6.009 Quiz 2: Practice Quiz A

Fall 2021

Name: **Answers**

Kerberos/Athena Username:

5 questions 1 hour and 50 minutes

- Please **WAIT** until we tell you to begin.
- This quiz is closed-book, but you may use two 8.5 $\times$ 11 sheets of paper (both sides) as a reference.
- 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. **Please 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 finishing their quizzes.
- You may not discuss the details of the quiz with anyone other than course staff until final quiz grades have been assigned and released.

1 Generators

Consider the following three generators:

```
def numbers():
    i = 1
    while True:
        yield i
        i += 1


def squared(gen):
    for x in gen:
        yield x ** 2


def first(gen, count):
    for ix, x in enumerate(gen):
        yield x
        if ix >= count:
            return
```

and the following three functions:

```
def run_generator_1(gen):
    for x in gen:
        print(x)
        if x > 10:
            pass

def run_generator_2(gen):
    for x in gen:
        print(x)
        if x > 10:
            break

def run_generator_3(gen):
    for x in gen:
        print(x)
        if x > 10:
            continue
```

For each function call on the facing page, indicate how many numbers will ultimately be printed to the screen. Write ∞ if the function call never terminates, and write `ERROR` if the program function call raises an exception.

`run_generator_1(numbers)` How many numbers? **ERROR**

`run_generator_2(numbers)` How many numbers? **ERROR**

`run_generator_3(numbers)` How many numbers? **ERROR**

`run_generator_1(numbers())` How many numbers? ∞

`run_generator_2(numbers())` How many numbers? 11

`run_generator_3(numbers())` How many numbers? ∞

`run_generator_1(squared(numbers()))` How many numbers? ∞

`run_generator_2(squared(numbers()))` How many numbers? 4

`run_generator_3(squared(numbers()))` How many numbers? ∞

`run_generator_1(first(numbers(), 8))` How many numbers? 9

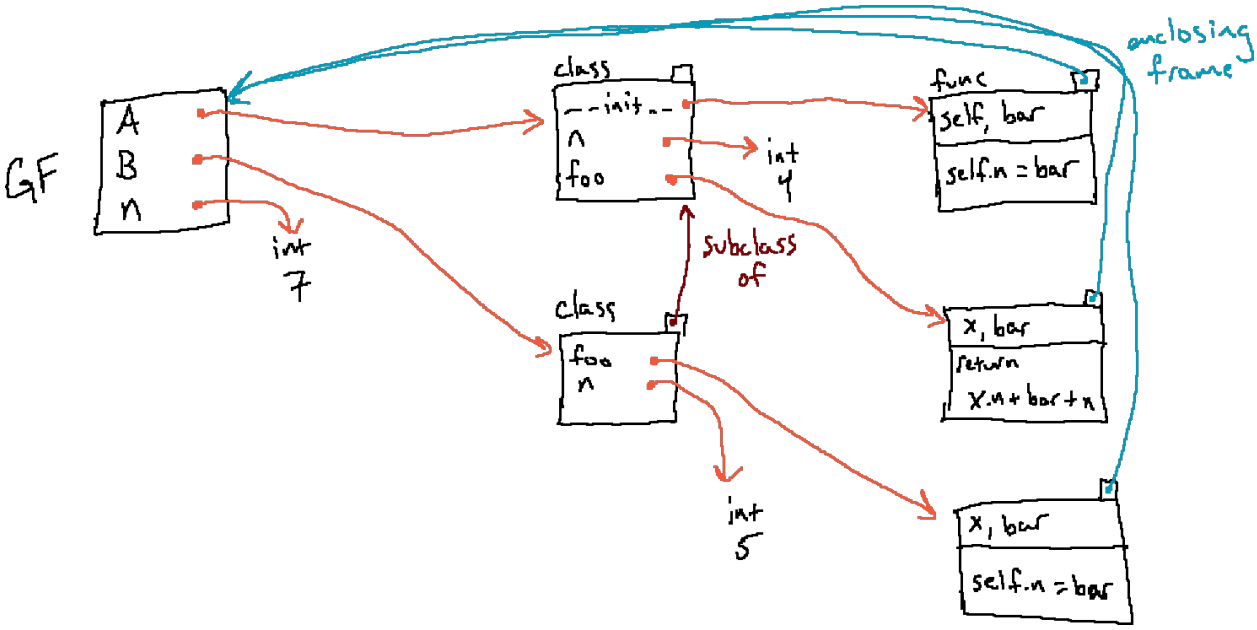
`run_generator_2(first(numbers(), 8))` How many numbers? 9

`run_generator_3(first(numbers(), 8))` How many numbers? 9

2 Environment Diagrams

Part a.

