

# MESA Cyber Robot Challenge: Robot Controller Guide

---



<b>Overview .....</b>	<b>1</b>
<b>Overview of Challenge Elements .....</b>	<b>2</b>
Networks, Viruses, and Packets .....	2
The Robot .....	4
<b>Robot Commands .....</b>	<b>6</b>
Moving Forward and Backward .....	6
Turning .....	7
Jumping .....	7
Detecting the Nearest Wall .....	8
Detecting Viruses in the Robot's Neighborhood .....	9
Detecting the Locations and ID Numbers of Packets .....	10
<b>Programming a Robot Controller .....</b>	<b>11</b>
Implementing a Robot Controller .....	11
Using Prewritten Code .....	12
<b>Appendix: Python IDLE Editor .....</b>	<b>13</b>

## Overview

The Cyber Robot Challenge is a high school MESA Day challenge that requires students to apply programming, cryptography, autonomous navigation, and number base conversion skills to guide a virtual robot through a series of networks infected by viruses. The overall challenge is described in detail in the *Cyber Robot Challenge* overview document. This document is a supporting document that provides background information required to program a virtual robot's controller.

This document provides

1. a brief overview of the elements of the Cyber Robot Challenge (also discussed in the overview document), including a detailed overview of the virtual robot to be programmed;
2. a thorough overview of the virtual robot Application Programmer's Interface (API), which is the list of commands student's can use to control the virtual robot;

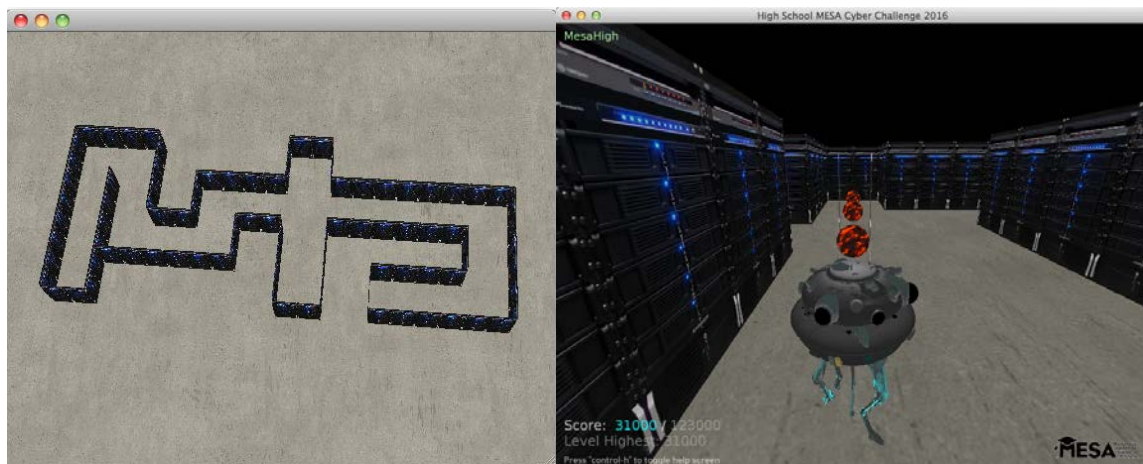
3. instructions on how to get started programming robot controllers;
4. (appendix) a tutorial on how to use the (optional) Python IDLE program editor to write, run, and debug virtual robot controllers.

## Overview of Challenge Elements

For the Cyber Robot Challenge, students will be required to create a robot controller, which is an intelligent program that tells a virtual robot how to navigate through network mazes in search of viruses and packets. In this section, the networks, viruses, and packets are described briefly and the robot and its sensors are described in detail.

### Networks, Viruses, and Packets

The Cyber Robot Challenge consists of a series of networks through which students must navigate their virtual robots. The networks are represented by mazes, and each network is made up of cells of equal sizes that are lined with walls. A maze can have an arbitrary number of cells and is not limited to a square or rectangular shape. Figure 1 (left) shows a top-down perspective of an example network without the robot or the viruses. During the challenge, the network will only be presented from the first-person perspective of the robot as in Figure 1 (right). Keep in mind that at the competition date, a new set of never-before-seen mazes will be provided. Therefore, it is crucial to create controller software that is flexible enough to navigate a network that has not been practiced on before.



