

Python 101

Lec06 Classes

thoum

September 4, 2019

Programming up till now


Procedure-Oriented Programming

We pass values to functions, get values, pass them to another function....

```
import sys
input = lambda: sys.stdin.readline().rstrip()

def solve(n, m, maze) :
    qu = [(0,0,1)]
    visited = [[False for c in range(m)] for r in range(n)]
    while len(qu) :
        cx,cy,ci = qu.pop(0)
        if cx == n-1 and cy == m-1 :
            return ci
        if visited[cx][cy] :
            continue
        visited[cx][cy] = True
        if 0 < cx and maze[cx-1][cy] == '1' and not visited[cx-1][cy] :
            qu.append((cx-1, cy, ci+1))
        if cx < n-1 and maze[cx+1][cy] == '1' and not visited[cx+1][cy] :
            qu.append((cx+1, cy, ci+1))
        if 0 < cy and maze[cx][cy-1] == '1' and not visited[cx][cy-1] :
            qu.append((cx, cy-1, ci+1))
        if cy < m-1 and maze[cx][cy+1] == '1' and not visited[cx][cy+1] :
            qu.append((cx, cy+1, ci+1))
    return -1

n, m = map(int, input().split())
maze = [input() for i in range(n)]
print(solve(n, m, maze))
```



Procedure Oriented Programming

When programs get large, Procedure-Oriented might be *too* complicated.

Object Oriented Programming

Combine data and functionality in to an *object*.

View programs as object communicating with each other.

Objects

Integers are objects (of the int class).

Strings are objects (of the str class).

`[1, 2, 3].sort()` are their class methods

`len([1, 2, 3])` returns their internal data: length

Creating Classes of our own

We don't usually use classes so much unless we start writing bigger programs.

We are going to build a basis for a simple RPG game.

The Basis

There are two characters in this game(for now).

The boss, and you.

They are both *beings*(there are other *beings* like the halflings, dragons, darkelves...).

Beings

This becomes the basis(or the *superclass*, *parentclass*) of all living things that freely roam the grounds of the middle earth. Every *being* can be characterized by a name, HP, MP, and their race.

Beings code

```
class Being():
    """Top level generic class for all living things"""

    # Class Variables
    population = 0

    def __init__(self, name, hp, mp, race):
        # Object Variables
        self.name = name
        self.hp = hp
        self.mp = mp
        self.race = race

        Being.population += 1
        print("A new", race, "is born.")

    def die(self):
        self.hp = 0
        print(self.name, " is dead.")
        Being.population -= 1
        print(Being.population,
              "being is alive on middle earth")
```

Explanation

Names of classes begin with capital letters. (Just a convention, but follow it.)

```
class Being():
```

Explanation

We annotate classes and functions with triple quotes.

```
class Being():  
    """Top level generic class for all living things"""
```

Explanation

Class Variables are shared by all instances of the class. We will see in detail later.

```
class Being():  
    """Top level generic class for all living things"""  
  
    # Class Variables  
    population = 0
```

Explanation

Methods whose names are surrounded by 2 underscores (`--XXX--`) are *internal* methods. They are not meant to be called by the user; they are automatically called based on varying situations.

```
class Being():  
    """Top level generic class for all living things"""  
  
    # Class Variables  
    population = 0  
  
    def __init__(self, name, hp, mp, race):
```

Explanation

`__init__()` is automatically called upon the creation of an object of the class.

```
def __init__(self, name, hp, mp, race):  
    # Object Variables  
    self.name = name  
    self.hp = hp  
    self.mp = mp  
    self.race = race  
  
    Being.population += 1
```

Explanation

Class methods are same as the functions we have learned, but for one **difference**. They need an extra argument at the beginning of the parameter list.

But we **do not** pass a value for this parameter when we **use** it. The parameter is used to indicate *itself*, hence the **self**. (Just a convention, but follow it.)

```
population = 0

def __init__(self, name, hp, mp, race):
    # Object Variables
```

Explanation

The fields(object variables) are created by `__init__`.

```
def __init__(self, name, hp, mp, race):  
    # Object Variables  
    self.name = name  
    self.hp = hp  
    self.mp = mp  
    self.race = race
```


Explanation

To see how we use class/object variables, see the *die(self)* method.¹

This is used when a battle arises (remember, we were pretending to make an RPG game).

```
def die(self):  
    self.hp = 0  
    print(self.name, " is dead.")
```

¹note the *self*!

Explanation

Just like how we use *self* to access object variables, we can access Class Variables by their class name (Here, *Being*).

Note that when an object changes its *class* variable, other objects also see the change.

