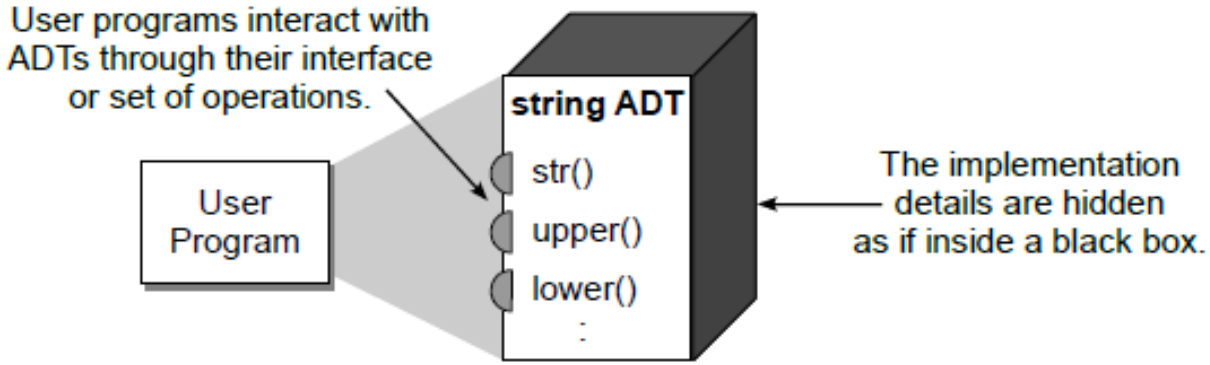

Figure 1.2: Separating the ADT definition from its implementation.

Advantages

- We can focus on solving the problem at hand instead of getting bogged down in the implementation details.
- We can reduce logical errors that can occur from accidental misuse of storage structures and data types by preventing direct access to the implementation.
- The implementation of the abstract data type can be changed without having to modify the program code that uses the ADT.
- It's easier to manage and divide larger programs into smaller modules, allowing different members of a team to work on the separate modules.

Date ADT

A *date* represents a single day in the proleptic Gregorian calendar in which the first day starts on November 24, 4713 BC.

constructor

- `Date(month, day, year)`: Creates a new `Date` instance initialized to the given Gregorian date which must be valid. Year 1 BC and earlier are indicated by negative year components.

accessors

- `day()`: Returns the Gregorian day number of this date.
- `month()`: Returns the Gregorian month number of this date.
- `year()`: Returns the Gregorian year of this date.
- `monthName()`: Returns the Gregorian month name of this date.
- `dayOfWeek()`: Returns the day of the week as a number between 0 and 6 with 0 representing Monday and 6 representing Sunday.
- `numDays(otherDate)`: Returns the number of days as a positive integer between this date and the `otherDate`.
- `isLeapYear()`: Determines if this date falls in a leap year and returns the appropriate boolean value.

mutator

- `advanceBy(days)`: Advances the date by the given number of days. The date is incremented if `days` is positive and decremented if `days` is negative. The date is capped to November 24, 4714 BC, if necessary.

- *`comparable`*`(otherDate)`: Compares this date to the `otherDate` to determine their logical ordering. This comparison can be done using any of the logical operators `<`, `<=`, `>`, `>=`, `==`, `!=`.
- *`toString`*`()`: Returns a string representing the Gregorian date in the format `mm/dd/yyyy`. Implemented as the Python operator that is automatically called via the `str()` constructor.

Sample usage of Date ADT

```
# Extracts a collection of birth dates from the user and determines
# if each individual is at least 21 years of age.
from date import Date

def main():
   # Date before which a person must have been born to be 21 or older.
  bornBefore = Date(6, 1, 1988)

   # Extract birth dates from the user and determine if 21 or older.
  date = promptAndExtractDate()
  while date is not None :
    if date <= bornBefore :
      print( "Is at least 21 years of age: ", date )
    date = promptAndExtractDate()

# Prompts for and extracts the Gregorian date components. Returns a
# Date object or None when the user has finished entering dates.
def promptAndExtractDate():
  print( "Enter a birth date." )
  month = int( input("month (0 to quit): ") )
  if month == 0 :
    return None
  else :
    day = int( input("day: ") )
    year = int( input("year: ") )
    return Date( month, day, year )

# Call the main routine.
main()
```

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), where x is a list
  - the ith item will be removed if i is a valid index.
- 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 handed 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 = 1 January, 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

```
class Date :
   # Creates an object instance for the specified Gregorian date.
  def __init__( self, month, day, year ):
    self._julianDay = 0
    assert self._isValidGregorian( month, day, year ), \
           "Invalid Gregorian date."

     # The first line of the equation, T = (M - 14) / 12, has to be changed
     # since Python's implementation of integer division is not the same
     # as the mathematical definition.
    tmp = 0
    if month < 3 :
      tmp = -1
    self._julianDay = day - 32075 + \
              (1461 * (year + 4800 + tmp) // 4) + \
              (367 * (month - 2 - tmp * 12) // 12) - \
              (3 * ((year + 4900 + tmp) // 100) // 4)
```

private Members

- Python does not provide for a technique to protect attributes and methods from direct access.
  - We use identifiers beginning with an underscore. [reference]
  - 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.

Exercise: Comparable Date

- Need only implement 3 of the 6 comparable operators.
  - d1 == d2: Implement __eq__()
  - d1 < d2: Implement __lt__()
  - d1 <= d2: Implement __le__()
- Python 3 will
  - swap operands when >, >= is used
  - Invert result when != is used

Homework

- Finish reading Chapter 1
- Do these **Exercises** (pg 28 & 29):
  - 1.1, 1.3, 1.4 (pg 28 & 29)
  - 1.10 (pg 32)
- Read Chapter 2.1 and 2.2 (pg 33-46) for next lesson

Summary

Abstract Data Type allows for the programmer to:

- Manage complexity – black box
- Allows us to change implementation (for efficiency) without affect the client code
- Allows for division of labour