**Figure 1: (left) Top-down view of a network. (right) Robot's perspective from inside network.**

Each network has been infected with viruses. The mission of the robot is to move around the network to neutralize as many of these viruses and collect points. There are two types of viruses: Benign Bugs and Vile Viruses (shown in Figure 2). To defeat a Benign Bug, a robot simply has to make contact with it. Vile Viruses are

protected by a lockbox that must first be disarmed by solving a cryptography puzzle. Once the secured box is disarmed, the virus inside is exposed and the robot can proceed on to eliminate the virus (again by making contact with it). Each defeated Benign Bug is worth 1,000 points, and each defeated Vile Virus is worth 5,000 points.



Figure 2: (left) Benign Bug. (right) Lockbox containing a Vile Virus.

In addition to the defeating the viruses, and optional side mission for the robot is to find and collect all the “packets” scattered about the network for bonus points. The packets are numbered, and at the start of a network all packets except the first are “disabled”. When the robot collects an “enabled” packet (i.e., packet 1 if the network was just started), the packet disappears and the next packet in the sequence becomes enabled. One by one, the robot can collect the packets in numerical order until all have been collected. If all packets are collected before the robot defeats a network’s final virus, 10,000 bonus points are awarded. Figure 3 shows examples of an enabled packet, which is green, and a disabled packet, which is translucent yellow.

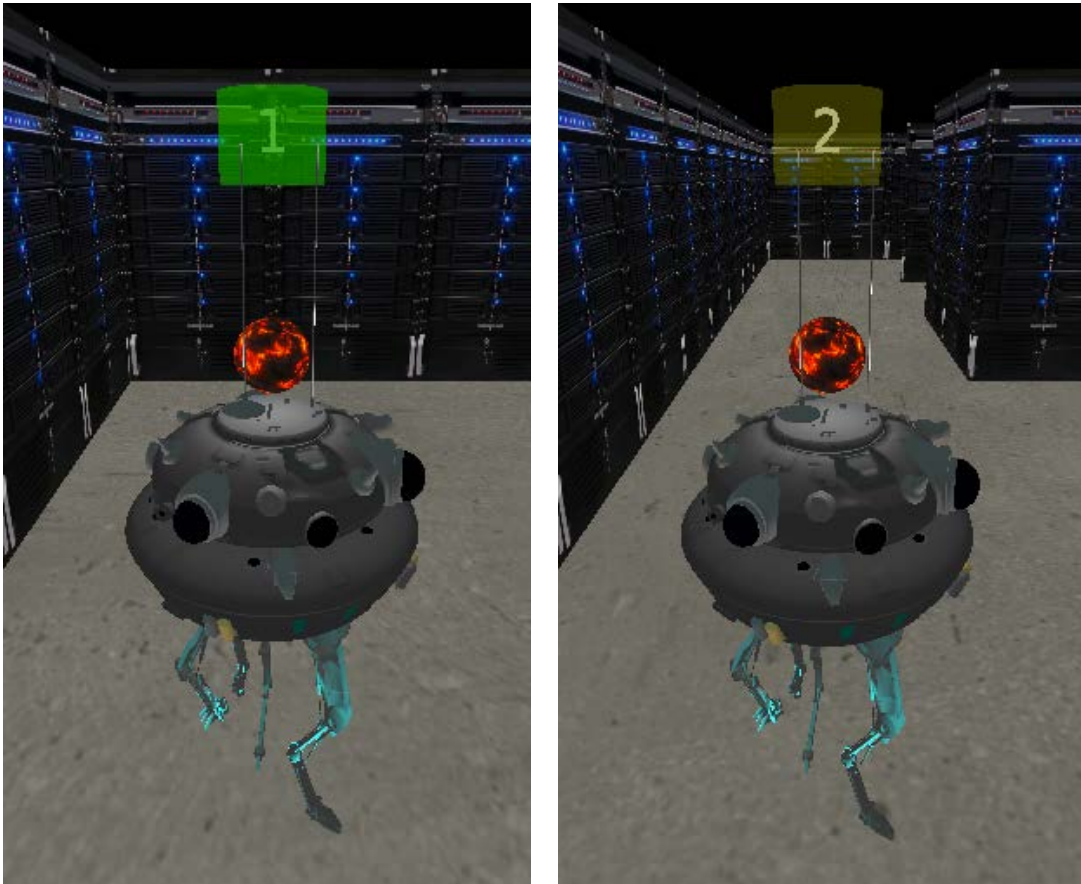


Figure 3: An enabled packet (packet 1, left) is green, while a disabled packet (packet 2, right) is translucent yellow

