

# MAKE YOUR OWN TEXT ADVENTURE WITH PYTHON



PHILLIP JOHNSON

# Make Your Own Text Adventure With Python

Phillip Johnson

This book is for sale at <http://leanpub.com/pythontextadventure>

This version was published on 2016-12-03



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

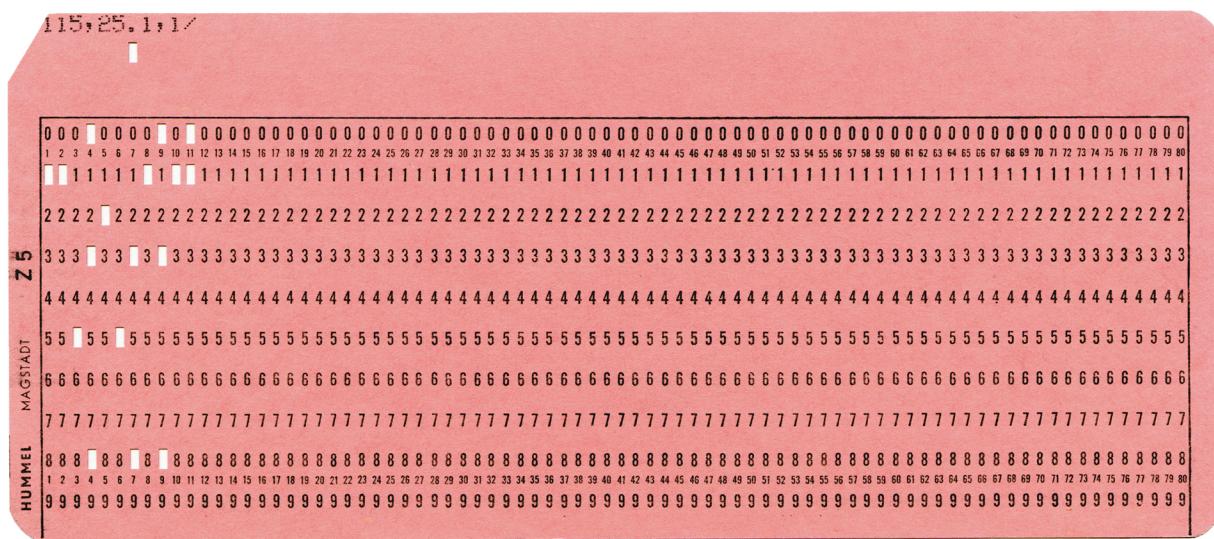
© 2015 - 2016 Phillip Johnson

# Contents

<b>Chapter 2: Your First Program . . . . .</b>	<b>1</b>
Creating a module . . . . .	1
Writing code . . . . .	2
Running Python programs . . . . .	2
Homework . . . . .	3
<b>Chapter 11: Building Your World . . . . .</b>	<b>4</b>
The X-Y grid . . . . .	4
Moving in the world . . . . .	7

# Chapter 2: Your First Program

When you open up an application on your computer, such as an Internet browser, at the lowest level the CPU is executing instructions to move around bytes of information. Early programs were painstakingly written on punch cards such as this:



Credit: Wikipedia user Harke

Thankfully, we have decades of improvements to computer programming that make it much easier to write those instructions! Now, programming languages lie on a spectrum of “lower-level” to “higher-level” with languages like C and C++ being “lower-level” and languages like Python and Ruby being “higher-level”. By design, higher-level languages allow programmers to ignore much of the behind-the-scenes details of computer programs. This is one reason why Python is often recommended as a first programming language.

To get started, create a folder on your computer where you will do all of the work for your game. From here on out, this directory will be referred to as the **root** directory of your project.

# Creating a module

Python code is organized into files called **modules**. Each module usually contains a significant amount of code that is all logically related. For example, our project will contain a module that runs the game, another module that contains the code for managing the enemies, another for the world, etc. To create your first module, navigate to your root directory and create an empty file called `game.py`.

## Writing code

When writing code, it's important that you write the code *exactly* as it appears in this book. However, I don't recommend simply copying and pasting. Especially when starting out, muscle memory will help you learn faster. If you run into errors, review your code line-by-line and check for typos, wrong casing, misplaced symbols, etc. If you really can't figure out the problem, then and only then is it OK to copy code. But always be sure to read over the pasted code to find your error.

I need to mention here one of the more controversial parts of the Python syntax: meaningful whitespace. Many languages ignore spaces and tabs, but Python does not. This means that you can run into problems caused by characters you can't (easily) see! Because of that, you need to decide if you will use tabs or spaces to indent your code. Most Python programmers have chosen to use spaces so I will stick with the convention of using four spaces to indent the code for this book. *If you choose to use tabs and copy code you must switch the indentation to tabs!* Some text editors can do this for you with a toolbar command. If yours does not, you should be able to replace four spaces with "\t" (which means "tab").

