Homework 6 Solutions **hw06.zip (hw06.zip)**

Solution Files

You can find the solutions in hw06.py (hw06.py).

Required Questions

OOP

Q1: Vending Machine

In this question you'll create a vending machine (https://en.wikipedia.org/wiki/Vending_machine) that only outputs a single product and provides change when needed.

Create a class called `VendingMachine` that represents a vending machine for some product. A `VendingMachine` object returns strings describing its interactions. Remember to match **exactly** the strings in the doctests -- including punctuation and spacing!

Fill in the `VendingMachine` class, adding attributes and methods as appropriate, such that its behavior matches the following doctests:

```
class VendingMachine:
    """A vending machine that vends some product for some price.

    >>> v = VendingMachine('candy', 10)
    >>> v.vend()
    'Nothing left to vend. Please restock.'
    >>> v.add_funds(15)
    'Nothing left to vend. Please restock. Here is your $15.'
    >>> v.restock(2)
    'Current candy stock: 2'
    >>> v.vend()
    'You must add $10 more funds.'
    >>> v.add_funds(7)
    'Current balance: $7'
    >>> v.vend()
    'You must add $3 more funds.'
    >>> v.add_funds(5)
    'Current balance: $12'
    >>> v.vend()
    'Here is your candy and $2 change.'
    >>> v.add_funds(10)
    'Current balance: $10'
    >>> v.vend()
    'Here is your candy.'
    >>> v.add_funds(15)
    'Nothing left to vend. Please restock. Here is your $15.'

    >>> w = VendingMachine('soda', 2)
    >>> w.restock(3)
    'Current soda stock: 3'
    >>> w.restock(3)
    'Current soda stock: 6'
    >>> w.add_funds(2)
    'Current balance: $2'
    >>> w.vend()
    'Here is your soda.'
    """
    def __init__(self, product, price):
        self.product = product
        self.price = price
        self.stock = 0
        self.balance = 0

    def restock(self, n):
        self.stock += n
        return f'Current {self.product} stock: {self.stock}'

    def add_funds(self, n):
        if self.stock == 0:
            return f'Nothing left to vend. Please restock. Here is your ${n}.'
        self.balance += n
        return f'Current balance: ${self.balance}'

    def vend(self):
        if self.stock == 0:
            return 'Nothing left to vend. Please restock.'
        difference = self.price - self.balance
        if difference > 0:
            return f'You must add ${difference} more funds.'
        message = f'Here is your {self.product}'
        if difference != 0:
            message += f' and ${-difference} change'
        self.balance = 0
        self.stock -= 1
        return message + '.'
```

You may find Python's formatted string literals, or f-strings (https://docs.python.org/3/tutorial/inputoutput.html#fancier-output-formatting) useful. A quick example:

```
>>> feeling = 'love'
>>> course = '61A!'
>>> f'I {feeling} {course}'
'I love 61A!'
```

Use Ok to test your code:

```
python3 ok -q VendingMachine
```

If you're curious about alternate methods of string formatting, you can also check out an older method of Python string formatting (https://docs.python.org/2/library/stdtypes.html#str.format). A quick example:

```
>>> ten, twenty, thirty = 10, 'twenty', [30]
>>> '{0} plus {1} is {2}'.format(ten, twenty, thirty)
'10 plus twenty is [30]'
```

Reading through the doctests, it should be clear which functions we should add to ensure that the vending machine class behaves correctly.

`__init__`

- This can be difficult to fill out at first. Both `product` and `price` seem pretty obvious to keep around, but `stock` and `balance` are quantities that are needed only after attempting other functions.

`restock`

- Even though `v.restock(2)` takes only one argument in the doctest, remember that `self` is bound to the object the `restock` method is invoked on. Therefore, this function has two parameters.
- While implementing this function, you will probably realize that you would like to keep track of the stock somewhere. While it might be possible to set the stock in this function as an instance attribute, it would lose whatever the old stock was. Therefore, the natural solution is to initialize stock in the constructor, and then update it in `restock`.

`add_funds`

- This behaves very similarly to `restock`. See comments above.
- Also yes, this is quite the expensive vending machine.

`vend`

- The trickiest thing here is to make sure we handle all the cases. You may find it helpful when implementing a problem like this to keep track of all the errors we run into in the doctest.
    1. No stock
    2. Not enough balance
    3. Leftover balance after purchase (return change to customer)
    4. No leftover balance after purchase
- We use some string concatenation at the end when handling case 3 and 4 to try and reduce the amount of code. This isn't necessary for correctness -- it's ok to have something like:

```
if difference != 0:
    return ...
else:
    return ...
```

  Of course, that would require decrementing the balance and stock beforehand.

Q2: Mint

A mint is a place where coins are made. In this question, you'll implement a `Mint` class that can output a `Coin` with the correct year and worth.

