ICS-33: In-Lab Programming Exam #2

Name (printed: Last, First): _____ Lab # (1-8): _____

Name (signed: First Last): _____ Seat # (1-46): _____

General Rules

This in-lab programming exam is worth a total of 100 points. It requires you to write a class that overloads some arithmetic and relational operators, as well as other double-underscore methods: e.g. **__init__**, **__len__**, **__getitem__**, **__iter__**, and **__call__**. I will supply a short script for calling these methods and printing their results, so that you can check them visually for correctness. This is followed by calling the **driver** with a more extensive batch self-check file that is similar to the one we will use for grading purposes. You can test each method independently: none will depend on whether any of the other methods that you must write work correctly. Although, sometimes using previously written methods (if you wrote them correctly) can simplify your code in later methods.

You will have approximately 105 minutes to work on the exam, after logging in and setting up your computer (I estimate that taking about 5 minutes). Make sure that you read the instructions carefully and understand the problems before writing/debugging their code: don't rush through this part of the exam. You may write-on/annotate these pages.

We will test your methods only for correctness; each test you pass will improve your grade, and methods that produce no correct results will earn no credit. This means that your methods must define exactly the parameters specified in the descriptions below and must work on all the example arguments; your methods should also work on any other similar arguments. To aid you writing and debugging these methods

1. Write clear, concise, and simple Python code (there are no statement restrictions).
2. Choose good names for local variables.

You do not need to include any comments in your code; but, feel free to add them to aid yourself. We will **not** grade you on appropriate names, comments, or Python idioms: only on correctness. You may also call extra **print** functions in your code or use the **Eclipse Debugger** to help you understand/debug your functions.

You may import and use any functions in the standard Python library and the **goody** and **prompt** modules (which will be included in the project file that you will download). Documentation for Python's standard library and these modules will be available during the exam. I have written all the standard import statements that I needed to write my solution; feel free to include other imports, or change the form of the included imports to be simpler to use.

If you are having problems with the operating system (logging on, accessing the correct folder/files, accessing the Python documentation, submitting your work) or Eclipse (starting it, setting it up for Python, running Python scripts, running the Eclipse debugger, displaying line numbers) please talk to the staff as soon as possible. We are trying to test only your programming ability, so we will help you with these other activities. But, we cannot help you understand, test, or debug your programming errors. I have asked the TAs and Tutors to NOT ANSWER ANY QUESTIONS about the exam itself: read it carefully and look at the test cases for clarification.

You should have a good understanding of the solution to Programming Assignments #1-#2; you should also have a good understanding of the material on Quizzes #1, #3, and #4 and questions 1, 3, 4, and 5 on the Midterm exam. You should know how to read files; create, manipulate, and iterate over **list**s, **tuple**s, **set**s, and dictionaries (**dict** and **defaultdict**) in 3 ways: by **keys**, **values**, and **items**; call the functions **all**, **any**; **min**, **max**, **sum**; **join**, **split**; **sort**,

1 of 5

reversed; **sort**, **sorted**; **enumerate**, **zip**. You may need to use ***args** as a parameter specification: we often use it to define a method that will match any number of positional (non-named) arguments. Note that you can call **min**, **max**, **sort**, and **sorted** with a **key** function (often a simple **lambda**) that determines how to order the values in the iterable argument on which these functions compute. You are free to use or avoid various Python language features (e.g., lambdas and comprehensions): do what is easiest for you, because we are grading only on whether your functions work correctly.

Before submitting your program, ensure that it runs (has no syntax errors) and ensure there are no strange/unneeded **import** statements at the top.

The DictList class

The **DictList** class is defined in the **exam.py** module. The **DictList** class represents a **list** of **dict**, with some keys appearing in more than one **dict**: think of the later **dict**s as representing updates to the earlier ones. For example, if we define either

```
d = DictList(dict(a=1,b=2,c=3), dict(c=13,d=14,e=15), dict(e=25,f=26,g=27))
```

or

```
d = DictList({'a':1, 'b':2, 'c':3}, {'c':13, 'd':14, 'e':15}, {'e':25, 'f':26, 'g':27})
```

the key **'c'** appears in the first and second dictionary; the key **'e'** appears in the second and third dictionary. The other keys (**'a'**, **'b'**, **'d'**, **'f'**, and **'g'**) appear in only a single dictionary. Later, we will overload **__getitem__** so that **d['c'] == 13**, because the last (highest index) dictionary that contains **'c'** as a key associates with the value **13**.

Details

1. Define a class named **DictList** in a module named **exam.py**

2. Define an **__init__** method that has one parameter: it matches **one or more** arguments, where each argument is expected to be a dictionary. See the example above, which creates a **DictList** object with three dictionaries.

   If there are no dictionaries or if any argument is not a dictionary, this method must raise an **AssertionError** exception with an appropiate message. For example, in my code writing **DictList('abc')** raises **AssertionError** with the message **DictList.__init__: 'abc' is not a dictionary**.

   If there are one or more dictionaries as arguments, store them in a **list** in the **same order** as they appear as arguments.