Packets always hover above Benign Bugs. To “collect” a packet, the packet must be enabled and the robot simply needs to move through it. However, the packets float too high for the robot to touch it using its normal motion. To collect a packet, then, a robot will need to position itself directly underneath the packet and “jump”.

### The Robot

The virtual robot is represented as a 3D model of a droid, depicted in Figure 3. The robot is capable of following basic movement commands, including commands to move forward and backward and to turn left and right. Turns occur in 90° increments, and steps forward and backward happen in cell-by-cell increments (e.g., the robot cannot move  $\frac{1}{4}$  of a cell forward). Robot movement can stop suddenly in the event of a collision with a wall. With regard to wall collisions, the robot is considered as a simple sphere with a diameter of half a cell size. “Collisions” occur when the robot’s spherical body comes into contact with a wall. Care should be taken when designing the robot controller to prevent the robot from ever colliding with a wall.

To detect the presence of walls (and viruses and exits), the robot possesses an array of sensors. The sensors include



- three laser ranging sensors that detect obstacles like walls in a specified direction (one each for the front, left, and right directions),
- a pulsed radar sensor for detecting viruses in the vicinity of the robot, and

Figure 4: The robot.

- a pulsed radar sensor capable of sensing the locations and IDs of all packets in the network.

A list of available commands to control the robot's movement and use of sensors is provided in the next section.

## Robot Commands

The following table is a quick reference to the commands available to control the robot:

Action	Command
step forward, backward	<code>step_forward()</code> , <code>step_backward()</code>
turn left, right	<code>turn_left()</code> , <code>turn_right()</code>
jump	<code>jump()</code>
detect nearest wall	<code>sense_steps()</code>
detect viruses	<code>sense_viruses()</code> , <code>num_viruses_left()</code>
detect packets	<code>sense_packets()</code>

All of the robot commands are methods called on a provided “robot” object called `robot`. Thus, for example, to move the robot one step forward, one would use the statement `robot.step_forward()`. To move the robot one step backward, one would use the statement `robot.step_backward()`. Most of the robot commands can also take optional arguments. For example, to move the robot three steps forward, one would use the statement above with 3 passed as an argument: `robot.step_forward(3)`. Each command is discussed in detail below.

### Moving Forward and Backward

Syntax:       `robot.step_forward()`  
                   `robot.step_forward(N)`  
                   `robot.step_backward()`  
                   `robot.step_backward(N)`

`step_forward()` and `step_backward()` use the same syntax: one moves the robot `N` steps forward and the other moves the robots `N` steps backward. If the optional argument `N` is not supplied, it is assumed that `N = 1` and the robot moves only one step forward/backward. Note that specifying `N` steps to be taken will not always mean the robot will actually take `N` steps: if the robot encounters a wall before finishing its `N`th step, its progress will suddenly stop. Take care to keep track of where walls are when moving the robot. If a robot encounters a Vile Virus before finishing its `N`th step, however, it will continue moving after the Vile Virus is defeated/skipped to finish out its `N` steps.

Examples:

```

robot.step_forward()    # move 1 step forward
robot.step_backward(3)  # move 3 steps
backward a = 5
robot.step_forward(a)   # move 5 steps forward

```

## Turning

Syntax:

```

robot.turn_left()
robot.turn_left(N)
robot.turn_right()
robot.turn_right(N)

```

`turn_left()` and `turn_right()` also use the same syntax: one turns the robot  $N$   $90^\circ$  turns to the left and the other turns the robot  $N$   $90^\circ$  turns to the right. As with `step_forward()` and `step_backward()`, the argument  $N$  is optional and  $N = 1$  is assumed if the argument is not supplied.

