Intoduction to Python

Mike Korcha, webmaster@cse-club.com

Fall 2014

Introduction

In this workshop, we will be creating a basic text-based game to learn the basics Python programming language. By the end of this workshop, you will learn:

- The syntax of the Python programming language
- Looping structures
- Control statements
- Basic data structures
- Input and output

Getting Set Up

To complete this workshop, you'll need the following installed:

- Your favourite text editor (gedit, Sublime Text, Notepad++, etc)
- Python 3.x.x (the more recent the better)

Python 2.7 will also work for this workshop due to certain language rules, however I prefer working with the most current stable version.

Getting Started

Every language has a version of a "Hello World" program. Start up your text editor, and write:

```
print('Hello world!')
```

Save it as `main.py` and run it by running the following command in your terminal:

```
python3 main.py
```

You should see "Hello world!" printed in your terminal. Note that this will be the way we run our program throughout the workshop.

Now let's try adding some personalization to this message. Clear your file, then let's ask the user for their name:

```
name = input('What is your name? ')
```

Here, we create a variable and put what is recieved as input into it. Notice that we don't have to define variables or a type for them. They can be changed to any type at any time, and declared and used at any time within scope.

Next, let's add the name to a string and output it:

```
print('Hello, {0}'.format(name))
```

This syntax may seem a little weird, so allow me to explain. Strings are container objects, which contain characters. Whenever you create a string, whether in a variable or in place as we have here, you can access it's operations and methods. The `format` method allows you to replace a bracketed expression with those in the arguments.

Setting up the Dungeon

Now, let's clear out our file again. We're going to be creating a simple text-based game, where the player explores a small dungeon and can possibly find a sword or magic, fight a dragon, and win by finding the treasure at the end. To do this, we'll first need to create our dungeon:

```
dungeon = [
#    0    1    2    3    4
    'x', 'x', 'x', 'x', 'x', # 0
    'x', 'x', 'd', 't', 'x', # 1
    'x', 'x', 'O', 'x', 'x', # 2
    'x', 'm', 'O', 'O', 'x', # 3
    'x', 'x', 'x', 'O', 'x', # 4
    'x', 's', 'O', 'O', 'x', # 5
    'x', 'x', 'x', 'x', 'x'] # 6
```

Here, we create a list. A list is a container that can contain elements of various types and can be iterated over. It's similar to an array or vector in C++. Each character represents a different 'tile' in our game -

- *x* for spaces that cannot be traversed
- *O* for spaces that can be traversed
- *d* for the location of the dragon
- *t* for the location of the treasure
- *s* for the location of the sword
- *m* for the location of the magic

As it's a single-dimensional list, we need a way to address individual spaces. Let's write a function to get the index of the desired space, based on the coordinates:

```
def getIndex(x, y):
    return y * 5 + x
```

In Python, we use the `def` keyword to define that we're creating a function. Arguments don't need a given type, for the same reason that variables don't. Notice the colon. Everything in the next indentation block belongs to everything under that colon. This means that *indentation is incredibly important*.

The formula gets the right value by multiplying the y value by the size of the row, then adding the x value. I've commented the coordinate system onto the list.

Let's also create a function to get the value of the tile in the dungeon:

```
def getTile(x, y):
    return dungeon[getIndex(x, y)]
```

Accessing values in a list is just like arrays in other languages - just pass the index into square brackets.

Let's run and test this by adding a couple of `print`s to the end of our file and running it:

```
print(getIndex(3, 5))
print(getTile(3, 5))
```

Try a few different coordinates to verify correctness, then remove these lines.

Defining the Player

Next, we need to define what makes our player: their hitpoints, whether or not they have the sword and the magic, their combat status, and their coordinates in the dungeon:

```
player = {
    'x'      : 3,
    'y'      : 5,
    'hp'     : 10,
    'sword'  : False,
    'magic'  : False,
    'combat' : False
}
```

This is called a dictionary. A dictionary is similar to a map or an associative array from other languages. However, each item in the dictionary can be defined with any other data type.

We need to know which directions the player can do in, so let's write another function:

```
def movableDirections(player):
    directions = []

    if getTile(player['x'], player['y'] - 1) != 'x':
        directions.append('N')

    if getTile(player['x'], player['y'] + 1) != 'x':
        directions.append('S')

    if getTile(player['x'] + 1, player['y']) != 'x':
        directions.append('E')

    if getTile(player['x'] - 1, player['y']) != 'x':
        directions.append('W')

    return directions
```

Here, we check in each cardinal direction to see which are available, and add them to the list if they are. At the end, we return the list. Now, let's create the main game loop:

```
command = None
gameOver = False

while not gameOver:
```

This loop will run only while the game is not over. Later on, we'll define what causes the game to end. For now, let's focus on movement:

```
for direction in movableDirections(player):
    print("{0} - move {0}".format(direction))

    command = input('What would you like to do? ').upper()

    moveCharacter(command)
```

Here, we go through each direction the player can move through and output them. Then, we ask the player which direction to move in. Finally, we call the `moveCharacter` function, which we will implement outside the loop:

```
def moveCharacter(direction):
    if direction in movableDirections(player):
        if direction == 'N':
            player['y'] -= 1

        if direction == 'S':
            player['y'] += 1

        if direction == 'E':
            player['x'] += 1

        if direction == 'W':
            player['x'] -= 1
```

Here, we check to make sure the direction is able to be moved to by the player. The `in` operator checks for inclusion in a collection. Then, we check what the direction was, and modify the coordinates accordingly.