   IMPORTANT: You **must** use the name **self.dl** (lower-case **D** followed by **L**) to store the **list** constructed from the argument dictionaries. The name **self.dl** is used in some tests in the batch self-check file. Store **no** other **self** variables.

3. Define a **__len__** method that returns the number of distinct keys in all the dictionaries in the **DictList**. For example **len(d)** for

   ```
   d = DictList({'a':1, 'b':2, 'c':3}, {'c':13, 'd':14, 'e':15}, {'e':25, 'f':26, 'g':2
   ```

   is **7**, because there are seven distinct keys in **d**: **'a'**, **'b'**, **'c'**, **'d'**, **'e'**, **'f'**, and **'g'**.

4. Define a **__repr__** method that returns a string, which when passed to **eval** returns a newly constructed **DictList** with the same dictionary arguments the **DictList** object **__repr__** was called on.

   The **DictList** example above, might (because the order of the keys/values in each dictionary makes no difference) return the string.

2 of 5

```
"DictList({'a':1, 'c':3, 'b':2}, {'c':13, 'e':15, 'd':14}, {'g':27, 'f':26, 'e':25})"
```

5. Define a **__contains__** method so that **in** returns whether or not its first argument is a key in **any** of the dictionaries in a **DictList**; it returns **True** if such a key is in **any** dictionary and **False** if such a key is not in **any** of the dictionaries.
   Hint: Just iterate through the data structure looking for the specified key; don't create any new data structures.

6. Define a **__getitem__** method so that calling **d[k]** on **DictList d** returns the value associated with the **latest** dictionary (the one with the highest index) in **d**'s **list** that has **k** as a key. If the key is in **no** dictionaries in the **list**, this method must raise the **KeyError** exception with an appropriate message. For example, in the **DictList**

   ```
   d = DictList({'a':1, 'b':2, 'c':3}, {'c':13, 'd':14, 'e':15}, {'e':25, 'f':26, 'g':2
   ```

   - **d['a']** returns **1** (**'a'** is only in the first dictionary)
   - **d['d']** returns **14** (**'d'** is only in the second dictionary)
   - **d['g']** returns **27** (**'g'** is only in the third dictionary)
   - **d['c']** returns **13** (**'c'** appears in the first and second dictionary: it returns the associated value from the second dictionary).
   - **d['e']** returns **25** (**'e'** appears in the second and third dictionary: it returns the associated value from the third dictionary).
   - **d['x']** raises **KeyError** (**'x'** appears in no dictionaries).

7. Define a **__setitem__** method so that executing **d[k] = v** on **DictList d** works as follows:
   - **if k** is in at least one dictionary, **then** change the association of **k** to **v**, only in the last dictionary (highest index) in which **k** is a key; the number of dictionaries in the **list** remains the same
   - **if k** is not in any dictionaries, **then** create a new dictionary at the **end** of the **list** of dictionaries, with only one item: associating **k** with **v** in that dictionary; the number of dictionaries in the **list** increases by one.

   For example, in the **DictList d**

   ```
   d = DictList({'a':1, 'b':2, 'c':3}, {'c':13, 'd':14, 'e':15}, {'e':25, 'f':26, 'g':2
   ```

   if we wrote **d['c'] = 'new'** then only the dictionary in index 1 (the last one/highest index with key **'c'**) would change **'c'**s to associate with **'new'**. It would now be

   ```
   DictList({'a':1, 'b'2, 'c':3}, {'c':'new', 'd':14, 'e':15}, {'e':25, 'f':26, 'g':27}
   ```

   In the example above, if we instead wrote **d['x'] = 'new'** then a new dictionary would be appended to the **list** (with **'x'** associated with **'new'** in that dictionary). It would now be

   ```
   DictList({'a':1,'b':2,'c':3},{'c':13,'d':14,'e':15},{'e':25,'f':26,'g':27},{'x':'new
   ```

8. Define the **__call__** method so that calling **d(k)** on **DictList d** returns a **list** of **2-tuples**: the list index for every dictionary in which **k** is a key and its associated value in that dictionary. If the key is in no dictionaries in the **list**, it returns **[]**. In the example above
   - **d('a')** returns **[(0, 1)]** (it is only in the list's index-**0** dictionary, with an associated value of **1**)
   - **d('e')** returns **[(1, 15), (2, 25)]** (it is in the list's index-**1** dictionary, with an associated value of **15** and it is in the list's index-**2** dictionary, with an associated value of **25**)
   - **d('x')** returns **[]** (it is in no dictionaries)

   Note that the indexes that appear first in the the 2-tuple must be increasing: for **d('e')** the uniquely correct answer is **[(1, 15), (2, 25)]**; the following answer is **wrong**: **[(2, 25), (1, 15)]**.

3 of 5

9. Define an **__iter__** method so that it produces 2-tuples (containing key-value pairs) according to the following rules.
   - Each key is produced only once, from the last (highest) index dictionary in which it appears.
   - All the keys coming from one dictionary are produced in alphabetical order.
   - It produces all the keys from the last (highest-indexed) dictionary, followed by all the keys needed from the second to last dictionary, etc.

