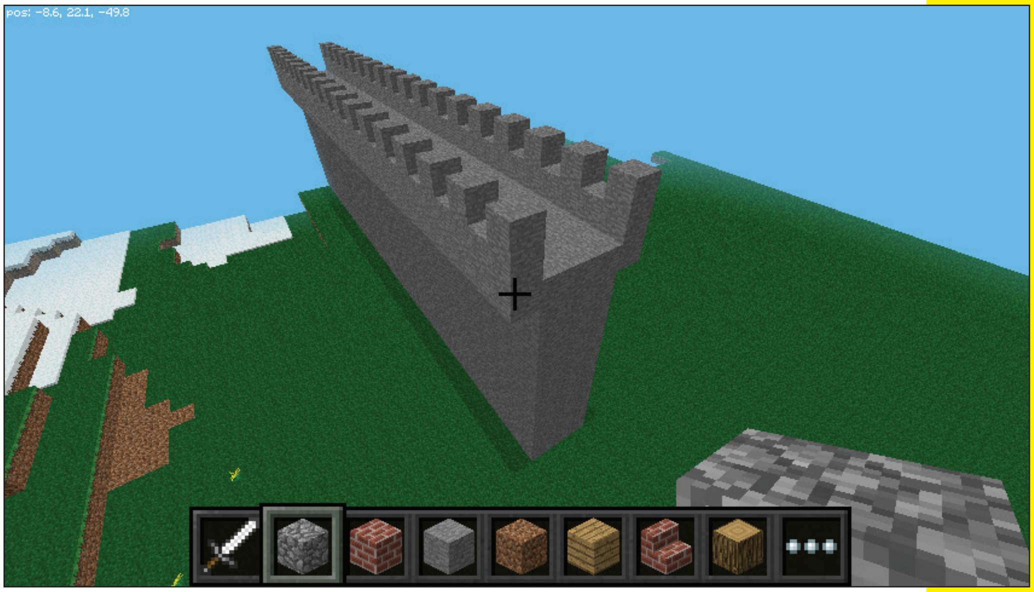
BONUS CHAPTER

3

Minecraft+Py+Pi

In this project you're going to drive your Minecraft world directly from Python on a Raspberry Pi. This is a big project. Find a responsible adult (an irresponsible one will do in a pinch) who can help you.



You're going to connect to your Minecraft game on the Raspberry Pi and change the Minecraft world from your Python interface. You can change and destroy blocks and teleport your player places. Then you'll make a script to clear land and build a castle.

What You Need

You need two things for this project:

- A Raspberry Pi, which is a small computer that you can plug into a regular-sized computer monitor (or a TV).
- A copy of the Raspbian operating system flashed to an SD card. Get a copy of Raspbian at `www.raspberrypi.org/downloads`. Get instructions for flashing the card at `www.raspberrypi.org/documentation/installation/installing-images/README.md`.

At the time of this writing, the Pi cost about $35. However, it doesn't come with a keyboard, mouse, SD card, monitor, or a power supply. See Figure 3-1.

Change the Keyboard

If you press a key and you get the wrong character, Pi is thinking you have a British keyboard. Changing the keyboard isn't tough.

1. **Click the console window in the top panel.**
2. **When the console opens up, type `sudo raspi-config`.**

   That starts a new application window. In this application you:

   - Use the arrow keys to highlight your choice.

- Press Tab until a button at the bottom like `<Select>` or `<Finish>` is highlighted. If you press Tab again, it'll go through your options.
- Press Enter.

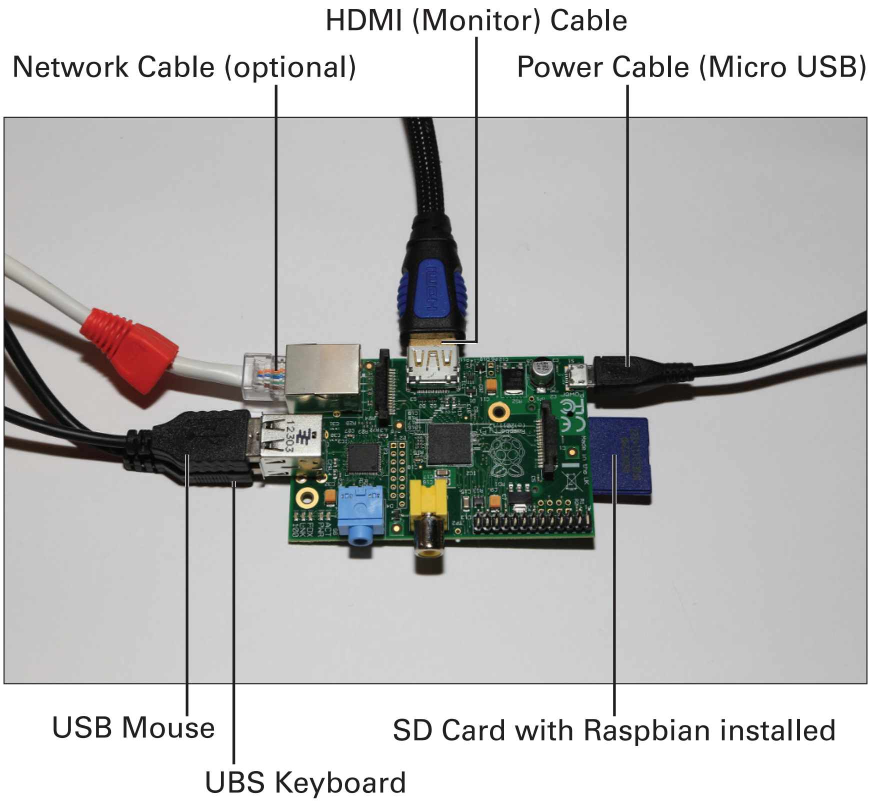
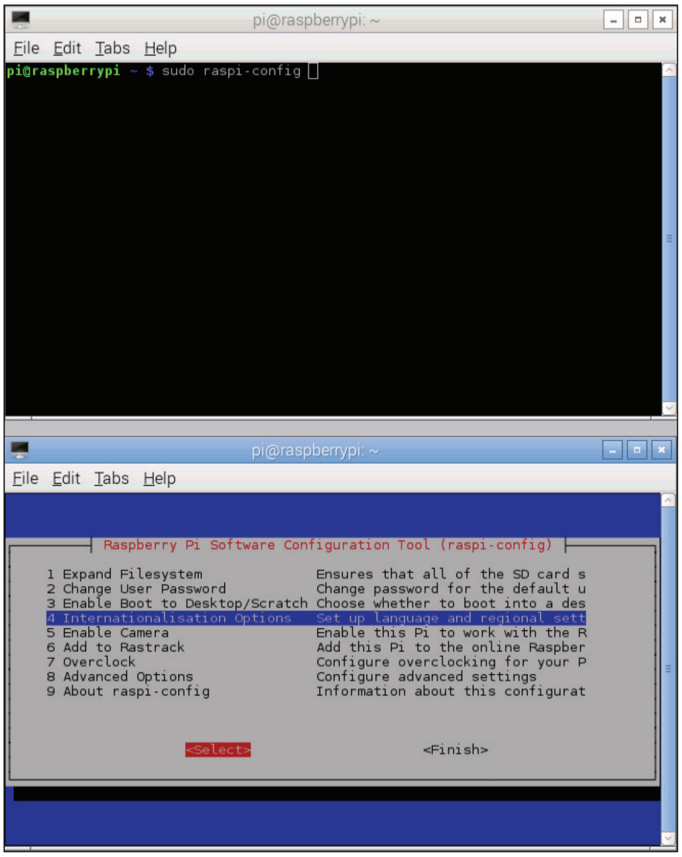


Figure 3-1: My Pi is connected to everything.

3. **On the first screen of the application, press the down arrow to highlight `4. Internationalisation Options`.**

4. **Press Tab once.**

   The word `<Select>` should be highlighted in red. See Figure 3-2. If not, press Tab until it is highlighted.



```
pi@raspberrypi ~ $ sudo raspi-config
```

```
Raspberry Pi Software Configuration Tool (raspi-config)

1 Expand Filesystem              Ensures that all of the SD card s
2 Change User Password           Change password for the default u
3 Enable Boot to Desktop/Scratch Choose whether to boot into a des
4 Internationalisation Options   Set up language and regional sett
5 Enable Camera                  Enable this Pi to work with the R
6 Add to Rastrack                Add this Pi to the online Raspber
7 Overclock                      Configure overclocking for your P
8 Advanced Options               Configure advanced settings
9 About raspi-config             Information about this configurat

              <Select>                        <Finish>
```

Figure 3-2: Setting up the keyboard.

5. **Press Enter.**

   You see three options.

6. **Press the down arrow to `Change Keyboard Layout`. Press Tab to highlight `<Select>`, and press Enter.**

   Wait. Go get a tasty snack or something. Seriously, this takes way too long. Finally, you'll see a list of keyboard configurations.

7. **Unless you know it's wrong, just accept the highlighted entry.**

   Do that by pressing Tab to highlight `<Ok>` and pressing Enter.

