Appendix H: Ruby Environment and Scripting

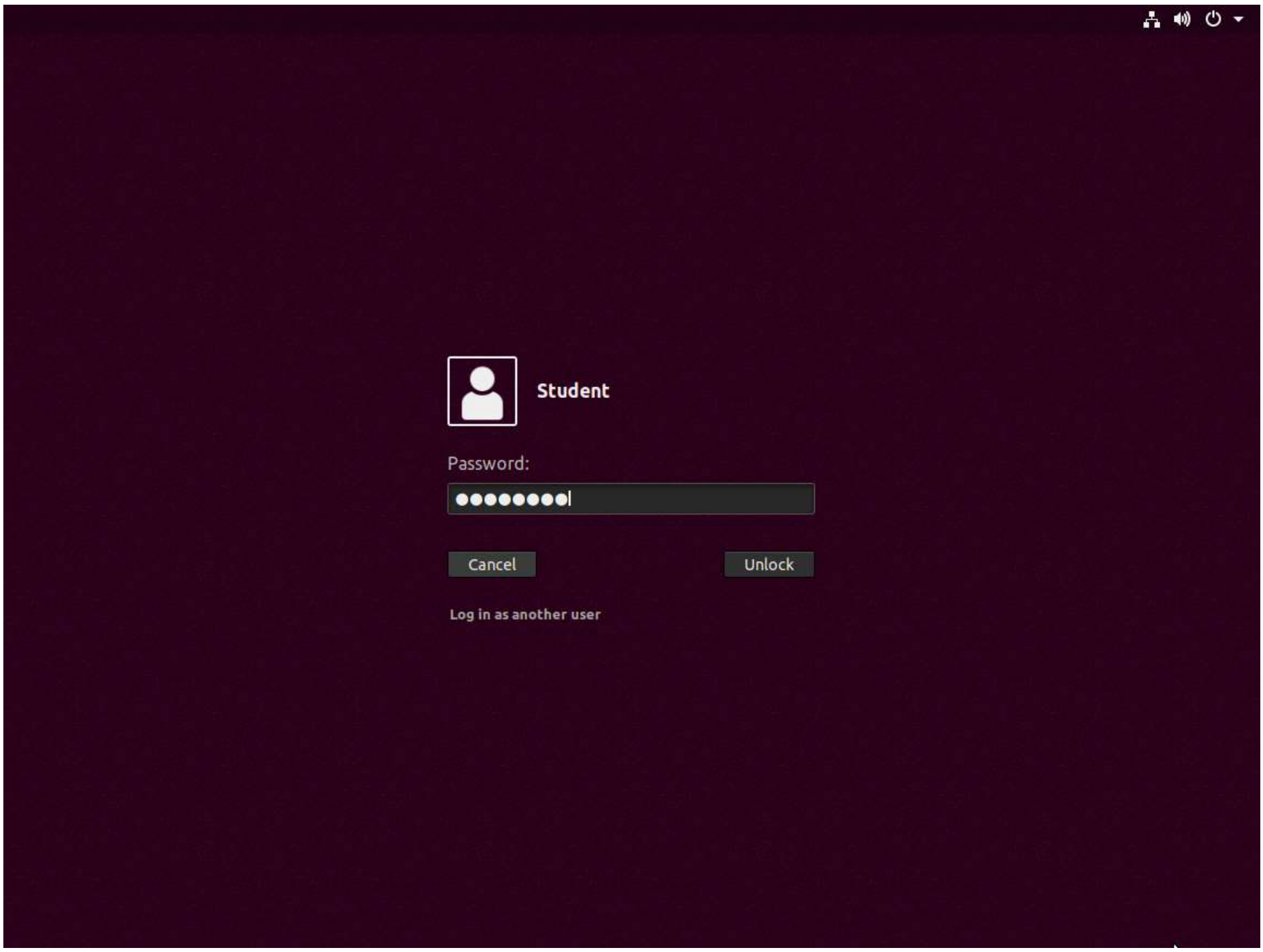
Exercise 1: Basic Ruby Programming

Objective

- Create basic Ruby scripts and extract data and explore the features and components of Ruby.

Lab Tasks

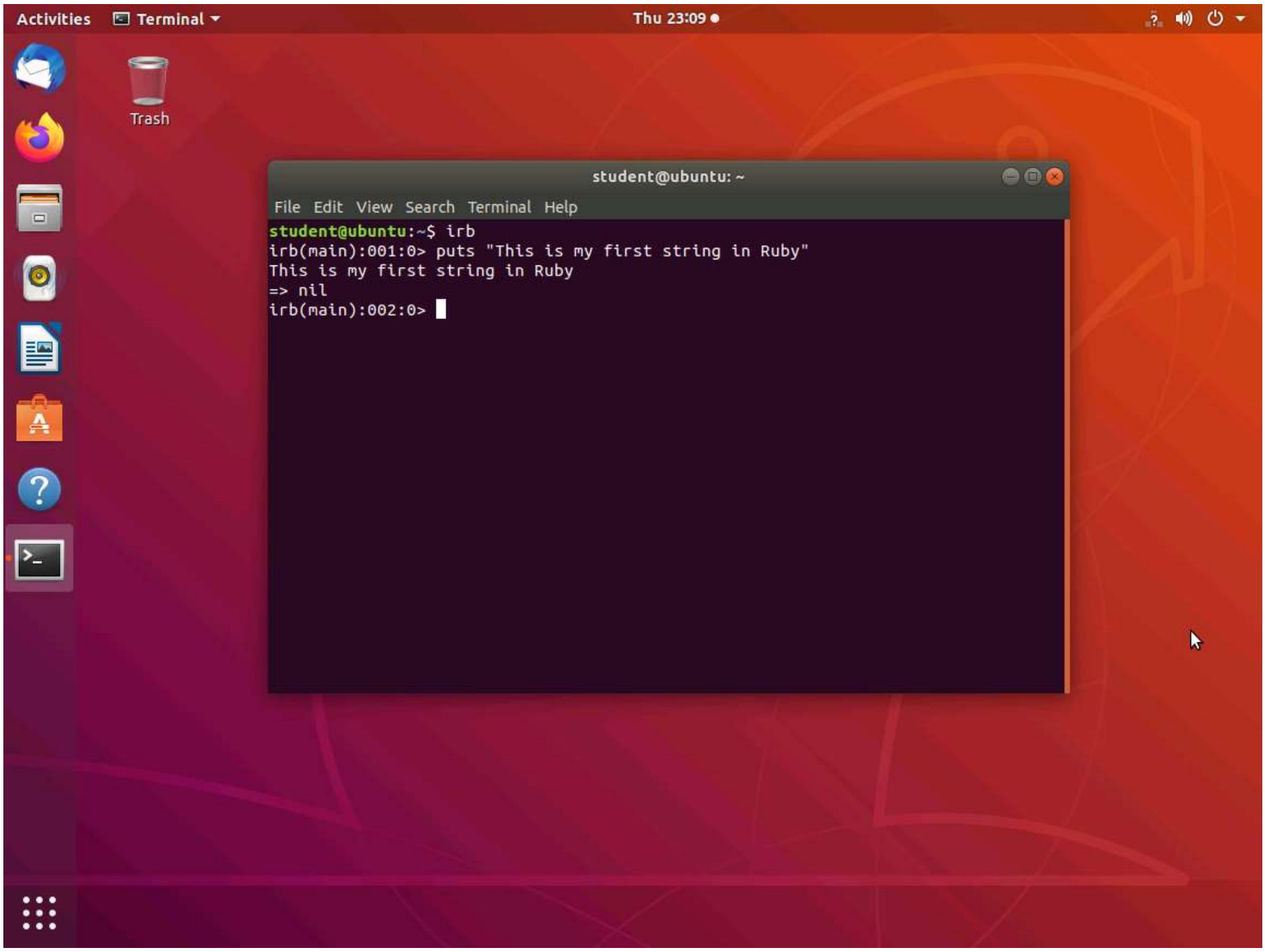
1. Click **CPENT Ruby-Machine** to go to the Ruby-Machine. Click **Student** profile, type **password** in the Password field and press **Enter**.



2. Create your first simple program, open a terminal window, and enter **irb**. Then, enter the code shown next:

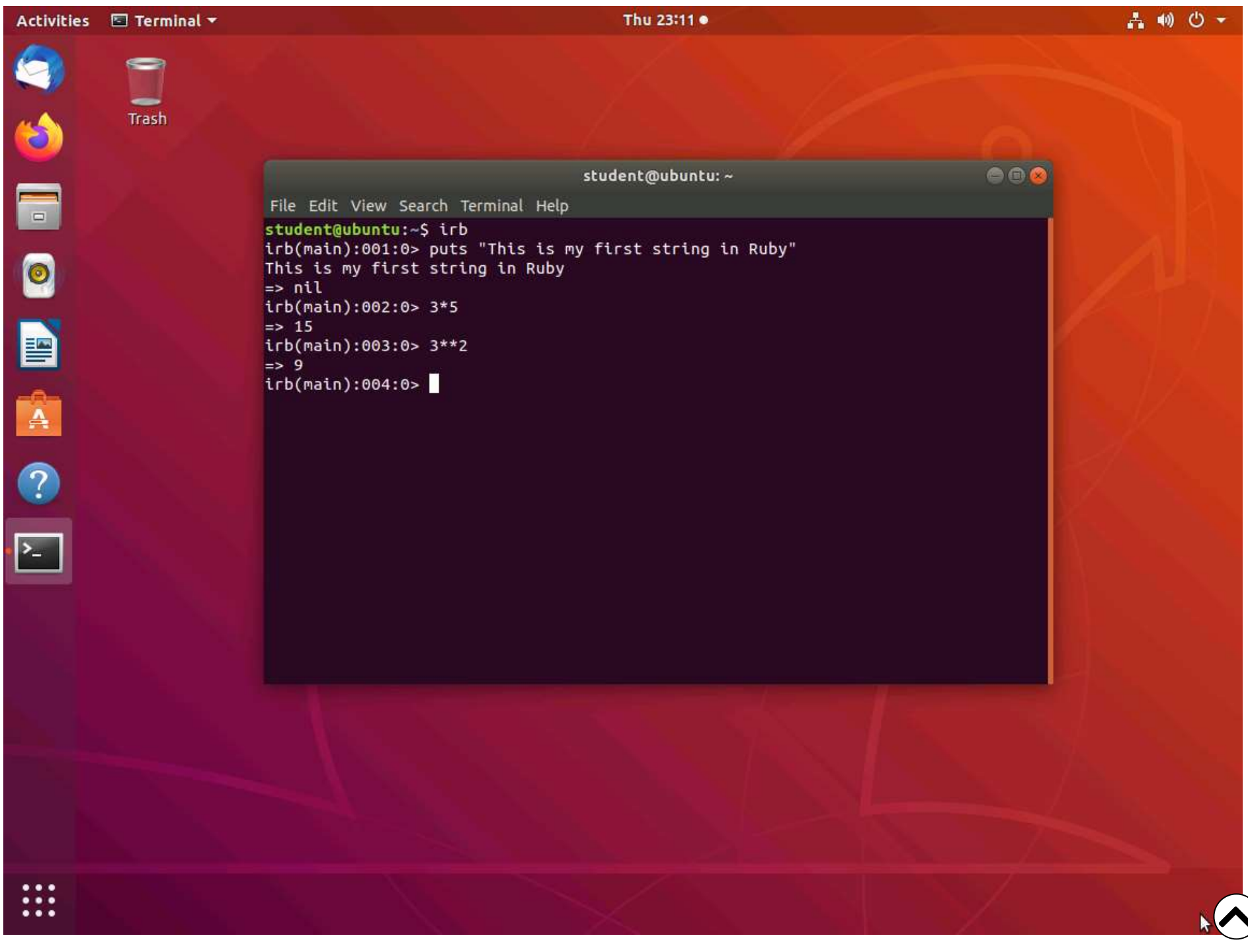
   puts "This is my first string in Ruby"

3. An example of the output from the command is shown in the screenshot.



4. This is a simple method of creating a program to create a string.

5. Note that the irb will perform calculations as entered. Enter **3*5** in the irb window and press **Enter**, and then enter **3**2** and press **Enter**. The output from both commands is shown in the screenshot .



