**Bazel Primer:**
Part 1: Install Bazel

1) Update System and install Bazel using Bazel's apt repository.
    a) `sudo apt update && sudo apt install bazel`
    b) `to update in the future run: sudo apt update && sudo apt full-upgrade`
2) Once Bazel is installed, retrieve the sample repository (this one is in Python)
    a) `git clone https://github.com/olin-electric-motorsports/AdvancedResearch.git`

Part 2: Get Familiar with Workspaces and Build Files

*The following passage were adopted from the Bazel C++ tutorial and translated into Python:*

## Set up the workspace

Before you can build a project, you need to set up its workspace. A workspace is a directory that holds your project's source files and Bazel's build outputs. It also contains files that Bazel recognizes as special:

- The WORKSPACE file, which identifies the directory and its contents as a Bazel workspace and lives at the root of the project's directory structure,
- One or more BUILD files, which tell Bazel how to build different parts of the project. (A directory within the workspace that contains a BUILD file is a *package*. You will learn about packages later in this tutorial.)

To designate a directory as a Bazel workspace, create an empty file named WORKSPACE in that directory.

When Bazel builds the project, all inputs and dependencies must be in the same workspace. Files residing in different workspaces are independent of one another unless linked, which is beyond the scope of this tutorial.

A BUILD file contains several different types of instructions for Bazel. The most important type is the *build rule*, which tells Bazel how to build the desired outputs, such as executable binaries or libraries. Each instance of a build rule in the BUILD file is called a *target* and points to a specific set of source files and dependencies. A target can also point to other targets.

Now back to Bazel:

Once the preliminary steps have been completed, cd into the directory that you cloned the repository into, then cd into `build_management/bazel_primer`. There should be two directories that are listed here being named `python_tutorial` and `c_tutorial`. First cd into the `python_tutorial/stage1 directory`; in it you will see a `README.md`, a `WORKSPACE` file, and the `main` folder. Open the `main` folder. In it there is a `BUILD` file and a file named `hello_world.py`. Open hello_world.py and make sure that you understand what is going on.

```
from datetime import datetime

def get_greet(who):
     return "Hello " + str(who);

def print_localtime():
     now = datetime.now()
     print(now.strftime("%a %b %d %H:%M:%S %Y"))

if __name__ == "__main__":
     who = "world"
     print(get_greet(who))
     print_localtime()
```

Once you get a general idea of how this file works you can now take a look at the `BUILD` file. The purpose of this file is a bit self-explanatory, but let's see what is going on:

```
py_binary(
     name = "hello_world",
     srcs = ["hello_world.py"],
)
```

Similarly to how a makefile allows us to compile several C/C++ files and any necessary header files into a single executable This is basically what the `BUILD` file does here. Above is the Python implementation of Bazel's `BUILD` file. We tell the `BUILD` file what we would like to name this executable and supply a list of the source scripts necessary.

Now that we have a better understanding of how Bazel can be used to build executables, let's actually build and run this executable. Open the `stage1` directory in terminal and run the following:

```
bazel build //main:hello_world
```

If the compile went through smoothly, this command will build the executable (in our case we will be building a Python executable) that can be run by simply calling the file's name in the terminal. Relative from our current terminal location, the executable is located in `bazel-bin/main`. Let's run the code and see what we get now:

```
bazel-bin/main/hello_world
```

If everything went through smoothly we should have received two lines of messages; the first displaying "Hello World" and the second displaying the current date and time in UTC.

Always keeping track of file dependencies and your build environment is a good practice to develop, and using a graphical representation of what is going on in your code is probably one of the simplest ways to do this. GraphViz and XDot are great tools for this, and we will be using them here.

First, install GraphViz and XDot onto your machine:

```
sudo apt update && sudo apt install graphviz xdot
```

Once the install finishes you can view the graphical representation with the following command:

```
xdot <(bazel query --notool_deps --noimplicit_deps
"deps(//main:hello_world)" \  --output graph)
```

Now that we understand the basics of what is going on regarding building with Bazel, you can look through the rest of this Python adaptation repository of the Bazel Tutorial to understand how to build with multiple source files.

Here is a link to the C++ tutorial that this primer was based off of:

https://docs.bazel.build/versions/master/tutorial/cpp.html

Although the code written in C++, the concepts explored in large part are the same. You can compare what is in `stage2` and `stage3` of this repository to that in the tutorial linked above, and from that a greater understanding of how Bazel works to build executables across varying languages can be achieved.

One thing to take note of however is that in C and C++ there are special file types known as 'header files'. These file have the .h extension and can be used to pre-define variables and functions in or across multiple `.c` and `.cc/.cpp` files. Because this feature is not a thing in Python, the header files have been replaced with just normal Python .py files that accomplish the same task as their lower-level counterparts.

The reason why we started off with looking into how to set up Bazel with Python is because Python is a relatively simple language to understand, and most individuals in the coding community are at least familiar with Python and its conventions. Now let us look at C. As was mentioned before the concepts have not changed, rather they are now undertaken using different implementations. Look through `c_tutorial` and see if you can make sense of what is happening in `stage1`, `stage2`, and `stage3`. Because we started with Python first this should not be as daunting as it could have been had we begun this with C.

Try running through the development process that we used for `python_tutorial`, but this time for the `c_tutorial`. All the terminal commands should be the same, so unless you want to try remixing one of the C files, you should be able to run through this version of the tutorial just as smoothly.

Once you finish building one of the executables, cd over to `bazel-bin/main` from whichever stage directory you are currently in. Once there you'll see that now our executable has no file extension in its name. This is because when working with C (one of the lower-level modern languages) we are in most cases operating on our machine's memory without a middleman interpreter.

In reality this extensionless file is actually in the Executable Linkable Format (`.elf`). Similarly to how Windows has it's `.exe` format, this format can be thought of as it's parallel on many of the several different types of Unix systems. This now brings us one step closer to the format that we need to program our microcontrollers. The microcontrollers that we use in most cases read Hexadecimal Source (`.hex`) files. One can convert a working `.elf` to a `.hex` simply by running the following command from their terminal in the same directory that the `.elf` file is located:

```
objcopy -O ihex input.elf output.hex
```

And there it is! You have successfully gone through the entire build process with Bazel by first getting into the swing of things with Python and then using C to create a `.elf` that could then be converted to a `.hex`. This `.hex` file can now be flashed to a microcontroller.

Congratulations!