6.101 Midterm

Fall 2024

- You have **110 minutes** to complete this exam. There are **4 problems**.
- The exam is **closed-book** and closed-notes, but you are allowed to bring a single 8.5×11" double-sided page of notes, handwritten directly on the paper (not computer-printed or photocopied), readable without a magnifying glass, created by you.
  - If you bring a handwritten page of notes, your name should be on the page, and you should **hand in your notes page** to a staff member at the end of the exam.
- You may also use blank scratch paper.
- You may use **nothing else on your computer** or other devices: no 6.101 website; no Python or programming tools; no web search or discussion with other people.
- Before you begin: you must **check in** by having the course staff scan the QR code at the top of the page.
- This page **automatically saves your answers** as you work. If you see a stuck yellow spinner, red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the exam.
- If you feel the need to **write a note to the grader**, you can click the gray pencil icon to the right of the answer.
- If you have a question, or need to use the restroom, please raise your hand.
- If you find yourself bogged down on one part of the exam, remember to keep going and work on other problems, and then come back.
- To **leave early**: enter *done* at the very bottom of the page, show your screen with the check-out code to a staff member, and give the staff member your handwritten page of notes (if any).
- You **may not discuss** details of the exam with anyone other than course staff until exam grades have been assigned and released.

Good luck!

Problem ⤫1

Recall from the audio processing lab that we represented sound recordings in two ways:

(you can **open this code in a separate tab**)

```
# mono sound
mono1 = {
  "rate": 8000, # samples per second
  "samples": [ 1.0, 0.97, 0.67, 0.31, -0.10 ],
}

# stereo sound
stereo1 = {
  "rate": 8000, # samples per second
  "left": [0.00, 0.59, 0.95, 0.95, 0.59, 0.00, -0.59, -0.95, -0.95, -0.59],
  "right": [1.00, 0.91, 0.67, 0.31, -0.10, -0.50, -0.81, -0.98, -0.98, -0.81],
}
```

This problem introduces a third representation:

```
# multitrack sound
multi1 = {
  "rate": 8000, # samples per second
  "tracks": {
    "vocals":   [0.00, 0.59, 0.95, 0.95, 0.59, 0.00, -0.59, -0.95, -0.95, -0.59],
    "keyboard": [1.00, 0.91, 0.67, 0.31, -0.10, -0.50, -0.81, -0.98, -0.98, -0.81],
    "drums":    [0.00, 0.00, 0.00, 0.00, 1.00, 1.00, 1.00, 1.00, 0.00, 0.00],
  }
}
```

In a multitrack sound, there may be any number of tracks in the `"tracks"` dictionary, with arbitrary names. They might represent specific instruments, different singers, different microphones in a room, etc.

Consider the functions below.

```
def stereo_to_multitrack(stereo):
  """
  Assuming stereo is a stereo sound, converts it to a multitrack sound with tracks named "left" and "r
  """
  return {
    "rate": stereo["rate"],
    "tracks": {
      "left": stereo["left"],
      "right": stereo["right"]
    }
  }

def mute_track(multitrack, track):
  """
  Mutes just one track in a multitrack sound by replacing all its samples with zeroes.
  """
  samples = multitrack["tracks"][track]
  for i in range(len(samples)):
    samples[i] = 0
```

Use `stereo_to_multitrack()` and `mute_track()`, together with any of the example sounds defined above (`mono1`, `stereo1`, `multi1`) to demonstrate a bug caused by *aliasing*.

Write a few lines of Python code that has the aliasing bug. Your answer should fit in the box without scrolling.

Give a brief explanation of the bug. Your answer should fit in the box without scrolling.

Describe a change to either `stereo_to_multitrack()` or `mute_track()` that would prevent this bug. Your answer should fit in the box without scrolling.

Consider the helper function below:

```
def get_samples(multitrack, track):
  """
  Gets the sample list for a given track. For example:

  >>> get_samples(multi1, "drums")
  [0.00, 0.00, 0.00, 0.00, 1.00, 1.00, 1.00, 1.00, 0.00, 0.00]
  """
  return multitrack["tracks"][track]
```

In a multitrack sound, it's often the case that one or more tracks is completely silent (sample `0.00`) for long stretches of the recording. Anyone who has played the triangle in an orchestra may be familiar with this phenomenon.

