Slicing and recursion

a. Design a function `first_is_unique` that takes a non-empty string `s` as input and returns a Boolean as output. The function should return `True` if the first character in `s` appears is unique – it doesn’t appear anywhere else in `s` – and `False` otherwise (that is, the first character does appear elsewhere in `s`).

While this could be solved recursively or iteratively, neither is necessary! Instead, use string slicing and Python’s `in` operator to write a very short solution.

```python
def first_is_unique(s: str) -> bool:
    """Return True if the first character in s doesn't appear anywhere else in the string and False if it does."""
    return s[0] not in s[1:]

def test_first_is_unique():
    assert first_is_unique("a") == True
    assert first_is_unique("ab") == True
    assert first_is_unique("aba") == False
```
b. Write a recursive function called `every_other_letter` that takes a string as input and returns a string that contains every other letter in the string starting with the first letter. The intended behavior is demonstrated by the provided test function.

```python
def every_other_letter(s: str) -> str:
    """Return a string consisting of every other letter in the input""
    if len(s) < 2:
        return s
    else:
        return s[0] + every_other_letter(s[2:])

def test_every_other_letter():
    assert every_other_letter("a") == "a"
    assert every_other_letter("kale") == "kl"
    assert every_other_letter("banana") == "bnn"
    assert every_other_letter("computer") == "cmue"
```
2 Iteration

a. Consider the following function:

```python
def mystery(l: list):
    for i in l:
        return i
```

What does this function return when \([1, 2, 3]\) is passed in?

1. This function will return the first element of the list in the first iteration of the for loop.

What does the same function return when \(\[]\) is passed in?

None. The function will not enter the for loop, and so it never executes a return statement; it returns None by default.

b. Consider the following function:

```python
def mystery(l: list):
    result = 0
    for i in l:
        result = i
    return result
```

What does this function return when \([1, 2, 3]\) is passed in?

3. In each iteration of the loop, result will be updated with the current value of i. In the last iteration of the loop, result will be updated with 3 and then returned from the function after the loop.

What does the same function return when \(\[]\) is passed in?

0. result will never be updated before it's returned from the function.
c. Consider the following Pyret function definition:

```pyret
fun short-strings(lst :: List<String>, threshold :: Number) -> List<String>:
    filter(lam(word): string-length(word) < threshold end, lst)

where:
    short-strings([list: ], 3) is [list: ]
    short-strings([list: "aaa", "b", "cccc", "dd"], 3) is [list: "b", "dd"]
end
```

Write a Python function `short_strings` that accomplishes the same task as `short-strings`.

**Note:** Your function should use a for loop to iterate through the elements of the input list. It should not use `filter` (or other higher-order functions) or a list comprehension.

```python
def short_strings(lst: list, threshold: int) -> list:
    """Return a list of strings in lst with lengths less than threshold""
    result_list = []
    for word in lst:
        if len(word) < threshold:
            result_list.append(word)
    return result_list

def test_short_strings():
    assert short_strings([], 3) == []
    assert short_strings(["aaa", "b", "cccc", "dd"], 3) == ["b", "dd"]
```
d. Write a function `count_changes` that takes in a string `s` of "0"s and "1"s and returns the number of times there is a change from a "0" to a "1" or vice versa in that input string.

For example, `count_changes("11010000")` would return 3.

```python
def count_changes(s: str) -> int:
    """Return the number of times the string changes from a 0 to a 1 or vice versa."""
    changes = 0
    if len(s) > 0:
        prev_char = s[0]
        for char in s[1:]:
            if char != prev_char:
                changes += 1
                prev_char = char
    return changes

def test_count_changes():
    assert count_changes("") == 0
    assert count_changes("00") == 0
    assert count_changes("11") == 0
    assert count_changes("11011000") == 3
```
3 Structured data and memory

a. Consider the following program:

```python
@dataclass
class Time:
    hour: int
    mins: int

noon = 12
quarter = 15
twelve_fifteen = Time(noon, quarter)
lunch = twelve_fifteen
phone_call = Time(noon, quarter)

# Can't eat till the call's over!
lunch.min = 30
```

List the entries in the directory and the heap after evaluating the program.

