00000100

CHAPTER 4

Project Walkthrough

Learning Objectives

- *State the purpose of the project approach strategy*
- *Apply the project approach strategy to implement a Python programming assignment*
- *State the purpose of the software development cycle*
- *Describe the activities performed in each phase of the development cycle*
- *Apply the software development cycle in sprints to implement a programming assignment*
- *State the actions performed by an analyst, architect, and programmer*
- *Translate a project specification into a software design that can be implemented in Python*
- *State the purpose and use of function and method stubbing*
- *State the purpose and use of a UML state transition diagram*
- *Explain data abstraction and the role it plays in the design of user-defined data types*
- *Execute Python programs from the command line*
- *State the importance of early code execution and testing during the development process*

Introduction

This chapter presents a complete example of the analysis, design, and implementation of a typical classroom programming project. The objective of this chapter is to demonstrate how to approach a project and, with the help of the *project-approach strategy* and the *software development cycle*, formulate and execute a successful project implementation plan.

I will confine the solution to two modules and use a class to represent the primary user-defined data type. The resulting program source code will not be difficult to understand although you will not be formally introduced to many of the topics discussed here until later in the book. I use a class in the solution because although you can create stand-alone functions in Python, I employ an object-oriented first approach to learning the language in my university-level courses.

You may be unfamiliar with many of the concepts discussed here. Don't panic! I wrote this material with the intention that you revisit it when necessary. As you start designing and writing your own programs, examine these pages for clues on how to approach your particular problem. *Practice breeds confidence!* In time, you will begin to make sense of all these confusing concepts. After a few small victories, you will never again have to refer to this chapter.

What you will learn over the course of this book is that there are many possible approaches and solutions to the project presented in this chapter. For example, a user interface can be as simple as console I/O with the `print()` and `input()` functions, or it could be a *curses* console interface or a *Tkinter* graphical user interface (GUI). The application architecture can range from a single module to a multilayer, multithreaded, client-server architecture.

Let's begin by reviewing the project-approach strategy.

Classes? I'm worried Jane!
Oh Jim, don't panic! It'll be easy, I promise.

1 The Project-Approach Strategy Summarized

The project-approach strategy presented in chapter 2, page 73, is summarized in table 4-1. Keep the project-approach strategy in mind as you formulate your solution. Remember, the purpose of the project-approach strategy is to kick-start the creative process and sustain your creative momentum. Feel free to tailor the project-approach strategy to suit your needs.

Strategy Area	Explanation
Application Requirements	Determine and clarify exactly what purpose and features the finished project must have. Clarify your understanding of the requirements with your instructor if the project specification is not clear. **This results in a clear problem definition and a list of required project features.**
Problem Domain	Study the problem until you have a clear understanding of how to solve it. Optionally, express your understanding of the solution by writing a pseudocode algorithm that describes, step-by-step, how you will solve the problem. You may need to do this several times on large, complex projects. **This results in a high-level solution statement that can be translated into an application design.**
Language Features	Make a list of all the language features you must understand and use to draft a competent design and later implement your design. As you study each language feature, check it off your list. Doing so will give you a sense of progress and forward momentum. **This results in a notional understanding of the language features required to effect a good design and solve the problem.**
High-Level Design & Implementation Strategy	Sketch out a rough application design. A design is simply a statement, expressed through words, pictures, or both, of how you plan to implement the problem solution derived in the Problem Domain strategy area. **This results in a plan of attack!**

Table 4-1: Project Approach Strategy

2 Software Development Cycle

When the time comes to start writing code, you will employ the *software development cycle*. It’s good to have a broad, high-level design idea to get you started, but don’t make the mistake of trying to design everything up front. Design until you can begin coding and then test some of your ideas. The software development cycle is summarized in table 4-2.

Step	Explanation
Plan	Design to the point where you can get started on the implementation. Do not attempt to design everything up front. The idea here is to keep your design flexible and open to change.
Code	Implement what you have designed.

Table 4-2: Software Development Cycle

00000100

Step	Explanation
Test	Thoroughly test each section or module of source code. The idea here is to try to break it before it has a chance to break your application. Even for small projects you will find yourself writing short *test-case* programs on the side to test something you have just finished coding.
Integrate & Regression Test	Add the tested piece of the application to the rest of the project and then test the whole project to ensure it didn't break existing functionality.
Refactor	Take a comprehensive look at your overall application architecture and migrate general functionality up into base, or even abstract, classes so the functionality can be utilized by more concrete derived classes. Consolidate repeated code where possible.

Table 4-2: Software Development Cycle (Continued)

Employ the software development cycle in an *iterative* fashion as depicted in figure 4-1.

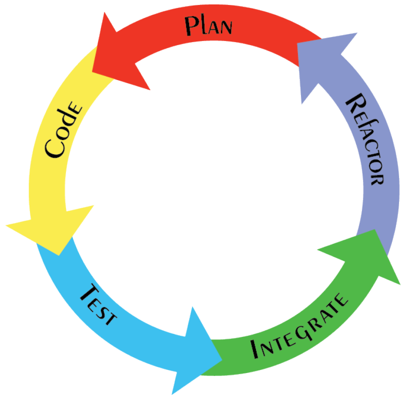


Figure 4-1: Iterative Software Development Cycle Deployment

Referring to figure 4-1 — By iterative I mean begin with the *Plan* step, followed by the *Code* step, followed by the *Test* step, followed by the *Integrate* step, optionally followed by the *Refactor* step. When you have finished a little piece of the project in this fashion, return to the Plan step and repeat the process. Each complete *Plan*, *Code*, *Test*, *Integrate*, and *Refactor* sequence is referred to as an *iteration*. As you iterate through the cycle, development progresses until you converge on the final solution.

2.1 Relationship To Agile Software Development

The software development cycle summarized above is the foundational process for Agile Software Development, whose original twelve principles can be viewed here: *Principles Behind The Agile Manifesto*. Small teams of software engineers and supporting technical specialists organized in *Scrum* teams apply the software development cycle in *sprints*. A sprint is a measured time period during which a scrum team will work collaboratively to complete a set amount of work. A project is decomposed into *Epics*, *Stories*, and *Tasks* which form a *backlog* of work required to complete the project, a designated milestone, or a release version. Epics, stories, and tasks are recorded, documented, estimated, and tracked within a supporting tool like Atlassian's *Jira*. You don't need to work in a team to apply the software development cycle in sprints. Organizing your development work into sprints is a great way to keep yourself organized, focused, and on track.

3 Project Specification

Keeping both the project-approach strategy and the software development cycle in mind, let’s look now at a typical project specification given in table 4-3.

IT-566 Computer Scripting Techniques
Project 1
Robot Rat

```
   Objectives:
   Demonstrate your ability to utilize the following language features in a
Python program:
   - Classes
   - Methods
   - Two-Dimensional Lists
   - Instance Variables
   - Local Method Variables
   - Program Control-Flow Statements
   - Console Input & Output

   Task:
   You are in command of a robot rat! Write a Python console application
that will allow you to control the rat’s movements around a 20 x 20 grid
floor.
   The robot rat is equipped with a pen. The pen has two possible positions:
UP or DOWN. When the pen is in the UP position the robot rat can move around
the floor without leaving a mark. If the pen is in the DOWN position, the
robot rat leaves a mark on visited floor grid positions. Moving the robot
rat about the floor with the pen UP or DOWN at various locations results in
a pattern written upon the floor.

   Hints:
   - The robot rat can move in four directions: NORTH, SOUTH, EAST, and
WEST. Implement diagonal movement if you desire.
   - Implement the floor as a two-dimensional list of boolean objects
   - Use the built-in functions input() and print() to read text from and
write text to the console.

   User Interface:
   At minimum, display a text-based command menu with the following or simi-
lar command choices:

                          1. Pen Up
                          2. Pen Down
                          3. Turn Right
                          4. Turn Left
                          5. Move Forward
                          6. Print Floor
                          7. Exit
```

Table 4-3: Project Specification

IT-566 Computer Scripting Techniques
Project 1
Robot Rat

    When menu choice 6 is selected to print the floor, the result might look something like this, assuming you chose a hyphen '-' to represent a marked area of the floor and a zero '0' to represent an unmarked area. You may use other pattern characters if desired.

```
-----000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
```

    In this example, the robot rat moved from the upper-left corner of the floor five spaced to the EAST with the pen DOWN.

Table 4-3: Project Specification (Continued)

3.1 Analyzing The Project Specification

Now is a good time to step through the project-approach strategy and analyze the Robot Rat project using each strategy area of concern as a guide starting with the application requirements.

3.1.1 Application Requirements

The Robot Rat project seems clear enough but omits a few details. It begins with a set of formally stated project objectives. It then states the task you must perform, namely, to write a program that lets you control a robot rat. But what, exactly, is a robot rat? That's a fair question whose answer requires a bit of abstract thinking. To clarify your understanding of the project's requirements, you decide to ask your instructor a few questions. Your first question is, "Does the robot rat exist?"

If your instructor answers the question by saying, "Well, obviously, the robot rat does not really exist!", he would be insulting you. Why? Because if you are wondering just what a robot rat is, then you are having difficulty abstracting the concept of a robot rat. He would be doing you a better service by saying, "The robot rat exists, but only as a collection of attributes that provide a limited description of the robot rat." He should also add that by writing a program to control the robot rat's movements around the floor, you are actually *modeling* the concept of a robot rat. And since a model of something usually leaves out some level of detail or contains some simplifying assumptions, he should also tell you that the robot rat does not have legs, fur, or a cute little nose.

Another valid requirements question might focus on exactly what is meant by the term *console application*. That too is a good question. A console application is a program that interacts with the terminal using the Python built-in `print()` and `input()` functions, and optionally can utilize command-line parameters. A console program does not usually have a graphical user interface (GUI), although nothing stops you from writing one that does.

What about error checking? Again, good question. In the real world, making sure an application behaves well under extreme user conditions and recovers gracefully in the event of an error consumes a good amount of programming effort. One area in particular that requires extra measures to ensure everything goes well is array or list processing. As the robot rat moves around the floor, you must take steps to prevent the program from letting it go beyond the bounds of the floor array. You need to use enough error checking to avoid major catastrophes.

Something else to consider is how to process menu commands. Since the project only calls for simple console input and output, one approach you could take is to treat all input as a text string. If you need to convert a text string into another data type, you can use the Python built-in functions like `int()` or `float()`, otherwise, you should concentrate on learning how to use the fundamental language features listed in the project's objectives section.

To summarize the requirements clarified thus far:

- Write a program that models the concept of a robot rat and its movement upon a floor.
- Think of the robot rat as an abstraction represented by a collection of attributes. (*I discuss these attributes in greater detail in the problem domain section that follows.*)
- Represent the floor as a two-dimensional array of boolean objects.
- Use just enough error checking to avoid catastrophe and focus on staying within the floor boundaries.
- Read user command input as a text string.
- Convert string input into other types as required.
- Put all program functionality into one user-defined class. This class will be a Python console application.

When you are sure you fully understand the project specification, you can proceed to the problem domain strategy area.

3.1.2 Problem Domain

In this strategy area, your objective is to learn as much as possible about what a robot rat is and how it works in order to gain insight into how to proceed with the project design. A good technique to help jump-start your creativity is to read through the project specification looking for relevant nouns and verbs or verb phrases. This is referred to as Noun-Verb Analysis.

A first pass at this activity yields two lists. The list of nouns suggests possible application objects, data types, and attributes. Nouns also suggest possible names for static (class-wide) and instance fields (*constants and/or variables*) and local method variables. The list of verbs suggests possible object interactions and method names.

Noun - Verb Analysis

A first pass at reviewing the project specification yields the list of nouns and verbs shown in table 4-4.

Nouns	Verbs
robot rat	move
floor	set pen up
pen	set pen down
pen position (up, down)	mark
mark	turn right
program	turn left
pattern	print floor
direction (north, south, east, west)	display menu
menu	exit

Table 4-4: Robot Rat Project Nouns and Verbs

This list of nouns and verbs is a good starting point. Now that you have it, what do you do with it? Good question. As I mentioned previously, each noun is a possible candidate for either a variable, a constant, or some other data type, data structure, object, or attribute within the application. A few of the nouns might not be used. Others have a direct relationship to a particular application feature. Some nouns look like they could be very useful but may not easily convert or map to any application feature. Also, the noun list may not be complete. You may discover additional application objects and object interactions as project analysis moves forward.

The verb list for this project example derives mostly from the suggested command menu. Verbs normally map directly to potential function or method names. You will need to create these methods as you write your program. Each method you identify will belong to a particular class, and may utilize some or all of the other objects, variables, constants, and data structures identified with the help of the noun list.

The noun list gleaned so far suggests that the Robot Rat project needs further analysis both to expand your understanding of the project's requirements and to reveal additional attribute candidates. How do you proceed? I recommend taking a closer look at several nouns that are currently on the list, starting with *robot rat*. Just what is a robot rat from the attribute perspective? Since pictures are always helpful, I suggest drawing a few in your engineer's notebook. (See "The Engineer's Notebook" on page 86) Figure 4-2 offers one for your consideration.

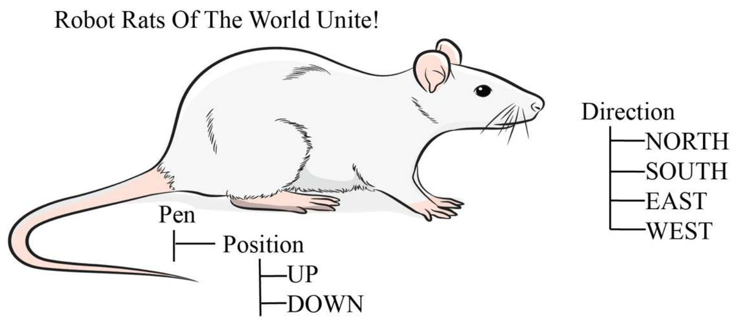


Figure 4-2: Robot Rat Viewed as a Collection of Attributes

Referring to figure 4-2 — This picture suggests that a robot rat, as defined by the current noun list, consists of a pen that has two possible positions, and the rat's direction. As described in the project specification and illustrated in figure 4-2, the pen can be either *UP* or *DOWN*. Regarding the robot rat's direction, it can face one of four ways: *NORTH*, *SOUTH*, *EAST*, or *WEST*. Can more attributes be derived? Perhaps another picture will yield more information. I recommend drawing a picture of the *floor* and run through several robot rat movement scenarios as illustrated in figure 4-3.

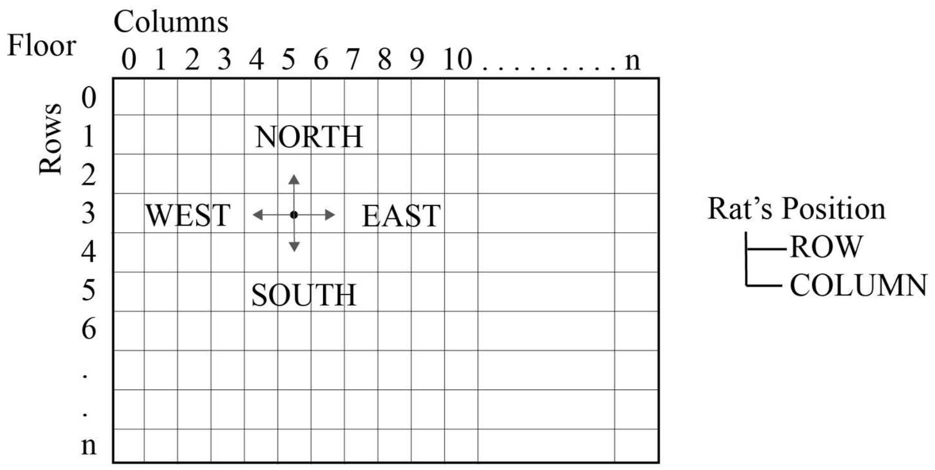


Figure 4-3: Robot Rat Floor Sketch

Figure 4-3 offers a lot of great information about the workings of a robot rat. The floor is represented by a collection of cells arranged by *rows* and *columns*. As the robot rat moves about the floor, its *position* can be determined by keeping track of its current *row* and *column*. These two nouns are good candidates to add to the list of relevant nouns and to the set of attributes that can be used to describe a robot rat. Before the robot rat can move, its current position on the floor must be determined. Upon completion of each robot rat movement, its current position must be updated. Armed with this information, you should now have a better understanding of what attributes are required to represent a robot rat, as figure 4-4 illustrates.

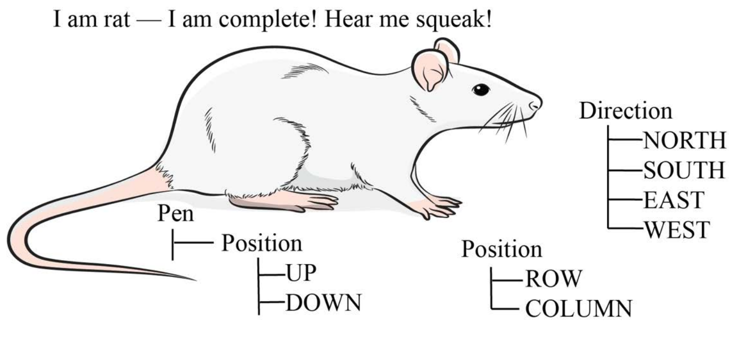


Figure 4-4: Complete Robot Rat Attributes

This seems to be a sufficient analysis of the problem at this point. You can return to this strategy area at any time should further analysis be required. It is now time to take a look at what language features you must understand to implement the solution.

3.1.3 Language Features

The purpose of the language features strategy area is two-fold: First, to derive a good design to a programming problem you must know what features the programming language supports and how it provides them. Second, you may be forced by a particular programming project to use language features you've never used before. It can be daunting to have lots of requirements thrown at you in one project. The complexities associated with learning the language, learning how to create projects, learning an integrated development environment (IDE), and learning the process of solving a problem with a computer can induce panic. Use the language features strategy area to overcome this problem and to *maintain a sense of forward momentum*.

Apply this strategy area by making a list of all the language features you need to study before starting your design and writing code. As you study each language feature, mark it off your list. Take notes about each language feature and how it can be applied to your particular problem.

Table 4-5 presents a sample check-off list for the language features used in the Robot Rat project.

Check	Feature	Considerations
	Python Applications	How do you write a Python application? What is a module? How do you run a Python application from the command line?
	Built-in Data Types	What are the Python built-in data types?
	Built-in Functions	What are the Python built-in functions?
	Arrays/Lists	What is an array? What is a list? How do you declare and initialize lists?
	Two-Dimensional Arrays/Lists	What is a two-dimensional array? How do you declare and initialize a two-dimensional array? How do you access each element in a two-dimensional array?
	Classes and Objects	How do you declare and implement a class? What's the structure of a class. What's the purpose of a class? What's the difference between a class definition vs. an object instance? How do you instantiate a class object? What's the purpose of the `__init__()` method?
	Instance Variables	What are instance variables? How do you declare and use an instance variable?
	Class Variables	What are class variables? How do you declare and use a class variable?
	Methods	What is a method? What are they good for? How do you declare and call a method? What are method parameters? How do you pass arguments to methods? How do you return values from methods? What's the difference between a method vs. a function?
	Local variables	What is a local variable? How does their use affect class or instance variables? How long does a local variable exist? What is the scope of a local variable?
	Special `__init__()` method	What's the purpose of the `__init__()` method. When is it called? How do you pass arguments to it?

Table 4-5: Language Feature Study Check-Off List For Robot Rat Project

Check	Feature	Considerations
	Control-Flow Statements	What is a control-flow statement? How do you use `if`, `while`, `for`, and `match` statements in a program? What's the difference between `for` and `while` statements? What's the difference between nested `if`/`elif`/`else` statements and `match` statements?
	Console I/O	What is console input and output? How do you write text to the console? How do you read text from the console and use it in your program? What built-in functions can you use to write text to the console and read text from the console?

