CS 61A
Spring 2024

Structure and Interpretation of Computer Programs

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address <EMAILADDRESS>. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

◯ You must choose either this option

◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

You can complete and submit these questions before the exam starts.

(a) What is your full name?

(b) What is your student ID number?

(c) What is your @berkeley.edu email address?

(d) Sign (or type) your name to confirm that all work on this exam will be your own. The penalty for academic misconduct on an exam is an F in the course.

1. (6.0 points)   What Would Python Display?

Assume the following code has been executed. No error occurs.

```
s = [2, 4]
t, u = [2, 0, s], [2, 0, s]
t.append(5)
u[2].extend(s)
v = t[2]
t[2] = [6, 7]
```

Write the output printed for each expression below. **Hint:** Draw an environment diagram!

(a) (0.5 pt) `print(s is u)`

○ True

○ False

(b) (0.5 pt) `print(s is v)`

○ True

○ False

(c) (0.5 pt) `print(s is u[2])`

○ True

○ False

(d) (0.5 pt) `print(s is t[2])`

○ True

○ False

(e) (2.0 pt) `print(t)`

(f) (2.0 pt) `print(u)`

2. **(9.0 points)   Only Up**

   Definition: The *increasing sublist* of a sequence of numbers s is a list containing each element of s that is larger than all previous elements.

   (a) **(4.0 points)**

   Implement up, which takes a **list** of numbers s and returns its *increasing sublist*.

   ```
   def up(s):
       """Return the increasing sublist of list s.

       >>> up([3, 1, 2, 5, 4, 5, 7, 6, 12, 11])
       [3, 5, 7, 12]
       """
       result = _____
                   (a)
       for x in s:

           if _____:
                 (b)

               result._____(x)
                       (c)
       return result
   ```

   i. **(1.0 pt)** Fill in blank (a).

      ○ []

      ○ [[]]

      ○ [s]

      ○ [s[0]]

      ○ None

   ii. **(2.0 pt)** Fill in blank (b).

      ○ result and s[0] < x

      ○ result and result[-1] < x

      ○ result or s[0] < x

      ○ result or result[-1] < x

      ○ not result and s[0] < x

      ○ not result and result[-1] < x

      ○ not result or s[0] < x

      ○ not result or result[-1] < x

   iii. **(1.0 pt)** Fill in blank (c).

      ○ append

      ○ extend

      ○ pop

      ○ list

(b) (5.0 points)

Implement `upiter`, which takes a finite **iterator** over numbers `t` and returns its *increasing sublist*. You may **not** call up in `upiter`.

```
def upiter(t):
    """Return the increasing sublist of iterator t.

    >>> upiter(iter([3, 1, 2, 5, 4, 5, 7, 6, 12, 11]))
    [3, 5, 7, 12]
    """
    try:
        x = next(t)
    except StopIteration:
        _____
          (d)
    return _____ + _____ (filter( _____ , t))
             (e)       (f)               (g)
```

i. (1.0 pt) Fill in blank (d).

  ○ `return t`

  ○ `return []`

  ○ `return None`

  ○ `x = t`

  ○ `x = []`

  ○ `x = None`

ii. (1.0 pt) Fill in blank (e).

  ○ `x`

  ○ `next(t)`

  ○ `[x]`

  ○ `[next(t)]`

  ○ `[x, next(t)]`

iii. (1.0 pt) Fill in blank (f).

  ○ `max`

  ○ `list`

  ○ `iter`

  ○ `next`

  ○ `upiter`

iv. (2.0 pt) Fill in blank (g). You **may not** call up.

3. **(26.0 points)   Dominos**

A *domino* is a tile with two numbers, such as 3-4. A 3-4 domino can be flipped around so that it is now a 4-3 domino.