- Each `Mint` instance has a `year` stamp. The `update` method sets the `year` stamp to the `present_year` class attribute of the `Mint` class.
- The `create` method takes a subclass of `Coin` and returns an instance of that class stamped with the `mint`'s year (which may be different from `Mint.present_year` if it has not been updated.)
- A `Coin`'s `worth` method returns the `cents` value of the coin plus one extra cent for each year of age beyond 50. A coin's age can be determined by subtracting the coin's year from the `present_year` class attribute of the `Mint` class.

```python
class Mint:
    """A mint creates coins by stamping on years.

    The update method sets the mint's stamp to Mint.present_year.

    >>> mint = Mint()
    >>> mint.year
    2021
    >>> dime = mint.create(Dime)
    >>> dime.year
    2021
    >>> Mint.present_year = 2101  # Time passes
    >>> nickel = mint.create(Nickel)
    >>> nickel.year     # The mint has not updated its stamp yet
    2021
    >>> nickel.worth()  # 5 cents + (80 - 50 years)
    35
    >>> mint.update()   # The mint's year is updated to 2101
    >>> Mint.present_year = 2176     # More time passes
    >>> mint.create(Dime).worth()    # 10 cents + (75 - 50 years)
    35
    >>> Mint().create(Dime).worth()  # A new mint has the current year
    10
    >>> dime.worth()     # 10 cents + (155 - 50 years)
    115
    >>> Dime.cents = 20  # Upgrade all dimes!
    >>> dime.worth()     # 20 cents + (155 - 50 years)
    125
    """
    present_year = 2021

    def __init__(self):
        self.update()

    def create(self, kind):
        return kind(self.year)

    def update(self):
        self.year = Mint.present_year

class Coin:
    def __init__(self, year):
        self.year = year

    def worth(self):
        return self.cents + max(0, Mint.present_year - self.year - 50)

class Nickel(Coin):
    cents = 5

class Dime(Coin):
    cents = 10
```

Use Ok to test your code:

```
python3 ok -q Mint
```

Linked Lists

Q3: Store Digits

Write a function `store_digits` that takes in an integer `n` and returns a linked list where each element of the list is a digit of `n`.

> **Important**: Do not use any string manipulation functions like `str` and `reversed`

```
def store_digits(n):
    """Stores the digits of a positive number n in a linked list.

    >>> s = store_digits(1)
    >>> s
    Link(1)
    >>> store_digits(2345)
    Link(2, Link(3, Link(4, Link(5))))
    >>> store_digits(876)
    Link(8, Link(7, Link(6)))
    >>> # a check for restricted functions
    >>> import inspect, re
    >>> cleaned = re.sub(r"#.*\\n", '', re.sub(r'"{3}[\s\S]*?"{3}', '', inspect.getsource(
    >>> print("Do not use str or reversed!") if any([r in cleaned for r in ["str", "revers
    >>> link1 = Link(3, Link(Link(4), Link(5, Link(6))))
    """
    result = Link.empty
    while n > 0:
        result = Link(n % 10, result)
        n //= 10
    return result
```

Use Ok to test your code:

```
python3 ok -q store_digits
```

Q4: Mutable Mapping

Implement `deep_map_mut(fn, link)`, which applies a function `fn` onto all elements in the given linked list `link`. If an element is itself a linked list, apply `fn` to each of its elements, and so on.

Your implementation should mutate the original linked list. Do not create any new linked lists.

Hint: The built-in `isinstance` function may be useful.

```
>>> s = Link(1, Link(2, Link(3, Link(4))))
>>> isinstance(s, Link)
True
>>> isinstance(s, int)
False
```

```
def deep_map_mut(fn, link):
    """Mutates a deep link by replacing each item found with the
    result of calling fn on the item.  Does NOT create new Links (so
    no use of Link's constructor)

    Does not return the modified Link object.

    >>> link1 = Link(3, Link(Link(4), Link(5, Link(6))))
    >>> # Disallow the use of making new Links before calling deep_map_mut
    >>> Link.__init__, hold = lambda *args: print("Do not create any new Links."), Link.__
    >>> try:
    ...     deep_map_mut(lambda x: x * x, link1)
    ... finally:
    ...     Link.__init__ = hold
    >>> print(link1)
    <9 <16> 25 36>
    """
    if link is Link.empty:
        return
    elif isinstance(link.first, Link):
        deep_map_mut(fn, link.first)
    else:
        link.first = fn(link.first)
    deep_map_mut(fn, link.rest)
```

