REPRESENTATION, MUTABLE TREES, LINKED LISTS Solutions

CSM 61A

March 7 - March 11, 2022

1 Representation

Representation Overview: `__repr__` and `__str__`

Classes can have ”magic methods” that add special built-in syntax features. They start and end with double underscores, such as in `__init__`. The goal of `__str__` is to convert an object to a human-readable string. The `__str__` function is helpful for printing objects and giving us information that's more readable than `__repr__`. Whenever we call `print()` on an object, it will call the `__str__` method of that object and print whatever value the `__str__` call returned. However, if a class only defines `__repr__` but not `__str__`, the `print()` call on an object will print what `__repr__` returns instead. For example, if we had a `Person` class with a name instance variable, we can create a `__str__` method like this:

```
def __str__(self):
    return "Hello, my name is " + self.name
```

This `__str__` method gives us readable information: the person's name. Now, when we call print on a person, the following will happen:

```
>>> p = Person("John Denero")
>>> str(p)
'Hello, my name is John Denero'
>>> print(p)
Hello, my name is John Denero
```

The `__repr__` magic method returns the ”official” string representation of an object. You can invoke it directly by calling `repr(<some object>)`. However, `__repr__` doesn't always return something that is easily readable, that is what `__str__` is for. Rather, `__repr__` ensures that all information about the object is present in the representation. Specifically, by convention, this should look like a valid Python expression that could be used to recreate an object with the same value. When you ask Python to represent an object in the Python interpreter, it will automatically call **repr** on that object and then print out the string that **repr** returns. If we were to continue our `Person` example from above, let's say that we added a **repr** method:

```
def __repr__(self):
    return f"Person({self.name})"
    # Note that this returns a string that is exactly the
    # same as the expression we use to construct this object.
```

Then we can write the following code:

```
# Python calls this object's repr function to see what
# to print on the line. Note, Python prints whatever
# result it gets from repr so it removes the quotes
# from the string.
>>> p
Person("John Denero")

# User is invoking the repr function directly.
# Since the function returns a string, its output
# has quotes. In the previous line, Python called
# repr and then printed the value. This line works
# like a regular function call: if a function
# returns a string, output that string with quotes.
>>> repr(p)
'Person("John Denero")'
```

1. **Musician** - What would Python display? Write the result of executing the code and the prompts below. If a function is returned, write ”Function”. If nothing is returned, write ”Nothing”. If an error occurs, write ”Error”.

```
class Musician:
    popularity = 1

    def __init__(self, instrument):
        self.instrument = instrument

    def perform(self):
        print("a rousing " + self.instrument + " performance")
        self.popularity = self.popularity + 2

    def __repr__(self):
        return f'Musician({self.instrument})'

    def __str__(self):
        return self.instrument

class BandLeader(Musician):
    def __init__(self):
        self.band = []

    def recruit(self, musician):
        self.band.append(musician)

    def perform(self, song):
        for m in self.band:
            m.perform()
        print(song)

    def __str__(self):
        band = ""
        for m in self.band:
            band += str(m) + ", "
        return band[:-2] + " - here's the band!"

miles = Musician("trumpet")
goodman = Musician("clarinet")
ellington = BandLeader()
```

Some Quick Refreshers

Defining attributes: Instance attributes are defined with the `self.attr_name` notation (usually in `__init__` but could be elsewhere like in this problem). Class attributes are defined outside of methods in the body of the class definition, like the variable `popularity` in the class `Musician`.

Accessing attributes: Instance attributes are referred to using `self.attr_name` Class attributes can be referred to using `classname.attr_name` or `self.attr_name` (Note: using the latter will only work if there are no instance attributes bound with the name `attr_name`).

