CS 1110, Spring 2022: Prelim 2 study guide

Prepared by Prof. L. Lee Monday April 11, 2022

Update Apr 14 morning: added "Student should be able to" appendix.
Update Apr 14 evening: note about Spring 2021 classes question.

Table of Contents

Topic coverage

The prelim covers material from lectures 1-18 inclusive (start of course until Tue Apr 12 inclusive), assignments A1-A5 and labs 01-16, with any exceptions noted below. Emphasis will be on material not tested on prelim 1.

Subclasses and inheritance will *not* be tested on Prelim 2.

A reference sheet for some functions and methods will be provided; exact contents to be announced later.

On the exam, you may not use Python we have not introduced in class (lectures, assignments, labs, readings).

The exam is to test your understanding of what we have covered in class.

On the exam, if you are asked to solve a problem a certain way, answers that use a different approach may receive no credit.

In particular, if we say you must make effective use of recursion, the question is to test your understanding of recursion, so a solution that is *essentially* just a for-loop or while-loop may well receive a score of 0. (You may be allowed to use a for-loop *in conjunction with* recursion, but if recursion is requested, then recursion must be the core of your solution/). Similarly, if we say you must use a loop, then map() or recursion is not allowed as the core of your solution, and so on.

Recommendations for preparing

Our recommendations from the Prelim 1 study guide still hold. We especially emphasize:

- *Work coding problems at your computer (including perhaps Python Tutor)*, not on paper. Fluency at the keyboard will translate to fluency on paper, and most beginning students *need* to start with the feedback that Python gives.
- Test your solutions by writing your own testing code --- at least include the examples given in function specifications --- or using any we provide; see the Exams archive page for previous Prelim 2 problems and solutions.

Making your own testing code

You can make quick-and-dirty testing code as follows. Say you're writing a function prelim2, and the spec says that prelim2("hi") returns "there" and prelim2(["hi", "there"]) returns the list ["no", "fair"].

Then, add to the bottom of the file, **non-indented**, something like the following:

Option 1 – simplest, but a little tedious

Write each test out as its own little block of code.

```
print("Testing input 'hi'")
expected = 'there'
result = prelim2("hi")
assert result == expected, "Error: Expected "+repr(expected)+" but got "+repr(result)

print('Testing input["hi", "there"]')
expected = ["no", "fair"]
result = prelim2(["hi", "there"])
 assert result == expected, "Error: Expected "+repr(expected)+" but got "+repr(result)
```

Option 2 – better if you have multiple tests

Better is to loop through a list consisting of input/output pairs.

```
tests = [
   ["hi", "there"],
   [["hi", "there"], ["no", "fair"]]
        ]

for [theinput, expected] in tests:    # convenient shorthand
    print("Testing input "+repr(theinput)")
    result = prelim2(theinput)
    assert result == expected, "Error: Expected "+repr(expected)+" but got "+repr(result)
```

Strategies for answering coding questions (same text as in the Prelim 1 study guide)

.

1. When asked to write a function body, always first read the specifications carefully: what are you supposed to return? Are you supposed to alter any lists or objects? What are the preconditions? Do you understand the given examples/test cases? I*f you aren't sure you understand a specification, ask.*

2. For this semester, do NOT spend time writing code that checks or asserts preconditions, in the interest of time. That is, don't worry about input that doesn't satisfy the preconditions.
3. After you write your answer, double-check that it gives the right answers on the test cases --- any we give you, plus any you think of. Also, double check that what your code returns on those test cases satisfies the specification.[1]
4. Comment your code if you're doing anything unexpected. But don't overly comment - you don't have that much time.
5. Use variable names that make sense, so we have some idea of your intent.
6. If there's a portion of the problem that you can't do and a part that you *can*, you can try for partial credit by having a comment like

   ```
   # I don't know how to do <x>, but assume that variable `start`
   # contains ... <whatever it is you needed>"
   ```

   That way you can use variable start in the part of the code you know how to do.