With that in mind, let's write your first line of code. Open `game.py` and add the following line:

`game.py`

---

1 `print("Escape from Cave Terror!")`

---



### Customization point

You can change the name of your game by replacing the text inside the quotation marks. Think about the scene your game takes place in. Is it a medieval forest, an alien spaceship, or a crime-ridden city?

## Running Python programs

Now, we'll execute the code we just wrote. Start by opening a command prompt or terminal and then use the `cd` command to navigate to your project root directory. For example, `cd ~/Documents/code/learn-python-game` or `cd C:\Code\my_python_adventure`. Finally, run the following command:

`python game.py`

(Note: Depending on how you installed Python, you may need to run `python3 game.py`.)

If all went well, you should see "Escape from Cave Terror!" printed out to the console. Congratulations! You just wrote your first Python program.

## Homework

1. Make a new module called `calculator.py` and write code that will print out “Which numbers do you want to add?” to the console.
2. Run the calculator program and make sure it works correctly.
3. Try removing the quotes from the code, what happens?

# Chapter 11: Building Your World

Early on, we gave our player the ability to move throughout the game world, but so far that world has been only a figment of our imaginations. In this chapter, we will finally create the world for the player to move around in.

## The X-Y grid

Since this is a text adventure, we only need to worry about the player moving in two directions: forward/backward and left/right. This allows us to build the world as if we were looking from above down onto the player, similar to Pacman or Chess. To keep track of where everything is, we use a coordinate plane similar to the one you learned about in math class. The X-axis represents the horizontal position of game objects and the Y-axis represents the vertical position of the game objects. However, in game programming, we orient the grid slightly differently.

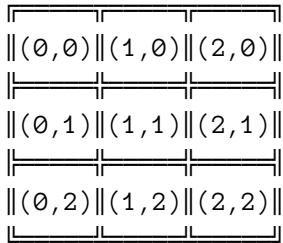
A typical coordinate plane in math and science looks like this:

$$\begin{array}{ccc} (0,2) & - & (1,2) & - & (2,2) \\ | & & | & & | \\ (0,1) & - & (1,1) & - & (2,1) \\ | & & | & & | \\ (0,0) & - & (1,0) & - & (2,0) \end{array}$$

But in game programming, we flip the Y-axis so that the numbers increase downwards instead of upwards.

$$\begin{array}{ccc} (0,0) & - & (1,0) & - & (2,0) \\ | & & | & & | \\ (0,1) & - & (1,1) & - & (2,1) \\ | & & | & & | \\ (0,2) & - & (1,2) & - & (2,2) \end{array}$$

If we label the spaces instead of the intersection, we end up with a grid of cells.



We can imagine each grid cell to be a different part of the cave (or room in a spaceship, or city block). The player will be in one cell at any time and in that cell they may encounter an enemy, loot, or some lovely scenery. They can move from cell to cell by using the already defined actions North, South, East, and West. These actions correspond to Up ( $y - 1$ ), Down ( $y + 1$ ), Left ( $x - 1$ ), and Right ( $x + 1$ ), respectively.

Before we go much farther, let's get some of this written up into code. Start by creating a new module `world.py` with the following tile classes.

```
1  class MapTile:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def intro_text(self):
7          raise NotImplementedError("Create a subclass instead!")
8
9
10 class StartTile(MapTile):
11     def intro_text(self):
12         return """
13             You find yourself in a cave with a flickering torch on the wall.
14             You can make out four paths, each equally as dark and foreboding.
15             """
16
17
18 class BoringTile(MapTile):
19     def intro_text(self):
20         return """
21             This is a very boring part of the cave.
22             """
23
24
25 class VictoryTile(MapTile):
26     def intro_text(self):
```

```
27     return """
28     You see a bright light in the distance...
29     ... it grows as you get closer! It's sunlight!
30
31
32     Victory is yours!
```



## Customization point

Change the intro text of the tiles to fit your game world.

The class `MapTile` is the superclass which defines the initializer. The following subclasses are specific types of tiles in the game. (Don't worry, we'll get rid of `BoringTile`!) The `intro_text()` method will be used in the next section, but you should be able to guess its purpose. Notice that we raise an exception if a naughty programmer tries to use `MapTile` directly.

You may have noticed the triple quote (""""") surrounding the intro text. Python allows us to write multi-line strings by surrounding text with triple quotes. This can make it easier to write long strings.