The following environment diagram represents the state of a program after executing some code:



Consider evaluating the expressions on the facing page, one after another, after running the code that led to the diagram above. In each box, write the result of evaluating the expression that preceded it. If an expression would result in an infinite loop, write `INFINITE LOOP`. If it would result in an error, write `ERROR`.

Note that the three angle brackets on each line represent the Python prompt, and the expressions to be evaluated are shown after the prompt.

```
>>> b = B(9)
>>> B.n
```

5

```
>>> b.n
```

9

```
>>> b.foo(10)
```

ERROR

```
>>> B.foo(b, 10)
```

ERROR

```
>>> A.foo(b, 3)
```

19

Part b.

Consider the following piece of code:

```
01 |   class Counter:
02 |       id = 0
03 |       def get_id(self):
04 |           self.id += 1
05 |           return self.id
06 |
07 |   class Counter2(Counter):
08 |       def get_id(self):
09 |           self.id += 2
10 |           return self.id
11 |
12 |   if __name__ == "__main__":
13 |       c2 = Counter2()
14 |       print(c2.get_id())
15 |       Counter2.id += 2
```

In the box on the facing page, draw the environment diagram corresponding to the state of this program after line 15 has executed.

Additionally, in the boxes below, write the value that will be printed by each of these print statements, starting after having run the code above. If an expression would result in an infinite loop, write `INFINITE LOOP` instead. If it would result in an error, write `ERROR`.

```
print(Counter().get_id())
```

1

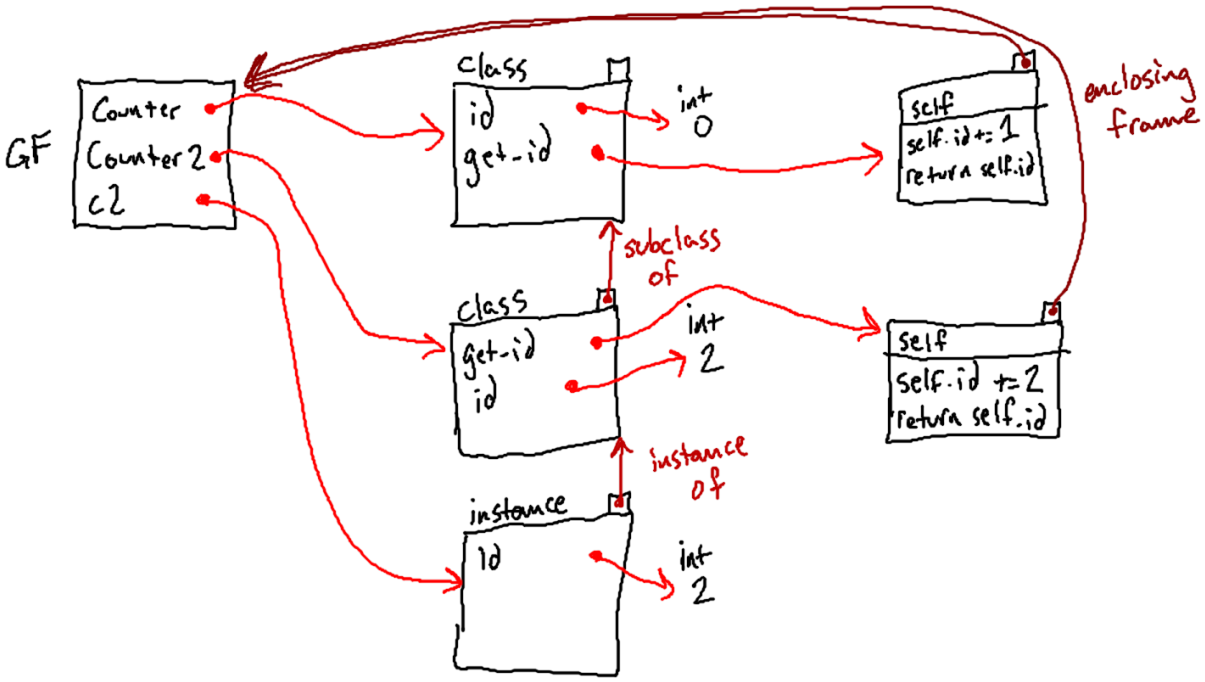
```
print(Counter2().get_id())
```

4

```
print(c2.get_id())
```

4

Environment Diagram:



