Python 101

Lec06

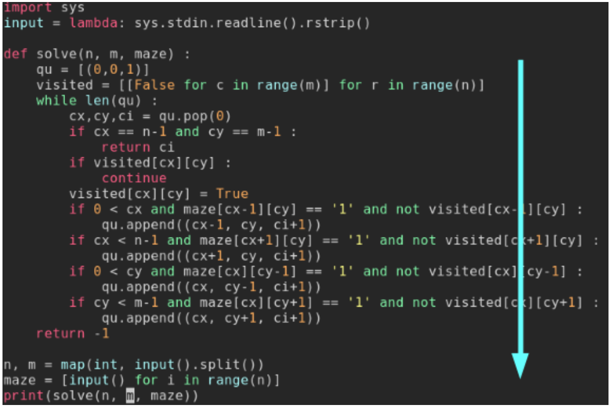
Classes

thoum

May 16, 2019

Programming up till now

Procedure-Oriented Programming
We pass values to functions, get values, pass them to another function....



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

Creating Classes of our own

We don't usually use classes so much unless we start writing bigger programs.
The usual Class tutorials force us to create boring examples, like a barking dog and a meowing cat.
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

```python
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 left standing 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.

```python
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. We will look into this later on.

```python
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__.
In English, its like saying my*self*'s *name* is *name*(given 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 show 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.")
```

[1]note 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 left standing 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")
```

[1]We 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)
print([1,2,3]): [1, 2, 3]
print(3): 3
print(boss): ?
print(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.