Mutability, Abstract Data Types

Discussion 4: September 22, 2025

Abstract Data Types

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.

Q1: Word

In this problem, we will implement two data abstractions from scratch to represent a language! Let's first build an abstraction for words, which will compose each language.

The `word` abstraction stores: 1. a word's `name` 2. a word's `definition` in English

Your first job will be to create the *constructors* and *selectors* for the `word` type *in two ways* (denoted by `a` and `b`).

- A clean, typical approach would be to use a `list` pair to bundle together attributes of our type. Can you come up with two other implementations of the word ADT? Some ideas include using a dictionary or higher-order functions!

  Note: Concerning ourselves with the implementation of this ADT does put us *"beneath" the abstraction barrier*. However, a takeaway from this problem is that as we move towards higher-level functionalities (like translation), we no longer have to worry about the specifics of our original implementation. These low-level details are *abstracted away* by our constructors and selectors!

```
#Implementation a
def make_word_a(name, definition):
    """
    >>> yes = make_word_a('yes', 'affirmative response')
    >>> get_word_name_a(yes), get_word_definition_a(yes)
    ('yes', 'affirmative response')
    """
    def select(item):
        if item == 'name':
            return name
        elif item == 'definition':
            return definition
    return select

def get_word_name_a(word):
    return word('name')

def get_word_definition_a(word):
    return word('definition')


#Implementation b
def make_word_b(name, definition):
    """
    >>> yes = make_word_b('yes', 'affirmative response')
    >>> get_word_name_b(yes), get_word_definition_b(yes)
    ('yes', 'affirmative response')
    """
    return {'a': name, 'b': definition}

def get_word_name_b(word):
    return word['a']

def get_word_definition_b(word):
    return word['b']


def check_word_abstraction():
    """Check whether both constructors and selectors work properly.
    >>> yes_a = make_word_a('yes', 'affirmative response')
    >>> yes_b = make_word_b('yes', 'affirmative response')
    >>> get_word_name_a(yes_a) == get_word_name_b(yes_b)
    True
    >>> get_word_definition_a(yes_a) == get_word_definition_b(yes_b)
    True
    >>> # Errors (can't mix implementations): get_word_name_a(yes_b)
    """
```

Q2: Language

Now, implement an abstraction for language. The `language` ADT stores:

1. a language's `name`
2. a list of `word`'s present in the language

Finally, using both the `word` and `language` ADTs, implement the two following functions:

- `translate`, which takes `source_word`, a `word` abstraction, and returns the `name` of the translated word in the `target` language. If the word cannot be found, return `'Undefined'`.
- `update`, which updates a language's list of words with `new_word`.

  Hint: When we translate a word to a different language, what attribute of the word remains the same? From there, it's just about searching for and accessing that attribute at the right location!

  Hint: With abstract data types, we can construct ("create") and select ("read") data, but we can't exactly mutate them. To "update" a language, we should instead construct and return a new language altogether!

```
# Paste your word ADT here!
def make_word(name, definition):
    """
    >>> yes = make_word('yes', 'affirmative response')
    >>> get_word_name(yes), get_word_definition(yes)
    ('yes', 'affirmative response')
    """
    return [name, definition]

def get_word_name(word):
    return word[0]

def get_word_definition(word):
    return word[1]

# Implement the language ADT here!
def make_language(name, words):
    """
    >>> hi, there = make_word('hi', 'friendly greeting'), make_word('there', 'at that
    place')
    >>> english = make_language('english', [hi, there])
    >>> get_language_name(english)
    'english'
    >>> for w in get_language_words(english):
    ...     print(get_word_name(w))
    ...
    hi
    there
    """
    return [name, words]

def get_language_name(language):
    return language[0]

def get_language_words(language):
    return language[1]

def translate(source_word, target):
    """Given a word, translate it into the 'target' language.

    >>> hi_def, there_def = 'friendly greeting', 'at that place'
    >>> hi, there = make_word('hi', hi_def), make_word('there', there_def)
    >>> hiss = make_word('hiss', hi_def)
    >>> english, parseltongue = make_language('english', [hi, there]), make_language('
    parseltongue', [hiss])
    >>> translate(hi, parseltongue)
    'hiss'
    """
    source_def = get_word_definition(source_word)
    translated = [w for w in get_language_words(target) if get_word_definition(w) ==
    source_def]
    if translated:
```

Mutability

Q3: Shuffle

Define a function `shuffle` that takes a sequence with an even number of elements (cards) and creates a new list that interleaves the elements of the first half with the elements of the second half.

To interleave two sequences `s0` and `s1` is to create a new sequence such that the new sequence contains (in this order) the first element of `s0`, the first element of `s1`, the second element of `s0`, the second element of `s1`, and so on.

> **Note:** If you're running into an issue where the special heart / diamond / spades / clubs symbols are erroring in the doctests, feel free to copy paste the below doctests into your file as these don't use the special characters and should not give an "illegal multibyte sequence" error.

```
def shuffle(s):
    """Return a shuffled list that interleaves the two halves of s.

    >>> shuffle(range(6))
    [0, 3, 1, 4, 2, 5]
    >>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
    >>> shuffle(letters)
    ['a', 'e', 'b', 'f', 'c', 'g', 'd', 'h']
    >>> shuffle(shuffle(letters))
    ['a', 'c', 'e', 'g', 'b', 'd', 'f', 'h']
    >>> letters  # Original list should not be modified
    ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
    """
    assert len(s) % 2 == 0, 'len(seq) must be even'
    half = len(s) // 2
    shuffled = []
    for i in range(half):
        shuffled.append(s[i])
        shuffled.append(s[half + i])
    return shuffled
```