Note that using  $N = 2$  amounts to a  $180^\circ$  turn and that various combinations of `turn_left()` and `turn_right()` with specific values of  $N$  are equivalent to each other. For instance, `robot.turn_left()`, `robot.turn_left(1)`, and `robot.turn_right(3)` are all left turns. Likewise, `robot.turn_right()`, `robot.turn_right(1)`, and `robot.turn_left(3)` are all right turns. Furthermore, `robot.turn_left(2)` and `robot.turn_right(2)` do the same thing (a  $180^\circ$  turn), and `robot.turn_left(4)` and `robot.turn_right(4)` effectively do nothing.

Examples:

```

robot.turn_left()    # turn left
robot.turn_right(1)  # turn right
robot.turn_right(2)  # U-turn
robot.turn_right(3)  # turn left

```

## Jumping

Syntax:

```

robot.jump()

```

`jump()` makes the robot jump one time. This is useful for collecting enabled packets, which normally hover too high for the robot to touch using its normal forward and backward motion.

Examples:

```

robot.jump()  # make the robot jump
robot.jump()  # make the robot jump again

```

### Detecting the Nearest Wall

Syntax:        `N = robot.sense_steps()`  
              `N = robot.sense_steps(dir)`

`sense_steps()` is used to detect the number of free steps the robot can move in a given direction before running into a wall. The number of free steps is returned by the call to `sense_steps()`, and the direction is specified by the optional argument `dir`, which must be one of the following predefined constants:

`robot.SENSOR_FORWARD`



```
robot.SENSOR_LEFT  
robot.SENSOR_RIGHT
```

If the optional argument `dir` is not supplied, it is assumed that `dir = robot.SENSOR_FORWARD`. Note that using `robot.SENSOR_LEFT` or `robot.SENSOR_RIGHT` does not actually turn the robot to the left or the right: the robot will still face forward after the call.

Example:

```
# Move to nearest wall to the left  
N = robot.sense_steps(robot.SENSOR_LEFT)  
robot.turn_left()  
robot.step_forward(N)
```

### Detecting Viruses in the Robot's Neighborhood

Syntax: `virus_list = robot.sense_viruses()`  
`N = robot.num_viruses_left()`

`sense_viruses()` causes the robot to send out a pulse that detects the presence of viruses near the robot. Only viruses whose centers are located within a 10-cell circular radius can be detected, and any viruses outside the radius will go undetected and will not be reported in `virus_list`. To illustrate, Figure 4 shows which viruses (light red dots) could be detected by a robot centered at the dark red dot.

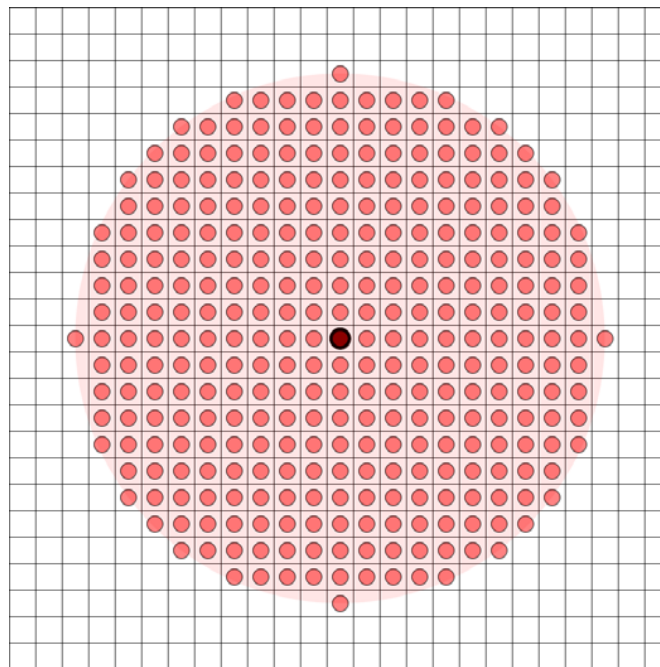


Figure 4: Virus detection radius.