Before running any of the code below, `miles` and `goodman` are set to the musicians created as a result of calling the `__init__` constructor method in `Musician`. `ellington` uses `BandLeader's` `__init__` method, since `BandLeader` is the subclass and has `__init__` defined.

```
>>> ellington.recruit(goodman)
>>> ellington.perform()
```

Error

`ellington.recruit(goodman)` adds `goodman` to the end of `ellington's` instance attribute, `band`. Then, `ellington` checks its class (`BandLeader`) for the `perform()` method. But this `perform()` is expecting an argument, so this errors.

```
>>> ellington.perform("sing, sing, sing")
```

a rousing clarinet performance
sing, sing, sing

Using the same `perform()` method, now providing the correct number of arguments. First, going through the band list, `goodman` calls its `perform()` method, which is defined in `Musician`. Here, we print `"a rousing"` + `goodman`'s instrument + `" performance"`, and then `goodman`'s `self.popularity = self.popularity + 2` happens. The `self.popularity` on the right of the equal sign is `Musician.popularity` because `goodman` doesn't have its own instance attribute named `popularity` yet; then it becomes `self.popularity = 1 + 2`, and this creates the instance attribute `popularity` for `goodman`. Then `Musician.popularity`, the class attribute, in incremented by 1.

```
>>> goodman.popularity, miles.popularity
```

(3, 1)

First, we try to get the value of `goodman.popularity`. In our environment diagram, we see that `goodman` has the instance variable `popularity` already defined. Therefore, we get that value, 3, back. Then, we try to access `miles.popularity`. In this case, `miles` doesn't have a `popularity` instance variable defined, so we default to the class variable. There, we see it defined as 1, so we get that value. Finally, since commas in Python define a tuple, we return the two values as `(3, 1)`.

```
>>> ellington.recruit(miles)
>>> ellington.perform("caravan")
```

a rousing clarinet performance
a rousing trumpet performance
caravan

First, we call `ellington.recruit(miles)`. This appends `miles` to `ellington`'s instance variable, `band`. After that, we call `ellington.perform("caravan")`. Similar to the previous call on perform, we will loop through all of the values in `ellington.band`, calling their perform methods in order. This causes the first two lines to be printed. Lastly, we print the `song` variable that was passed in, completing the last line.

```
>>> ellington.popularity, goodman.popularity, miles.popularity
```

(1, 5, 3)

```
>>> print(ellington)
```

clarinet, trumpet - here's the band!

`print``()` expects the string representation of `ellington`, which is given by calling the `__str__()` method of `ellington`. `ellington` checks to see if `BandLeader` has a `__str__()` method, which it does. Inside the for loop, we ask for the string representation for the musicians in this band, and concatenate them together, separated by a comma and a whitespace. The string slicing `band[:-2]` serves to remove the comma and space following the last musician. Finally, **`print`**`(ellington)` then becomes **`print`**`("clarinet, trumpet - here's the band!")`.

```
>>> miles
```

Musician(trumpet)

When prompting for `miles`'s value, we return the representation of ellington given by `__repr__()`. So, we call `Musician`'s `__repr__()` method.

2 Mutable Trees

For the following problems, use this definition for the Tree class:

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)
```

- The constructor constructs and returns a new instance of `Tree`

  `t = Tree(1)#creates a Tree instance with label 1 and no branches`
  .

- The `label` and `branches` are variables, and `is_leaf()` is a method of the class.

  `t.label #returns the label of the tree`

  `t.branches #returns the branches of the tree, which is a list of trees`

  `t.is_leaf()#returns True if the tree is a leaf`

- A tree object is mutable

  To modify a `Tree` object, simply reassign its attributes. For example, `t.label = 2`.

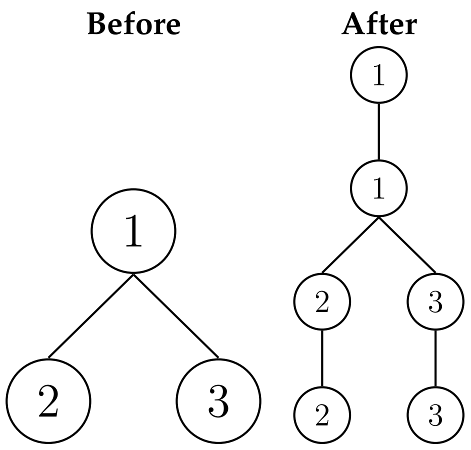
  This means we can mutate values in the tree object instead of making a new tree that we return. In other words, we can solve tree class problems non-destructively and destructively.

1. Implement `tree_sum` which takes in a Tree object and replaces the label of the tree with the sum of all the values in the tree. `tree_sum` should also return the new label.