8. **If `English (US)` isn't highlighted, use the arrow key to find it. Press Tab to highlight `<Ok>` and press Enter.**

   If you have the Russian (US, phonetic) version of the English (US) keyboard, then select that instead. Obvs.

   You're asked a series of detailed questions about your keyboard layout. No person in their right mind would have any clue.

9. **Accept the first two defaults: Tab to `<Ok>` and Enter for the first two. Tab to `<No>` and press Enter for the third option.**

   If you're a layout expert, go to town customizing these options.

10. **Reboot the Pi.**

    Your new keyboard won't work until you reboot. Then you'll get back the keys you know and love.

Turn Off the Pi

Just so you know for later, here's how you turn off the Pi:

1. **Make sure you've saved all your open files.**
2. **Close down the open applications.**
3. **Choose Shutdown from the Pi's menu button.**

   A dialog box asks which option you want.

4. **Choose Shutdown ➪ OK.**

Get Going

To use Python to drive a Minecraft world, you need to do the following:

1. **Start the Pi.**
2. **Start Minecraft.**
3. **Open your Minecraft world.**
4. **Start Python.**
5. **Connect to Minecraft from Python. Minecraft must be running before you try to connect.**

Start Minecraft

Plug in everything, make sure you have a working image of Raspbian on an SD card, that the card is in your Pi, and everything's booted up.

Using the Pi menu, choose Games ➪ Minecraft Pi.

1. **Click the Start Game button on the Minecraft Pi Edition intro screen.**

   You get an empty Select World screen.

2. **Click the Create New button.**

   Wait for the Pi to chug along and create a world for you. It takes a minute or so. See Figure 3-3.

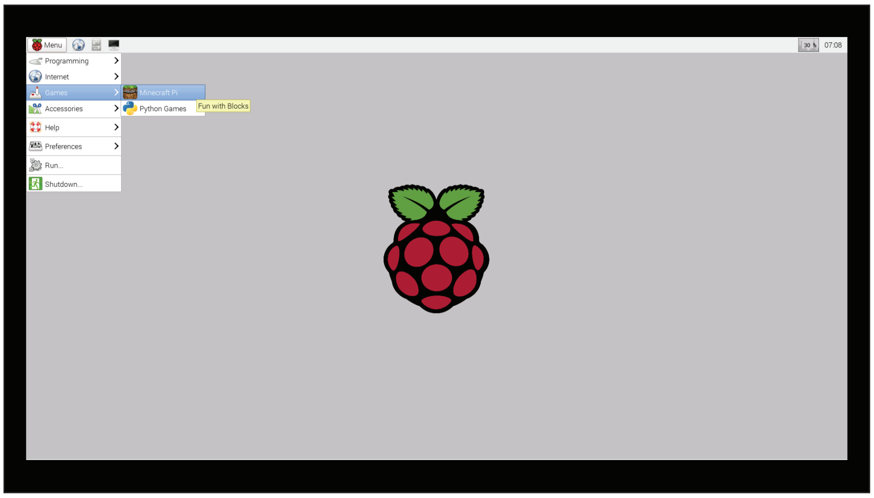


Figure 3-3: You're here for Minecraft Pi.

Know How to Close Minecraft

When Minecraft Pi edition starts, it opens a console window (the black window behind and to the right of the main Minecraft window).

WARNING!

Don't exit now. This is what you need to know when it's time to exit.

To exit Minecraft, do this:

1. **Press the Escape key.**
2. **Click `Quit to Title`.**
3. **Click the close widget (the X) in the corner of the console window.**

Switch Tasks

When you're inside Minecraft, it captures all of your mouse moves and key clicks. That's great if you're playing the game. It's not if you want to use Python. You'll be trapped inside Minecraft and unable to get to your Python window (to type your code, for example).

You have different ways to escape:

- **Press Tab.** This is your best option! The mouse cursor is released so you can pull it to the window you want to work in.
- **Press Alt+Tab** to switch between open windows. The window that you're switching to has a black border around it.

WARNING!

Pressing the Escape key is a bad idea. Don't use this method. It pauses the game, then Python can't do its thing either.

To get back to Minecraft at any time, just click in the Minecraft window.

Start Python

Once the Raspberry Pi is up and Minecraft has started, you need to start a Python programming environment. You're going to use Python to connect to Minecraft while the game is running, and then use Python to tell the game what to do.

Using the Pi menu in the top-left corner, choose Programming ➪ Python 2. (See Figure 3-4.) Voilà! You have an IDLE Shell window.

WARNING!

Stay away from Python 3!

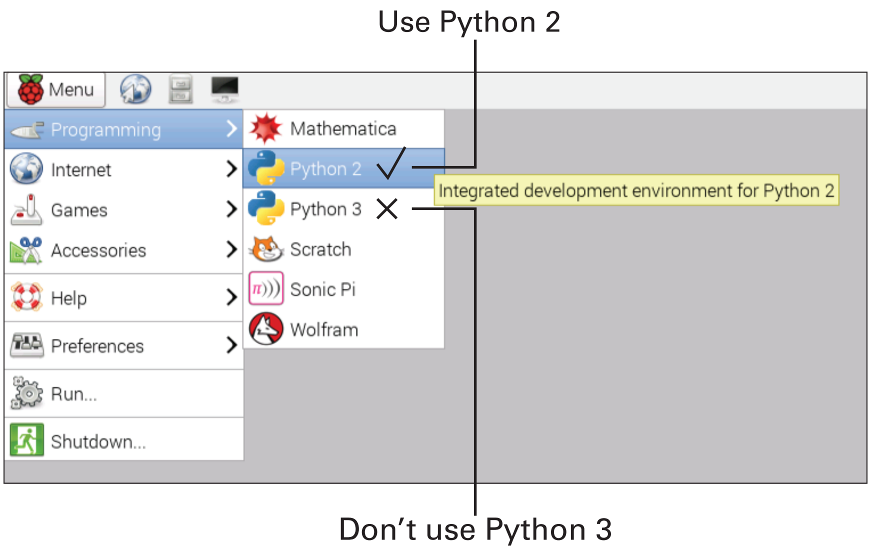


Figure 3-4: Select Python 2 from menu.

Connect Python to Minecraft

For Python to drive Minecraft, Minecraft has to be already running. If it is, then Python can connect to it when you type this:

```
>>> from mcpi import minecraft
>>> minecraft_connection = minecraft.Minecraft.create()
```

The `mcpi` module is on the Raspbian operating system. You work with the `mcpi` module during this project. It lets you work with and drive the Minecraft game. Nothing much seems to happen when you type these lines, but behind the scenes, Python is making a connection into Minecraft for you.

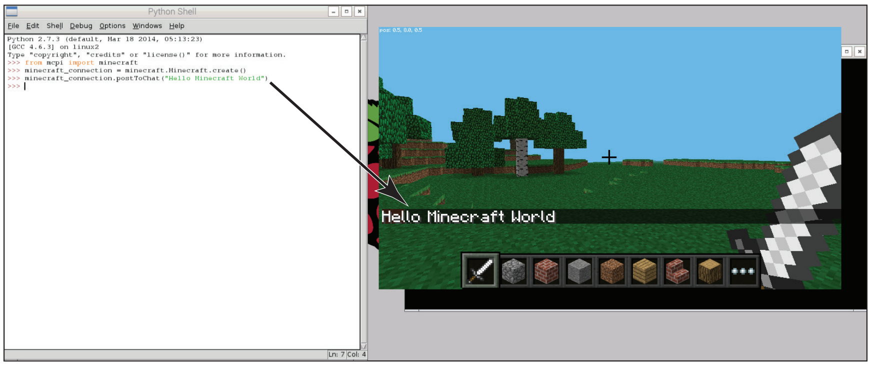
In the IDLE Shell window, type `minecraft_connection.` (include the dot). Then use the Tab key to show the attributes of your new connection.

Hello Minecraft World!

Use the postToChat method to put up a chat line in the world:

```
>>> minecraft_connection.postToChat("Hello Minecraft World")
```

See Figure 3-5.



```
Python 2.7.3 (default, Mar 18 2014, 05:13:23)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> from mcpi import minecraft
>>> minecraft_connection = minecraft.Minecraft.create()
>>> minecraft_connection.postToChat("Hello Minecraft World")
>>> 
```

Figure 3-5: Hello Minecraft World.

Are you upset that postToChat doesn't meet the naming conventions in PEP8? The mcpi module follows the naming conventions of a different programming language. In Python, postToChat would be named post_to_chat. You'll come across different naming conventions from time to time. Everything still works, it's just sometimes hard to follow. No biggie.

Unfortunately, the docstrings for these objects aren't very helpful. You can find some help at www.raspberrypi.org/learning/getting-started-with-minecraft-pi/worksheet.

The player Object

The player attribute of minecraft_connection can give information about the player, such as their current position:

```
>>> minecraft_connection.player.getPos()
Vec3(0.5,4.0,0.5)
```

You should see some numbers in the top-left corner of your Minecraft screen. Those numbers should be the same as the numbers reported by the `getPos()` method.

In Minecraft the player can move in three basic directions — *vertically* (up and down), *horizontally* (left and right), and forward/backward. That's why you need three numbers to describe the position.