The locations of the detected viruses are returned as a list of ordered pairs. For example, if a call to `sense_viruses()` results in a `virus_list` of

```
[(3,1), (-2,-4), (0,-1)],
```

then there are only three viruses located within a 10-cell radius of the robot. One is located 3 cells to the right and 1 cell ahead of the robot, one is located 2 cells to the left and 4 cells behind the robot, and one is located immediately behind the robot.

`num_viruses_left()` does not take any arguments and returns the total number of viruses remaining in the current network, regardless of how far away they are. Calling this method is useful in determining whether or not it is time to navigate to an exit.

Example:

```
virus_list = robot.sense_viruses()
for pos in virus_list:
    # "pos" is position of next virus in list...
    # ...go defeat virus located at "pos".
if robot.num_viruses_left() == 0:
    # all viruses cleared, go find an exit
```

### Detecting the Locations and ID Numbers of Packets

Syntax: `packet_dict = robot.sense_packets()`

`sense_packets()` detects the locations and ID numbers of packets in a manner similar to `sense_viruses()`. The main differences are that `sense_packets()` has an unlimited detection radius: it senses all the packets in the network no matter how far away they are, and it returns the information as a Python dictionary. (For more help on Python dictionaries, consult a reference on the Python programming language.) The dictionary that `sense_packets()` returns is keyed by ID number and contains ordered pairs that represent the location of packets relative to the robot's current location (similar to `sense_viruses()`).

Examples:

```
packet_dict = robot.sense_packets();
ids = packet_dict.keys()
    # "ids" is list of packet IDs left in network
pos3 = packet_dict[3]
    # "pos3" is the position of packet 3
    # if enabled, go collect packet 3 at "pos3"
```

## Programming a Robot Controller

One of the primary challenge components for which the students are responsible is the development of a series of robot controllers. Each network requires its own robot controller as it may be advantageous to use a different algorithm based on what is presented in each maze. To encourage code reuse, the students will be able to submit a code base that may include some core robot controller functions in a Python module that can be referenced from each of their individual robot controllers.

### Implementing a Robot Controller

The challenge software engine will automatically import the designated robot controller Python module for a given network and pass robot navigation responsibilities to that module. This module must define the `control_robot()` method that is called upon initialization. This is the only method that must be implemented to control the robot. This method has only one argument: the robot object itself.

The automatically imported robot controller Python modules have filenames that follows the pattern `controller_<NETWORK>.py`, where `<NETWORK>` is the name of the network being attempted (which is the same as the network name displayed on the main screen). On competition day, these modules will already be provided with an empty `control_robot()` method definition. For example, if attempting the first network (named N1), the module `controller_N1.py` will automatically be imported and the `controller_N1.control_robot()` method will be invoked. If a controller cannot be found to match the current network, the challenge software will search for a default module called `controller_default.py` and use it instead.

By default, each of the placeholder robot controllers will have the following contents

```
def control_robot(robot):  
    pass
```

As an example, the following is a simple robot controller for the first network (called N1) that causes the robot to go around in a square (assuming there are no walls to interfere):

```
Controller N1.py  
def control_robot(robot):  
    while True:                # continue forever  
        robot.step_forward()    # move forward 1 cell  
        robot.turn_right()      # turn right 90 degrees
```

If it is desired to use the front sensor to determine how far to move (instead of just moving one cell at a time), the example controller could be modified for network N2 as shown:

```
Controller N2.py
def control_robot(robot):
    while True:
        N = robot.sense_steps(robot.SENSOR_FORWARD)
        robot.step_forward(N) # move forward N cells
        robot.turn_right()    # turn right 90 degrees
```

### Using Prewritten Code

It is desired that students create their own robot controller building blocks and to reference the blocks in their robot controllers. This encourages reuse of code blocks that are common to several controllers. It also makes reconfiguring robot controllers on competition day much faster and less stressful. Designing software to maximize code reuse is an important programming skill. As mentioned in the overview, a code base that includes prewritten code for use on competition day may be emailed from the schools competition coordinator no later than two weeks before the competition. Any modifications to the code sooner than two weeks before the competition must be made manually on competition day.

