

Project 5: The Adventure Game

Welcome to the final project of CS 151! Your mission in this project is to write a simple text-based adventure game in the tradition of Will Crowther's pioneering "Adventure" program of the early 1970s, which was described in class. In games of this sort, the player wanders around from one location to another, picking up objects, and solving simple puzzles. The program you will create for this project is less elaborate than Crowther's original game and is therefore a bit more limited in terms of the types of puzzles one can construct for it. Even so, you can still write a program that captures much of the spirit and flavor of the original game.

Because this project is large and detailed, it takes quite a bit of writing to describe it all. This guide contains everything you need to complete the project, along with a considerable number of hints and strategic suggestions. To make it easier to read, the document is broken into sections detailed below. Try not to be daunted by the size of this handout. The code is not as large as you might think. If you start early and follow the suggestions in the "Milestones" section, things should work out well.

You can find the starting repository [here](#). To allow for the possibility of partners, everyone will need to make a team, even if you are just going to be working on things solo. Once one partner has made the team, the other can join it. If you accidentally join a team you did not mean to, let me know and I can see about getting you removed so you can join another. Only join an existing team if you intend to work with that person! There is a working demo of the project online [here](#), which you can compare your code output against.

Contents

1	Overview of the Adventure Game	2
2	Overview of the data files	4
2.1	The Rooms file	5
2.2	The Objects file	6
2.3	The Synonyms file	7
3	Milestones	8
3.1	Milestone 1: Modifying the teaching machine code so that it fits with Adventure	8
3.2	Milestone 2: Implement short descriptions of the rooms	9
3.3	Milestone 3: Implement the QUIT, HELP and LOOK commands	10
3.4	Milestone 4: Read in and distribute the objects	11
3.5	Milestone 5: Implementing TAKE, DROP, and INVENTORY	12
3.6	Milestone 6: Implement synonyms	13
3.7	Milestone 7: Implement locked passages	14
3.8	Milestone 8: Implement forced motion	15
4	Group Work	16
5	Possible Extensions	17

1 Overview of the Adventure Game

The Adventure game you will implement for this project—like any of the text-based adventure games that were the dominant genre before the advent of more sophisticated graphical adventures like the Myst series—takes place in a virtual world in which you, as the player, move about from one location to another. The locations, which are traditionally called *rooms* even though they may be outside, are described to you through a written textual description that gives you a sense of the geography. You move about in the game by giving commands, most of which are simply an indication of the direction of the motion. For example, in the classic Adventure game developed by Willie Crowther, you might move about as follows:

```
Adventure
You are standing at the end of a road before a small brick
building. A small stream flows out of the building and
down a gully to the south. A road runs up a small hill
to the west.
> WEST
You are at the end of a road at the top of a small hill.
You can see a small building in the valley to the east.
> EAST
```

In the console sessions shown in this guide, user input appears in uppercase so that it is easier to see. *Your program should ignore case distinctions in executing commands.*

In this example, you started outside the building, followed the road up the hill by typing WEST, and arrived at a new room on the top of the hill. Having no obvious places to go once you got there, you typed EAST to head back to where you started. As is typical in such games, the complete description of a location appears only the first time you enter it. The second time you come to the building, the program displays a much shorter identifying tag, although you can get the complete description by typing LOOK, as follows:

```
Adventure
> EAST
Outside building
> LOOK
You are standing at the end of a road before a small brick
building. A small stream flows out of the building and
down a gully to the south. A road runs up a small hill
to the west.
>
```

From here, you might decide to go inside the building by typing IN, which brings you to another room, as follows:

```
Adventure
> IN
You are inside a building, a well house for a large spring.
The exit door is to the south. There is another room to
the north, but the door is barred by a shimmering curtain.
There is a set of keys here.
>
```

In addition to the new room description, the inside of the building reveals that the Adventure game also contains objects: there is a set of keys here. You can pick up the keys

by using the **TAKE** command, which requires that you specify what object you are taking, like so:

```
Adventure
> TAKE KEYS
Taken.
>
```

The keys will, as it turns out, enable you to get through a grating at the bottom of the streambed that opens the door to Colossal Cave and the magic it contains.