(a) **(8.0 points)**

Implement the Domino class. A Domino instance is constructed from a two-element list of non-negative integers below 10 that is stored as an instance attribute `ns`. The `align` method takes an integer `first` which must be one of the numbers in `ns`. It returns the same domino instance on which it was invoked, but first mutates it so that `first` comes first in `ns`. A `Domino([3, 4])` is displayed as 3-4.

```
class Domino:
    """A Domino has two numbers and can be aligned to place one of them first.

    >>> d = Domino([3, 4])
    >>> original = d
    >>> d.align(3)
    3-4
    >>> d.align(4)
    4-3
    >>> d is original  # No new Domino instance was created
    True
    >>> d               # A Domino's order is changed by align
    4-3
    """
    def __init__(self, ns):
        assert len(ns) == 2 and ns[0] in range(10) and ns[1] in range(10)
        self.ns = ns

    def align(self, first):
        assert _____, 'first must be a number on the domino.'
                  (a)
        if _____:
             (b)

            _____
              (c)
        return _____
                 (d)

    def __repr__(self):
        return _____
                 (e)
```

i. **(2.0 pt)** Fill in blank (a). **Select all that apply.**

☐ `first == self.ns`

☐ `[first] == self.ns`

☐ `first == self.ns[0] or self.ns[1]`

☐ `first in self.ns`

☐ `self.first == self.ns`

☐ `[self.first] == self.ns`

☐ `self.first == self.ns[0] or self.ns[1]`

☐ `self.first in self.ns`

ii. **(1.0 pt)** Fill in blank (b).

○ `first == self.ns[0]`

○ `first != self.ns[0]`

○ `self.first == self.ns[0]`

○ `self.first != self.ns[0]`

iii. **(3.0 pt)** Fill in blank (c).

iv. **(1.0 pt)** Fill in blank (d).

○ `Domino`

○ `Domino(self.ns)`

○ `Domino([self.ns])`

○ `self`

○ `self.ns`

○ `repr(self)`

○ `repr(self.ns)`

v. **(1.0 pt)** Fill in blank (e).

○ `str(self)`

○ `str(self.ns)`

○ `ns[0] + '-' + ns[1]`

○ `self.ns[0] + '-' + self.ns[1]`

○ `str(self.ns[0] + '-' + self.ns[1])`

○ `str(self.ns[0]) + '-' + str(self.ns[1])`

(b) (4.0 points)

Implement `drop`, which takes a `Domino` instance `d` and a list of `Domino` instances `ds`. It returns a new list with all of the elements of `ds` except for `d`.

```
def drop(d, ds):
    """Return a new list of dominos with all elements of ds except d.

    >>> ds = [Domino(ns) for ns in [[5, 2], [3, 4], [5, 5], [3, 4]]]
    >>> drop(ds[1], ds)
    [5-2, 5-5, 3-4]
    >>> drop(ds[3], ds)
    [5-2, 3-4, 5-5]
    >>> ds
    [5-2, 3-4, 5-5, 3-4]
    """
    return _____
            (f)
```

i. (2.0 pt) Fill in blank (f).

```

```

ii. (2.0 pt) What is the value of `drop(ds[2], drop(ds[1].align(4), ds))` assuming `drop` and `Domino` are defined correctly and `ds` is assigned to:

```
ds = [Domino(ns) for ns in [[5, 2], [3, 4], [5, 5], [4, 3]]]
```

○ [5-2]

○ [5-2, 3-4]

○ [5-2, 4-3]

○ [5-2, 5-5]

○ [5-2, 3-4, 5-5]

○ [5-2, 4-3, 5-5]

○ [5-2, 5-5, 4-3]

○ [5-2, 3-4, 5-5, 4-3]

○ [5-2, 4-3, 5-5, 4-3]

(c) (10.0 points)

Definition: A *line* of Dominos is a sequence of dominos in which pairs of adjacent numbers on different dominos are equal, such as 2-4 4-3 3-3 3-6.

Implement `longest`, which takes a list of `Domino` instances `ds`. It returns a string that describes the longest *line* of dominos that can be formed from `ds`. This result should contain domino repr strings separated by spaces, such as `'2-4 4-3 3-3 3-6'`. The dominos described in the result can appear in any order and may be flipped using the `align` method. If more than one line is longest, describe any of the longest lines. You may call `drop`.

```
def longest(ds):
    """Return a string describing the longest line that can be formed from dominos in ds.

    >>> ds = [Domino(ns) for ns in [[5, 2], [3, 4], [5, 5], [6, 1], [3, 2]]]
    >>> print(longest(ds))
    4-3 3-2 2-5 5-5

    >>> ds = [Domino(ns) for ns in [[1, 2],[1, 2],[3, 2],[3, 2],[3, 2],[4, 3],[4, 3],[4, 3],[6,5]]]
    >>> print(longest(ds))
    2-1 1-2 2-3 3-2 2-3 3-4 4-3 3-4
    """
    def finish(first, rest):

        s = [ _____ + ' ' + finish( _____ , _____ ) for d in rest if _____ ]
                 (g)                     (h)       (i)                         (j)

        if not s:

            return _____
                      (k)

        return max(s, key=_____ )
                             (l)
    candidates = []
    for first in ds:
        candidates.append(finish(first,                         drop(first, ds)))
        candidates.append(finish(first.align(first.ns[1]), drop(first, ds)))

    return max(candidates, key=_____ )
                                  (m)
```

i. (1.0 pt) Fill in blank (g).

