ABSTRACT DATA TYPES 5

DATA C88C

February 19, 2024

1 Abstract Data Types

1.1 Data Abstraction

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects — for example, car objects, chair objects, people objects, etc. That way, programmers don't have to worry about *how* code is implemented — they just have to know *what* it does.

Data abstraction mimics how we think about the world. For example, when you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of. You just have to know how to turn the wheel and press the gas pedal.

An **abstract data type** consists of two types of functions:

- Constructors: functions that build the abstract data type.
- Selectors: functions that retrieve information from the data type.

For example, say we have an abstract data type called `city`. This `city` object will hold the `city`'s name, and its latitude and longitude. To create a `city` object, you'd use a constructor like

```
city = make_city(name, lat, lon)
```

To extract the information of a `city` object, you would use the selectors like

```
get_name(city)
get_lat(city)
get_lon(city)
```

For example, here is how we would use the `make_city` constructor to create a city object to represent Berkeley and the selectors to access its information.

```
>>> berkeley = make_city('Berkeley', 122, 37)
>>> get_name(berkeley)
'Berkeley'
>>> get_lat(berkeley)
122
>>> get_lon(berkeley)
37
```

The following code will compute the distance between two city objects:

```
from math import sqrt
def distance(city_1, city_2):

    lat_1, lon_1 = get_lat(city_1), get_lon(city_1)
    lat_2, lon_2 = get_lat(city_2), get_lon(city_2)

    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

Notice that we don't need to know how these functions were implemented. We are assuming that someone else has defined them for us.

It's okay if the end user doesn't know how functions were implemented. However, the functions still have to be defined by someone. We'll look into defining the constructors and selectors later in this discussion.

1.2 Abstraction Violations

Notice how we did not need to know how the constructors and selectors in the previous section were implemented in order to use them. This is what we mean by the *implementation* and *use* of an abstract data type being separate. In fact, you should never assume anything about how the constructors and selectors for an abstract data type are implemented. Doing so is called a **data abstraction violation**.

As an example, here is one implementation for the `rational` constructor.

```
def rational(n, d):
    return [n, d]
```

Given this constructor, the following would be considered a data abstraction violation:

```
>>> frac1 = rational(3, 4)
>>> frac2 = rational(5, 6)
>>> frac1[0] * frac2[0]
15
```

This is because we assumed rationals were represented as lists instead of accessing their elements using the selectors.

1.3 Questions

1. The CS 88 TAs have decided to call upon the power of data abstraction to organize their discussion sections. To do so, they've created a `discussion` abstract data type. A discussion contains three things:

   - The name of the TA running the section
   - The time the section starts, given as an integer
   - A list of students enrolled in the section

   Given this, the TAs come up with the following constructor and selectors:

   - `make_discussion(ta, time, students)`: Creates and returns a new discussion section.
   - `get_ta(disc)`: Returns the TA running the given discussion section.
   - `get_time(disc)`: Returns the start time of the given discussion section.
   - `get_students(disc)`: Returns the list of students enrolled in the given discussion section.

   The TAs have decided to reveal the implementation of the discussion section ADT. Use these function definitions to answer the next two questions:

```
def make_discussion(ta, time, students):
    return [ta, time, students]

def get_ta(disc):
    return disc[0]

def get_time(disc):
    return disc[1]

def get_students(disc):
    return disc[2]
```

2. Implement `add_student`, which takes in a discussion section and a string representing a student's name, and returns a new discussion with the new student added to the roster. The list of students for the new discussion should be a new list. Remember to use the constructor and selectors!

```
def add_student(disc, student):
    """ Adds a student to this discussion.
    >>> disc = make_discussion("Alex", 4, ["Srinath", "Brian
       "])
    >>> new_disc = add_student(disc, "Sophia")
    >>> get_students(new_disc)
    ["Srinath", "Brian", "Sophia"]
    >>> get_students(disc)
    ["Srinath", "Brian"]
    """
```

Solution:

```
    new_students = get_students(disc) + [student]
    ta = get_ta(disc)
    time = get_time(disc)
    return make_discussion(ta, time, new_students)
```

3. The TAs have written the following code using the above data abstraction. However, it contains some abstraction violations. Underline each occurence of an abstraction violation. Then, if possible, write the correct line of code to the right.

```
def check_start(disc1, disc2):
    """Checks whether disc1 and disc2 have the same starting
       time."""
    return disc1[1] == disc2[1]

def print_students(disc):
    """Prints the name of each student in the discussion."""
    for student in disc[2]:
        print(student)

def print_duplicates(disc1, disc2):
     """Prints each student that attended both disc1 and disc2
        ."""
    students_1 = get_students(disc1)
    students_2 = get_students(disc2)
    for i in range(len(students_1)):
        if students_1[i] in students_2:
            print(students_1[i])
```

Solution: Below is the code with all abstraction violations removed. Notice that `print_duplicates` did not contain any data abstraction violations.

```
def check_start(disc1, disc2):
    return get_time(disc1) == get_time(disc2)

def print_students(disc):
    for student in get_students(disc):
        print(student)

def print_duplicates(disc1, disc2):
    students_1, students_2 = get_students(disc1),
       get_students(disc2)
    for i in range(len(students_1)):
        if students_1[i] in students_2:
            print(students_1[i])
```

4. A disgruntled student makes changes to the discussion data abstraction in an attempt to disrupt the TAs' ability to run section. The new implementation is as follows:

```
def make_discussion(ta, time, students):
    return {"ta" : ta, "time" : time,  "students" : students}

def get_ta(disc):
    return disc["ta"]

def get_time(disc):
    return disc["time"]

def get_students(disc):
    return disc["students"]
```

Would the code in the previous question, with the corrections you made, still work with these changes? Would the code before removing abstraction violations still work?

Solution: After removing the abstraction violations, the code will work correctly. This is because we don't assume anything about the representation of a discussion object, so changing the representation doesn't affect anything.

Before, with abstraction violations, our code will no longer work correctly. When we try to index into a discussion as if it is a list, we will get an error, since it is now implemented as a dictionary.

5. Consider the City ADT (defined on the page 1 of the discussion worksheet). Write a function `in_range(host_city, new_city, diff)` that checks whether an input `new_city` is within `diff` distance of the `host_city`, inclusive.

   Hint: We have already defined a distance function for you on page 2!

```
def in_range(host_city, new_city, diff):
    """
    >>> host_city = make_city('Host City', 10, 20)
    >>> city1 = make_city('City 1', 5, 10)
    >>> in_range(host_city, city1, 5)
    False # the distance between host_city and city1 is > 5
    >>> in_range(host_city, city1, 100)
    True # the distance between host_city and city1 is <= 100
    """
```

Solution:

```
    dist = distance(host_city, new_city)
    return distance <= diff
```