6. As the screenshot shows, the is exponential. Next, explore the different extension capabilities of Ruby. Exit irb.
7. Enter **irb --prompt-mode simple**. Then, enter the following:

    a. char “A”

    b. char = “A”

    c. char.unpack(“C”)

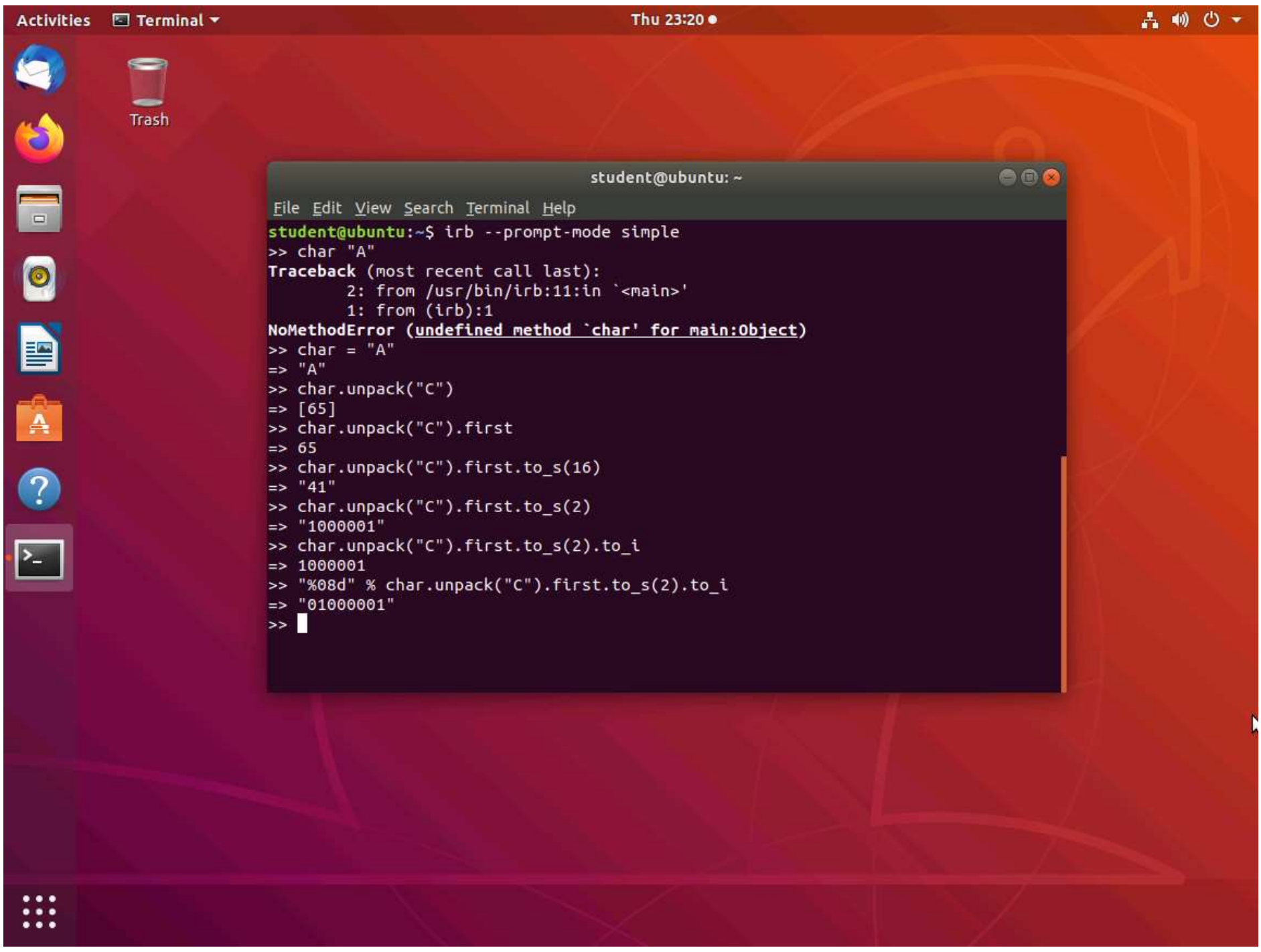
    d. char.unpack(“C”).first

    e. char.unpack("C").first.to_s(16)

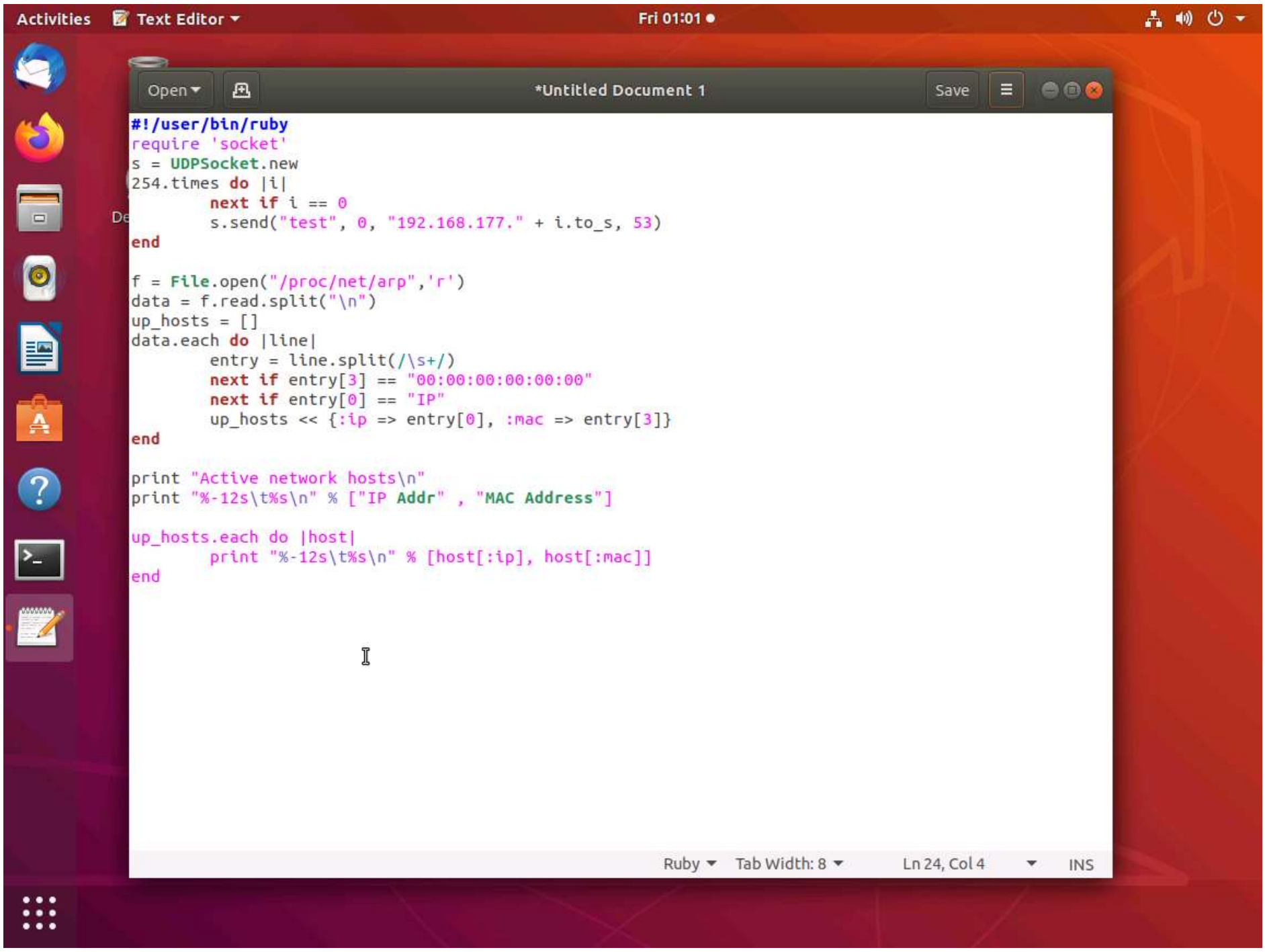
    f. char.unpack("C").first.to_s(2)

    g. char.unpack("C").first.to_s(2).to_i

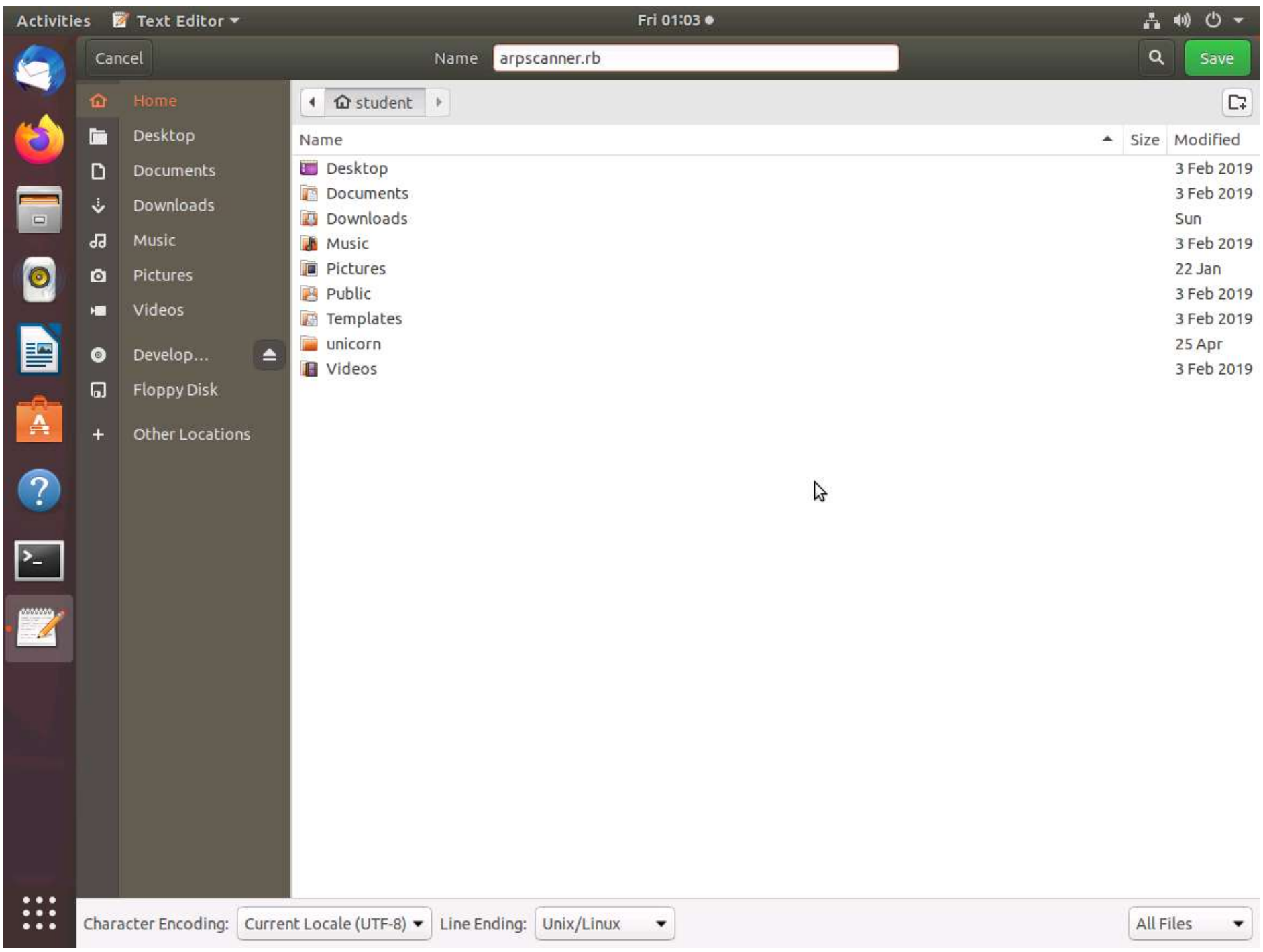
    h. "%08d" % char.unpack("C").first.to_s(2).to_i



