CS 111: Introduction to Computer Science

Fall 2022 Semester

Professors: Nancy Fulda and Brett Decker

MIDTERM 2 REVIEW

Student Name: _____

Student ID#: _____

1. (7 points) **StackingTurtles**

Look at the following class definition and answer the questions below.

```
class Turtle:

    def __init__(self, name, below=None, above=None):
        self.name     = name
        self.below    = below
        if below:
            self.below.above = self
        self.above = above
        _____(a)_____:
            _____(b)__________

    def climb_down(self):
        '''Cause this turtle to climb down from its current stack of turtles. The
turtles above this turtle stay in place, and a new stack of turtles is made, where
this turtle is the bottom turtle. If this turtle is already the bottom turtle, this
function does nothing.
        '''
        if self.below:
            self.below.above = None
        self.below = None

    def climb_up(self, other):
        '''Cause this turtle to climb up the stack of turtles containing the other
turtle, making one stack from two stacks, or two different stacks from two existing
stacks. (self and other should not be in the same stack of turtles). The turtles
above this turtle stay in place, and this turtle becomes the first turtle on top of
the other stack.
        '''
        if self.below:
            self.below.above = None
        next = other
        while _(c)_:
            next = next.above
        self.below = next
        next.above = self

    def stack_height(self):
         '''Return the height of the stack that includes this turtle (the number of
turtles in the stack).
```

```
        '''
        height = 1
        next = self
        while next.below:
            height += 1
            next = next.below
        next = self
        while next.above:
            height += 1
            _(d)_
        return height

    def __repr__(self):
        prev = '' if not self.behind else f',below={self.below}'
        nxt  = '' if not self.in_front else f',above={self.in_above}'
        return f'Turtle(\'{self.name}\'{prev}{nxt})'

    def __str__(self):
        return self.name
```

(a) (1 pt) What line of code could go in blank **(a)**?

(b) (1 pt) What line of code could go in blank **(b)**?

(c) (1 pt) What line of code could go in blank **(c)**?

(d) (1 pt) What line of code could go in blank **(d)**?

Now, consider objects being created and displayed as follows:

```
>>> bowser = Turtle('Bowser')
>>> franklin = Turtle('Franklin',below=bowser)
>>> michelangelo = Turtle('Michelangelo',below=franklin)
>>> yertle = Turtle('Yertle',below=michelangelo)
>>> yertle
Turtle('Yertle',below=Michelangelo)
>>> michelangelo
Turtle('Michelangelo',below=Franklin,above=Yertle)
>>> franklin
Turtle('Franklin',below=Bowser,above=Michelangelo)
>>> Bowser
Frog('Bowser',above=Franklin)
```

Here is the current state of our stack of turtles:

Yertle
Michelangelo
Franklin
Bowser

(e) (2 pts) What two lines of code (use the existing functions) would modify the stack of turtles so that it matched the two stacks below?

Michelangelo
Franklin
Bowser Yertle

(f) (1 pt) Now, after executing the lines below (from the state immediately above), how will the turtles be stacked? Use the letters B, F, M and Y to represent the stacked turtles.

```
>>> michelangelo.climb_up(bowser)
>>> yertle.climb_up(bowser)
>>> franklin.climb_down()
```

3. (7 points) **Infinite Generator for Fibonacci Numbers**

Definition. An *infinite* iterator, `t`, is one for which `next(t)` can be called any number of times and always returns a value.

Implement `fibonacci_numbers`, a generator function that creates an infinite iterator for fibonacci numbers.

```
def fibonacci_numbers():
    """Infinite Generator for fibonacci numbers starting at 0.
    >>> fibs = fibonacci_numbers()
    >>> next(primes)
    0
    >>> next(primes) # Second call
    1
    >>> next(primes) # Third call
    1
    >>> next(primes) # Fourth call
    2
    >>> next(primes) # Fifth call
    3
    """
    next = 0
    after = 1
    while( __(a)__ ):
        yield __(b)__
        temp = next + after
        next = __(c)__
        after = __(d)__
```

(a) (2 pts) What line of code could go in blank a)?

(b) (2 pts) What line of code could go in blank b)?

(c) (2 pts) What line of code could go in blank c)?

