Abstract Data Types and Mutable Data

Computer Science Mentors CS 88

March 1st to 5th

1 Conceptual Start

1. What are the two types of functions necessary to make an Abstract Data Type? What do they do?

   constructors & selectors

2. Assume that **rational**, **numer**, **denom**, and **gcd** run without error and behave as described below. Can you identify where the abstraction barrier is broken? Come up with a scenario where this code runs without error and a scenario where this code would stop working.

```
def rational(num, den): # Returns a rational number ADT
    #implementation not shown
def numer(x): # Returns the numerator of the given rational
    #implementation not shown
def denom(x): # Returns the denominator of the given rational
    #implementation not shown
def gcd(a, b): # Returns the GCD of two numbers
    #implementation not shown

def simplify(f1): #Simplifies a rational number
    g = gcd(~~f1[0], f1[1]~~)
    return rational(numer(f1) // g, denom(f1) // g)
            numer(f1), denom(f1)
def multiply(f1, f2): # Multiples and simplifies two rationals
    r = rational(numer(f1) * numer(f2), denom(f1) * denom(f2))
    return simplify(r)

x, y = rational(1, 2), rational(2, 3)
multiply(x, y)
```

3. Check your understanding

   1 How do we know when we are breaking an abstraction barrier?

   assume the underlying implementation is

   2 What are the benefits to Data Abstraction?

2 Code Writing

4. The following is an **Abstract Data Type (ADT)** for elephants. Each elephant keeps track of its name, age, and whether or not it can fly. Given our provided constructor, fill out the selectors:

```
def elephant(name, age, can_fly):
    """
    Takes in a string name, an int age, and a boolean can_fly.
    Constructs an elephant with these attributes.
    >>> dumbo = elephant("Dumbo", 10, True)
    >>> elephant_name(dumbo)
    "Dumbo"
    >>> elephant_age(dumbo)
    10
    >>> elephant_can_fly(dumbo)
    True
    """
    return [name, age, can_fly]
```

dumbo

name: "Dumbo"
age: 10
can_fly: True

```
def elephant_name(e):
```

```
def elephant_age(e):
```

```
def elephant_can_fly(e):
```

5. This function returns the correct result, but there's something wrong about its implementation. How do we fix it?

```
def elephant_roster(elephants):
    """
    Takes in a list of elephants and returns a list of their
       names.
    """
    return [elephant[0] for elephant in elephants]
            elephant_name(elephant)
```

6. Fill out the following constructor for the given selectors.

```
def elephant(name, age, can_fly):
```

```
def elephant_name(e):
    return e[0][0]

def elephant_age(e):
    return e[0][1]

def elephant_can_fly(e):
    return e[1]
```

7. How can we write the fixed `elephant_roster` function for the constructors and selectors in the previous question?

8. Fill out the following constructor for the given selectors.

```
def elephant(name, age, can_fly):
    """
    >>> chris = elephant("Chris Martin", 38, False)
    >>> elephant_name(chris)
        "Chris Martin"
    >>> elephant_age(chris)
        38
    >>> elephant_can_fly(chris)
        False
    """
    def select(command):
```

chris

name: "Chris Martin"
age: 38
can_fly: False

```
    return select
def elephant_name(e):
    return e("name")

def elephant_age(e):
    return e("age")

def elephant_can_fly(e):
    return e("can_fly")
```

3 Dictionaries

Dictionaries are containers that **map keys to values**. Let's look at an example:

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
25
>>> pokemon['jolteon'] = 135
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['ditto'] = 25
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148,
'ditto': 25, 'mew': 151}
```

The *keys* of a dictionary must be *immutable* values, such as numbers, strings, tuples, etc. Dictionaries themselves are mutable; we can add, remove, and change entries after creation. Finally, there is only one value per key, however — if we assign a new value to the same key, it overrides any previous value which might have existed. See below for some common uses of dictionaries:

- To add `val` corresponding to `key` *or* to replace the current value of `key` with `val`:
  ```
  dictionary[key] = val
  ```
- To iterate over a dictionary's keys:
  ```
  for key in dictionary: #OR for key in dictionary.keys()
      do_stuff()
  ```
- To iterate over a dictionary's values:
  ```
  for value in dictionary.values():
      do_stuff()
  ```
- To iterate over a dictionary's keys and values:
  ```
  for key, value in dictionary.items():
      do_stuff()
  ```
- To remove an entry in a dictionary:
  ```
  del dictionary[key]
  ```
- To get the value corresponding to `key` and remove the entry:
  ```
  dictionary.pop(key)
  ```

9. Given a list `key` that contains the keys, and another list `values` that contains all the values for a key-value pair. Write a function that returns a dictionary with key-values pairs for each element in the two lists that share the same index. However, if the `values` list is longer than the `keys` list, the subsequent elements in the `values` list will wrap around and replace the key-value pair starting from the beginning.