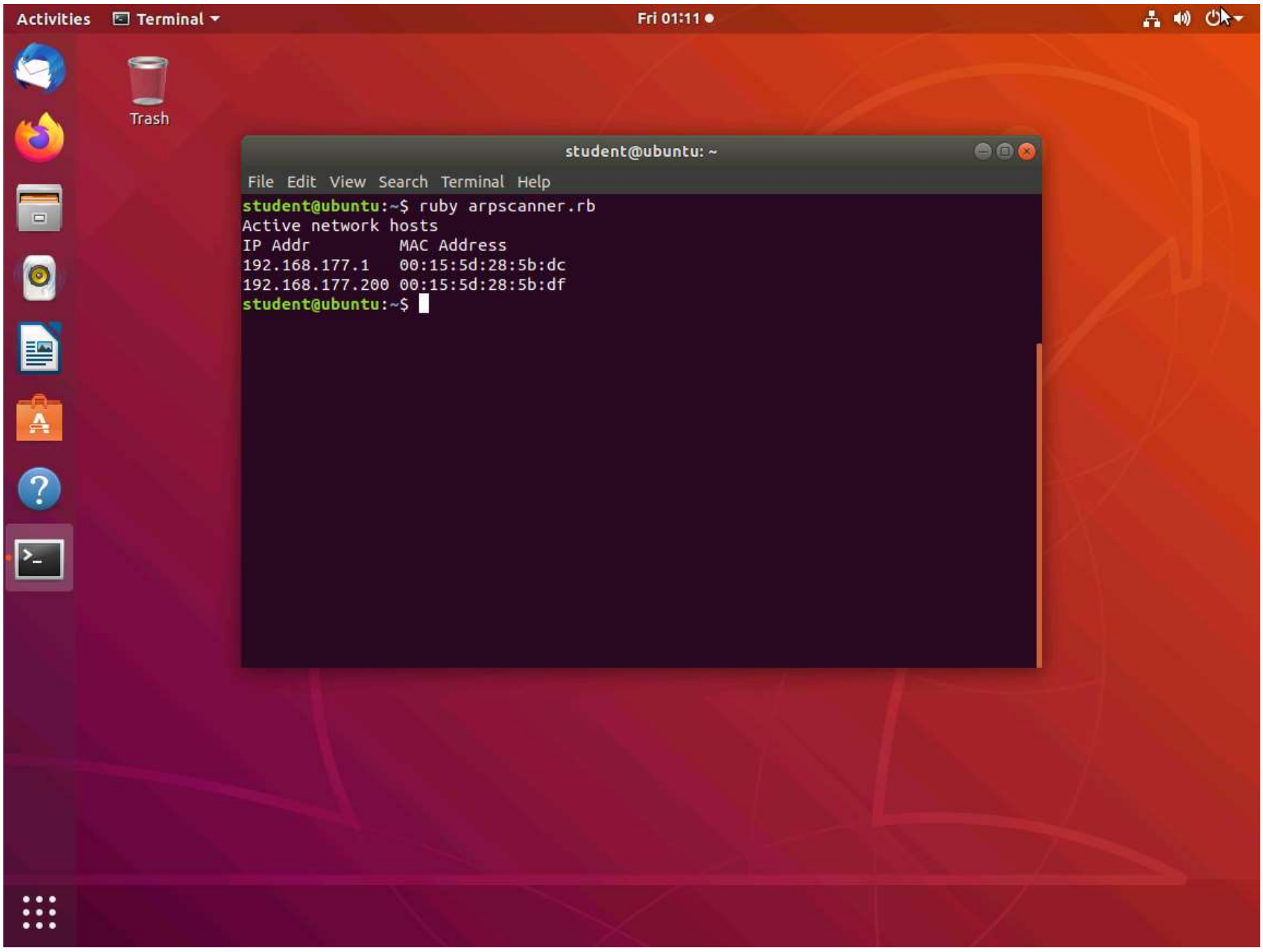
8. Now that you have explored these different extensions, explore another example.
9. Your objective is to build a simple Address Resolution Protocol (ARP) scanner in Ruby. Note that Ruby does not have an ARP module, so you can use the access to the /proc folder to grab data.
10. In the editor of your choice, enter the following code:



11. Once you have the code entered, save the file as **arpscanner.rb**.



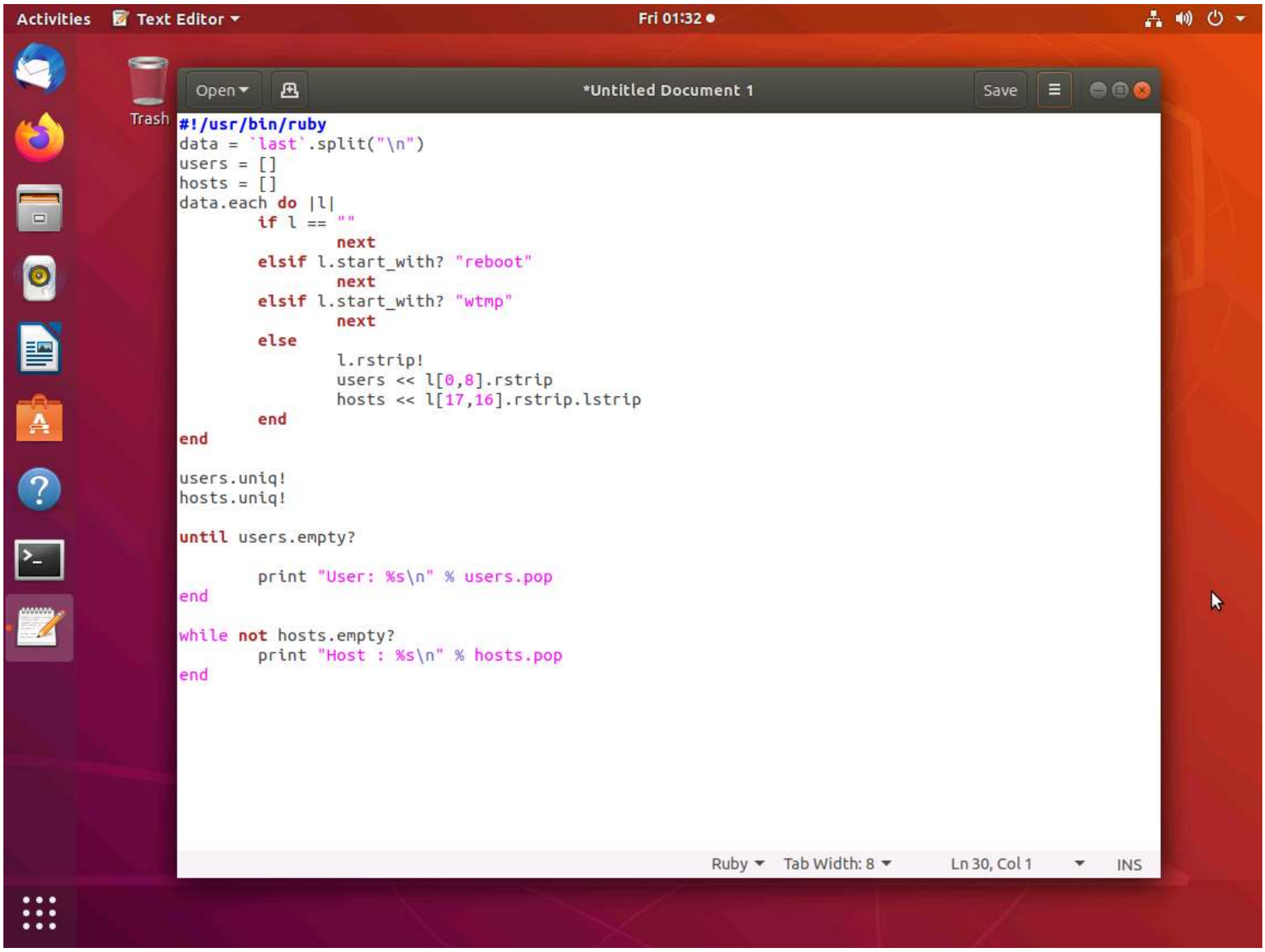
12. Run arpscanner.rb with the following command:



