6.101 Midterm

Spring 2024

Name: **Answers**

Kerberos/Athena Username:

5 questions 1 hour and 50 minutes

- Please **WAIT** until we tell you to begin.
- Write your name and kerberos **ONLY** on the front page.
- This exam is closed-book, and 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, but if your work is on another page, please include a reference to the relevant page number in or near the relevant box.
- **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 working.
- You may not discuss details of the exam with anyone other than course staff until exam grades have been assigned and released.

1 Operation Combinations

In the box on the facing page, write a function called `combinations` to solve the following puzzle: given a list of integers, what are all of the numbers that can be formed by sequentially operating on those numbers using `+`, `-`, `*`, and `//` (note: *integer division*), without changing the order of the numbers?

Your function should take as input a nonempty list of numbers, and it should return a set of the numbers that can result from applying these operations. You should assume that all operations happen in left-to-right order, regardless of the normal order of operations.

For example, consider the following:

```
>>> combinations([5])
{5}
>>> combinations([1, 2])
{0, 2, 3, -1}
>>> combinations([2, 1])
{1, 2, 3}
>>> combinations([1, 2, 3])
{0, 1, 2, 3, 5, 6, 9, -4, -3, -1}
```

Note how the numbers in the last example were generated:

- `-4` can arise from `(1 - 2) - 3`
- `-3` can arise from `(1 - 2) * 3`
- `-1` can arise from `(1 * 2) - 3`
- `0` can arise from `(1 + 2) - 3`
- `1` can arise from `(1 + 2) // 3`
- `3` can arise from `(1 // 2) + 3`
- etc.

Note that every number from the input is involved in every calculation, and always in the same order.

You do not need to worry about dividing by 0 (i.e., you may assume that your code will never be run on inputs that would lead to a correct implementation trying to divide by 0, so you don't need any code to handle that special case).

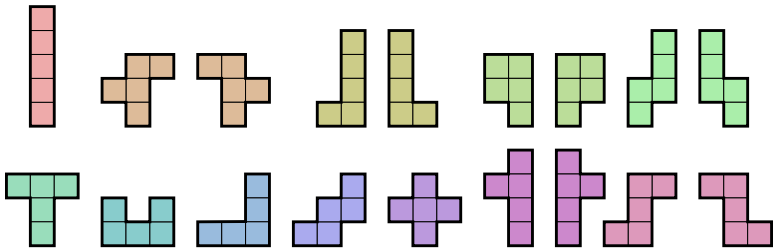
Your implementation should not use Python's `eval` function, and part of your grade will be based on style and avoiding repetitious code. You are welcome to define whatever additional variables and/or helper functions you wish.

```
def combinations(nums):
    if len(nums) < 2:
        return set(nums)
    return {op(old, nums[-1]) for old in combinations(nums[:-1])}

ops = [
    lambda x, y: x+y,
    lambda x, y: x-y,
    lambda x, y: x//y,
    lambda x, y: x*y,
]









# alternatively, (though less efficient):
def combinations(nums):
    if len(nums) < 2:
        return set(nums)
    out = set()
    for op in ops:
        out |= combinations([op(nums[0], nums[1])] + nums[2:])
    return out
```

Worksheet (intentionally blank)

2 Pentomin-oh no, it's so slow!

Your friend Ben Bitdiddle has been working away at a piece of code to represent *pentominoes* in Python. A pentomino is a shape made by connecting 5 squares along their edges. There are 18 distinct types of pentominoes, not accounting for rotations:



(This image was created by R.A. Nonenmacher and licensed for our use under a Creative Commons license. More information is available at: `https://en.wikipedia.org/wiki/File:All_18_Pentominoes.svg`*)*