```
def tree_sum(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4)])
    >>> tree_sum(t)
    10
    >>> t.label
    10
    >>> t.branches[0].label
    5
    >>> t.branches[1].label
    4
    """

    for b in t.branches:
        t.label += tree_sum(b)
    return t.label
```

2. DoubleTree hired you to architect one of their hotel expansions! As you might expect, their floor plan can be modeled as a tree and the expansion plan requires doubling each node (the patented double tree floor plan). Here's what some sample expansions look like:



Fill in the implementation for `double_tree`.

```
def double_tree(t):
    """
    Given a tree, return a new tree where entries appear
    twice.
    >>> double_tree(Tree(1))
    Tree(1, [Tree(1)])
    >>> double_tree(Tree(1, [Tree(2), Tree(3)]))
    Tree(1, [Tree(1, [Tree(2, [Tree(2)]),
                      Tree(3, [Tree(3)])
                     ])
            ])
    """

    if t.is_leaf():
        return Tree(t.label, [Tree(t.label)])
    else:
        dbl_branches = [double_tree(c) for c in t.branches]
        return Tree(t.label,
                    [Tree(t.label, dbl_branches)])
```

3 Linked Lists

Linked lists consists of a series of links which have two attributes: `first` and `rest`. The `first` attribute contains the value of the link (which can be an integer, string, list, even another linked list!). The `rest` attribute, on the other hand, is a pointer to another link or `Link.empty`, which is just an empty linked list represented traditionally by an empty tuple (but not necessarily, so never assume that it is represented by an empty tuple otherwise you will break an abstraction barrier!).

Because each link contains another link or `Link.empty`, linked lists lend themselves to recursion (just like trees). Consider the following example, in which we double every value in linked list. We mutate the current link and then recursively double the rest.

```
def double_values(link):
     if link is not Link.empty:
        link.first *= 2 # we mutate the value inside of the link
        double_val(link.rest) # we mutate the values in the rest
                              # of the linked list
    # if the link is empty then do nothing
```

However, unlike with trees, we can also solve many linked list questions using iteration. Take the following example where we have written `double_values` using a while loop instead of using recursion:

```
def double_values_iter(link):
    while link is not Link.empty:
        link.first *= 2
        link = link.rest # Note that this does not mutate
                         # the original linked list;
                         # it changes what link the variable
                         # link is pointing to
```

Note that unlike Python lists, for a given linked list, we do not know its length immediately by calling `len()`. If we really need its length, we can calculate its manually by iteration or recursion.

For each of the following problems, assume linked lists are defined as follows:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

To check if a `Link` is empty, compare it against the class attribute `Link.empty`:

```
if link is Link.empty:
    print('This linked list is empty!')
```

1. What will Python output? Draw box-and-pointer diagrams to help determine this.

```
>>> a = Link(1, Link(2, Link(3)))

+---+---+   +---+---+   +---+---+
| 1 | --|->| 2 | --|->| 3 | / |
+---+---+   +---+---+   +---+---+

>>> a.first

1

>>> a.first = 5

+---+---+   +---+---+   +---+---+
| 5 | --|->| 2 | --|->| 3 | / |
+---+---+   +---+---+   +---+---+

>>> a.first
```

5

```
>>> a.rest.first
```

2

```
>>> a.rest.rest.rest.rest.first
```

Error: tuple object has no attribute rest (Link.empty has no rest)

```
>>> a.rest.rest.rest = a

   +---+---+  +---+---+  +---+---+
+->| 5 | --|->| 2 | --|->| 3 | --|--+
|  +---+---+  +---+---+  +---+---+  |
|                                   |
+-----------------------------------+

>>> a.rest.rest.rest.rest.first

2

>>> repr(Link(1, Link(2, Link(3, Link.empty))))

"Link(1, Link(2, Link(3)))"

>>> Link(1, Link(2, Link(3, Link.empty)))

Link(1, Link(2, Link(3)))

>>> str(Link(1, Link(2, Link(3))))

'<1 2 3>'

>>> print(Link(Link(1), Link(2, Link(3))))