You can investigate what happens when you change these coordinates by using the `setPos` method of the `player` attribute. To use the `setPos` method you, give it three numbers as arguments. In the next code, you're going to

1. **Get the player's position.**
2. **Increment the `y` value.**
3. **Set the player's position using this new `y` value.**

```
>>> for i in range(10):
     x,y,z = minecraft_connection.player.getPos()
     minecraft_connection.player.setPos(x,y+i,z)
```

When you run this code, the player in your Minecraft window should quickly float into the sky. When the code finishes running, gravity takes over and the player falls back down. From this you can tell that the second number (`y`) is the player's vertical location (since it's the number that you were adding `i` to).

TIP

The `y` coordinate is especially important because the ground is down. If you set the `y` coordinate too low, you end up inside a block under the ground.

Try it again, changing the `x` and `z` coordinates.

Place Blocks

You can

- Create a block at a specific location using `minecraft_connection`'s `setBlock` method.
- Change an existing block to a new type of block.

To create a block, pass `x`, `y`, and `z` coordinates and the `id` of the block you want to create. (All of the `id`s of the blocks are kept in a separate class called `block`. To get a block `id`, import `block` from `mcpi`. When you do, use your Python introspection powers to get a list of attributes.)

Get the player's position with the `getPos` method, then use `setBlock` `block.DIAMOND_BLOCK` to change the block underneath the player to diamond. `setBlock` takes `x`, `y`, and `z` coordinates as well as the `id` of the block you want to place as arguments:

```
>>> from mcpi import block
>>> player_position = minecraft_connection.player.getPos()
>>> x,y,z = player_position
>>> minecraft_connection.setBlock(x, y-1, z, block.DIAMOND_BLOCK)
```

Click in the Minecraft window to move the player with the mouse. When you make the player look down, you'll see a diamond block like the one in Figure 3-6. It always changes the block underneath the player at the time it's called, even if you started in or moved to a different position in the world. The script first finds the current position of the player, then works out the location of the block below that position. (Note the `y-1`.)

Use `dir` on the `block` module that you imported to see the blocks that you can use:

```
>>> dir(block)
['AIR', 'BED', 'BEDROCK', 'BEDROCK_INVISIBLE', 'BOOKSHELF',
'BRICK_BLOCK', 'Block', 'CACTUS', 'CHEST', 'CLAY'
'COAL_ORE', 'COBBLESTONE', 'COBWEB', 'CRAFTING_TABLE',
'DIAMOND_BLOCK', 'DIAMOND_ORE', 'DIRT',...
```

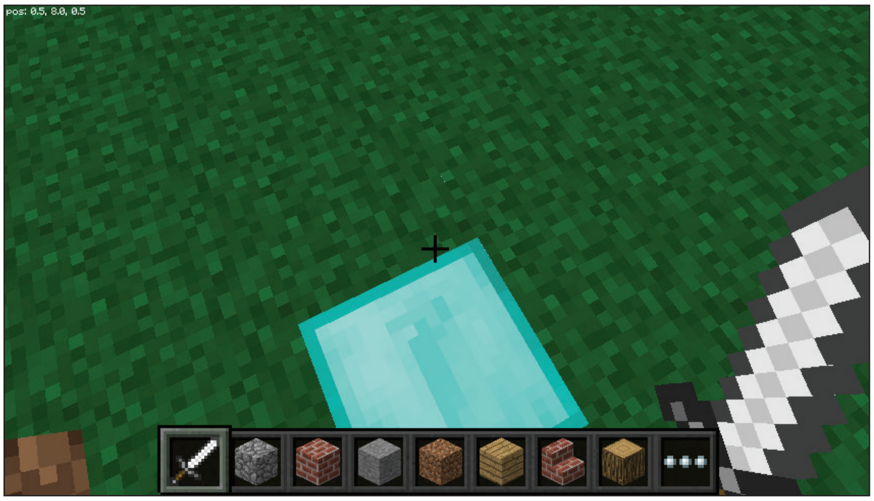


Figure 3-6: Diamonds on the soles of your shoes.

To do more than one block at a time, use the `setBlocks` (not `setBlock`) method. It fills a rectangular box. You need to give it coordinates of two opposite corners of the box you're creating. Give it six numbers. This is equal to two sets of `x`, `y`, `z` coordinates (which is equal to the location of two points in Minecraft space). Any two points define a box. The box is filled with the specified block type.

Try it:

```
>>> minecraft_connection.setBlocks(x, y-1, z, x+3, y+2,
             z+3, block.DIAMOND_BLOCK)
```

Who turned out the lights?!

You can't see anything because the blocks were put right over the player. Use the `setPos` method to move your player up:

```
>>> minecraft_connection.player.setPos(x, y+2, z)
```

The cube you created has four blocks to each side. This is because of the zero basing of the blocks. Don't change the first three numbers, but try different combinations of the second group of three numbers to get a feel for how the boxes are being created. You can destroy blocks by changing them to air (use `block.AIR` instead of `block.DIAMOND_BLOCK`). Try varying a single number at a time.

TIP

If you change these blocks to air, the cube extends below ground. You also fall into it. You can set blocks to air to clear an area.

```
>>> minecraft_connection.setBlocks(x, y-1, z, x+3, y+2,
             z+3, block.AIR)
```

For some blocks, like wool, you can specify additional information by passing extra data (an integer) to `setBlock`. For wool blocks, the integer represents the color. (It means different things for different blocks.) You can do this by using the `withData` method of the wool block.

Make sure you were looking at where the diamond cube was if you want to see this:

```
>>> import time
>>> for i in range(32):
        minecraft_connection.postToChat("Color: %s"%i)
        minecraft_connection.setBlock(x+1, y-1, z, block.WOOL.
             withData(i))
        time.sleep(0.5)
```

This cycles through the available colors. There are only 16 and they repeat, but you don't know this until you try it. You can use colored wool to decorate stuff.

Blow Things Up

Can you set an enormous cube of TNT and blow it up? In short, yes. But beware! The Pi isn't a very powerful computer. The more TNT you set, the harder Pi has to work. You'll get a giant explosion in slooooowwwww moooottiiiioooon . . . .

You can set activated TNT by using `withData(1)`. Then, if you hit a TNT block, it explodes. (Run!) See Figure 3-7.

```
>>> minecraft_connection.player.setPos(x-2,y,z-2)
>>> minecraft_connection.setBlocks(x, y-1, z, x+3, y+2,
             z+3, block.TNT.withData(1))
```

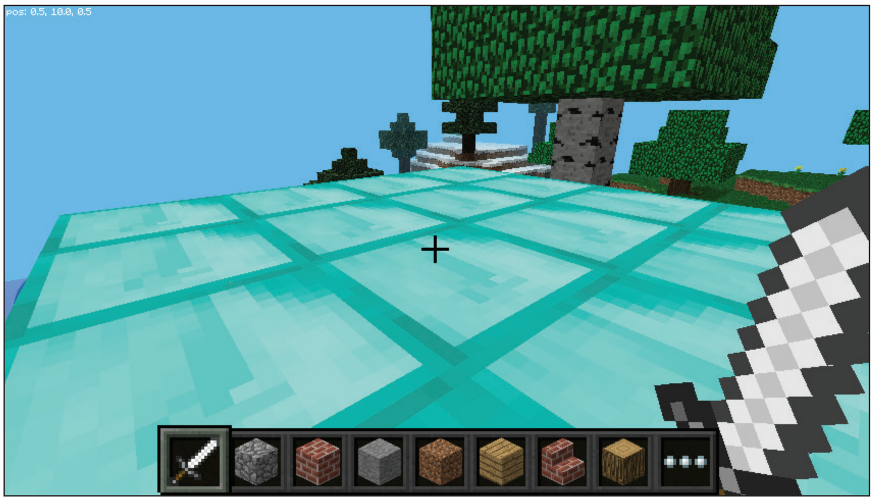


Figure 3-7: This is a small cube of TNT. Yes, small.

Save Your Scripts

You can save your Python scripts on the Pi pretty much the same way as you can on your other computer.

The Pi runs a version of Linux, which has different conventions than Windows. On Linux, IDLE will start in a directory called `/home/pi`. Just save your scripts into this directory. On Windows it was called `C:\Python27`. You can see that Linux uses / in its file paths, but Windows uses \.

REMEMBER

You code in your scripts in the same way that you code in the IDLE Shell window. The only thing you need to remember is that Minecraft needs to be running and not paused when you run your script. Otherwise your script will fail.

Build a Castle

Same old same old here. Set up your files.