The provided code should either be in one Python module file, or within one Python package directory. In addition, a default controller file named `controller_default.py` can also be submitted to replace the default controller placeholders. On competition day, students will find their submitted module/package in the same directory as the robot controller placeholders and all robot controllers for all levels will be initialized to the contents of the submitted `controller_default.py` file.

An example of a pre-written codebase (called `my_school` and including an implementation of a navigation algorithm called `zig_zag`) and a robot controller (for the network N3) that makes use of the codebase show below. Note that if the students decide on competition day that the “zig-zag” algorithm is an appropriate algorithm for controlling a robot in network N3, they could write `Controller_N3.py` very quickly and on the fly. Using (and reusing) prewritten code save considerable time on competition day. How the students decide to manage control of the robot is completely up to them. It is mainly important to keep in mind that on competition day there will be nine different networks, several of which may require the same types of control logic.

```
my_school.py
def zig_zag(robot):
    # prewritten implementation of the
    # “zig-zag” algorithm goes here
```

### Controller N3.py

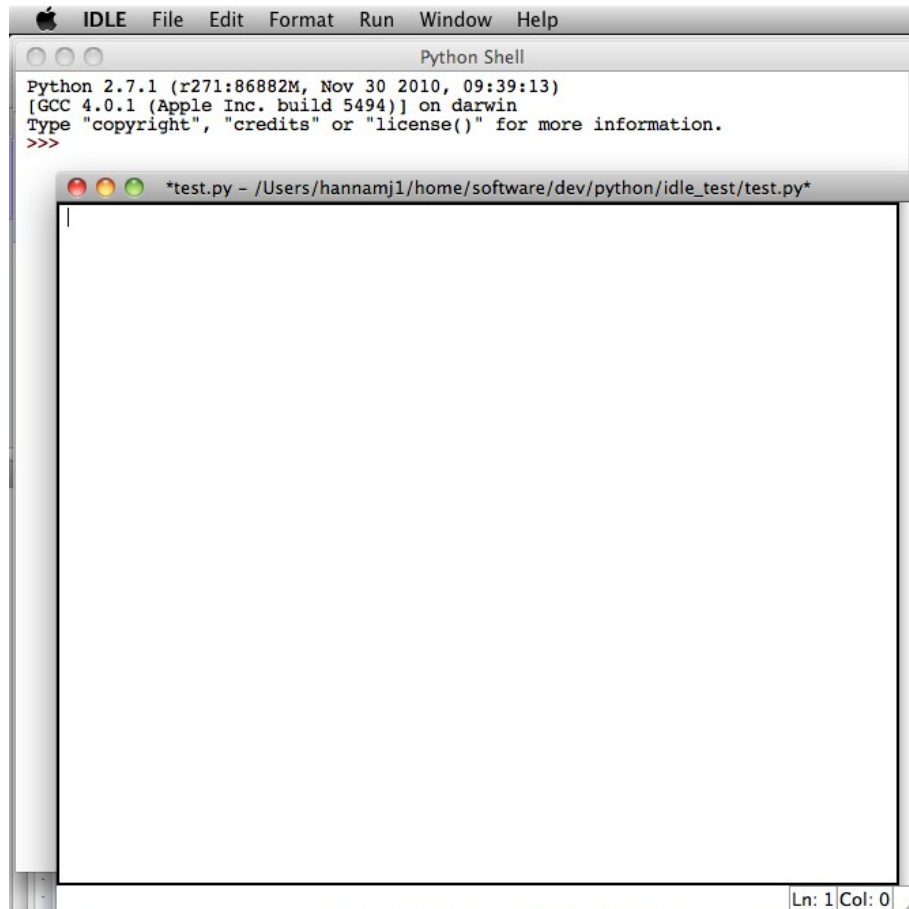
```
import my_school    # prewritten module from My School
def control_robot(robot):
    # execute prewritten "zig-zag" algorithm
    my_school.zig_zag(robot)
```

A generic robot controller might utilize the same controller logic for multiple networks, independent of what geometry the networks present. However, since a first person view of the network is displayed as each robot controller executes, it is possible to restart the network and use a different algorithm geared towards a specific problem. For example, a wall-following algorithm will not prove to be useful if there is a large room with a virus in the center. If a robot controller is generic enough, it is possible to use the same robot controller logic for multiple networks.

