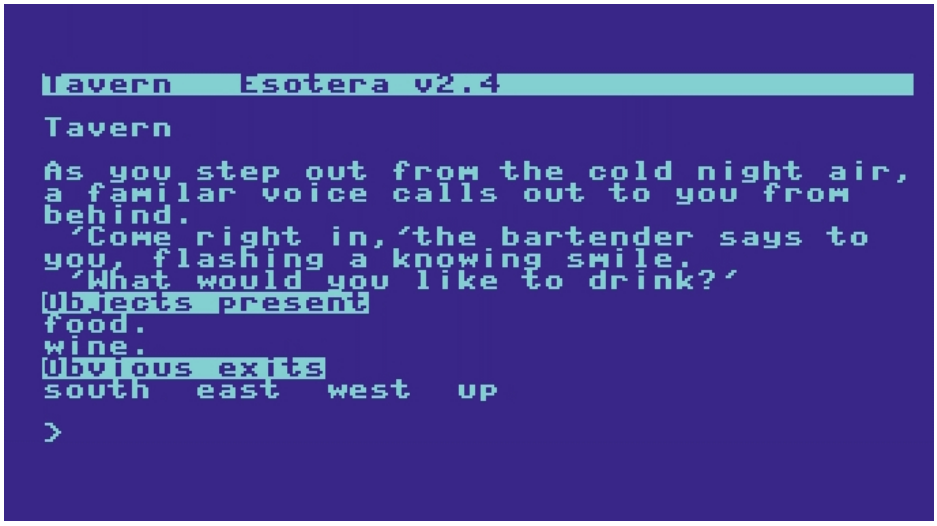
inventwithpython.com

Making a Text Adventure Game with the cmd and textwrap Python Modules

Al Sweigart

43-55 minutes

Text adventures are an old game genre where the entire world is textually described as a series of rooms. Play involves entering simple text commands such as "move north" or "eat pie". Each room in the game world has its own text description, items on the ground, and exits to adjacent rooms. "Room" is a general name for a single area in the game world: a room can be a large open canyon or the inside of a wardrobe. Multi-user text adventures, called MUDs or Multi-User Dungeons, were the precursor to modern MMORPGs. You can still play MUDs today by finding them on The Mud Connector.



Text adventures are easyto make because they don't require graphics. This tutorial uses two Python modules, `cmd` and `textwrap` and makes minimal use of object-oriented programming, but you don't have to know OOP concepts to follow. (But in general, text adventures would do very well with an object-oriented approach.) This tutorial is for beginner Python 3 programmers.

The code for Text Adventure Demo is available on GitHub.

The `cmd` module provides a generic command-line interface that has several useful features:

- **Tab-completion** - Type a partial command and press tab, and the game can guess the rest of the command.
- **History browsing** - Press the up arrow key to bring up the previously entered commands.
- **Automatic help** - A help system for commands is automatically generated for your commands.

The Python Debugger (tutorial here) in the `pdb` module makes use of the `cmd` module for its command line interface.

To get tab-completion to work on Windows, download and install this module: https://pypi.python.org/pypi/pyreadline/2.0 (On Windows with Python 3, open a command window and type `cd c:\Python34\Scripts` and then `pip install pyreadline`.

Our program will be about turn-based, single-player, and about 800 lines long. You can download the complete version from here: Download textadventuredemo.py or look at the GitHub page.

Demo of Game Play

Here's what our text adventure game will look like. It won't have combat, but will have a small town that you can navigate while picking up and looking at different things. Money isn't implemented, but shops to buy and sell at are.

```
Text Adventure Demo!
====================

(Type "help" for commands.)

Town Square
===========
The town square is a large open space with a fountain in the
center. Streets
lead in all directions.

A welcome sign stands here.
A bubbling fountain of green water.
```

North: North Y Street
South: South Y Street
East: East X Street
West: West X Street

> **look sign**
The welcome sign reads, "Welcome to this text adventure demo.
You can type
"help" for a list of commands to use. Be sure to check out Al's cool
programming
books at http://inventwithpython.com"

> **look fountain**
The water in the fountain is a bright green color. Is that... gatorade?

> **north**
You move to the north.
North Y Street
==========
The northern end of Y Street has really gone down hill. Pot holes
are everywhere, as are stray cats, rats, and wombats.

A sign stands here, not bolted to the ground.

South: Town Square
East: Bakery
West: Thief Guild

> **look sign**
The sign reads, "Do Not Take This Sign"

> **take sign**
You take a sign.

> **inventory**
Inventory:
Do Not Take Sign Sign
Donut
Sword
README Note

> **look readme**
The README note reads, "Welcome to the text adventure demo.
Be sure to check out the source code to see how this game is put

together."

> **eat donut**
You eat a donut

> **eat readme**
You eat a README note

> **eat sword**
You cannot eat that.

> **inventory**
Inventory:
Sword
Do Not Take Sign Sign

> **e**
You move to the east.
Bakery
======
The delightful smell of meat pies fills the air, making you hungry.
The baker flashes a grin, as he slides a box marked "Not Human Organs" under a table with his foot.

A "Shopping HOWTO" note rests on the ground.

South: East X Street
West: North Y Street

> **list**
For sale:
- Meat Pie
- Donut
- Bagel

> **buy pie**
You have purchased a meat pie

> **buy pie**
You have purchased a meat pie

> **buy pie**
You have purchased a meat pie

> **inventory**
Inventory:

> Meat Pie (3)
> Sword
> Do Not Take Sign Sign
>
> > **quit**
> Thanks for playing!

