TUTORIAL 9

DICTIONARIES AND CLASSES
REMINDER

• Assignment 08 is due July 12\textsuperscript{nd} at 10am
• 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 produces the value of k
• k in d -> produce True if k is a key in d
CLASSES

- 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 produce True.
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 produces a list in alphabetical order containing every character in s that appears more than once. Use dictionaries.

Examples:

```
list_multiples("abcd") => []
list_multiples("bacaba") => ["a", "b"]
list_multiples("gtddyucaadsa") => ["a", "d"]
```
QUESTION 2 - XOR

Write a function \texttt{xor} that consumes two dictionaries (\texttt{d1} and \texttt{d2}) and returns a dictionary.

The produced dictionary will contain all the keys that appear in exactly one of \texttt{d1} or \texttt{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) => {}

CLASS DEFINITION FOR PERSON

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 Person objects representing the person's parents (if the parents are unknown, the list will be empty)
  - **children** is a list of Person objects 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 produces `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, `l0p`, and produces the number of people who have no children. Write the solution using a loop.

Example:

\[ L = [elizabeth, charles, george, andrew] \]

\[ no\_kids(L) => 2 \]
Write a class method `new_child` that consumes a `Person` object (`self`), a `string`, `new_name`, representing the name of a newly born child, and information about a second parent (`parent2`). If the second parent is known, `parent2` is of type `Person`. Otherwise, it is `None`. The function does the following:

- Produces a new `Person` object to represent the child
- Updates the ‘children’ field of each parent to include the new child (this is not done if `parent2` is `None`).