1. **Create a file called `build_castle.py`.**
2. **In that file, import `minecraft` and `blocks` from the `mcpi` module.**

   ```
   from mcpi import minecraft, block
   ```

3. **Set up a `mine_craft` connection as you did earlier.**

   ```
   if __name__ == "__main__":
       minecraft_connection = minecraft.Minecraft.create()
   ```

4. **Send a `Hello Minecraft World` message.**

   ```
       message = "Hello out there! We have a connection"
       minecraft_connection.postToChat(message)
   ```

5. **Create a cube of blocks at the player's location.**

   ```
       minecraft_connection.setBlocks(x, y, z,
                                      x+2, y+2, z+2,
                                      block.GRASS)
   ```

6. **Move the player back so it isn't trapped in the blocks.**

   ```
       minecraft_connection.player.setPos(x-1, y, z-1)
   ```

Your code should look something like this:

```
"""
build_castle.py
Build a castle in Minecraft on the Pi!
                   Brendan

Possible do list:In order to build the castle I'm going to:
#TODO: level out an area for the castle to go on
#TODO: build a tower
#TODO: add battlements with merlonations
#TODO: moat?
"""
```

```
# Imports Section
from mcpi import minecraft, block

# Functions Section


# Main Section
if __name__ == "__main__":
    minecraft_connection = minecraft.Minecraft.create()
    x, y, z = minecraft_connection.player.getPos()
    message = "Hello out there! We have a connection"
    minecraft_connection.postToChat(message)
    minecraft_connection.setBlocks(x, y, z,
                                   x+2, y+2, z+2,
                                   block.GRASS)
    minecraft_connection.player.setPos(x-1, y, z-1)
```

Run it like you would normally run a script from IDLE — press F5. Remember that Minecraft must be running first for it to work.

I get something like Figure 3-8 when I run this.

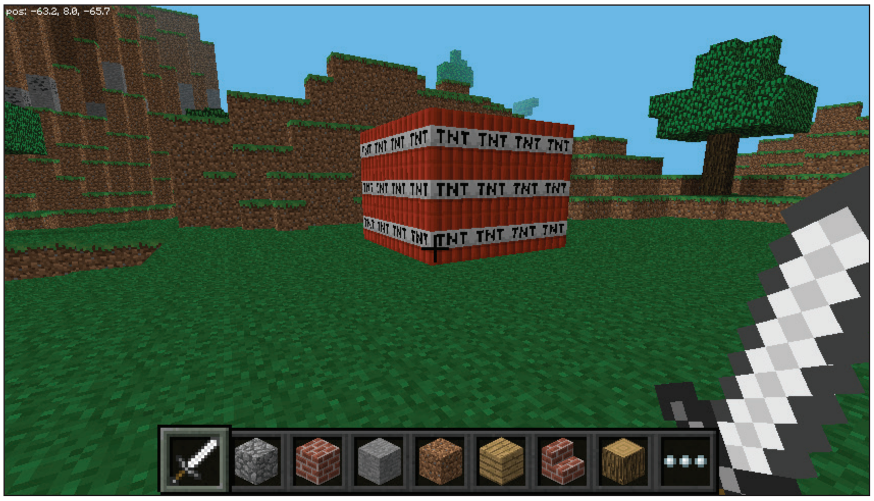


Figure 3-8: Running my first MineCraft Pi Python script.

WARNING!

The code in this project has very long names for its methods and variables. I used long names so that they would be more meaningful. Every once in a while, a line is too long and there is no sensible way to split it. These lines are wrapped in this project. Where you see code that doesn't line up with anything, it's actually part of the previous line. Type it and the previous line as a single line.

Here's an example. Don't add this anywhere:

```
self.restore_blocks[location] = self.minecraft_
 connection.getBlockWithData(location)
```

Create a `Castle` Class and an Interface

The castle will have different pieces that work according to where the player is. Your script will put up a `Tkinter` interface. When you are where you want the castle, click the button on your `Tkinter` window.

You'll use the `grid` method for laying out your widgets.

1. **Delete the stuff you've just done, other than the `Import` section. Import `Tk` and `Button` from `Tkinter`.**

   ```
   from Tkinter import Tk, Button
   ```

2. **Create a `Castle` class, inheriting from `object`.**

   ```
   # Classes Section
   class Castle(object):
       """object for interfacing with minecraft to create and
               destroy
       blocks to create a castle"""
   ```

3. **Create a `View` class, also inheriting from `object`. You don't need a `Frame`, because you're going to be using the grid geometry manager.**

   ```
   class View(object):
       """Interface to Castle object"""
   ```

4. **In the `Castle` constructor accept a `parent` argument and store it in an attribute. Create a view, passing the parent to the view.**

```
def __init__(self, parent=None):
    self.parent = parent
    self.view = View(parent)
```

5. **In the `View` constructor, save a copy of the parent it receives.**

6. **Create two `Buttons` — `button_drop` and `button_clear`.**

   `button_drop` should have the text `'drop'`, and `button_clear` should have the text `'clear'`.

```
def __init__(self, parent=None):
    self.parent = parent
    self.button_drop = Button(self.parent, text='drop')
    self.button_clear = Button(self.parent, text='clear')
```

7. **Arrange `button_drop` by using `button_drop.grid(row=0, column=0)`.**

   Arrange `button_clear` in the same way, but in column 1.

```
    self.button_drop.grid(row=0, column=0)
    self.button_clear.grid(row=0, column=1)
```

8. **In the `Main` section, create a root window.**

```
root = Tk()
```

9. **Instantiate a `Castle` (a class you make in the next step), passing the root window that you've created as the parent.**

10. **Call `mainloop` on the root window and run `Tkinter`'s `mainloop`.**

```
castle = Castle(parent=root)
root.mainloop()
```

The code now looks like this:

```
"""
build_castle.py
Build a castle in Minecraft on the Pi!
Brendan

Possible do list:In order to build the castle I'm going to:
#TODO: level out an area for the castle to go on
#TODO: build a tower
#TODO: add battlements with merlonations
#TODO: moat?
"""

# Imports Section
from mcpi import minecraft, block
from Tkinter import Tk, Button

# Constants Section

# Classes Section
class Castle(object):
    """object for interfacing with minecraft to create and destroy
    blocks to create a castle"""
    def __init__(self, parent=None):
        self.parent = parent
        self.view = View(parent)
        self.minecraft_connection = minecraft.Minecraft.create()


class View(object):
    """Interface to Castle object"""
    def __init__(self, parent=None):
        self.parent = parent
        self.button_drop = Button(self.parent, text='drop')
        self.button_clear = Button(self.parent, text='clear')
        self.button_drop.grid(row=0, column=0)
        self.button_clear.grid(row=0, column=1)

# Functions Section
# Main Section
if __name__ == "__main__":
    root = Tk()
    castle = Castle(parent=root)
    root.mainloop()
```

The connection to Minecraft is basically acting like a place to store data. You're communicating with the Minecraft server application. That application is communicating with its own view — the screen where you normally play Minecraft.

When you run this code, the buttons don't do anything because they're not hooked up *(bound)* to any methods yet.

Create Methods to Drop and Clear Blocks

Now create methods for dropping and clearing blocks. It's too bad `mcpi` doesn't tell you which direction that the player is facing. You're just going to place a block at the position with an `x` coordinate 1 greater than the player's `x` coordinate.

Drop a block by creating a `GRASS` block (or whatever block you want) at the location and clear it by setting it to `AIR`.

1. **Create a method called `drop_block` that takes:**

   - A `minecraft Vec3` object (call it `drop_location`), defaulting to `None`
   - A kind of block defaulting to `block.GRASS` (or whatever you want)

   ```
   def drop_block(self, drop_location=None, block=block.GRASS):
       """ drop a block of block (a 2 tuple of block id
           and data)
       at drop_location (a Vec3 or 3 tuple). If None, use
           player
       position at x+1."""
   ```

2. **In the method, test if the `Vec3` received is `None`.**

   If so, set it to the player's current position, but with the `x` coordinate increased by 1.

   ```
   if drop_location is None:
       x,y,z = self.minecraft_connection.player.getPos()
       drop_location = minecraft.Vec3(x+1, y, z)
   ```

3. **Call Minecraft's `setBlock` method passing `drop_location` and `block` as arguments.**

   ```
   self.minecraft_connection.setBlock(drop_location, block)
   ```

4. **Create a method called `clear_block` that takes a `Vec3` object as an argument called `clear_location`, defaulting to `None`.**

   Call `drop_block` passing `clear_location` and `block.AIR`.

   ```
   def clear_block(self, clear_location=None):
       """ set the block at clear_location (Vec3 or 3 tuple)
       to Air. If None, use player position at x+1"""
       self.drop_block(clear_location, block.AIR)
   ```

5. **In `Castle`'s constructor, hook up the buttons to these new methods.**

   ```
   # Hook up callbacks
   self.view.button_drop.config(command=self.drop_block)
   self.view.button_clear.config(command=self.clear_block)
   ```

   Now it's time to test your code.

6. **Find a flat place to stand so you'll notice a block when it's dropped.**

7. **Click the drop button.**

   You may need to click it twice.

   If the block doesn't appear, go back to the Minecraft window and move the character around to see if it's there. When it's in view, test the clear and drop buttons by clicking them to create/destroy the block.

TIP It's often a good idea to print out the name of the callback from in the callback when you first write it. If the callback is correctly bound, then the print will show up in the IDLE Shell window. This helps you debug by proving that the callback is hooked up the right way. Once you know the callback is correctly hooked up, comment out the `print` statement.