13. The code is explained below.

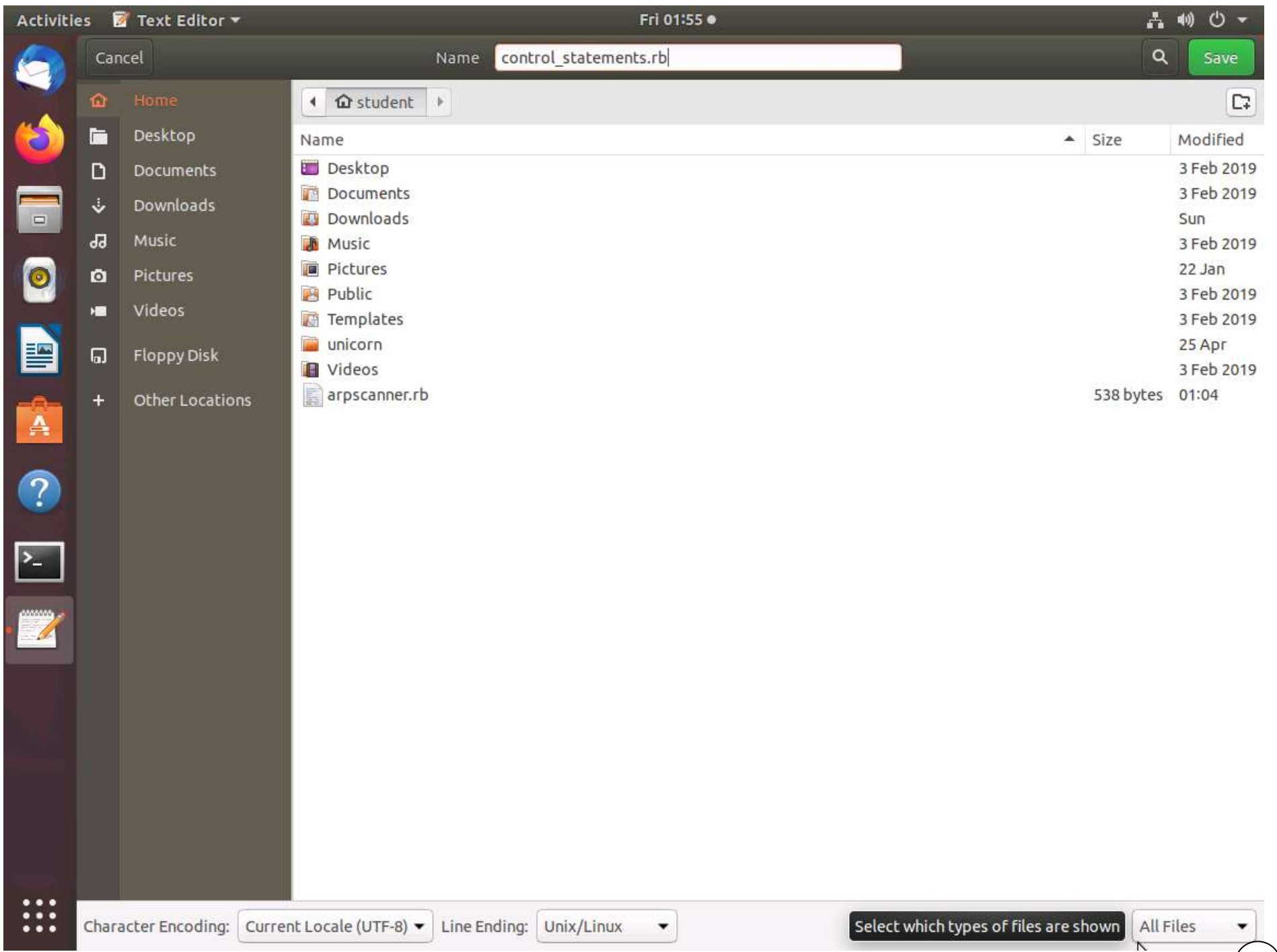
    - The code starts with standard calls to create a socket. Then, use the send capability to send data into port User Diagram Protocol (UDP) 53, since you need to retrieve ARP responses, which are stored in the s type of socket.
    - If the host is up, the ARP entry will be added to the OS's ARP cache and will have an Internet provider (IP) address assigned with it. Browsing this ARP cache shows which hosts are up.
    - To enumerate the IP space, use the times operator of the Fixnum class—the class for numbers that have not been assigned to a variable. The times operator iterates from 0 to 254, and assigns the current value to our i variable. If i is equal to 0, skip it, as you only want the usable IP addresses in the subnet. For the subsequent iterations, append it to the subnet 192.168.177.
    - Once the UDP packets are sent, make use of the proc file system to directly access the system's ARP table. For this step, you need to run a Linux or UNIX system.
    - Because you have already split the output on newline characters, you will have an array, with each value being an ARP entry. Create a new empty array called up_hosts using the array syntax of []. Iterate through each line, each time assigning the line from the ARP cache to the line variable.
    - Finally, if you have a good ARP entry, create a hash using the {} syntax and use the :ip and :mac symbols as the keys for the parsed values. Assign each key the value with the => operator. The two fields you want are the IP address, which is the first element in the field, and the MAC address, which is the fourth element. Because arrays start with an index of 0, count up from 0, yielding the array fields of 0 and 3. Now that you have the hash, append it to the end of the up_hosts array using the << append syntax, resulting in the hash appearing as the last element of that array.
    - You now have an array of hashes containing the hosts that were found via the ARP resolution. Print them in an easy-to-view format. Start by printing the header using the print format string. Use %s to indicate a string, and %-12s to indicate a 12-character string that is left-aligned.

14. Next, explore the control statements. Open a new text editor and enter the code, as shown in the screenshot.

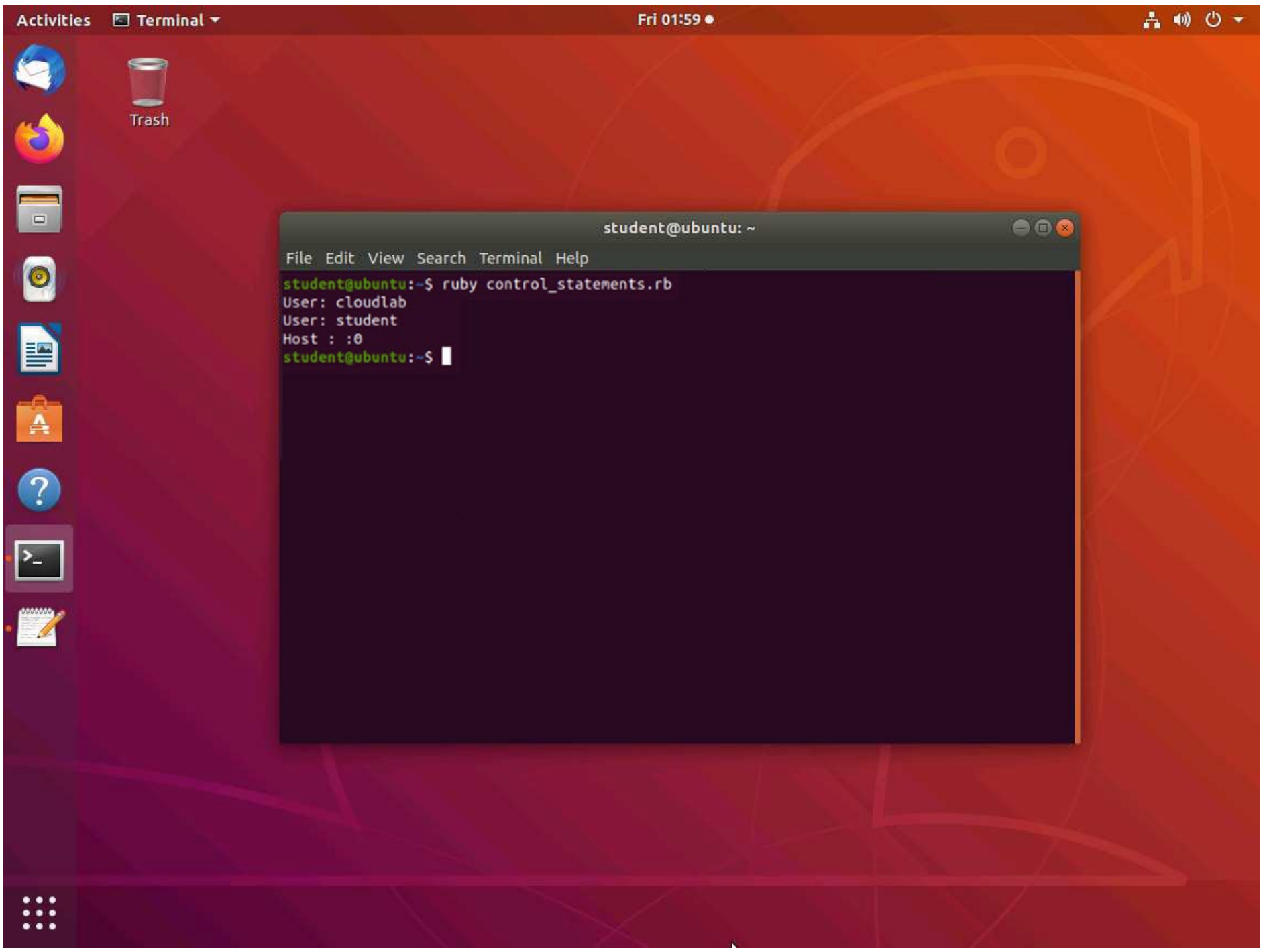


15. This code is to show an example of the “if else ” statements, which mostly work as the others do.

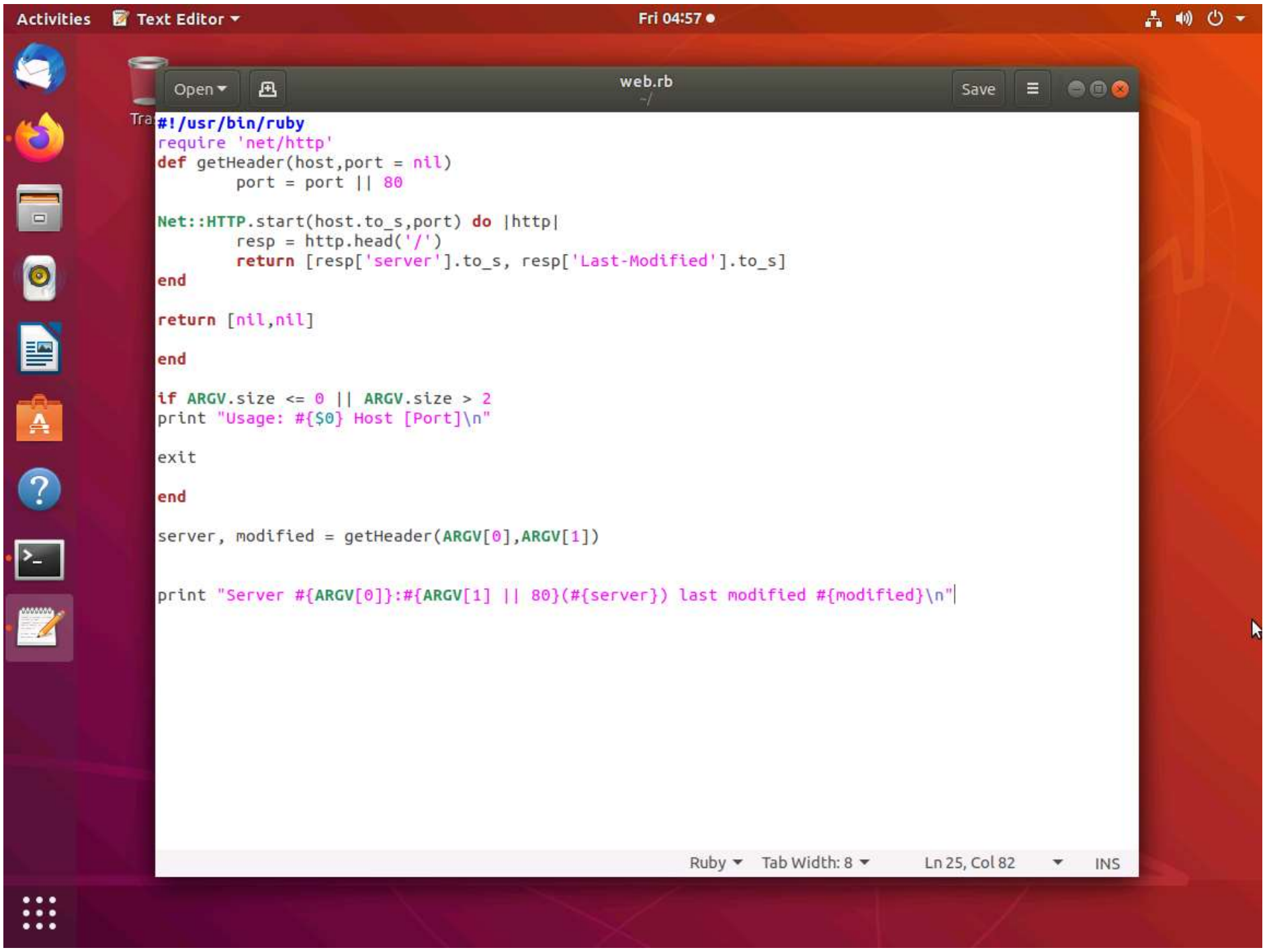
16. Save the file as **control_statements.rb**.



