

# 6.101 Exam 2

Spring 2023

Name:

Kerberos/Athena Username:

5 questions

2 hours

- Please **WAIT** until we tell you to begin.
- Write your name and kerberos **ONLY** on the front page.
- This exam is closed-book, but you may use one  $8.5 \times 11$  sheet of paper (both sides) as a reference.
- You may **NOT** use any electronic devices (including computers, calculators, phones, etc.).
- If you have questions, please **raise your hand** and a staff member will come to you.
- Enter all answers in the boxes provided. Work on other pages with QR codes may be taken into account when assigning partial credit. **Please do not write on the QR codes.**
- If you finish the exam more than 10 minutes before the end time, please quietly bring your exam to us at the front of the room. If you finish within 10 minutes of the end time, please remain seated so as not to disturb those who are still finishing their exams.
- You may not discuss the details of the exam with anyone other than course staff until final exam grades have been assigned and released.

*Worksheet (intentionally blank)*

# 1 Swap Pairs

Given a list of elements, we would like you to implement a recursive function that returns a new list with the adjacent pairs of elements in the original list swapped. If the original list has an odd number of elements, include but do not swap the last element at the end of the new list.

```
>>> swap_pairs([])
[]

>>> swap_pairs([1])
[1]

>>> swap_pairs([1, 2])
[2, 1]

>>> swap_pairs([1, 2, 3])
[2, 1, 3]

>>> swap_pairs([1, 2, 3, 4])
[2, 1, 4, 3]
```

Complete the definition of `swap_pairs` below by filling the blanks with appropriate Python code segments.

```
def swap_pairs(inp):
```

```
    if
```

```
        :
```

```
        return
```

```
    return
```

## 2 DefaultDict

We would like to implement a dictionary-like `DefaultDict` class that is a subclass of the built-in `dict` class. Instances of the `DefaultDict` class act like a normal dictionary except that when we call `__getitem__` on a missing key, we would like to insert the key with a default value instead of raising a `KeyError`, and return that value. The default value is created by a call to a `default_factory` function with no arguments that was provided during instance creation. For example:

```
>>> dd = DefaultDict(lambda : "my default")
>>> dd[5]
"my default"
>>> dd
{5: "my default"}
>>> dd[5] = 99
>>> dd.get(7, "is a lucky number")
"is a lucky number"
>>> dd
{5: 99}
>>> dd[7]
"my default"
>>> dd
{5: 99, 7: "my default"}
```

```
>>> ss = DefaultDict(list)
>>> ss['first'].append(8)
>>> ss
{'first': [8]}
```

```
>>> s = 'mississippi'
>>> d = DefaultDict(int)
>>> for k in s:
...     d[k] += 1
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

If a `default_factory` is not provided, its value should default to `None`. In that case, if the key is missing, `__getitem__` should raise a `KeyError` exception with the key as the argument. If a value for `default_factory` is provided and calling it raises an exception, this exception is propagated unchanged. Note that instance creation only supports the optional argument `default_factory` (i.e., it does not take additional initialization arguments).

```
>>> ee = DefaultDict()
>>> ee[5]
Traceback (most recent call last):
...
KeyError: 5
```

Complete the definition of the `DefaultDict` class below to implement all of the specified and example behavior above. In order to receive full credit, `DefaultDict` should only override two methods and add one instance attribute.