Here are the two new methods:

```
def drop_block(self, drop_location=None, block=block.GRASS):
    """ drop a block of block (a 2 tuple of block id and data)
    at drop_location (a Vec3 or 3 tuple). If None, use player
    position at x+1."""
    if drop_location is None:
        x,y,z = self.minecraft_connection.player.getPos()
        drop_location = minecraft.Vec3(x+1, y, z)
    self.minecraft_connection.setBlock(drop_location, block)

def clear_block(self, clear_location=None):
    """ set the block at clear_location (Vec3 or 3 tuple)
    to Air. If None, use player position at x+1"""
    self.drop_block(clear_location, block.AIR)
```

This is the code to hook them up in the constructor:

```
# Hook up callbacks
self.view.button_drop.config(command=self.drop_block)
self.view.button_clear.config(command=self.clear_block)
```

Show Your Axes

To know where things are going when you create them, you need to know what direction you're facing. As I mention, the Python interface on the Pi will tell you where the player is, but not which way they're facing, so there's no way to be sure of putting a block in front of the player from the Python end. However, if you know which way is which, then you can make sure that the player is facing the right way before you drop the block (or, later, castle).

With colored wool, create some coordinates that you can display and remove with a button.

1. **Create two constants: `WOOL_COLORS` and `AXIS_COLORS`.**

   You'll use these constants to give a different color to each axis. That way you can tell the x axis from the z axis.

```
# Constants Section
WOOL_COLORS = {"white":0,
               "orange":1,
               "pink":2,
               "light_blue":3,
               "yellow":4,
               "light_green":5,
               "light_red":6,
               "dark_gray":7,
               "light_gray":8,
               "middle_blue":9,
               "violet":10,
               "dark_blue":11,
               "brown":12,
               "dark_green":13,
               "red":14,
               "black":15}

AXIS_COLORS = [block.WOOL.withData(WOOL_COLORS['white']),
                block.WOOL.withData(WOOL_COLORS['black']),
                block.WOOL.withData(WOOL_COLORS['light_green']),
                block.WOOL.withData(WOOL_COLORS['yellow'])]
# Axis colors are positive x, negative x, positive z, negative z
```

2. **In `Castle`, create methods called `show_axes` and `hide_axes`.**

```
    def show_axes(self):
        """ Display colored blocks at x and z each axis
            directions
        to allow player to orient themself.
        """
    def hide_axes(self):
        """ Remove axes previously placed. Restore original
            blocks if
        saved """
```

3. **In `Castle`'s constructor, create an empty dictionary called `restore_blocks`.**

```
        self.restore_blocks = {}
```

4. **In `View's` constructor, create a button to hook up to each of these new methods.**

```
        self.button_show_axes = Button(self.parent, text='Axes +')
        self.button_show_axes.grid(row=1, column=0)
        self.button_hide_axes = Button(self.parent, text='Axes -')
        self.button_hide_axes.grid(row=1, column=1)
```

5. **In `Castle`'s constructor, hook up the buttons you just created.**

```
        self.view.button_show_axes.config(command=self.show_axes)
        self.view.button_hide_axes.config(command=self.hide_axes)
```

6. **In your new `show_axes` method, get the player's position as `x, y, z`.**

```
        x, y, z = self.minecraft_connection.player.getPos()
```

7. **Create a list with the elements `[(x+2,y+2,z), (x-2,y+2,z), (x,y+2,z+2), (x, y+2,z-2)]`.**

```
        locations = [(x+2, y+2, z), (x-2, y+2, z),
                     (x, y+2, z+2), (x, y+2, z-2)]
```

These indicate your plus and minus x and z axes centered at the player's location. Please note that each of the elements is a tuple.

REMEMBER

Tuples are like lists only you can't change them once you make them. The order of the elements in a tuple should be meaningful.

8. **Set your `restore_blocks` dictionary to be empty.**

```
        self.restore_blocks = {}
```

9. **Enumerate each of the four locations in your list.**

   As you do, use the `getBlockWithData` method to get the information of the block at that location.

10. **Store that data in the dictionary using `location` as a key.**

    This restores the blocks that were originally there.

11. **Get the corresponding block type from `AXIS_COLORS` and call `drop_block`, passing the location and block type to that method.**

```
        for i, location in enumerate(locations):
            self.restore_blocks[location] = self.minecraft_
              connection.getBlockWithData(location)
            block = AXIS_COLORS[i]
            self.drop_block(location, block)
```

12. **In `hide_axes`, run through each item in the `restore_blocks` dictionary.**

13. **Call `drop_block`, passing the key as the location and the value as the block.**

```
        for location, block in self.restore_blocks.items():
            self.drop_block(location, block)
```

The consolidated changes from this code follow.

In the `Constants` section:

```
# Constants Section
WOOL_COLORS = {"white":0,
               "orange":1,
               "pink":2,
               "light_blue":3,
               "yellow":4,
               "light_green":5,
               "light_red":6,
               "dark_gray":7,
               "light_gray":8,
               "middle_blue":9,
               "violet":10,
               "dark_blue":11,
               "brown":12,
               "dark_green":13,
               "red":14,
               "black":15}

AXIS_COLORS = [block.WOOL.withData(WOOL_COLORS['white']),
               block.WOOL.withData(WOOL_COLORS['black']),
               block.WOOL.withData(WOOL_COLORS['light_green']),
               block.WOOL.withData(WOOL_COLORS['yellow'])]
# Axis colors are positive x, negative x, positive z, negative z
```

In `Castle`'s constructor:

```
        self.restore_blocks = {}
```

And a little later (make sure it's after `self.view` has been instantiated):

```
        self.view.button_show_axes.config(command=self.show_axes)
        self.view.button_hide_axes.config(command=self.hide_axes)
```

In the `View`'s constructor:

```
        self.button_show_axes = Button(self.parent, text='Axes +')
        self.button_show_axes.grid(row=1, column=0)
        self.button_hide_axes = Button(self.parent, text='Axes -')
        self.button_hide_axes.grid(row=1, column=1)
```

In the `Castle` class I added these new methods:

```
    def show_axes(self):
        """ Display colored blocks at x and z each axis directions
        to allow player to orient themself.
        """
        x, y, z = self.minecraft_connection.player.getPos()
        locations = [(x+2, y+2, z), (x-2, y+2, z),
                     (x, y+2, z+2), (x, y+2, z-2)]
        self.restore_blocks = {}
        for i, location in enumerate(locations):
            self.restore_blocks[location] = self.minecraft_
              connection.getBlockWithData(location)
            block = AXIS_COLORS[i]
            self.drop_block(location, block)

    def hide_axes(self):
        """ Remove axes previously placed. Restore original blocks if
        saved """
        for location, block in self.restore_blocks.items():
            self.drop_block(location, block)
```

Landscape (More Like Terraform)

To build a castle, you need to

- Clear some area. You can set blocks to `block.AIR` to do this.
- Create some ground on which the castle rests. You can use `block.BEDROCK` to do this.

You can create blocks that float in air. If you do, then any area underneath your castle will be a cave or cavern.

I'm going to put the castle on a slab of bedrock blocks. You can choose the filler material if you like — and it can't be air!

Create a `set_blocks` Function

I don't like the `setBlocks` interface because it needs absolute (not relative) coordinates. With relative coordinates you can basically say "Build a box starting at x, y, z and with a width of w, a height of h, and a length of l." In absolute coordinates you have to say "Build a box starting at x, y, z and ending at x+w, y+h, z+l," which you'd need to do each time.

Instead you're going to write a method to use relative coordinates.

1. **Create a function called `set_blocks`. Make it accept the following:**

   - `x`, `y`, and `z` coordinates for a starting block
   - Values `w`, `h`, `l` for the width, height, and length
   - A block `id`
   - Block data

```
def set_blocks(self, x, y, z, w, h, l, block_id,
        block_data):
    """ Set blocks as a box of width, height and
    length w, h and l starting at location x, y z
    """
```

2. **Add `w` to `x`, `h` to `y` and `l` to `z`, then call `setBlocks`**

```
    x2, y2, z2 = x+w, y+h, z+l
    self.minecraft_connection.setBlocks(x, y, z,
                                        x2, y2, z2,
                                        block_id,
        block_data)
```

3. **Add a button in `View`'s constructor called `button_test`.**

```
    self.button_test=Button(self.parent, text="Test")
    self.button_test.grid(row=2,column=0)
```

4. **Create a new method called `test_button`.**

```
def test_button(self):
    """ A short method to test whether the new
    set_blocks method is working"""
```

5. **Hook up the new button to the new method in `Castle`'s constructor:**

```
    self.view.button_test.config(command=self.test_button)
```

6. **Add a `player` attribute to the `Castle` so you can get the position a little more easily.**

   This goes in the constructor after `minecraft_connection` has been initialized.

   ```
   self.player = self.minecraft_connection.player
   ```

7. **In your `test_button` method, get the current position of the player.**

   ```
   x, y, z = self.player.getPos()
   ```

8. **Call `set_blocks` with the player's position as the first three arguments (add 1 to the `x` value to create the blocks away from the player) and `9, 1, 4` as the next three values.**

9. **Pass `block.GLASS` as the block type and `0` as the block data.**

   ```
   w, h, l = (9, 1, 4)
   self.set_blocks(x+1, y, z, w, h, l, block.GLASS, 0)
   ```

   When you run the program, use the coordinate axes function you just wrote to test and make sure you're facing the right direction (the grey wool block) before pressing the `Test` button. When you create the blocks, they're transparent. (They're made of glass!) Press the S key to step back and see them more clearly.