○ d

○ first

○ rest[0]

○ repr(d)

○ repr(first)

○ repr(rest[0])

ii. (3.0 pt) Fill in blank (h).

```
┌────────────────────────────────────────────────────────────────┐
│                                                                  │
│                                                                  │
│                                                                  │
└────────────────────────────────────────────────────────────────┘
```

iii. (1.0 pt) Fill in blank (i).

○ `rest`

○ `rest[1:]`

○ `drop(d, rest)`

○ `drop(first, rest)`

iv. (2.0 pt) Fill in blank (j).

```
┌────────────────────────────────────────────────────────────────┐
│                                                                  │
│                                                                  │
│                                                                  │
└────────────────────────────────────────────────────────────────┘
```

v. (2.0 pt) Fill in blank (k).

○ `''`

○ `None`

○ `first`

○ `repr(first)`

○ `max(rest)`

○ `repr(max(rest))`

○ `longest(ds[1:])`

vi. (1.0 pt) Fill in blanks (l) and (m) with the same expression. Numbers on a domino must be below 10.

○ `len`

○ `max`

○ `str`

○ `repr`

○ `lambda s: s`

○ `lambda s: s[0]`

○ `lambda s: len(s[0])`

(d) (4.0 points)

The `Domino` table has one row per domino that gives its unique `id` (number) and the numbers appearing on the domino `m` and `n` (numbers).

```
CREATE TABLE dominos AS
  SELECT 0 AS id, 3 AS m, 2 AS n UNION
  SELECT 1        , 3       , 1       UNION
  SELECT 2        , 6       , 3       UNION
  SELECT 3        , 4       , 4       UNION
  SELECT 4        , 3       , 2       UNION
  SELECT 5        , 2       , 6;
```

Complete this query that creates one row per domino for which at least one other domino has a number equal to its `n` value. The row contains the domino's `id`, `m` value, `n` value, and the count of other dominos that have a number equal to its `n` value.

```
SELECT a.id, a.m, a.n, COUNT(*) FROM dominos AS a, dominos AS b

  WHERE _____ AND (_____)
          (o)           (p)

  GROUP BY _____;
            (q)
```

i. (1.0 pt) Fill in blank (o).

○ a.id = b.id

○ a.id < b.id

○ a.id != b.id

ii. (2.0 pt) Fill in blank (p).

○ a.m = b.m OR a.n = b.n

○ a.m = b.m AND a.n = b.n

○ a.n = b.m OR a.n = b.n

○ a.n = b.m AND a.n = b.n

○ a.m = b.n OR a.n = b.n

○ a.m = b.n AND a.n = b.n

iii. (1.0 pt) Fill in blank (q).

○ m

○ a.m

○ b.m

○ id

○ a.id

○ b.id

4. **(12.0 points)   Merry-Go-Round**

  (a) **(6.0 points)**

   Implement `cycle`, which takes a non-empty finite linked list `s` (a `Link` instance). It returns a linked list in which the first `Link` of `s` has been moved to become the last `Link` of the result. If `s` has only one element, then it is not moved. The `Link` class appears on Page 2 of the Midterm 2 Study Guide. A linked list is finite if it contains a `Link` that has `Link.empty` as its `rest` attribute rather than having a cyclical structure.

```
def cycle(s):
    """Move the first Link of a non-empty finite linked list to be the last.

    >>> a = Link(3, Link(5, Link(7, Link(9))))  # <3 5 7 9>
    >>> print(cycle(a))
    <5 7 9 3>
    >>> print(a)
    <3>
    """
    assert isinstance(s, Link)
    if s.rest is Link.empty:
        return s
    last = s
    while last.rest is not Link.empty:
        last = last.rest
    s.rest, last.rest, result = _____ , _____ , _____
                                  (a)        (b)        (c)
    return result
```

   i. **(2.0 pt)** Fill in blank (a).