```
class DefaultDict(dict):
    def __init__(self, default_factory=None):

    def __getitem__(self, key):
```

*Worksheet (intentionally blank)*

### 3 Scoping out your Inheritance

Write what will get printed for each of these code sequences. If running the code sequence would result in an error, write ERROR instead.

**Part a:**

```
class A:
    foo = 1

    def update(self, i):
        self.foo = self.foo + i

a = A()
a.update(10)
print(a.foo, A.foo)
```

**Part b:**

```
class A:
    foo = 1

    def update(self, i):
        self.foo += i

a = A()
a.update(10)
print(a.foo, A.foo)
```

**Part c:**

```
class A:
    foo = [1]

    def update(self, i):
        self.foo = self.foo + [i]

a = A()
a.update(10)
print(a.foo, A.foo)
```

**Part d:**

```
class A:
    foo = [1]

    def update(self, i):
        self.foo += [i]

a = A()
a.update(10)
print(a.foo, A.foo)
```

*Worksheet (intentionally blank)*



**Part e:**

```
class A:
    foo = [1]

    def update(self, i):
        self.foo.extend([i])

class B(A):
    foo = [100]

    def update(self, i):
        self.foo.extend([i, i])

b = B()
b.update(10)
print(b.foo, A.foo, B.foo)
```

**Part f:**

```
class A:
    foo = [1]

    def update(self, i):
        self.foo.extend([i])

class B(A):
    foo = [100]

    def update(self, i):
        super().update(i)

b = B()
b.update(10)
print(b.foo, A.foo, B.foo)
```

**Part g:**

```
class A:
    foo = [1]

    def update(self, i):
        self.foo.extend([i])

class B(A):
    foo = [100]

    def update(self, i):
        super().update(i)
        self.foo.extend([i, i])

b = B()
b.update(10)
print(b.foo, A.foo, B.foo)
```

**Part h:**

```
class A:
    foo = [1]

    def update(self, i):
        A.foo.extend([i])

class B(A):
    foo = [100]

    def update(self, i):
        super().update(i)

b = B()
b.update(10)
print(b.foo, A.foo, B.foo)
```

## 4 Combinations

### Part a: Function

Given a list of elements, write a function that returns a list of lists, consisting of all of the possible combinations of the elements of the original list. Elements in the original list can be assumed to be unique (i.e., appear in the list only once). Note that the order of the elements in each combination, and the order of combinations in the output list, do not matter.

```
>>> sorted(combos([1, 2]))  
[[], [1], [1, 2], [2]]
```

```
>>> sorted(combos([1, 2, 3]))  
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
```

Complete the definition of `combos` below by filling the blanks with appropriate Python code segments.

```
def combos(inp):
```

```
    # base case
```

```
    clist = []
```

```
    # recursive case
```

```
    return clist
```

**Part b: Generator**

Given a list of elements, write a generator that yields all of the possible combinations of the elements of the original list. Elements in the original list can be assumed to be unique (i.e., appear in the list only once). Note that the order of the elements in each combination, and the order in which combinations are yielded, do not matter.

```
>>> sorted(combos_gen([1, 2]))  
[[], [1], [1, 2], [2]]
```

```
>>> sorted(combos_gen([1, 2, 3]))  
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
```

Complete the definition of `combos_gen` below by filling the blanks with appropriate Python code segments.

```
def combos_gen(inp):
```

```
    # base case
```



```
    # recursive case
```



*Worksheet (intentionally blank)*

## 5 Crossword Grids

We're going to write a program for finding crossword-like "word grids" in which all rows (read left to right) and all columns (read top to bottom) corresponds to an English word. For example:

```
D I G
O R E
E E L
```

Notice that to fill in the above grid, we need to find a group of six words: three going across (DIG, ORE, and EEL) and three going down (DOE, IRE, and GEL). Our solver will search over word slots (rows or columns where a word will go) one at a time, and return a filled set of slots: e.g.,

```
0 across: DIG
1 across: ORE
2 across: EEL
0 down: DOE
1 down: IRE
2 down: GEL
```

Our program yields a sequence of valid solutions. Here's an example where we look at the first ten solutions for one run:

```
puzzle = format_data(3)
solutions = solve(puzzle)
for _ in range(10):
    print_solution(next(solutions))
```