The "Choose Your Own Adventure" Mistake

Beginners tend to write code that looks like a "Choose Your Own Adventure" book: the execution starts at the beginning of the program, then it jumps to another part of the code (just like the reader would turn to some page at a branching point in the book). I have an example of this type of program in my dragon2.py game. This works, but the program becomes unwieldy as it gets larger.

You won't have a `townSquare()` function that displays the Town Square text when called like so:

```
def townSquare():
    print('The town square is a large open space
with a fountain in the center. Streets lead in all
directions.')
    print()
    if TOWN_SQUARE_HAS_SIGN == True:
        print('A welcome sign stands here.')
    if TOWN_SQUARE_HAS_FOUNTAIN == True:
        print(getFountainDescription())
    print()
    print('North: North Y Street')
    print('South: South Y Street')
    print('East: East X Street')
    print('West: West X Street')
```

A better way to program a game is a more **data structure-centric** approach. This sounds very abstract. Here's an example of the data structure you'll have for locations in your game:

```
worldRooms = {
    'Town Square': {
        DESC: 'The town square is a large open
space with a fountain in the center. Streets lead
```

```
in all directions.',
        NORTH: 'North Y Street',
        EAST: 'East X Street',
        SOUTH: 'South Y Street',
        WEST: 'West X Street',
        GROUND: ['Welcome Sign', 'Fountain']},
    'North Y Street': {
        DESC: 'The northern end of Y Street has
really gone down hill. Pot holes are everywhere,
as are stray cats, rats, and wombats.',
        WEST: 'Thief Guild',
        EAST: 'Bakery',
        SOUTH: 'Town Square',
        GROUND: ['Do Not Take Sign Sign']},
        ...
```

You can see that the general structure of the `worldRooms` dictionary is:

```
worldRooms = {
    'Room Name': {
        DESC: 'Description of room.',
        NORTH: 'Name of room to the north.',
        GROUND: ['Name of an item on the ground in
this room', 'Another item name']
    },
        ...
}
```

And here's a data structure to hold the different types of items in the game:

```
worldItems = {
    'Welcome Sign': {
        GROUNDDESC: 'A welcome sign stands here.',
        SHORTDESC: 'a welcome sign',
        LONGDESC: 'The welcome sign reads,
"Welcome to this text adventure demo. You can type
"help" for a list of commands to use. Be sure to
check out Al\'s cool programming books at
http://inventwithpython.com"',
```

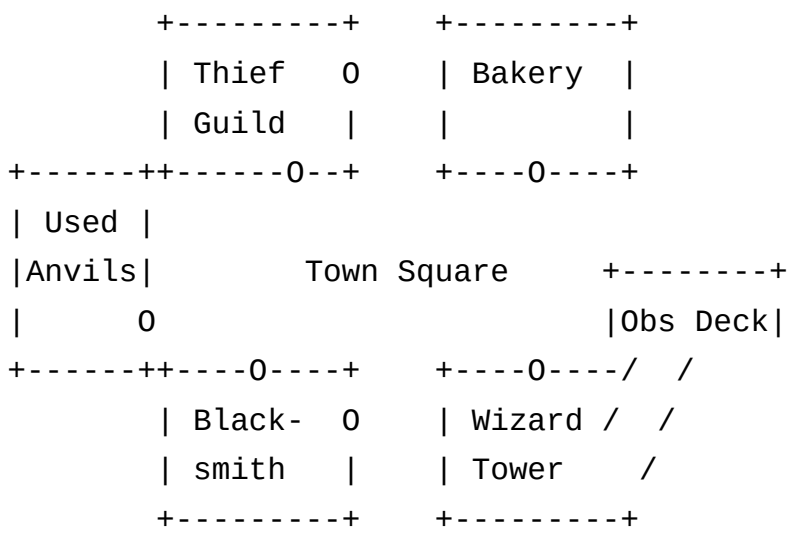
```
        TAKEABLE: False,
        DESCWORDS: ['welcome', 'sign']},
    'Fountain': {
        GROUNDDESC: 'A bubbling fountain of green
water.',
        SHORTDESC: 'a fountain',
        LONGDESC: 'The water in the fountain is a
bright green color. Is that... gatorade?',
        TAKEABLE: False,
        DESCWORDS: ['fountain']},
        ...
```

You can see that the general structure of the `worldRooms` dictionary is:

```
worldItems = {
    'Item Name': {
        GROUNDDESC: 'How this item is described
when on the ground.',
        SHORTDESC: 'A short description of this
item.',
        LONGDESC: 'A long description of this
item, used when the player looks at it.',
        TAKEABLE: True, # whether this item can be
taken and put in your inventory
        DESCWORDS: ['a word the player can use to
refer to this item', 'another word']
    },
        ...
}
```

By putting the rooms and items into data structures, now you can write a **single** `displayLocation()` function that can display *any* room in the game because it works off of the values in the `worldRooms` data structure. You can write code that deals with rooms and items generically. Writing a big game world will be much faster, just like how interchangeable parts speed up manufacturing and factories.

A map of the full world looks something like this:



(Data structures like these are normally put into classes in objet-oriented programming. To keep this text adventure demo simple, I'm using dictionaries and lists.)

Constant Variables and Global Variables

Start from scratch with a blank `textadventuredemo.py` file. Add a `#! python3` shebang line, some comments describing the program, and the `worldRooms` and `worldItems` data structures. Since this is mostly text and not code, you might want to just copy and paste the code from the snippet on GitHub.