   ○ `s`

   ○ `s.rest`

   ○ `last`

   ○ `last.rest`

   ○ `Link.empty`

   ii. **(2.0 pt)** Fill in blank (b).

   ○ `s`

   ○ `s.rest`

   ○ `last`

   ○ `last.rest`

   ○ `Link.empty`

iii. **(2.0 pt)** Fill in blank (c).

○ `s`

○ `s.rest`

○ `last`

○ `last.rest`

○ `Link.empty`

(b) **(6.0 points)**

Implement `repeat`, a generator function that takes a non-empty finite linked list `s` and a positive integer `k`. It yields `k` elements from `s`, restarting from the first element of `s` each time the end is reached. It is ok to mutate `s`. You may call `cycle`.

```python
def repeat(s, k):
    """Yield k items from a non-empty finite linked list, starting over as needed.
    It is ok to mutate s in the process.

    >>> t = repeat(Link(4, Link(2, Link(6))), 10)
    >>> [next(t), next(t), next(t), next(t)]
    [4, 2, 6, 4]
    >>> list(t)  # 6 of the 10 elements remain
    [2, 6, 4, 2, 6, 4]
    """
    for x in _____:
                (d)
        yield _____
               (e)

           _____
             (f)
```

i. **(2.0 pt)** Fill in blank (d).

○ s

○ iter(s)

○ next(s)

○ range(k)

○ range(s)

○ range(len(s))

ii. **(2.0 pt)** Fill in blank (e).

○ s

○ s.first

○ s.rest.first

○ from repeat(s.rest, k-1)

○ x

○ s[x]

○ s[x % len(s)]

iii. **(2.0 pt)** Fill in blank (f).

○ cycle(s)

○ s = cycle(s)

○ cycle(s.rest)

○ s = cycle(s.rest)

○ s.rest = cycle(s.rest)

5. (11.0 points)   Trees

(a) (4.0 points)

Implement `count_ones`, which takes a `Tree` instance `t`. It returns the number of nodes in `t` that have one child. The `Tree` class is defined on Page 2 of the Midterm 2 study guide.

```
def count_ones(t):
    """Return the number of nodes that have one child.

    >>> count_ones(Tree(1, [Tree(2), Tree(3, [Tree(4)])]))
    1
    >>> count_ones(Tree(1, [Tree(2), Tree(3), Tree(4)]))
    0
    >>> count_ones(Tree(1, [Tree(2, [Tree(3, [Tree(4)]), Tree(5, [Tree(6)])])]))
    3
    """
    count = sum([ count_ones(_____) for b in t.branches if _____ ])
                               (a)                              (b)
    if _____:
         (c)
        return count + 1
    else:
        return count
```

i. (1.0 pt) Fill in blank (a).

○ b

○ t

○ b.branches

○ t.branches

ii. (1.0 pt) Fill in blank (b).

○ True

○ is_leaf(b)

○ is_leaf(t)

○ len(b.branches) == 1

○ len(t.branches) == 1

iii. (2.0 pt) Fill in blank (c).

○ True

○ is_leaf(b)

○ is_leaf(t)

○ len(b.branches) == 1

○ len(t.branches) == 1

○ all([len(b.branches) == 1 for b in t.branches])

○ any([len(b.branches) == 1 for b in t.branches])

(b) **(7.0 points)**

Implement `shorten`, which takes a `Tree` instance `t`. It removes all non-root nodes that have no siblings (no other nodes with the same parent), then returns `t`.

```
def shorten(t):
    """Remove all non-root nodes with no siblings and return t.

    >>> shorten(Tree(1, [Tree(2), Tree(3, [Tree(4)])]))
    Tree(1, [Tree(2), Tree(3)])
    >>> shorten(Tree(1, [Tree(2, [Tree(3, [Tree(4)])])]))
    Tree(1)
    >>> shorten(Tree(1, [Tree(2, [Tree(3), Tree(4)])]))
    Tree(1, [Tree(3), Tree(4)])
    >>> shorten(Tree(1, [Tree(2, [Tree(3, [Tree(4), Tree(5)]), Tree(6, [Tree(7)])])]))
    Tree(1, [Tree(3, [Tree(4), Tree(5)]), Tree(6)])
    """
    _____:
        (d)

        _____ = _____
          (e)        (f)
    for b in t.branches:
        shorten(b)
    return t
```

i. **(3.0 pt)** Fill in blank (d). Include the type of the control statement you are using (`if`, `while`, `for`, etc.).

ii. **(1.0 pt)** Fill in blank (e).