```
SIC      SIC      SIC      SIC      SIC      SIC      SIC      WHO      WHO      WHO
PRO      PRO      PRO      ERA      ERA      ERA      ERA      EAR      EAR      EAR
YEN      YEW      YET      WET      TEN      WED      WEN      EWE      DYE      BYE
```

In the code below, we've provided a few possible implementations of some critical functions. For each function, please (1) indicate which options will produce correct behavior and explain the failure for those that do not; and (2) indicate which option among the correct ones is best (for speed or memory efficiency reasons) and explain your answer. While considering the options for each critical function, you can assume that the other critical functions are all working correctly.

Before answering any of the question parts, you should read the documentation and provided options for code blocks in the `format_data`, `solve`, `update_puzzle`, and `check_puzzle` functions. The documentation in the `solve`, `update_puzzle`, and `check_puzzle` functions may be incomplete; you will need to infer what these functions should do in order to find overall puzzle solutions.

The last page of the exam handout has a copy of the docstrings for these functions, which you may tear off for reference and keep.

```

GRID_SIZE = 3
WORDS = load_words(length=GRID_SIZE) # list of English words of given length

def format_data(grid_size):
    """ Builds a data structure containing three mappings:
    slot_to_word: Holds the eventual solution to the puzzle, which maps each slot like
                   (2, "down") or (3, "across") to a word. Initially, these are all None.
    slot_to_cells: For each slot, contains a list of cells the slot occupies, along
                   with their offset (letter index) into the solution.
                   For example: (0, "down"): [(0, 0), 0), ((1, 0), 1), ((2, 0), 2)],
                   means slot "0 down" has its 0-th letter at (0,0), its 1-st letter at
                   (1,0), and its letter 2 at (2, 0).
    cell_to_slots: For each cell, contains a list of the slots that cross that slot.
                   For example, (0, 1): [(0, "across"), 1), ((1, "down"), 0)] means
                   that cell (0, 1) is a part of "0 across" (holding its letter 1)
                   and "1 down" (holding its letter 0).

    >>> format_data(3)
    {'slot_to_word': {(0, 'across'): None, (0, 'down'): None, (1, 'across'): None,
                      (1, 'down'): None, (2, 'across'): None, (2, 'down'): None},
     'slot_to_cells': {(0, 'across'): [(0, 0), 0), ((0, 1), 1), ((0, 2), 2)],
                      (0, 'down'): [(0, 0), 0), ((1, 0), 1), ((2, 0), 2)],
                      (1, 'across'): [(1, 0), 0), ((1, 1), 1), ((1, 2), 2)],
                      (1, 'down'): [(0, 1), 0), ((1, 1), 1), ((2, 1), 2)],
                      (2, 'across'): [(2, 0), 0), ((2, 1), 1), ((2, 2), 2)],
                      (2, 'down'): [(0, 2), 0), ((1, 2), 1), ((2, 2), 2)]},
     'cell_to_slots': {(0, 0): [(0, 'across'), 0), ((0, 'down'), 0)],
                      (0, 1): [(0, 'across'), 1), ((1, 'down'), 0)],
                      (0, 2): [(0, 'across'), 2), ((2, 'down'), 0)],
                      (1, 0): [(1, 'across'), 0), ((0, 'down'), 1)],
                      (1, 1): [(1, 'across'), 1), ((1, 'down'), 1)],
                      (1, 2): [(1, 'across'), 2), ((2, 'down'), 1)],
                      (2, 0): [(2, 'across'), 0), ((0, 'down'), 2)],
                      (2, 1): [(2, 'across'), 1), ((1, 'down'), 2)],
                      (2, 2): [(2, 'across'), 2), ((2, 'down'), 2)]}}

    """
    # code omitted; can assume it works correctly

def solve(puzzle):
    """ Given a partially completed puzzle (in the format returned by
    format_data), fills in the rest of the values. """

def update_puzzle(puzzle, slot, word):
    """ Return a new puzzle with word in the given slot """

def check_puzzle(puzzle, slot):
    """ For a given slot, return True if the word in that slot does not
    conflict with any letters in any other slots filled in so far. """

```

*Worksheet (intentionally blank)*