<<1> 2 3>
```

2. Write a function `skip`, which takes in a `Link` and returns a new `Link` with every other element skipped.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> a
    Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    Link(1, Link(3))
    >>> a
    Link(1, Link(2, Link(3, Link(4)))) # Original is unchanged
    """
    if ___________________________________________:

        __________________________________________

    elif _________________________________________:

        __________________________________________

    ______________________________________________
```

```
    if lst is Link.empty
        return Link.empty
     elif lst.rest is Link.empty:
        return Link(lst.first)
    return Link(lst.first, skip(lst.rest.rest))
```

Base cases:

- When the linked list is empty, we want to return a new Link.empty.
- If there is only one element in the linked list (aka the next element is empty), we want to return a new linked list with that single element.

Recursive case:
All other longer linked lists can be reduced down to either a single element or empty linked list depending on whether it has odd or even length. Therefore, we want to keep the first element, and recurse on the element after the next (skipping the immediate next element with `lst.rest.rest`). To build a new linked list, we can add new links to the end of the linked list by calling skip recursively inside the `rest` argument of the `Link` constructor.

3. Now write function `skip` by mutating the original list, instead of returning a new list. Do NOT call the `Link` constructor.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> skip(a)
    >>> a
    Link(1, Link(3))
    """
```

```
def skip(lst): # Recursively
    if lst is Link.empty or lst.rest is Link.empty:
        return
    lst.rest = lst.rest.rest
    skip(lst.rest)
```

```
def skip(lst): # Iteratively
    while lst is not Link.empty and lst.rest is not Link.empty
       :
        lst.rest = lst.rest.rest
        lst = lst.rest
```

Because this problem is mutative, we should never be creating a new list - we should never have `Link(x)`, or the creation of a new Link instance, anywhere in our code! Instead, we'll be reassigning `lst.rest`.

In order to skip a node, we can assign `lst.rest = lst.rest.rest`. If we have lst assigned to a link list that looks like the following:

```
1 -> 2 -> 3 -> 4 -> 5
```

Setting `lst.rest = lst.rest.rest` will take the arrow that points form 1 to 2 and change it to point from 1 to 3. We can see this by evaluating `lst.rest.rest`. `lst.rest` is the arrow that comes from 1, and `lst.rest.rest` is the link with 3.

Once we've created the following list:

```
1 -> 3 -> 4 -> 5
```

we just need to call skip on the rest of the list. If we call skip on the list that starts at 3, we'll skip over the link with 4 and set the pointer from 3 to point to the link with 5. This is the behavior that we want! Therefore, our recursive call is `skip(lst.rest)`, since `lst.rest` is now the link that contains 3.

4. **(Optional)** Write `has_cycle` which takes in a `Link` and returns `True` if and only if there is a cycle in the `Link`. Note that the cycle may start at any node and be of any length. Try writing a solution that keeps track of all the links we've seen. Then try to write a solution that doesn't store those witnessed links (consider using two pointers!).

```
def has_cycle(s):
    """
    >>> has_cycle(Link.empty)
    False
    >>> a = Link(1, Link(2, Link(3)))
    >>> has_cycle(a)
    False
    >>> a.rest.rest.rest = a
    >>> has_cycle(a)
    True
    """

    seen_before = []
    while s is not Link.empty:
        if s in seen_before:
            return True
        seen_before.append(s)
    return False

    # Alternative solution - less intuitive but more efficient
    if s is Link.empty:
        return False
    slow, fast = s, s.rest
    while fast is not Link.empty:
        if fast.rest is Link.empty:
            return False
        elif fast is slow or fast.rest is slow:
            return True
        slow, fast = slow.rest, fast.rest.rest
    return False
```

4 Magic Methods Extension

Magic Methods

There's so much more to magic methods than meets the eye. These functions constantly work behind scenes to make object oriented programming easier. Unfortunately, the official Python documentation is sparse and confusing, so much of this is borrowed from this blog post.

Let's start with one you're already familiar with, `__init__`. Consider the `Book` class below.