  ○ `t`

  ○ `t.branches`

  ○ `t.branches[i]`

  ○ `Tree(t.label, t.branches)`

iii. **(2.0 pt)** Fill in blank (f). You may **not** use `and`, `or`, `if`, or `lambda` in your answer.

iv. **(1.0 pt)** What order of growth describes the time it takes to execute `u = Tree(1, [t])` in terms of the number of nodes in a `Tree` instance `t`. Assume every non-leaf node in `t` has two children.

  ○ constant

  ○ logarithmic

  ○ linear

  ○ quadratic

  ○ exponential

6. **(6.0 points)**   **Scheme**

A nested list of numbers is a list containing numbers and nested lists of numbers. For example: ((7 5) 3 ((1)) 9)

Implement `truths`, which takes a nested list of numbers `s` and a one-argument procedure `f`. It returns the count of numbers `x` in `s` for which `(f x)` returns a true value.

```
scm> (define (eight? x) (and (number? x) (= x 8)))
scm> (truths '(8 (8 ((3 8)) 8 ())) eight?)
4
scm> (truths '(8 (8 ((3 8)) 8 ())) number?)
5
```

Hint: The built-in `number?` procedure returns whether its argument is a number.

```
(define (truths s f)
  (if (null? s) 0
    (+
       (if (number? (car s)) _____ (truths _____ f))  ; first argument to +
                             (a)               (b)
       _____ )))                                         ; second argument to +
          (c)
```

(a) **(3.0 pt)** Fill in blank (a).

[blank answer box]

(b) **(1.0 pt)** Fill in blank (b).

○ s

○ (car s)

○ (cdr s)

○ (list s)

(c) **(2.0 pt)** Fill in blank (c).

○ nil

○ 0

○ 1

○ f

○ s

○ (f s)

○ truths

○ (truths (cdr s) f)

(d) (0.0 pt) This A+ question is not worth any points. It can only affect your course grade if you have a high A and might receive an A+. Finish the rest of the exam first!

Fill in blank (d) to implement `eights`, which returns the number of eights that appear in a row at the beginning of a nested list `s`.

```
scm> (eights '(8 8 8 3 8))
3
scm> (eights '((8 (8) 8) (3) 8 ()))
3
scm> (eights '(8 (8 ((8 3 8)) 8 ())))
3
scm> (eights '(7 (8 ((3 8)) 8 ())))
0
```

You may call `truths`. The **only** special form you may use to fill in the blank is `if`.

```
;;; Return the number of eights at the start of a nested list of numbers s.
(define (eights s)
  (cond
    ((null? s) 0)
    ((eight? (car s)) (+ 1 (eights (cdr s))))
    ((number? (car s)) 0)
    (else (let ((k (eights (car s)))) (+ k _____ ) ))))
                                               (d)
```

7. (0.0 points) Tree Swap

This A+ question is not worth any points. It can only affect your course grade if you have a high A and might receive an A+. Finish the rest of the exam first!

Implement `swap`, which takes a `Tree` instance `t` with unique node labels and two different values `x` and `y` that are labels of `t`. It mutates the tree by placing the node labeled `x` in the position of the node labeled `y`, and the node labeled `y` in the position of the node labeled `x`.

Hint: The built-in `exec` function takes a string and executes the Python statement it contains in the current environment. For example:

```
>>> exec('x = 1 + 1')
>>> x
2
```