The keys in the above data structures are constant variables:

```
DESC = 'desc'
NORTH = 'north'
SOUTH = 'south'
EAST = 'east'
WEST = 'west'
UP = 'up'
DOWN = 'down'
GROUND = 'ground'
SHOP = 'shop'
GROUNDDESC = 'grounddesc'
SHORTDESC = 'shortdesc'
LONGDESC = 'longdesc'
TAKEABLE = 'takeable'
EDIBLE = 'edible'
DESCWORDS = 'descwords'
```

The reason that constant variables are used instead of typing the string directly is to minimize the chance of errors. If you mistype the string `'desc'` as `'dsec'`, Python will not immediately complain because since `'dsec'` is a valid string. Later there will be a KeyError when you try to use `'dsec'` as a key, but this is a vague error message that could have many causes.

But if you are using constant variables instead, mistyping `DESC` as `DSEC` will immediately crash the program with a NameError exception since `DSEC` isn't a variable name. **The sooner your program crashes, the easier it is to find the crash-causing bug.** If your program silently continued to work, it will take longer to trace the original bug.

There is also the `SCREEN_WIDTH` constant, which will be used by the `textwrap` module to determine how wide the program should assume the screen is:

Th `SCREEN_WIDTH` constant will be explained in the next section.

The `location` and `inventory` global variables will keep track of the player's position and inventory. The `showFullExits` variable tracks whether to show full or brief versions of the current room's exits (this is explained later in the `displayLocation()` function). Add the following code to your program:

```
location = 'Town Square' # start in town square
inventory = ['README Note', 'Sword', 'Donut'] #
start with blank inventory
showFullExits = True
```

(Normally your programs should avoid using global variables, but they're used here to keep the text adventure demo code simple.)

Also, import the following modules into you program:

```
import cmd, sys, textwrap
```

The textwrap Module

The `textwrap` module can intelligently break up a string into multiple lines of a given width. You might think this is easily done with code such as the following, which breaks the `desc` string up into lines 80 characters wide:

```
desc = 'The town square is a large open space with
a fountain in the center. Streets lead in all
directions.'

for i in range(0, len(desc), 80):
    print(desc[i:i + 80])
```

But this can cut off a line right in the middle of words, like this:

```
The town square is a large open space with a
fountain in the center. Streets lea
d in all directions.
```

The `textwrap.wrap()` function is smart enough to avoid chopping lines in the middle of a word. It returns a list, with each line as a string in the list:

```
import textwrap
desc = 'The town square is a large open space with
a fountain in the center. Streets lead in all
directions.'

for line in textwrap.wrap(desc, 80):
    print(line)
```

```
The town square is a large open space with a
fountain in the center. Streets
lead in all directions.
```

To make it easy to change the screen width that `textwrap.wrap()` uses, pass the `SCREEN_WIDTH` constant for the second argument. That way, if you want to change the screen width you only need to change the `SCREEN_WIDTH` value.

Location and Movement Functions

When the player walks into a room, the displayed text looks something like this:

> Town Square
> ==========
> The town square is a large open space with a fountain in the center. Streets
> lead in all directions.

> A welcome sign stands here.
> A bubbling fountain of green water.
>
> North: North Y Street
> South: South Y Street
> East: East X Street
> West: West X Street

The data for each room is tied up in the `worldRooms` variable. You need a function that, given the name of the room as a string, will print out text like the above. This will need to print out:

- The room's name
- The room's description
- A list of the items on the ground in this room
- A list of all the available exits

Add the following code to your program:

```
def displayLocation(loc):
    """A helper function for displaying an area's
description and exits."""
    # Print the room name.
    print(loc)
    print('=' * len(loc))

    # Print the room's description (using
textwrap.wrap())
    print('\n'.join(textwrap.wrap(worldRooms[loc]
[DESC], SCREEN_WIDTH)))

    # Print all the items on the ground.
    if len(worldRooms[loc][GROUND]) > 0:
        print()
        for item in worldRooms[loc][GROUND]:
            print(worldItems[item][GROUNDDESC])

    # Print all the exits.
    exits = []
    for direction in (NORTH, SOUTH, EAST, WEST,
UP, DOWN):
```

```
            if direction in worldRooms[loc].keys():
                exits.append(direction.title())
        print()
        if showFullExits:
            for direction in (NORTH, SOUTH, EAST,
WEST, UP, DOWN):
                if direction in worldRooms[location]:
                    print('%s: %s' %
(direction.title(), worldRooms[location]
[direction]))
        else:
            print('Exits: %s' % ' '.join(exits))
```

There will also be a `moveDirection()` function that, given a north/south/east/west/up/down string argument, will change the `location` global variable to the new location.

But it will only do this if the direction was a valid one to make. If the player is able to move in the given direction, the new room's description will be displayed on the screen. Add the following code to your program:

```
def moveDirection(direction):
    """A helper function that changes the location
of the player."""
    global location

    if direction in worldRooms[location]:
        print('You move to the %s.' % direction)
        location = worldRooms[location][direction]
        displayLocation(location)
    else:
        print('You cannot move in that direction')
```

You'll see how these functions are used by the command line interface in a bit. But if you want to see how the code you have so far works, add the following temporary code:

```
# TEMPORARY CODE:
while True:
    displayLocation(location)
    response = input()
```

```
    if response == 'quit':
        break
    if response in (NORTH, SOUTH, EAST, WEST, UP,
DOWN):
        moveDirection(response)
```

Your program will now look like this source code on GitHub. When you run this program, you will be able to move around the game world by typing `north`, `south`, `east`, `west`, `up`, and `down`.

Notice a few things about this game world:

- Moving in an invalid direction results in `'You cannot move in that direction'` printed to the screen.
- At the top of the wizard's tower, the "Magical Escalator to Nowhere" room's up exit *leads to itself*. This means if you exit up from this room, you will arrive in the same room. You can effectively go up an infinite amount of times. This is a side effect of the way the room data structures are set up.