Notes on questions from prior exams and review materials

In general

1. Fall questions for which one-diagram-drawn-per-line notation is used (i.e., the very, very long folder/call-frame solutions) would need to be converted to our one-frame-per-function notation. **Do not use the fall notation on spring exams --- so skip the Fall diagram questions.**

   In general, Spring 2015 and Spring 2016 are quite different than what one can expect for this semester's exams. (If you do look at them, note that they use different variable naming conventions from what we use: we would reserve capital letters for class names, and use more evocative variable names.)

2. In general, Fall class and sub-class questions have included sub-problems involving implementing getters and setters, mutable vs. immutable attributes, and asserting preconditions. We will not have such sub-problems, but other parts of the class ~~and sub-class~~ questions are fair game.

3. Where you see lines of the form "if __name__ == '__main__':", think of them as indicating that the indented body underneath it should be executed for doing the problem.

4. Before Fall 2017, the course was taught in Python 2; perhaps the biggest difference this makes in terms of the relevance of previous prelims is that questions regarding division (/) need to be rephrased. Also, python2's print didn't require parentheses and allowed you to give multiple items of various types separated by commas (which would print as spaces). In some cases, instances of range() in a Python 2 for-loop header might need to be replaced with list(range()),

---

[1] It seems to be human nature that when writing code, we focus on what the code *does* rather than what the code was *supposed* to do. This is one reason we so strongly recommend writing test cases before writing the body of a function.

and similarly for map() and filter().

5. Another difference for Python 3 is that one can omit "object" from inside the parentheses in the header of class definitions and the class will still be a subclass of object.

6. You may notice that many solutions check whether something is None by "if x is None:" rather than "x==None:". We haven't discussed this in Spring 2022 (yet), but the former is preferred.

Previous prelim 2s

2021 Fall

We had access to the source files, so some comments have been incorporated in orange in the pdfs posted to the Spring 2022 Exam Archive page.

- Q4:
  - Skip subclass Choice (Spring 2022 Prelim 2 will not cover subclasses).
  - Skip writing setters/getters
  - In Spring 2022, you *should* be able to correctly initialize class attribute Question.USED_INDICES and write the __init__, __str__, and __eq__ methods.
- Q5(a): skip subclass B (Spring 2022 Prelim 2 will not cover subclasses)
- Q5(b): skip (different notation for call frames, and Spring 2022 Prelim 2 will not cover subclasses)

2021 Spring:

- I like Q2. We should have asked for an explanation for the last part.
- Q4: **If** you are able to do class-related questions on other past prelims, and are finding this one difficult simply because you don't have good intuition about what the classes Shell and File are supposed to represent, it is OK to skip this question!

  In terms of having intuitions about what this question is asking for, a key line is this on in the specification for class Shell:

  "Objects represent an instance of a command shell
  (think Terminal for MacOS or Powershell on Windows".

  Think about when you open a Terminal/Powershell window. That window is viewing the currently open directory, hence the attribute openDir.

  And you can change directories by typing `cd CS1110` (say). That's why there's a Shell method called cd().

  Inside a directory would be different files. That's why the class File exists.

  The addFile() method for Shell was, in part, our way of testing whether students understood how to create a new object of a given class given the specification for that class's __init__() method.

Q4(b): The hint could have been written as "make sure you update the contents of the current open directory".

2020 Fall:

- Q2(a): make sure you can do this question without using the built-in sum() function.
  (At the same time, we suggest not using sum as a variable name, since that over-writes the built-in function.)
  Be aware that an assignment statement like lst = newlst would not actually change the input list, but only the value of the local parameter lst.

