LINKED LISTS AND EXCEPTIONS

COMPUTER SCIENCE MENTORS CS 88

April 12th to April 16th

1 Linked Lists

There are many different implementations of sequences in Python. Today, we'll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a `first` value and the `rest` of the linked list.

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

```
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

Linked lists are a recursive data structure.

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

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

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

>>> a.first = 5
>>> a.first

>>> a.rest.first

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


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

2. Given a number `num`, return a linked list containing the digits of `num` in reversed order.

```
def reverse_digits(num):
    """
    >>> num = 1234
    >>> reverse_digits(num)
    Link (4, Link (3, Link (2, Link (1))))
    """

    if _____________________________________________:

        _____________________________________________

    else:

        _____________________________________________
```

3. Implement the `help` method in the `Mentor` class. In this method, the mentor should help all the `students` that need help (`students` is a linked list of `Student` instances). If a student does not need help, the mentor should move on to the next student. See the doctests for an example of how the `help` method should work!

```
class Student:
    def __init__(self, name, needs_help):
        self.name = name
        self.needs_help = needs_help

class Mentor:
    def __init__(self, name):
        self.name = name

    def help(self, students):
        """
        >>> rahul = Student("Rahul", True)
        >>> kaitlyn = Student("Kaitlyn", False)
        >>> jessica = Student("Jessica", True)
        >>> hetal = Student("Hetal", True)
        >>> ada = Mentor("Ada")
        >>> students = Link(rahul, Link(kaitlyn, Link(jessica,
            Link(hetal))))
        >>> ada.help(students)
        Ada helped Rahul!
        Ada helped Jessica!
        Ada helped Hetal!
        >>> ada.help(students) ## No one needs help anymore,
           so nothing should be printed!
        """
```

2 Additional Linked List Problems

4. 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 _____:

        _____

    _____
```

5. Write a function `combine_two`, which takes in a linked list of integers `lnk` and a two-argument function `fn`. It returns a new linked list where every two elements of `lnk` have been combined using `fn`.

```
def combine_two(lnk, fn):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> combine_two(lnk1, add)
    Link(3, Link(7))
    >>> lnk2 = Link(2, Link(4, Link(6)))
    >>> combine_two(lnk2, mul)
    Link(8, Link(6))
    """
    if _____________________________________:
        return _____________________________
    elif _____________________________________
        return _____________________________
    combined = _____________________________
    return _____________________________________
```

3 Exceptions

Python code could raise exceptions when run, so it's important to catch these exceptions when necessary, instead of letting the exception propogate back to the user. To do this, we can use a `try...except` block and allow the code to continue.

```
try:
    <try suite>
except Exception as e:
    <except suite>
```

We put the code that might raise an exception in the `<try suite>`. If an exception of type `Exception` is raised, then the program will skip the rest of that suite and execute the `<except suite>`. Generally, we want to be specify exactly which `Exception` we want to handle, such as `TypeError` or `ZeroDivisionError`.

Notice that we can catch the exception `as e`. This assigns the exception object to the variable `e`. This can be helpful when we want to use information about the exception that was raised.

Some common exceptions you might encounter are:
`AttributeError` - This occurs when you try to reference an attribute that does not exist.
`IndexError` - Occurs when you try to access an index for a sequence that is out of range.
`KeyError` - Occurs when you try to access a key that does not exist in a dictionary.
`TypeError` - Occurs when an operation or function is applied to an object of inappropriate type.
`ZeroDivisionError` - Occurs when you try to divide a number by zero.

1. You have seen that indexing a list with an index that is not contained in the list generates and exception, as does looking up a key that does not exist in a dictionary. However, the `get` method of `dict` is more forgiving. If the key is not in the dictionary it returns a value that you provide, defaulting to None. Use exception handling in the function `quiet_get` to obtain similar behavior for both lists and dictionaries.

```
def quiet_get(data, selector, missing=None):
    """Return data[selector] if it exists, otherwise missing.
    >>> quiet_get([1,2,3], 1)
    2
    >>> quiet_get([1,2,3], 4)
    >>> quiet_get({'a':2, 'b':5}, 'a', -1)
    2
    >>> quiet_get({'a':2, 'b':5}, 'd', -1)
    -1
    """
```