Table 4-5: Language Feature Study Check-Off List For Robot Rat Project (Continued)

Armed with your list of language features, you can now study each one, marking it off as you go. When you discover a good code example that shows you how to use a particular language feature, copy it down or print it out and save it in your engineer's notebook for future reference.

Learning to program is a lot like learning to play a musical instrument. It takes observation and practice. You must put your trust in the masters and mimic their style. You may not at first fully understand why a particular piece of code works the way it does, or why they wrote it the way they did. But copy their style until you start to understand the underlying principles. Doing this builds confidence — slowly but surely. Soon you will have the skills required to set out on your own and write code with no help at all. In time, your programming skills will surpass those of your teachers.

After you have compiled and studied your list of language features, you should have a sense of what you can do with each feature and how to start the design process. More importantly, you will know where to refer when you need to study a particular language feature in more depth. However, by no means will you have mastered the use of these features. So don't feel discouraged if, having arrived at this point, you still feel a bit overwhelmed by all that you must know. *I must emphasize here that to master the art of programming takes practice, practice, practice!*

Once you have studied each required language feature, you are ready to move on to the design strategy area of the project-approach strategy.

3.1.4 Design

You must derive a plan of attack before you can solve the robot rat problem! Your plan will consist of two essential elements: a high-level *software architecture diagram* and an *implementation approach*.

3.1.4.1 High-Level Software Architecture Diagram

A high-level software-architecture diagram is a picture of both the software components needed to implement the solution and their relationship to each other. Creating the high-level software-architecture diagram for the Robot Rat project is easy, as the application will contain only one class. On the other hand, complex projects usually require many different classes, and each of these classes may interact with the others in some way. For these types of projects, application architecture diagrams play a key role in helping software engineers understand how the application works.

The Unified Modeling Language (UML) is used industry-wide to model software architectures. The UML class diagram for the RobotRatApp class at this early stage of your project's design will look similar to figure 4-5.

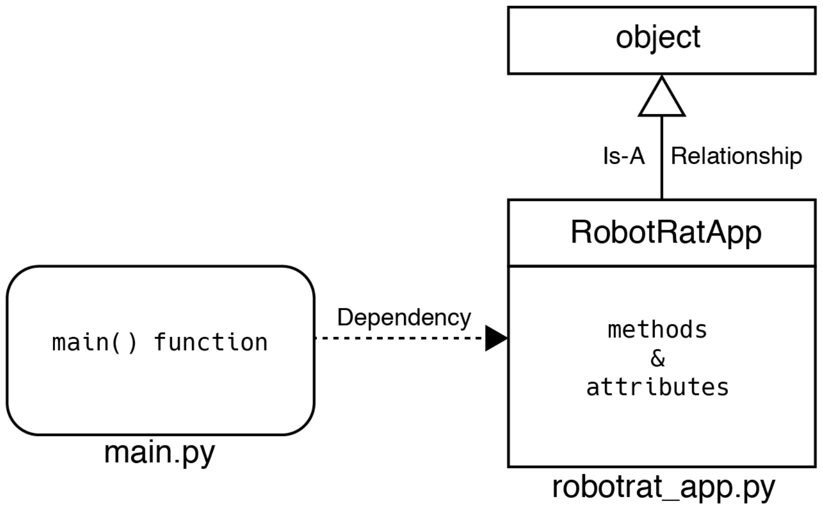


Figure 4-5: RobotRatApp UML Class Diagram

Referring to figure 4-5 — The RobotRatApp class extends (*inherits*) the functionality provided by the Python object class. This is indicated by the hollow-pointed arrow pointing from the RobotRatApp class to object. In Python 3, all user-defined classes implicitly extend object so you don't have to do anything special to achieve this functionality. The RobotRatApp class will have attributes (*class* and *instance variables*) and *methods*. The main.py module will provide the main() function and serve as the point of entry for the application.

3.1.4.2 Implementation Approach

Before you begin coding you must have some idea of how you are going to translate the design into a finished project. Essentially you must answer the following question: "Where do I start?" Getting started is easily 90% percent of the battle!

When formulating an implementation approach, you can proceed *macro-to-micro*, *micro-to-macro*, or a combination of both. I realize this sounds like unorthodox terminology, but bear with me.

If you use the macro-to-micro approach, you build and test an application *framework* to which you incrementally add functionality that ultimately results in a finished project. If you use the micro-to-macro approach, you build and test small pieces of functionality first and then, bit-by-bit, combine them into a finished project.

More often than not, you will use a combination of these approaches. Object-oriented design begs for macro-to-micro as a guiding approach. But both approaches play well with each other, as you will soon see.

There will be many unknowns when you start your design. For example, you could attempt to specify all the methods required for the RobotRatApp class up front, but as you progress through development, you will surely see the need for a function or method you didn't initially envision.

The following general steps outline a viable implementation approach to the Robot Rat project:

- Proceed from macro-to-micro by first creating and testing the RobotRatApp application class, devoid of any real functionality.
- Add and test a menu display capability.
- Add and test a menu-command processing framework by creating several empty methods that will serve as placeholders for future functionality. These methods are known as *method stubs*. (*Method stubbing is a great programming trick!*)
- Once you have tested the menu-command processing framework, you must implement each menu item's functionality. This means that you must implement and test the stub methods created in the previous step. The Robot Rat project is complete when all required functionality has been implemented and successfully tested.
- Develop the project iteratively in sprints. This means that you will repeatedly execute the *Plan-Code-Test-Integrate-Refactor* cycle many times on small pieces of the project until the project is complete.

Now that you have an overall implementation strategy, you can proceed with development. The following sections walk you step-by-step through the iterative application of the software development cycle.

4 Coding The Robot Rat Application

It's time now to begin coding the Robot Rat application. This section dives deeper into the thought process and coding techniques you can apply to implement a project of this nature. Note that unless you've worked through the project-approach strategy and have a good idea of where to begin you shouldn't be here. Have you ever sat down to write an essay for a class assignment only to stare hopelessly at the blank piece of paper laying before you with not one idea of what to write or how to begin. It's the same way for software development and the physical process of sitting at the keyboard and writing the code.

I'll proceed by applying the software development cycle in an iterative fashion called sprints. The first sprint, formally titled Sprint 0 (Zero), is usually included at the start of a development project to provide a period of time for a development team, or a team of one, to prepare for subsequent development activities. Shall we begin?

4.1 Planning — Sprint 0

Before writing one line of code you'll need to take care of a few administrative tasks. These could be just about anything that prepares a team for the work ahead, but here I'll focus on creating the project folder and project repository. I'll be using Git for source code configuration management and GitHub for the remote repository, however, I will not go into detail here about how to use Git or GitHub. For detailed coverage please refer to *Part II: Chapter 8: Configuration Management with Git & GitHub*.

Pro Tip: Use Sprint 0 to identify and execute tasks required to ensure project success

4.1.1 Activities

A few important or even critical activities need to be completed before writing one line of code for the Robot Rat application. You should do the ones discussed here for all the programming projects you work on. The exact activities included in a Sprint 0 in the real world vary according to company, project, and even personalities working on the team. For Robot Rat, I'm primarily concerned with *where the source code is stored*, *how the project folder is organized*, and *setting up the repository*. Table 4-6 lists the task items I'll work during this sprint.

Check	Task	Action
	Create Repository in GitHub	Create a GitHub repository for the project.
	Create Local Projects Directory	Create a local projects directory. (*~/dev* or *~/projects*)
	Clone the Repository	Clone the repository into the projects folder.
	Create Robot Rat Project Directory	Navigate to the cloned repository directory and create the Robot Rat project directory.
	Add README.md File	Add a README.md file with project details
	Add .gitignore File	Add a .gitignore file which indicates which project files to ignore
	Verify Repository Operations	Add, commit, and push changes to the local repository up to GitHub to test repository operations.

Table 4-6: Sprint 0 Activities

Referring to table 4-6 — As I said earlier, you may not know one thing about Git or GitHub at this point, and that is perfectly OK. I'm not going into details here, but you'll find it helpful to observe the thought process and motions from the 10,000 foot level.

4.1.1.1 Create Repository in GitHub

If you don't already have a repository for the project create one in GitHub. I have a repository I use for this book so I'll be using it to store the code for the Robot Rat project. Here's the link to the repository: *https://github.com/pulpfreepress/cst_with_python_1st_ed*

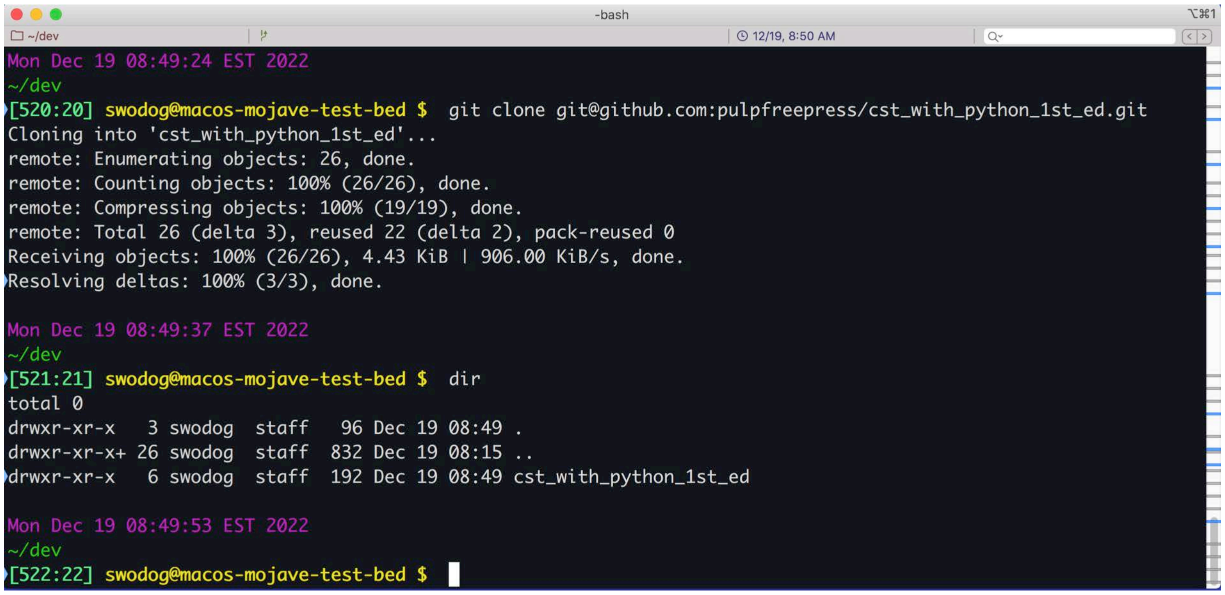
So, you can proceed a couple ways here. You can create a unique repository for each programming project you work on, or, you can create a catch-all repository to which you add your projects to as you create and work on them. The former works well for big projects and the later works well for students who have multiple projects related to a course. It also works well for books with source code located in different chapters and is the approach I use here.

4.1.1.2 Identify Local Projects Directory

I've recommended since chapter 1 you should store your programming projects in a dedicated subdirectory off your home directory. I locate all my programming projects in a subdirectory named `dev` (*~/dev*).

4.1.1.3 Clone Repository

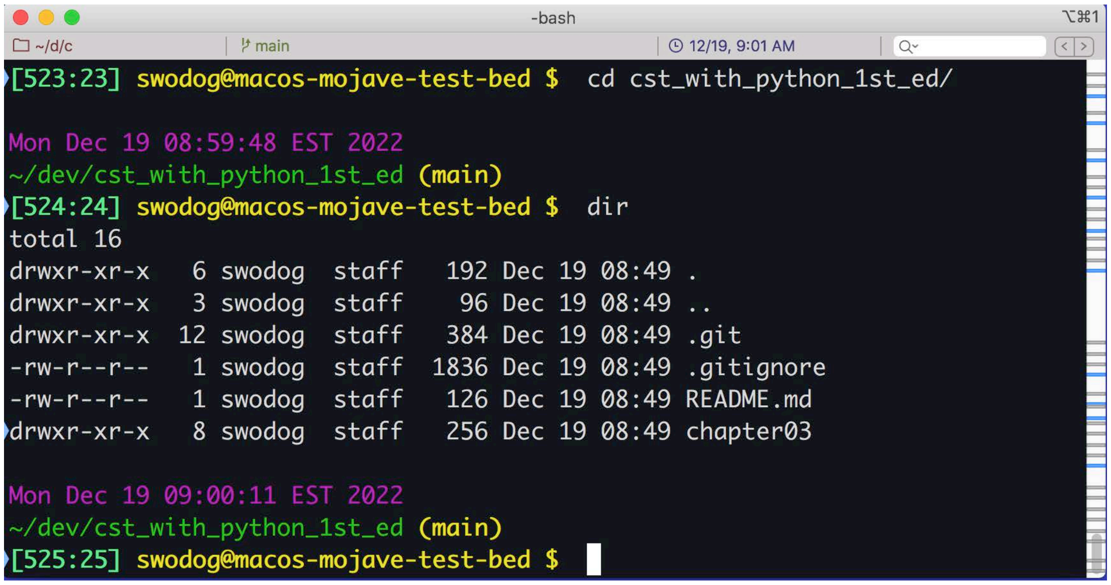
When you clone a GitHub repository, the name of the repository will be the name of the folder created in your projects folder. See figure 4-6.



```
Mon Dec 19 08:49:24 EST 2022
~/dev
[520:20] swodog@macos-mojave-test-bed $  git clone git@github.com:pulpfreepress/cst_with_python_1st_ed.git
Cloning into 'cst_with_python_1st_ed'...
remote: Enumerating objects: 26, done.
remote: Counting objects: 100% (26/26), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 26 (delta 3), reused 22 (delta 2), pack-reused 0
Receiving objects: 100% (26/26), 4.43 KiB | 906.00 KiB/s, done.
Resolving deltas: 100% (3/3), done.

Mon Dec 19 08:49:37 EST 2022
~/dev
[521:21] swodog@macos-mojave-test-bed $  dir
total 0
drwxr-xr-x   3 swodog  staff   96 Dec 19 08:49 .
drwxr-xr-x+ 26 swodog  staff  832 Dec 19 08:15 ..
drwxr-xr-x   6 swodog  staff  192 Dec 19 08:49 cst_with_python_1st_ed

Mon Dec 19 08:49:53 EST 2022
~/dev
[522:22] swodog@macos-mojave-test-bed $
```

Figure 4-6: Cloning Repository Into Development Projects Folder

Referring to figure 4-6 — I cloned the repository into my *~/dev* directory using SSH (Secure Shell). (See *Chapter 8: Configuration Management with Git & GitHub*). This creates a folder in the *~/dev* directory with the same name as the repository (*cst_with_python_1st_ed*). This is referred to as the local repository folder and notice when I change directory into that folder the prompt will change to reflect the repository branch name as shown in figure 4-7.



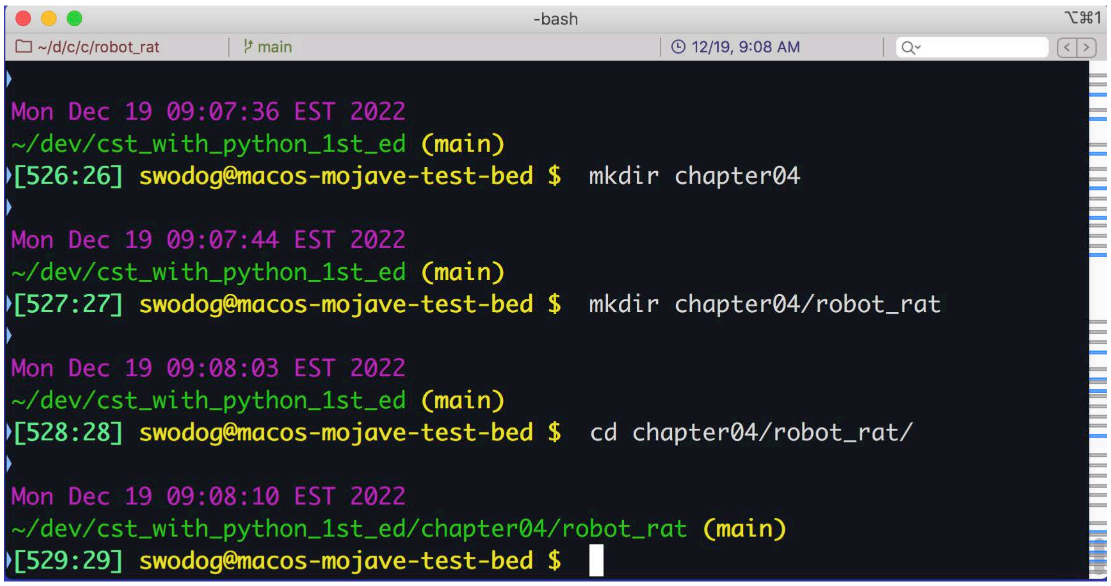
```
[523:23] swodog@macos-mojave-test-bed $  cd cst_with_python_1st_ed/

Mon Dec 19 08:59:48 EST 2022
~/dev/cst_with_python_1st_ed (main)
[524:24] swodog@macos-mojave-test-bed $  dir
total 16
drwxr-xr-x   6 swodog  staff   192 Dec 19 08:49 .
drwxr-xr-x   3 swodog  staff    96 Dec 19 08:49 ..
drwxr-xr-x  12 swodog  staff   384 Dec 19 08:49 .git
-rw-r--r--   1 swodog  staff  1836 Dec 19 08:49 .gitignore
-rw-r--r--   1 swodog  staff   126 Dec 19 08:49 README.md
drwxr-xr-x   8 swodog  staff   256 Dec 19 08:49 chapter03

Mon Dec 19 09:00:11 EST 2022
~/dev/cst_with_python_1st_ed (main)
[525:25] swodog@macos-mojave-test-bed $
```

Figure 4-7: Repository Branch Displayed

Referring to figure 4-7 — The (main) indicates the current repository branch. This is the *local repository*. Now it's time to create the Robot Rat project folder.

4.1.1.4 Create Robot Rat Project Directory

Notice in figure 4-7 I have organized the code for this book into chapter folders. I'll create a new directory named *chapter04* and in there I'll create a subdirectory named *robot_rat* as shown in figure 4-8.



```
Mon Dec 19 09:07:36 EST 2022
~/dev/cst_with_python_1st_ed (main)
[526:26] swodog@macos-mojave-test-bed $  mkdir chapter04

Mon Dec 19 09:07:44 EST 2022
~/dev/cst_with_python_1st_ed (main)
[527:27] swodog@macos-mojave-test-bed $  mkdir chapter04/robot_rat

Mon Dec 19 09:08:03 EST 2022
~/dev/cst_with_python_1st_ed (main)
[528:28] swodog@macos-mojave-test-bed $  cd chapter04/robot_rat/

Mon Dec 19 09:08:10 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat (main)
[529:29] swodog@macos-mojave-test-bed $
```

Figure 4-8: Creating Robot Rat Project Folder

Referring to figure 4-8 — The full path to the Robot Rat project folder on my machine is *~/dev/cst_with_python_1st_ed/chapter04/robot_rat*. Your repository structure may be completely different or similar with slight variations. Either way, you should now navigate to the *robot_rat* project folder and add a few important artifacts as discussed below.