```
class Book:
        def __init__(self, title):
                self.title = title
```

We know that `Book("Dr. Suess")` somehow calls `__init__` with the supplied `title` argument, but where does `self` come from? In fact, when instantiating a new object, the *first* method that gets called is `__new__` which returns an instance of the object (the `self`) and *then* calls `__init__`. This makes it useful for subclassing immutable types like strings and numbers.

If `__new__` is the constructor, `__del__` is the destructor. It handles the process for ”cleaning up” and object. Suppose we want to remove a `Book` from our database. We might need to delete some files as well, in which case we may need to override `__del__` and implement a custom delete procedure. The process is called **garbage collection**, which is an interesting topic in its own right.

Next, let's consider comparisons. We're familiar with numeric comparisons like `6 > 4`, and maybe some other ones too, like `"cat"\leq "dog"` (alphabetical ordering). But suppose we wanted to compare two `Book`s to see which one was longer. Here, we can use magic to methods to specify behavior for comparisons, each of which compares the `self` to some `other` object:

```
class Book:
        def __init__(self, title, pages=0):
                self.title = title
                self.pages = pages
        def __eq__(self, other):
                return self.pages == other.pages
        def __ne__(self, other):
                return self.pages != other.pages
        def __lt__(self, other):
                return self.pages < other.pages
        def __gt__(self, other):
                return self.pages > other.pages
        def __le__(self, other):
                return self.pages <= other.pages
        def __ge__(self, other):
                return self.pages >= other.pages
```

Other numeric operations are also implement this way. For example, we've seen before how to use + for numbers, strings, and lists. But we're only able to specify behaviors for +, -, *, and so on through their corresponding magic methods: `__add__`, `__sub__`, `__mul__`, and so on. And so we could support "adding" two books together by combining titles and pages:

```
class Book:
        def __init__(self, title, pages=0):
                self.title = title
                self.pages = pages
        def __add__(self, other):
                return Book(self.title += " and " + other.title,
                   self.pages + other.pages)
```

Other binary operators include `__floordiv__` (`//`), `__div__` (`/`), `__mod__` (`%`), `__pow__` (`**`), and more. Likewise, magic methods also implement the logic for unary operators like `__abs__` (**abs**), conversions like `__int__` (**int**), and representations like `__str__` (**str**).

At this point, you might be wondering about the += we used earlier. Does it also use `__add__`? No, as it turns out. These methods are called **augmented assignments**, and they use a similar set of magic methods, except they are prepended with an "i" (so `__iadd__` instead of `__add__`). They also don't have a **return** statement, instead mutating the object itself.

```
class Book:
        def __init__(self, title, pages=0):
                self.title = title
                self.pages = pages
        def __iadd__(self, other):
                self.title += " " + other.title
                self.pages += other.pages
```

Let's use this knowledge to revisit an old puzzle. Recall that when concatenating two lists `l1 = [1, 2, 3]` and `l2 = [4, 5, 6]`, using `l1 += l2` is mutative/destructive while using `l1 = l1 + l2` creates a new list. The reason is that the **list** class' `__add__` and `__addi__` functions are implemented differently, so that the former is non-mutative and the latter is mutative. Now it makes sense! — + and += are not the same.

A final word on magic methods and containers. Have you ever stopped to think how `l1[1]` ”gets” the second element of `l1`? Or how Python knows that `1 in l1`? Here’s how magic methods implement each of these operations:

- `__getitem__` and `__setitem__` control what happens when an item is accessed or assigned, e.g. in `l1[1] = 2`.
- `__len__` returns the count of items.
- `__contains__` defines behavior for membership testing.
- `__iter__` returns an iterator, which is used in **`for`** loops.

So if we wanted our `Book` to be a container of ”pages” of text, we could have:

```
class Book:
        def __init__(self, title, pages={}):
                self.title = title
                self.pages = pages
        def __getitem__(self, key):
                return self.pages[key]
        def __setitem__(key, value):
                self.pages[key] = value
        def __contains__(self, item):
                return item in self.pages
        def __len__(self):
                return len(pages)
        def __iter__(self):
                return iter(pages)
```

Other cool magic method topics you should check out include:

- The `with` keyword, and the associated `__enter__` and `__exit__` methods, for context management.
- The `__copy__` and `__deepcopy__` methods, the latter of which also copies the data of the object.
- The `__call__` method, which makes an object callable, behaving a function.