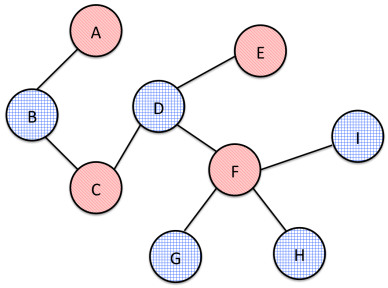
3 Graph Coloring

In this problem, we will consider the goal of "coloring" nodes in a graph with two colors, such that each node receives a color of either `"red"` or `"blue"`, subject to the constraint that no two nodes that are directly connected have the same color.

In this problem, we will represent a graph as a dictionary, mapping each node label to a list of the labels of the nodes that are directly connected to it. Your goal is to check whether the graph can be colored using two colors subject to the constraint mentioned above.

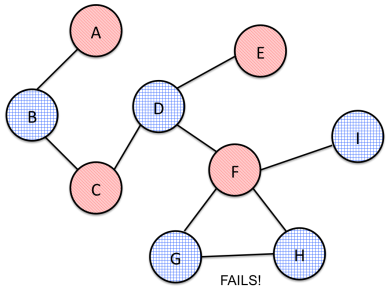
If the graph *cannot* be colored in this way, your function should return an empty dictionary `{}`. If the graph *can* be colored in this way, your function should return a dictionary mapping each node label to either the string `"red"` or the string `"blue"`, representing a valid coloring.

Examples of function invocation and return are given below:




```
alternating_colors({'A': ['B'], 'B': ['A', 'C'], 'C': ['B', 'D'], 'D': ['C', 'E', 'F'],
                    'E': ['D'], 'F': ['D', 'G', 'H', 'I'], 'G': ['F'], 'H': ['F'],
                    'I': ['F']}, 'A')
```

could return `{'A': 'red', 'B': 'blue', 'C': 'red', 'D': 'blue', 'E': 'red', 'F': 'red', 'G': 'blue', 'H': 'blue', 'I': 'blue'}` or a different valid coloring with `'red'` and `'blue'` interchanged.




```
alternating_colors({'A': ['B'], 'B': ['A', 'C'], 'C': ['B', 'D'], 'D': ['C', 'E', 'F'],
                    'E': ['D'], 'F': ['D', 'G', 'H', 'I'], 'G': ['F', 'H'],
                    'H': ['F', 'G'], 'I': ['F']}, 'A')
```

should return an empty dictionary since there is no valid coloring.

The facing page contains a partial definition of a function designed to solve this problem. Fill in the blanks to complete the function so that it works as described above.

```
def alternating_colors(graph, start):
    coloring = {}

    # color in nodes (by mutating the coloring dictionary)
    # return True if the coloring is OK, False otherwise
    def color_node(node, color):
        # base case if we're trying to color a node that has already been colored in (what
        # should we return here?)
        if node in coloring:
```

```
return coloring[node] == color
```

```
        # color the node (mutating the dictionary from above)
        coloring[node] = color

        other_color = 'blue' if color == 'red' else 'red'

        # recursively try neighbors
        for neighbor in graph[node]:
```

```
if not color_node(neighbor,other_color):
    return False
```

```
        # back outside the loop, what should we do/return (if anything)?
```

```
return True
```

```
    # call the helper function and return the appropriate
    # result
    return coloring if color_node(start,'red') else {}
```

4 Expressions Puzzle

In this problem, we will explore an interesting kind of arithmetic puzzle.

Given a number target number T and a single-digit number n, we want to form a mathematical expression containing only the number n that evaluates to T. For example, with $T = 7$ and $n = 4$, one such expression is $((((4 + 4) + 4)/4) + 4) = 7$. In particular, we want to find a `minimal` expression of this form (i.e., the one that contains the fewest instances of n).

We will consider the function `find_expression(T, n, limit=None)` on the second-to-last page of this handout, which was written to solve this problem (you may remove that page). `find_expression` takes integers representing T and n as inputs, as well as a third optional argument `limit`; if `limit` is specified, the code should not consider expressions that contain more than `limit` number of operations.