The best model for the Adventure project is the teaching machine example that appears in Chapter 12. The starting repository for Project 5 includes the code for the teaching machine so that you can copy and adapt whatever parts of the code you think will be useful. The repository also includes the `tokenscanner` library, the various data files described later in this handout, and the following template files for the adventure game:

- **Adventure.py**—This file defines the **Adventure** function, which is just a few lines long and looks almost exactly the same as the **TeachingMachine.py** file in the example. The complete code for **Adventure.py** is given to you in the repository, and you should not need to change anything in this file except for the definition of the **ADVENTURE_PREFIX** constant when you want to work with other data files—or write your own.
- **AdvGame.py**—This file defines the **AdvGame** class, which implements the game and is therefore analogous to the **TMCourse** class in the teaching machine. The **AdvGame** class exports a static method called **read_data_files**, which is responsible for reading the various files and storing the information in a suitable internal structure. This class also exports the **run** method, which is called by the **Adventure** function to start the game. Although the **run** method is complex—and certainly complex enough to warrant decomposition—you will have a chance to build it up gradually as you go through the milestones.
- **AdvRoom.py**—The file defines the **AdvRoom** class, which represents a single room in the game and is analogous to the **TMQuestion** class in the teaching machine. The repository contains the header lines for the methods you need for Milestone 1. As you move on to later milestones, you will need to add a few more methods as described later in this guide.
- **AdvObject.py**—This file defines the **AdvObject** class, which represents an object in the game. This file specifies the header lines for the methods that **AdvObject** supports. You will have a chance to implement these methods in Milestone 4.

2 Overview of the data files

Like the teaching machine program, the Adventure program you create for this project is entirely *data driven*. The program itself does not know the details of the game geography, the objects that are distributed across the various rooms, or even the words used to move from place to place. All such information is supplied in the form of data files, which the program uses to control its own operation. If you run the program with different data files, the same program will guide its players through different sets of rooms that presumably have different interconnections, objects, and puzzles.

The starting repository includes data files for three different adventures of varying sizes. The smallest of these is the **Tiny** adventure, which describes an adventure with four rooms and no objects. The largest is the **Crowther** adventure, which includes a relatively sizable subset of Will Crowther's original Adventure game. In between those extremes is the **Small** adventure, which includes examples of the main features in the game in a space containing just 12 rooms. You should use the **Tiny** adventure until you get to Milestone 4. After that, you should use the **Small** adventure whenever you are debugging. Once you have got things working, you can move on to the **Crowther** adventure.

To indicate which data files you would like to use, the **Adventure.py** program defines a constant called **ADVENTURE_PREFIX**, which identifies the string that appears at the beginning of the filenames used for that adventure. To get the adventure game illustrated in the earlier examples, **ADVENTURE_PREFIX** is set to "**Crowther**", which selects the collection of files associated with a large chunk of Will Crowther's original Adventure game. For each adventure, there are three associated data files that contain the name of the adventure as a prefix. For the **Crowther** adventure, for example, these files are:

- **CrowtherRooms.txt**, which defines the rooms and the connections between them. In these examples, you have visited three rooms: outside of the building, the top of the hill, and the inside of the well house.
- **CrowtherObjects.txt**, which specifies the descriptions and initial locations of the objects in the game, such as the set of keys.
- **CrowtherSynonyms.txt**, which defines several words as synonyms of other words so that you can use the game more easily. For example, the compass points **N**, **S**, **E**, and **W** are defined to be equivalent to **NORTH**, **SOUTH**, **EAST** and **WEST**. Similarly, if it makes sense to refer to an object by more than one word, this file can define the two as synonyms. As you explore the Crowther cave, for example, you will encounter a gold nugget, and it makes sense to allow players to refer to it using either **GOLD** or **NUGGET**.

These data files are not Python programs, but are instead text files that describe the structure of a particular adventure game in a form that is easy for game designers to write. The Adventure program reads these files into an internal data structure, which it then uses to guide the player through the game.