Ben has chosen to represent each of these shapes as a list of `(row, column)` offsets from some roughly central point. In Ben's representation, these shapes are stored in a list called `SHAPES`:

```
SHAPES = [
    [(0,0), (1,0), (2,0), (3,0), (4,0)],
    [(0,0), (0,1), (-1,1), (-1,2), (1,1)],
    [(0,0), (0,1), (1,1), (1,2), (2,1)],
    [(0,0), (0,1), (-1,1), (-2,1), (-3,1)],
    [(0,0), (-1,0), (-2,0), (-3,0), (0,1)],
    # ...
    # imagine that this list continues with the rest of the shapes above, as
    # well as additional shapes comprised of rotated versions of each of the
    # above
]
```

Ben has written a function to compute whether a given set of pentominoes "covers" a grid of a particular size, which is shown (and documented) on the following page. This code is documented correctly, and it works; but it is quite slow, particularly for big boards with lots of pentominoes.

In the box, write a version of `covers` that computes the same result but operates substantially more efficiently. You are welcome to use Ben's helper functions (you do not need to rewrite them), and you may define additional helper functions of your own.

```
def covers(grid_height, grid_width, pentominoes):
    """
    Given a grid of locations and a list of pentominoes, return True if the given
    arrangement of pentominoes completely covers the board, or False otherwise.  The grid
    should be considered covered if every location from (0, 0) through (grid_height-1,
    grid_width-1) has a part of a pentomino in it, no two pentominoes overlap, and no part
    of any of the given pentominoes extends outside the grid.

    Inputs:
        * grid_height is an integer indicating the height of the grid
        * grid_width is an integer indicating the width of the grid
        * pentominoes is a list of (shape, row, column) tuples, where shape is an
          integer index into the SHAPES array, and row and column represent the location
          in the grid where the (0,0) anchor point of the given shape is placed
    """
    success = True
    for row in range(grid_height):
        for col in range(grid_width):
            if not cell_covered_once(row, col, pentominoes):
                success = False
    for p in pentominoes:
        if not in_bounds(grid_height, grid_width, p):
            success = False
    return success

def add_tuples(t1, t2):
    return tuple(i+j for i,j in zip(t1, t2))

def absolute_locations(p):
    shape, row, column = p
    return [add_tuples((row, column), offset) for offset in SHAPES[shape]]

def in_bounds(grid_height, grid_width, p):
    locations = absolute_locations(p)
    for location in locations:
        if not (0 <= location[0] < grid_height and 0 <= location[1] < grid_width):
            return False
    return True

def cell_covered_once(row, col, pentominoes):
    for p1 in pentominoes:
        if (row, col) in absolute_locations(p1):
            others = [p for p in pentominoes if p != p1]
            for p2 in others:
                if (row, col) in absolute_locations(p2):
                    return False
            return True
    return False
```

```
# your code for covers (and any helpers you want) here:
def covers(grid_height, grid_width, pentominoes):
    open_spaces = {(r, c) for r in range(grid_height) for c in range(grid_width)}
    for p in pentominoes:
        this_pentomino = set(absolute_locations(p))
        if this_pentomino - open_spaces:
            # there's a spot in the pentomino that isn't available
            return False
        # update the set of open spaces; the ones covered by this pentomino are
        # no longer available
        open_spaces -= this_pentomino
    return True
```

3 GPT

You've been asked to write a function to find the first sentence in a Python string, where a sentence is defined to be a sequence of adjacent characters that starts with a capital letter, ends with a period, and doesn't contain any other periods. If there are no sentences in the given input, your function should return `None` instead.

But it's a nice day outside and you'd rather not spend it sitting in front of your computer, so you ask ChatGPT to write it for you. ChatGPT cheerfully replies: "Certainly, here is a Python function that finds the first sentence in a Python string," and then it produces the following code:

```
def find_sentence(string):
    start = 0
    i = start
    scanning = False
    while i < len(string)-1:
        if string[i].isupper() and not scanning:
            start = i
            scanning = True
        elif string[i] == '.' and scanning:
            return string[start:start+i]
        i += 1
```

To test it out, you run the code through a small test case:

```
>>> find_sentence('not a sentence. This is a sentence. also not a sentence.')
'This is a sentence. also not a sen'
```

Oh, no! Despite ChatGPT's cheerful response, the code is broken. The test case above should have returned `'This is a sentence.'` instead. Seeing this behavior, you decide to test the code further.