    - Alternate solutions. The one on the right takes advantage of the fact that the changed list element "just to the left" is already an accumulation!

```
sum_so_far = 0  # sum of lst from 0..i-1
   for i in range(len(lst)):
      lst[i] = sum_so_far + lst[i]
      sum_so_far = lst[i]
```

```
for i in range(1, len(lst)):
      lst[i] += lst[i-1]
```

- Q3(a): Assume the question also ruled out while-loops (but see loop-based solutions below, for the record)
    - the given solution makes more recursive calls than is necessary: if s[0] != s[1], there's no reason to compute prefix(s[1:]); and, at the point where there's an assignment to left via a recursive call, one already knows the string has length at least two, and that prefix(s[:1]) will always be 1, so there's no reason for that recursive call. (Also, typo "rhgt" should be right). Hence, alternate solution:

```
def prefix(s):
   if len(s) == 0:
      return 0
   elif len(s) == 1:
      return 1
   # if here, len(s) >= 2
   elif s[0] != s[1]: #
      return 1
   else:
      # if here, we know s[0] == s[1]; need to "save" s[1] for recursive call.
      # That is, return 2 + prefix(s[2:]) would make a mistake on 'aab' (as
      # well as the given test input xxxxxxyzx)
      return 1 + prefix(s[1:])
```

    - Interestingly(-ish), the recursive solution is arguably more elegant than a loop-based solution because one doesn't need to do as much "book-keeping". But here is a while-loop solution, even though loop-based solutions were (presumably) ruled out:

```
if len(s) <=1:
    return len(s)

  # if here, len(s) >= 2
  i = 1
  num_so_far = 1 # prefix length found in s[0..i-1]
  while i < len(s) and s[i] == s[0]:
    num_so_far+=1
    i += 1
  return num_so_far
```

A while-loop is arguably preferable to a for-loop (because with a for-loop one would have to use "break" or to go through more of the string that is necessary) so we do not provide a for-loop solution. (But an advantage of for-loops is that one doesn't have to remember to increment the index i.)

- Q3(b):
  - The hint could be rephrased as "pulling off one element at the start will most likely lead to a solution that is more complicated than a different way of dividing the string". **The hint is a good one.**
  - The given solution uses negative indexing. Negative indices are a convenient feature of Python (although they can lead to quite unexpected behavior if one isn't careful when using the find() string method, which returns -1 for "not found"), but other languages do not have this feature; this is a tradeoff that, honestly, causes us to waver every semester about whether to introduce them or not.

```
if len(s) == 0:
  return {}
elif len(s) == 1:
  return {s[0]: [0]}

mid = len(s)//2  # splitting in half.  Just pulling off one item from the start would be OK, too
left_res = invert(s[:mid])
right_res = invert(s[mid:])

result = {}
# add items in left_res
for c in left_res: # c is a character
  result[c] = left_res[c] # this is a list
  if c in right_res:
    right_list = right_res[c]
    # have to "shift" the indices in list right_list by mid
    for i in range(len(right_list)):
      result[c].append(right_list[i] + mid)
# add items in right_res but not left_res
for c in right_res:
  if c not in left_res: # which means c is not in result yet
    result[c] = []
    right_list = right_res[c]
    # have to "shift" the indices in list right_list by mid
    for i in range(len(right_list)):
      result[c].append(right_list[i] + mid)

return result
```

- while this is a legitimate question for applying recursion, I have a personal preference to not ask students to apply recursion when a loop would be the more "natural" solution. This question seems more suited to loops. Here is a loop-based solution:

```
result = {}
for i in range(len(s)):
    c = s[i]    # this is a character
    if c in result:
        result[c].append(i)
    else:
        result[c] = [i]
return result
```

Note that you wouldn't want a for-each loop here.