2.1 The Rooms file

The `Rooms` file contains the names and descriptions of the rooms along with the passages that connect them. The contents of `TinyRooms.txt`, for example, are shown below in Figure 1.

```
OutsideBuilding
Outside building
You are standing at the end of a road before a small brick
building. A small stream flows out of the building and
down a gully to the south. A road runs up a small hill
to the west.
-----
WEST: EndOfRoad
UP: EndOfRoad
NORTH: InsideBuilding
IN: InsideBuilding
SOUTH: Valley
DOWN: Valley

EndOfRoad
End of road
You are at the end of a road at the top of a small hill.
You can see a small building in the valley to the east.
-----
EAST: OutsideBuilding
DOWN: OutsideBuilding

InsideBuilding
Inside building
You are inside a building, a well house for a large spring.
The exit door is to the south.
-----
SOUTH: OutsideBuilding
OUT: OutsideBuilding

Valley
Valley beside a stream
You are in a valley in the forest beside a stream tumbling
along a rocky bed.
-----
NORTH: OutsideBuilding
UP: OutsideBuilding
```

Figure 1: The `TinyRooms.py` text file, which contains the details about what rooms exist in the adventure and how they are connected to one another.

The first thing to notice about the `TinyRooms.txt` data file is that it matches almost exactly the format used for the teaching machine application. The only real difference is that each room includes a one-line short description as well as the multi-line text. When you implement the code to read the data file for the rooms, you will have to make a few small changes to accommodate this difference. The more substantial part of revising the implementation, however, lies in making sure that the names of all the variables and methods match the metaphor of the Adventure game and not the teaching machine. It would certainly confuse anyone looking at your code to see names like `questions` and `answers`! In the context of the Adventure game, the code needs to refer to `rooms` and `passages` instead.

2.2 The Objects file

Although you will not need to think about this file until you get to Milestone 4, both the `Small` and `Crowther` adventures define a set of objects that are distributed somewhere in the cave, such as the keys you saw in the earlier example. Like the rooms, the objects in the game are specified using a data file, such as the one for the `Small` adventure shown in Figure 2.

```
KEYS
a set of keys
InsideBuilding

LAMP
a brightly shining brass lamp
BeneathGrate

ROD
a black rod with a rusty star
DebrisRoom

WATER
a bottle of water
PLAYER
```

Figure 2: The `SmallObjects.txt` data file, which contains information about the available objects that can appear in the rooms and where they would initially appear.

The contents of the `Objects` file consist of a series of three-line entries, one for each object. The first line is the name of the object, which is also the word that the player uses to refer to the object. The second is a short description of the object, usually beginning with an article like *a* or *an*. The third is the name of the room in which the object is placed at the beginning of the game. For example, the lines

```
KEYS
a set of keys
InsideBuilding
```

defines an object whose name is `"KEYS"` and whose description is `"a set of keys"`. At the beginning of the game, that object is placed in the room whose identifying name is `"InsideBuilding"`, which is precisely where you saw it in the sample runs shown in the earlier section.


The last entry in the `SmallObjects.txt` data file illustrates a feature that requires special handling. The lines

```
WATER
a bottle of water
PLAYER
```

specify that the bottle of water starts off in the player's possession. You will have to check for this case in the code that distributes the objects, starting in Milestone 4.

2.3 The Synonyms file

The last data file is the **Synonyms** file, which consists of a sequence of definitions that allow the player to use more than one word to refer to a direction or an object. The **SmallSynonyms.txt** file in Figure 3, for example, defines **BOTTLE** as a synonym for **WATER**, since both nouns appear in the description. It also defines abbreviated forms of the standard directions so that the player can type **N** instead of **NORTH**, along with a few equivalent words for verbs like **TAKE** and **DROP**.



```
N=NORTH
S=SOUTH
E=EAST
W=WEST
U=UP
D=DOWN
Q=QUIT
L=LOOK
I=INVENTORY
CATCH=TAKE
RELEASE=DROP
BOTTLE=WATER
```

Figure 3: The **SmallSynonyms.txt** data file, which includes alternative commands or object labels that can be used within the game.

3 Milestones

For a project of any reasonable complexity, it is important to implement the project in stages rather than trying to get it all going at once. As with all our other projects this semester, this project is organized into a series of milestones designed to lead you through the process in a series of manageable steps.

3.1 Milestone 1: Modifying the teaching machine code so that it fits with Adventure

As you saw at the end of our lecture, the `TeachingMachine.py` program works as a rudimentary Adventure-style game if you simply change the data file. The results of doing so, however, does not constitute a useful basis for building up a more sophisticated Adventure game. If nothing else, the metaphors used in the code are entirely inappropriate to the new context. The teaching machine program talks about courses, questions and answers, none of which make sense in the Adventure world. The corresponding concepts in Adventure are games, rooms, and passages. Your first step is to take the code for the teaching machine and adapt it so that it makes sense for the Adventure-game model.

