Name: Answers

Kerberos/Athena Username:

5 questions  1 hour and 50 minutes

- Please WAIT until we tell you to begin.
- This quiz is closed-book, but you may use one 8.5 × 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 come to us at the front to ask them.
- 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 quizzes.
- You may not discuss the details of the quiz with anyone other than course staff until final quiz grades have been assigned and released.
Worksheet (intentionally blank)
1 Environment Model: Function Calls

For each of the two pieces of code on the following pages, fill in the associated environment diagram and indicate what will be printed to the screen when the code is run. Each partial environment diagram contains all of the frames and objects necessary but is missing several pieces, which you should fill in. You need not write the full body of each function inside the associated object, but your diagram should otherwise be complete and represent the state of the program when all code is run but before garbage collection removes any objects. If a piece of code would raise an exception, instead write a brief description of the error and its cause.

Note also that print does not create new frames, so there is no need to include those in your environment diagram.
1.1 Fragment 1

```python
x = 2
def foo(x):
    def bar():
        print("bar:", x)

    bar()
    print("foo:", x)

foo(0)
print("outside:", x)
```

Output or brief description of error:

```
bar: 0
foo: 0
outside: 2
```
1.2 Fragment 2

```python
x = 2
def foo(x):
    bar()
    print("foo:", x)

def bar():
    print("bar:", x)

x = 1
foo(0)
print("outside:", x)
```

Output or brief description of error:

```
bar: 1
foo: 0
outside: 1
```
2 Refactoring

Your roommate, Lem E. Tryit, is trying their hand at implementing code for dealing with color images like those we saw in 6.101 lab 2. They show you the following code for a helper function they have been working on. This code works exactly as they intend, but they ask if it could be written more concisely.

def get_color_value(image, color, x, y):
    red = {
        "height": image["height"],
        "width": image["width"],
        "pixels": [
            image["pixels"][(j*image["width"] + i)[0]
            for j in range(image["height"]) for i in range(image["width"])
        ],
    }
    green = {
        "height": image["height"],
        "width": image["width"],
        "pixels": [
            image["pixels"][(j*image["width"] + i)[1]
            for j in range(image["height"]) for i in range(image["width"])
        ],
    }
    blue = {
        "height": image["height"],
        "width": image["width"],
        "pixels": [
            image["pixels"][(j*image["width"] + i)[2]
            for j in range(image["height"]) for i in range(image["width"])
        ],
    }
    if color == "red":
        if y >= image["height"] or y < 0 or x >= image["width"] or x < 0:
            return None
        return red["pixels"][y*image["width"] + x]
    if color == "green":
        if y >= image["height"] or y < 0 or x >= image["width"] or x < 0:
            return None
        return green["pixels"][y*image["width"] + x]
    if color == "blue":
        if y >= image["height"] or y < 0 or x >= image["width"] or x < 0:
            return None
        return blue["pixels"][y*image["width"] + x]
    return None
In the box below, write a version of Lem’s `get_color_value` function that avoids as much repetitious and/or redundant code as possible.

```python
def get_color_value(image, color, x, y):
    index_map = {'red': 0, 'green': 1, 'blue': 2}
    if (0 <= x < image['width'])
        and (0 <= y < image['height'])
        and color in index_map:
        rgb_value = image['pixels'][y*image['width'] + x]
        return rgb_value[index_map[color]]
```

(many solutions are reasonable; this is just one example)
3 Batter Up!

Your roommates are hungry, and so they ask you to make breakfast. You oblige, but they are very picky. By the end of the morning, you have produced some number of pancakes, each with a different size, and each of which is burnt on one side. But your roommates insist that you serve the pancakes in a big stack so that the pancakes are sorted (with the smallest pancake on top), and so that the burnt side on each is down.

The only action you can take when sorting the pancakes is to insert the spatula at any point in the stack and flip all the pancakes above it (inverting their order and changing which of their sides is up), as illustrated in the image below:

Our ultimate goal is to determine: what is the minimum number of flips you need to make in order to satisfy your roommates? Thinking back to week 4 of 6.101, you realize that you can use the general-purpose find_path function from the week 4 readings to solve this problem. The code for this function is included on the second-from-last page of this handout, which you may remove.

Your job for this problem is to fill in the definition of `pancake_flips` on the following pages. Given a list of pancake diameters, the function should return the minimum number of flips (regardless of location) that are needed in order to put the stack in order with all of the burnt sides down.

Note that `pancake_flips` takes as input a single list of numbers, representing the diameters of the pancakes in the stack, with the first value (at index 0) representing the diameter of the pancake on top of the stack of pancakes. You may assume that all pancakes start with their burnt sides facing up (i.e., each pancake starts in the wrong orientation).

Fill in the definitions of `start_state`, `neighbors`, `goal_test`, and `finalize_answer` on the following pages to complete this task. You may not change the return statement at the bottom of `pancake_flips`.

For example, consider the following example:

```python
>>> pancake_flips([3, 2, 1])
1
```

This input represents a stack of three pancakes, where the top pancake is the biggest and the bottom pancake is the smallest (and all three pancakes have their burnt side facing up).