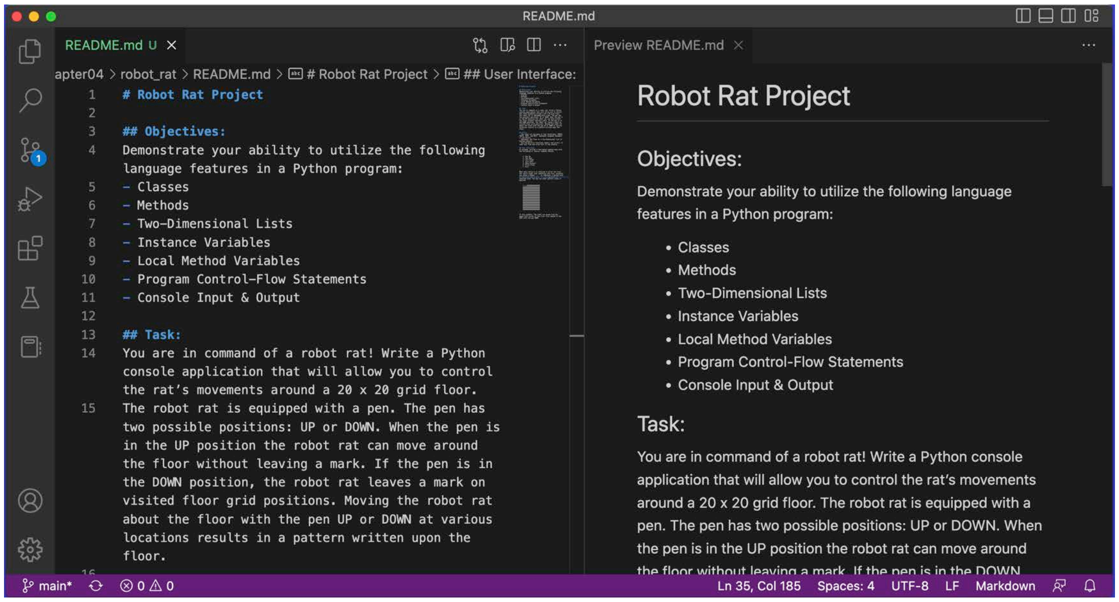
4.1.1.5 Add README.md File

Add a *README.md* file. The purpose of a README.md file is to provide documentation about your project. Topics you might want to add include notes on how to configure and run the project, any preliminary actions users must take before they can run the program like what required packages they may need install, and so on.

A README.md file contains *Markdown* code. You can learn more about Markdown here: *https://www.markdownguide.org/cheat-sheet/* It's easy to figure out. Start by adding a project title and short description. You can always add more information to your README.md as development progresses.

Figure 4-9 shows the raw Markdown code in the left editor panel and the rendered Markdown in the right panel. The rendered Markdown panel shows what your README.md file will look like when displayed on GitHub.

Referring to figure 4-9 — Refer to the Markdown cheat-sheet to decipher the formatting used in the left-hand panel. Although my primary repository has a README.md file at the root level, I add one to sub-projects when necessary to add clarification or specific instructions. I've often referred to my README.md files to refresh my memory on a particular project. Documentation on school projects may seem silly but your instructor will be impressed.


README.md

README.md U ×   Preview README.md ×

apter04 > robot_rat > README.md > # Robot Rat Project > ## User Interface:

```
# Robot Rat Project

## Objectives:
Demonstrate your ability to utilize the following
language features in a Python program:
- Classes
- Methods
- Two-Dimensional Lists
- Instance Variables
- Local Method Variables
- Program Control-Flow Statements
- Console Input & Output

## Task:
You are in command of a robot rat! Write a Python
console application that will allow you to control
the rat's movements around a 20 x 20 grid floor.
The robot rat is equipped with a pen. The pen has
two possible positions: UP or DOWN. When the pen is
in the UP position the robot rat can move around
the floor without leaving a mark. If the pen is in
the DOWN position, the robot rat leaves a mark on
visited floor grid positions. Moving the robot rat
about the floor with the pen UP or DOWN at various
locations results in a pattern written upon the
floor.
```

Robot Rat Project

Objectives:

Demonstrate your ability to utilize the following language features in a Python program:

- Classes
- Methods
- Two-Dimensional Lists
- Instance Variables
- Local Method Variables
- Program Control-Flow Statements
- Console Input & Output

Task:

You are in command of a robot rat! Write a Python console application that will allow you to control the rat's movements around a 20 x 20 grid floor. The robot rat is equipped with a pen. The pen has two possible positions: UP or DOWN. When the pen is in the UP position the robot rat can move around the floor without leaving a mark. If the pen is in the DOWN

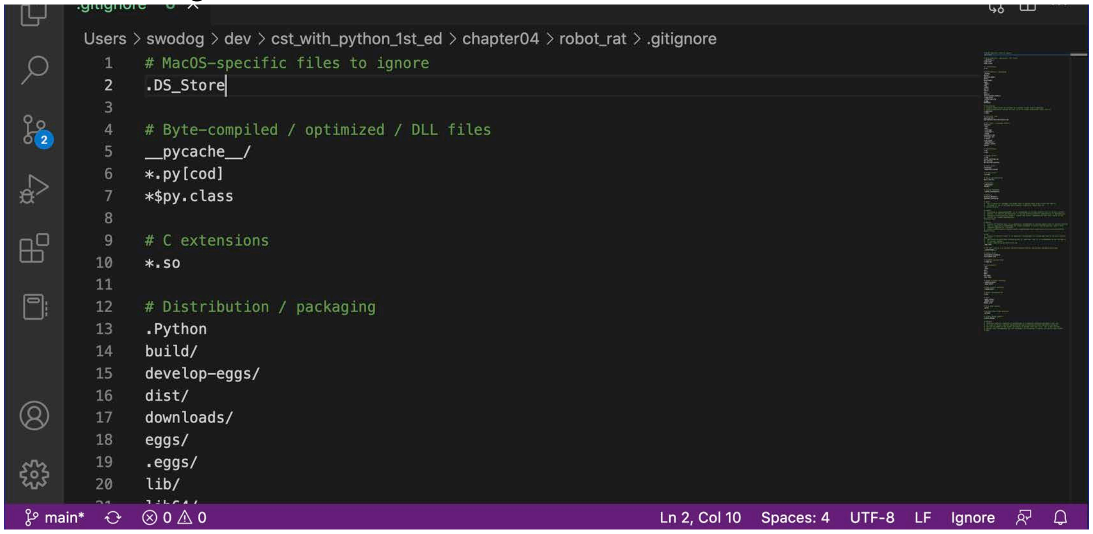
main* ⊗ 0 ⚠ 0   Ln 35, Col 185   Spaces: 4   UTF-8   LF   Markdown

Figure 4-9: Adding Markdown-Formatted Content to README.md

4.1.1.6 Add .gitignore File

Next, create a *.gitignore* file. The .gitignore file is a list of files and directories to ignore when committing project artifacts to the local repository. You can find .gitignore templates, or pre-populated .gitignore files, tailored to specific project types. When you create a repository in GitHub you have the chance to select a .gitignore template to use. Here's a link to a Python project .gitignore template on GitHub: *https://github.com/github/gitignore/blob/main/Python.gitignore*

Like README.md files, .gitignore files can sit at the root of a repository with additional .gitignore files in sub-directories as required. That's the approach I'm taking here. To the template file found at the other end of that link I pasted above, I'm going to add a common macOS file to ignore as shown in figure 4-10.


Users > swodog > dev > cst_with_python_1st_ed > chapter04 > robot_rat > .gitignore

```
# MacOS-specific files to ignore
.DS_Store

# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
```

main* ⊗ 0 ⚠ 0   Ln 2, Col 10   Spaces: 4   UTF-8   LF   Ignore

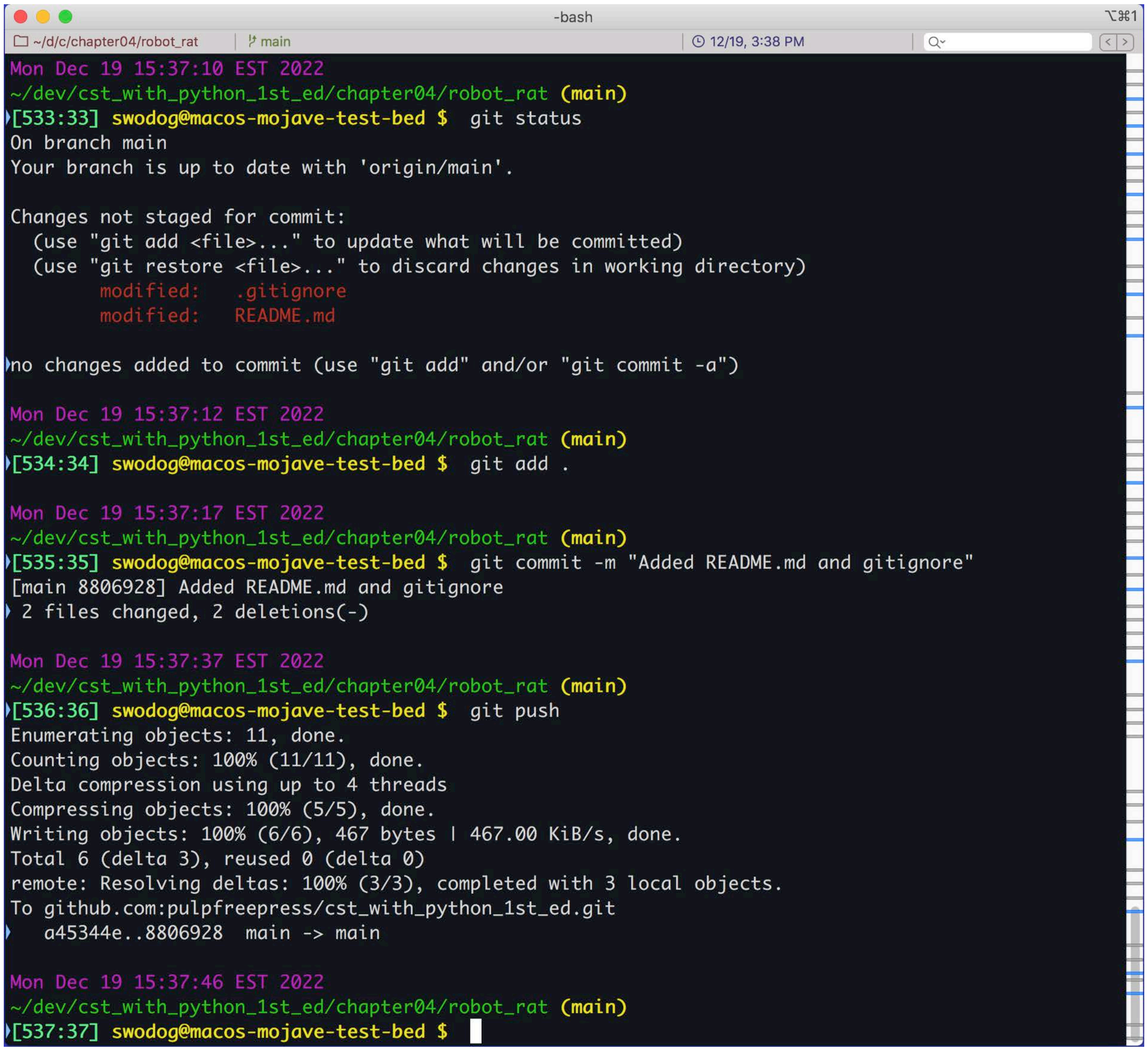
Figure 4-10: Adding .gitignore File to Project

Referring to figure 4-10 — I will run this project on multiple platforms including Windows, Linux, and macOS. On macOS, browsing to a directory with a Finder window creates a hidden file named .DS_Store. I always add an entry in my .gitignore files to ignore .DS_Store files.

OK, it's time to verify repository operations.

4.1.1.7 Verify Repository Operations

Now that you have your project directory and have added the README.md and .gitignore files, you can use the `git` command to *add* and *commit* them to the local repository, and then *push* them to the remote repository on GitHub. Figure 4-11 shows you the console session for this series of `git` operations.



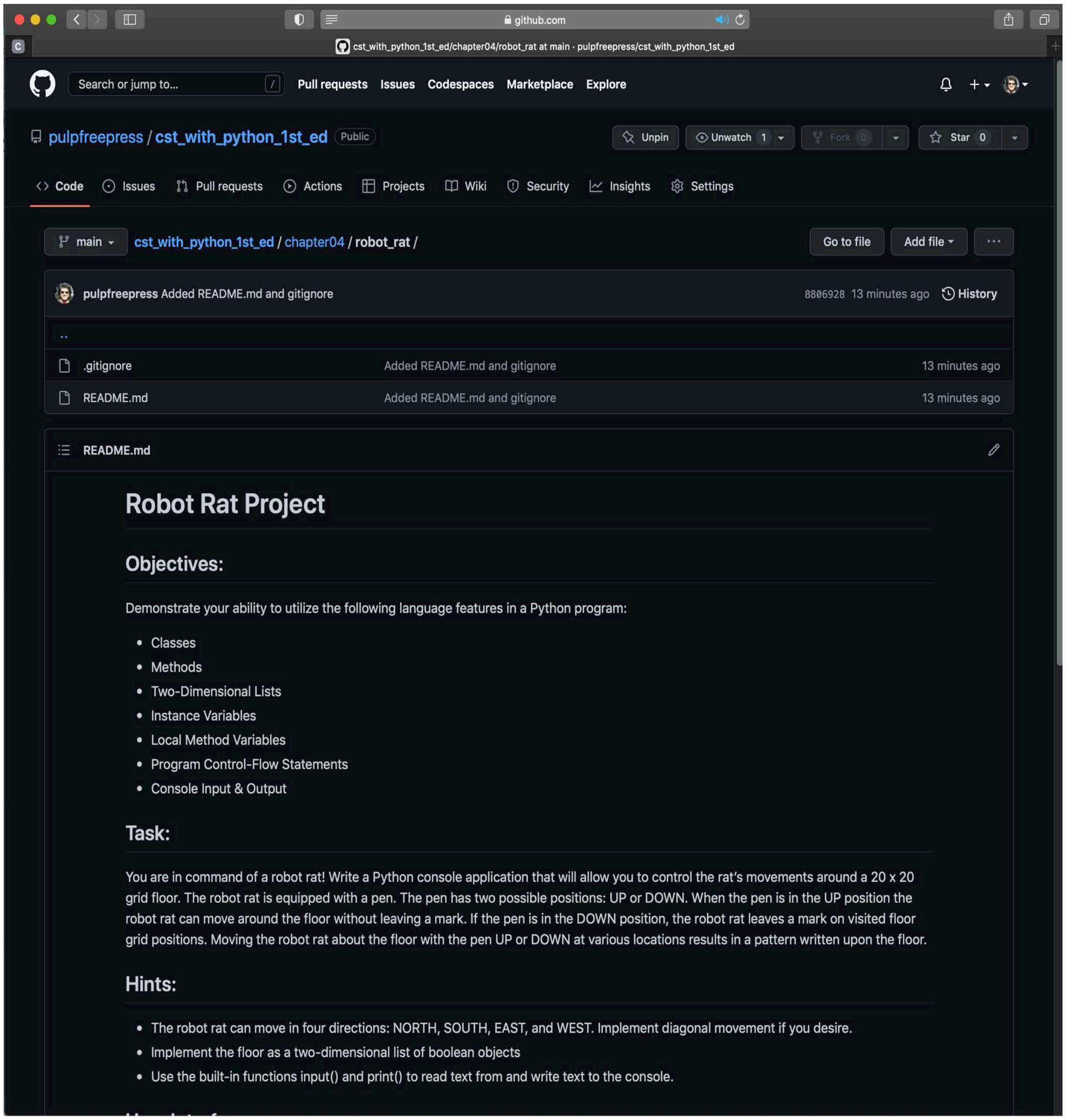
```
Mon Dec 19 15:37:10 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat (main)
[533:33] swodog@macos-mojave-test-bed $  git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .gitignore
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")

Mon Dec 19 15:37:12 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat (main)
[534:34] swodog@macos-mojave-test-bed $  git add .

Mon Dec 19 15:37:17 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat (main)
[535:35] swodog@macos-mojave-test-bed $  git commit -m "Added README.md and gitignore"
[main 8806928] Added README.md and gitignore
 2 files changed, 2 deletions(-)

Mon Dec 19 15:37:37 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat (main)
[536:36] swodog@macos-mojave-test-bed $  git push
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 4 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 467 bytes | 467.00 KiB/s, done.
Total 6 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To github.com:pulpfreepress/cst_with_python_1st_ed.git
   a45344e..8806928  main -> main

Mon Dec 19 15:37:46 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat (main)
[537:37] swodog@macos-mojave-test-bed $
```

Figure 4-11: Git Operations *add* and *commit* with a *push* to the Remote Repository

Referring to figure 4-11 — I personally like to check the status of the local repository by running the `git status` command. I then do a `git add .` to add everything that is new or modified, followed by a `git commit -m "message..."` where `"message..."` is a comment on the commit. Finally, I push the local changes to the remote repository with `git push`.

0
0
0
0
0
1
0
0

To see the remote repository updates, navigate to your GitHub repository. Figure 4-12 shows the *robot_rat* project repository on GitHub.



pulpfreepress Added README.md and gitignore 8806928 13 minutes ago

.gitignore	Added README.md and gitignore	13 minutes ago
README.md	Added README.md and gitignore	13 minutes ago

README.md

Robot Rat Project

Objectives:

Demonstrate your ability to utilize the following language features in a Python program:

- Classes
- Methods
- Two-Dimensional Lists
- Instance Variables
- Local Method Variables
- Program Control-Flow Statements
- Console Input & Output

Task:

You are in command of a robot rat! Write a Python console application that will allow you to control the rat's movements around a 20 x 20 grid floor. The robot rat is equipped with a pen. The pen has two possible positions: UP or DOWN. When the pen is in the UP position the robot rat can move around the floor without leaving a mark. If the pen is in the DOWN position, the robot rat leaves a mark on visited floor grid positions. Moving the robot rat about the floor with the pen UP or DOWN at various locations results in a pattern written upon the floor.

Hints:

- The robot rat can move in four directions: NORTH, SOUTH, EAST, and WEST. Implement diagonal movement if you desire.
- Implement the floor as a two-dimensional list of boolean objects
- Use the built-in functions input() and print() to read text from and write text to the console.

Figure 4-12: GitHub Repository with robot_rat Project Folder Added

Referring to figure 4-12 — Notice how nice the README.md file looks. Thoughtful documentation makes even mundane projects look more professional. As the Robot Rat project progresses through its development sprints, I will create new subdirectories for each sprint so you can see the evolution of the project. With Sprint 0 activities complete it's time to start the first development sprint.

Pro Tip: Use Sprint 0 to execute tasks that set subsequent sprints up for success

0
0
0
0
0
1
0
0

4.2 Development — Sprint 1

With Sprint 0 complete it's time to start coding. This is where the software development cycle kicks in. While Sprint 0 is considered a planning sprint of sorts, at the heart of the Agile philosophy is the notion of not planning too much in advance. Each development sprint consists of a planning phase. That's where I'll start. Think of development as getting a flywheel going. You seek momentum and you get into a rhythm when things start humming along.

4.2.1 Plan

The focus of this sprint will be to create an application architecture which will support further Robot Rat development. Using figure 4-5 as a reference I'll be creating a two-module application architecture with a *robotrat_app.py* module and a *main.py* module. The robotrat_app.py module will contain the *RobotRatApp* class definition and the main.py module will serve as the point of entry for the application. Table 4-7 lists the design considerations and decisions for what needs to be done.

Check	Design Consideration	Design Decision
	Create a *src* directory to store source code files.	All Python source code files will be located in the *src* subdirectory.
	Create RobotRatApp class	The objective here is to define the class so that an object can be instantiated. It doesn't need to have any real functionality. The class will be defined in the *robotrat_app.py* module.
	Define a *main()* method	Locate the *main()* method in the *main.py* module

Table 4-7: Sprint 1 Design Considerations

Referring to table 4-7 — This is a great start. It may not seem like a lot but it lays the code foundation that will facilitate further application development.

The first thing I'll do is the first activity on the list — create a new subdirectory in the robot_app project folder named *src*, which is where I will store all project source code files. You'll want to do this to avoid cluttering the project's root directory. Now, let's start coding.

4.2.2 Code

I'll start with the robotrat_app.py file shown in example 4.1.