You have two starting points for this phase of the project. The `TeachingMachine.py` folder contains the code for the teaching machine application presented in class. The main repository folder contains the starter versions of the files you need in order to implement the **Adventure** classes used in the Adventure game. Your task for this milestone is to adapt the code from the `TMCourse.py` and `TMQuestion.py` files into their `AdvGame.py` and `AdvRoom.py` counterparts (you do not have to do anything with `AdvObject.py` until Milestone 4).

The code you need to complete this milestone is entirely there already, at least in a functional sense. All you have to do is copy the code from the teaching machine application into the corresponding classes in the Adventure game, changing the names of the fields and methods so that they fit the Adventure game metaphor and making the small changes discussed in Section 2.1. The new names of the exported methods are given to you as part of the starter files, but you will also need to change the names of a few variables so that they make sense in the context of the game.

This milestone has two primary purposes:

1. To ensure that you understand what is going on in the teaching machine application.
2. To give you some practice in debugging. Even though the structure of the code remains exactly the same, this milestone is not as easy as you might think. Nearly all the variable and method names will have to change, and you will need to be careful to make sure that your changes are consistent. Since you will probably make some mistakes along the way, you will need to polish up your debugging skills to figure out what exactly you did wrong or missed.

There is one important difference that you should be aware of. In the teaching machine example a static method was used to read in and construct the original `TMCourse` object. If

you look at `Adventure.py` however, you will note a similar model is not being employed here. Instead, the constructor for `AdvGame` is being called directly with the prefix as an argument. The reason for this difference is because you will need to eventually load several different files using the same prefix, and so it makes more sense to handle this all within the `AdvGame` constructor. Note that you will still need to read in all the game and room information, just like was done in the teaching machine static method, but in this case you will want to set the needed attributes directly.

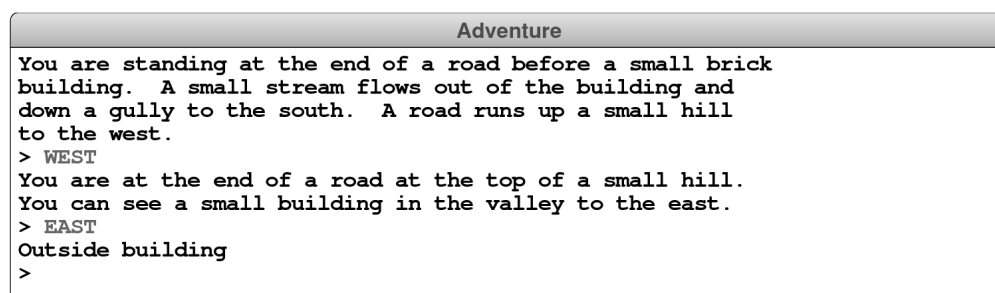
When you finish this milestone, you should be able to wander a bit around the surface of the Adventure world, heading up to the top of the hill, inside the building, and down to the grate. You will not, unfortunately, be able to get past this grate until Milestone 7.

3.2 Milestone 2: Implement short descriptions of the rooms

The Adventure game would be tedious to play—particularly when output devices were as slow as they were in the 1970s—if the program always gave the full description of the room every time you entered it. Crowther’s game introduced the idea of short descriptions, which were one-line descriptions for rooms that the player has already visited. The long description appears the first time a room is entered or when the player types `LOOK`, and the short description appears thereafter.

Your job in this milestone is to implement this feature in your program. You presumably already implemented the `get_short_description` method in the `AdvRoom` class, but you need to add two new methods to `AdvRoom` to keep track of whether the room has been visited and change the code in `AdvGame` so that it checks for that condition and prints out the short description for rooms the player has already seen. The new methods in the `AdvRoom` class are `set_visited` and `has_been_visited`. The first takes a Boolean value and stores that with the room as a flag indicating that the room has been visited. The second returns the value of that flag.

Once you have completed this milestone, your program should be able to generate the following sample run:



```
Adventure
You are standing at the end of a road before a small brick
building. A small stream flows out of the building and
down a gully to the south. A road runs up a small hill
to the west.
> WEST
You are at the end of a road at the top of a small hill.
You can see a small building in the valley to the east.
> EAST
Outside building
>
```

Note that the player sees the short description after returning to the initial room.

Table 1: The build-in action verbs in the Adventure game.