(*Class* variables are not unique to the object).

```
Being.population -= 1  
print(Being.population,  
      "being is alive on middle earth")
```

Using Classes

We usually put Class definitions in different files, but for the sake of simplicity, lets do it in the same file.

We create an object of a class like the following.

```
boss = Being("Smaug", 10, 5, "Dragon")  
you = Being(name="Your Name", hp=10,  
            mp=5, race="human")
```

Using Classes

Two things to note

1. We didn't call `__init__`.
2. We didn't add `self`.

```
boss = Being("Smaug", 10, 5, "Dragon")
```

Using Classes

We can explicitly use the names of the parameters, for better understanding of the code.¹

```
you = Being(name="Your Name", hp=10,  
            mp=5, race="human")
```

¹We can actually do this with all functions.

The Dragon Slayer

We call an object's method like the following. Familiar?

```
boss.die()
```

Practice

Type and Try

Overriding Internal Functions

```
print((1,2,3)): (1, 2, 3)\nprint([1,2,3]): [1, 2, 3]\nprint(3): 3\nprint(boss): ?\nprint(you): ?
```


Overriding Internal Functions

To control how *print* prints a class, we can fill in

`--repr--`

The return value has to be of type *string*, and the return value is what is printed.

Practice

In our *Being* Class, define the `__repr__`, so that printing an object of *Being* Class prints its name, and race. (*i.e.* "This being is a Dragon, of name Smaug")

Combat

Now we implement combat for Beings. The combat method gets another *Being*, decrease its hp by 3, and if its hp is less or equal than zero, make it die.

Inheritance

One usage of classes is Inheritance.

Inheritance

The child inherits every thing about its parent, and $+\alpha$.

Inheritance

```
class MyList(list):
    """a list that keeps track of max and min"""
    def __init__(self, iterable):
        print("I am created")
        super().__init__(iterable)
        self._max = max(iterable)
        self._min = min(iterable)

    # # always return 100, just for demonstrational purposes
    # def __len__(self):
    #     return 100

a = MyList((1, 2, 3, 4, 5, 100, 1000))
a.append(-100)
a[3] = 10000
```

Explained

MyList(list) means that this class will inherit from *list*.

```
class MyList(list):
```

Explained

The *super()* returns the parent class. We use *super()* to access the parent classes data methods etc. Here, we initiate the parent first so that parameters are automatically filled in.

```
class MyList(list):  
    """a list that keeps track of max and min"""  
    def __init__(self, iterable):  
        print("I am created")
```


Explained

min(lst), *max(lst)* takes time proportional to N .

Here, we keep track of min and max so that it can be known regardless of size.

(Of course, there is no free lunch, there is extra cost of comparing at append.)

```
class MyList(list):  
    """a list that keeps track of max and min"""  
    def __init__(self, iterable):  
        print("I am created")  
        super().__init__(iterable)  
        self._max = max(iterable)
```

Explained

Here, we override the parent's `__len__`, which determines the value returned when we do `len(lst)`.

```
self._min = min(iterable)  
  
# # always return 100, just for demonstrational purposes
```

Explained

We override the `append()` of list, so that we keep track of `_min` and `_max`.

After updating `_min` and `_max`, we insert to the list via `super()` call.

```
# return 100  
  
a = MyList((1, 2, 3, 4, 5, 100, 1000))  
a.append(-100)  
a[3] = 10000
```

Question

Are we done implementing MyList so that it correctly keep track of min and max?

Question

NO

The Catch

Everything that can be done to a *list* can be done to a *MyList*.
pop(), *del*, *insert()*, *mylst[3] = 4*, *mylst[3 : 4]*... you name it.

This means that to correctly keep track of min & max, we have to override *every single* method of a list that is capable of changing its contents.

Can you remember all of them?

Composition over Inheritance¹

So, it is often wise to compose your class with a list, rather than inheriting it.

¹Look this up in Google

MyList Composition Ver.

We can control the methods we provide for interaction with the internal list.

```
class MyList():
    def __init__(self, iterable=[]):
        self.lst = list(iterable)
        self._max = max(self.lst)
        self._min = min(self.lst)

    def __len__(self):
        return len(self.lst)

    def __repr__(self):
        return str(self.lst)

    def __getitem__(self, index):
        return self.lst[index]

    def append(self, item):
        if item < self._min:
            self._min = item
        if item > self._max:
            self._max = item
        self.lst.append(item)
```


Inheritance?

We might use inheritance when the child has to provide every method its parents provide + α .

(*i.e.* The Gun class that inherits the Weapon class in an RPG game?).

But Design Patterns are a complicated subject by itself, so we won't deal it in detail.