4.1 robotrat_app.py (Sprint 1)

```
"""Implements the Robot Rat Application."""

class RobotRatApp():
    """A Remote-Controlled Robot Rat Application."""

    def __init__(self):
        print('I am Robot Rat! I am alive!')
```

Referring to example 4.1. The goal of this code is to just define the RobotRatApp class and see some type of indication when an object is instantiated and initialized. The `__init__()` method is a special method called after an object is instantiated. Its purpose is to initialize the object. Here it’s only printing a message to the console. What, exactly, goes here you may not fully understand at this point of development. By that I mean that when you first start coding, you may not have a full grasp of what needs to be initialized. Those issues will come into better focus as development progresses.

OK, now it’s time to create the main.py module, the code for which is given in example 4.2.

4.2 main.py

```
1    """Serves as the point of entry to the Robot Rat Application."""
2
3    from robotrat_app import RobotRatApp
4
5    def main():
6        robot_rat_app = RobotRatApp()
7
8
9    if __name__ == '__main__':
10       main()
```

Referring to example 4.2 — The main.py module is the file that will be executed by the Python interpreter to run the Robot Rat application. On line 3 it imports the RobotRatApp class from the robotrat_app module. Importing makes namespaces available to a program by loading and executing the imported module code. The `main()` method is defined on line 5. On line 6 an instance of the `RobotRatApp` class is created by invoking or "calling" an instantiation operation with a function call "`()`" represented by a pair of parentheses. The instantiation call returns an object which is an instance of the RobotRatApp class. The `main()` method is called on line 10 when the main.py module is run directly by the Python interpreter.

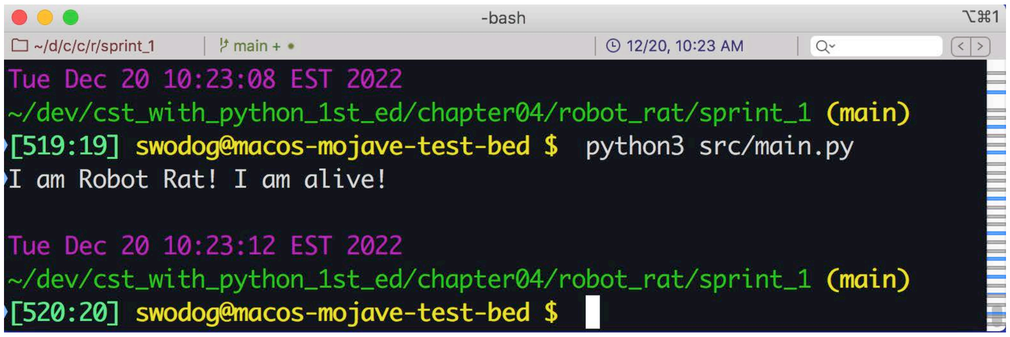
Coding is complete. If you’re following along, save these two files in the project *src* directory and proceed to the Test phase.

4.2.3 Test

Testing at this point is simple — just run the *main.py* module with the Python interpreter. Remember to use the appropriate Python interpreter command that corresponds to your operating system. It will be either `python` (Windows) or `python3` (Linux/macOS). Here’s the command I run on macOS:

```
python3 src/main.py
```

Run this command from the Robot Rat root project directory. The results of this command are shown in figure 4-13.



```
Tue Dec 20 10:23:08 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat/sprint_1 (main)
[519:19] swodog@macos-mojave-test-bed $  python3 src/main.py
I am Robot Rat! I am alive!

Tue Dec 20 10:23:12 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat/sprint_1 (main)
[520:20] swodog@macos-mojave-test-bed $
```

Figure 4-13: Results of Running *main.py* Module Sprint 1

Referring to figure 4-13 — Everything is working as it should be. Time to move on.

4.2.4 Integrate

There's really nothing to integrate so we can largely skip this step of the development cycle at this stage.

4.2.5 Add, Commit, And Push Changes To Repository

This does not necessarily have to wait until the end of the sprint. In the real world, a sprint may last several weeks or more, and you may make several commits during that time. However, at the end of a sprint you'll want to commit any remaining additions and modifications to the code base and push them up to the remote repository. I use the following series of commands:

```
git status # List what's new and modified in the working area
git add . # Add all new and modified artifacts to the staging area
git commit -m "Completed Sprint 1." # Record the changes to the repository
git push # Update remote (GitHub) repository
```

Going forward, I will omit this section. However, at minimum, you'll want to commit changes at the end of each development sprint. Another good time to commit changes to your repository is right before you start to make a sweeping change to existing code. An example of this would be right before refactoring. If something goes wrong, you can back out those changes and recover to a known good state.

4.2.6 Refactor

There's nothing to refactor so we can skip this step for this sprint.

4.2.7 Parting Thoughts

This marks the end of Sprint 1. While it may not seem like a lot was accomplished, that notion couldn't be farther from the truth. Novices often, well, more often than not, make the mistake of trying to do too much coding before they try to run their project. Then they get depressed.

What normally happens, especially to those new to programming or new to Python, is they make mistakes when entering the code. Learning how to physically enter code using a keyboard takes an immense amount of focus, concentration, and hand-eye coordination. Python is case-sensitive, so you have to pay attention to what you're typing and be exact. Getting good at *paying attention* and *being exact* is a large part of learning to code. OK, let's move on to the next sprint.

4.3 Development — Sprint 2

At this point, I have laid a good code foundation upon which to start adding functionality that will bring the Robot Rat project to life. A lot of what a Robot Rat can and must do is pretty much spelled out in the command menu. That seems like the next logical thing to work on.

4.3.1 Plan

The project specification says the Robot Rat application must display a menu with a list of command choices from which a user can select. Referring to the noun/verb analysis table, there's an entry in the Verbs column for "display menu". Recall that nouns represent potential entities within a program while verbs represent potential actions an application must perform. Verbs also map nicely to functions or methods.

Now, get up from your desk and take a walk. Walking provides a great opportunity to think. Issues to contemplate include the weather, how nice the sun feels upon your face, the sound of the leaves rustling in the trees, or the sound of traffic, beeping, honking, and *how to render the menu*. Since the project specification requires console I/O you'll use the built-in `print()` function to render menu items. But you'll need to display the menu repeatedly, and although you may not know how to do that just yet, it makes sense to put the menu rendering code in a dedicated method named `display_menu()`. Table 4-8 lists the design considerations for sprint 2.

Check	Design Consideration	Design Decision
	Display command menu	Use the built-in `print()` function to print each menu item to the console. Consolidate the menu rendering code in a method named `display_menu()`,

Table 4-8: Sprint 2 Design Considerations

This looks like plenty to do for sprint 2. Let's get coding.

4.3.2 Code

I'll add the `display_menu()` method definition to the existing *RobotRatApp class*. Example 4.3 gives the code listing.

4.3 robotrat_app.py (Sprint 2)

```
1    """Implements the Robot Rat Application."""
2
3    class RobotRatApp():
4       """A Remote-Controlled Robot Rat Application."""
5
6       def __init__(self):
7          print('I am Robot Rat! I am alive!')
8
9       def display_menu(self):
10         """Prints menu items to the console."""
11         print('\n\t\tRobot Rat Control Menu')
12         print('\t1. Pen Up')
13         print('\t2. Pen Down')
14         print('\t3. Turn Right')
15         print('\t4. Turn Left')
16         print('\t5. Move Forward')
17         print('\t6. Print Floor')
18         print('\t7. Exit')
19
```

Referring to example 4.3 — The `display_menu()` method definition begins on line 9. It begins with a docstring on line 10 followed by a series of built-in `print()` function calls that print various strings to the console related to the menu. Notice all the strings are single-quoted as that is the style I have adopted. Notice also the strings contain special escaped characters `'\t'`, `'\n'` or both. The `'\t'` is the escaped *tab character*. I use them to position the menu a little to the right in the console, as you'll see when I run the program. — OK, to see the menu, I'll need to *call* the `display_menu()` method somewhere. I'll do that in the *main.py* module listed in example 4.4.

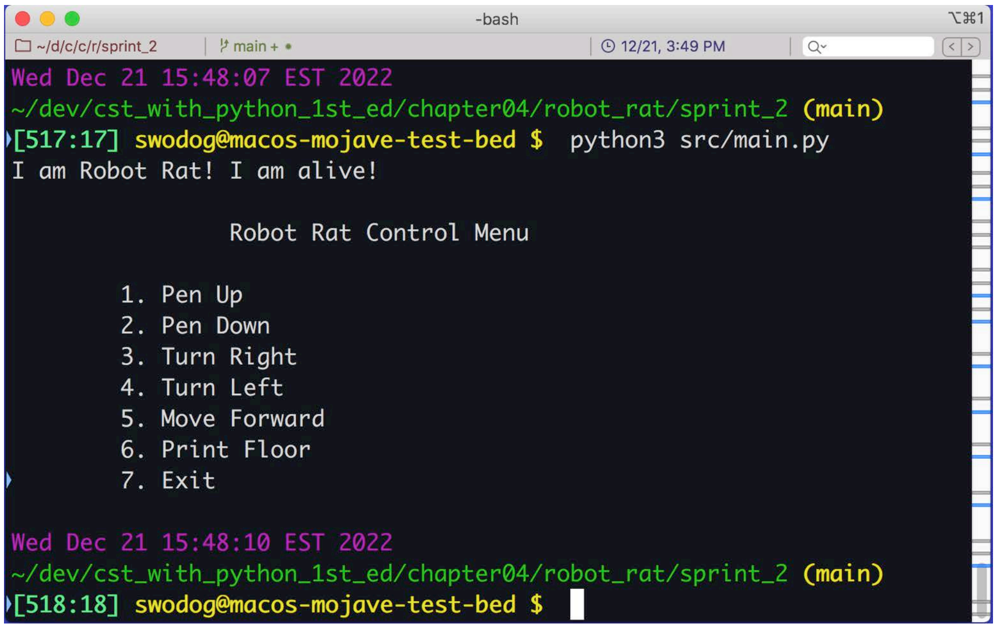
4.4 main.py (Sprint 2)

```
1    """Serves as the point of entry to the Robot Rat Application."""
2
3    from robotrat_app import RobotRatApp
4
5    def main():
6        robot_rat_app = RobotRatApp()
7        robot_rat_app.display_menu()
8
9
10   if __name__ == '__main__':
11       main()
12
```

Referring to example 4.4 — On line 7, I added a call to the `robot_rat_app.display_menu()` method. A few things to note here. First, on line 6, I'm creating an instance of the RobotRatApp class whose location in memory is assigned to the variable `robot_rat_app`. This variable now points to an object of type *RobotRatApp*. On line 7, I apply the dot `'.'` operator to call a method defined by that type, namely, in this case, the `display_menu()` method. It is the call operator, a pair of parentheses, `'()'`, that actually invokes the method call. — OK, time to test this code.

4.3.3 Test

Testing in this case involves nothing more than running the code. Once again, since I'm programming on macOS, I open a terminal and run the main.py module with the Python interpreter like so: `python3 src/main.py` Figure 4-14.



```
Wed Dec 21 15:48:07 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat/sprint_2 (main)
[517:17] swodog@macos-mojave-test-bed $  python3 src/main.py
I am Robot Rat! I am alive!

                Robot Rat Control Menu

        1. Pen Up
        2. Pen Down
        3. Turn Right
        4. Turn Left
        5. Move Forward
        6. Print Floor
        7. Exit

Wed Dec 21 15:48:10 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat/sprint_2 (main)
[518:18] swodog@macos-mojave-test-bed $
```

Figure 4-14: Testing Robot Rat Control Menu — Sprint 2

Referring to figure 4-14 — You can see the effects of the tab characters. Placing character stream output in the console is tedious so this menu doesn't need to be perfect. I may adjust the menu in another sprint, but for now it looks fine. The only real issue I have at this point is the menu displays and then the program exists. How does a user make a menu selection and have the program respond? Good question, and that sounds like something interesting to work on in the next sprint.

4.3.4 Integrate

Nothing to integrate, but I can remove the message from the `__init__()` method since I can now see the menu when I run the program. The last thing I'll do before moving to Sprint 3 is push all my changes to the remote repository.

4.4 Development — Sprint 3

At this point the menu displays once and the program exits. A good set of objectives for this sprint will be to continuously display the menu until the user exits the program and handle, in some way, user menu selections, if not completely, then in some limited fashion.

There's a technique programmers use called *stubbing* to implement a feature without having to fully develop application code. For example, it would be nice to work on menu selection processing so that when a user selects, say, the Pen Up command, we can test the menu processing feature without actually implementing Robot Rat's Pen Up capability. You'll find stubbing to be a valuable tool to add to your programmer's tool belt.

4.4.1 Plan

Table 4-9 lists the design considerations for this sprint.

Check	Design Consideration	Design Decision
	Menu Item Processing	When a user selects a menu item the application needs to read the user's command-line input and execute the indicated command. This can be handled in a separate method named `process_menu_choice()`.
	Menu Display and Processing Loop	Upon completion of a Robot Rat menu command, the application should redisplay the menu and await user input.
	Menu Command Method Stubs	Stub out the methods for each menu command. Display a message to the screen to indicate when a method has been called.

Table 4-9: Sprint 3 Design Considerations

Referring to table 4-9 — This looks like plenty to do for this sprint. Let's write some code.

4.4.2 Code - First Iteration

The best advice I can offer here is not to try to code everything before doing some testing to see how things are going. To show you what I mean, I'll post the examples in stages to show you how far I'd go before stopping and testing to make sure what I am coding actually works.

I'll start by creating a method named `process_menu_choice()` and write enough code to test one or two commands before proceeding farther. What this means is that within this sprint there will be several *Plan*, *Code*, & *Test* sub-cycles. While table 4-9 above lists the overall sprint objectives, each design consideration may entail deeper thought and additional planning to fully implement.

To process a menu choice a user must enter the command number at the console. The program should prompt the user for input. The built-in `input()` function is perfect for the job. When the user enters a menu command the application must then try to figure out if what the user typed is a valid menu command and if not it should display a warning message and let the user try again. Note that if you don't know how to do these things you can always create a recipe for what needs to be done using pseudocode. You could write these steps in your engineer's notebook or put them in your program as comments.

Example 4.5 gives the code for my first stab at the `process_menu_choice()` method.

4.5 robotrat_app.py (Sprint 3 v1)

```
1    """Implements the Robot Rat Application."""
2
3    class RobotRatApp():
4        """A Remote-Controlled Robot Rat Application."""
5
6        def __init__(self):
7            print('I am Robot Rat! I am alive!')
8
9        def display_menu(self):
10           """Prints menu items to the console."""
11           print('\n\t\tRobot Rat Control Menu\n')
12           print('\t1. Pen Up')
13           print('\t2. Pen Down')
14           print('\t3. Turn Right')
15           print('\t4. Turn Left')
16           print('\t5. Move Forward')
17           print('\t6. Print Floor')
18           print('\t7. Exit')
19
20       def process_menu_choice(self):
21           # Prompt user for input
22           # Assign input string to variable
23           user_input = input('\n\tEnter Command Number: ')
24           # Use first character of input as menu choice
25           menu_choice = user_input[0]
26           if __debug__:
27               print(f'You entered command number:  {menu_choice}')
28           # Is menu_choice valid command?
29           # YES - Execute command
30           # NO - Display error message and try again
31
```

Referring to example 4.5 — The `process_menu_choice()` method definition starts on line 20. The comments represent the algorithm or steps necessary to fully implement the method although some details about how to "try again" are still fuzzy. So, at this stage of development, all the method does is display a prompt and assign the console input to a variable named `user_input` on line 23. Then, on line 25, the first character of the `user_input` string (`user_input[0]`) is assigned to the variable `menu_choice`. Then, on line 26, if the Python built-in constant `__debug__` is true, then the message on line 27 prints to the console. This may not seem like a lot, but it's worth testing to see if everything works up to this point.

To test the `process_menu_choice()` method I'll need to call it from the *main.py* module as is shown in example 4.6.

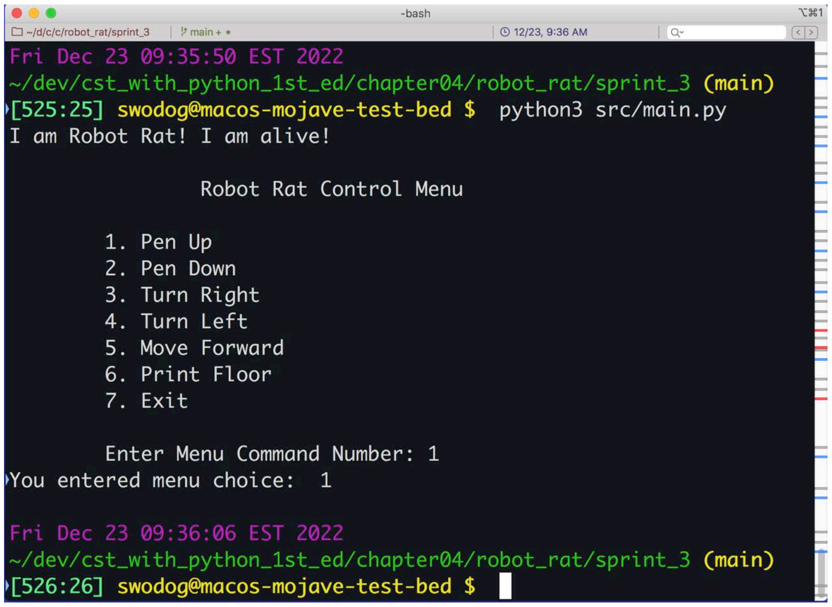
4.6 main.py (Sprint 3 v1)

```
1    """Serves as the point of entry to the Robot Rat Application."""
2
3    from robotrat_app import RobotRatApp
4
5    def main():
6        robot_rat_app = RobotRatApp()
7        robot_rat_app.display_menu()
8        robot_rat_app.process_menu_choice()
9
10
11   if __name__ == '__main__':
12       main()
13
```

Referring to example 4.6 — I'm making a call to the `process_menu_choice()` method on line 8. So, when this program runs, it'll display the menu, then prompt the user for input. When the user enters a command and hits return, the program will display the user's command. Let's see this in action.

4.4.3 Test - First Iteration

Testing at this point is still just running the program and eyeballing the results as shown in figure 4-15.



```
Fri Dec 23 09:35:50 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat/sprint_3 (main)
[525:25] swodog@macos-mojave-test-bed $  python3 src/main.py
I am Robot Rat! I am alive!

                Robot Rat Control Menu

        1. Pen Up
        2. Pen Down
        3. Turn Right
        4. Turn Left
        5. Move Forward
        6. Print Floor
        7. Exit

        Enter Menu Command Number: 1
You entered menu choice:  1

Fri Dec 23 09:36:06 EST 2022
~/dev/cst_with_python_1st_ed/chapter04/robot_rat/sprint_3 (main)
[526:26] swodog@macos-mojave-test-bed $
```

Figure 4-15: Testing Menu Command Processing

Referring to figure 4-15 — I just noticed I didn't remove the message from the RobotRatApp constructor. Need to take care of that. OK, the menu displays, a user can make a menu choice and the choice is printed to the console. What if the user enters something other than a 1 through 7? I'll just ignore erroneous input but more on that later. I'm going to move to the second item listed in table 4-9 and work on looping the menu display and menu processing. Time to write more code.

4.4.4 Code – Second Iteration

OK, taking a walk and thinking...the application should display the menu and process the user's menu choice...over and over... until the user exits the application. Since I already have two methods that perform each of those functions, I just need to put them in a loop and call them repeatedly. I'll add another method to the RobotRatApp class to implement this feature. I'll call it start_application(). Example 4.7 gives the code.