```
def solve(puzzle):
    """ Given a partially completed puzzle (in the format returned by
    format_data), fills in the rest of the values. """

    if all(v is not None for v in puzzle["slot_to_word"].values()):
        yield puzzle
        return

    next_slot = next(
        slot for slot, slot_value in puzzle["slot_to_word"].items()
        if slot_value is None
    )

    # OPTION SOLVE-1

    for word in WORDS:
        new_puzzle = update_puzzle(puzzle, next_slot, word)
        if not check_puzzle(new_puzzle, next_slot):
            continue
        yield from solve(new_puzzle)

    # OPTION SOLVE-2

    for word in WORDS:
        new_puzzle = update_puzzle(puzzle, next_slot, word)
        if check_puzzle(new_puzzle, next_slot):
            return solve(new_puzzle)

    # OPTION SOLVE-3

    for word in WORDS:
        new_puzzle = update_puzzle(puzzle, next_slot, word)
        for solved_puzzle in solve(new_puzzle):
            if check_puzzle(solved_puzzle, next_slot):
                yield solved_puzzle
```



**OPTION SOLVE-1:**

Does option SOLVE-1 work correctly?    Circle one:    **Yes**    **No**

If it does not work correctly, explain why not:

**OPTION SOLVE-2:**

Does option SOLVE-2 work correctly?    Circle one:    **Yes**    **No**

If it does not work correctly, explain why not:

**OPTION SOLVE-3:**

Does option SOLVE-3 work correctly?    Circle one:    **Yes**    **No**

If it does not work correctly, explain why not:

**BEST SOLVE OPTION:**

Explain which is the best option (among the correct solve options above), for speed or memory efficiency reasons:

```
def update_puzzle(puzzle, slot, word):
    """ Return a new puzzle with word in the given slot """

    # OPTION UPDATE-1

    new_puzzle = puzzle.copy()
    new_puzzle["slot_to_word"][slot] = word
    return new_puzzle

    # OPTION UPDATE-2

    slot_to_word = puzzle["slot_to_word"].copy()
    slot_to_word[slot] = word
    return {
        "slot_to_word": slot_to_word,
        "slot_to_cells": puzzle["slot_to_cells"],
        "cell_to_slots": puzzle["cell_to_slots"],
    }

    # OPTION UPDATE-3

    new_puzzle = {k: v.copy() for k, v in puzzle.items()}
    new_puzzle["slot_to_word"][slot] = word
    return new_puzzle
```

**OPTION UPDATE-1:**

Does option UPDATE-1 work correctly?    Circle one:    **Yes**    **No**

If it does not work correctly, explain why not:

**OPTION UPDATE-2:**

Does option UPDATE-2 work correctly?    Circle one:    **Yes**    **No**

If it does not work correctly, explain why not:

**OPTION UPDATE-3:**

Does option UPDATE-3 work correctly?    Circle one:    **Yes**    **No**

If it does not work correctly, explain why not:

**BEST UPDATE OPTION:**

Explain which is the best option (among the correct solve options above), for speed or memory efficiency reasons:

```
def check_puzzle(puzzle, slot):
    """ For a given slot, return True if the word in that slot does not
        conflict with any letters in any other slots filled in so far. """

    # OPTION CHECK-1

    my_word = puzzle["slot_to_word"][slot]
    for cell, offset in puzzle["slot_to_cells"][slot]:
        my_letter = my_word[offset]
        for other_slot, other_offset in puzzle["cell_to_slots"][cell]:
            if other_slot == slot:
                continue
            other_word = puzzle["slot_to_word"][other_slot]
            if other_word is None or other_word[other_offset] == my_letter:
                continue
            else:
                return False
    return True