QUIT	This command signal the end of the game. Your program should call the build-in function <code>quit</code> to exit from the program.
HELP	This command should print instructions for the game on the console. You need not duplicate the instructions from the starter file exactly, but you should certainly give the users an idea of how your game is played. If you make any extensions, you should describe them in the output of your <code>HELP</code> command so that I can easily see what exciting things I should look for!
INVENTORY	This command should list what objects the user is holding. If the user is holding no objects, your program should say so with a message along the lines of “You are empty-handed.”
LOOK	This command should type the complete description of the room and its contents, even if the user has already visited the room.
TAKE <i>obj</i>	This command requires a direct object and has the effect of taking the objects out of the room and adding it to the set of objects the user is carrying. You need to check to make sure that the object is actually in the room before you let the user take it.
DROP <i>obj</i>	This command requires a direct object and has the effect of removing the object from the set of objects the user is carrying and adding it back to the list of objects in the room. You need to check to make sure that the user is carrying the object.

3.3 Milestone 3: Implement the QUIT, HELP and LOOK commands

Most of the commands entered by the player are words like `WEST` or `EAST` that indicate a passage to another room. Collectively, these words are called *motion verbs*. Motion verbs, however, are not the only possible commands. The Adventure game allows the player to enter various build-in commands called *action verbs*. The six action verbs you are required to implement (although you only need to implement `QUIT`, `HELP`, and `LOOK` as part of Milestone 3) are described in Table 1.

The first thing you need to do to implement this milestone is to subdivide the user’s input into individual words—a process that the `tokenscanner` module makes easy. Once you have done that, you need to look at the first word to see if it is one of the action verbs before checking whether a motion verb applies. You then need to implement the first three action verbs. The `QUIT` command stops the program from reading any more user commands, just as a new room with the name `"EXIT"` does in the code you adapted from the teaching machine. The `HELP` command prints the contents of the `HELP_TEXT` constant out to the console. The `LOOK` command prints the long description of the current room.

Once you have finished this milestone, your program should be able to produce the following sample run:

```
Adventure
You are standing at the end of a road before a small brick
building. A small stream flows out of the building and
down a gully to the south. A road runs up a small hill
to the west.
> WEST
You are at the end of a road at the top of a small hill.
You can see a small building in the valley to the east.
> EAST
Outside building
> LOOK
You are standing at the end of a road before a small brick
building. A small stream flows out of the building and
down a gully to the south. A road runs up a small hill
to the west.
> HELP
Welcome to Adventure!
Somewhere nearby is Colossal Cave, where others have found fortunes in
treasure and gold, though it is rumored that some who enter are never
seen again. Magic is said to work in the cave. I will be your eyes
and hands. Direct me with natural English commands; I don't understand
all of the English language, but I do a pretty good job.

It's important to remember that cave passages turn a lot, and that
leaving a room to the north does not guarantee entering the next from
the south, although it often works out that way. You'd best make
yourself a map as you go along.

Much of my vocabulary describes places and is used to move you there.
To move, try words like IN, OUT, EAST, WEST, NORTH, SOUTH, UP, or DOWN.
I also know about a number of objects hidden within the cave which you
can TAKE or DROP. To see what objects you're carrying, say INVENTORY.
To reprint the detailed description of where you are, say LOOK. If you
want to end your adventure, say QUIT.
> QUIT
```

3.4 Milestone 4: Read in and distribute the objects

The most important extension that separates the Adventure game from the teaching machine application is the introduction of objects like keys and treasures. The objects are specified in the `Objects` file described in Section 2.2. For example, the first object in the `SmallObjects.txt` data file is the set of keys, for which the description consists of the following three lines:

```
KEYS
a set of keys
InsideBuilding
```

Your first task in this milestone is to implement the `AdvObject` class, which is given to you in skeletal form in the repository. The `AdvObject` class defines a constructor and the getter methods `get_name`, `get_description`, and `get_initial_location`, along with a static `read_object` method that reads an object from a data file. Your model for this file is the `AdvRoom` class, which implements the same mechanism for the more complicated data structure used for rooms. You also need to add the necessary code to the `AdvGame` class to

read in all the objects and store them in a dictionary, just as you do for the rooms. It is not an error if the `Objects` file is missing; a missing file just means that there are no objects.