Ultimately, the goal is to return the answer as a tuple of symbols, where each symbol is either `'*'`, `'/'`, `'%'`, `'+'`, or `'-'`. This tuple represents a sequence of operations that should be performed left-to-right. For example, the tuple `('*', '-', '/', '+')` represents the expression $((((n \times n) - n)/n) + n)$. Note that in this setup, the empty tuple represents n.

If there are no possible answers, we want to return `None`.

Here are some examples of the desired behavior (not necessarily the code that the given implementation produces):

```
>>> find_expression(7, 4, 10)
('+', '+', '/', '+')
>>> find_expression(7, 4, 4)
('+', '+', '/', '+')
>>> find_expression(7, 4, 3) is None
True
>>> find_expression(102, 9, 10)
('*', '+', '+', '*', '+', '+', '+', '/')
>>> find_expression(102, 9, 5) is None
True
>>> find_expression(3, 3, 5)
()
>>> find_expression(1, 20, 5)
('/',)
>>> find_expression(300, 20, 10)
('*', '-', '-', '-', '-', '-')
```

The given implementation of `find_expression` contains a number of deficiencies. There may be issues with correctness, efficiency, and/or style. On the facing page, briefly describe any issues you find with the given implementation, as well as how to fix them (if appropriate).

There is only one real issue with correctness, that is that `limit` is currently ignored completely. Some kind of check like "`if len(expr) > limit:  break`" is necessary after line 19 to handle this case.

However, there are multiple issues with efficiency. For one, we are repeating a lot of work by calling `evaluate` on the entire expression each time. We could save ourselves a lot of effort by storing not only the expression (sequence of operations) but also the result of that sequence of operations. This would allow us to replace the call to `evaluate` with a single call of the operation. In this case, we would initialize "`to_do = [((), num)]`" or similar (storing both the expression and the result), and add children like "`to_do.append((expr[0] + (op,), operations[op](expr[1], num))`".

Even with those changes, this solution will repeat a lot of effort by re-considering multiple expressions that arrive at the same numerical result. We can see a huge performance gain by keeping a "seen" set, containing all of the values we have seen as the result of evaluating an expression. Then, any time we consider a new child, we should only add it to the agenda if its numerical value is one that we haven't already considered.

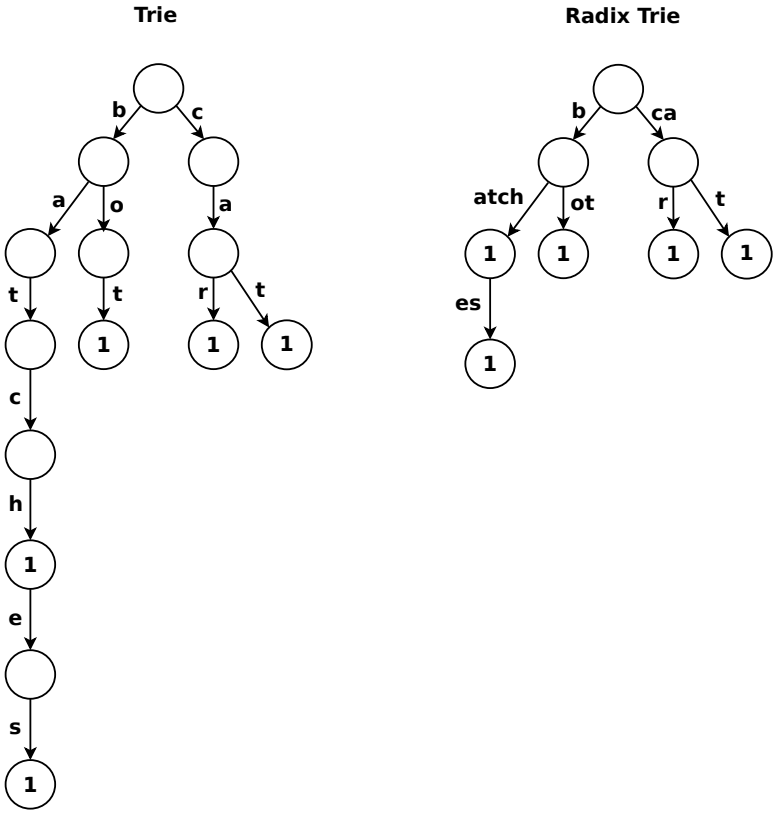
After all of these changes, the code might look something like the following:

```
def find_expression(target, num, limit=None):
    to_do = [((), num)]
    seen = set()
    while to_do:
        expr, val = to_do.pop(0)
        if limit is not None and len(expr) > limit:
            return None
        if val == target:
            return expr
        for op in operations:
            new_val = operations[op](val, num)
            new_expr = expr + (op,)
            if new_val not in seen:
                seen.add(new_val)
                to_do.append((new_expr, new_val))
```

