Lab 16: OOP continued

Mar 29, 2018

Main Event

1. Introduction

"Particles" are ASCII characters that dance around your screen. In this lab we will build such particles, along with the infrastructure required to do so.

2. Particle class

Here, we will simulate particles in space. In the global scope, we'll need the following code:

```
p = Particle(0, 0, 1, 1, 'o')
for i in range(100):
    board = make_board(10, 10)
    update_board(board, p)
    print_board(board)
    p.move()
```

This is how the infrastructure will be used. Notice that we'll need to build a Particle class, whose constructor takes five arguments. The class also contains a move method that takes no parameters. Finally, there's the old board code. Notice now, however, that `make_board` doesn't take a filler, and `update_board` only takes a board and a Particle. Let's take a closer look at how this is done:

A particle is an object that has a position, a velocity, and an identifier; all of which should be attributes of the class. The position consists of an *x* and *y* coordinate—the first and second argument of the Particle constructor above. The velocity can also be represented as a pair: a velocity in the *x*-direction, and a velocity in the *y*-direction. Representing both as integers is fine; they should be the third and fourth parameters, respectively, of the Particle constructor. A particle should also have an identifier: a single character representation of itself; `'o'`, for example, as denoted by the fifth parameter above.

A particle should be able to “move”; this is the purpose of the `move` method. The method essentially increases the objects *x*-position by the *x*-velocity, and its *y*-position by the *y*-velocity.

Solution:

```
class Particle:
    def __init__(self, x, y, xvel, yvel, identifier):
        self.x = x
        self.y = y
        self.xvel = xvel
        self.yvel = yvel
        self.indentifier = identifier

    def move(self):
        self.x += self.xvel
        self.y += self.yvel
```

3. Board class

Now that the particle is in place, alter your board code to be object-oriented instead of procedural. Thus, the code that previously created a Particle and printed the board should be changed:

```
p = Particle(0, 0, 1, 1, 'o')
for i in range(100):
    board = Board(10, 10)
    board.update(p)
    print(board)
    p.move()
```

For this to happen, you need to create a Board class:

1. Instead of being standalone functions—`make_board`, `update_board`, `print_board`—these should now be methods in a Board class.
2. Recall that the first argument to a method in Python is the variable `self`. Typically, when converting functions to methods you must insert this parameter

into the existing list. In this case, however, one of the parameters lends itself naturally to being `self`…

3. Notice that `make_board` is essentially the constructor.

4. In order to use print on an object of type Board, Python needs to know how to cast a Board to string. It calls `__str__` to do so. Thus, your `print_board` should be called `__str__`.

   In Python, the semantics of `__str__` are that it does not actually print anything, but instead returns a value of type string (which in turn can be printed by the caller). Thus, instead of making several calls to print, and thereby outputting various board elements to the screen, *it will create a string and return it*.

Solution:

```
class Board:
    def __init__(self, rows, columns):
        self.rows = rows
        self.columns = columns
        self.board = []
        for i in range(self.rows):
            row = [' '] * self.columns
            self.board.append(row)

    def __str__(self):
        board = ''
        for row in self.board:
            board += ' '.join(row + ['\n'])

        return board

    def update(self, particle):
        in_rows = 0 <= particle.x < len(self.rows)
        in_columns = 0 <= particle.y < len(self.columns)

        if in_rows and in_columns:
            self.board[particle.x][particle.y] = particle.identifier
```

4. Pair class

Notice that position and velocity are both pairs of numbers. To *encapsulate* this abstraction, create a class Pair that contains two integers and can be added (mathematically) to another Pair. From a programming standpoint, we would like Particle creation to look like this:

```
position = Pair(0, 0) # initial spatial position
velocity = Pair(1, 1) # initial particle velocity
p = Particle(position, velocity, 'o')
```

Specifically, a pair is essentially a class that contains two attributes of type integer. The ability to add one Pair to another requires the implemention of the `__add__` method such that is sums both attributes and returns a new Pair. A programmer would use your class as follows:

```
>>> p1 = Point(0, 1)
>>> p2 = Point(1, 1)
>>> p3 = p1 + p2
>>> print(p3.x, p3.y)
1 2
```

Once implemented, alter your Particle code use this class.