## Appendix: Python IDLE Editor

Python is an interpreted language that does not require a compiler. This means that you can write Python code and run it directly after writing it. Python code is just text in a text file that has been saved with a `.py` file extension. This file is called a Python module, with the name of the module being the name of the file. By convention, Python modules are lower case with underscores and cannot have any spaces in the filename. Any text editor can be used to modify a Python code file, but it certainly helps if you utilize an editor that understands Python and can offer syntax highlighting, error checking, block commenting, block indentation and more. For this reason, Python is deployed with an editor called IDLE.

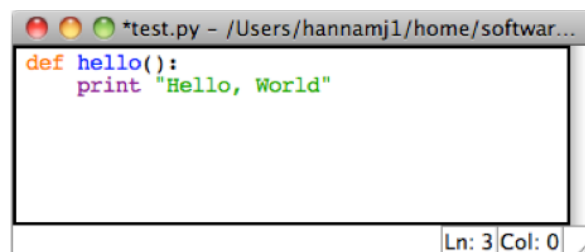
The easiest way to launch IDLE is to make a text file with a `.py` file extension, right click on the file, and select `Edit with IDLE` (or `Open With`). This will open both the IDLE text editor as well as the interactive Python terminal. Opening IDLE this way is beneficial since the Python terminal is initialized to the same directory as the script being edited. This means that the module being edited can be directly imported from the terminal. Below is a snapshot of the result of opening an empty file called `test.py`:



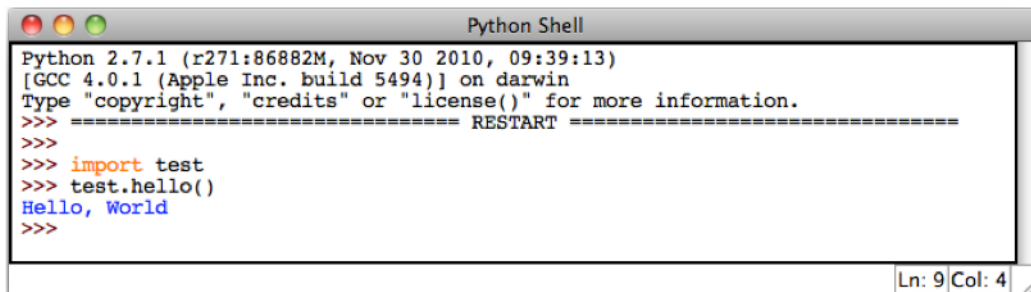
Another way of opening IDLE is to launch the IDLE program and then select New Window. This will open a blank editor window. Select Save As to save this blank file as a new Python file. Opening IDLE this way does not result in the terminal having direct access to the Python file, so the first method is preferred if possible.

The interactive Python terminal is a great environment for testing snippets of code, playing with the syntax, or using Python as a calculator. However, in most cases you want to write the code in a Python module so that it can be executed and reused by being imported into another Python module.

Now you can edit the file and add some content to the Python module. In this example we will create a simple hello world function:



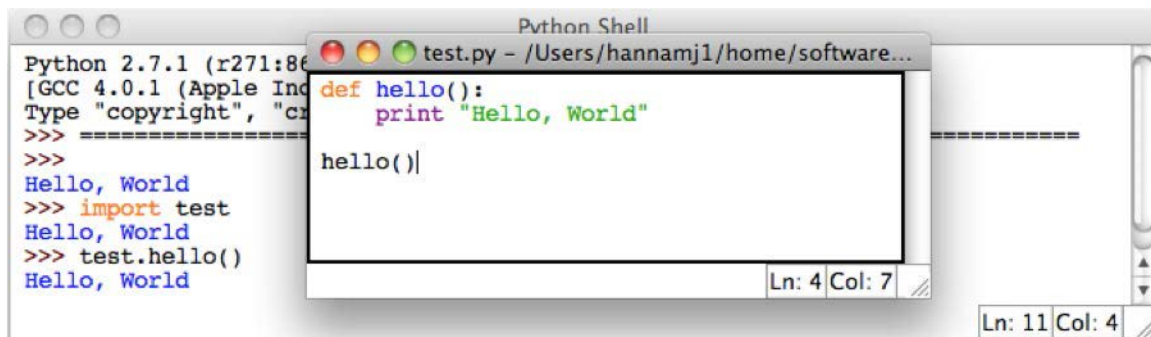
This defines the `hello()` function within the `test` module that prints out the string `Hello, World`. This test module can then be imported by another module and the `hello()` function can be called. One important note about the terminal is that if a module is already loaded, and the code is updated, the terminal will not process the changes made to the code until the terminal is restarted. The best way in IDLE to avoid these problems is to select the `Run->Run Module` option from the editor menu (or hit `F5`). This will execute the module and keep the terminal loaded with the updated module changes. For example, if we run the `test` module we have access to import the module and call the `hello()` function:



```
Python 2.7.1 (r271:86882M, Nov 30 2010, 09:39:13)
[GCC 4.0.1 (Apple Inc. build 5494)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> import test
>>> test.hello()
Hello, World
>>>
```