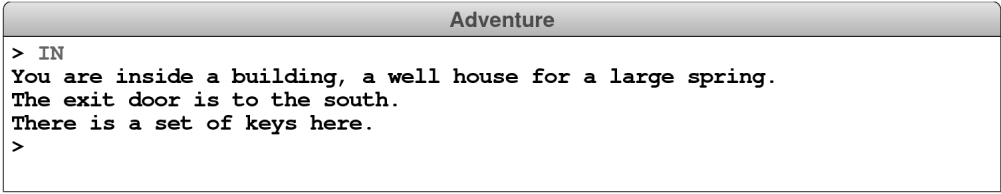
Your second task is to add methods to the `AdvRoom` class so that rooms can keep track of the objects they contain. The easiest strategy for doing so is to keep track of the names of the objects in the room using a Python set stored in the `AdvRoom` object. In keeping with the principles of data abstraction, clients should not look at this set directly but should instead obtain that information through method calls. The following methods are sufficient to get you through all of the milestones:

- An `add_object` method that takes an object name and adds it to the room contents.
- A `remove_object` method that takes an object name and removes it from the contents.
- A `contains_object` method that checks whether an object name is in the room.
- A `get_contents` method that returns a *copy* of the set of object names.

The third task is to add code to `AdvGame` so that it places the objects in the appropriate rooms at the start of the game. You could do this either at the end of the constructor once the rooms and objects have been read in, or at the start of the `run` method. This code should iterate through the dictionary containing all the objects and then call `add_object` to put it in the room specified by its initial location. Your code should simply skip over any objects whose location is `PLAYER` until you implement Milestone 5.

The fourth and final task is to extend the code that describes a room so that it also prints out the descriptions of the objects contained in that room. The objects are listed on single lines, one for each object that the room contains.

Because Milestone 4 does not yet allow you to pick up and drop objects, the only thing you can do to test whether this part is working is to check and see whether the objects are listed as part of the room descriptions. For example, you should make sure that the keys are listed inside the building, as shown in the following session:



```

Adventure
> IN
You are inside a building, a well house for a large spring.
The exit door is to the south.
There is a set of keys here.
>

```

3.5 Milestone 5: Implementing TAKE, DROP, and INVENTORY

The next step is to add the `TAKE`, `DROP`, and `INVENTORY` commands to the command processor you implemented for Milestone 3. The `TAKE` command checks to see if an object is in the room, and if so, removes it from the room and adds it to the player's inventory. The `DROP` command reverses the process, removing an object from the player's inventory and then adding that object to the room. The `INVENTORY` command goes through the player's inventory and prints the description of each object. If the player's inventory is empty, the

INVENTORY command should display the string "You are empty-handed.". These behaviors are illustrated in the following sample run from the beginning of the game:

```
Adventure
You are standing at the end of a road before a small brick
building. A small stream flows out of the building and
down a gully to the south. A road runs up a small hill
to the west.
> INVENTORY
You are carrying:
  a bottle of water
> DROP WATER
Dropped.
> INVENTORY
You are empty-handed.
> IN
You are inside a building, a well house for a large spring.
The exit door is to the south.
There is a set of keys here.
> TAKE KEYS
Taken.
> OUT
Outside building
There is a bottle of water here.
> DROP KEYS
Dropped.
> LOOK
You are standing at the end of a road before a small brick
building. A small stream flows out of the building and
down a gully to the south. A road runs up a small hill
to the west.
There is a bottle of water here.
There is a set of keys here.
>
```

Implementing this milestone requires several steps:

- Add a new attribute to the `AdvGame` class that keeps track of the set of objects the player is carrying.
- Implement the special case for "PLAYER" in the code you wrote for Milestone 4.
- Write code to implement the TAKE, DROP, and INVENTORY commands. Note that TAKE and DROP require you to read an object name using the token scanner.

3.6 Milestone 6: Implement synonyms

At this point in your implementation, your debugging sessions will have you wandering through the Adventure game more than you did in the beginning. As a result, you will almost certainly find it convenient to implement the synonym mechanism, so that you can type N, S, E, and W instead of the full names for the compass directions. The format of the `Synonyms` file is described in Section 2.3.

To implement the synonym processing, you need to read through the `Synonyms` file and create a dictionary in the `AdvGame` class that contains the synonym definitions. Whenever you read a word—which might be a motion verb, an action verb, or the name of an object—you need to see if that word exists in the synonym dictionary and, if so, substitute the standard definition.