I added this code to `Castle`'s constructor:

```
self.view.button_test.config(command=self.test_button)
```

I also added these two new methods:

```
def set_blocks(self, x, y, z, w, h, l, block_id, block_data):
    """ Set blocks as a box of width, height and
    length w, h and l starting at location x, y z
    """
```

```
        x2, y2, z2 = x+w, y+h, z+l
        self.minecraft_connection.setBlocks(x, y, z,
                                            x2, y2, z2,
                                            block_id, block_data)

    def test_button(self):
        """ A short method to test whether the new
        set_blocks method is working"""
        self.player = self.minecraft_connection.player
        x, y, z = self.player.getPos()
        w, h, l = (9, 1, 4)
        self.set_blocks(x+1, y, z, w, h, l, block.GLASS, 0)
```

In `View`'s constructor I added a Test button:

```
        self.button_test = Button(self.parent, text="Test")
        self.button_test.grid(row=2, column=0)
```

TIP

You can use your `set_blocks` method to quickly clear an area where you want to build something. If you want to add a single layer (of grass, for example), set the height to `0`.

WARNING!

Try to avoid using the letter `l` (the letter after `k`) as the name of a variable. It looks too much like a `1` (the number after `0`). I use it here as a compromise because of line length problems.

Make Towers and Walls

You can use your new `set_blocks` method to create towers and walls. For each, you create the base of the structure, a battlement at the top, and crenellations in the battlement.

A *crenel* is a hole at the top of the wall (on either side of the crenel is a *merlon*). *Battlements* are crenellated so defenders can fire down from behind cover.

Creating the towers and walls is somewhat boring addition and subtraction. I give you the source code to create a tower and a wall and then I explain them to you.

First, I created some constants in the `Constants` section to make life easier (`BLOCK_AIR` and `BLOCK_STONE`) and some to avoid magic numbers (`TOWER_HEIGHT` and `TOWER_WIDTH`):

```
BLOCK_AIR = block.AIR
BLOCK_STONE = block.STONE

# Castle Constants
TOWER_HEIGHT = 12
TOWER_WIDTH = 4
```

I added a new method, `make_tower` that, you know, makes a tower.

```
    def make_tower(self,
                   location=None,
                   tower_height=TOWER_HEIGHT,
                   tower_width=TOWER_WIDTH,):
        """Make a Tower at location (or player pos)"""
        if location is None:
            x, y, z = self.player.getPos()
            x = x+1 # So player is outside tower.
        else:
            x, y, z = location

        w, h, l = tower_width, tower_height, tower_width
        block_data = 0
        # Tower
        self.set_blocks(x, y, z,
                        w, h, l, BLOCK_STONE, block_data)
        # Battlement at top
        w, h, l = tower_width+2, 2, tower_width+2
        self.set_blocks(x-1, y+tower_height, z-1,
                        w, h, l, BLOCK_STONE, block_data)
        # Hollow out battlement
        w, h, l = tower_width, 1, tower_width
        self.set_blocks(x, y+tower_height+1, z,
                        w, h, l, BLOCK_AIR, block_data)
```

```
# Add further hollows for merlonations/crenellations
# cut three times on x and three times on z:
for i in range(3):
    self.set_blocks(x+2*i, y+tower_height+2, z-1,
                    0, 0, 6, BLOCK_AIR, block_data)
    self.set_blocks(x-1, y+tower_height+2, z+2*i,
                    6, 0, 0, BLOCK_AIR, block_data)
# Hollow out tower
w, h, l = tower_width-2, tower_height+1, tower_width-2
self.set_blocks(x+1, y, z+1, w, h, l, BLOCK_AIR, block_data)
```

The method `make_tower` takes `location` as an argument (so that it knows where in the world you want the tower to be made) defaulting to `None`. If a location is provided to the function, it'll use it. If not, it uses the player's current location (section `# Set location`). Look at Figure 3-9 and the comments in the code.

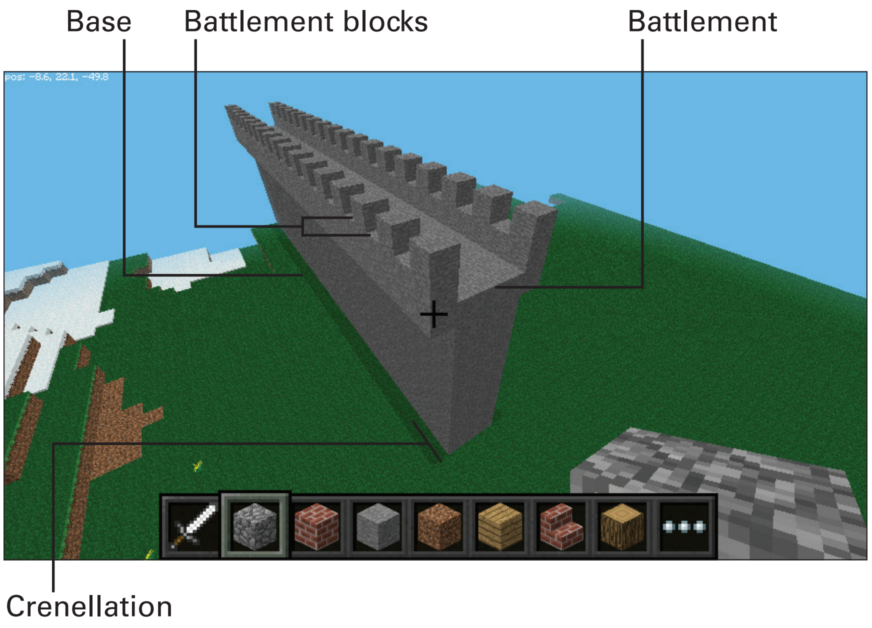


Figure 3-9: Building a tower.

It makes a column of stone (section `# Tower`). Then, at the top of that column, it makes a box of stone three blocks high and two blocks wider than the column it's on (so it has an overhang of one block on each side). This becomes the battlement (section `# Battlement at top`).

The center of the battlement is scooped out by creating a box of air (section `# Hollow out battlement`). Then you cut out three parallel lines in each direction to create the crenellations (section `# Add further hollows for merlonations/crenellations`). Finally, you scoop out a hole down the center of the tower (section `# Hollow out tower`).

If it's hollow you can put a ladder in there.

In addition, you change the `Main` section so that it doesn't open up the `Tk` window while you're testing. The new `Main` section looks like this:

```
# Main Section
if __name__ == "__main__":
##    root = Tk()
##    castle = Castle(parent=root)
##    root.mainloop()
    castle = Castle()
    castle.make_tower()
```

This sped things up a little since the Pi has trouble opening the Tk window. If you want to test within the GUI, you could hook up these new methods to the Test button you created.

TIP

These methods create a heap of blocks. To test them, you'll want an easy way to clear them. You can either

- Write some code to clear them (using `setBlocks` and setting them to `block.AIR`) in a separate file.
- Clear them through the IDLE Shell window (using the command history).

I imported the `Castle` class and used its methods to clear space (sample Shell excerpt):

```
>>> import build_castle as bc
>>> c = bc.Castle()
>>> x,y,z = c.player.getPos()
>>> x,y,z
(-16.7641, 7.0, -53.9972)
>>> c.set_blocks(x,y,z,x+90,y+60,z+90, bc.block.AIR,0)
```

To make the walls, I created these constants (to avoid magic numbers) and put them in the `Constants` section. The length and height of the wall can be whatever, but these numbers give a wall with good proportions.

```
WALL_LENGTH = 40
WALL_HEIGHT = 9
WALL_WIDTH = 2
```

This is the code that creates a wall (also a method in `Castle`):

```
    def make_wall(self,
                  location=None, orientation="x",
                  wall_length=WALL_LENGTH,
                  wall_height=WALL_HEIGHT,
                  wall_width=WALL_WIDTH):
        """ make a wall at location (or player pos) parallel to
        x axis by default otherwise parallel to z axis """

        block_data = 0

        if location is None:
            x, y, z = self.player.getPos()
            x = x+1 # So player is outside wall.
        else:
            x, y, z = location

        w, h, l = wall_length, wall_height, wall_width # wall base
        w1, h1, l1 = wall_length, 2, wall_width+2 # battlement
        w2, h2, l2 = wall_length, 1, wall_width # hollow out battlement
        xoffset, zoffset = 0, -1

        if orientation != "x": # swap dimensions
            w, l = l, w
            w1, l1 = l1, w1
            w2, l2 = l2, w2
            xoffset, zoffset = zoffset, xoffset


        # Wall support
        self.set_blocks(x, y, z, w, h, l, BLOCK_STONE, block_data)

        # Add Battlement
        self.set_blocks(x+xoffset, y+wall_height, z+zoffset,
                        w1, h1, l1, BLOCK_STONE, block_data)
        # hollow out battlement
        self.set_blocks(x, y+h+1, z,
                        w2, h2, l2, BLOCK_AIR, block_data)

        # create crenellations/merlonations
        if orientation == "x":
            for i in range(1, wall_length, 2):
                self.set_blocks(x+i, y+h+h1, z-1,
                                0, 0, 4, BLOCK_AIR, block_data)
        else:
            for i in range(1,  wall_length, 2):
                self.set_blocks(x-1, y+h+h1, z+i,
                                4, 0, 0, BLOCK_AIR, block_data)
```

Towers are the same when you turn them around, so you don't need to say where the tower is facing. Walls are different. A wall going left-right is different from one going forward-backward.