Because of this, you decide to change your multitrack representation so that it omits streaks of `0.00` from the sample lists, and just keeps track of the sample index where each streak of non-zeroes starts. So the old representation `multi1` becomes `multi2` below:

(you can **open this code in a separate tab**)

```
# old representation of multitrack sound
multi1 = {
  "rate": 8000, # samples per second
  "tracks": {
    "vocals":   [0.00, 0.59, 0.95, 0.95, 0.59, 0.00, -0.59, -0.95, -0.95, -0.59],
    "keyboard": [1.00, 0.91, 0.67, 0.31, -0.10, -0.50, -0.81, -0.98, -0.98, -0.81],
    "drums":    [0.00, 0.00, 0.00, 0.00, 1.00, 1.00, 1.00, 1.00, 0.00, 0.00],
  }
}

# new representation of the same multitrack sound, keeping only the nonzero samples
multi2 = {
  "rate": 8000, # samples per second
  "tracks": {
    "vocals": {
      1: [0.59, 0.95, 0.95, 0.59],
      6: [-0.59, -0.95, -0.95, -0.59],
    },
    "keyboard": {
      0: [1.00, 0.91, 0.67, 0.31, -0.10, -0.50, -0.81, -0.98, -0.98, -0.81],
    },
    "drums": {
      4: [1.00, 1.00, 1.00, 1.00],
    },
  }
}
```

Rewrite `get_samples()` so that it uses the new representation of multitrack sounds, behaving the same way on `multi2` that the old version did on `multi1`. Your rewrite doesn't need to handle the old representation, just the new one.

```
def get_samples(multitrack, track):
  """
  Gets the sample list for a given track, using the new multitrack representation. For example:

  >>> get_samples(multi2, "drums")
  [0.00, 0.00, 0.00, 0.00, 1.00, 1.00, 1.00, 1.00, 0.00, 0.00]
  """
```

The famous piece *4'33"* by John Cage consists of 4 minutes and 33 seconds of silence:

```
# old representation of 4'33"
cage1 = {
  "rate": 1, # samples per second
  "tracks": {
    "silence": [0.00] * (4*60 + 33)      # a list of 273 zeroes
  }
}
```

To represent this recording in the new multitrack representation, Louis Reasoner suggests the following:

```
# Louis's proposed representation of 4'33"
cage_louis = {
  "rate": 1, # samples per second
  "tracks": {
    "silence": { } # empty dictionary
  }
}
```

Briefly explain what your code for `get_samples()` does when given `get_samples(cage_louis, "silence")`. Mention values of local variables, and describe the final return value. You don't have to change your code, just describe what your code does. Your answer should fit in the box without scrolling.

Alyssa Hacker points out that Louis's version omits an essential piece of information. She suggests this instead:

```
# Alyssa's representation of 4'33"
cage_alyssa = {
  "rate": 1, # samples per second
  "tracks": {
    "silence": {
      273: []     # 273 seconds = 4 minutes and 33 seconds
    }
  }
}
```

Briefly explain what your code for `get_samples()` does when given `get_samples(cage_alyssa, "silence")`. Mention values of local variables, and describe the final return value. You don't have to change your code, just describe what your code does. Your answer should fit in the box without scrolling.

Problem ⋊2

Here is the `find_path()` function defined in the course readings and used in recitation, with one difference: it now takes a *strategy* argument that specifies how the search uses the agenda.

(you can **open this code in a separate tab**)

```python
def find_path(neighbors_function, start, goal_test, strategy):
  """
  Find a path through a state graph defined by `neighbors_function`,
  starting from the `start` state and reaching a state satisfying
  `goal_test`.

  Note that all state representations must be hashable.

  neighbors_function: function that takes a state and returns its
                      neighbors (as an iterable)
  start: starting state
  goal_test: function that takes a state and returns True (or a truthy value)
      if and only if the state satisfies the goal condition
  strategy: function that takes an agenda and removes and
      returns the next path to explore.  Typical strategies
      are `dfs` and `bfs`, defined below.

  Returns the path of states from start to a goal state,
  or None if no path exists
  """
  if goal_test(start):
    return (start, )

  agenda = [(start, )]
  visited = {start}

  while agenda:
    this_path = strategy(agenda)
    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)

  return None
```

We discussed two kinds of strategy: depth-first or breadth-first. Define these strategies as functions, `dfs` and `bfs`, which could be passed as the `strategy` argument of `find_path`. Your answers should fit in the boxes without scrolling.

```
def dfs(agenda):
  """
  Implements a strategy that, when passed as the strategy argument to find_path(),
  causes it to explore the state space depth-first.
  """
```

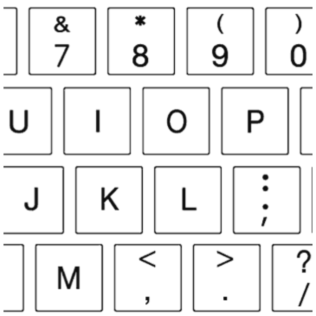
```
def bfs(agenda):
  """
  Implements a strategy that, when passed as the strategy argument to find_path(),
  causes it to explore the state space breadth-first.
  """
```

Now, using this version of `find_path()` (including the strategies `dfs` and/or `bfs`), implement a spelling corrector for an alien language. The aliens are excellent spellers, but they use keyboards like ours, so when they make a typing error, it's because their tentacle pressed a key adjacent to the key they intended, substituting the adjacent key's letter in place of the right one. They might make multiple typing errors within the same word, but every error will be adjacent to the intended key.

