Bryan Jensen’s

Programming Assignment #1:
A Poke of Platonic Polyhedrons

Date submitted:

Total Number of Pages:

gist

Jalanorian / gist:1705ba7120b7ccc4dd27
Created 9 minutes ago

gistfile1.py Python (/Jalanorian/1705ba7120b7ccc4dd27/raw/380d90280cee1b12eef98988f2e02703fe8e539a/gistfile1.py)

```
from random import randint

class DiceDict(object):
    """
    A simple "dictionary" implementation. Simple list where the first element
    of any index is the "key" - an integer representing the number of sides of
    that die - of the dictionary, and the second is a sublist of
    two elements: an int representing the number of that die already added
    (to the poke) and an instance of the Die class with the number of sides
    corresponding to the "key".
    """


    def __init__(self):
        """
        Dict is initialized with no components. Format:
        List to simulate the dictionary.
        List[i][0] is the "key"
        List[i][1] is the "value"
        --List[i][1][0] is the number of dice of that type
        --List[i][1][1] is the instance of the Die class
        """
        self.List = []

    def add_or_increment_die(self, dieNumSides):
        """
        Accepts one argument, the number of sides the (possibly) new die will
        have. If the die already exists, increments the count of that die.
        Otherwise creates a new list slot with that die and creates the
        corresponding instance of the Die class, appending it to the list.
        """

        if len(self.List) > 0:
            for i in range(len(self.List)):
                if (self.List[i])[0] == dieNumSides:
                    self.List[i][1][0] += 1
                    break
            else:
                key = dieNumSides
                value = [1, Die(dieNumSides)]
                self.List.append([key, value])

        else:
            key = dieNumSides
            value = [1, Die(dieNumSides)]
            self.List.append([key, value])

    def values(self):
        """
        A simple function to simulate the .values() function of the built-in
        Python dictionary. Returns a list of the second column of the
        "dictionary" list
        """
        return [sublist[1] for sublist in self.List]

class Poke(object):
    """
    A "Poke" (essentially a bag) to vitually hold dice. Has a DiceDict to keep
    track of the dice it currently holds, and various functions to allow
    for randomized access to the Poke, such as rolling and adding dice.
    """
    def __init__(self):
        """
        Initializes empty with a new DiceDict and a variable to keep track of
        how many dice it holds (used later for quicker randomized dice
        selection).
        """
        """
        self.diceDict = DiceDict()
        self.totalNumDice = 0

    def add_die(self, numDieSides):
        """
        Called with one argument, the side of the die to be added, it will then
        call the DiceDict to add it to the list and also increment its running
        total of the currently held number of dice. Exception catching is
        performed at the level of the poke.
        """

        if type(numDieSides) != int:
            return

        self.diceDict.add_or_increment_die(numDieSides)
        self.totalNumDice += 1

    def pick_die(self):
        """
        Randomly returns one Die instance from the poke.
        """
        r = randint(1,self.totalNumDice)

        for eachDieType in self.diceDict.values():
            if r > eachDieType[0]:
                r -= eachDieType[0]
            else:
                return eachDieType[1]

    def sample_poke(self):
        """
        Calls pick_die to randomly select a die from the poke and then calls
        roll() on that Die instance to obtain the numerically rolled value,
        returning that to the caller.
        """
        return self.pick_die().roll()

    def print_poke(self):
        """
        Prints the contents of the poke to the console, omitting any slots of
        the poke that have no dice and therefore haven't been initialized in
        the DiceDict yet.
        """

        for dieType, value in sorted(self.diceDict.List):
            print str(value[0]), str(dieType) + "-sided dice"

class Die(object):
    """
    A simple class to represent a die, upon which can be called only the roll
    method. Keeping track of the number of sides internally, it uses
    random.randint to generate an appropriate value and returns that.
    """
    def __init__(self, sides):
        """
        Initializes with a simple internal value of the number of sides of that
        instance of the Die class.
        """
        self.sides = sides

    def roll(self):
        """
        Returns a random integer value based on the number of sides on that die.
        """
        return randint(1,self.sides)

def main():

    # Initalize the (empty) poke
    poke = Poke()

    # A list of the possible dice being used in this particular test.
    possDice = [4,6,8,12,20]

    # A testing of the Exploration case of 2 tetra-, 0 hexa-, 3 octa-, 1 dodeca-,
    # and 4 icosa-hedrons. Also testing the catching of unintended inputs.
    # Using the list of possible dice to make assignment easier.
    poke.add_die(possDice[0])
    poke.add_die(possDice[0])
    poke.add_die(possDice[2])
    poke.add_die(possDice[2])
    poke.add_die(possDice[2])
    poke.add_die(possDice[3])
    poke.add_die(possDice[4])
    poke.add_die(possDice[4])
    poke.add_die(possDice[4])
    poke.add_die(possDice[4])
    poke.add_die("fas")

    # Testing of the print_poke() poke method and double-checking the
    # implementation of add_die()
    poke.print_poke()

    # Variables for Observed Expected Values
    total = 0.0
    testSize = 1

    # Iterating through for the three (3) cases for Observed Expected Values
    # and printing the results to the console.
    for i in range(3):
        total = 0.0
        testSize *= 100

        for i in range(testSize):
            total += poke.sample_poke()


        print total
        print total / testSize

    # Testing again with another poke (POKE)
    POKE = Poke()

    for i in range(1000):
        r = randint(0,4)

        POKE.add_die(possDice[r])

    POKE.print_poke()


if __name__ == '__main__':
    main()
```

Output from main():

```
2 4-sided dice
3 8-sided dice
1 12-sided dice
4 20-sided dice
767.0
7.67
66638.0
6.6638
6693679.0
6.693679
206 4-sided dice
205 6-sided dice
210 8-sided dice
181 12-sided dice
198 20-sided dice
```

Answers to Explorations/Questions:

1. Theoretical Expected Values of Single Die:
   a. Tetrahedron: 2.5
   b. Hexahedron: 3.5
   c. Octahedron: 4.5
   d. Dodecahedron: 6.5
   e. Icosahedron: 10.5
2. Observed Expected Values of Single Die (100, 10000, and 1000000 tries):
   a. Tetrahedron:
      i. 2.51
      ii. 2.4892
      iii. 2.498193
   b. Hexahedron:
      i. 3.56
      ii. 3.5155
      iii. 3.499269
   c. Octahedron:
      i. 4.63
      ii. 4.5187
      iii. 4.495409

d. Dodecahedron:

   i. 6.35
   ii. 6.5157
   iii. 6.495648

e. Icosahedron:

   i. 10.57
   ii. 10.3865
   iii. 10.501721

3. Theoretical Expected Values of All Dice:
   a. One of each: 5.5
4. Observed Expected Value of All Dice:
   a. One of each(100, 10000, and 1000000 tries):
      i. 5.58
      ii. 5.4866
      iii. 5.500946
5. A poke expected value is based on the dice contained therein.
   a. This means that not only can changing the dice in any given poke affect the expected value, but also the expected given value can be computed given the explicit knowledge of the contents of any given poke.
   b. I.e. in a poke of a single icosahedron, with an even likelihood of rolling any of the 20 values, you will get an expected value of the average of all the possible rolls: 10.5.
6. Poke of 10 dice, 2 in 10 chance of getting a tetrahedron, then a 1 in 4 chance of rolling a 1 from those, repeated for each die and computed in Microsoft Excel…:
   a. 5.65
7. Observed Expected Value of Example Poke:
   a. 2 Tetra, 0 Hexa, 3 Octa, 1 Dodeca and 4 Icosa:
      i. 6.25
      ii. 6.7671
      iii. 6.705972