5 Radix Tries

A *radix trie* is a variant of the trie data structure we explored in lab 6. In a radix trie, each node that has no value and has only one child is merged with that child. One effect of this is that the edge labels (the keys in the `children` dictionary) are no longer necessarily of length 1; another effect is that it is often possible to make a radix trie with many fewer nodes than the original trie.

For example, consider the following example trie, and a radix trie containing the same keys (`'car'`, `'cat'`, `'bot'`, `'batch'`, and `'batches'`):



Note that we have provided a complete implementation of `Trie` on the last page of this quiz (which you may remove). We have also provided a placeholder for a class called `RadixTrie` (which inherits from `Trie`) to represent radix tries. You should not modify either class.

In the box on the facing page, write a function called `compress_trie(trie)`. This function should take an instance of the `Trie` class as its input, and it should return an instance of the `RadixTrie` class, representing the corresponding radix trie (where nodes have been merged as described above).

Your function should not modify the structure that was passed in, but rather should make an entirely new instance of `RadixTrie`. All nodes internal to the structure should also be `RadixTrie` instances.

Your function only needs to handle keys that are strings. (Unlike the lab implementation, you do not need to handle tuples as valid keys.)

```
class RadixTrie(Trie):
    pass

def compress_trie(trie):
    rt, _ = helper_trie(trie, '')
    return rt

def helper_trie(trie, prefix=''):
    rt = RadixTrie(str)
    rt.value = trie.value

    if len(trie.children) == 0:
        return rt, prefix

    if len(trie.children) == 1 and trie.value is None:
        (c, ct), = trie.children.items()
        new_child, new_key = helper_trie(ct, prefix + c)
        if prefix == '':
            rt.children[new_key] = new_child
            return rt, prefix
        else:
            return new_child, new_key

    # can't compress
    for c, ct in trie.children.items():
        new_child, new_key = helper_trie(ct, c)
        rt.children[new_key] = new_child
    return rt, prefix
```

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Code for `find_expression`

```
01 |   operations = {
02 |       '+': lambda x, y: x+y,
03 |       '-': lambda x, y: x-y,
04 |       '/': lambda x, y: x//y,
05 |       '*': lambda x, y: x*y,
06 |       '%': lambda x, y: x%y,
07 |   }
08 |
09 |   def evaluate(expr, num):
10 |       out = num
11 |       for op in expr:
12 |           out = operations[op](out, num)
13 |       return out
14 |
15 |
16 |   def find_expression(target, num, limit=None):
17 |       to_do = [()]
18 |       while to_do:
19 |           expr = to_do.pop(0)
20 |           val = evaluate(expr, num)
21 |           if val == target:
22 |               return expr
23 |           for op in operations:
24 |               to_do.append(expr + (op,))
```

Worksheet (intentionally blank)

Code For Tries

```
class Trie:
    def __init__(self, key_type):
        self.key_type = key_type
        self.children = {}
        self.value = None

    def _get_node(self, key, make_new=False):
        if not isinstance(key, self.key_type):
            raise TypeError
        if not key:
            return self
        if key[:1] not in self.children:
            if not make_new:
                raise KeyError
            self.children[key[:1]] = Trie(self.key_type)
        return self.children[key[:1]]._get_node(key[1:], make_new)

    def __setitem__(self, key, value):
        self._get_node(key, make_new=True).value = value

    def __getitem__(self, key):
        out = self._get_node(key).value
        if out is not None:
            return out
        raise KeyError

    def __delitem__(self, key):
        node = self._get_node(key)
        if node.value is None:
            raise KeyError
        node.value = None

    def __contains__(self, key):
        try:
            return bool(self[key] or True)
        except:
            return False

    def __iter__(self):
        if self.value:
            yield (self.key_type(), self.value)
        for child, ctree in self.children.items():
            yield from ((child + key, val) for key, val in ctree)
```

Worksheet (intentionally blank)