You don't know the alien language or the alien keyboard, but your function is given two inputs with the information you need:

- a `lexicon` function that tells you whether a given word is in the alien language or not;
- a `nearby_keys` function that tells you which letters are adjacent to a given letter on the alien keyboard.

To use more familiar examples, if the lexicon is English and the nearby keys are taken from the standard QWERTY keyboard (part of which is shown below), then your spelling corrector should be able to turn `"helli"` into `"hello"`.



Fill in the box below with the body of the `correct()` function. Feel free to define helper functions inside it. Again, you **must use** `find_path()`.

```
def correct(typed_word, lexicon, nearby_keys):
  """
  Returns any legal word found in the alien lexicon that
  can be reached from typed_word using the fewest number of
  substitutions of nearby_keys on the alien keyboard.
  Returns None if no legal word can be found.

  typed_word: string
  lexicon: function that takes a word and returns True if and only if
          it is an actual word in the alien language.
  nearby_keys: function that takes a letter and returns the letters adjacent to it
          on the alien keyboard (as an iterable)

  >>> correct('hello', english_lexicon, qwerty_keyboard)
  'hello'
  >>> correct('helli', english_lexicon, qwerty_keyboard)
  'hello'
  """
```

Is it possible for your code to correct `"hellm"` into `"hello"` using the English lexicon and the standard QWERTY keyboard? Given the problem description stated above, *should* this be an acceptable correction? Explain briefly. Your answer should fit in the box without scrolling.

For full credit, make sure your code does the right thing for `"hellm"` and `"hello"`, and say “code is correct” in the box below. Alternatively, if your code doesn’t do the right thing, use the box below to describe how you could change it.

Problem ⋊3

This problem explores variations of the following function:

(you can **open this code in a separate tab**)

```
def intersect(a, b):
  """
  given two lists of numbers (each containing no repeats), returns the numbers that are in both a and

  >>> intersect([0,1,2,3],[4,1])
  [1]
  >>> intersect([],[4,1])
  []
  """
  out = []
  for x in a:
    if x in b:
      out.append(x)
  return out
```

Rewrite `intersect()` using a list comprehension.

```
def intersect(a, b):
```

Instead of a list comprehension, let's compare some other ways to write `intersect()`, using the following example call:

```
small = list(range(5))
big = list(range(10_000_000))
intersect(small,big)
```

Suppose the code is changed as shown below:

```
# original version
def intersect(a,b):
  out = []
  for x in a:
    if x in b:
      out.append(x)
  return out
```

```
# new version
def intersect(a,b):
  out = []
  for x in b:
    if x in a:
      out.append(x)
  return out
```

Will this new version run `intersect(small,big)` *much faster*, *much slower*, or *about the same* as the original version?

Explain briefly. Your answer should fit in the box without scrolling.

Suppose the code is changed as shown below:

```python
# original version
def intersect(a,b):
  out = []
  for x in a:
    if x in b:
      out.append(x)
  return out
```

```python
# new version
def intersect(a,b):
  out = []
  for x in a:
    if x in set(b):
      out.append(x)
  return out
```

Will this new version run `intersect(small,big)` *much faster*, *much slower*, or *about the same* as the original version?

Explain briefly. Your answer should fit in the box without scrolling.

Suppose the code is changed as shown below:

```python
# original version
def intersect(a,b):
  out = []
  for x in a:
    if x in b:
      out.append(x)
  return out
```

```python
# new version
def intersect(a,b):
  out = []
  seta = set(a)
  setb = set(b)
  for x in seta:
    if x in setb:
      out.append(x)
  return out
```

Will this new version run `intersect(small,big)` *much faster*, *much slower*, or *about the same* as the original version?

Explain briefly. Your answer should fit in the box without scrolling.