Ln: 9 Col: 4

Notice that the `hello` function was never actually called until we imported the module and called the function explicitly. If we want it to actually execute when we run it we need to add a call to `hello()` in the `test` module as shown:



```
Python 2.7.1 (r271:86882M, Nov 30 2010, 09:39:13)
[GCC 4.0.1 (Apple Inc. build 5494)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> import test
Hello, World
>>> test.hello()
Hello, World
>>>
```

Ln: 4 Col: 7

Ln: 11 Col: 4

So now, when we select `Run Module`, the `hello()` function is invoked. However, there is also a side effect: when the module is loaded with the `import test` command, the `hello()` function is also invoked. Python provides a mechanism to prevent executing a script when you are trying to import its contents. Every module has a `__name__` attribute that is equal to the value `__main__` if it is being executed (rather than imported). So the logic to actually run when executed should be placed within the following conditional statement:



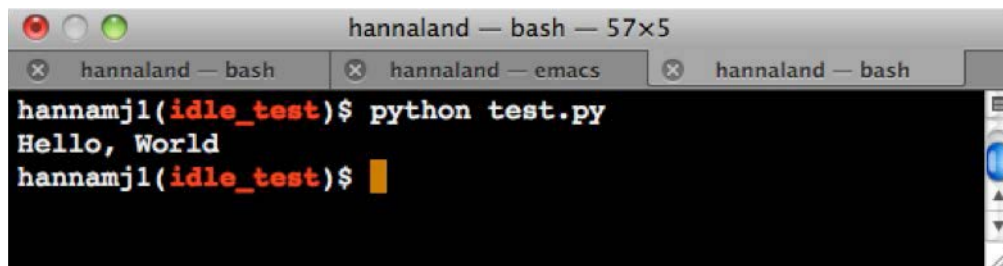
```
Python 2.7.1 (r271:8/28821)
[GCC 4.0.1 (Apple In
Type "copyright", "c
>>> =====
>>>
Hello, World
>>> import test
>>> test.hello()
Hello, World
>>>

def hello():
    print "Hello, World"

if __name__ == "__main__":
    hello()
```

Now you can notice that when the module is executed, the `hello()` function is called, but when it is imported, it is not. This is a very common pattern for most Python modules and enables a module to contain a library of utilities as well as provide executable behavior or tests that use the library functions.

For more advanced users, it is good to understand that Python modules can also be called outside of the IDLE environment. In fact, this is the most common means of execution as it is robust and can be highly automated. This is simply done on the command line (DOS/bash/csh etc.) by using the `python` (`python.exe` on Windows) command followed by the module to be executed. Below is an example:



```
hannamjl(idle_test)$ python test.py
Hello, World
hannamjl(idle_test)$
```

This is just a high-level guide for getting started with Python and IDLE. There is much more documentation on the web. Below are some additional references on Python and on IDLE:

- <http://docs.python.org/library/idle.html>
- <http://wiki.python.org/moin/SimplePrograms>
- <http://docs.python.org/tutorial/>
- [http://en.wikibooks.org/wiki/Python\\_Programming](http://en.wikibooks.org/wiki/Python_Programming)
- <http://www.korokithakis.net/tutorials/python/>