TUTORIAL 9

DICTIONARIES AND CLASSES
REMINDER

• Assignment 08 is due Nov 22\textsuperscript{nd} at 10am
REVIEW

- Dictionary
- Classes
  - __init__
  - __repr__
  - __eq__
  - class methods
DICTIONARY

{key1:value1, key2:value2, ...}

- Each element has a **key** (a way to look up info) and a **value** associated with the **key**
- Unordered list (with each element being a **key-value** pair)
- Like a dictionary (word = **key**, definition = **value**)

USEFUL DICTIONARY FUNCTIONS

- \( d[k] \): Get the value of \( k \)
- \( d[k]=v \): Add key-value pair or change value to be \( v \) if \( k \) already exists in \( d \)
- \( d.keys() \): Creates a view of all the keys in \( d \)
- \( d.values() \): Creates a view of all the values in \( d \)
- \( d.pop(k) \): Removes key-value pair of \( k \) from \( d \) and returns the value of \( k \)
- \( k \text{ in } d \): returns True if \( k \) is a key in \( d \)
• Python’s version of structures in Scheme
• Allows related information to be grouped together
• We’ll use `__init__`, `__repr__`, and `__eq__` with the class
• We'll also write new class methods
class name:
    def __init__(self, f1, f2, ...):
        self.field1 = f1
        self.field2 = f2
        ...

• Creates an object of this class:
  x = name(field1_val, field2_val, ...)

• Call the fields by:  x.field1

• Racket’s version:
  (define-struct name (field1_val field2_val ...))
  (name-field1 x)
def __repr__(self):
    return "name: {0},{1},..."\
        .format(self.field1,
                self.field2,...)

• If we try to print a class object, we’d get something like
  <__main__.name instance at 0x12361c0>
• We can print a more informative message using the __repr__ command within the class definition

You can put the class representation into any form you like, so long as you understand what each field is
def __eq__(self, other):
    return isinstance(other, name) and \
    self.field1 == other.field1 and \
    self.field2 == other.field2 and \
    ...

• If two classes have the same field values, it is used to ensure that they return True.
CLASS METHODS

class name:
    def __init__(self, f1, f2, ...):
    def __repr__(self):
    def __eq__(self, other):

def fn(self, ...):
    # Access field values: self.field1, ...
    # fn may update field values, use field values
    # for calculations, print information, or
    # return information
Write a function `list_multiples` that consumes a string `s` and returns a list in **alphabetical order** containing every character in `s` that appears more than once. Use dictionaries.

**Examples:**

```python
list_multiples("abcd") => []
list_multiples("bacaba") => ["a", "b"]
list_multiples("gtddyucaadsa") => ["a", "d"]
```
CQ1: What should `list_multiples("abc")` and `list_multiples("bacaba")` return?

A. `[]` and `"a", "b"` 
B. `"a", "b", "c"` and `"a", "b"`
C. `"a", "b"` and `"a’, "b", "c"`
D. `"a", "b"` and `"b", "a", "c"`
Write a function `xor` that consumes two dictionaries (`d1` and `d2`) and returns a dictionary.

The returned dictionary will contain all the keys that appear in exactly one of `d1` or `d2` (but not both).

The value associated with each key will be the same as the one found in the original dictionary.
EXAMPLES

d1 = {1: 'a', 2: 'b', 3: 'c', 4: 'd'}
d2 = {5: 'e', 6: 'f', 7: 'g', 8: 'h'}
xor(d1, d2) =>
    {1: 'a', 2: 'b', 3: 'c', 4: 'd',
     5: 'e', 6: 'f', 7: 'g', 8: 'h'}

d3 = {5: 'q', 6: 'l', 7: 'c', 8: 'e'}
xor(d2, d3) => {}
The remaining questions will use the following class:

- A **Person** is a class with fields **name**, **parents**, and **children**
  - **name** is a non-empty string representing the person's name
  - **parents** is a list of at most two strings representing the person's parents (if the parents are unknown, the list will be empty)
  - **children** is a list of strings representing the person's children
EXAMPLES OF PERSON OBJECTS:

- george = Person("George", [], ["elizabeth"])  
- elizabeth = Person("Elizabeth", ["george"], ["charles", "andrew", "edward"])  
- philip = Person("Philip", [], ["charles", "andrew", "edward"])  
- charles = Person("Charles", ["elizabeth", "philip"], [])  
- andrew = Person("Andrew", ["philip", "elizabeth"],[])  
- edward = Person("Edward", ["philip", "elizabeth"],[])
Write a function `full_siblings` which consumes two `Person` objects, `p1` and `p2`, and returns `True` if they have the same non-empty set of known parents (i.e. their only parent is the same, or their two parents are the same - though possibly in a different order), and `False` otherwise.

Examples:

```
full_siblings(elizabeth, edward) => False
full_siblings(edward, charles) => True
```
Write a function `no_kids` which consumes a list of Person objects, `lop`, and returns the number of people who have no children. Write the solution using a loop.

Example:

```python
L = [elizabeth, charles, george, andrew]
no_kids(L) => 2
```
Write a class method `is_bio_family` that consumes two lists of `Person` objects, `lop` and `loc`, representing the parents and the children and returns True if the two lists form a biological family. A biological family occurs when there are two `Person` objects in `lop` that have the same children that are found in `loc`. Otherwise, the method returns False.

Example:

```python
Person.is_bio_family([[elizabeth, philip], [charles, edward, andrew]]) => True
```