- Q4(a):
  - In Spring 2022, we wouldn't take off points for function "signatures" of the form __init__() instead of __init__(self,x) or f() instead of f(self, x) .
  - In Spring 2022, we haven't covered subclasses, so one would only be responsible for drawing the class folder for class A (which we would declare with class A: , no parentheses, although class A(object): and class A(): are fine, too).
- Q4(b): not applicable for Spring 2021 (we haven't covered subclasses yet)
- Q5(a):
  i. As noted in the "In General" section above, you can skip parts of the question dealing with getters and setters
  ii. In the header for class Date, this semester, you can omit the "(object) " part; class Date: suffices.
  iii. Typo in __init__(): "assert assert isinstance(y, int)" should be "assert isinstance(y, int)"
  iv. An argument could be made that "assert m in self.MONTHS" should be "assert m in Date.MONTHS", but self.MONTHS is fine. (If one wanted a subclass that could have its own MONTHS class attribute, then self.MONTHS would be preferable. If one didn't want to allow such a thing, Date.MONTHS would be more appropriate.)
  v. In Spring 2022, we aren't working with getters/setters, so the __init__ method's last assignment statement would be self._day = d and one would add the appropriate precondition assertion for d.
  vi. For the __lt__ method, for SP2022,
    1. We haven't covered raising errors yet, so you would not have to implement the causing of a TypeError.
    2. Replace self.getMonth() with self._month and similarly for the other "get" methods.

3. OK to have self.MONTHS instead of Date.MONTHS

- Q5(b: skip for Spring 2022 (we haven't done subclasses yet)

2020 Spring had no prelim 2

2019 Fall:

- Q2(b) alternate solution, with conditional expression for compactness

```
out = {}

for k in dict1:
   out[k] = dict1[k] + (0 if k not in dict2 else dict2[k])
for k in dict2:
   if k not in dict1:
      out[k] = dict2[k]

return out
```

- Q3(a), recursive clamp(). Assume the question also ruled out while-loops (but see loop-based solutions below, for the record).
  - The line "if len(alist):" is a typo; it should be "if len(alist) == 1:"
  - The problem as stated uses the names of pre-existing functions min and max as parameter names. There is a reason this makes sense for this specific exam problem (we wouldn't want students using the functions min() or max()), but in general, we recommend not using a variable name that is the same as some builtin. (So, avoid using max, min, list, sum, string, and so on as variable names.)
  - for-loops seem like a more natural solution to this question than recursion. You may have created your own loop-based solution in a previous lab.
- Q3(b):
  - for-loops seem like a more natural solution to this question than recursion.
  - Alternate solution, which makes fewer recursive calls (no need to run do recursion on text[:1], and which also makes use of the ability to assign to multiple components of a tuple simultaneously:

```
vowel_list = ['a', 'e', 'i', 'o', 'u']

if len(text) == 0:
   return ('', '')
if len(text) == 1:
   if text in vowel_list:
      return ('', text)
   else:
      return(text, '')

(consonants, vowels) = disemvowel(text[1:]) # trick: this can be empty
if text[0] in vowel_list:
   vowels = text[0] + vowels
else:
   consonants = text[0] + consonants

return (consonants, vowels)
```

- Q4:
  - As noted in the "In General" section above, you can skip parts of the question dealing with getters and setters. Replace self.set<whatever> and self.get<whatever> with direct accesses of self._<whatever>, and add assertions regarding preconditions to the __init__() method.
  - In the header for class Cornellian, this semester, you can omit the "(object) " part; class Cornellian: suffices.
  - In __eq__(), assume that the method should return False if it should not return True.
  - Skip the Student subclass for Spring 2022; we haven't covered subclasses yet. But: for the __init__() method of Student, you *should* know how to write a header that gives a default value for an optional parameter.
    - getGPA has a typo; it should return self._gpa

- Q5:
  - In Spring 2022, we wouldn't take off points for function "signatures" of the form __init__() instead of __init__(self,x,y) or f() instead of f(self, y) .
  - In Spring 2022, we haven't covered subclasses, so one would only be responsible for drawing the class folder for class A (which we would declare with class A: , no parentheses, although class A(object): and class A(): are fine, too).
  - In Spring 2022, skip part (b).