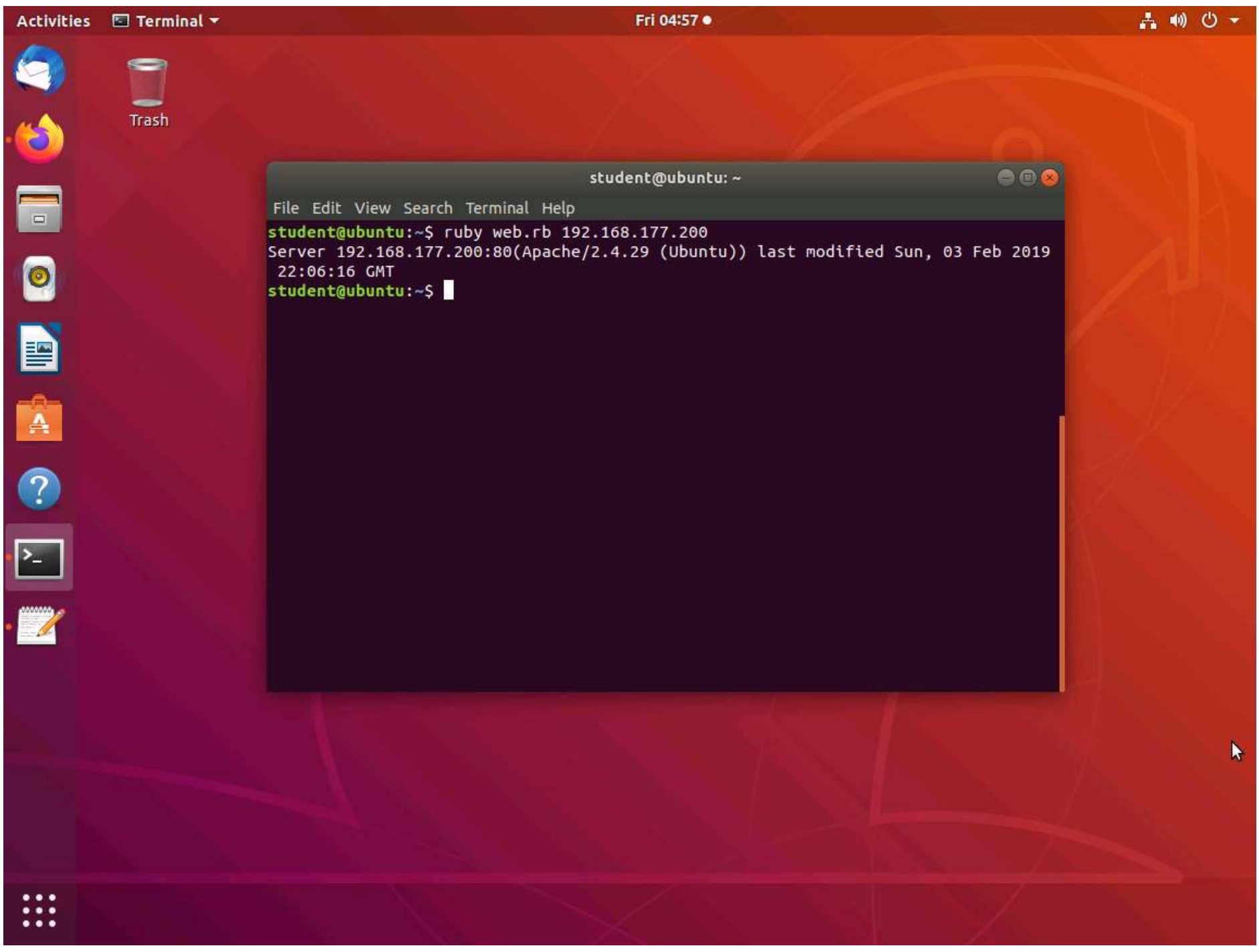
17. Run the **ruby control_statements.rb** command, which lists the last login attempts for the system. Place the last command in grave accents or back quotes (`) in order to instruct Ruby to run the command within the OS, and then return stdout to the application. The execution returns a string; split it based on newlines and assign to the data array.



18. Create a script to connect and grab the web banner; enter the code that is shown in the screenshot. Save the file as **web.rb**.



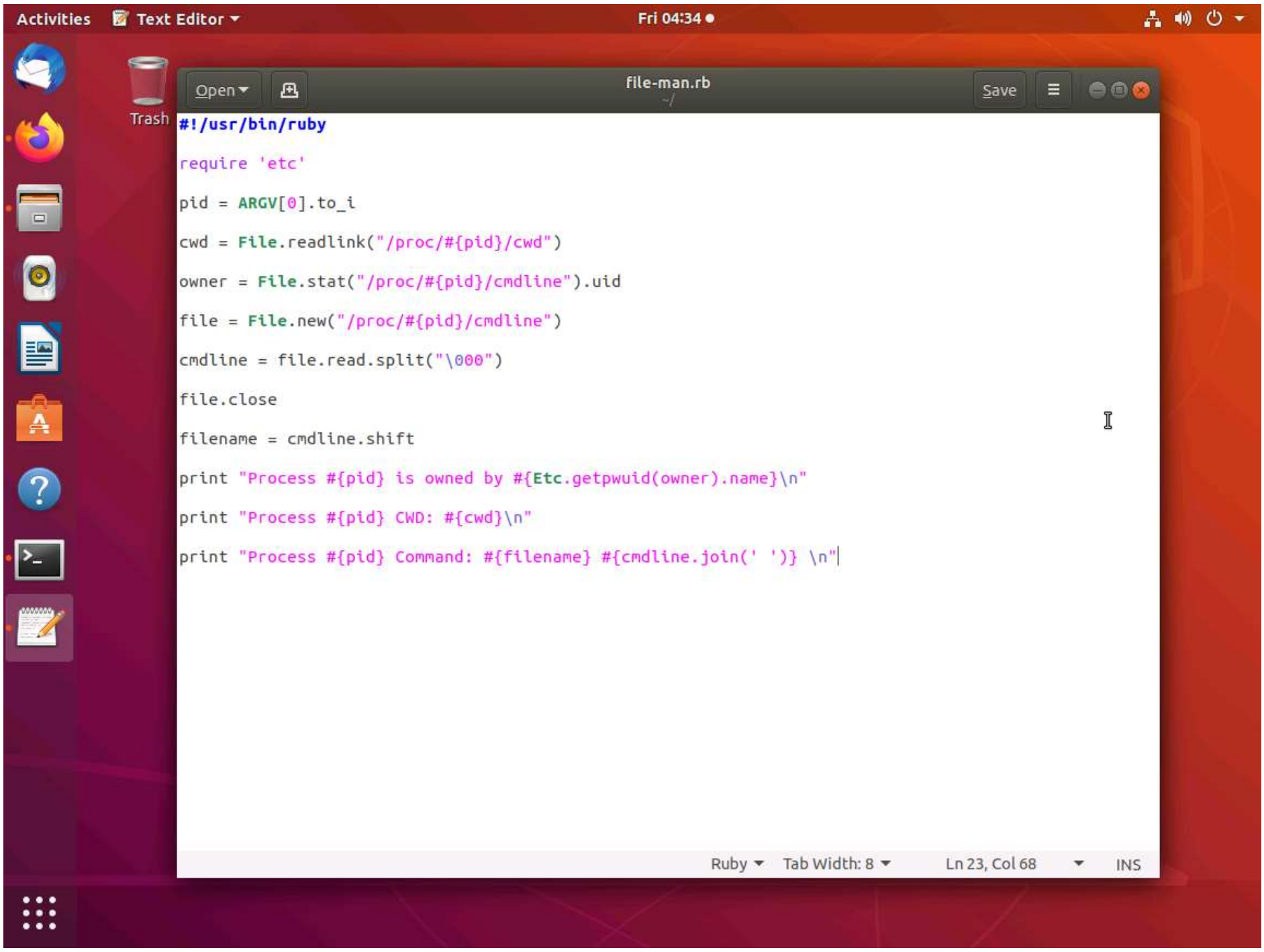
19. Run the **ruby web.rb** command. An example of the results when the script is run is shown in the screenshot.