There are just a few more helper functions you'll need.

Item Helper Functions

These functions all work with the item data structure you have already set up. You won't quite see how these functions are needed right now, but they will be explained later.

Remember that the `worldItems` variable is a dictionary with key-value pairs that look like this:

```
    'Fountain': {
        GROUNDDESC: 'A bubbling fountain of green
water.',
        SHORTDESC: 'a fountain',
        LONGDESC: 'The water in the fountain is a
bright green color. Is that... gatorade?',
        TAKEABLE: False,
        DESCWORDS: ['fountain']},
```

The helper functions for items are:

- `getAllDescWords()` - Given a list of strings of item names, return a list with all of these items' DESCWORDS strings.

- `getAllFirstDescWords()` - Given a list of strings of item names, return a list with all of these items' first DESCWORDS strings.
- `getFirstItemMatchingDesc()` - Given a string and a list of item names, return a string of the name of the first item that has a DESCWORD string that matches the given string argument.
- `getAllItemsMatchingDesc()` - Given a string and a list of item names, return a list of strings of item names for items that have the given string argument as one of their DESCWORD strings.

Add the following code to your programs:

```
def getAllDescWords(itemList):
    """Returns a list of "description words" for
each item named in itemList."""
    itemList = list(set(itemList)) # make itemList
unique
    descWords = []
    for item in itemList:
        descWords.extend(worldItems[item]
[DESCWORDS])
    return list(set(descWords))

def getAllFirstDescWords(itemList):
    """Returns a list of the first "description
word" in the list of
    description words for each item named in
itemList."""
    itemList = list(set(itemList)) # make itemList
unique
    descWords = []
    for item in itemList:
        descWords.append(worldItems[item]
[DESCWORDS][0])
    return list(set(descWords))

def getFirstItemMatchingDesc(desc, itemList):
    itemList = list(set(itemList)) # make itemList
unique
```

```
    for item in itemList:
        if desc in worldItems[item][DESCWORDS]:
            return item
    return None


def getAllItemsMatchingDesc(desc, itemList):
    itemList = list(set(itemList)) # make itemList
unique
    matchingItems = []
    for item in itemList:
        if desc in worldItems[item][DESCWORDS]:
            matchingItems.append(item)
    return matchingItems
```

The `list(set(itemList))` code is a Pythonic way to get rid of duplicate values in a list by coverting the list to a set and back to a list. Note that this might change the order of the values in the list, since the set data type is unordered.

Creating a Command Line Interface with the cmd Module

The `cmd` module frees us from reinventing the wheel when creating the command line interface for the text adventure. First, create a class that subclasses `cmd.Cmd`. The `cmd` module will read any methods that begin with `do_`, `complete_` and `help_` for it's command, tab-completion, and help system features, respectively.

Here's a short example. Add the following code to your program:

```
class TextAdventureCmd(cmd.Cmd):
    prompt = '\n> '


    # The default() method is called when none of
the other do_*() command methods match.
    def default(self, arg):
        print('I do not understand that command.
Type "help" for a list of commands.')


    # A very simple "quit" command to terminate
the program:
    def do_quit(self, arg):
```

```
        """Quit the game."""
        return True # this exits the Cmd
application loop in TextAdventureCmd.cmdloop()

    def help_combat(self):
        print('Combat is not implemented in this
program.')
```

There are several cmd-specific things about this code:

- The string in the `prompt` member variable is printed when the player is expected to begin typing in a command. In your text adventure's case, this will be the `'\n>'` string: a newline followed by a > character.
- The `default()` method will be called by the command line interface when it cannot understand the command the player typed. In this case, a "I do not understand" message is printed to the screen.
- When the player types in a command, the command line interface checks if there are any methods of the same name as the command to handle it. For example, "quit" would be handled by `do_quit()`. If the player entered the "qwerty" command but there is no `do_qwerty()` method, the `default()` method is called instead.
- When a `do_<command>()` returns, the command line interface will let the player type in another command. If a `do_<command>()` returns the value `True`, the command line interface will stop asking for commands.
- The `do_quit()` method implements the "quit" command. Since it returns `True`, the command line interface will stop asking the player for commands.
- The docstring at the top of each `do_<command>()` function will automatically be used for the command line interface's help system.
- The `help_combat()` method implements the "help combat" command. You can add as many help topics as you want: typing "help" by itself lists all of them. Since there is no combat in this text adventure, this function prints a message telling the player there is

no combat.

- The `arg` parameter in `default()` and `do_quit()` will be explained later.

To kick off the command line interface code, you will need to call the `cmd.Cmd` object's `cmdloop()` method. Take out the temporary code in the `while True:` loop and add the following to your program:

```
if __name__ == '__main__':
    print('Text Adventure Demo!')
    print('====================')
    print()
    print('(Type "help" for commands.)')
    print()
    displayLocation(location)
    TextAdventureCmd().cmdloop()
    print('Thanks for playing!')
```

Your program should now look like this file on GitHub.

The cmd Help System

When you run this program, the movement commands won't work (they were implemented in the temporary `while` loop you removed) but you can run the "help", "help quit", and "help combat" commands:

> **help**
>
> Documented commands (type help):
> ==
> help quit
>
> Miscellaneous help topics:
> ==========================
> combat
>
> **help quit**
> Quit the game.
>
> **help combat**
> Combat is not implemented in this program.

Notice that the "quit" command's docstring has been used for the

"help quit" command. You can end this program by running "quit", which makes the command line interface call the `do_quit()` method. Since it is a `do_<command>()` method that returns `True`, it causes the `TextAdventureCmd().cmdloop()` function return and the execution reaches the end of the source code.

