CS61A Lab 7: Objected-Oriented Programming

Week 7, Spring 2012

Debugging Practice

As projects get larger and more complicated, it's going to be increasingly important to be able to debug code quickly and accurately. And while the TAs and here to help, remember that there's only 9 of us and 365 of you! So with that in mind, let's go through some examples.

0.) Copy the following code into a file named `bugs0.py` and try running it. Do not start by reading the code and trying to understand it. Instead, the goal is going to be to try and find the bug without needing to spend the time understanding precisely what this code does. After running the code, look at the stack trace indicating where the error occurred during execution. Using only the stack trace (so no modifying and re-running the code) answer the follwing questions:

> What does the stack trace tell you about exactly which line caused the problem?
> What function was being called before the error occurred?
> What does the specific error message tell you about what value the variables had where the error occurred?
> Where did the value of those variables come from? Were they generated inside this function or returned from a different function?
> Since you know what variable is problematic and which function returned that value, let's now look at that function. Find all the places this function returns a value. Where do those values come from? Are those variables being derived locally or returned from yet another function?
> You should be able to see the problem pretty quickly now. Fix the bug and re-run the code to check that it works now.

```
def isprime(n):
    """ Returns true if n is a prime. """
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True

def first_prime_in_list(lst):
    """ Returns the first prime in the list and its index. Returns 0,len(lst)
        if there is no prime.
    """
    first_prime = None
    for i in range(len(lst)):
        if isprime(lst[i]):
            first_prime == lst[i]
```

```
                return first_prime, i
            else:
                None
        first_prime = 0
        return first_prime, len(lst)

def sum_primes_in_list(lst):
    """ Returns the sum of primes in the list. """
    sum = 0
    while len(lst) > 0:
        p, index = first_prime_in_list(lst)
        lst = lst[index+1:]
        sum = sum + p
    return sum

print(sum_primes_in_list([4,6,8,9,11,14,15,20]))
```

1.) Copy this next code segment to a new file named `bugs1.py` and try running it. This is similar to the previous exercise, but the problem isn't caused by what a function returns. Nonetheless, answer the same questions from the first question as you try to track down the bug. Again, try not to focus too much on figuring out what the code is trying to do; instead see if you can solve the problem by just looking at the stack trace and following where variables came from.

```
def find_longest_value_in_dict(e):
    largest=-1
    largest_key = -1
    for i in e.keys():
        if len(e[i]) > largest:
            largest = len(e[i])
            largest_key = i
    return e[largest_key]

def get_largest_range(ranges, values):
    # Sort values into a dictionary with keys from ranges
    d = {}
    for v in values:
        for i in range(len(ranges)-1):
            if v >= ranges[i] and v < ranges[i+1]:
                d.get(ranges[i], ()) + (v,)
    return find_longest_value_in_dict(d)

print(get_largest_range([0,10,15,17,20],[2,4,68,2,13,16,17,17,16,15,15,15,4,3,12,15]))
```

Practice With Classes

2.) Predict the result of evaluating the following calls in the interpreter. Then try them out yourself!

```
>>> class Account(object):
...     interest = 0.02
...     def __init__(self, account_holder):
...         self.balance = 0
...         self.holder = account_holder
...     def deposit(self, amount):
...         self.balance = self.balance + amount
...         print("Yes!")
...
>>> a = Account("Billy")
>>> a.account_holder
______

>>> a.holder
______

>>> class CheckingAccount(Account):
...     def __init__(self, account_holder):
...         Account.__init__(self, account_holder)
...     def deposit(self, amount):
...         Account.deposit(self, amount)
...         print("Have a nice day!")
...
>>> c = CheckingAccount("Eric")
>>> a.deposit(30)
______

>>> c.deposit(30)
______
```

3.) Consider the following basic definition of a `Person` class:

```
class Person(object):

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

    def say(self, stuff):
        return stuff

    def ask(self, stuff):
```

```
        return self.say("Would you please " + stuff)

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

Modify this class to add a `repeat` method, which repeats the last thing said. Here's an example of its use:

```
>>> john = Person("John")
>>> john.repeat()        # starts at whatever value you'd like
"I squirreled it away before it could catch on fire."
>>> john.say("Hello")
"Hello"
>>> john.repeat()
"Hello"
>>> john.greet()
"Hello, my name is John"
>>> john.repeat()
"Hello, my name is John"
>>> john.ask("preserve abstraction barriers")
"Would you please preserve abstraction barriers"
>>> john.repeat()
"Would you please preserve abstraction barriers"
```

4.) Suppose now that we wanted to define a class called `DoubleTalker` to represent people who always say things twice:

```
>>> steven = DoubleTalker("Steven")
>>> steven.say("hello")
"hello hello"
>>> steven.say("the sky is falling")
"the sky is falling the sky is falling"
```

Consider the following three definitions for `DoubleTalker`:

```
class DoubleTalker(Person):
    def __init__(self, name):
        Person.__init__(self, name)
    def say(self, stuff):
        return Person.say(self, stuff) + " " + self.repeat()

class DoubleTalker(Person):
    def __init__(self, name):
        Person.__init__(self, name)
    def say(self, stuff):
        return stuff + " " + stuff
```

```
class DoubleTalker(Person):
    def __init__(self, name):
        Person.__init__(self, name)
    def say(self, stuff):
        return Person.say(self, stuff + " " + stuff)
```

Determine which of these definitions work as intended. Also determine for which of the methods the three versions would respond differently. (Don't forget about the `repeat` method!)

5.) Here are the `Account` and `CheckingAccount` classes from lecture:

```
class Account(object):
    """A bank account that allows deposits and withdrawals."""

    interest = 0.02

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        """Increase the account balance by amount and return the new balance."""
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        """Decrease the account balance by amount and return the new balance."""
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance


class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""

    withdraw_fee = 1
    interest = 0.01

    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Modify the code so that both classes have a new attribute, `transactions`, that is a list keeping track of any

transactions performed. For example:

```
>>> eric_account = Account(“Eric”)
>>> eric_account.deposit(1000000)    # depositing my paycheck for the week
1000000
>>> eric_account.transactions
[(‘deposit’, 1000000)]
>>> eric_account.withdraw(100)       # buying dinner
999900
>>> eric_account.transactions
[(‘deposit’, 1000000), (‘withdraw’, 100)]
```

Don't repeat code if you can help it; use inheritance!

6.) We'd like to be able to cash checks, so let's add a `deposit_check` method to our `CheckingAccount` class. It will take a `Check` object as an argument, and check to see if the `payable_to` attribute matches the `CheckingAccount`'s holder. If so, it marks the `Check` as deposited, and adds the amount specified to the `CheckingAccount`'s total. Here's an example:

```
>>> check = Check(“Steven”, 42)  # 42 dollars, payable to Steven
>>> steven_account = CheckingAccount(“Steven”)
>>> eric_account = CheckingAccount(“Eric”)
>>> eric_account.deposit_check(check)  # trying to steal steven’s money
The police have been notified.
>>> eric_account.balance
0
>>> check.deposited
False
>>> steven_account.balance
0
>>> steven_account.deposit_check(check)
42
>>> check.deposited
True
>>> steven_account.deposit_check(check)  # can't cash check twice
The police have been notified.
```

Write an appropriate `Check` class, and add the `deposit_check` method to the `CheckingAccount` class. Make sure not to copy and paste code! Use inheritance whenever possible.