4.7 robotrat_app.py (Sprint 3 v2)

```
1    """Implements the Robot Rat Application."""
2
3    class RobotRatApp():
4        """A Remote-Controlled Robot Rat Application."""
5
6        def __init__(self):
7            print('I am Robot Rat! I am alive!')
8
9        def display_menu(self):
10           """Prints menu items to the console."""
11           print('\n\t\tRobot Rat Control Menu\n')
12           print('\t1. Pen Up')
13           print('\t2. Pen Down')
14           print('\t3. Turn Right')
15           print('\t4. Turn Left')
16           print('\t5. Move Forward')
17           print('\t6. Print Floor')
18           print('\t7. Exit')
19
20       def process_menu_choice(self):
21           # Prompt user for input
22           # Assign input string to variable
23           user_input = input('\n\tEnter Command Number: ')
24           # Use first character of input as menu choice
25           menu_choice = user_input[0]
26           if __debug__:
27               print(f'You entered command number:  {menu_choice}')
28           # Is menu_choice valid command?
29           # YES - Execute command
30           # NO - Display error message and try again
31
32       def start_application(self):
33           while True:
34               self.display_menu()
35               self.process_menu_choice()
36
```

Referring to example 4.7 — The `start_application()` definition begins on line 32. A `while` loop continuously calls the `display_menu()` and `process_menu_choice()` methods repeatedly allowing the application continue running between menu commands. The problem now is...well...you'll see the problem here shortly.

I modified the main.py module to call the `start_application()` method as shown in example 4.8.

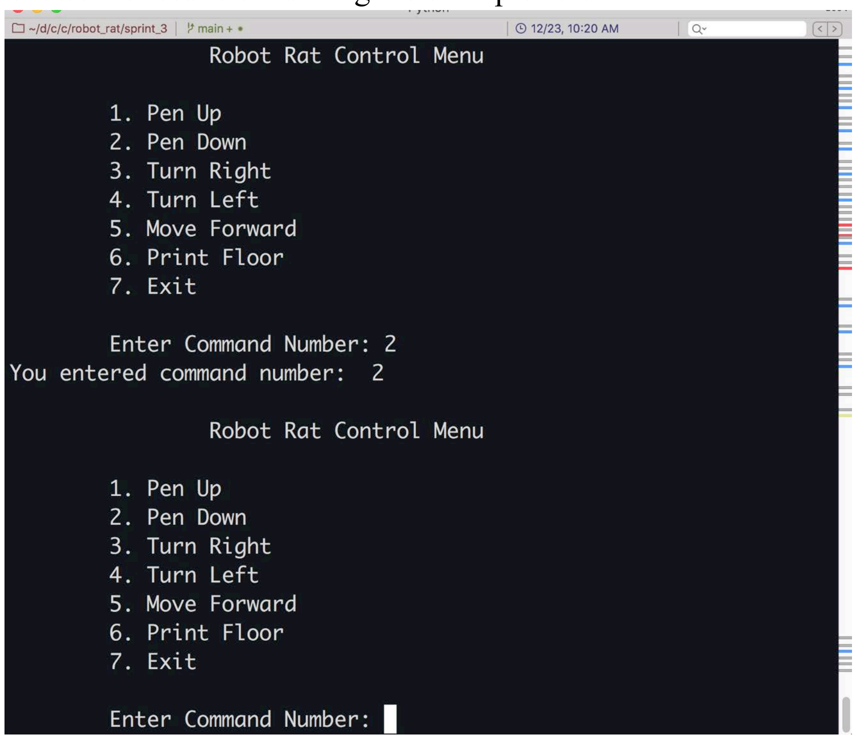
4.8 main.py (Sprint 3 v2)

```
"""Serves as the point of entry to the Robot Rat Application."""

from robotrat_app import RobotRatApp

def main():
    robot_rat_app = RobotRatApp()
    robot_rat_app.start_application()


if __name__ == '__main__':
    main()

```

Referring to example 4.8 — I replaced the calls to `display_menu()` and `process_menu_choice()` with one call to `start_application()`. Let's run the application and see how things look.

4.4.5 Test - Second Iteration

Figure 4-16 shows the results of running the example 4.8.



```
                Robot Rat Control Menu

        1. Pen Up
        2. Pen Down
        3. Turn Right
        4. Turn Left
        5. Move Forward
        6. Print Floor
        7. Exit

        Enter Command Number: 2
You entered command number:  2

                Robot Rat Control Menu

        1. Pen Up
        2. Pen Down
        3. Turn Right
        4. Turn Left
        5. Move Forward
        6. Print Floor
        7. Exit

        Enter Command Number: 
```

Figure 4-16: Menu Display Process Loop Testing

Referring to figure 4-16 — Menu display and input processing is looping fine, but requires a `ctrl-c` to exit, so time to return to the `process_menu_choice()` method and add a graceful way to quit the application.

4.4.6 Code - Third Iteration

I need to continue to develop the `process_menu_choice()` method so that when the user enters a command something useful actually happens. I'm still in the stage of creating application scaffolding, but all this will be completed, for the most part, by the end of this sprint. For now, I want to exit the application when user chooses menu item `'7'` and call method stubs on all the other commands. Example 4.9 give the updated code.

4.9 robotrat_app.py (Sprint 3 v3)

```
"""Implements the Robot Rat Application."""
import sys
```

```python

class RobotRatApp():
    """A Remote-Controlled Robot Rat Application."""

    def __init__(self):
       pass

    def display_menu(self):
        """Prints menu items to the console."""
        print('\n\t\tRobot Rat Control Menu\n')
        print('\t1. Pen Up')
        print('\t2. Pen Down')
        print('\t3. Turn Right')
        print('\t4. Turn Left')
        print('\t5. Move Forward')
        print('\t6. Print Floor')
        print('\t7. Exit')

    def process_menu_choice(self):
        # Prompt user for input
        # Assign input string to variable
        user_input = input('\n\tEnter Command Number: ')
        # Use first character of input as menu choice
        menu_choice = user_input[0]
        if __debug__:
            print(f'You entered command number:  {menu_choice}')
        # Is menu_choice valid command?
        # YES - Execute command
        # NO - Display error message and try again
        match menu_choice:
            case '1': self.set_pen_up()
            case '2': self.set_pen_down()
            case '3': self.turn_right()
            case '4': self.turn_left()
            case '5': self.move_forward()
            case '6': self.print_floor()
            case '7': sys.exit()
            case _: self.print_error_message(menu_choice)


    def start_application(self):
        while True:
            self.display_menu()
            self.process_menu_choice()

    def set_pen_up(self):
        if __debug__:
            print('set_pen_up() method called...')

    def set_pen_down(self):
        if __debug__:
            print('set_pen_down() method called')

    def turn_left(self):
        if __debug__:
            print('turn_left() method called...')

    def turn_right(self):
        if __debug__:
```

```
62              print('turn_right() method called...')
63
64      def move_forward(self):
65          if __debug__:
66              print('move_forward() method called...')
67
68      def print_floor(self):
69          if __debug__:
70              print('print_floor() method called')
71
72      def print_error_message(self, menu_choice):
73          print(f'WARNING: {menu_choice} is an invalid command!')
74
```

Referring to example 4.9 — Notice first that all the method names begin with action verbs: *display*, *process*, *start*, *set*, *turn*, *move*, *print*. To exit the application I called the `sys.exit()` method on line 39. This required me to import the `sys` module on line 2. I use a `match` statement which begins on line 32 to process the `menu_choice` variable. I could have used nested `if/elif` statements here but I find the `match` statement, available since Python 10, cleaner and easier to decipher.

The `match` statement is easy to understand even for novice programmers. The value of the variable `menu_choice` is examined by the `match` statement and if any `case` matches its value the corresponding method is called. For example, if a user enters the `'1'` at the console, the `menu_choice` variable will contain the value `'1'`, which will match the first case and execute the `self.set_pen_up()` method, which is defined on line 48.

The `case _:` represents the *default case*, meaning it will catch anything not handled by any of the previous cases. OK, let's take this for a test drive.

4.4.7 Test - Third Iteration

The *main.py* module remains unchanged at this point so just run it again. Partial results from running the application are shown in figure 4-17

Referring to figure 4-17 — Everything seems to be running fine. I can enter commands and exit the application when I enter `'7'`. If I enter an invalid command I see a warning message. What would be nice is if the application could clear the screen between commands. I'll work on that in a later sprint.

4.4.8 Integrate

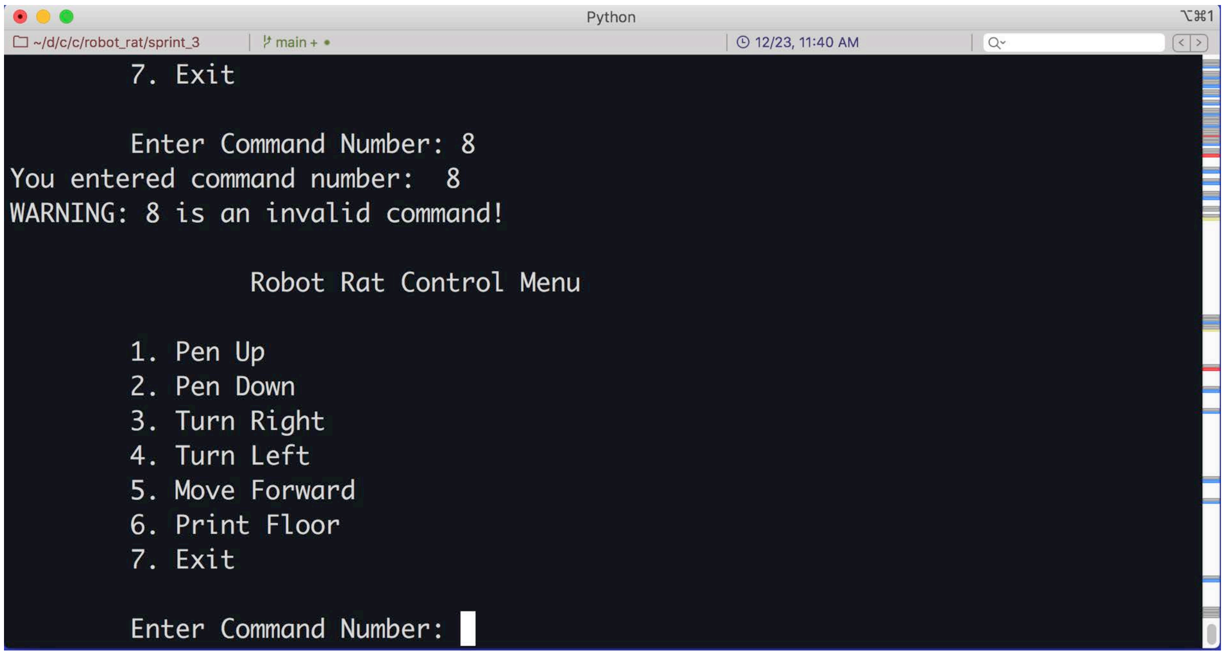
Nothing to integrate, really, but there is something to refactor.

4.4.9 Refactor

You'll find that your first attempt at coding something represents a brute-force approach and results in what I call an Ugly Baby. Another term used widely in the software development industry is Code Smell. And the code that's smelling pretty bad at the moment is the use of string literals in the match statement. Before moving on I'd like to switch those out for constants that represent menu choice values. Example 4.10 lists the refactored code.

4.10 robotrat_app.py (Sprint 3 refactored)

```
1    """Implements the Robot Rat Application."""
2    import sys
3
```



```
        7. Exit

        Enter Command Number: 8
You entered command number:  8
WARNING: 8 is an invalid command!

                Robot Rat Control Menu

        1. Pen Up
        2. Pen Down
        3. Turn Right
        4. Turn Left
        5. Move Forward
        6. Print Floor
        7. Exit

        Enter Command Number: 
```

Figure 4-17: Handling Invalid Commands with Default Match Case

```
class RobotRatApp():
    """A Remote-Controlled Robot Rat Application."""

    # Menu Choice Constants
    _PEN_UP='1'
    _PEN_DOWN='2'
    _TURN_RIGHT='3'
    _TURN_LEFT='4'
    _MOVE_FORWARD='5'
    _PRINT_FLOOR='6'
    _EXIT='7'

    def __init__(self):
       pass

    def display_menu(self):
        """Prints menu items to the console."""
        print('\n\t\tRobot Rat Control Menu\n')
        print('\t1. Pen Up')
        print('\t2. Pen Down')
        print('\t3. Turn Right')
        print('\t4. Turn Left')
        print('\t5. Move Forward')
        print('\t6. Print Floor')
        print('\t7. Exit')

    def process_menu_choice(self):
        # Prompt user for input
        # Assign input string to variable
        user_input = input('\n\tEnter Command Number: ')
        # Use first character of input as menu choice
        menu_choice = user_input[0]
        if __debug__:
            print(f'You entered command number: {menu_choice}')
```

```
38          # Is menu_choice valid command?
39          # YES - Execute command
40          # NO - Display error message and try again
41          match menu_choice:
42              case self._PEN_UP: self.set_pen_up()
43              case self._PEN_DOWN: self.set_pen_down()
44              case self._TURN_RIGHT: self.turn_right()
45              case self._TURN_LEFT: self.turn_left()
46              case self._MOVE_FORWARD: self.move_forward()
47              case self._PRINT_FLOOR: self.print_floor()
48              case self._EXIT: sys.exit()
49              case _: self.print_error_message(menu_choice)
50
51
52      def start_application(self):
53          while True:
54              self.display_menu()
55              self.process_menu_choice()
56
57      def set_pen_up(self):
58          if __debug__:
59              print('set_pen_up() method called...')
60
61      def set_pen_down(self):
62          if __debug__:
63              print('set_pen_down() method called')
64
65      def turn_left(self):
66          if __debug__:
67              print('turn_left() method called...')
68
69      def turn_right(self):
70          if __debug__:
71              print('turn_right() method called...')
72
73      def move_forward(self):
74          if __debug__:
75              print('move_forward() method called...')
76
77      def print_floor(self):
78          if __debug__:
79              print('print_floor() method called')
80
81      def print_error_message(self, menu_choice):
82          print(f'WARNING: {menu_choice} is an invalid command!')
83
```

Referring to example 4.10 — Beginning on line 7, I've defined a set of constants that map to the menu choice number strings. There are no constants in Python, but the PEP 8 style guide offers guidance on how to represent the notion of a constant in a program. I've used *all uppercase characters* for the constant names and separated words within the constant name with an underscore character `'_'`. Notice I have also started the name of each constant with an underscore. This is a signal, or perhaps better described as a developer's agreement, denoting these as being private to the class and that they are not to be considered part of the class's public interface. I then replace the string literals formerly used for each of the `match` cases with the constants. This makes the code a little more self-documenting in that you need not remember that string literal `'1'` corresponds to the Pen Up menu choice, etc.

At this point, I retest the application to ensure I haven't broken anything during the refactoring and move on to the next sprint.

4.5 Development — Sprint 4

The application framework is well laid. What's left now is to add substance to the method stubs. I'll start by implementing the `print_floor()` method, which will require a `floor` to print.

4.5.1 Plan

Table 4-10 lists the design considerations for Sprint 4.

Check	Design Consideration	Design Decision
	Need `floor` instance variable	The `floor` instance variable will be a two-dimensional array of boolean values. The dimensions of the array should be set with constructor arguments.
	Implement the `print_floor()` method	Use a nested `for` loop to iterate over the rows and columns of the floor array. Since the floor array contains boolean values (`True` or `False`), the method will need to visit each element of the array to determine what character to print for `True` or `False` elements.
	Pass rows and cols dimensions to RobotRatApp() constructor	Modify the `__init__()` method to add rows and cols parameters. Use these parameters in the constructor to initialize `floor` array dimensions.

Table 4-10: Sprint 4 Design Considerations

Referring to table 4-10 — Getting the floor to print is a big deal so this is plenty to do for this sprint. The floor instance variable needs to be declared and initialized. The RobotRatApp's `__init__()` method needs to be modified to accept row and column arguments when a RobotRatApp object is created. When the floor is initialized, the `print_floor()` method will step through the rows and columns, determine if an element is `True` or `False` and print a corresponding character to the console. Some adjustments will need to be made to ensure the floor prints legibly. To test the `print_floor()` method, I'll create a temporary utility method that sets a test pattern on the floor so I can see what a Robot Rat pen-down movement will look like. Let's look at the code for this sprint.

4.5.2 Code

Example 4.11 gives the modified robotrat_app.py code for this sprint.

4.11 robotrat_app.py (Sprint 4 v1)

```python
"""Implements the Robot Rat Application."""
import sys

class RobotRatApp():
    """A Remote-Controlled Robot Rat Application."""

    # Menu Choice Constants
    _PEN_UP='1'
    _PEN_DOWN='2'
    _TURN_RIGHT='3'
    _TURN_LEFT='4'
    _MOVE_FORWARD='5'
    _PRINT_FLOOR='6'
    _EXIT='7'

    def __init__(self, rows, cols):
       """Initialize RobotRatApp object."""
       self._rows = rows
       self._cols = cols
       self._floor = [[ False for i in range(cols)] for j in range(rows)]
       self._initialize_test_patern()

    def display_menu(self):
        """Prints menu items to the console."""
        print('\n\t\tRobot Rat Control Menu\n')
        print('\t1. Pen Up')
        print('\t2. Pen Down')
        print('\t3. Turn Right')
        print('\t4. Turn Left')
        print('\t5. Move Forward')
        print('\t6. Print Floor')
        print('\t7. Exit')

    def process_menu_choice(self):
        # Prompt user for input
        # Assign input string to variable
        user_input = input('\n\tEnter Command Number: ')
        # Use first character of input as menu choice
        menu_choice = user_input[0]
        if __debug__:
            print(f'You entered command number: {menu_choice}')
        # Is menu_choice valid command?
        # YES - Execute command
        # NO - Display error message and try again
        match menu_choice:
            case self._PEN_UP: self.set_pen_up()
            case self._PEN_DOWN: self.set_pen_down()
            case self._TURN_RIGHT: self.turn_right()
            case self._TURN_LEFT: self.turn_left()
            case self._MOVE_FORWARD: self.move_forward()
            case self._PRINT_FLOOR: self.print_floor()
            case self._EXIT: sys.exit()
            case _: self.print_error_message(menu_choice)


    def start_application(self):
        while True:
            self.display_menu()
            self.process_menu_choice()

    def set_pen_up(self):
        if __debug__:
            print('set_pen_up() method called...')

    def set_pen_down(self):
```

```
66           if __debug__:
67               print('set_pen_down( ) method called')
68
69       def turn_left(self):
70           if __debug__:
71               print('turn_left( ) method called...')
72
73       def turn_right(self):
74           if __debug__:
75               print('turn_right( ) method called...')
76
77       def move_forward(self):
78           if __debug__:
79               print('move_forward( ) method called...')
80
81       def print_floor(self):
82           if __debug__:
83               print('print_floor( ) method called')
84           for row in self._floor:
85               print('\t', end='')
86               for col in row:
87                   if col:
88                       print('- ', end='')
89                   else:
90                       print('0 ', end='')
91               print( )
92
93       def _initialize_test_patern(self):
94           self._floor[0][0] = True
95           self._floor[0][1] = True
96           self._floor[0][2] = True
97           self._floor[0][3] = True
98           self._floor[0][4] = True
99           self._floor[1][4] = True
100          self._floor[2][4] = True
101          self._floor[3][4] = True
102
103      def print_error_message(self, menu_choice):
104          print(f'WARNING: {menu_choice} is an invalid command!')
105
```

Referring to example 4.11 — Working from top-to-bottom, notice first that the `__init__( )` method on line 16 has been modified. I added two parameters, `rows` and `cols`, which I use in the body of the method to initialize two instance fields, `self._rows` and `self._cols`, and to initialize the `self._floor` instance variable using a *list comprehension*. After I've initialized the floor, I call the `self._initialize_test_pattern( )` method, which is defined on line 93, to set a test pattern on the floor. The term "setting a test pattern" involves nothing more than setting a handful of individual array elements to `True` (They are all initialized to `False` in the constructor.) so when I print the floor I can see something besides zeros. This method is only needed for testing and I'll delete it in the final version of the program.