As a free feature provided by the command line interface, pressing the up arrow key will cycle through the command history. This provides an easy way to re-enter commands you previously typed.

The Move Commands and Tab Completion

In order to move the player around, you'll need a `do_north()` method for the "north" command, a `do_south()` method for the "south" command, and so on. All of these methods will simply call the `moveDirection()` function with the appropriate argument. Add the following to your program:

```
    # These direction commands have a long (i.e.
north) and show (i.e. n) form.
    # Since the code is basically the same, I put
it in the moveDirection()
    # function.
    def do_north(self, arg):
        """Go to the area to the north, if
possible."""
        moveDirection('north')

    def do_south(self, arg):
        """Go to the area to the south, if
possible."""
        moveDirection('south')

    def do_east(self, arg):
        """Go to the area to the east, if
possible."""
        moveDirection('east')

    def do_west(self, arg):
        """Go to the area to the west, if
```

```
possible."""
        moveDirection('west')

    def do_up(self, arg):
        """Go to the area upwards, if possible."""
        moveDirection('up')

    def do_down(self, arg):
        """Go to the area downwards, if
possible."""
        moveDirection('down')

    # Since the code is the exact same, we can
just copy the
    # methods with shortened names:
    do_n = do_north
    do_s = do_south
    do_e = do_east
    do_w = do_west
    do_u = do_up
    do_d = do_down
```

Because the "n" and "north" commands would do the same thing, we can create another function with the same code by assigning `do_n = do_north`, and so on.

To toggle the Boolean value in the global `showFullExits` variable, an "exits" command can be implemented in `do_exits()`. Add the following to your program:

```
    def do_exits(self, arg):
        """Toggle showing full exit descriptions
or brief exit descriptions."""
        global showFullExits
        showFullExits = not showFullExits
        if showFullExits:
            print('Showing full exit
descriptions.')
        else:
            print('Showing brief exit
```

```
descriptions.')
```

With this code, your program should look like this file on GitHub. When you run this program, the north/south/east/west/up/down commands will all work.

Inventory

The "inventory" command will display the contents of the `inventory` global variable. Rather than just a `do_inventory()` method that runs `print(inventory)`, there's additional code that checks if the player has more than one of a type of item. So an `inventory` value of `['Donut', 'Donut', 'Donut']` will display a single line: `Donut (3)`.

Add the following code to your program:

```
    def do_inventory(self, arg):
        """Display a list of the items in your
possession."""

        if len(inventory) == 0:
            print('Inventory:\n  (nothing)')
            return

        # first get a count of each distinct item
in the inventory
        itemCount = {}
        for item in inventory:
            if item in itemCount.keys():
                itemCount[item] += 1
            else:
                itemCount[item] = 1

        # get a list of inventory items with
duplicates removed:
        print('Inventory:')
        for item in set(inventory):
            if itemCount[item] > 1:
                print('  %s (%s)' % (item,
itemCount[item]))
```

```
            else:
                print('  ' + item)

    do_inv = do_inventory
```

To add an identical "inv" shortcut command, the `do_inv = do_inventory` line is added after the `do_inventory()` function.

Taking and Dropping Items

Every room in the `worldRooms` data structure has a `GROUND` key whose value is a list of items on the ground of that room. By removing items from the `inventory` list and adding them to the `GROUND` list, the player can "drop" the item in a certain room. The items will continue to be on the ground even if the player leaves the room and comes back. Similarly, by removing a value from the `GROUND` list and adding it to the `inventory` list, the player can "take" the item.

The "take" and "drop" commands are a bit more complicated: There is text that follows the "take" and "drop" words such as in "take sign" or "drop sword". The additional text is passed to the `do_<command>()` function for its `arg` parameter. So if the player enters the command "drop sword", the `do_drop()` method is called with `'sword'` passed for its `arg` parameter.

The drop and take methods will have to do more than modify `inventory` and the `GROUND` key's value. They must handle the player forgetting to specify what to take/drop, check that the item is actually on the ground/inventory, and if the object is "takeable" (that is, it's `TAKEABLE` key's value is `True`).

Also, items can be referred to by any of it's `DESCWORDS` from the `worldItems` data structure. For example, the sword item:

```
    'Sword': {
        GROUNDDESC: 'A sword lies on the ground.',
        SHORTDESC: 'a sword',
        LONGDESC: 'A longsword, engraved with the
word, "Exkaleber"',
        DESCWORDS: ['sword', 'exkaleber',
'longsword']},}
```

...has a DESCWORDS value of `['sword', 'exkaleber', 'longsword']`. So the "drop sword", "drop exkaleer", and "drop longsword" would all refer to the same sword to drop.

Add the following code to your program:

```
    def do_take(self, arg):
        """take <item> - Take an item on the
ground."""

        # put this value in a more suitably named
variable
        itemToTake = arg.lower()

        if itemToTake == '':
            print('Take what? Type "look" the
items on the ground here.')
            return

        cantTake = False

        # get the item name that the player's
command describes
        for item in
getAllItemsMatchingDesc(itemToTake,
worldRooms[location][GROUND]):
            if worldItems[item].get(TAKEABLE,
True) == False:
                cantTake = True
                continue # there may be other
items named this that you can take, so we continue
checking
            print('You take %s.' %
(worldItems[item][SHORTDESC]))
            worldRooms[location]
[GROUND].remove(item) # remove from the ground
            inventory.append(item) # add to
inventory
            return
```

```
        if cantTake:
            print('You cannot take "%s".' %
(itemToTake))
        else:
            print('That is not on the ground.')

    def do_drop(self, arg):
        """drop <item> - Drop an item from your
inventory onto the ground."""

        # put this value in a more suitably named
variable
        itemToDrop = arg.lower()

        # get a list of all "description words"
for each item in the inventory
        invDescWords = getAllDescWords(inventory)

        # find out if the player doesn't have that
item
        if itemToDrop not in invDescWords:
            print('You do not have "%s" in your
inventory.' % (itemToDrop))
            return

        # get the item name that the player's
command describes
        item =
getFirstItemMatchingDesc(itemToDrop, inventory)
        if item != None:
            print('You drop %s.' %
(worldItems[item][SHORTDESC]))
            inventory.remove(item) # remove from
inventory
            worldRooms[location]
[GROUND].append(item) # add to the ground
```