(d) (2 pts) What line of code could go in blank d)?

(e) (1 pt) Does an *infinite* iterator ever throw a `StopIteration` exception?

- ☐ Yes
- ☐ No

4. (8 points) **Generators for Recursive Objects: Link and Tree**

(This practice problem only includes trees. Try thinking about how you might print the labels in a list in order from the beginning of the list? From the end of the list?)

A *Binary Tree* is a tree data structure where each Tree node only has two branches, which are called *left* and *right*. Each Tree node's **left** branch contains zero or more Tree nodes. Each Tree node's **right** branch contains zero or more Tree nodes.

The code below defines a class called `BinaryTree` that is a recursive object and has a `left` and `right` branch.

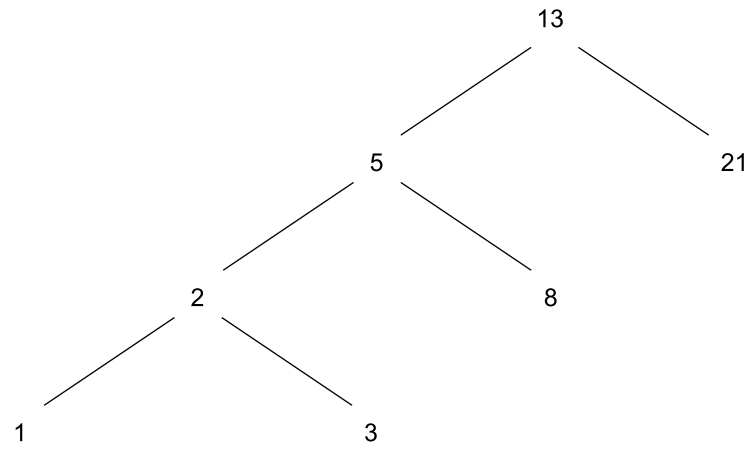
```
class BinaryTree:
    """A binary tree."""
    empty = ()

    def __init__(self, label, left=empty, right=empty):
        self.label = label
        self.left = left
        self.right = right

    def is_leaf(self):
        """
        >>> t = BinaryTree(1)
        >>> t.is_leaf()
        True
        >>> t = BinaryTree(5, BinaryTree(3), BinaryTree(7))
        >>> t.is_leaf()
        False
        """
        return not self.left and not self.right
```

We want to create a generator for a `BinaryTree` to return the labels of the tree in a specific order, called preorder (you did something similar for our regular `Tree` objects in HW05). A preorder traversal first visits the node itself, then the left and right nodes in order.

For the tree below, it would visit in this order:
13, 5, 2, 1, 3, 8, 21



Given these three code snippets:

```
1. if t.left:
       yield from preorder(t.left)
2. yield t.label
3. if t.right:
       yield from preorder(t.right)
```

```
def preorder(t):
    """Yield the entries in this tree in the order that they
    would be visited by a preorder traversal (see problem description).

    "*** YOUR CODE HERE ***"
    __(A)__
    __(B)__
    __(C)__
```

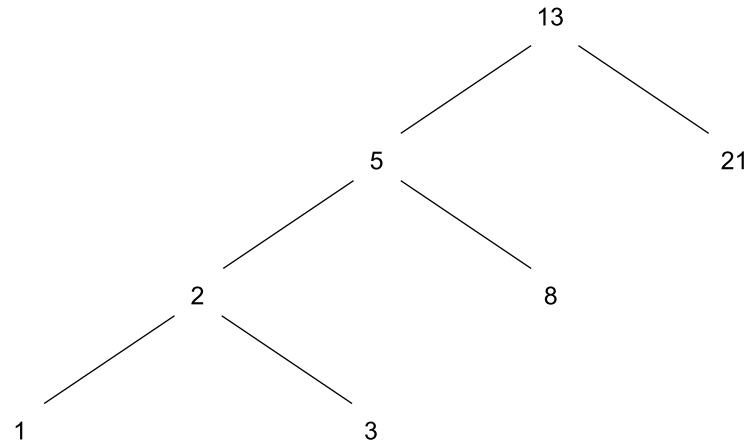
Which snippet would go in (A)?

Which snippet would go in (B)?

Which snippet would go in (C)?