Having defined the classes, we need to place them into a grid.

```
35 world_map = [
36     [None,VictoryTile(1,0),None],
37     [None,BoringTile(1,1),None],
38     [BoringTile(0,2),StartTile(1,2),BoringTile(2,2)],
39     [None,BoringTile(1,3),None]
40 ]
```

This list of lists is a way of representing the grid pattern. The "outside" list represents the Y-axis. So the first item in the "outside" list is the entire first row, the second item in the "outside" list is the entire second row. Each "inside" list represents a single row. The first item in the first row is the tile in the top-left corner in the grid. The last item in the last row is the tile in the bottom-right corner of the grid. The `None` value is used for the grid spaces where we do not want a map tile to exist.

For convenience, let's also add a function that locates the tile at a coordinate.

```
42 def tile_at(x, y):
43     if x < 0 or y < 0:
44         return None
45     try:
46         return world_map[y][x]
47     except IndexError:
48         return None
```

The syntax `world_map[y][x]` may look confusing, but that's because we're working with a list of lists. The `world_map[y]` part selects the row of the map and adding `[x]` selects the specific cell in that row. Catching `IndexError` will handle the situation where we pass in a coordinate greater than the bounds of the map and `if x < 0 or y < 0` handles coordinates smaller than the bounds of the map. Without this function, we'd have to continually check the bounds of the world whenever we wanted to see if a tile exists.

## Moving in the world

One of the first features we added to the game was getting user input for moving in the game world. However, until now, those have just been placebo actions. To make the player move, we need to add X-Y coordinates to the `Player` class to represent the player's position, and we need to add methods that modify those coordinates. Start by adding `self.x` and `self.y` in the initializer.

```
4 class Player:
5     def __init__(self):
6         self.inventory = [items.Rock(),
7                           items.Dagger(),
8                           'Gold(5)',
9                           'Crusty Bread']
10
11     self.x = 1
12     self.y = 2
```

Next, add these methods inside the class:

```

34     def move(self, dx, dy):
35         self.x += dx
36         self.y += dy
37
38     def move_north(self):
39         self.move(dx=0, dy=-1)
40
41     def move_south(self):
42         self.move(dx=0, dy=1)
43
44     def move_east(self):
45         self.move(dx=1, dy=0)
46
47     def move_west(self):
48         self.move(dx=-1, dy=0)

```

If you didn't read through the homework answers, the syntax `move(dx=0, dy=-1)` may be new to you. This code calls the `move` method using **named parameters**. Named parameters are never required, but they can make it easier to read code, especially when you have parameters of the same type in the method. The names `dx` and `dy` come from math and mean "change in *x*" and "change in *y*" respectively. So the `move()` method accepts a generic change in the *x*- and/or *y*-direction and the specific methods define the amount of the change.

Finally, our main game loop needs to actually use these methods instead of just printing out the placeholder text. Jump over to `game.py` and change the `play()` function as follows.

```

12     if action_input in ['n', 'N']:
13         player.move_north()
14     elif action_input in ['s', 'S']:
15         player.move_south()
16     elif action_input in ['e', 'E']:
17         player.move_east()
18     elif action_input in ['w', 'W']:
19         player.move_west()
20     elif action_input in ['i', 'I']:
21         player.print_inventory()

```

Now the player will be able to actually move around in the map, but we should also display the intro text for each tile so the player knows where they are. Don't forget to import the `world` module.

```
1 from player import Player
2 import world
3
4
5 def play():
6     print("Escape from Cave Terror!")
7     player = Player()
8     while True:
9         room = world.tile_at(player.x, player.y)
10        print(room.intro_text())
11        action_input = get_player_command()
```



## Help! What's an AttributeError?

A very common problem at this point in the game is that you get an error that says `AttributeError: 'NoneType' object has no attribute 'intro_text'.`

### What does it mean?

It means that the Python code says to run the `intro_text()` method on an object, but that object is actually the `None` type.

### Why does it happen?

The error occurs when the player moves into a room that doesn't exist. More specifically, when the player moves into a part of the map that is marked `None`.

### How do I fix it?

If the error shows up right away it probably means the starting location for your player is wrong. Check the `__init__` of the `Player` class and make sure the `self.x` and `self.y` coordinates are correct. Remember to start counting at zero!

If the error shows up while moving around, you are moving into a room that doesn't exist. If you want the room to exist, change your map. If you moved there by mistake, you found a bug that we'll fix soon!

You should be able to test the game out now and verify that you can move around through the world. There's some bugs right now. Notably, the game doesn't end when you reach the `VictoryTile` and the player can also wrap around the map. We'll fix those, but for now, enjoy the fact that this is starting to feel more like a game!