Directory:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>noon</td>
<td>12</td>
</tr>
<tr>
<td>quarter</td>
<td>15</td>
</tr>
<tr>
<td>twelve_fifteen</td>
<td>@1000</td>
</tr>
<tr>
<td>lunch</td>
<td>@1000</td>
</tr>
<tr>
<td>phone_call</td>
<td>@1001</td>
</tr>
</tbody>
</table>

Heap:

<table>
<thead>
<tr>
<th>Location</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>@1000</td>
<td>Time(12, 30)</td>
</tr>
<tr>
<td>@1001</td>
<td>Time(12, 15)</td>
</tr>
</tbody>
</table>
b. Consider the following program:

```python
@dataclass
class Course:
    code: str
    students: list  # of strings
    term: str

current_term = "Fall 2022"
next_term = "Spring 2023"

course1 = Course("cmpu101", ["Ada", "Alan"], current_term)
course2 = Course("cmpu102", ["Grace", "Winifred"], next_term)
course3 = course1
course1.students.append("Matthew")
course1.students = ["Ada", "Alan", "Matthew"]
course1.current_term = "Spring 2023"
course1.next_term = "Fall 2023"

print(course1)
print(course2)
print(course3)
```

Write the results of the print statements.

```
Course(code='cmpu101', students=['Ada', 'Alan', 'Matthew'],
       term='Fall 2022')

Course(code='cmpu102', students=['Grace', 'Winifred'],
       term='Spring 2023')

Course(code='cmpu101', students=['Ada', 'Alan', 'Matthew'],
       term='Fall 2022')
```
4 Dictionaries

a. What output is printed by the following code?

d = {"May": "spring"}
d["July"] = "summer"
d["April"] = "spring"

for month in ["March", "April", "May", "June", "July"]:  
  if month in d:  
    print(d[month])  
  else:  
    print("error")

error
spring
spring
error
summer
b. Consider a Python dictionary whose keys are words, and where the associated values are definitions of those words. For example,

```python
animals = {
    "zebra": "a striped animal",
    "aardvark": "an animal that likes to eat ants",
    "skunk": "a striped animal"
}
```

Write a function called `reverse_lookup` that takes as input a dictionary and a definition string. The function then returns the list of all keys that have that definition.

For example, if we passed in the dictionary `animals` and the string "a striped animal", we would get back a list containing "zebra" and "skunk".

```python
def reverse_lookup(d: dict, definition: str) -> list:
    """Return a list of all words in d that have the given definition."""
    matches = []
    for word in d:
        if d[word] == definition:
            matches.append(word)
    return matches

def test_reverse_lookup():
    assert reverse_lookup({}, "a striped animal") == []
    assert reverse_lookup(animals, "an animal with polka dots") == []
    assert sorted(reverse_lookup(animals, "a striped animal")) == ["skunk", "zebra"]
```
5 Tables revisited

Consider the table students, which has columns for the first name, birth month, and favorite color of several students:

<table>
<thead>
<tr>
<th>name</th>
<th>month</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>&quot;Jan&quot;</td>
<td>&quot;red&quot;</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>&quot;Aug&quot;</td>
<td>&quot;red&quot;</td>
</tr>
<tr>
<td>&quot;Cathy&quot;</td>
<td>&quot;Jan&quot;</td>
<td>&quot;blue&quot;</td>
</tr>
</tbody>
</table>

a. Write a line or two of code to add a column called house to the (updated) students table, choosing whichever Vassar houses you'd like for Alice, Bob, and Cathy. The resulting table should have the same name, students.

```python
students = students.with_columns(
    "house", np.array(["Cushing", "Davison", "Jewett")
)
```

b. Write a function whose_birth_month that takes in a string mnth indicating a month and returns an array of the people in the students table who were born in month mnth.

```python
def whose_birth_month(mnth: str) -> np.array:
    
    """Return an array of students born in the given month."""
    t = students.where("month", lambda m: m == mnth)
    return t["name"]

def test_whose_birth_month():
    assert np.array_equal(whose_birth_month("Apr"), np.array([], dtype="<U5"))
    assert np.array_equal(whose_birth_month("Jan"), np.array(["Alice", "Cathy")])
```

c. Write a line of code that calls whose_birth_month to create an array named similar_to_me that has the names of all the people listed in students who were born in the same month as you. (Pass your birth month in as a string, like "Jan", "Feb", etc.)

```python
similar_to_me = whose_birth_month("May")
```