After you are finished, your program will look like this file on GitHub.

When you run the program, you will be able to drop and take items (provided they have the TAKEABLE setting).

Tab Completion

if the player types a partial command, such as "nor", they can press the Tab key and the command line interface will complete the command: "north". The command line interface knows this because "north" is the only command that begins with "nor". If there were other commands that began with "nor", pressing Tab would bring up a list of possible complete commands.

However, say that the player has an `inventory` value of `['sword', 'swingset', 'swampwater']` and had entered "drop sw" and pressed Tab. The command line interface knows all the commands because it can see what `do_%lt;command>()` functions there are. But items on the ground or in the inventory are specific to your program, so you will need a way to tell the command line interface what it should return for possible tab completions.

This is done by the `complete_<command>()` methods. These methods have the following parameters:

- `text` is the part after the command. If "drop sw" were being completed, `text` would be set to `'sw'`.
- `line` is the entire command that was entered. If "drop sw" were being completed, `line` would be set to `'drop sw'`.
- `begidx` is index in `line` where the last word begins. If "drop sw" were being completed, `begidx` would be `5`, which is where `'sw'` begins.
- `endidx` is index in `line` where the last word begins. If "drop sw" were being completed, `endidx` would be `7`, which is where `'sw'` ends.

When the Tab key is pressed, the command's `complete_<command>()` method is called and passed the command typed in so far. For example, typing "drop sw"-Tab will call the `complete_drop()` method and typing "take sw"-Tab will call the `complete_take()` method. If a

`complete_<command>()` method command doesn't exist, such as not `complete_qwerty()` method for "qwerty sw", then the command line interface does nothing.

Using the `getAllFirstDescWords()` helper function, you can add `complete_take()` and `complete_drop()` methods to determine what item the player is trying to take or drop. Add the following to your program:

```
    def complete_take(self, text, line, begidx,
endidx):
        possibleItems = []
        text = text.lower()

        # if the user has only typed "take" but no
item name:
        if not text:
            return
getAllFirstDescWords(worldRooms[location][GROUND])

        # otherwise, get a list of all
"description words" for ground items matching the
command text so far:
        for item in list(set(worldRooms[location]
[GROUND])):
            for descWord in worldItems[item]
[DESCWORDS]:
                if descWord.startswith(text) and
worldItems[item].get(TAKEABLE, True):
                    possibleItems.append(descWord)

        return list(set(possibleItems)) # make
list unique


    def complete_drop(self, text, line, begidx,
endidx):
        possibleItems = []
        itemToDrop = text.lower()
```

```
        # get a list of all "description words"
for each item in the inventory
        invDescWords = getAllDescWords(inventory)

        for descWord in invDescWords:
            if line.startswith('drop %s' %
(descWord)):
                return [] # command is complete

        # if the user has only typed "drop" but no
item name:
        if itemToDrop == '':
            return getAllFirstDescWords(inventory)

        # otherwise, get a list of all
"description words" for inventory items matching
the command text so far:
        for descWord in invDescWords:
            if descWord.startswith(text):
                possibleItems.append(descWord)

        return list(set(possibleItems)) # make
list unique
```

Your program will look like this file on GitHub. When you run the program, you will be able to type "drop sw", then press Tab, and the command will complete to "drop sword".

Looking at Things

The "look" command will have similar `do_look()` and `complete_look()` methods. Typing "look" will print the current room's `DEC` key's value. Typing "look exits" will print the names of all the adjacent rooms, while "look " will print the name of the room in that direction. The player can also type "look " to look at a item that is either on the ground or in their inventory.