Use Ok to test your code:

```
python3 ok -q deep_map_mut
```

Q5: Two List

Implement a function `two_list` that takes in two lists and returns a linked list. The first list contains the values that we want to put in the linked list, and the second list contains the number of each corresponding value. Assume both lists are the same size and have a length of 1 or greater. Assume all elements in the second list are greater than 0.

```
def two_list(vals, amounts):
    """
    Returns a linked list according to the two lists that were passed in. Assume
    vals and amounts are the same size. Elements in vals represent the value, and the
    corresponding element in amounts represents the number of this value desired in the
    final linked list. Assume all elements in amounts are greater than 0. Assume both
    lists have at least one element.

    >>> a = [1, 3, 2]
    >>> b = [1, 1, 1]
    >>> c = two_list(a, b)
    >>> c
    Link(1, Link(3, Link(2)))
    >>> a = [1, 3, 2]
    >>> b = [2, 2, 1]
    >>> c = two_list(a, b)
    >>> c
    Link(1, Link(1, Link(3, Link(3, Link(2)))))
    """
    def helper(count, index):
        if count == 0:
            if index + 1 == len(vals):
                return Link.empty
            return Link(vals[index + 1], helper(amounts[index + 1] - 1, index + 1))
        return Link(vals[index], helper(count - 1, index))
    return helper(amounts[0], 0)
```

Use Ok to test your code:

```
python3 ok -q two_list
```

Extra Questions

Q6: Next Virahanka Fibonacci Object

Implement the `next` method of the `VirFib` class. For this class, the `value` attribute is a Fibonacci number. The `next` method returns a `VirFib` instance whose `value` is the next Fibonacci number. The `next` method should take only constant time.

Note that in the doctests, nothing is being printed out. Rather, each call to `.next()` returns a `VirFib` instance. The way each `VirFib` instance is displayed is determined by the return value of its `__repr__` method.

> *Hint*: Keep track of the previous number by setting a new instance attribute inside `next`. You can create new instance attributes for objects at any point, even outside the `__init__` method.

```
class VirFib():
    """A Virahanka Fibonacci number.

    >>> start = VirFib()
    >>> start
    VirFib object, value 0
    >>> start.next()
    VirFib object, value 1
    >>> start.next().next()
    VirFib object, value 1
    >>> start.next().next().next()
    VirFib object, value 2
    >>> start.next().next().next().next()
    VirFib object, value 3
    >>> start.next().next().next().next().next()
    VirFib object, value 5
    >>> start.next().next().next().next().next().next()
    VirFib object, value 8
    >>> start.next().next().next().next().next().next() # Ensure start isn't changed
    VirFib object, value 8
    """

    def __init__(self, value=0):
        self.value = value

    def next(self):
        if self.value == 0:
            result = VirFib(1)
        else:
            result = VirFib(self.value + self.previous)
        result.previous = self.value
        return result

    def __repr__(self):
        return "VirFib object, value " + str(self.value)
```

Use Ok to test your code:

```
python3 ok -q VirFib
```

Remember that `next` must return a `VirFib` object! With this in mind, our first goal is to calculate the next `VirFib` object and return it. One approach is to figure out the base case (`self.value == 0`) and then decide what information is needed for the following call to `next`.

