Lab 15: Object Oriented Programming

Mar 27, 2018

Main Event

1. Slots!

The only thing better than playing slot machines is playing slot machines with someone else’s money. In this exercise you’ll build your own slot machine, and it won’t cost you any money!

First, define the class Slot that contains the following methods:

- Constructor

  ```
  def __init__(self, credit)
  ```

  The constructor takes a single argument corresponding to the number of credits added to this slot machine. The value should in turn be used to initialise an attribute of the same name, which will keep track of the amount of “money” available to the player.

- Pull the handle!

  ```
  def pull(self, bet)
  ```

  Pull the handle! Generate three random numbers. If all three are the same, the player wins: increment their credits by twice the bet amount; if they lose, decrement their credits by the bet amount.

In the global scope, instantiate the class with some amount of credit, then create a loop to repeatedly play the game. Print the amount of credit the user has after each play.

You can bet a constant amount each time, or some random amount, the choice is yours. Once there are zero credits in the machine, the loop should stop. If you give yourself 10-pulls, how often do you come out ahead?

Play with the parameters of the game—the range of the random numbers that the slot machine generates, the number of pulls you allow, and the amount you bet each pull—to see if you can come up with a strategy to beat the machine and end with more credits than you started.

Solution:

```
import random

class Slot:
    def __init__(self, credit):
        self.credit = credit

    def pull(self, bet):
        if self.credit < bet:
            return

        previous = None
        for i in range(3):
            current = random.randrange(1000)
            if previous is not None and current != previous:
                self.credit -= bet
                return
            previous = current

        self.credit += bet * 2

bank = 1000
slot = Slot(bank)
for i in range(10):
    if not slot.credit:
        break

    bet = random.randrange(slot.credit) + 1
    slot.pull(bet)

print(slot.credit, 'credits after', str(i + 1), 'spins')
```

2. Timed Q/A

Ever want to see how quickly you can answer a series of questions (that you made up)? Well today's your lucky day! To do this, we will build two classes, `Question` and `Timer`, and use those classes to ask a series of questions.

The first class is QuestionAnswer. It should contain two attributes, `question` and `answer`, along with the following methods:

- Constructor

  ```
  def __init__(self, question, answer)
  ```

  The constructor should take two strings and create attributes with those values.

- Pose the question

  ```
  def ask(self)
  ```

  Ask the question contained in the `question` attribute to the user (input). Return the users answer to the caller. Note that this method has nothing to do with the `answer` attribute.

The second class is Timer. It should contain two attributes: a list called "watches" and an integer called "begin." The class should contain four methods:

- Constructor

  ```
  def __init__(self)
  ```

  The constructor should initialize attributes `watches` and `begin` to something sensible based on their types; for example, the empty list and zero, respectively. There is no need to take them as parameters.

- Start the timer!

  ```
  def start(self)
  ```

  We will use the time module to do this:

  ```
  >>> import time
  >>> time.time()
  ```

```
1446610532.788471
```

The value reported is the number of seconds since 1st January 1970. You should save this to the `begin` attribute. Note that although the import statement in this example came immediately before the its usage, import statements, in general, should go at the top of your Python file.

- Stop the timer

  ```
  def stop(self)
  ```

  Should append the elapsed time—the amount of time since `start` was called—to the watches list.

- Results

  ```
  def results(self)
  ```

  Print the time it took for each question, and the average time required for all questions. For example, if there were a single question:

  ```
  0 7.266237020492554
  Average 7.266237020492554
  ```

In the main body of the program (the “global” scope), create several questions: instantiate several instances of QuestionAnswer with differing question/answer pairs. Next create an instance of Timer. For each question, begin the timer and ask the question. If the users input is a match, stop the timer and move to the next question; if it is not, continue to ask the question until the correct answer is given. Do not stop the time until the answer is correct!

Remember your constructs! QuestionsAnswers should be asked in a loop—in what data type should questions be stored to make this possible? The timers shouldn’t be stopped until the answer provided is correct—what programming mechanisms should you use to support this? (Hint: remember the phone directory exercise!)

Solution:

```
import time
import statistics as st

class QuestionAnswer:
```

```python
    def __init__(self, question, answer):
        self.question = question
        self.answer = answer

    def ask(self):
        return input(self.question + '? ')

class Timer:
    def __init__(self):
        self.watches = []
        self.begin = None

    def start(self):
        self.begin = time.time()

    def stop(self):
        self.watches.append(time.time() - self.begin)

    def results(self):
        for i in range(len(self.watches)):
            print(i, self.watches[i])

        print('Average', st.mean(self.watches))

questions = [
    QuestionAnswer('What is the name of our planet', 'Earth'),
    QuestionAnswer('Where is NYUAD', 'Abu Dhabi'),
    QuestionAnswer('Is Computer Science fun', 'Yes!'),
]

t = Timer()
for q in questions:
    t.start()
    while True:
        if q.ask() == q.answer:
            break
    t.stop()

print(t.results())
```

3. Rock-Paper-Scissors

Recall our lab on rock-paper-scissors. We will now redo that lab with objects. First, define a class called RockPaperScissors

- Constructor

  ```
  def __init__(self)
  ```

  The object should contain an attribute called `hand` that should be initialized to one of three choices: rock, paper, or scissors. Again, the type you use to represent those values is up to you.