There are two blanks in `swap`, but they both contain exactly the same code, so you just need to write that code once.

```python
def find(t, x, s, f):
    result = []
    def g(t, s):
        if t.label == x:
            result.append(s)
        for i in range(len(t.branches)):
            g(t.branches[i], s + f(i))
    g(t, s)
    assert len(result) == 1, 'x should appear as a label exactly once'
    return result[0]


def swap(t, x, y):
    """Swap the positions of the nodes labeled x and y in Tree t.

    >>> n = Tree(1, [Tree(2, [Tree(3, [Tree(4)]), Tree(5)]), Tree(6, [Tree(7)])])
    >>> b = n.branches[0].branches[0]
    >>> b is n.branches[1]
    False
    >>> swap(n, 3, 6)
    >>> n
    Tree(1, [Tree(2, [Tree(6, [Tree(7)]), Tree(5)]), Tree(3, [Tree(4)])])
    >>> b is n.branches[1]
    True
    """
    px = find(t, x, _____ )
    py = find(t, y, _____ )
    exec(px + ',' + py + '=' + py + ',' + px)
```
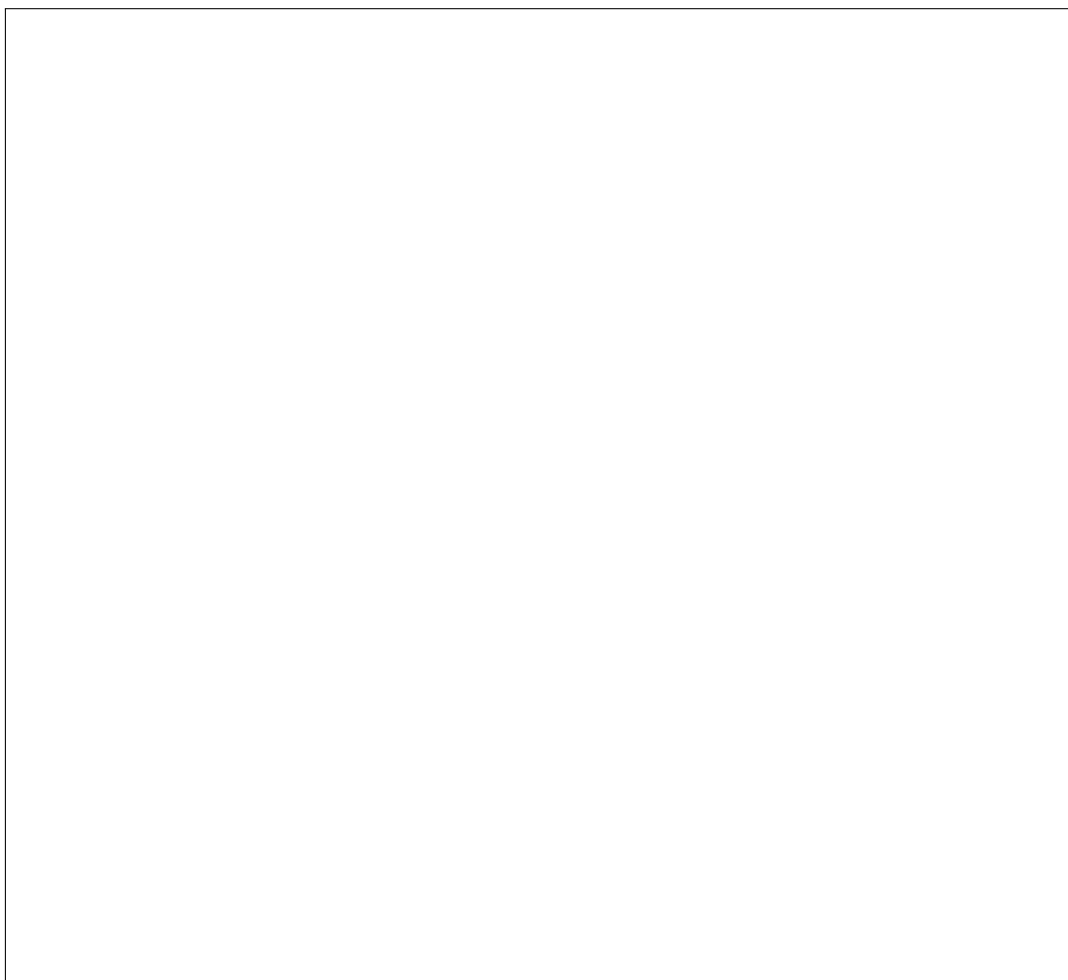
(a) Write the code that fills in both blanks. (The same code fills in each blank.) The `find` function takes four arguments, so your answer should contain two expressions separated by a comma.

```



```

8. **Art Break**

    (a) Draw a picture or write a poem: if the 61A homework bot took on a physical form, what would it look like?

No more questions.