```
def create_dict(keys, values):
    """
    >>> prompts = ["Movie", "Song", "Food", "Shop"]
    >>> answers = ["Brave", "Yellow", "Steak", "Target"]
    >>> favorites = create_dict(prompts, answers)
    >>> favorites
    {"Movie": "Brave", "Song": "Yellow", "Food": "Steak", "
       Shop": "Target"}
    >>> keys = [0, 1, 2, 3]
    >>> values = ["ice", "cream", "is", "yummy", "vanilla", "
       cake"]
    >>> d = create_dict(keys, values)
    >>> d
    {0: "vanilla", 1: "cake", 2: "is", 3: "yummy"}
```

d

0	~~"ice"~~ "vanilla"
1	~~"cream"~~ "cake"
2	"is"
i = 3	"yummy"

↓

i = 4 % len(keys)

4 % 4

12 % 4

10. Given two dictionaries a and b, mutate a to contain all of the keys-values pairs from b. Note if the value in a is a list, insert the value from b in the end of the list (you may assume the values in b will never be lists).

* matching key-value pairs, which may not be in the same order in a & b

* hint: isinstance(x, list) checks if x is a list

a → {0:0, 1:1, 2:2}

b → {0:1, 1:1}

```
def add_all(a, b):
    """
    >>> a = {x: x for x in range(3)}
    >>> b = {x: 1 for x in range(2)}
    >>> c = {0: "who is tony"}
    >>> add_all(a, b)
    >>> a
    {0: [0, 1], 1: [1, 1], 2: 2}
    >>> add_all(a, c)
    >>> a
    {0: [0, 1, 'who is tony'], 1: [1, 1], 2: 2}
    """
```

ABSTRACT DATA TYPES AND MUTABLE DATA

COMPUTER SCIENCE MENTORS CS 88

March 1st to 5th

1 Conceptual Start

1. What are the two types of functions necessary to make an Abstract Data Type? What do they do?

   constructors - make the ADT
   selectors - return important info stored in an ADT

2. Assume that **rational**, **numer**, **denom**, and **gcd** run without error and behave as described below. Can you identify where the abstraction barrier is broken? Come up with a scenario where this code runs without error and a scenario where this code would stop working.

```
def rational(num, den): # Returns a rational number ADT
    #implementation not shown
def numer(x): # Returns the numerator of the given rational
    #implementation not shown
def denom(x): # Returns the denominator of the given rational
    #implementation not shown
def gcd(a, b): # Returns the GCD of two numbers
    #implementation not shown

def simplify(f1): #Simplifies a rational number
    g = gcd(f1[0], f1[1])
    return rational(numer(f1) // g, denom(f1) // g)

def multiply(f1, f2): # Multiples and simplifies two rationals
    r = rational(numer(f1) * numer(f2), denom(f1) * denom(f2))
    return simplify(r)

x, y = rational(1, 2), rational(2, 3)
multiply(x, y)
```

assumes rational is a data type that uses indices

simplify would work for a list, but not for a dictionary

3. Check your understanding

   1 How do we know when we are breaking an abstraction barrier?

   bypass constructors & selectors, assume implementation details

   2 What are the benefits to Data Abstraction?

   can change implementation without creating a bunch of problems

2 Code Writing

4. The following is an **Abstract Data Type (ADT)** for elephants. Each elephant keeps track of its name, age, and whether or not it can fly. Given our provided constructor, fill out the selectors:

```
def elephant(name, age, can_fly):
    """
    Takes in a string name, an int age, and a boolean can_fly.
    Constructs an elephant with these attributes.
    >>> dumbo = elephant("Dumbo", 10, True)
    >>> elephant_name(dumbo)
    "Dumbo"
    >>> elephant_age(dumbo)
    10
    >>> elephant_can_fly(dumbo)
    True
    """
    return [name, age, can_fly]

def elephant_name(e):
    return e[0]

def elephant_age(e):
    return e[1]
```

```
def elephant_can_fly(e):
```

return e[2]

5. This function returns the correct result, but there's something wrong about its implementation. How do we fix it?