Enter the following code for the "look" command:

```
    def do_look(self, arg):
```

```
        """Look at an item, direction, or the
area:
"look" - display the current area's description
"look <direction>" - display the description of
the area in that direction
"look exits" - display the description of all
adjacent areas
"look <item>" - display the description of an item
on the ground or in your inventory"""

        lookingAt = arg.lower()
        if lookingAt == '':
            # "look" will re-print the area
description
            displayLocation(location)
            return

        if lookingAt == 'exits':
            for direction in (NORTH, SOUTH, EAST,
WEST, UP, DOWN):
                if direction in
worldRooms[location]:
                    print('%s: %s' %
(direction.title(), worldRooms[location]
[direction]))
            return

        if lookingAt in ('north', 'west', 'east',
'south', 'up', 'down', 'n', 'w', 'e', 's', 'u',
'd'):
            if lookingAt.startswith('n') and NORTH
in worldRooms[location]:
                print(worldRooms[location][NORTH])
            elif lookingAt.startswith('w') and
WEST in worldRooms[location]:
                print(worldRooms[location][WEST])
            elif lookingAt.startswith('e') and
EAST in worldRooms[location]:
```

```
                print(worldRooms[location][EAST])
            elif lookingAt.startswith('s') and
SOUTH in worldRooms[location]:
                print(worldRooms[location][SOUTH])
            elif lookingAt.startswith('u') and UP
in worldRooms[location]:
                print(worldRooms[location][UP])
            elif lookingAt.startswith('d') and
DOWN in worldRooms[location]:
                print(worldRooms[location][DOWN])
            else:
                print('There is nothing in that
direction.')
            return

        # see if the item being looked at is on
the ground at this location
        item = getFirstItemMatchingDesc(lookingAt,
worldRooms[location][GROUND])
        if item != None:

print('\n'.join(textwrap.wrap(worldItems[item]
[LONGDESC], SCREEN_WIDTH)))
            return

        # see if the item being looked at is in
the inventory
        item = getFirstItemMatchingDesc(lookingAt,
inventory)
        if item != None:

print('\n'.join(textwrap.wrap(worldItems[item]
[LONGDESC], SCREEN_WIDTH)))
            return

        print('You do not see that nearby.')
```

```
    def complete_look(self, text, line, begidx,
endidx):
        possibleItems = []
        lookingAt = text.lower()

        # get a list of all "description words"
for each item in the inventory
        invDescWords = getAllDescWords(inventory)
        groundDescWords =
getAllDescWords(worldRooms[location][GROUND])
        shopDescWords =
getAllDescWords(worldRooms[location].get(SHOP,
[]))

        for descWord in invDescWords +
groundDescWords + shopDescWords + [NORTH, SOUTH,
EAST, WEST, UP, DOWN]:
            if line.startswith('look %s' %
(descWord)):
                return [] # command is complete

        # if the user has only typed "look" but no
item name, show all items on ground, shop and
directions:
        if lookingAt == '':

possibleItems.extend(getAllFirstDescWords(worldRooms[location]
[GROUND]))

possibleItems.extend(getAllFirstDescWords(worldRooms[location].g
[]))))
            for direction in (NORTH, SOUTH, EAST,
WEST, UP, DOWN):
                if direction in
worldRooms[location]:

possibleItems.append(direction)
            return list(set(possibleItems)) # make
```

```
list unique

        # otherwise, get a list of all
"description words" for ground items matching the
command text so far:
        for descWord in groundDescWords:
            if descWord.startswith(lookingAt):
                possibleItems.append(descWord)

        # otherwise, get a list of all
"description words" for items for sale at the shop
(if this is one):
        for descWord in shopDescWords:
            if descWord.startswith(lookingAt):
                possibleItems.append(descWord)

        # check for matching directions
        for direction in (NORTH, SOUTH, EAST,
WEST, UP, DOWN):
            if direction.startswith(lookingAt):
                possibleItems.append(direction)

        # get a list of all "description words"
for inventory items matching the command text so
far:
        for descWord in invDescWords:
            if descWord.startswith(lookingAt):
                possibleItems.append(descWord)

        return list(set(possibleItems)) # make
list unique
```

The program will now look like this file on GitHub.

Shops

Shops are rooms that have a SHOP key, such as the Bakery room:

```
    'Bakery': {
        DESC: 'The delightful smell of meat pies
```

```
fills the air, making you hungry. The baker
flashes a grin, as he slides a box marked "Not
Human Organs" under a table with his foot.',
        WEST: 'North Y Street',
        SOUTH: 'East X Street',
        SHOP: ['Meat Pie', 'Donut', 'Bagel'],
        GROUND: ['Shop Howto']},
```

The value for the `SHOP` is a list of items that the shop sells. Inside these rooms, the player can run the "list" command (to see what is for sale), the "buy" command (to purchase an item), and the "sell" command (to pawn an item from the player's inventory). For simplicity, any item can be sold to any shop. Also, money is currently not implemented in this game, so items in the shop are free.

Several things must be checked when the player tries to run these commands: Is the player currently in a shop room? Did the player forget to specify what they want to buy or sell? Does the shop sell what the player wants to buy? Does the player have the item they want to sell to the shop? The following code implements the "list", "buy", and "sell" commands and addresses all these issues. Add the following code to your program:

```
    def do_list(self, arg):
        """List the items for sale at the current
location's shop. "list full" will show details of
the items."""
        if SHOP not in worldRooms[location]:
            print('This is not a shop.')
            return

        arg = arg.lower()

        print('For sale:')
        for item in worldRooms[location][SHOP]:
            print('  - %s' % (item))
            if arg == 'full':

print('\n'.join(textwrap.wrap(worldItems[item]
```

```
[LONGDESC], SCREEN_WIDTH)))


    def do_buy(self, arg):
        """buy <item>" - buy an item at the
current location's shop."""
        if SHOP not in worldRooms[location]:
            print('This is not a shop.')
            return

        itemToBuy = arg.lower()

        if itemToBuy == '':
            print('Buy what? Type "list" or "list
full" to see a list of items for sale.')
            return

        item = getFirstItemMatchingDesc(itemToBuy,
worldRooms[location][SHOP])
        if item != None:
            # NOTE - If you wanted to implement
money, here is where you would add
            # code that checks if the player has
enough, then deducts the price
            # from their money.
            print('You have purchased %s' %
(worldItems[item][SHORTDESC]))
            inventory.append(item)
            return

        print('"%s" is not sold here. Type "list"
or "list full" to see a list of items for sale.' %
(itemToBuy))


    def complete_buy(self, text, line, begidx,
endidx):
        if SHOP not in worldRooms[location]:
```

```
            return []

        itemToBuy = text.lower()
        possibleItems = []

        # if the user has only typed "buy" but no
item name:
        if not itemToBuy:
            return
getAllFirstDescWords(worldRooms[location][SHOP])

        # otherwise, get a list of all
"description words" for shop items matching the
command text so far:
        for item in list(set(worldRooms[location]
[SHOP])):
            for descWord in worldItems[item]
[DESCWORDS]:
                if descWord.startswith(text):
                    possibleItems.append(descWord)

        return list(set(possibleItems)) # make
list unique


    def do_sell(self, arg):
        """sell <item>" - sell an item at the
current location's shop."""
        if SHOP not in worldRooms[location]:
            print('This is not a shop.')
            return

        itemToSell = arg.lower()

        if itemToSell == '':
            print('Sell what? Type "inventory" or
"inv" to see your inventory.')
            return
```

```
        for item in inventory:
            if itemToSell in worldItems[item]
[DESCWORDS]:
                # NOTE - If you wanted to
implement money, here is where you would add
                # code that gives the player money
for selling the item.
                print('You have sold %s' %
(worldItems[item][SHORTDESC]))
                inventory.remove(item)
                return

        print('You do not have "%s". Type
"inventory" or "inv" to see your inventory.' %
(itemToSell))


    def complete_sell(self, text, line, begidx,
endidx):
        if SHOP not in worldRooms[location]:
            return []

        itemToSell = text.lower()
        possibleItems = []

        # if the user has only typed "sell" but no
item name:
        if not itemToSell:
            return getAllFirstDescWords(inventory)

        # otherwise, get a list of all
"description words" for inventory items matching
the command text so far:
        for item in list(set(inventory)):
            for descWord in worldItems[item]
[DESCWORDS]:
                if descWord.startswith(text):
```

```
                    possibleItems.append(descWord)

        return list(set(possibleItems)) # make
list unique
```