As the with the `Objects` file, it is not an error if the `Synonyms` file is missing. In that case, the synonym dictionary should just be empty.

3.7 Milestone 7: Implement locked passages

When you modified the teaching machine code for Milestone 1, you presumably defined the structure representing passages to be a dictionary that maps direction names to room names. That is, after all, how the teaching machine worked, and this guide did not give you any reason to change that model.

Unfortunately, using a dictionary does not quite work for the Adventure game. If you look closely at the list of passages in the `SmallRooms.txt` and `Crowther.txt`, you will discover that certain rooms include the same direction name more than once in the list. For example, the entry for the room above the grate that leads to the underground part of the cave looks like this:

```
OutsideGrate
Outside grate
You are in a 20-foot depression floored with bare dirt.
Set into the dirt is a strong steel grate mounted in
concrete. A dry streambed leads into the depression from
the north.
-----
NORTH: SlitInRock
UP: SlitInRock
DOWN: BeneathGrate/KEYS
DOWN: MissingKeys
```

As you can see, the motion verb `"DOWN"` appears twice in the list of passages. The first one ends with a slash and the word `KEYS`, which indicates that the object whose name is `"KEYS"` is required to traverse this passage. The second has no such modifier, which means that the passage is always available. This definition is an example of a *locked passage*, which is one that requires the player to be holding a specified object that is call its *key*. In this case, the key is literally the set of keys that starts off inside the building. If the keys are in the player's inventory, applying the motion verb `DOWN` uses the first passage; if not, applying `DOWN` skips over that passage and follows the one to the room named `"MissingKeys"`, which is described in Milestone 8.

This new feature requires you to change the implementation of the data structure used to represent passages, since a dictionary does not allow multiple values with the same key. What you need to do is change the data structure used to represent the passages from a dictionary to an array in which the individual elements are tuples containing three values: the direction name, the name of the destination room, and the key required to traverse the passage, which may be `None`. The code that moves from one room to another based on the player's input must search through the array to find the first option that applies.

The changes you need to make for Milestone 7 are in the `AdvRoom` and `AdvGame` classes. You will, for example, have to change your implementation of `read_room` so that it stores

the data for the passages in an array rather than a dictionary. You also need to change the way that the `get_next_room` method works, since this method now has to take account of what the player is carrying. That requirement, however, creates a bit of a problem. The `AdvRoom` class does not *know* what the player is carrying, since that information is available only inside the `AdvGame` class.

There are several strategies you might use to solve this problem. One possibility is to pass the player's inventory—along with the map from object names to `AdvObject` structures—as additional arguments to the `get_next_room` method. One the whole, however, it is probably simpler to reassign the task of figuring out which room you reach after moving in a particular direction into the `AdvGame` class, which already has the necessary information. Doing so requires making the structure containing the array of passages available to clients of `AdvRoom`, which is easily done by adding a `get_passages` method to the `AdvRoom` class.

Once you have implemented this change, you should be able to explore the entire Adventure game, picking up objects and using them as keys to get through previously closed passages. You still, however, will not be able to escape if you try to go through a locked passage without the necessary key. For that, you need to implement the final milestone.

3.8 Milestone 8: Implement forced motion

When the player tries to go through a locked passage without the necessary key, the game has to indicate that the motion is prohibited. One possible strategy would be to design a whole new data structure to represent messages of this type. A simpler way, however, is to make a small extension to the structure that is already in place.

When Willie Crowther faced this problem in his original Adventure game, he chose the simple approach. He simply created new rooms whose descriptions contained the necessary messages he wanted to deliver. When the player entered one of those rooms, the code that you have been running all along would print out the necessary message, just like any other room description. The only problem is that you don't actually want the player to end up in that room, but rather to be moved automatically to some other room. To implement this idea, Crowther came up with the idea of using a special motion verb called **"FORCED"** to specify *forced motion*.

When the player enters a room in which one of the connections is associated with the motion verb **FORCED** (and the player is carrying any object that the **FORCED** verb requires to unlock the passage), your program should display the description of that room and then immediately move the player to the specified destination without waiting for the player to enter a command. This feature makes it possible to display a message to the player and then continue on from there.

The facility is illustrated by the room named **"MissingKeys"**, which has the following definition:

```
MissingKeys
-
The grate is locked and you don't have any keys.
-----
```

```
FORCED: OutsideGrate
```

The effect of this definition is to ensure that whenever the player enters this room, the room will automatically be set to "OutsideGrate".