Run your game, exploring your map. If you want, output the tile at the player location to see what's going on.

Discovering the Sword

Now that we are able to move our player around, lets allow them to pick up the sword when they encounter it. We're going to change our game loop a bit to accomodate different kinds of tiles:

```
while not gameOver:
    tile  = getTile(player['x'], player['y'])
    index = getIndex(player['x'], player['y'])

    if tile == 's':
        player['sword'] = True
        dungeon[index]  = 'x'

        print('You have found the sword!')

    else:
        for direction in movableDirections(player):
            print("{0} - move {0}".format(direction))

        command = input('What would you like to do? ').upper()

        moveCharacter(command)
```

Now, instead of only asking for movement, we check first if the tile is the sword tile. If so, we set the player's sword value to true, and set the tile to be non-traversable. We also let the player know. Otherwise, we move as normal. We can do the same for the magic, putting this between the `if` and `else` statements:

```
elif tile == 'm':
    player['magic'] = True
    dungeon[index]  = 'x'

    print('You have found the hidden defensive magic!')
```

`elif` is the same as else-if in other languages, just shorter.

While we're adding these discoveries, lets make sure to add the most important one - treasure! This is one of two ways that the game ends - the other being death!

```
elif tile == 't':
    gameOver = True

    print('Congratulations, you have found the treasure!')
```

If we find the treasure, we tell the player they win, and set the gameOver variable to True. This will cause the loop to end at the end of the current iteration, ending the program.

Fighting the Dragon

We have one last component to take care of - fighting the dragon! First, let's add the dragon's HP to our declarations:

```
dragonHp = 12
```

Let's also add another `elif` block to our main loop.

```
elif tile == 'd':
```

The rest of this section will happen in this block. First, let's check to make sure that we haven't started the fight yet. If we haven't let the user know that the dragon appears:

```
if not player['combat']:
    player['combat'] = True

    print('The dragon appears before you!')
```

While we're at it, let's be a little condescending to those who don't find the sword. They'll be in a lot of trouble!

```
if not player['sword']:
    print('A weapon would\'ve been useful...')
```

Every time we input a command, we should let the player know how the fight is going, so let's output each side's health:

```
print('Dragon HP: {0}'.format(dragonHp))
        print('Player HP: {0}'.format(player['hp']))
```

Let's also show available options for the user in combat, and ask them for their choice:

```
if player['sword']:
    print('S - stab with your sword')

print('P - punch with your fist')

command = input('What will you do? ').upper()
```

We can't just have static damage, that'd be boring! Let's add a bit of randomness. At the top, we'll import the `random` module:

```
from random import randint
```

When importing from another module, you can specify specific variables, classes, and functions to get. In this case, we're importing the function `randint`.

Now, lets check our input that we just recieved:

```
playerDamage = 0

if command == 'S':
    if player['sword']:
        playerDamage = randint(2, 5)
        dragonHp -= playerDamage
        print('You stab the Trogdor for {0} damage!'
          .format(playerDamage))

    else:
        player['hp'] -= 2
        print('While you try to stab with your
          pretend sword, you hurt yourself for 2 damage')


elif command == 'P':
    playerDamage = randint(1, 2)
    dragonHp -= playerDamage
    print('You punch the dragon for {0} damage!'
      .format(playerDamage))
```

Here, we check our input, and if a sword is chosen, we'll make sure the player actually has the sword. Otherwise, let's penalize them for trying to pull a fast one. The `randint` function we imported earlier allows us to generate a number within a range of values.

Now, before the dragon can attack, lets check to make sure the dead isn't attacking the living:

```
if dragonHp <= 0:
    print('You have slain the burninator!')
    dungeon[index] = '0'
```

If the dragon is dead, we'll make the tile a walkable tile.

We're going to use the magic as a defense against the dragon, which has two attacks - a fire attack and a punching attack:

```
else:
    if randint(1, 4) == 1:
        if player['magic']:
            print('The dragon breathes fire, but the magic
              you found earlier protects you!')

        else:
            dragonDamage = randint(4, 6)
            player['hp'] -= dragonDamage
            print('The dragon breathes fire and burns you
              for {0} damage!'.format(dragonDamage))

    else :
        dragonDamage = randint(2, 3)
        player['hp'] -= dragonDamage
        print('The dragon punches you for {0} damage!'
          .format(dragonDamage))
```

Finally, we'll check to make sure the dragon hasn;t slain the player. If so, let them know and break out of the loop:

```
if player['hp'] <= 0:
    print('Oh dear, you are dead!')
    gameOver = True
```

Save and test your program. If everything works correctly, then congratulations! You've created your game, and have a working knowledge of the Python programming language.

Future Work

Here are a few things to try to help further your learning of the language, and improve your game along the way.

- Using classes, create a Player object. Import it, and have any operations.
- Have other monsters in the dungeon to fight. Randomly assign them hitpoints and power levels.
- Create a more sophisticated combat system.
- Randomly generate new dungeons for each run of the game.