When you are finished typing in this code, your program will look like this file on GitHub.

Eating

Some items are marked as edible by having an `EDIBLE` key set to `True` For example, Meat Pies, Bagels, and Donuts are all edible items:

```
    'Meat Pie': {
        GROUNDDESC: 'A suspicious meat pie rests
on the ground.',
        SHORTDESC: 'a meat pie',
        LONGDESC: 'A meat pie. It tastes like
chicken.',
        EDIBLE: True,
        DESCWORDS: ['pie', 'meat']},
    'Bagel': {
        GROUNDDESC: 'A bagel rests on the ground.
(Gross.)',
        SHORTDESC: 'a bagel',
        LONGDESC: 'It is a donut-shaped bagel.',
        EDIBLE: True,
        DESCWORDS: ['bagel']},
    'Donut': {
        GROUNDDESC: 'A donut rests on the ground.
(Gross.)',
        SHORTDESC: 'a donut',
        LONGDESC: 'It is a bagel-shaped donut.',
        EDIBLE: True,
        DESCWORDS: ['donut']},
```

For simplicity, eating things doesn't do anything other than remove them from your inventory. But you could implement health or hunger levels in your text adventure, and need eat (or drink) items or else have adverse effects. There's nothing new about how the

do_eat() and complete_eat() functions work. Add the following code to your program:

```
    def do_eat(self, arg):
        """eat <item>" - eat an item in your
inventory."""
        itemToEat = arg.lower()

        if itemToEat == '':
            print('Eat what? Type "inventory" or
"inv" to see your inventory.')
            return

        cantEat = False

        for item in
getAllItemsMatchingDesc(itemToEat, inventory):
            if worldItems[item].get(EDIBLE, False)
== False:
                cantEat = True
                continue # there may be other
items named this that you can eat, so we continue
checking
            # NOTE - If you wanted to implement
hunger levels, here is where
            # you would add code that changes the
player's hunger level.
            print('You eat %s' % (worldItems[item]
[SHORTDESC]))
            inventory.remove(item)
            return

        if cantEat:
            print('You cannot eat that.')
        else:
            print('You do not have "%s". Type
"inventory" or "inv" to see your inventory.' %
(itemToEat))
```

```
    def complete_eat(self, text, line, begidx,
endidx):
        itemToEat = text.lower()
        possibleItems = []

        # if the user has only typed "eat" but no
item name:
        if itemToEat == '':
            return getAllFirstDescWords(inventory)

        # otherwise, get a list of all
"description words" for edible inventory items
matching the command text so far:
        for item in list(set(inventory)):
            for descWord in worldItems[item]
[DESCWORDS]:
                if descWord.startswith(text) and
worldItems[item].get(EDIBLE, False):
                    possibleItems.append(descWord)

        return list(set(possibleItems)) # make
list unique
```

With the above code added, you will have finished the entire text adventure program. This program can be downloaded from GitHub: https://raw.githubusercontent.com/asweigart/textadventuredemo/master/textadventuredemo.py

Ideas for New Features

That's it for this text adventure tutorial. Because of the `cmd` module's command line interface features, it is fairly easy to add new commands to your game. From here, there are several things you could add to your game:

- Add HP, hunger/thirst levels, and status effects. (These are common to RPG-like games.)
- Combat with randomly wandering monsters.
- Casting magic spells, or learning new spells from spellbook items.

- Drinking items, including potions which can have magical effects.
- Equipping items, such as wearing helmets or wielding swords.
- Money, including different types of currencies that shops can accept or deny.
- Several new rooms to expand the world.

  If you are interested in creating a roguelike (a genre that is sort of like the Diablo games except with ASCII-art graphics), the libtcod module will be very helpful. There are tutorials here and here.

  I detail the differences between these genre of text-based games in my blog post, Text Adventure vs. MUD vs. Roguelike vs. Dwarf Fortress.

  You can try playing MUDs you find through The Mud Connector to get new ideas for additional features you'd like to add. Good luck, and have fun!