2019 Spring:

- Q1(b): "Within a given class … possibly many instance attributes named x" means, "for a given class, there can be many objects of that class that have different values for attribute x".
- Q2: assume you are not allowed to call the max() builtin function.
  - The given solution uses max as a local variable. That's OK for the given solution, which doesn't rely on the max() built-in function, but in general, we recommend not using a variable name that is the same as some builtin. (So, avoid using max, min, list, string, and so on as variable names.)
  - A for-loop seems more natural than recursion for this question.
- Q3. Alternate solution using nested for-loops. Note that `"x"*5` and `5*"x"` evaluate to the same thing.

```
for email in email_list:
    for i_word in range(len(email)):
        word = email[i_word]
        if token in word:
            email[i_word] = "x"*(len(word))
```

- Q4: skip for sp2022 (we haven't done subclasses yet)
- Q5:
  - __init__ method, with default value: there is some very unexpected behavior that can happen when using a default value of [] (or any other mutable default argument). See https://docs.python-guide.org/writing/gotchas/#mutable-default-arguments and https://docs.python.org/3/reference/compound_stmts.html#function-definitions, where the text says "A way around this is to use None as a default".

    So, **it is much better practice** to set the default value for parameter parents to be **None,** and then change the value of self.parents to be [] if the parameter parents has the value None:

```
def __init__(self, first, last, parents=None):
    self.first = first
    self.last = last
    Person.population += 1

    if parents is None:
        parents = []

    self.parents = []
    self.add_parents(parents)
    for p in parents:
        p.add_children([self])
    self.children = []
```

2018 Fall:

- Q2(a): alternate solution using join and list(<a string>), since lists are mutable

```
listversion = list(text)

for i in range(len(listversion)):
    c = listversion[i] # a character
    if c in cipher:
        listversion[i] = cipher[c]

return ''.join(listversion)
```

- Q2(b): we would give you the specification for dictionary method clear() .
- Q3(a): More natural would be a loop-based solution.
- Q3(b): recursion does mean less explicit book-keeping than in a loop-based solution. This question involves nested lists, but the nested lists are only one level deep. It's an instance of recursion involving nested lists for which the "for each subitem in the input list, apply recursion to the subitem" pattern does *not* necessarily seem the most natural approach. An alternate solution:

```
if text == '':
    return []
elif len(text) == 1:
    return [[text, 1]]
else:
    # len(text) >=2, so encode(text[1:]) will return a list with an item in it
    right = encode(text[1:])  # text[1:] might be the empty string
    first_right_c = right[0][0]
    if text[0] == first_right_c:
        right[0][1] += 1
        return right
    else:
        return [[text[0], 1]] + right
```

- Q4:
  - In Spring 2022, we wouldn't take off points for function "signatures" of the form __init__() instead of __init__(self,x) or f() instead of f(self, x) .
  - In Spring 2022, we haven't covered subclasses yet, so one would only be responsible for drawing the class folder for class A (which we would declare with class A: , no parentheses, although class A(object): and class A(): are fine, too).
  - In Spring 2022, skip part (b).
- Q5:
  - As noted in the "In General" section above, you can skip parts of the question dealing with getters and setters. Replace self.set<whatever> and self.get<whatever> with direct accesses of self._<whatever>, and add assertions regarding preconditions to the __init__() method.
  - In the header for class License, this semester, you can omit the "(object)" part; class License: suffices.
  - In License.__init__(), where the spec says "the pair (prefix, suffix)", read this as "the pair [prefix, suffix]".
  - Skip the Vanity subclass for Spring 2022; we haven't covered subclasses yet.
-

2018 Spring:

- Q3: it would have been better for the question to explicitly state that count_from could start negative, but it is OK as written.