Solution:

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, right):
        return Pair(self.x + right.x, self.y + right.y)
```

5. Multiple particles

Instead of having a single particle, have multiple particles! Think about the data types, and additional program structure, required to do this:

- You will need to construct, and store, several particles.
- Each of these particles will need to be individually placed and moved.

- You can print the board after each, but doing so isn’t optimal!

Solution:

```
import string
import random

rows = 10
columns = 10
population = 10
particles = []

for i in range(population):
    x = random.randrange(rows)
    y = random.randrange(columns)
    position = Pair(x, y)

    while True:
        x = random.randint(-1, 1)
        y = random.randint(-1, 1)
        if x or y:
            velocity = Pair(x, y)
            break

    identifier = random.choice(string.ascii_letters)

    p = Particle(position, velocity, identifier)
    particles.append(p)

while True:
    board = Board(rows, columns)

    for p in particles:
        while True:
            p.move()

            in_rows = 0 <= p.position.x < board.rows
            in_columns = 0 <= p.position.y < board.columns

            if in_rows and in_columns:
                break
            if not in_rows:
                p.velocity.x *= -1
            if not in_columns:
                p.velocity.y *= -1

        board.update(p)

    print(board)
```

Additional Practice

1. Animated particles

Instead of printing the board sequentially, we'll now "update" the screen such that only a single board appears to be printed. This functionality works best when running your program from the command line.

First, a few imports:

```
import os
import time
```

Next, prior to printing your board, make the following calls:

```
os.system('clear') # for Linux/Mac users
os.system('cls')   # for Windows users
print(board)       # the original statement
time.sleep(2 ** -2)
```

The first statement clears the screen. You only need one of the `os.system` statements—choose the one appropriate for your platform. The second pauses the program for a quarter of a second.

Next, run the program in terminal. For Mac users, find the location of your Python file. If the location is /User/name/directory/file.py, for example, then you can run the program as follows:

```
$> python3 "/User/name/directory/file.py"
```

Normally you don't need quotes, but if there are spaces anywhere in the path—the stuff leading up to, and including, the file name—then quotes are the easiest way to deal with it. If you're having trouble, that's fine, just ask for assistance.

2. Binary trees

Recall our discussion on trees: structures that stem “downward” from a root node. Tree’s can be general—each node having *n* number of children—but for now we’ll focus on binary trees: trees in which each node has at most two children.

If you draw a tree, it’s essentially a recursive data type: each node has a value—the data that you want to store in the tree—and attributes pointing toward its children. What makes a tree recursive is that in order to point to the children, the variables have to be of the same type as the class itself!

Implement such a class (call it Node):

- The constructor

  ```
  def __init__(self, value)
  ```

  The class constructor should take a single parameter, `value`, representing the data of that child. It should also define two attributes, `left` and `right`, that *will eventually* be references to other Node’s—at construction time you can give them default values that evaluate to false (`None` is generally best).

- Adding a value to the tree

  ```
  def add(self, value)
  ```

  Given a single value, create a place for it in the tree. Specifically it should recursively walk the tree—going left or right depending on how the value compares to the current Node (`self`)—and adds itself as the appropriate (left or right) child of a `Node` that doesn’t have one.

- Find a value within a tree

  ```
  def find(self, value)
  ```

  Logically similar to `add`, but returns true if the value is found, and false otherwise.

There are numerous Python implementations of binary trees online. I highly suggest you give this a try before resorting looking them up (ask for help if you need!). The reason for this is because the ability to implement a tree is a fundamental skill for a computer scientist. In the near future, you will almost certainly have to implement this next semester in data structures. In the longer term, it’s not uncommon for this to an expected ability during a technical interview. Thinking through it now, when it doesn’t really count, will put you ahead of the game.

Solution:

```
class Node:
    def __init__(self, value):
        self.left = None
        self.right = None
        self.value = value

    def add(self, value):
        if self.value < value:
            if self.left:
                self.left.add(value)
            else:
                self.left = Node(value)
        else:
            if self.right:
                self.right.add(value)
            else:
                self.right = Node(value)

    def find(self, value):
        if self.value < value:
            if self.left:
                return self.left.find(value)
        elif value < self.value:
            if self.right:
                return self.right.find(value)
        else:
            return True

        return False

nodes = 10
root = None
for i in range(nodes):
    if root is None:
        root = Node(i)
    else:
        root.add(i)
    print(i, 'added')

print('-' * 10)

for i in range(-nodes, nodes * 2):
    found = 'found'
    if not root.find(i):
        found = 'not ' + found
    print(i, found)
```