With the floor created and initialized, I can then implement the `print_floor( )` method, which begins on line 81. Essentially, the two-dimensional array is processed via a set of nested `for` loops. The *outer loop* iterates over the array rows while the *inner loop* iterates over the array columns. You could say it in plain English or pseudocode like so:

```
for each row in floor
    print a tab character to push the start of each row a little to the right
    visit each column in each row
        check whether the element is true or false
            if the element is true print the '-' character
            if the element is false print the '0' character
    print a new line to start a new row
```

Now that the RobotRatApp constructor has been modified, the main.py module will need to be tweaked as well. Example 4.12 give the code for the modified main.py module.

4.12 main.py (Sprint 4 v1)

```
1    """Serves as the point of entry to the Robot Rat Application."""
2
3    from robotrat_app import RobotRatApp
4
5    def main():
6        robot_rat_app = RobotRatApp(20, 20)
7        robot_rat_app.start_application()
8
9
10   if __name__ == '__main__':
11       main()
12
```

Referring to example 4.12 — The only changes required appear on line 6 in the `RobotRatApp()` constructor call. I've added two integer arguments that represent row and column dimensions. Let's test the code.

4.5.3 Test

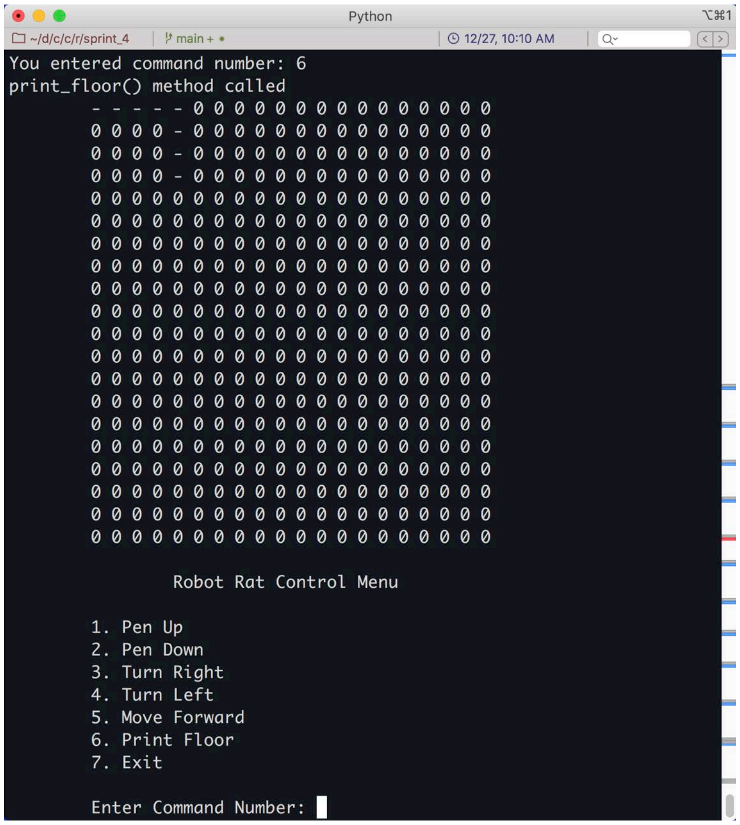
Figure 4-18 gives the results of running the program.

Referring to figure 4-18 — Everything looks good! You can experiment with the code and use different characters to print the floor. This marks the end of this sprint. I'll push my changes to my remote repository before starting the next sprint.

4.6 Development — Sprint 5

I'll aim high during this sprint and implement the remaining Robot Rat commands including the Move Forward command. Along the way I'll introduce you to State Transition Diagrams. You'll find these helpful to work out object or entity state changes and the transitions that force a change from one state to another.

The concept of state, as it relates to a Robot Rat, is simply the collection of attributes that describe where upon the floor the Robot Rat is at any given moment, the direction it's facing, and the position of its pen. I'll need to create instance variables to hold those values. I'll also need to create new data types to represent the concepts of Direction and Pen Position.



```
You entered command number: 6
print_floor() method called
        - - - - - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

                Robot Rat Control Menu

        1. Pen Up
        2. Pen Down
        3. Turn Right
        4. Turn Left
        5. Move Forward
        6. Print Floor
        7. Exit

        Enter Command Number:
```

Figure 4-18: Printing the Floor with Test Pattern

4.6.1 Plan

Table 4-11 lists the design considerations and decisions for sprint 5.

Check	Design Consideration	Design Decision
	Implement Pen Up and Pen Down Commands	Create an enumerated type to represent the concept of a Pen Position. Create a state transition diagram to illustrate how to change from the Pen Up state to the Pen Down state. I'll also need an instance variable to store a Robot Rat's `pen_position`.
	Implement the Turn Right and Turn Left commands	Create an enumerated type to represent the concept of Direction. The Robot Rat can face in one of four directions: NORTH, SOUTH, EAST, or WEST. Create a state transition diagram to illustrate how to change from one direction to another during a Turn Right or Turn Left command. I'll also need an instance variable to store a Robot Rat's `direction`.
	Implement the Move Forward command.	The concept of a Robot Rat's Current Position, which represents a state, must be represented with instance variables. Referring to figures 4-3 and 4-4, I'll use the variables `current_row` and `current_column`.

Table 4-11: Sprint 5 Design Considerations

Referring to table 4-11 — Completing the items listed will take the project to the dang-near-done point. The Move Forward command represents some of the more involved coding of the project. But let's start with the first item on the list and work the Pen Position problem.

4.6.1.1 State Transition Diagrams

A state transition diagram is a graphical representation of object or entity *states* drawn as a directed graph consisting of vertices (circles) representing possible entity states, and transitions between states, *edges* (arrows).

The state of an object is expressed by the collective states of its attributes. (Refer to figure 4-2 to see the set of attributes that comprise a Robot Rat.) Figure 4-19 shows the state transition diagram for a Robot Rat's pen position.

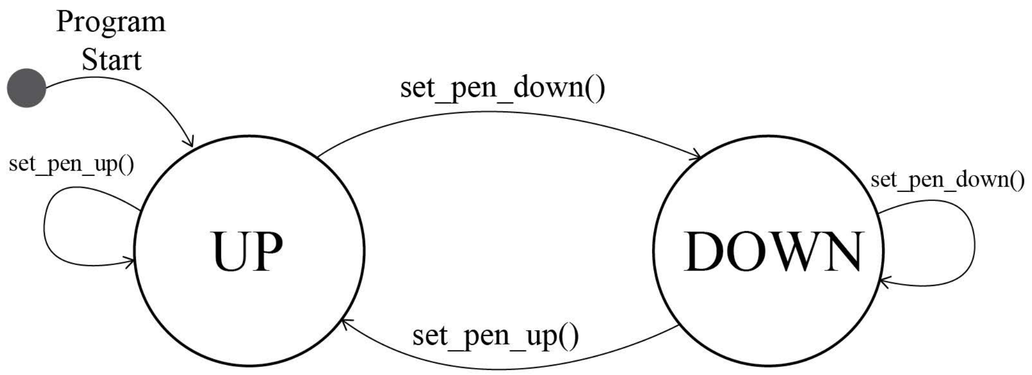


Figure 4-19: Pen Position State Transition Diagram

Referring to figure 4-19 — A Robot Rat's pen position has two possible states: UP and DOWN. When the program starts its pen will be set to the UP position. At some point during program execution, a user may set the pen to the DOWN position to start drawing a pattern on the floor. The state transition diagram shows the states (vertices) and how one state can transition to another state (edges). If the pen is in the UP position, a call to the set_pen_down() method will change its state from UP to DOWN. If the set_pen_down() method is called again, the pen remains in the down position.

Figure 4-20 shows the state transition diagram for the Robot Rat's direction.

Referring to figure 4-20 — This diagram is a bit more involved. It represents a Robot Rat's direction attribute, which, according to the project specification, can be in one of four possible states: NORTH, SOUTH, EAST, or WEST. The state transition diagram shows that when the program starts the Robot Rat's direction is set to EAST. A call to the `turn_right()` method will change direction state to SOUTH. You can see from the diagram how successive calls to the `turn_right()` and `turn_left()` methods changes a Robot Rat's direction.

4.6.1.2 Implementing Move Forward

There are a few different ways I can implement the move forward command. Moving the Robot Rat about the floor differs depending on whether the pen is UP or DOWN. If the pen is UP then moving is simply a matter of updating a Robot Rat's position (current row and current column) to a new location based on how far the rat moves in a particular direction. If the pen is

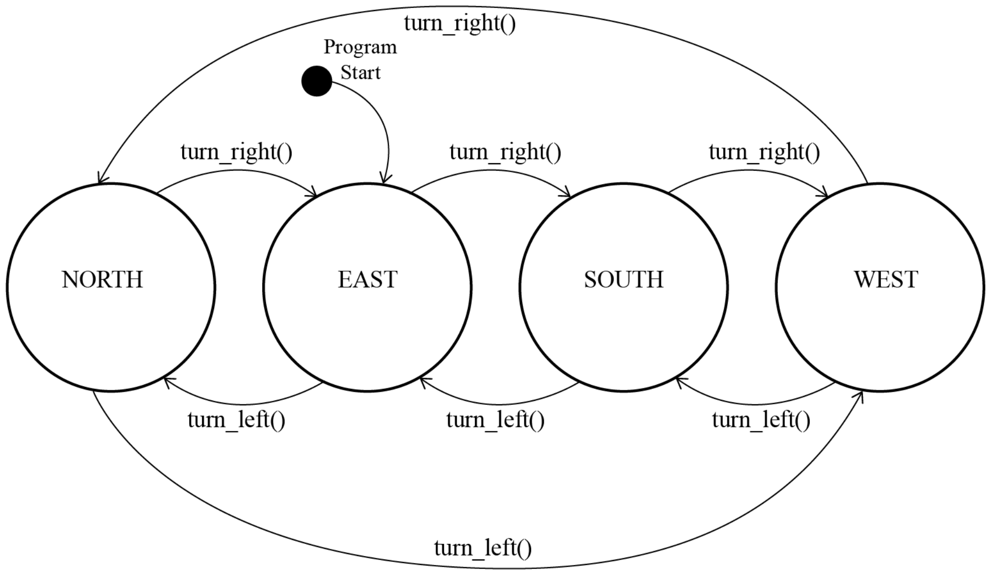


Figure 4-20: Direction State Transition Diagram

DOWN then the floor array must be manipulated to indicate where the Robot Rat traversed so a pattern can be drawn. When the pen is down, each visited array element must be set to `True`.

Upon further analysis, another set of questions regarding movement emerge:

- How far can a Robot Rat move?
- Should it stop at the floor's edges or be allowed to continue outside the visible bounds of the floor, as if there were a virtual floor extending infinitely in all directions?
- Should the movement wrap? Say, for example, when the Robot Rat moves past the floor's edge, should it reemerge on the opposite side on the same column or row or even perhaps on a different column or row?

I've seen students implement all kinds of movement scenarios. I will take the finite edges approach and stop the Robot Rat dead in its tracks if it tries to move beyond the visible bounds of the floor. OK, let's see some code.

4.6.2 Code

Example 4.13 gives the code for the modified RobotRatApp class.

4.13 robotrat_app,py (Sprint 5 v1)

```
"""Implements the Robot Rat Application."""
import sys
from enum import Enum


class RobotRatApp():
    """A Remote-Controlled Robot Rat Application."""

    # Menu Choice Constants
    _PEN_UP = '1'
    _PEN_DOWN = '2'
    _TURN_RIGHT = '3'
    _TURN_LEFT = '4'
    _MOVE_FORWARD = '5'
    _PRINT_FLOOR = '6'
    _EXIT = '7'

    # Enumerated Types
    class PenPositions(Enum):
        UP = 0
        DOWN = 1

    class Directions(Enum):
        NORTH = 0
        EAST = 1
        SOUTH = 2
        WEST = 3

    def __init__(self, rows, cols):
        """Initialize RobotRatApp object."""
        self._rows = rows
        self._cols = cols
        self._floor = [[False for i in range(cols)] for j in range(rows)]
        self._initialize_test_patern()
        self._pen_position = self.PenPositions.UP
        self._direction = self.Directions.EAST
        self._current_row = 0
        self._current_col = 0

    def display_menu(self):
        """Prints menu items to the console."""
        print('\n\t\tRobot Rat Control Menu\n')
        print('\t1. Pen Up')
        print('\t2. Pen Down')
        print('\t3. Turn Right')
        print('\t4. Turn Left')
        print('\t5. Move Forward')
        print('\t6. Print Floor')
        print('\t7. Exit')

    def process_menu_choice(self):
        # Prompt user for input
        # Assign input string to variable
        user_input = input('\n\tEnter Command Number: ')
        # Use first character of input as menu choice
        menu_choice = user_input[0]
        if __debug__:
            print(f'You entered command number: {menu_choice}')
        # Is menu_choice valid command?
        # YES - Execute command
        # NO - Display error message and try again
        match menu_choice:
            case self._PEN_UP: self.set_pen_up()
            case self._PEN_DOWN: self.set_pen_down()
            case self._TURN_RIGHT: self.turn_right()
            case self._TURN_LEFT: self.turn_left()
            case self._MOVE_FORWARD: self.move_forward()
            case self._PRINT_FLOOR: self.print_floor()
            case self._EXIT: sys.exit()
            case _: self.print_error_message(menu_choice)

    def start_application(self):
        while True:
            self.display_menu()
            self.process_menu_choice()

    def set_pen_up(self):
        if __debug__:
            print('set_pen_up() method called...')
        self._pen_position = self.PenPositions.UP
        print(f'Pen is {self._pen_position}')

    def set_pen_down(self):
        if __debug__:
            print('set_pen_down() method called')
        self._pen_position = self.PenPositions.DOWN
        print(f'Pen is {self._pen_position}')

    def turn_left(self):
        if __debug__:
            print('turn_left() method called...')
        match(self._direction):
            case self.Directions.NORTH:
                self._direction = self.Directions.WEST
            case self.Directions.WEST:
                self._direction = self.Directions.SOUTH
            case self.Directions.SOUTH:
                self._direction = self.Directions.EAST
            case self.Directions.EAST:
                self._direction = self.Directions.NORTH
            case _: self._direction = self.Directions.EAST

        print(f'Robot Rat is facing {self._direction}')

    def turn_right(self):
        if __debug__:
            print('turn_right() method called...')
        match(self._direction):
            case self.Directions.NORTH:
                self._direction = self.Directions.EAST
            case self.Directions.EAST:
                self._direction = self.Directions.SOUTH
            case self.Directions.SOUTH:
                self._direction = self.Directions.WEST
            case self.Directions.WEST:
                self._direction = self.Directions.NORTH
            case _: self._direction = self.Directions.EAST

        print(f'Robot Rat is facing {self._direction}')

    def move_forward(self):
        if __debug__:
            print('move_forward() method called...')
        spaces_to_move = 0
        try:
            spaces_to_move = int(input("Enter spaces to move: "))
        except Exception as e:
            print('Invalid movment number. Setting to 1')
            spaces_to_move = 1

        match(self._pen_position):
```

```python
            case self.PenPositions.UP:
                match(self._direction):
                    case self.Directions.NORTH:
                        self._current_row -= spaces_to_move
                        if self._current_row < 0:
                            self._current_row = 0
                    case self.Directions.EAST:
                        self._current_col += spaces_to_move
                        if self._current_col > (self._cols - 1):
                            self._current_col = (self._cols - 1)
                    case self.Directions.SOUTH:
                        self._current_row += spaces_to_move
                        if self._current_row > (self._rows - 1):
                            self._current_row = (self._rows - 1)
                    case self.Directions.WEST:
                        self._current_col -= spaces_to_move
                        if self._current_col < 0:
                            self._current_col = 0
            case self.PenPositions.DOWN:
                match(self._direction):
                    case self.Directions.NORTH:
                        while (self._current_row > -1) and (spaces_to_move > 0):
                            self._floor[self._current_row][self._current_col] = True
                            if self._current_row > 0:
                                self._current_row -= 1
                            else:
                                break
                            spaces_to_move -= 1
                    case self.Directions.EAST:
                        while (self._current_col < self._cols) and (spaces_to_move > 0):

                            self._floor[self._current_row][self._current_col] = True
                            if self._current_col < (self._cols - 1):
                                self._current_col += 1
                            else:
                                break
                            spaces_to_move -= 1
                    case self.Directions.SOUTH:
                        while (self._current_row < self._rows) and (spaces_to_move > 0):

                            self._floor[self._current_row][self._current_col] = True
                            if self._current_row < (self._rows - 1):
                                self._current_row += 1
                            else:
                                break
                            spaces_to_move -= 1
                    case self.Directions.WEST:
                        while (self._current_col > -1) and (spaces_to_move > 0):
                            self._floor[self._current_row][self._current_col] = True
                            if self._current_col > 0:
                                self._current_col -= 1
                            else:
                                break
                            spaces_to_move -= 1

        print(f'Robot Rat at [{self._current_row}][{self._current_col}] \
facing {self._direction} {self._pen_position}')

    def print_floor(self):
```

```
189         if __debug__:
190             print('print_floor() method called')
191         for row in self._floor:
192             print('\t', end='')
193             for col in row:
194                 if col:
195                     print('- ', end='')
196                 else:
197                     print('0 ', end='')
198             print()
199
200     def _initialize_test_patern(self):
201         self._floor[0][0] = True
202         self._floor[0][1] = True
203         self._floor[0][2] = True
204         self._floor[0][3] = True
205         self._floor[0][4] = True
206         self._floor[1][4] = True
207         self._floor[2][4] = True
208         self._floor[3][4] = True
209
210     def print_error_message(self, menu_choice):
211         print(f'WARNING: {menu_choice} is an invalid command!')
212
```

Referring to example 4.13 — Starting from the top, on line 3, I import the `Enum` class from the `enum` module. I'm using enumerated types to represent the pen position and direction states. In Python, an enumerated type is a class that extends the `Enum` class. I have defined these as inner classes within the RobotRatApp class.

The `PenPositions` enumerated type is defined on line 19. It defines two possible values: `UP`, which is assigned the value 0, and `DOWN`, which is assigned the value 1. Note that it doesn't really matter what values are assigned, as long as they are unique. The `Directions` enumerated type is defined on line 23. It defines four possible values: NORTH, EAST, SOUTH, and WEST. You can see the assigned integer values by reading the code. These two new types are used to initialize the Robot Rat instance variable `_pen_position` and `_direction` as shown on lines 35 and 36.

On lines 37 and 38, I added two new instance variables `_current_row` and `_current_col` and initialize each to `0`. This completes the set of Robot Rat attributes.

Moving down to line 80 in the body of the `set_pen_up()` method, I set the `_pen_position` variable to `PenPositions.UP`. Following that, on line 81, I print the current state of the Robot Rat's pen position. The `set_pen_down()` method works in similar fashion. I leave it to you as an exercise to verify this code satisfies the Pen Position State Transition Diagram shown earlier in figure 4-19

The `turn_left()` method begins on line 89. I use a `match` statement to check the current state of the `_direction` variable. The default case sets the Robot Rat's direction to EAST. You can verify for yourself the code in both the `turn_left()` and `turn_right()` methods satisfy the Direction State Transition Diagram shown in figure 4-20.

The `move_forward()` method that starts on line 121 required the most work during this sprint. Lines 124 through 129 get the `spaces_to_move` from the user. The outermost match statement, which begins on line 133, checks the state of the `_pen_position` variable. Floor movement is processed according to whether the pen is `UP` or `DOWN`. Inner `match` statements check the state of the `_direction` variable and proceed according to which direction the Robot Rat is facing. You can step through the code to see how movement is processed and how each element of the `_floor` array is set when the pen is down.