A postorder traversal for a `BinaryTree` would return the labels of the tree in a different order. A postorder traversal first visits the left and right nodes in order, then the node itself.

For the tree below, it would visit in this order:
1, 3, 2, 8, 5, 21,13



Given these three code snippets:

```
4. if t.left:
       yield from postorder(t.left)
5. yield t.label
6. if t.right:
       yield from postorder(t.right)
```

```
def preorder(t):
    """Yield the entries in this tree in the order that they
    would be visited by a preorder traversal (see problem description).

    "*** YOUR CODE HERE ***"
    __(A)__
    __(B)__
    __(C)__
```

Which snippet would go in (A)?

Which snippet would go in (B)?

Which snippet would go in (C)?

There are lots of other ways to order a tree. See Tree traversal - Wikipedia for more information.

6. (8 points) **Classes/Objects - Fill-in-the-blank and WWPD**

Consider the following class definitions:

```
class Bookshelf:
    def __init__(self, capacity, books=[]):
        self.capacity = capacity
        self.books = []
        for book in books:
            self.addBook(book)

    def addBook(self, book):
        if len(self.books) == capacity:
            print(f'Bookshelf is full. Could not add \'{book.title}\'.')
            return
        if ___(a)___:
            self.books.append(book)

    def __add__(self, other):
        if isinstance(other, Bookshelf):
            return [self, other]
        elif isinstance(other, Book):
            shelf2 = Bookshelf(self.capacity,list(self.books))
            shelf2.addBook(other)
            return shelf2

    def __str__(self): # this gets called by print() and str()
        book_string = ', '.join([str(a) for a in self.books])
        space = self.capacity - len(books)
        return f'Books: {book_string}; This shelf can fit {space} more books'

    def __repr__(self): # this gets called by repr() or when the object is displayed
within an iterable/collection
        book_string = ','.join([repr(a) for a in self.books])
        return f'Bookshelf({self.capacity},[{book_string}])'

class Book:
    def _______(b)_______:
        self.title, self.author = title, author

    def _______(c)_______:
        return f'Book(\'{self.title}\',\'{self.author}\')'
```

```
def _____(d)_____:
    return self.title + ', written by ' + self.author
```

Indicate what should appear in blanks **(a)** - **(d)** above:

(a) (1 pt) Which of the following should appear in blank **(a)**

- ☐ `is Book('Frankenstein','Mary Shelley')`
- ☐ `== Book('Frankenstein','Mary Shelley')`
- ☐ `isinstance(book, Bookshelf)`
- ☐ `isinstance(book, Book)`
- ☐ `== new Book()`

(b) (2 pts) Which of the following should appear in blank **(b)**

- ☐ `__init__(self, title, author)`
- ☐ `__add__(self, other)`
- ☐ `__repr__(self)`
- ☐ `__act__(self)`
- ☐ `__str__(self)`

(c) (1 pt) Which of the following should appear in blank **(c)**

- ☐ `__init__(self, author, title)`
- ☐ `__add__(self, other)`
- ☐ `__repr__(self)`
- ☐ `__act__(self)`
- ☐ `__str__(self)`

(d) (1 pt) Which of the following should appear in blank **(d)**

- ☐ `__init__(self, author, title)`
- ☐ `__add__(self, other)`
- ☐ `__repr__(self)`
- ☐ `__act__(self)`
- ☐ `__str__(self)`

Given the code below, what would Python display for each of the following?

```
fiction_shelf = Bookshelf(10)
nonfiction_shelf = Bookshelf(1)
frankenstein = Book('Frankenstein','Mary Shelley')
coraline = Book('Coraline','Neil Gaiman')
print(frankenstein) (e)
adams = Book('John Adams','David McCullough')
hamilton = Book('Alexander Hamilton','Ron Chernow')
nonfiction_shelf.addBook(adams)
nonfiction_shelf += hamilton (f)
fiction_shelf.addBook(frankenstein)
fiction_shelf += coraline
str(fiction_shelf) (g)
```