`make_wall` needed a parameter to control which way the wall faces (that is, either the `x` or `z` direction). When I put the code together, I tested it on the `x` direction only. After it was working, I added some code to swap the widths and lengths around to turn the wall sideways. Then I tested that it worked in the `z` direction.

The wall has two main parts — a battlement along the top and the main body of the wall supporting it. The main body is pretty simple, it's just a box (section `# Wall support`). Adding a battlement (section `# Add battlement`) is like the battlement at the top of the tower, except that it's a rectangle.

To add crenellations, slice every second block along the length of the battlement. How you do this depends on the wall's orientation, so handle these cases separately (section `# create crenellations/merlonations`).

Finally, I changed the code in the `Main` section to test `make_wall` in both the `x` and `z` directions:

```
        castle = Castle()
##      castle.make_wall()
        castle.make_wall(orientation="z")
```

The line `castle.make_wall()` is commented out because I tested both the `x` direction (default) *or* the `z` direction — only one at a time. See Figure 3-10.

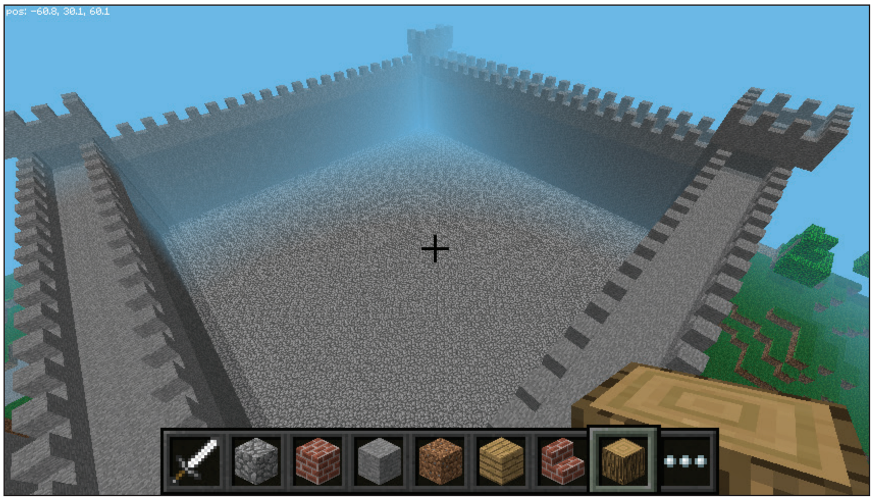


Figure 3-10: Building a wall.

Make the Castle

To make the castle, build four towers and four walls. Clear out an area big enough to fit the castle onto. Again, you mostly use addition and subtraction to work out which blocks to change. Because you can make towers and walls by giving a location, call `make_tower` and `make_wall` four times each, with different parameters each time.

First, constants in the `Constants` section. They're for working out

- How much stuff to clear. (It needs to be at least as big as the largest side of the castle — length of wall plus width of the battlements of two towers.)
- How much bedrock to put under the castle.
- How far it is from one tower/wall to another. (You can calculate this or figure it out by trial and error.)

An interesting extension would be to change the code so you could specify the length of a wall. The code would build a castle to that dimension.

```
CLEAR_SPACE = 60
CASTLE_SIDE = 51
TOWER_OFFSET = 46
BEDROCK_HEIGHT = 8
BEDROCK_DEPTH = 10
```

Here's the source code to make the castle. I comment on it after the code:

```
    def make_castle(self, location=None):
        """ Create a castle at the specified location, or at the
             player's
         position if none specified"""
        if location is None:
            x, y, z = self.player.getPos()
            x = x+1 # So player is outside
        else:
            x, y, z = location

        Vec3 = minecraft.Vec3 #convenience assignment

        # Terraforming!
        # Clear the area
        self.set_blocks(x-(TOWER_WIDTH+1), y, z-(TOWER_WIDTH+1),
                        CLEAR_SPACE, CLEAR_SPACE, CLEAR_SPACE,
                        block.AIR.id, 0)
        # lay down some bedrock underneath the castle
        self.set_blocks(x, y-BEDROCK_DEPTH, z,
                        CASTLE_SIDE, BEDROCK_HEIGHT, CASTLE_SIDE,
                        block.BEDROCK.id, 0)
        # one layer of cobblestone
        self.set_blocks(x, y-1, z,
                        CASTLE_SIDE, 0, CASTLE_SIDE,
                        block.COBBLESTONE.id, 0)

        # create all the towers at the location
        self.make_tower(Vec3(x, y, z))
        # now make another two towers - each at the end of each wall
        # assumes length of wall is 40 (actually 41 bc of 0 basing)
        # and width of towers is 4 (actually 5)
        self.make_tower(Vec3(x+TOWER_OFFSET, y, z))
        self.make_tower(Vec3(x, y, z+TOWER_OFFSET))
        # make last tower is diagonally opposite the first
        self.make_tower(Vec3(x+TOWER_OFFSET, y, z+TOWER_OFFSET))

        # now make two walls to attach to the first tower.
        # Starting point of each wall needs to be offset
        # by (width of tower)+1 blocks in the x axis for the
        # first wall and in the z axis for the second wall.
        self.make_wall(Vec3(x+TOWER_WIDTH+1, y, z+1))
        self.make_wall(Vec3(x+1, y, z+TOWER_WIDTH+1),
                       orientation="z")

        # repeat the walls above, but offset them by the
        # tower offset to the z and x axes
        # This is so that the end of the wall meets the side
        # of the tower
        self.make_wall(Vec3(x+TOWER_WIDTH+1, y, z+1+TOWER_OFFSET))
        self.make_wall(Vec3(x+1+TOWER_OFFSET, y, z+TOWER_WIDTH+1),
                       orientation="z")
```

Since I was going to be using a few `minecraft.Vec3` objects, I assigned the object to the name `Vec3` to save a bit on typing. First I cleared out the area where the castle is going to be built.

Mountain in the way? No problem! I also lay bedrock and a cobblestone floor underneath. This removes any underground ores, so you might prefer to skip this step (section `# Terraforming!`).

Then I built the four towers (`# Build towers`). The towers are spaced 46 blocks apart from each other. This number comes from the length of 40 for the walls, plus an allowance for the widths of a tower at each end. Finally, I make the four walls to connect the towers.

TIP

If you change the length of the walls or the width of the towers, update this number as well.

You can improve your castle by adding a roof, ladders in the towers, and access to the towers from the top of the walls.

I also changed the `Main` section to test the method:

```
castle = Castle()
castle.make_castle()
```