4.6.2.0.1 A Word On Implementation

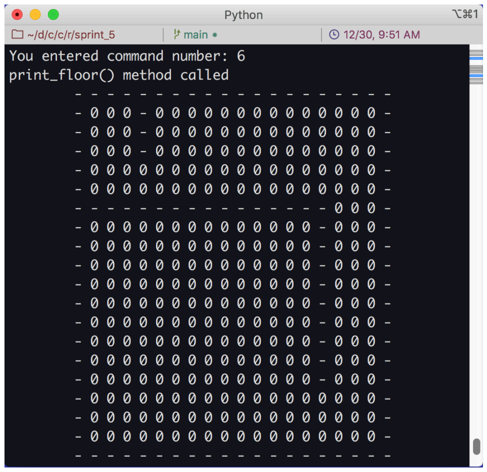
Like Sprint 3, this sprint required multiple sub-iterations of *plan*, *code*, and *test.* I started by researching Python enumerated types. I then had to decide on whether to declare the enumeration classes `PenPositions` and `Directions` inside the `RobotRatApp` class or in a separate module. I opted to make them inner classes because that just seemed like the right thing to do. Sometimes a design decision comes down to gut instinct. Only time will tell if such decisions ultimately prove to be good or bad.

Once I had the enumerated types defined, I added the `_direction` and `_pen_position` instance variables and initialized them in the `__init__()` method using the enumerated types. I then completed the `set_pen_up()`, `set_pen_down()`, `turn_left()`, and `turn_right()` methods. Once they were done, I turned my attention to the `move_forward()` method.

I implemented the move_forward() method in sub-iterations as well, starting first with movement when the pen is `UP`. Once I had that working, I implemented movement when the pen was `DOWN`, which represents the most challenging section of the code. When I completed the `move_forward()` method, or rather, when I *thought* I had completed the method, it was time to give everything a good, thorough shakedown.

4.6.3 Test

Testing the code in this sprint is quite involved. I need to validate menu commands 1 through 6 and move the Robot Rat all around the floor with the pen in the UP and DOWN positions. I need to try to break the program by trying to move outside the bounds of the floor. Figure 4-21 shows the floor pattern after an extended testing session.



```
You entered command number: 6
print_floor() method called
       - - - - - - - - - - - - - - - - - - - -
       - 0 0 0 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
       - 0 0 0 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
       - 0 0 0 - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
       - - - - - - - - - - - - - - - - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 - 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
       - 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
       - - - - - - - - - - - - - - - - - - - -
```

Figure 4-21: Testing Sprint 5 Features

Referring to figure 4-21 — Everything seems to run fine. There are a handful of issues I need to fix. During the course of development I've accumulated some technical debt, which is a term used to refer to known issues with the code that need to be addressed but for one reason or another have been ignored. I'll address technical debt in the next sprint.

4.6.4 Integrate

Integration has taken place organically. By this I mean I've added new features (code) to existing code. For larger more complex projects, integration of new or separately-developed components would be more involved.

4.6.5 Sprint Retrospective

I added a lot of meaningful functionality to the Robot Rat application. The analysis performed earlier on Robot Rat attributes and movement helped a lot. I did notice that I've omitted several docstrings here and there, and I can remove the test pattern code as it's no longer necessary. Time to move on to the next and final sprint of the chapter.

4.7 Development — Sprint 6

This is the final Robot Rat application development sprint. While there are a lot of cool features I could add to Robot Rat, like diagonal directions or randomized movement, the project as it stands satisfies the project specification so instead I'll focus my efforts on reducing accumulated technical debt. The technical debt as I see it includes removing the test pattern code, reorganizing the code for better aesthetics, adding missing docstrings, and adding a project header to the top of each file. If you are a student you'll want to add a project header to identify ownership of your source files when you submit them to your instructor.

Something else bugs me about the application. I'd like to see where the Robot Rat is on the floor when I print the floor pattern. I think I'll rework the `print_floor()` method to display the Robot Rat's location on the floor.

4.7.1 Plan

Table 4-12 lists the project's technical debt and remediation actions.

Check	Technical Debt	Remediation Action
	Add Project Header	Expand the module docstring to add Date, Project, Student, and Class fields. Add this header to both project files robotrat_app.py and main.py
	Add Missing Docstrings	Add docstrings to all class methods.
	Mark Robot Rat's Floor Position	Rework the `print_floor()` method to print a symbol on the Robot Rat's current floor position.
	Reorganize Code	Reorganize code to put method definitions in the order they appear in the command menu. Place the `start_application()` method at the bottom.

Table 4-12: Sprint 6 Technical Debt and Remediation Actions

Referring to table 4-12 — Completing these activities should put the project in good form. I may find something else to improve as I dig into the code.

4.7.2 Code

Example 4.14 gives the listing of the cleaned up robotrat_app.py module.

4.14 robotrat_app.py (Sprint 6)

```python
"""Implements the Robot Rat Application.

Date: 31 December 2022
Project: Robot Rat
Student: Rick Miller
Class: IT-566: Computer Scripting Techniques
"""

import sys
from enum import Enum

class RobotRatApp( ):
   """A Remote-Controlled Robot Rat Application."""

   # Menu Choice Constants
   _PEN_UP = '1'
   _PEN_DOWN = '2'
   _TURN_RIGHT = '3'
   _TURN_LEFT = '4'
   _MOVE_FORWARD = '5'
   _PRINT_FLOOR = '6'
   _EXIT = '7'

   # Enumerated Types
   class PenPositions(Enum):
      """Valid Pen Position States"""
      UP = 0
      DOWN = 1


   class Directions(Enum):
      """Valid Directions"""
      NORTH = 0
      EAST = 1
      SOUTH = 2
      WEST = 3


   def __init__(self, rows, cols):
      """Initialize RobotRatApp object."""
      self._rows = rows
      self._cols = cols
      self._floor = [[False for i in range(cols)] for j in range(rows)]
      self._pen_position = self.PenPositions.UP
      self._direction = self.Directions.EAST
      self._current_row = 0
      self._current_col = 0


   def display_menu(self):
      """Prints menu items to the console."""
      print('\n\t\tRobot Rat Control Menu\n')
      print('\t1. Pen Up')
      print('\t2. Pen Down')
```

```python
        print('\t3. Turn Right')
        print('\t4. Turn Left')
        print('\t5. Move Forward')
        print('\t6. Print Floor')
        print('\t7. Exit')


    def process_menu_choice(self):
        """Process menu commands."""
        # Prompt user for input
        # Assign input string to variable
        user_input = input('\n\tEnter Command Number: ')
        # Use first character of input as menu choice
        menu_choice = user_input[0]
        if __debug__:
            print(f'You entered command number: {menu_choice}')
        # Is menu_choice valid command?
        # YES - Execute command
        # NO - Display error message and try again
        match menu_choice:
            case self._PEN_UP: self.set_pen_up()
            case self._PEN_DOWN: self.set_pen_down()
            case self._TURN_RIGHT: self.turn_right()
            case self._TURN_LEFT: self.turn_left()
            case self._MOVE_FORWARD: self.move_forward()
            case self._PRINT_FLOOR: self.print_floor()
            case self._EXIT: sys.exit()
            case _: self.print_error_message(menu_choice)


    def set_pen_up(self):
        """Changes pen state to UP."""
        if __debug__:
            print('set_pen_up() method called...')
        self._pen_position = self.PenPositions.UP
        print(f'Pen is {self._pen_position}')


    def set_pen_down(self):
        """Changes pen state to DOWN."""
        if __debug__:
            print('set_pen_down() method called')
        self._pen_position = self.PenPositions.DOWN
        print(f'Pen is {self._pen_position}')


    def turn_right(self):
        """Turns Robot Rat to the left."""
        if __debug__:
            print('turn_right() method called...')
        match(self._direction):
            case self.Directions.NORTH:
                self._direction = self.Directions.EAST
            case self.Directions.EAST:
                self._direction = self.Directions.SOUTH
            case self.Directions.SOUTH:
                self._direction = self.Directions.WEST
            case self.Directions.WEST:
                self._direction = self.Directions.NORTH
            case _: self._direction = self.Directions.EAST


    def turn_left(self):
        """Turns Robot Rat to the right."""
        if __debug__:
            print('turn_left() method called...')
        match(self._direction):
            case self.Directions.NORTH:
                self._direction = self.Directions.WEST
            case self.Directions.WEST:
                self._direction = self.Directions.SOUTH
            case self.Directions.SOUTH:
                self._direction = self.Directions.EAST
            case self.Directions.EAST:
                self._direction = self.Directions.NORTH
            case _: self._direction = self.Directions.EAST


    def move_forward(self):
        """ Moves Robot Rat forward by indicated number of spaces.
            If the pen is UP the Robot Rat does not leave a mark on
            the floor. If the pen is DOWN the Robot Rat leaves a mark
            on the floor.
        """
        if __debug__:
            print('move_forward() method called...')
        spaces_to_move = 0
        try:
            spaces_to_move = int(input("Enter spaces to move: "))
        except Exception as e:
            print('Invalid movment number. Setting to 1')
            spaces_to_move = 1

        match(self._pen_position):
            case self.PenPositions.UP:
                match(self._direction):
                    case self.Directions.NORTH:
                        self._current_row -= spaces_to_move
                        if self._current_row < 0:
                            self._current_row = 0
                    case self.Directions.EAST:
                        self._current_col += spaces_to_move
                        if self._current_col > (self._cols - 1):
                            self._current_col = (self._cols - 1)
                    case self.Directions.SOUTH:
                        self._current_row += spaces_to_move
                        if self._current_row > (self._rows - 1):
                            self._current_row = (self._rows - 1)
                    case self.Directions.WEST:
                        self._current_col -= spaces_to_move
                        if self._current_col < 0:
                            self._current_col = 0
            case self.PenPositions.DOWN:
                match(self._direction):
                    case self.Directions.NORTH:
                        while (self._current_row > -1) and (spaces_to_move > 0):
                            self._floor[self._current_row][self._current_col] = True
                            if self._current_row > 0:
                                self._current_row -= 1
                            else:
                                break
                            spaces_to_move -= 1
                    case self.Directions.EAST:
                        while (self._current_col < self._cols) and (spaces_to_move >
0):
                            self._floor[self._current_row][self._current_col] = True
                            if self._current_col < (self._cols - 1):
                                self._current_col += 1
                            else:
                                break
                            spaces_to_move -= 1
                    case self.Directions.SOUTH:
                        while (self._current_row < self._rows) and (spaces_to_move >
0):
                            self._floor[self._current_row][self._current_col] = True
                            if self._current_row < (self._rows - 1):
                                self._current_row += 1
                            else:
                                break
                            spaces_to_move -= 1
                    case self.Directions.WEST:
                        while (self._current_col > -1) and (spaces_to_move > 0):
                            self._floor[self._current_row][self._current_col] = True
                            if self._current_col > 0:
                                self._current_col -= 1
                            else:
                                break
                            spaces_to_move -= 1


    def print_floor(self):
        """ Prints the floor pattern."""
        if __debug__:
            print('print_floor() method called')
        for rindx, row in enumerate(self._floor):
            print('\t', end='')
            for cindx, col in enumerate(row):
                if((rindx == self._current_row) and (cindx == self._current_col)):
                    print('X ', end='')
                elif col:
                    print('- ', end='')
                else:
                    print('0 ', end='')

            print()


    def print_status(self):
        """Displays Robot Rat current position, direction, and pen position."""
        print(f'\n\tRobot Rat at [{self._current_row}][{self._current_col}] \
facing {self._direction} {self._pen_position}')


    def print_error_message(self, menu_choice):
        """Warns of an invalid command entry."""
        print(f'WARNING: {menu_choice} is an invalid command!')

```

```
230
231     def start_application(self):
232         """Start application processing loop."""
233         while True:
234             self.display_menu()
235             self.print_status()
236             self.process_menu_choice()
237
```

Referring to example 4.14 — You can see the header information at the top of the file provides clarifying information about the purpose of the module and who wrote it. If you're a student most professors require similar identifying information be included at the top of each source file. If you're a practicing professional, your header information may be set by engineering team policy.

I put most of the work into the revised `print_floor()` method. Instead of using implied iterations, I'm now using the `enumerate()` method to extract both the index and the value from the iterable object. The `enumerate()` method returns a two-item tuple with the first item being the index (an integer) and the second item being the value. The outer `for` loop begins on line 207 and processes the rows. So, for each row, the `enumerate()` method extracts the row index, and the list of columns associated with that row. These tuple values are unpacked into the variables `rindx` and `row`. The inner `for` loop which begins on line 209 processes the columns. The `enumerate()` method again returns a two-item tuple consisting of the item index and value. The tuple is unpacked into the variables `cindx` and `col`. Now that I have the row index and column index, I can compare those values with the Robot Rat's `_current_row` and `_current_col` attributes and if they match then draw an `'X'` on that spot.

While I was in the code I did some refactoring. I consolidated several status report messages, like when a user entered a Turn Right, Turn Left, or Pen Up and Pen Down command. I removed those from the code and created the `print_status()` method which begins on line 220. I then added a call to the `print_status()` method in the body of the `while` loop on line 235 in the `start_application()` method.

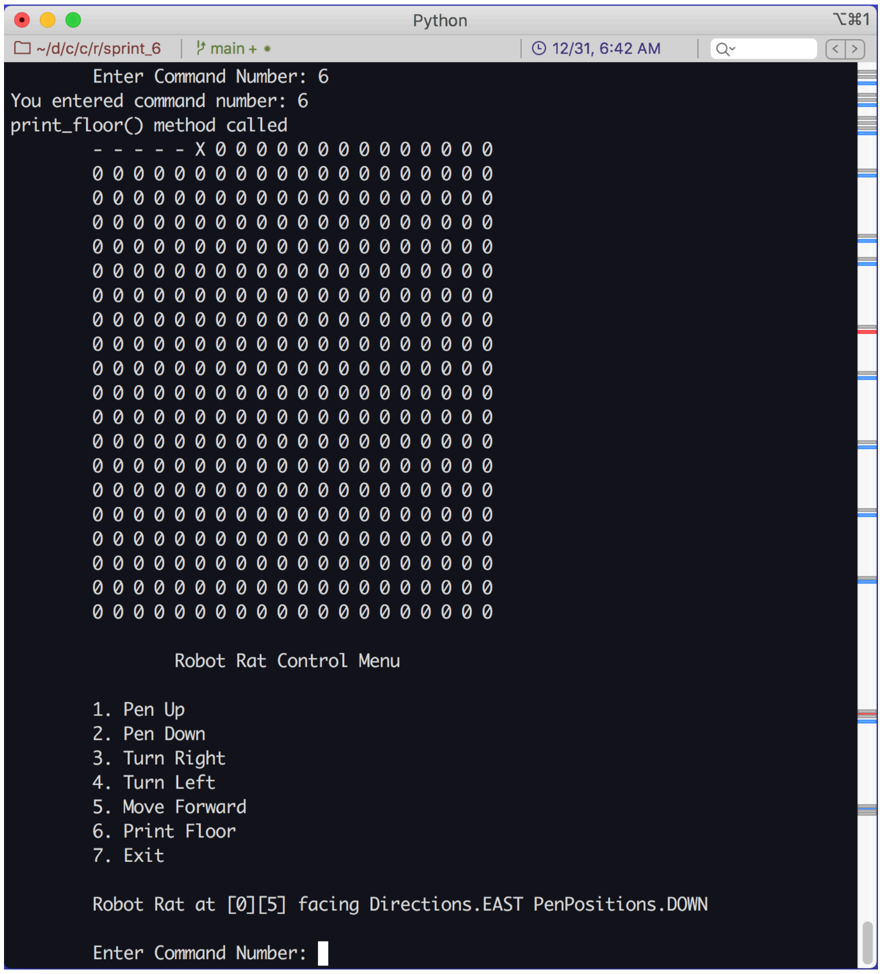
Finally, I used *pylint* to check the code for PEP 8 code format violations. In Visual Studio Code open the command palette and select your Python linter of choice, install it, and enable Python linting.

4.7.3 Test

Figure 4-22 shows the results of running the improved program. Referring to figure 4-22 — This looks much cleaner and better organized. Plus it's nice to see where the Robot Rat is on the floor.

5 Final Considerations

At the end of Sprint 6 I have a program that satisfies and in some areas exceeds the Robot Rat project specification, but this is a very simple application. If the project specification said you needed to control multiple robot rats around a floor, then that would have demanded a different architecture. Piling it on..., if the project required different types of remote controlled objects, that would again require a completely different architecture. You'll learn more about these advanced application architectural concepts as you progress through the book.



```
        Enter Command Number: 6
You entered command number: 6
print_floor() method called
        - - - - - X 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

                Robot Rat Control Menu

        1. Pen Up
        2. Pen Down
        3. Turn Right
        4. Turn Left
        5. Move Forward
        6. Print Floor
        7. Exit

        Robot Rat at [0][5] facing Directions.EAST PenPositions.DOWN

        Enter Command Number:
```

Figure 4-22: Robot Rat Location Marked with an X

5.1 Clearing The Terminal Screen

As it stands now, the Robot Rat application runs perfectly fine on all three operating systems: Linux, macOS, and Microsoft Windows. Python does not have a built-in clear screen function and so developers are left to their own devices to create one. This requires making a call to an available shell command which essentially boils down to two different commands depending on whether a program is running in a Windows (`cls`) or Unix/Linux (`clear`) terminal.

Table 4-13 lists operating systems, terminal types, clear screen commands, and their effects.

OS (name)	Terminal Type	Command	Effect
Windows (nt)	Git Bash	`clear`	Clears screen and screen buffer.
	PowerShell	`clear, cls`	Clears screen and screen buffer.
	Command Prompt	`cls`	Clears screen and screen buffer.

Table 4-13: Effects of Clear Screen Commands on OS Terminal Types

OS (name)	Terminal Type	Command	Effect
macOS (posix)	iTerm	`clear`	Clears screen only. Leaves buffer intact.
	Terminal	`clear`	Clears screen only. Leaves buffer intact.
Linux (posix)	Terminator	`clear`	Clears screen and screen buffer.
	Terminal	`clear`	Clears screen and screen buffer

Table 4-13: Effects of Clear Screen Commands on OS Terminal Types (Continued)

Referring to table 4-13 — You can see the effects of clearing the terminal screen differ between operating systems. Clearing the terminal screen on Windows and Linux results in everything being cleared. You're left with the command prompt. On macOS, clearing the screen clears the visible terminal window but leaves the screen buffer intact. You can scroll up to see the terminal history. The effects of the different commands means little for our purposes. What does matter is that there are different commands between Windows and Linux/macOS.

The takeaway from table 4-13 is that if you want to clear the screen via Python code, you'll need to make a call to either `'cls'` or `'clear'`. If you want your program to run error-free and cross-platform in all the terminals listed above, then you'll need to determine in which operating system the program is running and call the corresponding command. Fortunately, Python makes this easy to do. Example 4.15 give the code for the modified RobotRatApp class.