This call returns 1 because a single operation (resulting from putting the spatula under the bottom pancake and flipping) is required to have a stack that satisfies your roommates, as illustrated by the following picture:
def pancake_flips(diameters):
    # state needs to tell us both the size and orientation of each pancake
    # here, the first element in each tuple is the size of a pancake, and
    # the second indicates whether it is in the proper orientation
    start_state = tuple((size, False) for size in diameters)

    # in order to win, each pancake needs to be smaller than (or the same size
    # as) the pancake that follows it, and they all need to be in the proper
    # orientation
    def goal_test(state):
        return all((i[0] <= j[0] and i[1] and j[1]) for i, j in zip(state, state[1:]))

    def flip(stack, N):
        # helper to flip the first N+1 pancakes on the stack
        # we reverse the first N elements and change their orientation
        return tuple((i, not j) for i, j in stack[:N+1][::-1]) + stack[N+1:]

    def neighbors(state):
        return [flip(state, ix) for ix in range(len(cur_state))]

    def finalize_answer(path):
        return len(path) - 1 if path is not None else None

    return finalize_answer(find_path(neighbors, start_state, goal_test))
4 Undo

Wanting to go back in time to undo his promise to purchase the massive social-media company Fritter, powerful CEO Merlon Tusk has been experimenting with a new kind of programming-language feature that allows for "undo" functionality, which can be used to revert variables to older values. Merlon gave the following transcript to his engineering team, representing the desired behavior:

```python
>>> x_set, x_get, x_undo = undoable("no i don't want to buy")
>>> x_get()
"no i don't want to buy"

>>> x_set("fifty cents")
>>> x_set("40 billion dollars")
>>> x_get()
"40 billion dollars"

>>> y_set, y_get, y_undo = undoable("self-driving cars may be possible at some point")
>>> y_set("self-driving cars by 2010")
>>> y_get()
"self-driving cars by 2010"

>>> x_undo()
>>> x_get()
"fifty cents"

>>> y_undo()
>>> y_get()
"self-driving cars may be possible at some point"

>>> x_undo()
>>> x_get()
"no i don't want to buy"

>>> x_undo()
>>> x_get()
"no i don't want to buy"

>>> # (continuing to 'undo' does not change the value at this point)
```

Consider the incomplete code on the facing page, which represents a start toward implementing the "undo" functionality described above. If the function can be completed so that it behaves as expected, fill in the boxes to complete the definition. Otherwise (if this structure cannot be made to work), instead briefly describe why in the box below the function.
def undoable(initial_value):
    values = [initial_value]

    def set_value(value):
        values.append(value)

    def undo():
        if len(values) > 1:
            values.pop()

    def get_value():
        return values[-1]

    return (set_value, get_value, undo)

If the function cannot be completed correctly given this structure, briefly describe why below:
N/A (the answer above works)
5 Gone Fishing

Your good friend Ben Bitdiddle is undertaking a summer internship as a park ranger, and they have been tasked with finding ponds in satellite imagery to help people find good fishing spots. The images are presented as a list of strings, where a "." character represents a region of land and a "W" represents water. A group of connected "W" characters represents a pond. For example, the following grid and image represent the same region of land, which contains 4 ponds:

```
grid = [
    "W...WW",
    "..W..W",
    "..WWW..",
    "..W.W..",
    "..WWW.W",
    ".....W",
]
```

Ben decides to take an approach based on the flood-fill algorithm presented in 6.101, writing a function called `count_ponds` that takes a grid of this form as input and returns an integer indicating the number of ponds in that grid. Code for `count_ponds` and all necessary helpers are included on the last page of this handout, which you may remove.

However, Ben’s code contains one or more errors, which are causing it not to produce the desired results. In the space below (and continued on the facing page), describe the issue(s) with the code as written, and briefly describe a fix for each.

The high-level approach here is reasonable, but there are several issues with Ben’s code:

- On line 4, we are not looping as far as we should (this will only consider the lower-left half of the grid). Line 4 should read: `for column in range(len(grid[row]))`

- In `get_neighbors`, we should only be including neighbors that are within the bounds of the grid and that are marked as water.

- On line 5, we should only be calling `get_pond` starting at locations that contain a "W" character.

- `found_ponds` is initialized on line 2 to be a set, but the things we are trying to put in it (the values returned from calling `get_pond` are themselves sets; this will raise an exception because sets cannot contain other sets (because they are not hashable). As such, line 6 should be changed to read: `found_ponds.add(frozenset(pond))`

With those changes made, Ben’s code should now work as expected!
Worksheet (intentionally blank)
Worksheet (intentionally blank)
Worksheet (intentionally blank)
Code for `find_path` Function

```python
def find_path(neighbors_function, start, goal_test):
    if goal_test(start):
        return (start,)
    agenda = [(start,)]
    visited = {start}
    while agenda:
        this_path = agenda.pop(0)
        terminal_state = this_path[-1]
        for neighbor in neighbors_function(terminal_state):
            if neighbor not in visited:
                new_path = this_path + (neighbor,)
                if goal_test(neighbor):
                    return new_path
                agenda.append(new_path)
                visited.add(neighbor)
```

Code For "Gone Fishing"

```python
def count_ponds(grid):
    found_ponds = set()
    for row in range(len(grid)):
        for column in range(row):
            pond = get_pond(grid, row, column)
            found_ponds.add(pond)
    return len(found_ponds)

def get_pond(grid, row, column):
    pond = set()
    locations = [(row, column)]
    while locations:
        one_loc = locations.pop()  # DFS
        pond.add(one_loc)
        for neighbor in get_neighbors(one_loc):
            if neighbor not in pond:
                locations.append(neighbor)
    return pond

def get_neighbors(location):
    r, c = location
    return [(r+1, c), (r-1, c), (r, c+1), (r, c-1)]
```