```
def elephant_roster(elephants):
    """
    Takes in a list of elephants and returns a list of their
       names.
    """
```

breaking abstraction barrier!

return [~~elephant[0]~~ **for** elephant **in** elephants]

elephant_name(elephant)

6. Fill out the following constructor for the given selectors.

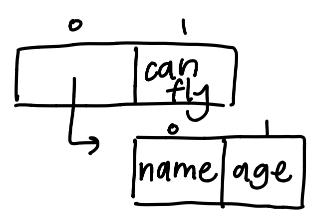
```
def elephant(name, age, can_fly):
```

return [[name, age], can_fly]

```
def elephant_name(e):
    return e[0][0]

def elephant_age(e):
    return e[0][1]

def elephant_can_fly(e):
    return e[1]
```



7. How can we write the fixed `elephant_roster` function for the constructors and selectors in the previous question?

no change! since we're using the selector instead of assuming anything about the underlying implementation, the elephant_roster function will work :)

8. Fill out the following constructor for the given selectors.

```python
def elephant(name, age, can_fly):
    """
    >>> chris = elephant("Chris Martin", 38, False)
    >>> elephant_name(chris)
        "Chris Martin"
    >>> elephant_age(chris)
        38
    >>> elephant_can_fly(chris)
        False
    """
    def select(command):
        if command == "name":
            return name
        elif command == "age":
            return age
        elif command == "can_fly":
            return can_fly

    return select
def elephant_name(e):
    return e("name")

def elephant_age(e):
    return e("age")

def elephant_can_fly(e):
    return e("can_fly")
```

3 Dictionaries

Dictionaries are containers that **map keys to values**. Let's look at an example:

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
25
>>> pokemon['jolteon'] = 135
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['ditto'] = 25
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148,
'ditto': 25, 'mew': 151}
```

The *keys* of a dictionary must be *immutable* values, such as numbers, strings, tuples, etc. Dictionaries themselves are mutable; we can add, remove, and change entries after creation. Finally, there is only one value per key, however — if we assign a new value to the same key, it overrides any previous value which might have existed. See below for some common uses of dictionaries:

- To add `val` corresponding to `key` *or* to replace the current value of `key` with `val`:
  ```
  dictionary[key] = val
  ```
- To iterate over a dictionary's keys:
  ```
  for key in dictionary: #OR for key in dictionary.keys()
      do_stuff()
  ```
- To iterate over a dictionary's values:
  ```
  for value in dictionary.values():
      do_stuff()
  ```
- To iterate over a dictionary's keys and values:
  ```
  for key, value in dictionary.items():
      do_stuff()
  ```
- To remove an entry in a dictionary:
  ```
  del dictionary[key]
  ```
- To get the value corresponding to `key` and remove the entry:
  ```
  dictionary.pop(key)
  ```

9. Given a list `key` that contains the keys, and another list `values` that contains all the values for a key-value pair. Write a function that returns a dictionary with key-values pairs for each element in the two lists that share the same index. However, if the `values` list is longer than the `keys` list, the subsequent elements in the `values` list will wrap around and replace the key-value pair starting from the beginning.

```
def create_dict(keys, values):
    """
    >>> prompts = ["Movie", "Song", "Food", "Shop"]
    >>> answers = ["Brave", "Yellow", "Steak", "Target"]
    >>> favorites = create_dict(prompts, answers)
    >>> favorites
    {"Movie": "Brave", "Song": "Yellow", "Food": "Steak", "
       Shop": "Target"}
    >>> keys = [0, 1, 2, 3]
    >>> values = ["ice", "cream", "is", "yummy", "vanilla", "
       cake"]
    >>> d = create_dict(keys, values)
    >>> d
    {0: "vanilla", 1: "cake", 2: "is", 3: "yummy"}
     d = {}
     num_vals = len(values)
    for i in range(num_vals):
         k = keys[i % num_vals]
         v = values[i]
         d[k] = v
     return d
```

10. Given two dictionaries `a` and `b`, mutate `a` to contain all of the keys-values pairs from `b`. Note if the value in `a` is a list, insert the value from `b` in the end of the list (you may assume the values in `b` will never be lists).

```
def add_all(a, b):
    """
    >>> a = {x: x for x in range(3)}
    >>> b = {x: 1 for x in range(2)}
    >>> c = {0: "who is tony"}
    >>> add_all(a, b)
    >>> a
    {0: [0, 1], 1: [1, 1], 2: 2}
    >>> add_all(a, c)
    >>> a
    {0: [0, 1, 'who is tony'], 1: [1, 1], 2: 2}
    """
```

* matching key-value pairs, which may not be in the same order in a & b

* hint: isinstance(x, list) checks if x is a list

```
    for key in b:
        if key in a:
            if isinstance(a[key], list):
                a[key].append(b[key])
            else:
                a[key] = [a[key], b[key]]
        else:
            a[key] = b[key]
```