    # OPTION CHECK-2

    for cell, slots in puzzle["cell_to_slots"].items():
        for slot, offset in slots:
            letters = [
                puzzle["slot_to_word"][other_slot][other_offset]
                for other_slot, other_offset in slots
                if puzzle["slot_to_word"][other_slot]
            ]
            if len(set(letters)) > 1:
                return False
    return True
```

**OPTION CHECK-1:**

Does option CHECK-1 work correctly?    Circle one:    **Yes**    **No**

If it does not work correctly, explain why not:

**OPTION CHECK-2:**

Does option CHECK-2 work correctly?    Circle one:    **Yes**    **No**

If it does not work correctly, explain why not:

**BEST CHECK OPTION:**

Explain which is the best option (among the correct solve options above), for speed or memory efficiency reasons:

*Worksheet (intentionally blank)*

```

GRID_SIZE = 3
WORDS = load_words(length=GRID_SIZE) # list of English words of given length

def format_data(grid_size):
    """ Builds a data structure containing three mappings:
    slot_to_word: Holds the eventual solution to the puzzle, which maps each slot like
                    (2, "down") or (3, "across") to a word. Initially, these are all None.
    slot_to_cells: For each slot, contains a list of cells the slot occupies, along
                    with their offset (letter index) into the solution.
                    For example: (0, "down"): [(0, 0), 0), ((1, 0), 1), ((2, 0), 2)],
                    means slot "0 down" has its 0-th letter at (0,0), its 1-st letter at
                    (1,0), and its letter 2 at (2, 0).
    cell_to_slots: For each cell, contains a list of the slots that cross that slot.
                    For example, (0, 1): [(0, "across"), 1), ((1, "down"), 0)] means
                    that cell (0, 1) is a part of "0 across" (holding its letter 1)
                    and "1 down" (holding its letter 0).

    >>> format_data(3)
    {'slot_to_word': {(0, 'across'): None, (0, 'down'): None, (1, 'across'): None,
                      (1, 'down'): None, (2, 'across'): None, (2, 'down'): None},
     'slot_to_cells': {(0, 'across'): [(0, 0), 0), ((0, 1), 1), ((0, 2), 2)],
                      (0, 'down'): [(0, 0), 0), ((1, 0), 1), ((2, 0), 2)],
                      (1, 'across'): [(1, 0), 0), ((1, 1), 1), ((1, 2), 2)],
                      (1, 'down'): [(0, 1), 0), ((1, 1), 1), ((2, 1), 2)],
                      (2, 'across'): [(2, 0), 0), ((2, 1), 1), ((2, 2), 2)],
                      (2, 'down'): [(0, 2), 0), ((1, 2), 1), ((2, 2), 2)]},
     'cell_to_slots': {(0, 0): [(0, 'across'), 0), ((0, 'down'), 0)],
                      (0, 1): [(0, 'across'), 1), ((1, 'down'), 0)],
                      (0, 2): [(0, 'across'), 2), ((2, 'down'), 0)],
                      (1, 0): [(1, 'across'), 0), ((0, 'down'), 1)],
                      (1, 1): [(1, 'across'), 1), ((1, 'down'), 1)],
                      (1, 2): [(1, 'across'), 2), ((2, 'down'), 1)],
                      (2, 0): [(2, 'across'), 0), ((0, 'down'), 2)],
                      (2, 1): [(2, 'across'), 1), ((1, 'down'), 2)],
                      (2, 2): [(2, 'across'), 2), ((2, 'down'), 2)]}}

    """
    # code omitted; can assume it works correctly

def solve(puzzle):
    """ Given a partially completed puzzle (in the format returned by
    format_data), fills in the rest of the values. """

def update_puzzle(puzzle, slot, word):
    """ Return a new puzzle with word in the given slot """

def check_puzzle(puzzle, slot):
    """ For a given slot, return True if the word in that slot does not
    conflict with any letters in any other slots filled in so far. """

```

*Worksheet (intentionally blank)*