On the facing page are several examples of results that the code could have when it is run. For each, if it is possible for the code above to produce that result, provide an input that would lead to that result. If an outcome is not possible, write an "X" in the box instead. Your inputs should all be distinct from the example given above, but they should all be valid (i.e., they should be strings).

Valid input that produces the correct string as a result (or X if not possible):

```
"aSentence. wheee"
```
(there must be a single character before the first sentence, and the period can't be the least character)

Valid input that correctly produces `None` as a result (or X if not possible):

```
'no sentences here.'
```
(any string with no sentence in it)

Valid input that should produce a string but produces an incorrect string (or X if not possible):

```
'this is like. The example. from above.'
```
(any string where the first sentence starts at an index other than 1)

Valid input that should produce a string but instead produces `None` (or X if not possible):

```
'hey, This is my sentence.'
```
(any string where nothing follows the first sentence)

Valid input that should produce `None` but instead produces a string (or X if not possible):

X
(this can't happen; we only return a string if there is a sentence)

Valid input that causes an exception to be raised (or X if not possible):

X
(this can't happen for well-formed input)

Valid input that enters an infinite loop (or X if not possible):

X
(this can't happen either)

4 Environment Diagram

In the box below, write what will be printed to the screen when the program is run. If an error occurs, write all of the output up to that point as well as the approximate error message Python would produce.