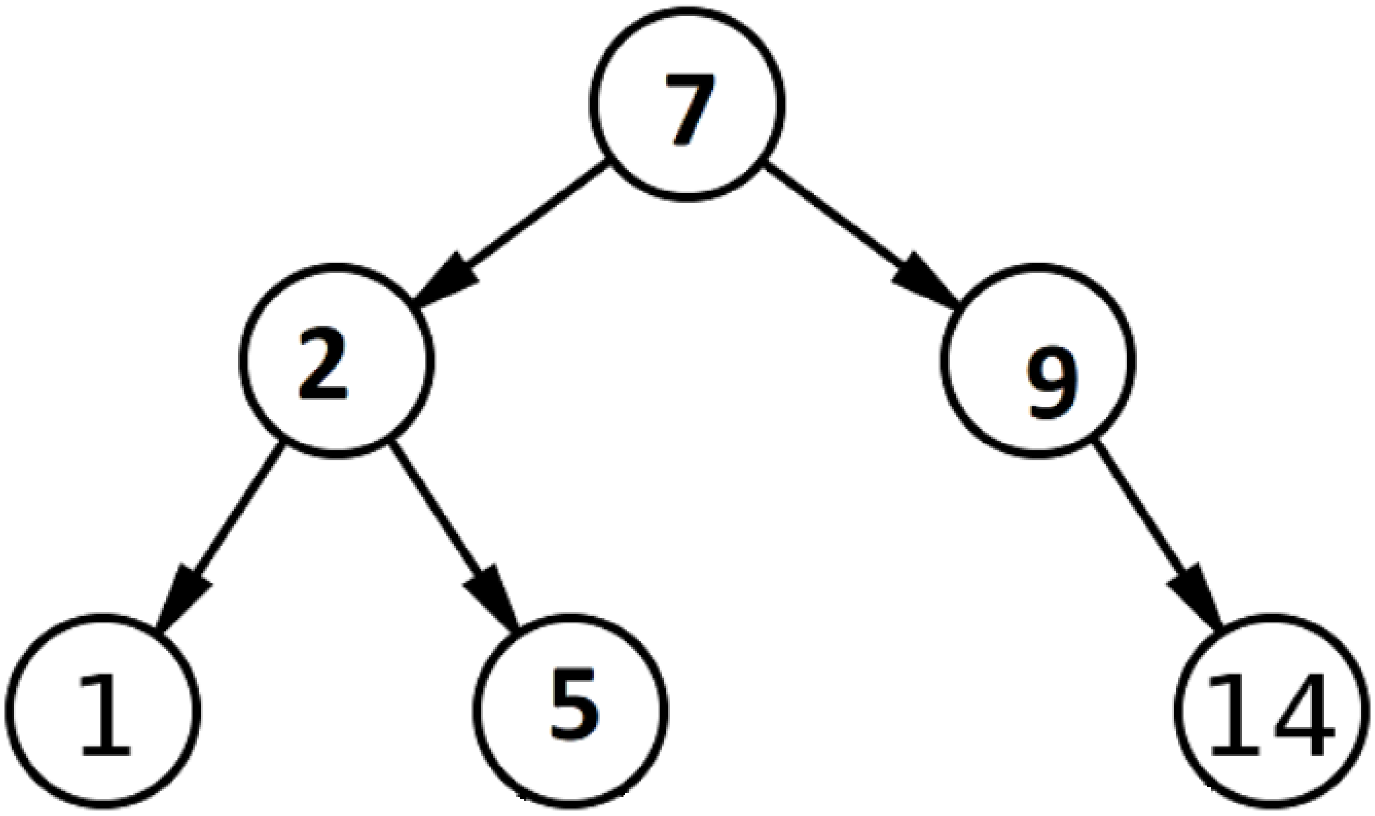
You might also note that storing the current value makes the solution look very similar to the iterative version of the `virfib` problem.

Q7: Is BST

Write a function `is_bst`, which takes a Tree `t` and returns `True` if, and only if, `t` is a valid binary search tree, which means that:

- Each node has at most two children (a leaf is automatically a valid binary search tree)
- The children are valid binary search trees
- For every node, the entries in that node's left child are less than or equal to the label of the node
- For every node, the entries in that node's right child are greater than the label of the node

An example of a BST is:



Note that, if a node has only one child, that child could be considered either the left or right child. You should take this into consideration.

Hint: It may be helpful to write helper functions `bst_min` and `bst_max` that return the minimum and maximum, respectively, of a Tree if it is a valid binary search tree.

```
def is_bst(t):
    """Returns True if the Tree t has the structure of a valid BST.

    >>> t1 = Tree(6, [Tree(2, [Tree(1), Tree(4)]), Tree(7, [Tree(7), Tree(8)])])
    >>> is_bst(t1)
    True
    >>> t2 = Tree(8, [Tree(2, [Tree(9), Tree(1)]), Tree(3, [Tree(6)]), Tree(5)])
    >>> is_bst(t2)
    False
    >>> t3 = Tree(6, [Tree(2, [Tree(4), Tree(1)]), Tree(7, [Tree(7), Tree(8)])])
    >>> is_bst(t3)
    False
    >>> t4 = Tree(1, [Tree(2, [Tree(3, [Tree(4)])])])
    >>> is_bst(t4)
    True
    >>> t5 = Tree(1, [Tree(0, [Tree(-1, [Tree(-2)])])])
    >>> is_bst(t5)
    True
    >>> t6 = Tree(1, [Tree(4, [Tree(2, [Tree(3)])])])
    >>> is_bst(t6)
    True
    >>> t7 = Tree(2, [Tree(1, [Tree(5)]), Tree(4)])
    >>> is_bst(t7)
    False
    """
    def bst_min(t):
        """Returns the min of t, if t has the structure of a valid BST."""
        if t.is_leaf():
            return t.label
        return min(t.label, bst_min(t.branches[0]))

    def bst_max(t):
        """Returns the max of t, if t has the structure of a valid BST."""
        if t.is_leaf():
            return t.label
        return max(t.label, bst_max(t.branches[-1]))

    if t.is_leaf():
        return True
    if len(t.branches) == 1:
        c = t.branches[0]
        return is_bst(c) and (bst_max(c) <= t.label or bst_min(c) > t.label)
    elif len(t.branches) == 2:
        c1, c2 = t.branches
        valid_branches = is_bst(c1) and is_bst(c2)
        return valid_branches and bst_max(c1) <= t.label and bst_min(c2) > t.label
    else:
        return False
```

Use Ok to test your code:

```
python3 ok -q is_bst
```