The Complete Code

Here is the complete code, with the GUI reinstated. To do this I put a new `Castle` button where the old Test button was and hooked it up to the `make_castle` method. I also restored the previous `Main` section.

```
"""
build_castle.py
Build a castle in Minecraft on the Pi!
Brendan

Possible do list:In order to build the castle I'm going to:
#DONE: level out an area for the castle to go on
#DONE: build a tower
#DONE: add battlements with merlonations
#TODO: moat? NAH
"""
```

```
# Imports Section
from mcpi import minecraft, block
from Tkinter import Tk, Button

# Constants Section
WOOL_COLORS = {"white":0,
               "orange":1,
               "pink":2,
               "light_blue":3,
               "yellow":4,
               "light_green":5,
               "light_red":6,
               "dark_gray":7,
               "light_gray":8,
               "middle_blue":9,
               "violet":10,
               "dark_blue":11,
               "brown":12,
               "dark_green":13,
               "red":14,
               "black":15}

AXIS_COLORS = [block.WOOL.withData(WOOL_COLORS['white']),
               block.WOOL.withData(WOOL_COLORS['black']),
               block.WOOL.withData(WOOL_COLORS['light_green']),
               block.WOOL.withData(WOOL_COLORS['yellow'])]
# Axis colors are positive x, negative x, positive z, negative z

BLOCK_AIR = block.AIR
BLOCK_STONE = block.STONE

# Castle Constants
TOWER_HEIGHT = 12
TOWER_WIDTH = 4
WALL_LENGTH = 40
WALL_HEIGHT = 9
WALL_WIDTH = 2
CLEAR_SPACE = 60
CASTLE_SIDE = 51
TOWER_OFFSET = 46
```

```
BEDROCK_HEIGHT = 8
BEDROCK_DEPTH = 10

# Classes Section
class Castle(object):
    """object for interfacing with minecraft to create and destroy
    blocks to create a castle"""
    def __init__(self, parent=None):
        self.parent = parent
        self.view = View(parent)
        self.restore_blocks = {}
        # Hook up callbacks
        self.view.button_drop.config(command=self.drop_block)
        self.view.button_clear.config(command=self.clear_block)
        self.view.button_show_axes.config(command=self.show_axes)
        self.view.button_hide_axes.config(command=self.hide_axes)
        self.view.button_castle.config(command=self.make_castle)

        self.minecraft_connection = minecraft.Minecraft.create()
        self.player = self.minecraft_connection.player

    def drop_block(self, drop_location=None, block=block.GRASS):
        """ drop a block of block (a 2 tuple of block id and data)
        at drop_location (a Vec3 or 3 tuple). If None, use player
        position at x+1."""
        if drop_location is None:
            x, y, z = self.minecraft_connection.player.getPos()
            drop_location = minecraft.Vec3(x+1, y, z)
        self.minecraft_connection.setBlock(drop_location, block)

    def clear_block(self, clear_location=None):
        """ set the block at clear_location (Vec3 or 3 tuple)
        to Air. If None, use player position at x+1"""
        self.drop_block(clear_location, block.AIR)

    def show_axes(self):
        """ Display colored blocks at x and z each axis directions
        to allow player to orient themself.
        """
        x, y, z = self.minecraft_connection.player.getPos()
        locations = [(x+2, y+2, z), (x-2, y+2, z),
                     (x, y+2, z+2), (x, y+2, z-2)]
        self.restore_blocks = {}
        for i, location in enumerate(locations):
            self.restore_blocks[location] = self.minecraft_
             connection.getBlockWithData(location)
            block = AXIS_COLORS[i]
            self.drop_block(location, block)

    def hide_axes(self):
        """ Remove axes previously placed. Restore original blocks if
        saved """
        for location, block in self.restore_blocks.items():
            self.drop_block(location, block)

    def set_blocks(self, x, y, z, w, h, l, block_id, block_data):
        """ Set blocks as a box of width, height and
        length w, h and l starting at location x, y z
        """
        x2, y2, z2 = x+w, y+h, z+l
        self.minecraft_connection.setBlocks(x, y, z,
                                            x2, y2, z2,
                                            block_id, block_data)

    def test_button(self):
        """ A short method to test whether the new
        set_blocks method is working"""
        self.player = self.minecraft_connection.player
        x, y, z = self.player.getPos()
        w, h, l = (9, 1, 4)
        self.set_blocks(x+1, y, z, w, h, l, block.GLASS, 0)

    def make_tower(self,
                   location=None,
                   tower_height=TOWER_HEIGHT,
                   tower_width=TOWER_WIDTH,):
        """Make a Tower at location (or player pos)"""
        if location is None:
            x, y, z = self.player.getPos()
            x = x+1 # So player is outside tower.
        else:
            x, y, z = location

        w, h, l = tower_width, tower_height, tower_width
        block_data = 0
        # Tower
        self.set_blocks(x, y, z,
                        w, h, l, BLOCK_STONE, block_data)
        # Battlement at top
        w, h, l = tower_width+2, 2, tower_width+2
        self.set_blocks(x-1, y+tower_height, z-1,
                        w, h, l, BLOCK_STONE, block_data)
        # Hollow out battlement
        w, h, l = tower_width, 1, tower_width
        self.set_blocks(x, y+tower_height+1, z,
                        w, h, l, BLOCK_AIR, block_data)
        # Add further hollows for merlonations/crenellations
        # cut three times on x and three times on z:
        for i in range(3):
            self.set_blocks(x+2*i, y+tower_height+2, z-1,
                            0, 0, 6, BLOCK_AIR, block_data)
            self.set_blocks(x-1, y+tower_height+2, z+2*i,
                            6, 0, 0, BLOCK_AIR, block_data)
        # Hollow out tower
        w, h, l = tower_width-2, tower_height+1, tower_width-2
        self.set_blocks(x+1, y, z+1, w, h, l, BLOCK_AIR, block_data)

    def make_wall(self,
                  location=None, orientation="x",
                  wall_length=WALL_LENGTH,
                  wall_height=WALL_HEIGHT,
                  wall_width=WALL_WIDTH):
        """ make a wall at location (or player pos) parallel to
        x axis by default otherwise parallel to z axis """

        block_data = 0
```

```
if location is None:
    x, y, z = self.player.getPos()
    x = x+1 # So player is outside wall.
else:
    x, y, z = location

w, h, l = wall_length, wall_height, wall_width # wall base
w1, h1, l1 = wall_length, 2, wall_width+2 # battlement
w2, h2, l2 = wall_length, 1, wall_width # hollow out
    battlement
xoffset, zoffset = 0, -1

if orientation != "x": # swap dimensions
    w, l = l, w
    w1, l1 = l1, w1
    w2, l2 = l2, w2
    xoffset, zoffset = zoffset, xoffset


# Wall support
self.set_blocks(x, y, z, w, h, l, BLOCK_STONE, block_data)

# Add Battlement
self.set_blocks(x+xoffset, y+wall_height, z+zoffset,
                w1, h1, l1, BLOCK_STONE, block_data)
# hollow out battlement
self.set_blocks(x, y+h+1, z,
                w2, h2, l2, BLOCK_AIR, block_data)

# create crenellations/merlonations
if orientation == "x":
    for i in range(1, wall_length, 2):
        self.set_blocks(x+i, y+h+h1, z-1,
                        0, 0, 4, BLOCK_AIR, block_data)
else:
    for i in range(1, wall_length, 2):
        self.set_blocks(x-1, y+h+h1, z+i,
                        4, 0, 0, BLOCK_AIR, block_data)
```

```
def make_castle(self, location=None):
    """ Create a castle at the specified location, or at the
        player's
     position if none specified"""
    if location is None:
        x, y, z = self.player.getPos()
        x = x+1 # So player is outside
    else:
        x, y, z = location

    Vec3 = minecraft.Vec3 #convenience assignment

    # Terraforming!
    # Clear the area
    self.set_blocks(x-(TOWER_WIDTH+1), y, z-(TOWER_WIDTH+1),
                    CLEAR_SPACE, CLEAR_SPACE, CLEAR_SPACE,
                    block.AIR.id, 0)
    # lay down some bedrock underneath the castle
    self.set_blocks(x, y-BEDROCK_DEPTH, z,
                    CASTLE_SIDE, BEDROCK_HEIGHT, CASTLE_SIDE,
                    block.BEDROCK.id, 0)
    # one layer of cobblestone
    self.set_blocks(x, y-1, z,
                    CASTLE_SIDE, 0, CASTLE_SIDE,
                    block.COBBLESTONE.id, 0)

    # create all the towers at the location
    self.make_tower(Vec3(x, y, z))
    # now make another two towers - each at the end of each wall
    # assumes length of wall is 40 (actually 41 bc of 0 basing)
    # and width of towers is 4 (actually 5)
    self.make_tower(Vec3(x+TOWER_OFFSET, y, z))
    self.make_tower(Vec3(x, y, z+TOWER_OFFSET))
    # make last tower is diagonally opposite the first
    self.make_tower(Vec3(x+TOWER_OFFSET, y, z+TOWER_OFFSET))

    # now make two walls to attach to the first tower.
    # Starting point of each wall needs to be offset
    # by (width of tower)+1 blocks in the x axis for the
    # first wall and in the z axis for the second wall.
```

```
        self.make_wall(Vec3(x+TOWER_WIDTH+1, y, z+1))
        self.make_wall(Vec3(x+1, y, z+TOWER_WIDTH+1),
                       orientation="z")

        # repeat the walls above, but offset them by the
        # tower offset to the z and x axes
        # This is so that the end of the wall meets the side
        # of the tower
        self.make_wall(Vec3(x+TOWER_WIDTH+1, y, z+1+TOWER_OFFSET))
        self.make_wall(Vec3(x+1+TOWER_OFFSET, y, z+TOWER_WIDTH+1),
                       orientation="z")

class View(object):
    """Interface to Castle object"""
    def __init__(self, parent=None):
        self.parent = parent
        self.button_drop = Button(self.parent, text='drop')
        self.button_clear = Button(self.parent, text='clear')
        self.button_drop.grid(row=0, column=0)
        self.button_clear.grid(row=0, column=1)
        self.button_show_axes = Button(self.parent, text='Axes +')
        self.button_show_axes.grid(row=1, column=0)
        self.button_hide_axes = Button(self.parent, text='Axes -')
        self.button_hide_axes.grid(row=1, column=1)
        self.button_castle = Button(self.parent, text="Castle")
        self.button_castle.grid(row=2, column=0)

# Functions Section


# Main Section
if __name__ == "__main__":
    root = Tk()
    castle = Castle(parent=root)
    root.mainloop()
```

Summary

This project is all about interfacing with another program. You

- Connected to Minecraft from Python.
- Posted a chat message to Minecraft.
- Created and destroyed blocks of different kinds.
- Got the player's location and moved it around.
- Blew stuff up with TNT.
- Created a helper function to see the axis directions in Minecraft.
- Created a helper function to set blocks using relative, rather than absolute, coordinates.
- Used the `grid` geometry manager.
- Built a whole castle using Python.