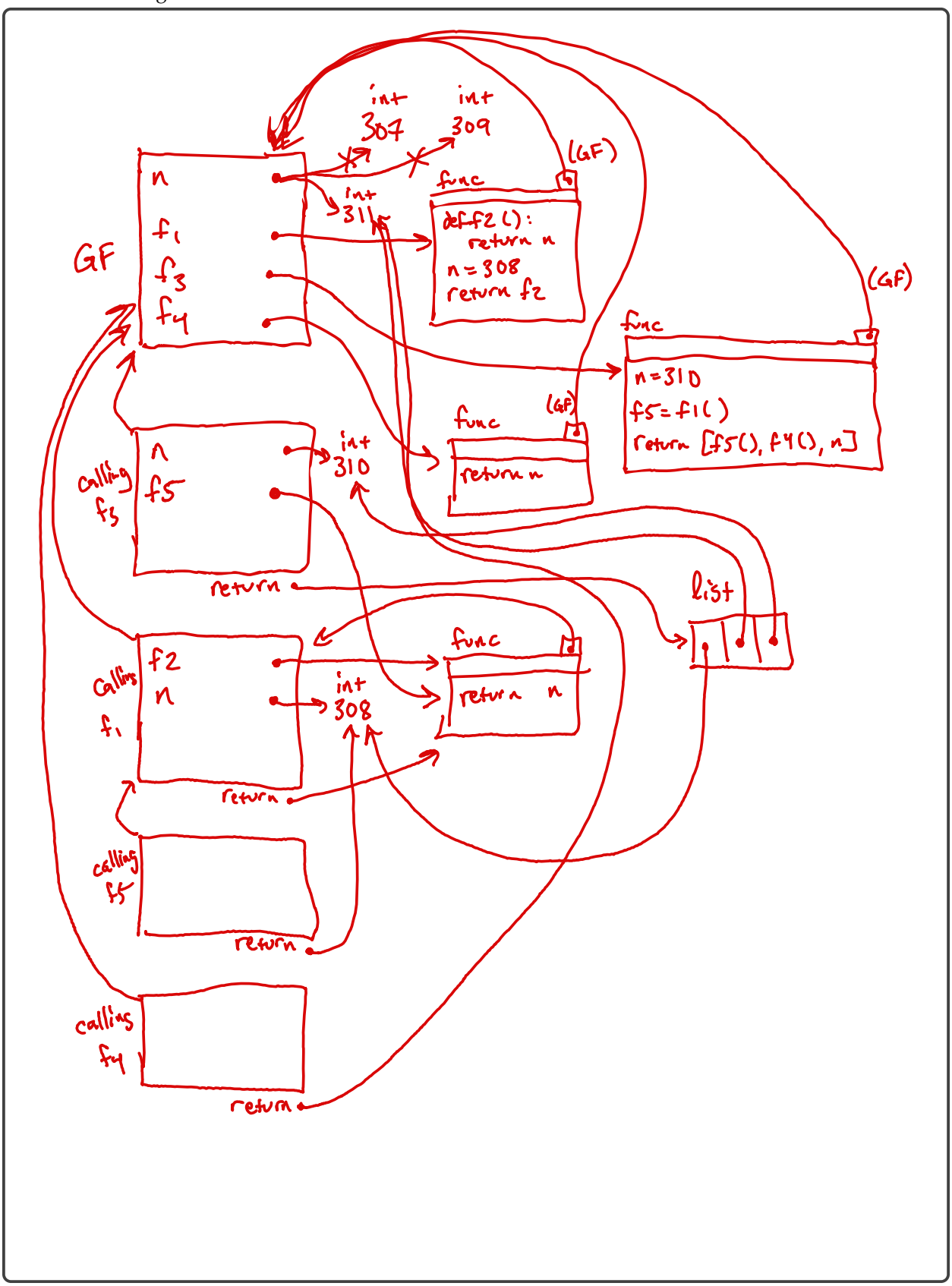
On the facing page, draw an environment diagram showing the state of the program just before it finishes (but do not delete frames or objects even if they have been garbage-collected). You do not need to rewrite the bodies of the functions in the diagram unless you want to.

```
n = 307

def f1():
    def f2():
        return n

    n = 308
    return f2

n = 309

def f3():
    n = 310
    f5 = f1()
    return [f5(), f4(), n]

def f4():
    return n

n = 311

print(f3())
```

Output:

```
[308, 311, 310]
```

Environment Diagram:



5 Number Puzzle

In this problem, we'll work to implement a function that will tell us, given a starting number, and ending number, and some ways of perturbing numbers, the smallest number of perturbations we need to make in order to change our the starting number into the target number.

For example, consider converting from `0` to `101` using the following operations: squaring, adding 1, subtracting 1, negating, and doubling. In this case, the shortest sequence of operations is 7 operations long:

- We start with `0`,
- we add 1 to get `1`,
- we add 1 (or double) to get `2`,
- we double (or square) to get `4`,
- we add 1 to get `5`,
- we double to get `10`,
- we square to get `100`,
- and we add 1 to get `101`, and we're done!

On the facing page, implement a function `how_many_steps` to solve puzzles of this form. Your function should take three inputs:

- a starting integer,
- an ending integer, and
- a list of function objects representing the allowed operations, where each function takes an integer as input and returns an integer as output.

For example, the specific puzzle above might be encoded as follows using this function:

```
>>> funcs = [lambda x: x**2, lambda x: x+1, lambda x: x-1, lambda x:-x, lambda x:2*x]
>>> how_many_steps(0, 101, funcs)
7
```

You may assume that the given target number can always be reached from the given start number using the given operations.

```
def how_many_steps(start, target, operations):
    agenda = [(start_number, 0)]  # agenda elements are (result, # steps) tuples
    visited = {start_number}

    while agenda:
        num, steps = agenda.pop(0)
        if num == target_number:
            return steps
        for f in step_functions:
            new = f(num)
            if new in visited:
                continue
            agenda.append((new, steps+1))
            visited.add(new)
```

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)