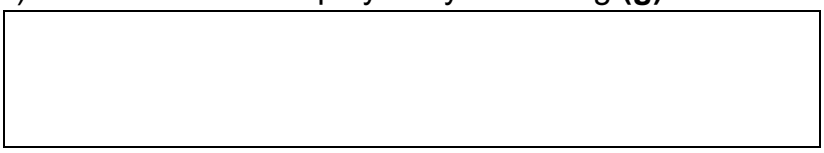
(e) (1 pt) Which of the following would be displayed by executing **(e)**

- ☐ `Coraline`
- ☐ `Frankenstein`
- ☐ `Book('Frankenstein','Mary Shelley')`
- ☐ `'Frankenstein'`
- ☐ `'Frankenstein, written by Mary Shelley'`

(f) (1 pt) Which of the following would be displayed by executing **(f)**

- ☐ `Nothing`
- ☐ `Bookshelf is full. Could not add 'Alexander Hamilton'.`
- ☐ `[Book('John Adams','David McCullough'),Book('Alexander Hamilton','Ron Chernow')]`
- ☐ `Alexander Hamilton, written by Ron Chernow`
- ☐ `[Bookshelf(1,'John Adams, Alexander Hamilton')]`

(g) (1 pt) What would be displayed by executing **(g)**



7. (7 points) **OOP Inheritance / Polymorphism: Programmers**

Suppose we have software that simulates the effectiveness of programmers. Programmers have different amounts of experience and respond differently to stimuli. Each programmer has shaped properties, but there are many different types of programmers. This type of categorization and hierarchy lends itself to using inheritance and polymorphism with Object-Oriented Programming (OOP). Consider the following class `Programmer`.

```
class Programmer:

    def __init__(self, name, typing_speed, experience):
        """Create a Programmer with the given NAME, TYPING_SPEED, and EXPERIENCE.
        name -- A string; The name of the programmer.
        typing_speed -- A number; How quickly this programmer can type (lines per
minute) when they understand the problem they are solving.
        experience -- A number; Number of years the programmer has been programming,
corresponds to the difficulty of problems they can immediately understand.
        """
        self.name = name
        self.typing_speed = typing_speed
        self.experience = experience

    def __str__(self):
        return f"{self.name}: typing_speed {self.typing_speed}, and experience
{self.experience}"

    def action(self):
        """The action performed by the programmer.
        """
```

Every `Programmer` has a name, typing_speed (how many lines can they add to the program in one action), and a number of years of experience. Let us consider two possible subclasses of `Programmer`: `CS111Student` and `TA`. Their implementations are on the next few pages.

```
class CS111Student(Programmer):
    recharge_speed = 10;

    def __init__(self, typing_speed, experience=1, ta):
        """The CS111Student has a default experience of 1.
        name -- A string; the student's name.
        typing_speed -- A number; the lines of code per action this student can write.
        experience -- A number; Total years of experience.
        """
```

```
        super().__init__(name, typing_speed, experience)
        self.ta = ta
        ta.add_student(self)
        self.energy_supply = 10

    def action(self, problem_difficulty):
        """The action performed by the student.
        problem_difficulty -- A number; the experience needed to immediately
understand the problem being worked on.
        """
        if self.energy_supply > 0:
            if (self.experience >= problem_difficulty):
                self.energy_supply -= 1
                print(f"All right! {self.name} added {self.typing_speed } lines of
code.")
            else:
                print("We need to ask a TA for help")
                ta.give_help(self)
        else:
            print("We need to take a rest")
            self.energy_supply += self.recharge_speed

    def receive_help(self, helper):
        """The action performed by the student when they receive help.
        helper -- A programmer; The programmer helping this programmer
        """
        if helper.experience > self.experience:
            self.experience += 1
            print("Thanks for the help! I'll keep trying on this problem.")
        else:
            self.experience += .5
            helper.experience += .5
            print("Two heads are better than one! Let's keep trying on this
problem.")

    def give_help(self, other):
        if self.energy_supply > 0:
            self.energy_supply -= 1
            other.receive_help(self)
            print("Thanks for letting me help you!")
        else:
            print("Sorry, I need to take a rest first.")
```

```
            self.energy_supply += self.recharge_speed



class TA(Programmer):
    recharge_speed = 10;
    lines_per_project = 15;


    def __init__(self, name, typing_speed, experience=3, students=[]):
        """The TA writes code to increase their experience, and gives help to all
           their students when they finish what they are working on.
        name -- A string; The TA's name.
        typing_speed -- A number; How many lines this TA can type in one action.
        experience -- A number; The difficulty of problem this TA can work on without
            needing extra help.
        students -- A list of CS111Students; the students this TA is responsible for.
        """
        super().__init__(name, typing_speed, experience)
        self.students = students
        self.lines_left = self.lines_per_project
        self.energy_supply = 10;

    def add_student(self, student):
        self.students.append(student)

    def action(self, problem_difficulty):
        """The action performed by the TA.
        """
        if self.energy_supply > 0:
            if (self.experience >= problem_difficulty):
                self.energy_supply -= 1
                self.lines_left -= self.typing_speed

                print(f"All right! {self.name} added {self.typing_speed} lines of
code.")
            else:
                print("We need to do some reading")
                self.read_textbook()
        else:
            print("We need to take a rest")
            self.energy_supply += self.recharge_speed
```

```
        if self.lines_left <= 0:
            print("Let's help some students!")
            self.lines_left = self.lines_per_project
        for student in self.students:
            give_help(student)

    def give_help(self, other):
        other.receive_help(self)
        print("Thanks for letting me help you!")

    def read_textbook(self):
        self.experience += 0
```