- Q6: skip in Spring 2022

2017 Fall:

- Q2 pairswap:
  - Alternate solution:

```
for ind in range(len(nlist) – 1):  # will not include the last index
    if ind % 0 == 0:
        temp = nlist[ind]
        nlist[ind] = nlist[ind+1]
        nlist[ind+1] = temp
```

  - An alternative while-loop solution (for 2022 spring, you wouldn't have to write it~~, but you should be able to analyze it~~) is:

```
even_pos = 0  # Next even position to handle
while even_pos + 1 < len(nlist):
    # We know here exists a later item to swap with
    temp = nlist[even_pos]
    nlist[even_pos] = nlist[even_pos+1]
    nlist[even_pos+1] = temp

    even_pos += 2
```

- Q2 colavg: might be wise to add to specification that the table should not be altered.
  - Alternative solution:

```
sum_list = table[0][:]
for row in table[1:]:  # Add in all the other rows
    for i in range(len(row)):
        sum_list[i] += row[i]
n = len(table)
for i in range(len(sum_list)):
    sum_list[i] /= n
return sum_list
```

- Q3 segregate:

- alternate solution that does not use conditional expressions, but does use assignment to a tuple:

```
if len(nlist) == 0:
    return (-1, [])

head = nlist[0]
(tail_pos, outlist) = segregate(nlist[1:])
# tail_pos is start of nonnegs in initial outlist. We must now add the head to outlist.
if head < 0:
    outlist.insert(0, head)
    if tail_pos != -1:
        # There were non-negative numbers in outlist already
        return (tail_pos+1, outlist)
    else:
        return (-1, outlist)
else:
    # head is non-negative
    if tail_pos != -1:
        outlist.insert(tail_pos, head)
        return (tail_pos, outlist)
    else:
        # There were no non-negative numbers before
        return (len(outlist), outlist + [head])
```

- If you want to test your own implementation, here's some code you can use:

```
# keys are tuple versions of input lists.
test_cases = {(1, -1, 2, -5, -3, 0): (3, [-1, -5, -3, 1, 2, 0]),
    (-1, -3, -3): (-1, [-1, -5, -3]),
    (1, 5, 4): (0, [1, 5, 4]),
    (1, 2, 3, 4, -1, -5, -2): (2, [-1, -2, 1, 2, 3, 4, 5]),
    (): (-1, [])
}

for tc in test_cases:
    print('Testing ' + str(list(tc)))
    expected = test_cases[tc]
    result = segregate(list(tc))
    assert result == expected, "Got "+repr(result)+" instead of "+repr(expected)
```

- Q4: (a) skip the class folders for B and C for Spring 2022; (b) skip for Spring 2022
- Q5: we would say that you **do** need to provide specifications for any helpers.

- As noted in the "In General" section above, you can skip parts of the question dealing with getters and setters. Replace self.set<whatever> and self.get<whatever> with direct accesses of self._<whatever>, and add assertions regarding preconditions to the __init__() method.
- In the header for class File, this semester, you can omit the "(object) " part; class File: suffices.
- Spring 2022: skip the subclass

2017 Spring:

- Q1: some online versions have some weird stray (and incorrect) code included after the line
  ### Your implementation must make effective use of a for-loop.
  If you see that stray code, cross it out; and such versions also don't make it clear that: repeats *should* be included.
- Q2: assumes one has done A4 of 2017 Spring, so skipping this question certainly makes sense. But do observe that you should not be surprised for the exam to have questions that assume significant experience with this semester's assignments.
- Q3:
  - Assume that the direction of a train, and thus the direction that is "outward" or "facing out", is predetermined.
  - __str__: Some posted versions of the 2017 spring solutions have the line "text = "Domino" + …. .
    Replace "text = " with "return"
  - Alternate solution to addDomino:

```
def addDomino(self, d):
    dsides = [d.side1, d.side2]
    if self.outwardFacingSide not in dsides:
        return False
    elif not self.canExtend or d.prior is not None:  # Yes, in Python you can say "is not"!
        return False

    # We now know we can add d to self
    d.prior = self
    self.next = d
    dsides.remove(self.outwardFacingSide)   # this leaves the value *not* matching self's end
    d.outwardFacingSide = dsides[0]
    return True
```