20. The last script in this lab will be the File Manipulation script to extract information about the file system.

21. Note that the Ruby interpreter is not as robust as some of the other languages by default.

22. In the editor of your choice , enter the following code and save it as **file-man.rb**.

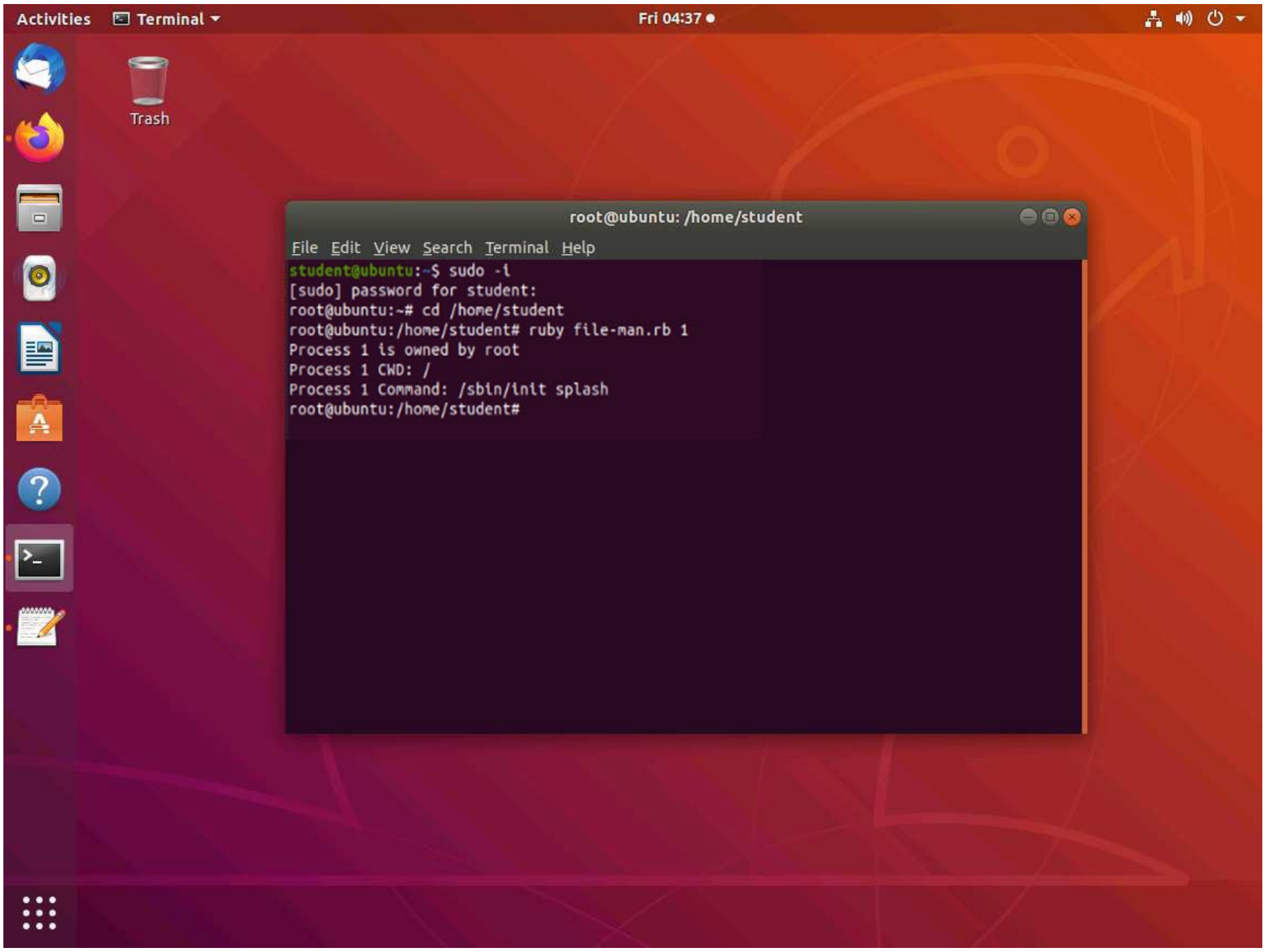


23. When you run this command, ensure the root. Enter **sudo -i**.

24. Then, change back to the "home for student," by entering **cd /home/student**.

25. Enter **ruby file-man.rb 1**.

26. First reference the PID number 1, and then dump the details.



27. This concludes the lab exercise.

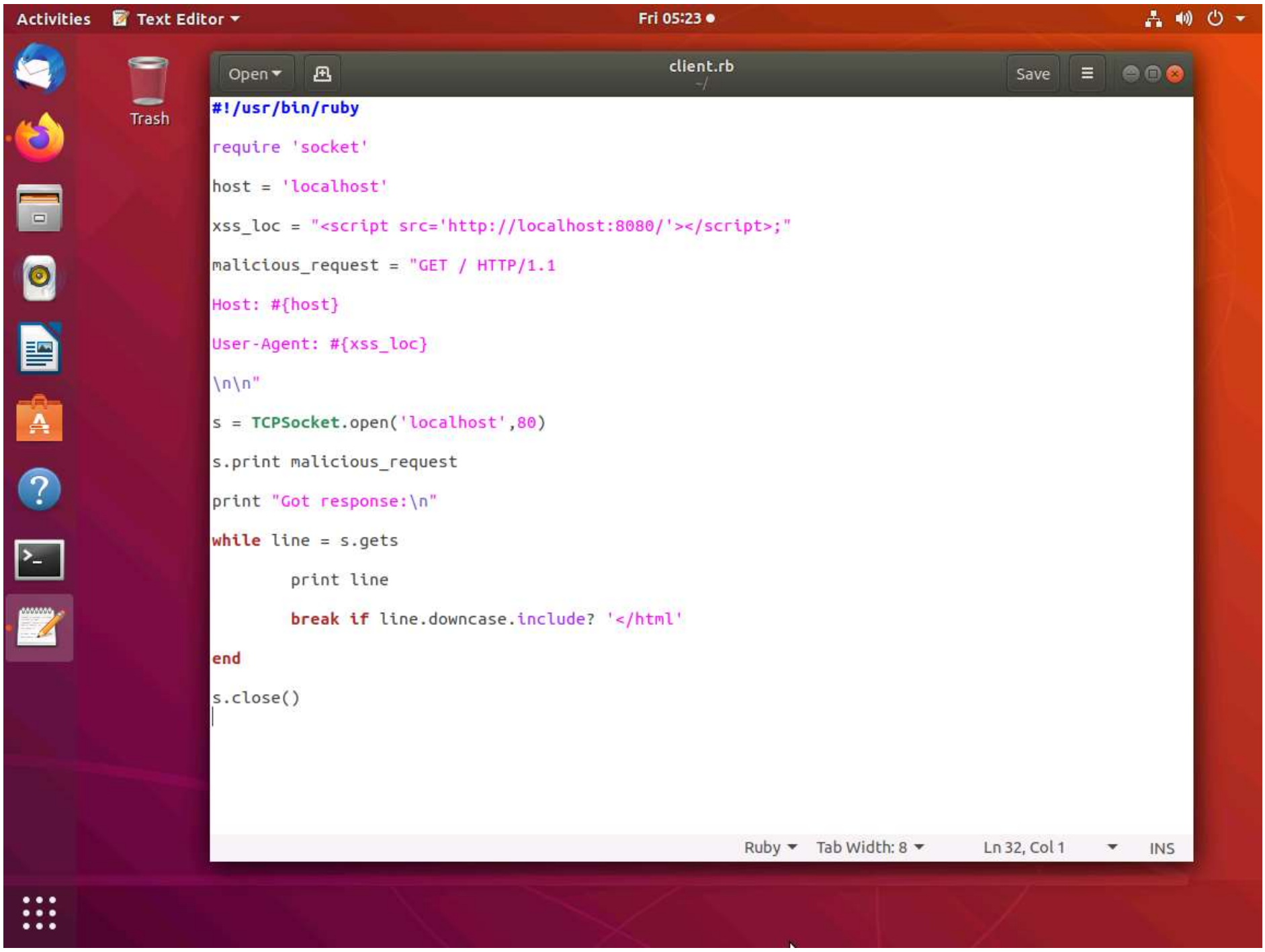
Exercise 2: Basic Ruby Socket Programming

Objective

- Create network connection Ruby scripts and extract data and explore the features and components of Ruby sockets.

Lab Tasks

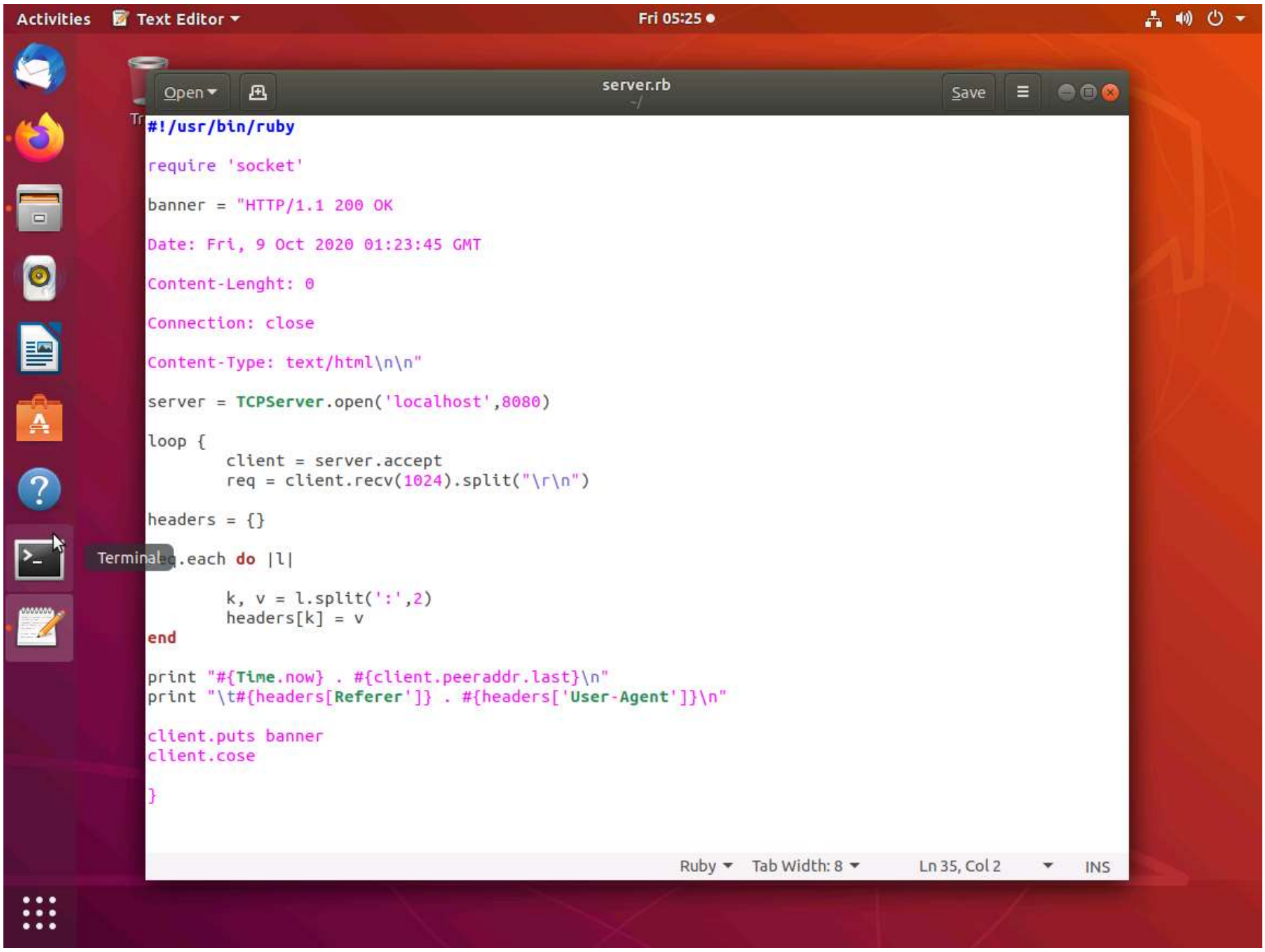
1. For the next lab, build your client and server. Enter the code shown in the screenshot and save it as **client.rb**.



2. Set up your Ruby script and include the socket module. Then, define the host to connect to, and insert the malicious HTML. In order for the HTML to be valid, use single quotes. Apache will escape double quotes, but not single quotes. Lastly, set up your malicious request. Your request issues a GET command to the server requesting the root page of the Web server using HTTP 1.1 syntax. Include the hostname of your target server and add a User-Agent field with the malicious HTML in it.

3. Then, create a socket object using the TCPSocket open method using the hostname and port for the target server. The socket will behave like any generic socket. Regardless of the type of socket you open, these commands will be identical. Use the socket's print method to send data to the remote server. In this case, you are sending the malicious HTTP request to the server. Observe what comes back, so you know if your request worked. To do this, set up a loop that retrieves data from the remote host one string at a time using the "gets" method and print it to the screen until you receive a close HTML tag. Once you receive the close HTML tag, stop reading and close the connection.

4. Next, enter your server code and save it as **server.rb**.



5. Create a server that will listen on port 8080. When an incoming connection occurs, record the request information and return something to the browser to let it know there is no content. This allows the web page that called your script to finish rendering. Next, create a listening socket. Use the TCPServer class, as it will allow you to accept new connections as socket objects.