   These requirements ensure that there is only one correct sequence of values produced by the iterator. For the **DictList d**

   ```
   d = DictList(dict(a=1,b=2,c=3), dict(c=13,d=14,e=15), dict(e=25,f=26,g=27)
   ```

   the values are produced in the order

   ```
   ('e', 25), ('f', 26), ('g', 27), ('c', 13), ('d', 14), ('a', 1), ('b', 2)
   ```

   Note that the keys **'e'**, **'f'**, and **'g'** are produced in alphabetical order from the **index-2** dictionary; keys **'c'** and **'d'** are produced in alphabetical order from the **index-1** dictionary (key **'e'** has already been produced); keys **'a'** and **'b'** are produced in alphabetical order from the **index-0** dictionary (key **'c'** has already been produced).

10. Define the == operator for comparing two **DictList**s or for comparing a **DictList** and a **dict** for equality. We define the meaning of **d1 == d2** as follows:
    - The keys in the left operand are the same as the keys in the right operand.
      The keys in a **DictList** operand are all the keys appearing in any of the its dictionaries; the keys in a **dict** operand are all the keys in that dictionary (the standard meaning).

      and

    - For each of the keys **k** computed above, **d1[k] == d2[k]**.
      [k] in a **DictList** is the value associated with **k** in the latest dictionary (the one with the highest index in the list); **[k]** in a **dict** is the value assocated with key **k** (the standard meaning).

    If the right operand is neither a **DictList** nor a **dict**, raise the **TypeError** exception with an appropriate message.

    For example, if **d1 = DictList(dict(a=1,b=2), dict(b=12,c=13))** and **d2 = DictList(dict(a=1,b=12), dict(c=13))** then **d1 == d2** is **True**: both have keys **a**, **b**, and **c**; and, both have **['a'] == 1**, **['b'] == 12**, and **['c'] == 13**. For the same reasons, **d1 == dict(a=1,b=12,c=13)** would also be **True**.

    But **d1 == dict(a=1,c=13)** is **False** because the **dict** has operand has no **'b'** key; and **d1 == dict(a=1,b=2,c=13)** is **False** because the **d1['b'] == 12** but the **dict** operand associates **'b'** with the value **2**.

    Hint:

    - Assuming that you implemented **__getitem__** correctly (in part 6), use it when appropriate here.

You may define other (helper) Python methods in this class, but you do not have to define any.

IMPORTANT: The following method is worth just 1 **extra credit** point. Do not solve it unless you have correctly written all the other methods, which are worth many more points. I included this problem to test the programming speed/ability of the more advanced students.

---

4 of 5

11. Define adding two **DictList**s and adding a **DictList** and a **dict** as follows.

    - To add two **DictList**s, create a new **DictList** that contains exactly two dictionaries: at index 0, a dictionary containing all the keys and values that appear in the left **DictList** argument; at index 1, a dictionary containing all the keys and values that appear in the right **DictList** argument.

      For example, if **d1 = DictList(dict(a=1,b=2), dict(b=12,c=13))** and **d2 = DictList(dict(a='one',b='two'), dict(b='twelve',c='thirteen'))** then **d1+d2** would be equivalent to

      ```
      DictList({'a': 1, 'b': 12, 'c': 13}, {'a': 'one', ''b': 'twelve', 'c': 'thirtee
      ```

      and **d2+d1** would be equivalent to

      ```
      DictList({'a': 'one', 'b': 'twelve', 'c': 'thirteen'}, {'a': 1, 'b': 12, 'c': 1
      ```

      So addition is not commutative for the **DictList** class: **d1+d2** produces a different result than **d2+d1**.

      Note that each **DictList** argument is reduced to a single dictionary, containing every possible key in the **DictList**, and each key is associated with the value that it is associated with in its highest index dictionary.

    - To add **DictList** + **dict**, create a new **DictList** with a list of dictionaries that contains a **copy** of all the **dict**s in the **DictList** operand (in order) followed by a **copy** of the **dict** operand.

      For example, if **adl = DictList(dict(a=1,b=2), dict(b=12,c=13))** and **adict = dict(a='one',b='two')** then **adl+adict** would return the equivalent to

      ```
      DictList({'a': 1, 'b': 2}, {'b': 12, 'c': 13}, {'a': 'one', 'b': 'two'})
      ```

    - To add **dict** + **DictList**, create a new **DictList** with a list of dictionaries that contains a **copy** of **dict** operand followed a **copy** of all the **dict**s in the **DictList** operand (in order).

      For example, if **adl = DictList(dict(a=1,b=2), dict(b=12,c=13))** and **adict = dict(a='one',b='two')** then **adict+adl** would be return the equivalent to

      ```
      DictList({'a': 'one', 'b': 'two'}, {'a': 1, 'b': 2}, {'b': 12, 'c': 13})
      ```

      Note the use of **copy** in the specifications above: changing an argument dictionary after + should not affect the resulting dictionary. For example, if we declare **adl** and **adict** as above, and compute **d = adl+adict** and then write **adl['c'] = 'new'**, nothing is changed in **d**.

    If the right operand is neigher a **DictList** nor a **dict**, raise **TypeError** with an appropriate message.

5 of 5