- Define string (str)

  ```
   def __str__(self)
  ```

  such that it returns the hand that was chosen, *as a string*:

- Define greater-than (gt)

  ```
  def __gt__(self, opponent)
  ```

  such that compares the hand of the current instance with the hand of an opponent. It should return True if the hand is greater and False otherwise.

  This method will be called whenever you compare two RockPaperScissors objects, which generally happens when the value on the left side of the greater-than (`>`) operator is of type RockPaperScissors.

Finally, play the game! Create a for-loop (in the global scope) that repeatedy creates two hands and compares them. It should print the winner, or whether there was a tie.

Remember, you can only perform greater-than on two RockPaperScissors objects because you've only implemented `__gt__`. That's okay as that's all that is required for this lab. Moreover, to determine the relationship between two types, greater-than *is actually all that you need*.

Solution:

```
import random
```

```python
class RockPaperScissors:
    def __init__(self):
        self.possibilities = ['rock', 'paper', 'scissors']
        self.hand = random.choice(self.possibilities)

    def __str__(self):
        return self.hand

    # Logic comes from the previous lab involving RPS
    def __gt__(self, opponent):
        choices = len(self.possibilities)

        assignment = {}
        for i in range(choices):
            c = self.possibilities[i]
            assignment[c] = i

        mine = assignment[str(self)]
        theirs = assignment[str(opponent)]

        return (mine - theirs + choices) % choices == 1

for _ in range(10):
    p1 = RockPaperScissors()
    p2 = RockPaperScissors()

    if p1 > p2:
        print(p1, 'beats', p2)
    elif p2 > p1:
        print(p2, 'beats', p1)
    else:
        print(p1, 'ties', p2)
```

Additional Practice

1. CSV QA

In the original timed question-answer, questions and answers came from strings that were hard coded into the program. Update your code such that the strings are taken from a CSV file. Specifically, assume the program is presented with a CSV file containing several lines of question-comma-answer’s. Read this file to build your question bank.

Solution:

```
# ...

question = []
with open('qa.csv') as fp:
    for line in fp:
        qa = line.strip().split(',')
        questions.append(qa[0], qa[1])

# ...
```

2. Fibonacci iterator: Recursive

That’s right, an exercise that combines objects and recursion! When we want to iterate over a series of numbers we generally call range to produce a sequential list. This time we will create a class called Fibonacci that will generate a Fibonacci sequence. Sure, we could write a function that builds a list of the first *n* Fibonacci numbers, but that’s not taking advantage of the language—we want native support for this:

```
>>> for i in Fibonacci(5):
...     print(i)
...
0
1
1
2
3
```

You will need to define the class Fibonacci with four methods: a constructor, a function that implements Fibonacci, an iterator, and a next method:

- Constructor

  ```
  def __init__(self, bound)
  ```

  Initializes the class; in particular, stores the required (upper) bound.

- Fibonacci

  ```
  def fib(self)
  ```

  Returns the i^{th} Fibonacci number.

- Iterator

  ```
  def __iter__(self)
  ```

  When Python creates the iterator, this is the first method called; a sort of iterator constructor. It should return the current instance of the class.

- Next

  ```
  def __next__(self)
  ```

  To move through the sequence, Python makes repeated calls to this method. It should return the *next* value in the sequence, unless there are no more items over which to iterate. In this case, the upper bound of Fibonacci sequence that was specified during construction.

  When the upper bound is reached, the method should raise an exception: instead of returning a value, raise StopIteration. For example, normally a function `add` would return the sum of its arguments:

  ```
  def add(x, y):
      return x + y
  ```

  However, if you wanted it to raise an exception, you'd write:

  ```
  def add(x, y):
      raise StopIteration
  ```

  (If you don't understand, create the function and try it.)

Solution:

```
import datetime

class Fibonacci:
    def __init__(self, bound):
```

```
        self.bound = bound

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n >= self.bound:
            raise StopIteration

        fnum = self.fib(self.n)
        self.n += 1

        return fnum

    def fib(self, i):
        if i < 2:
            return i
        else:
            return self.fib(i - 1) + self.fib(i - 2)

start = datetime.datetime.now()
for i in Fibonacci(36):
    print(i)
print('Elapsed:', datetime.datetime.now() - start)
```

3. Fibonacci iterator: On-the-fly

Rather than call the `fib` method during each call to next, update your code such that you return the next value in the relation. Are you able to notice a speed-up for larger i^{th} number requests?

Solution:

```
import math
import datetime

class Fibonacci:
    def __init__(self, bound):
        self.bound = bound
        self.five = math.sqrt(5)
        self.golden = (1 + self.five) / 2
    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n >= self.bound:
            raise StopIteration

        fnum = (self.golden ** self.n - (-self.golden) ** -self.n) / s

        self.n += 1

        return int(fnum)

start = datetime.datetime.now()
for i in Fibonacci(36):
    print(i)
print('Elapsed:', datetime.datetime.now() - start)
```

Introduction to Computer Science

Introduction to Computer Science
jerome.white@nyu.edu

jerome-white

Learning computer science concepts through practice.