4.15 robotrat_app.py (clear_screen() method)

```
"""Implements the Robot Rat Application.

Date: 31 December 2022
Project: Robot Rat
Student: Rick Miller
Class: IT-566: Computer Scripting Techniques
"""

import os
import sys
from enum import Enum

class RobotRatApp():
    """A Remote-Controlled Robot Rat Application."""

    # Menu Choice Constants
    _PEN_UP = '1'
    _PEN_DOWN = '2'
    _TURN_RIGHT = '3'
    _TURN_LEFT = '4'
    _MOVE_FORWARD = '5'
    _PRINT_FLOOR = '6'
    _EXIT = '7'

    # Enumerated Types
    class PenPositions(Enum):
        """Valid Pen Position States"""
        UP = 0
```

```python
        DOWN = 1


    class Directions(Enum):
        """Valid Directions"""
        NORTH = 0
        EAST = 1
        SOUTH = 2
        WEST = 3


    def __init__(self, rows, cols):
        """Initialize RobotRatApp object."""
        self._rows = rows
        self._cols = cols
        self._floor = [[False for i in range(cols)] for j in range(rows)]
        self._pen_position = self.PenPositions.UP
        self._direction = self.Directions.EAST
        self._current_row = 0
        self._current_col = 0


    def display_menu(self):
        """Prints menu items to the console."""
        print('\n\t\tRobot Rat Control Menu\n')
        print('\t1. Pen Up')
        print('\t2. Pen Down')
        print('\t3. Turn Right')
        print('\t4. Turn Left')
        print('\t5. Move Forward')
        print('\t6. Print Floor')
        print('\t7. Exit')


    def process_menu_choice(self):
        """Process menu commands."""
        # Declare variable to hold user input
        user_input = self._PRINT_FLOOR
        # Declare variable to hold converted menu choice
        menu_choice = self._PRINT_FLOOR
        try:
            # Prompt user for input
            user_input = input('\n\tEnter Command Number: ')
            # Use first character from input string
            menu_choice = user_input[0]
        except:
            # If there's a problem just print the floor
            menu_choice = self._PRINT_FLOOR

        if __debug__:
            print(f'You entered command number: {menu_choice}')
        # Is menu_choice valid command?
        # YES - Execute command
        # NO - Display error message and try again
        match menu_choice:
            case self._PEN_UP: self.set_pen_up()
            case self._PEN_DOWN: self.set_pen_down()
            case self._TURN_RIGHT: self.turn_right()
            case self._TURN_LEFT: self.turn_left()
            case self._MOVE_FORWARD: self.move_forward()
            case self._PRINT_FLOOR: self.print_floor()
            case self._EXIT: sys.exit()
            case _: self.print_error_message(menu_choice)


    def set_pen_up(self):
        """Changes pen state to UP."""
        if __debug__:
            print('set_pen_up() method called...')
        self._pen_position = self.PenPositions.UP
        print(f'Pen is {self._pen_position}')


    def set_pen_down(self):
        """Changes pen state to DOWN."""
        if __debug__:
            print('set_pen_down() method called')
        self._pen_position = self.PenPositions.DOWN
        print(f'Pen is {self._pen_position}')


    def turn_right(self):
        """Turns Robot Rat to the left."""
        if __debug__:
            print('turn_right() method called...')
        match(self._direction):
            case self.Directions.NORTH:
                self._direction = self.Directions.EAST
            case self.Directions.EAST:
                self._direction = self.Directions.SOUTH
            case self.Directions.SOUTH:
                self._direction = self.Directions.WEST
            case self.Directions.WEST:
                self._direction = self.Directions.NORTH
            case _: self._direction = self.Directions.EAST


    def turn_left(self):
        """Turns Robot Rat to the right."""
        if __debug__:
            print('turn_left() method called...')
        match(self._direction):
            case self.Directions.NORTH:
                self._direction = self.Directions.WEST
            case self.Directions.WEST:
                self._direction = self.Directions.SOUTH
            case self.Directions.SOUTH:
                self._direction = self.Directions.EAST
            case self.Directions.EAST:
                self._direction = self.Directions.NORTH
            case _: self._direction = self.Directions.EAST


    def move_forward(self):
        """ Moves Robot Rat forward by indicated number of spaces.
            If the pen is UP the Robot Rat does not leave a mark on
            the floor. If the pen is DOWN the Robot Rat leaves a mark
            on the floor.
        """
        if __debug__:
            print('move_forward() method called...')
        spaces_to_move = 0
        try:
            spaces_to_move = int(input("Enter spaces to move: "))
        except Exception as e:
            print('Invalid movment number. Setting to 1')
            spaces_to_move = 1

        match(self._pen_position):
            case self.PenPositions.UP:
                match(self._direction):
                    case self.Directions.NORTH:
                        self._current_row -= spaces_to_move
                        if self._current_row < 0:
                            self._current_row = 0
                    case self.Directions.EAST:
                        self._current_col += spaces_to_move
                        if self._current_col > (self._cols - 1):
                            self._current_col = (self._cols - 1)
                    case self.Directions.SOUTH:
                        self._current_row += spaces_to_move
                        if self._current_row > (self._rows - 1):
                            self._current_row = (self._rows - 1)
                    case self.Directions.WEST:
                        self._current_col -= spaces_to_move
                        if self._current_col < 0:
                            self._current_col = 0
            case self.PenPositions.DOWN:
                match(self._direction):
                    case self.Directions.NORTH:
                        while (self._current_row > -1) and (spaces_to_move > 0):
                            self._floor[self._current_row][self._current_col] = True
                            if self._current_row > 0:
                                self._current_row -= 1
                            else:
                                break
                            spaces_to_move -= 1
                    case self.Directions.EAST:
                        while (self._current_col < self._cols) and (spaces_to_move > 0):

                            self._floor[self._current_row][self._current_col] = True
                            if self._current_col < (self._cols - 1):
                                self._current_col += 1
                            else:
                                break
                            spaces_to_move -= 1
                    case self.Directions.SOUTH:
                        while (self._current_row < self._rows) and (spaces_to_move > 0):

                            self._floor[self._current_row][self._current_col] = True
                            if self._current_row < (self._rows - 1):
                                self._current_row += 1
                            else:
                                break
                            spaces_to_move -= 1
                    case self.Directions.WEST:
                        while (self._current_col > -1) and (spaces_to_move > 0):
```

```
204                             self._floor[self._current_row][self._current_col] = True
205                             if self._current_col > 0:
206                                 self._current_col -= 1
207                             else:
208                                 break
209                             spaces_to_move -= 1
210
211
212         def print_floor(self):
213             """ Prints the floor pattern."""
214             if __debug__:
215                 print('print_floor() method called')
216             for rindx, row in enumerate(self._floor):
217                 print('\t', end='')
218                 for cindx, col in enumerate(row):
219                     if((rindx == self._current_row) and (cindx == self._current_col)):
220                         print('X ', end='')
221                     elif col:
222                         print('- ', end='')
223                     else:
224                         print('0 ', end='')
225
226                 print()
227
228
229         def print_status(self):
230             """Displays Robot Rat current position, direction, and pen position."""
231             print(f'\n\tRobot Rat at [{self._current_row}][{self._current_col}] \
232     facing {self._direction} {self._pen_position}')
233
234         def print_error_message(self, menu_choice):
235             """Warns of an invalid command entry."""
236             print(f'WARNING: {menu_choice} is an invalid command!')
237
238         def pause_program(self):
239             input('Press any key to continue... ')
240
241         def clear_screen(self):
242             match(os.name):
243                 case 'posix':
244                     os.system('clear')
245                 case 'nt':
246                     os.system('cls')
247
248         def start_application(self):
249             """Start application processing loop."""
250             while True:
251                 self.clear_screen()
252                 self.display_menu()
253                 self.print_status()
254                 self.process_menu_choice()
255                 self.pause_program()
256
```

Referring to example 4.15 — Starting from the top. On line 9, I import the `os` module, which I use in the `clear_screen()` method, which begins on line 241. The `match` statement checks the name of the operating system as determined by checking the `os.name` property. If the name is `'posix'` then a call is made to `os.system('clear')`. If the name is `'nt'` the `'cls'` command is called instead. Note that `os.system('cls')` works just fine in the Git Bash terminal even though typing the `cls` command directly in the terminal does not unless it's aliased to the `clear` command.

Clearing the screen requires a pause or else a user will be unable to see the floor when it's printed to the console, so I created a `pause_program()` method on line 238, which prompts the user with the well-known `"Press any key to continue..."` message.

Next, I added these two methods to the `start_application()` method as you can see from the code.

While I was poking around and doing some more testing, I noticed that if a user just hits enter without typing a command, the program throws an exception due to there being no string from which to extract a command. To fix this problem, I modified the `process_menu_choice()` method, which begins on line 63, and enclosed the command input code in a `try/except` block.

5.2 Command-Line Parameters

What if you wanted a bigger or smaller floor? The current version of the main.py module hard codes the floor dimensions into the `RobotRatApp()` constructor call at 20 rows and 20 columns. To set the floor dimensions when running the program from the command line, you'll need to process command-line arguments. Example 4.16 gives the modified main.py module.

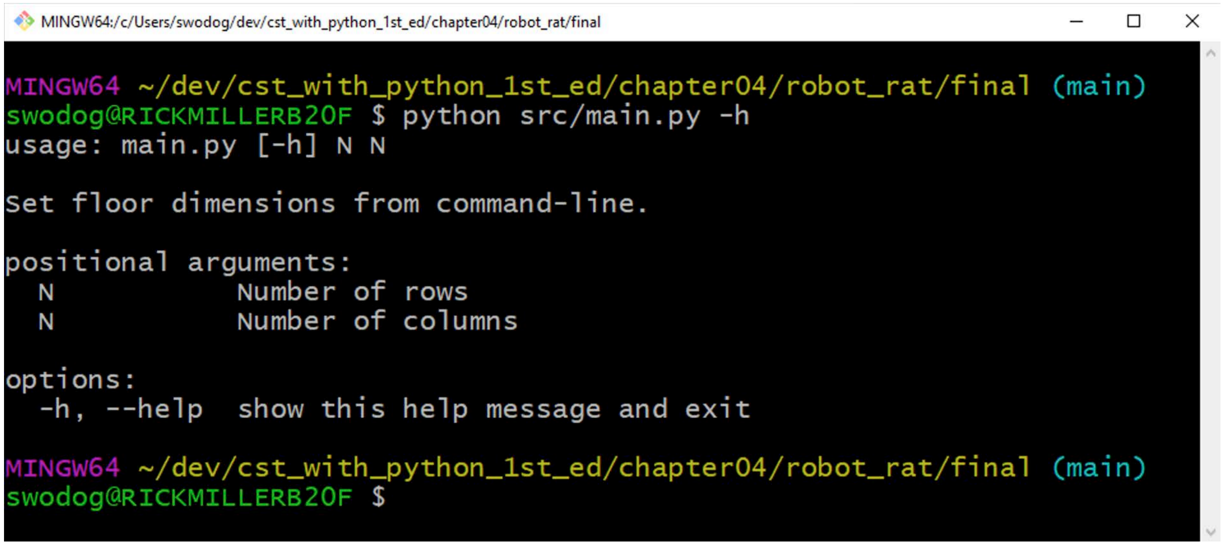
4.16 main.py (process command-line arguments)

```
1    """Serves as the point of entry to the Robot Rat Application.
2
3    Date: 31 December 2022
4    Project: Robot Rat
5    Student: Rick Miller
6    Class: IT-566: Computer Scripting Techniques
7    """
8
9    from robotrat_app import RobotRatApp
10   import argparse
11
12   def main():
13       parser = argparse.ArgumentParser(description='Set floor dimensions \
14   from command-line.')
15       parser.add_argument('rows', metavar='N', type=int, help='Number of rows')
16       parser.add_argument('cols', metavar='N', type=int, help='Number of columns')
17       args = parser.parse_args()
18       robot_rat_app = RobotRatApp(args.rows, args.cols)
19       robot_rat_app.start_application()
20
21
22   if __name__ == '__main__':
23       main()
24
```

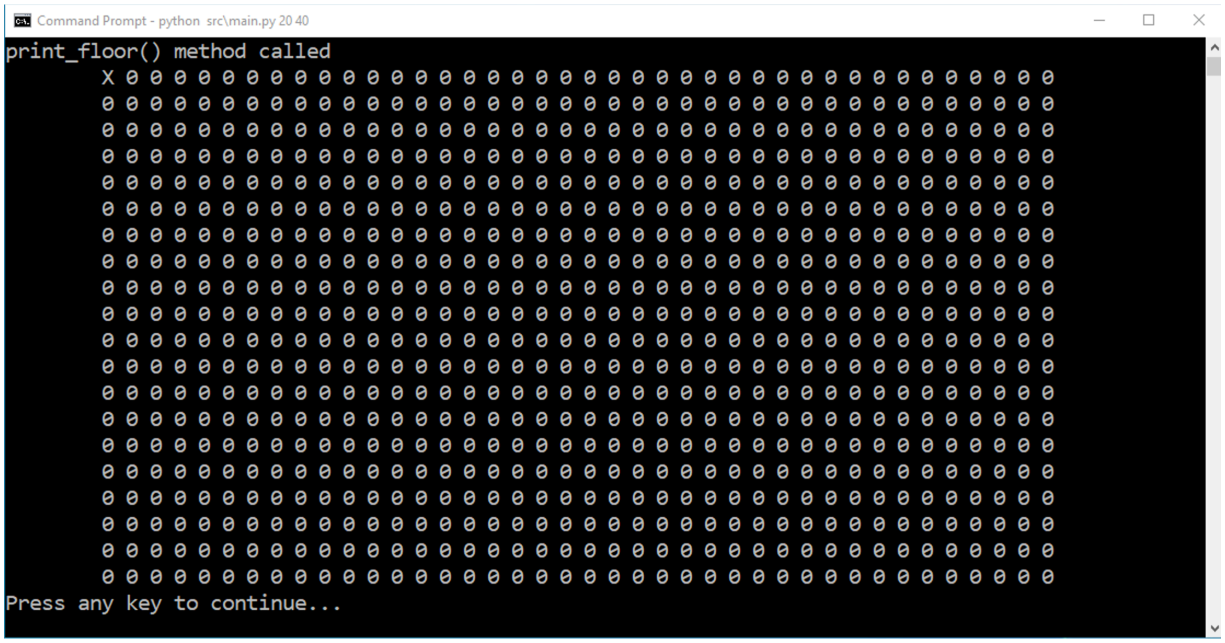
Referring to example 4.16 — On line 10, I import the `argparse` module. Next, in the body of the main() method, I create an ArgumentParser and then, on lines 15 and 16, I add two arguments, one named `rows` and the other `cols`. On line 17, a call to the `parser.parse_args()` method assigns the arguments read from the command line to the `args` variable. On line 18, I use the `args` variable to set the `RobotRatApp()` constructor arguments as shown in the code.

The argparse module provides a ton of great features right out of the box, like automatic string to numeric conversion, automatically generated help and command-line error messages. Figure 4-

23 shows how the automatically generated help looks in a Git Bash terminal along with how a 20 x 40 floor looks when printed to a command-prompt terminal.



```
MINGW64:/c/Users/swodog/dev/cst_with_python_1st_ed/chapter04/robot_rat/final

MINGW64 ~/dev/cst_with_python_1st_ed/chapter04/robot_rat/final (main)
swodog@RICKMILLERB20F $ python src/main.py -h
usage: main.py [-h] N N

Set floor dimensions from command-line.

positional arguments:
  N           Number of rows
  N           Number of columns

options:
  -h, --help  show this help message and exit

MINGW64 ~/dev/cst_with_python_1st_ed/chapter04/robot_rat/final (main)
swodog@RICKMILLERB20F $
```



```
Command Prompt - python  src\main.py 20 40
print_floor() method called
        X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Press any key to continue...
```

Figure 4-23: Above, Getting Help with `-h` Argument in Git Bash - Below, a 20 x 40 in Command-Prompt

Summary

When you're unsure of how to start a software development project, use the project-approach strategy to get organized and maintain a sense of forward momentum. Avoid the mistake of trying to write code before you've clarified application requirements, studied programming language features, created a high-level design, and given some thought to an implementation plan.

Apply the software development cycle iteratively or in sprints. Don't try to code the whole application all at once. Instead, plan a little, code a little, test a lot, and refactor the code as necessary until you converge upon a solution.

Thorough analysis will yield insights into potential solution approaches. You may not know how, at first, to solve a particular coding problem. When this happens, get up from your desk and take a break. Walking is a great way get some exercise and to unleash your mind's subconscious problem-solving power a problem.

Dedicate time at the start of the software development cycle to complete activities that pave the way for overall project success. This time can be formally allocated in a planning sprint, also referred to as Sprint 0 (Zero).

During software development you may accumulate technical debt. Dedicate a sprint (or two) to address outstanding technical debt before it becomes too overwhelming.

Clearing a terminal window from Python code is about the least cross-platform thing you can try to do. Use the `os` module to determine the name of the operating system and call the appropriate command.

Use the argparse module to parse command-line arguments.

Skill-Building Exercises

1. **Experiment with the Code:** Clone the book's repository, step through each of the Robot Rat application development sprints, and run the code. Experiment by making changes and observing the effects. Don't worry about breaking things. You can always delete your local copy and re-clone the repository. The goal of this activity is for you to gain a complete understanding of how the code works

2. **Lists and Tuples:** Read *Chapter 14: Lists & Tuples*. The Robot Rat application uses a two-dimensional array of boolean types (`True`/`False`) to indicate marked and unmarked floor cells. Could an array of characters have worked? What changes to the code would you have to make to use characters verses boolean types? What's the difference between a list and a tuple? Where in the Robot Rat application are tuples used? (Hint: `print_floor()` method.) Can you find more?

3. **For Statement:** A set of nested `for` statements are used to step through the two-dimensional floor array in the `print_floor()` method. Why was it necessary to change from implicit iteration to using the `enumerate()` function? What type of object does the `enumerate()` function return?

4. **Enumerated Types:** Research the enum.py module and the Enum class. What functionality does extending the Enum class provide to derived classes?

5. **Match Statement:** Research the `match` statement. Compare it to nested `if`/`elif`/`else` statements. Which do you prefer?

6. **Classes:** Read Chapter 17: Classes, and Chapter 18: Inheritance. In your opinion, do you think

placing the Robot Rat application code in a class aided or hindered your understanding and comprehension of the code?

7. **Agile Methodology:** Research the Agile Methodology and compare it to the development process followed in this chapter? In your opinion do you think working in sprints is helpful to an individual programmer or a hindrance?

8. **Type Hinting:** Research type hinting and add it to the final version of the RobotRatApp class.

Suggested Projects

1. **Diagonal Movement:** Revisit the Robot Rat project and implement diagonal movement. Currently there are four valid directions: NORTH, SOUTH, EAST, and WEST. Add the following diagonal directions: NORTH-EAST, NORTH-WEST, SOUTH-EAST, and SOUTH-WEST. Modify the `move_forward()` method to support the diagonal movement directions.

2. **Random Direction:** Add a feature to the Robot Rat project that allows a user to move in a random direction.

3. **Movement Recording And Playback:** Read Chapter 16: File I/O, and implement a feature in the Robot Rat application that enables a user to record and playback robot rat movements. Movement details should be saved to and read from a file. Save movement data in JSON format. Movement data should include starting position, pen up and pen down events, direction changes, and spaces moved.

4. **Validate Row And Column Dimensions:** In its current state, a user can launch the Robot Rat application with negative row and column values. Edit the main.py module to ensure row and column values are non-negative and that the minimum row and column dimension is five.

5. **Set Starting Position And Direction:** Modify the Robot Rat application to allow a user to specify the Robot Rat's starting position and direction when launching the application from the command line.

6. **Launch Application With `__debug__` False:** By default, the built-in constant `__debug__` is set to `True` when an application is launched from the command line. Research the `__debug__` constant and figure out how to launch the Robot Rat application with `__debug__` set to `False` to suppress the program's debug messages.

7. **Print Floor After Every Move:** Modify the Robot Rat application so that it prints the floor after each move. Better yet, enable the user to turn this feature on and off.

8. **Directional Indicators:** Currently, the Robot Rat's position on the floor is marked with an `'X'`. Modify the `print_floor()` method to display a different character based on which way the critter is facing. (Hints: You could use the following characters: <, >, ^, v, or research special

Unicode characters.

Self-Test Questions

1. What's the purpose of the project-approach strategy?

2. Explain in your own words why the software development cycle should be executed in sprints.

3. List and describe the Project Approach Strategy areas of concern.

4. List and describe the phases of the software development cycle.

5. What's the purpose of method stubbing?

6. What's the purpose of a state transition diagram?

7. What's the difference between a function and a method?

8. What's the purpose of the `__init__()` method?

9. What's the default value of the built-in `__debug__` constant?

10. What's the purpose of the `argparse.py` module?

References

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, 2000. ISBN 201-61641-6

Python Documentation, *https://www.python.org/doc/*

Agile Manifesto, *https://agilemanifesto.org*

Atlassian Jira, *https://www.atlassian.com/software/jira*

What is Scrum?, Scrum.org, *https://www.scrum.org/resources/what-is-scrum*

argparse Module, *https://docs.python.org/3/library/argparse.html*

Notes