Finally, let's consider a recursive approach to `intersect()`. Suppose the Python library gives you a fast way to split a list in half:

```
def halve(lst):
  """
  divides an input list into a pair of lists (first_half, second_half)
  where first_half + second_half == lst
  and first_half is either the same length or 1 element longer than second_half

  >>> halve([0,1,2,3])
  ([0,1],[2,3])
  >>> halve([4])
  ([4],[])
  """
```

Use this function to write `intersect()` recursively, filling in the placeholders in the skeleton below. Your answers should not iterate over `a` or `b` or any other list, but may use `len()` or list indexing. Your answers to the recursive cases must use the provided local variables a1,a2,b1,b2 apropriately.

```
def intersect(a, b):
  """
  given two lists of numbers (each containing no repeats), returns the numbers that are in both a and

  >>> intersect([0,1,2,3],[4,1])
  [1]
  >>> intersect([],[4,1])
  []
  """
  # base case(s) (should not iterate over a or b)
```

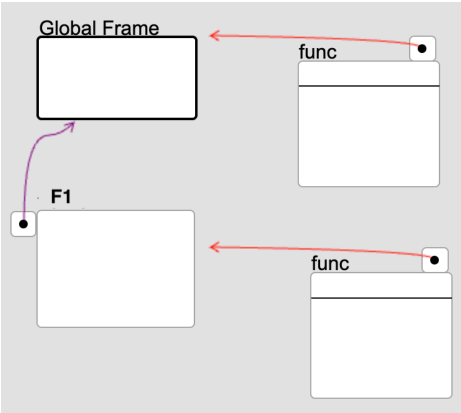
```
  if len(a) >= len(b):
    # recursive case A (should not iterate over a or b)
    a1,a2 = halve(a)
```

```
  else:
    # recursive case B (should not iterate over a or b)
    b1,b2 = halve(b)
```

Problem ⤫4

(you can **open this preamble in a separate tab**)

All the environment diagrams in this problem are based on the following template:

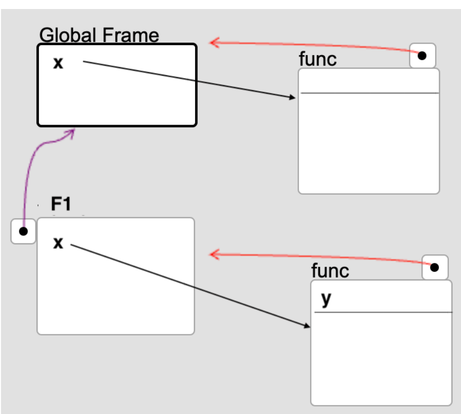


- two frames:
  - the global frame;
  - a frame F1 whose parent is the global frame.
- two function objects, one above the other:
  - the top function's enclosing frame is the global frame;
  - the bottom function's enclosing frame is F1.

Each diagram may add variables, numbers, and/or lists to this basic template. Assume every frame or object created remains in the diagram (the garbage collector hasn't removed anything yet).

All these diagrams intentionally omit function bodies and return-value pointers, so the code inside the functions and their return values can be whatever you need them to be.

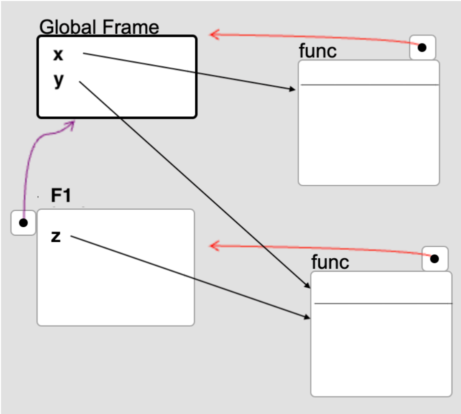
Write Python code that would produce an environment diagram in which:



- the global frame contains only:
  - x pointing to the top function
- frame F1 contains only:
  - x pointing to the bottom function
- the top function has no parameters
- the bottom function has a parameter y

Your answer should fit in the box without scrolling.

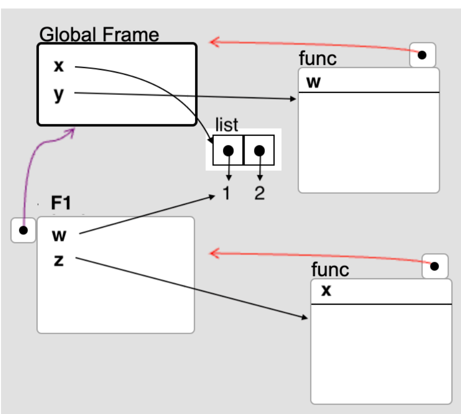
Write Python code that would produce an environment diagram in which:



- the global frame contains only:
  - x pointing to the top function
  - y pointing to the bottom function
- frame F1 contains only:
  - z pointing to the bottom function
- the top function has no parameters
- the bottom function has no parameters

Your answer should fit in the box without scrolling.

Write Python code that would produce an environment diagram in which:



- the global frame contains only:
  - x pointing to a two-element list, whose slots point to the integers 1 and 2
  - y pointing to the top function
- frame F1 contains only:
  - `w` pointing to the same integer 1 just mentioned
  - z pointing to the bottom function
- the top function has a parameter `w`
- the bottom function has a parameter x

Your answer should fit in the box without scrolling.