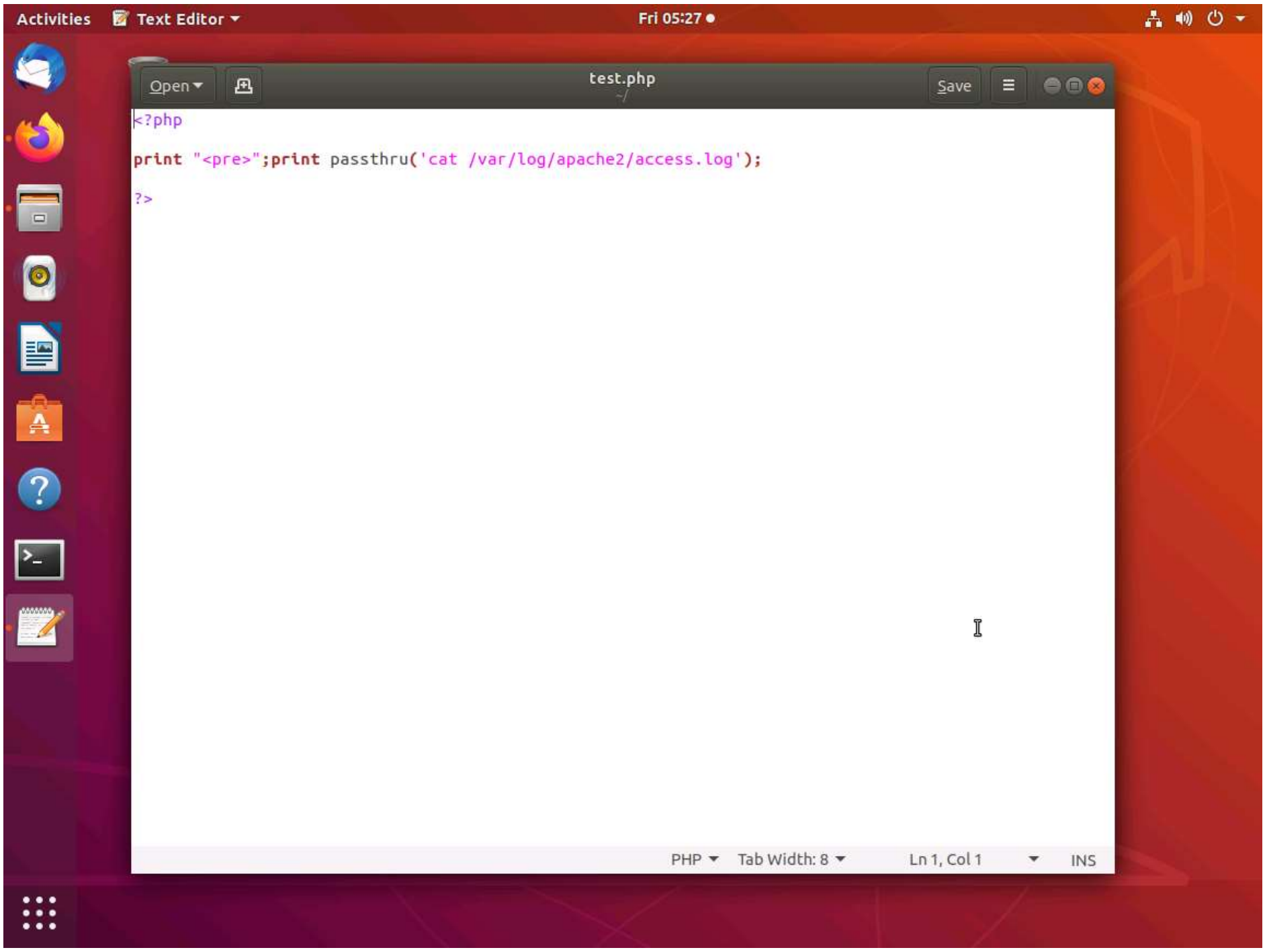
6. Include your socket module and create a banner that returns a 200 OK message. This tells the web browser that the query was accepted smoothly. Next, set up headers to indicate there is no content, providing date and content type information to make the browser happy. End with two newlines, so that the browser knows your message is finished. The banner message is what you provide to any connection regardless of what is requested. The interest here is who requested your page, and from where they were referred to you. Once you know this information, move on. The TCPServer open method takes two options: the host to bind the socket to and the port.

7. Next, create a loop that will accept incoming requests. Process the request for the information you want to log, and send the client on its way. Once you have the information you want, print it to the screen for logging purposes and wait for the next connection. To do this, create an endless loop waiting for incoming connections. The only way to stop the script will be to issue a Ctrl+c sequence.

8. Create an endless loop and use the TCPServer accept method to accept new connections. The script will hang until a new connection comes in.

9. Create a new hash called headers, and process the request lines by assigning each line in the loop to the line variable. Each line of the header will be a string containing a key–value pair separated by colons. Use the split method of the String class to generate key–value pairs and assign them to k and v, respectively. Then, use the hash to store the key–value pair, so you can directly access only the fields needed. Once the headers are parsed, print the time and remote address of the client

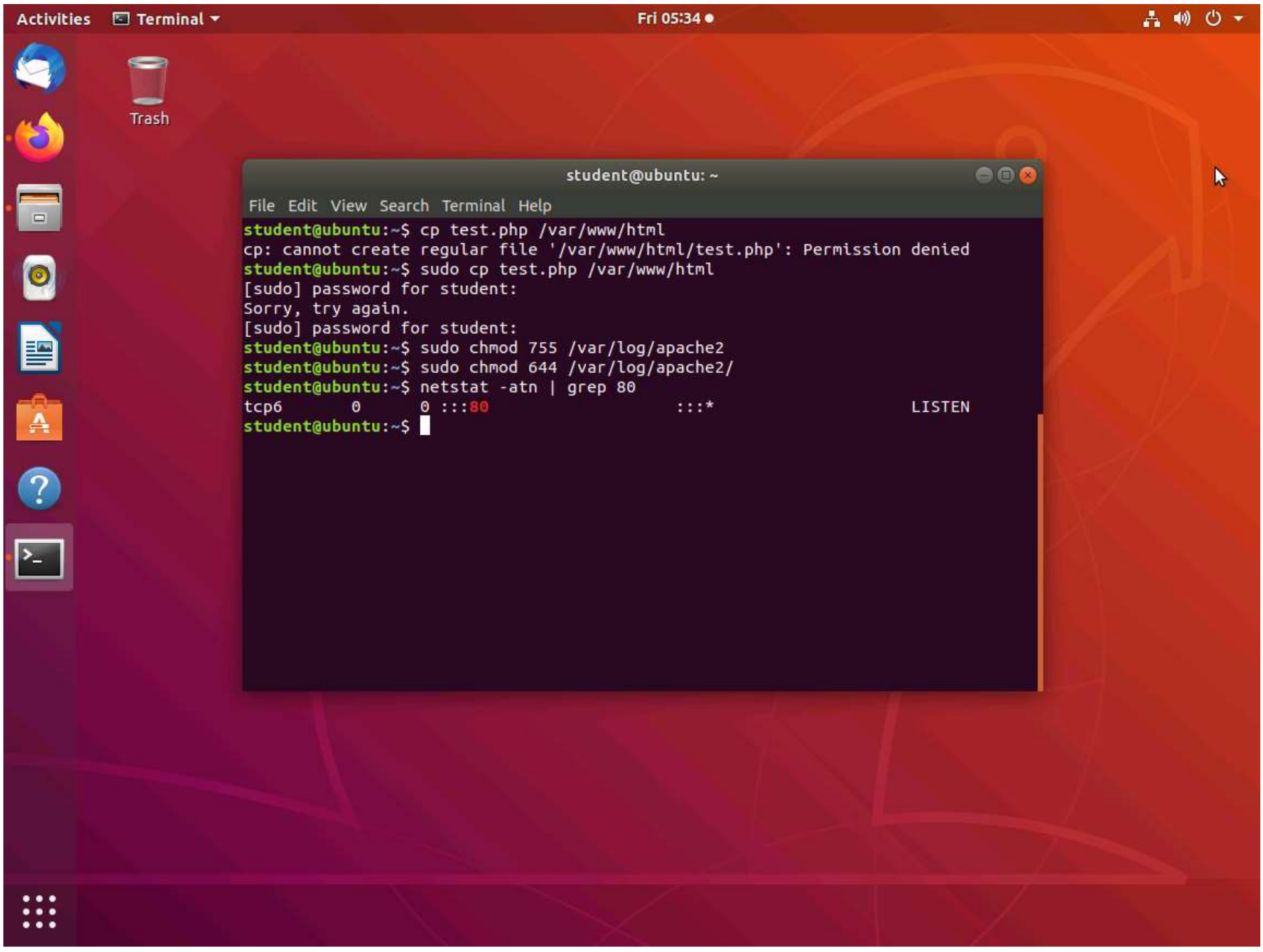
10. Create a piece of PHP code that will allow you to run the script; enter the following and save it as **test.php**:



11. This PHP code prints out a pre-HTML tag to indicate that the output has been preformatted. It then executes the cat command on the web access log, and uses the PHP passthru command to print the output to the screen. This is a quick and easy log viewer with a critical vulnerability. It does no output cleansing. If you create a script that injects HTML into the log, the PHP script will happily render it in the unsuspecting administrator's web browser. In order for this to work, the system administrator has to also change the mode on the Apache log directory to be world-readable.

12. Copy the file into the folder **/var/www/html**, and then enter the commands **chmod 755 /var/log/apache2** and **chmod 644 /var/log/apache2/**. Verify that the web server is running by entering **netstat -atn | grep 80**.

Note: If Apache is not running, enter **service apache2 start**.