It is possible for a single room to use both the locked passage and forced motion options. The `CrowtherRooms.txt` file, for example, contains the following entry for the room just north of the curtain in the building:

```
Curtain1
-
-----
FORCED: Curtain2/NUGGET
FORCED: MissingTreasures
```

The effect of this set of motion rules is to force the player to the room named `Curtains2` if the player is carrying the nugget and to the room named `MissingTreasures` otherwise. When you are testing your code for locked and forced passages, you might want to pay particular attention to the last eight rooms in the `CrowtherRooms.txt` file. These rooms, all of which have no lines at all in their long description, implement the shimmering curtain that marks the end of the game.

You should also notice that the rooms with forced motions do not supply a meaningful short description, although the data files use a single hyphen so that the code to read the file still works. Forced motion should always display the long description, which is easy to achieve by making sure that your program sets the visited flag only after the forced motion is complete. Also, be aware that sometimes several forced rooms can be chained together, so it is not enough to just check for a single forced room at a time.

4 Group Work

Because large projects like this one typically involve more than one programmer, you are encouraged to work on this project in teams of two, although you are free to work individually as well. Each person in a two-person team will receive the same grade.

In remote times, working together on this sort of project can seem technically challenging, but there are good sources at your disposal that can help. If you work in a pair, you will both have access to a shared repository on Github. This means that, as long as you are good about uploading your files, each person can usually work on the latest version of the code without too much issue. There are some built in ways within VSCode that you can sync your local files with Github files as well, so let me know if you'd like some guidance in that. VSCode also has a remote sharing extension, which can essentially give you collaborative control over your code, similar to Google docs. The extension is called "Live Share" if you want to search for it within the VSCode extensions.

5 Possible Extensions

The following extensions would make the Adventure program much more powerful and would allow the construction of more interesting puzzles and scenarios:

- *Active objects.* The biggest weakness in the current game is that the objects are entirely passive. All you can do with an object is to pick it up or drop it. Moreover, the only way in which the objects enter into the play of the game is in the specification of locked passages in the room data file: if you are carrying an object, some passage is open that would otherwise be locked. It would be wonderful if it were possible to type `WAVE WAND` or `UNLOCK GRATE` and have the appropriate thing happen. Moreover, being able to `READ` or `EXAMINE` an object adds a lot of interest to the game.
- *Object state.* In the original version of Adventure, objects can have different states. For example, the grate at the entrance to the cave can be either locked or unlocked; similarly, the snake in the Hall of the Mountain King can be blocking your path or driven away. You might add some way to allow the program to keep track of the state of each object and then make it possible for the motion rules to indicate that a particular passage can only be taken if an object is in a certain state: you can go through the grate only if it is unlocked.
- *Containment.* In Don Wood's extension to Adventure, some objects can contain other objects. Putting this concept into the game adds dimensionality to puzzle construction, but also requires implementing prepositional phrases in the parser so that the program can parse such constructions as:

```
> PUT NUGGET IN CHEST
```

- *Filler words.* The current parser limits the player to using commands that consist of one or two words. Saying:

```
> TAKE THE KEYS
```

causes an error because the program does not know the word `THE`; if the parser ignored articles and other filler words, the program would seem more conversational.

- *Adjectives.* A similar extension to the parser is the introduction of adjectives that allow the player to issue commands like:

```
> TAKE BLACK ROD
```

In the classic Adventure game, adjectives are associated uniquely with the noun to which they refer. In Zork, on the other hand, adjectives were used to differentiate many different objects of the same name, so that there could be both a black rod and a green rod in the same game.

- *Convenient shortcuts for "all" and "it".* When you are in a room with many objects, it is extremely useful to be able to type

```
> TAKE ALL
```

to take all the objects at that location. Similarly, the conversation flows more smoothly if you can refer to the last mentioned object as `IT`.

- *Random passages.* There are several rooms in the original Adventure game at which the motion through a passage is probabilistic. you could implement this sort of feature by specifying a percentage chance on a locked passage rather than an object. Thus, if the data for a room specified the passage entries:

```
SOUTH : RoomA/30  
SOUTH : RoomB
```

moving south would go to `RoomA` 30 percent of the time and to `RoomB` the rest of the time. (The program can differentiate this specification syntax from the traditional locked passages because the percentage chance starts with a digit.)