- Q4: skip the subclass stuff in Spring 2022
- Q5: skip: we haven't covered invariants (yet)

2016 Fall:

- Q2: see remarks about Q5 in 2017 Fall.
- Q3: see remarks about Q4 in 2017 Fall.
  - (a) skip the subclass folder
  - (c) typo in solutions, animation cells 4-6: self should be id3, not id2

- Q4a: typo in solutions: if statement should have “!=”, not “==”

2016 Spring:

- Q3(c): solution typo: “lineseg.P2.pos()” should be “lineseg.P2.Pos()”
- Q4: We would explain that estimating the probability would just mean counting the number of times the dice came up with exactly two having the same value, divided by the number of rolls.

2015 Fall:

- Q2: skip in Spring 2022 – haven’t done subclasses yet.
- Q3(b): skip in Spring 2022: haven’t done subclasses yetQ3(d): skip in Spring 2022 – haven’t done error raising yet.
- Q6: see remarks about Q4 in 2017 Fall.

2015 Spring:

- Q2: ignore remark about being familiar with the Traveling Fanatic problem
- Q6(d): answer is “no”: if L is a list, it doesn’t have an attribute nWords.
- skip 4(d) (graphics), 6(b) (try/except) , 6(c) (timing)

2014 Fall:

- Q2: see remarks about Q5 in 2017 Fall.
- Q5: ignore questions involving subclasses in Spring 2022

2014 Spring:

- skip Q3 and Q4 (writing while-loops); Q6 (loop invariants)
- Q5: the “separate handout” being referred to is http://www.cs.cornell.edu/courses/cs1110/2017sp/exams/prelim2/2014-spring-prelim2-enroll.pdf

2013 Fall:

- Q2: see remarks about Q5 in 2017 Fall.
- Q3: ignore questions involving subclasses in Spring
- skip Q6(b) (exception types) in Spring
- skip Q6(c) in Spring

2013 Spring:

skip Q3 (loop invariants), Q4 (writing while-loops)

Appendix: “Students should be able to…” list

1. Everything from the first prelim study guide's "students should be able to", adding in using sequences appropriately in for-loops.
2. Understand and use for-loops effectively.
    1. For demonstrating understanding, I really like Q6 from Spring 2017 prelim 1
    2. For writing, ideally, recognize when it's "best" to:
        1. loop over `range(len(somelist))`
        2. loop over `somelist` itself
        3. loop over `range(some_int)`)
        4. loop over a dictionary (which will set the loop variable to the keys of the dictionary, one at a time)
4. Be able to work with nested lists, with for-loops and/or recursion as best fits the situation
5. Understand and use recursion effectively
    1. A nice understanding-of-code question is Q2 from Spring 2021 prelim 2
    2. Understand and be able to diagram what is going on with the call frames in recursive calls
    3. write recursive implementations
        1. Ideally, recognize when recursion is more suitable vs when a loop (without recursion) is more suitable
        2. At a minimum, be able to write recursive implementations for "naturally recursive" situations --- nested lists and other recursive data structures (classes of objects that can refer to "subobjects" of the same class, like Mixes that can contain subMixes)
6. Work with classes
    1. Know how to call methods (requires an object "to the left of the dot")
    2. Understand how to access/update class variables and object attributes,
    3. Know how to write methods, including understanding the "self" parameter
    4. Understand what double-underscore methods like _ _ init _ _ , _ _ eq _ _ , _ _ str _ _ are for and how to write them
    5. Be able to diagram objects, class folders, method call frames.
        1. For class folders, we won't be too picky, but do have the "tab" on the top right with the class name (not the object ID), and do put the

class attributes and the names of methods in the class folders (*not* the object folders.)

7. Be able to work with dictionaries (*without* relying on methods key() or values() [these return "iterators", which we will not cover in this class])
    1. Ideally, recognize when a dictionary is "better" vs when a list is "better" vs when it doesn't really make a difference