13. The sequence of events for the attack is as follows:

    a. Client code

    i. Open a connection to the vulnerable web server

    ii. Make a request for the page with a malicious script in the User-Agent field

    iii. Retrieve and print the output from the request to the screen

    iv. Wait for the administrator to run the script

    b. Server code

    i. Open a socket, and wait for the administrator to run the script

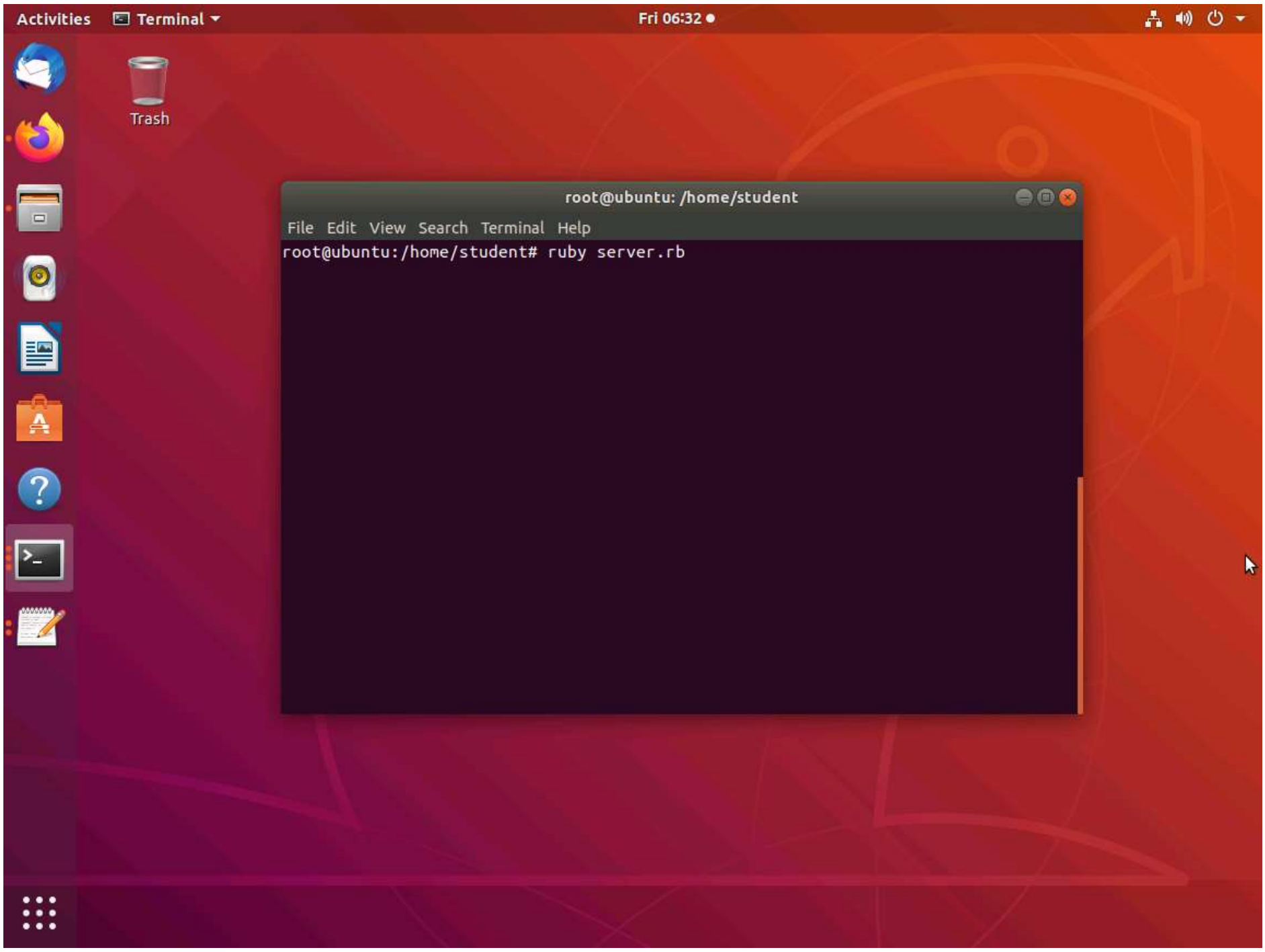
    ii. Receive the HTTP request from the administrator's web browser, and print the data to the screen

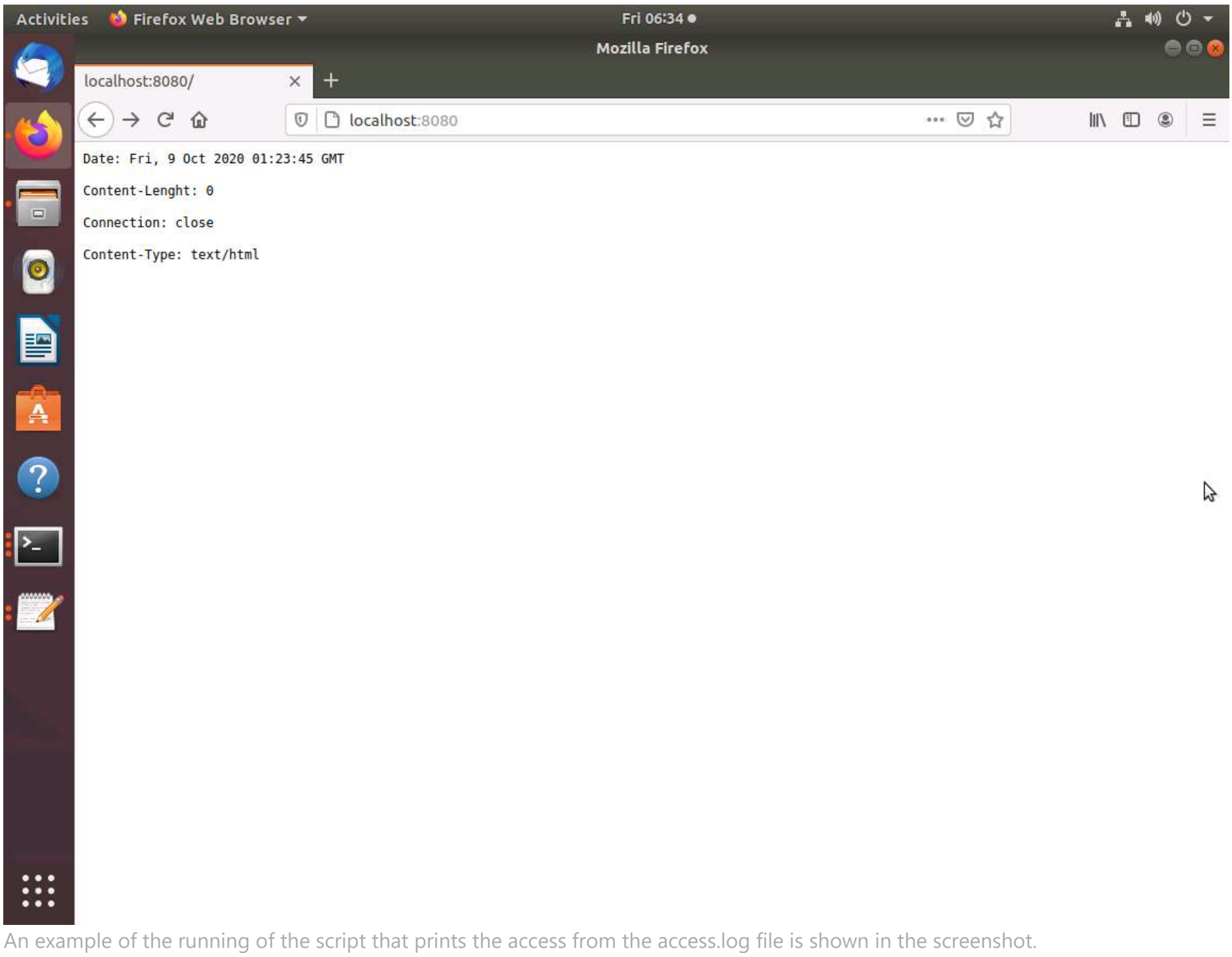
    iii. Return a 200 code to the administrator's web browser, indicating all is okay

14. When your administrator next views the logs, the browser will see the malicious HTML and execute a request in the background to grab a script from localhost on port 8080. You could deliver almost any script to the browser. For the purpose of this example, you will need a server that logs the connection, so you know what was executed, and then return an empty message. This way, the browser continues rendering and the sys-admin is none the wiser.

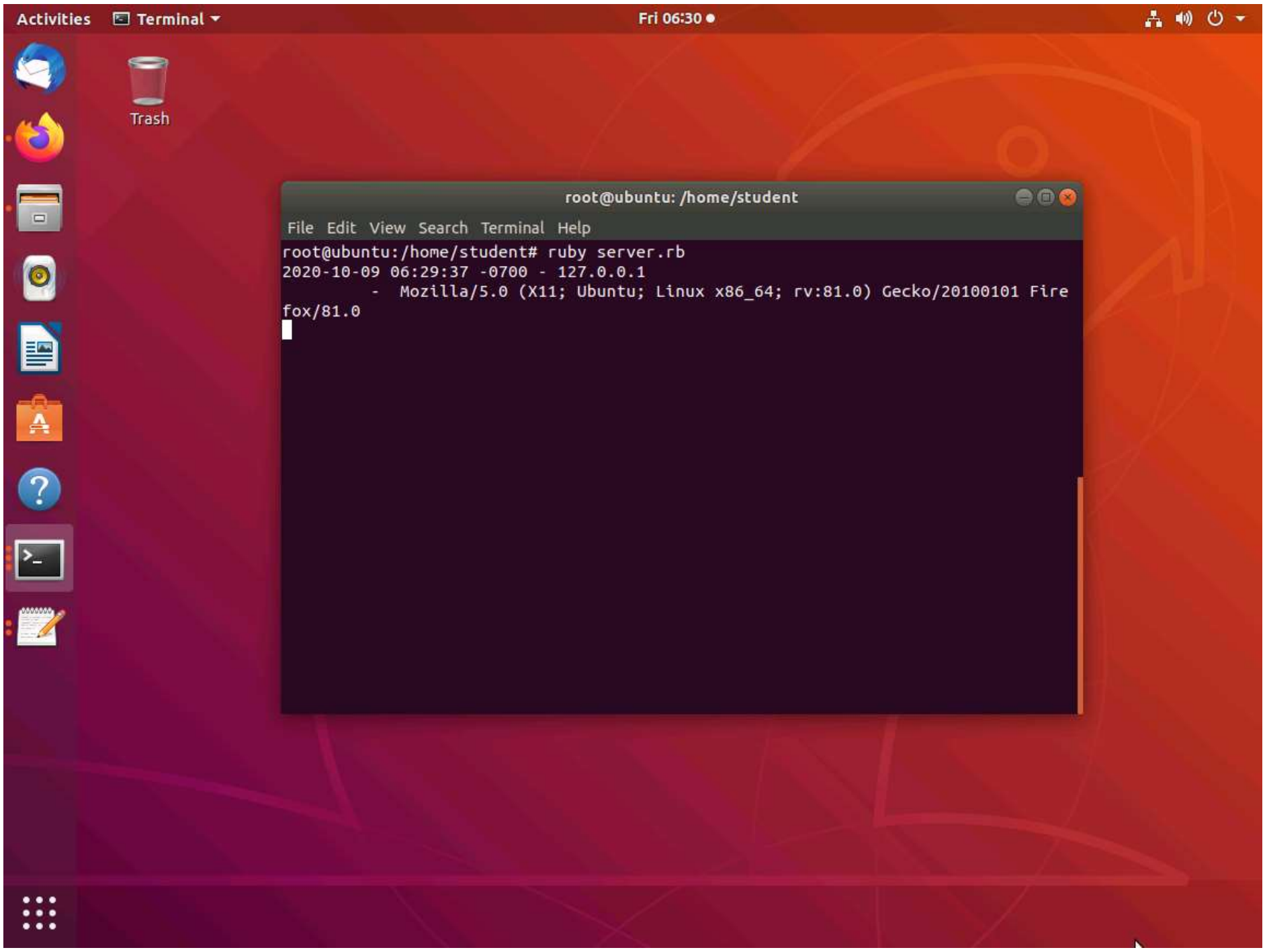
15. Create a server that will listen on port 8080. When an incoming connection occurs, record the request information and return something to the browser to let it know there is no content. This allows the web page that called the script to finish rendering.

16. Once you have the server running, connect to the web page and process the script.





17. An example of the running of the script that prints the access from the access.log file is shown in the screenshot.



18. The modern browsers can strip out the tags. By the time you are reading this, the browser might protect from this and the string will require modification. This can be a further challenge, which necessitates the use of Metasploit.

19. This concludes the lab exercise.