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?

> **Solution:** Constructors make the ADT.
> Selectors take instances of the ADT and output relevant information stored in it.

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)
```

> **Solution:** The abstraction barrier is broken inside simplify(f1) when calling gcd(f1[0], f1[1]). This assumes rational returns a type that can be indexed through. i.e. This would work if rational returned a list. However, this would not work if rational returned a dictionary.
>
> The correct implementation of simplify would be
>
> ```
> def simplify(f1):
>     g = gcd(numer(f1), denom(f1))
>     return rational(numer(f1) // g, denom(f1) // g)
> ```

3. Check your understanding

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

   > **Solution:** Put simply, a Data Abstraction Violation is when you bypass the constructors and selectors for an ADT, and directly use how it's implemented in the rest of your code, thus assuming that its implementation will not change.
   > We cannot assume we know how the ADT is constructed except by using constructors and likewise, we cannot assume we know how to access details of our ADT except through selectors. The details are supposed to be abstracted away by the constructors and selectors. If we bypass the constructors and selectors and access the details directly, any small change to the implementation of our ADT could break our entire program.

   2 What are the benefits to Data Abstraction?

   > **Solution:** With a correct implementation of these data types, it makes for more readable code because:
   > -You can make constructors and selectors have more informative names.
   > -Makes collaboration easier
   > -Other programmers don't have to worry about implementation details.
   > -Prevents error propagation.
   > -Fix errors in a single function rather than all over your program.

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:

```python
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):
```

> **Solution:**
> ```python
> return e[0]
> ```

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

> **Solution:**
> ```python
> return e[1]
> ```

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

> **Solution:**
> ```python
> return e[2]
> ```

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

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

> **Solution:** `elephant[0]` is a Data Abstraction Violation (DAV). We should use a selector instead. The corrected function looks like:
> ```python
> def elephant_roster(elephants):
>         return [elephant_name(elephant) for elephant in
>             elephants]
> ```

6. Fill out the following constructor for the given selectors.
```python
def elephant(name, age, can_fly):
```

> **Solution:**
> ```python
>     return [[name, age], can_fly]
> ```

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

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

```python
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?

> **Solution:** No change is necessary to fix `elephant_roster` since using the `elephant` selectors "protects" the roster from constructor definition changes.

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):
```

Solution:

```python
        if command == "name":
            return name
        elif command == "age":
            return age
        elif command == "can_fly":
            return can_fly
```

```python
    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"}
```

Solution:
```
def create_dict(keys, values):
    d = {}
    for i in range(len(values)):
        k = keys[i % lens(keys)]
        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).

```python
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}
    """
```

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