CSCI 121: Computer Science Fundamentals I
Pratice Second Midterm Exam

The next pages give practice problems for the second midterm exam being held in lecture on Wednesday Apr 19th, 2023. The exam covers these topics:

- recursive functions
- object orientation
- class inheritance
- higher-order functions and `lambda`
- Python code execution and environments
- linked lists

You can use these to test your knowledge in preparation for taking the exam. I will post my solutions to these problems on Friday, Apr 14th.

----

1. Write a *recursive* function `def output_digits_most(n)` that, when given a positive integer `n`, outputs the digits of `n` in order from most- to least-significant. You cannot use any string operations to write this code.

```
>>> output_digits_most(375)
3
7
5
>>> output_digits_most(3)
3
```

2. Write the code for `def make_checker(value)`. It should return a function that, when given a parameter, return `True` if that parameter is equal to the value originally given to `make_checker`. It should return `False` otherwise.

```
>>> x = make_checker(8)
>>> x(6)
False
>>> x(8)
True
>>> x(10)
False
```

Now write the code for `def make_reporter(value)`. It should return a procedure that, when given a parameter, outputs (using `print`) whether that parameter is larger or smaller than the value originally given to `make_reporter`. If it is equal, it should not do anything.

```
>>> r = make_reporter(8)
>>> r(6)
smaller
>>> r(8)
>>> r(10)
larger
```

3. Below is a Python script. Tell us what the script outputs when it is run.

```
def f(x,z):
    def g(x):
        y = x*10 + z
        print(y)
        return [x,y,z]
    z = x + 1000
    return [g,g(4)]
a = f(5,6)
b = f(7,8)
b[1] = a[1]
a[1][2] = 100
c = b[0](9)
print(a[1],b[1],c)
```

4. Write the code for a "zap buzz" counter. It should be a class named `ZapBuzz` that inherits from the `Counter` class given below:

```
class Counter:

    def __init__(self,start):
        self.count = start

    def increment(self):
        self.count = self.count + 1
        return self.count
```

When constructed, a zap buzz counter starts its count at the value 0. When you `increment` it, the method returns the string "zap" if the counter has just reached a value that is a multiple of seven. It returns "buzz" if it has just reached a value that contains the digit 3 in its decimal representation. It returns "zap buzz" if both are true. Otherwise, `increment` just returns that next integer value.

```
>>> zb = ZapBuzz()
>>> zb.increment()
1
>>> zb.increment()
2
>>> zb.increment()
'buzz'
>>> zb.increment(); zb.increment(); zb.increment()
4
5
6
>>> zb.increment()
'zap'
```

When writing the code for the class `ZapBuzz` you can assume you have already defined a function `multipleOf7` and a function `contains3` that check an integer and return `True` or `False` for their conditions.

5. Recall that we built a `Taxi` class that performs `driveTo(x,y)` when it has enough gas to move to position `(x,y)`, and can `pickup()` a passenger when it is at a customer's location. You can get the information about a taxi using the methods `getGas`, `getLocationX`, and `getLocationY`. Also assume that there is a method `getStatus` that returns `True` if a taxi currently carries a passenger, and `False` if it does not.

   Let's organize the taxis. Create a 'Dispatcher' class whose constructor takes no information. Instead, a dispatcher object can hire a taxi object, adding that taxi to its fleet of taxis. This fleet is a list of taxis, but is initially empty, and gets built up with each taxi it hires. Write a 'hire' method to Dispatcher that takes a 'Taxi' object and adds it to its fleet.

   Let's now allow potential customers to hail taxis through a dispatcher. Add a `hail` method that takes the coordinates of the location of a passenger. When that method is run, the dispatcher should scan through its fleet of taxis and find ones that are available, that is, ones that are not occupied by any passenger. It should ignore taxis in its fleet that do not have enough gas to pick up the passenger. Of those taxis in its fleet that can perform the pick-up, the dispatcher should choose the one that's closest to the passenger, ask it to `driveTo` that passenger's location, and then have it perform the pickup.

   If such a taxi from its fleet can be chosen, the `hail` method should return which taxi was hailed. If no taxi can pick up a passenger at that location, the `hail` method should return `None`.

6. In this problem we work with a linked list. We define two classes `LLNode` and `LinkedList` as in lecture:

```
class LLNode:

    def __init__(self,v):
        self.value = v
        self.next = None

class LinkedList:

    def __init__(self):
        self.first = None

    def prepend(self,v):
        n = LLNode(v)
        n.next = self.first
        self.first = n

    def output(self):
        n = self.first
        while n is not None:
            print(n.value)
```

Write a method for `LinkedList` called `swap` at that takes an integer position and restructures the list (by relinking nodes) so that two consecutive nodes swap positions. If the position given is 1, it should swap the first and the second nodes. If the position given is 2, it should swap the second and the third. And so on. For example:

```
>>> xs = LinkedList()
>>> xs.prepend(20); xs.prepend(100); xs.prepend(15); xs.prepend(87)
>>> xs.output()
87
15
100
20
>>> xs.swap_at(1); xs.output()
15
87
100
20
>>> xs.swap_at(3); xs.output()
15
87
20
100
```

Your code can assume that the position parameter is at least 1 and is less than the length of the list.