Fill in the blanks for the following Python program:

```
john = CS111Student("John", 2)
amy = CS111Student("Amy", 2)]
ta = TA("Will", 15, students=[john,amy])

rumpelstiltskin = CS111Student("Rumpelstiltskin", 3, experience=.5)

ta.action(__(a)__)
ta.add_student(rumpelstiltskin)
rumpelstiltskin.action(1)
ta.action(3)
rumpelstiltskin.action(1)

john.energy_supply = 0
_____(b)_____
_____(c)_____

for student in ta.students:
    ___(d)___
```

When the output is as follows:

```
We need to do some reading.
We need to ask a TA for help.
Thanks for the help! I'll keep trying on this problem.
```

```
Thanks for letting me help you!
Alright! Will added 15 lines of code!
Let's help some students.
Thanks for the help! I'll keep trying on this problem.
Thanks for letting me help you!
Thanks for the help! I'll keep trying on this problem.
Thanks for letting me help you!
Thanks for the help! I'll keep trying on this problem.
Thanks for letting me help you!
Alright! Rumpelstiltskin added 3 lines of code!
We need to rest.
Alright! Amy added 2 lines of code!
John: typing speed 2 and experience 2
Amy: typing speed 2 and experience 2
Rumpelstiltskin: typing speed 3 and experience  (e)
```

(a) (1 pt) What must be true about the value in blank (a)?

(b) (1 pt) What line of code could go in blank (b)?

- ☐ john.action(1)
- ☐ amy.action(3)
- ☐ ta.action(7)

(c) (1 pts) What line of code could go in blank (c)?

- ☐ amy.action(3)
- ☐ amy.action(1)
- ☐ ta.action(1)

(d) (1 pts) What line of code could go in blank (d)?

(e) (1 pts) What value would be printed in space (e)?

- ☐ .5
- ☐ 1.5
- ☐ 2.5

(Test continues on next page – last four questions!)

If we want to create a class SuperTA that inherits from TA, fill in the blank with the code below:

```
class SuperStudent( __(e)__ ):

    def __init__(self, typing_speed, experience=1, ta, study_group=[]):

      """The SuperStudent is just like a normal CS111Student, but has a study group
         that they help every action. (You can learn a lot by helping others!!
      """
      super().__init__( __(f)__ , experience, ta);
      self.study_group = study_group


    def action(self, problem_difficulty):
      super().action(problem_difficulty);
      for student in self.study_group:
          __(g)__.give_help( __(h)__ )
```

(f) (1 pt) What value could go in blank (e)?

(g) (0.5 pts) What value could go in blank (f)?

(h) (0.5 pts) What value could go in blank (g)?

(i) (0.5 pts) What value could go in blank (h)?

If all of these classes were in one file, `programmers.py`, give an import statement for the following code:

_____(i)_____

```
liam = CS111Student("Liam", 2)
michael = TA("Michael", 15, students=[liam])
```

(j) (1 pt) What line of code could go in blank (i)?