# **CTI 110**

# **M1T1 (Module 1, Tutorial 1): “Hello World”**

A common programming problem when learning a new language is to write a program that does one thing: printing “Hello World” as output.

While this is a trivial problem, it confirms that the programming language you wish to use is installed properly and that you are able to run a program with it. In this tutorial we’ll look at “Hello World” in several languages, and then you will tell your computer to print “Hello World” in two, Python (a programming language) and HTML (a markup language).

As you work through this tutorial you will see sections marked

**TO DO:** Follow these instructions when they appear!

They indicate steps you will need to complete for this assignment.

This tutorial consists of two parts, **Background** and **Assignment**. The Module 1 Quiz contains questions about both parts.

# **Part I -- Background**

# Computer Languages

Computers don’t speak fluent English. Even if they did, English (and most other human languages) are not precise enough to give them exact instructions to follow. (For example, try asking Google or Siri something in conversational speech, you’ll often be misunderstood.)

Computers speak **binary**, a series of ones and zeros. These binary digits (or **bits**) are put together to form larger numbers, and each number has a specific meaning to a particular CPU. This is called **machine code**. In the early days of computing, humans hand-translated instructions into a series of ones and zeros and then entered those values into a computer by flipping a series of switches on and off. The computer replied by flashing a series of lights on and off.

Later systems used decimal or hexadecimal numbers to save space, but the underlying ones and zeros were still there.

Here’s one way to write “Hello, world!” in machine code:

48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21 0a 00 2e |Hello, world!...|

This was a difficult system to use, so programmers developed **programming languages** that were easier for us to ‘speak’ in. As these languages became more complex, a distinction was made between **low-level languages** and **high-level languages**.

The first low-level languages were **assembly language**. Instead of binary numbers for instructions, assembly language uses mnemonics – short phrases that are easier for humans to remember. Instead of memorizing the special number used to tell a processor to add two values, you might type add. Moving a value from one place to another in memory might be mov. Programmers used a tool called an assembler to translate this language into binary.

With that in mind, here’s “Hello World” in 8086 Assembly as someone might have used during the 1980s. (This and all other examples are from the Rosetta Code website.)

# Hello World – 8086 Assembly (IBM PC compatible)

DOSSEG  
.MODEL TINY  
.DATA  
TXT DB "Hello world!$"  
.CODE  
START:  
 MOV ax, @DATA  
 MOV ds, ax   
 MOV ah, 09h ; prepare output function  
 MOV dx, OFFSET TXT ; set offset  
 INT 21h ; output string TXT  
  
 MOV AX, 4C00h ; go back to DOS  
 INT 21h  
END START

While this is easier to read than a series of ones and zeros, it’s not very programmer-friendly. A programmer still has to write down each step the computer has to take, and they are very small steps. For example, here’s part of 8086 assembly program to add two numbers.

# Add two numbers – 8086 Assembly

; This program adds two numbers together and stores the result

ASSUME CS:CODE,DS:DATA  
START:  
 MOV ax, @DATA  
 MOV ds, ax  
 MOV al, a ; get the first number a   
 MOV bl, b ; get the second number b  
 ADD al, bl ; add the two  
 MOV c, ax ; save result in c  
   
END START

Four or more lines to say c = a + b, and that’s not even the whole program.

In addition, every processor had a different set of instructions, and this meant a different assembly language dialect was used on each. A 6502 (Apple II) spoke a different language than an 8086 (IBM PC).

# High-level languages

Just as you can tell someone “Go to the store and pick up some milk” without having to tell every step including “open the front door”, “close the front door”, and “stop at red lights”, **high-level languages** allow programmers to bundle a lot of small steps into a simple abstraction.

Instead of specifying each step of addition, you might write x = 2 + 2. Because computers only speak binary, another translation program was needed to convert this language into assembly and then into native machine code. These translation programs are called **compilers** if they convert the entire program at once, and **interpreters** if they convert it while the program is running.

High-level languages aren’t English, but they allow you to think in English-sized steps.

***To recap:* High-level Language** is interpreted into **assembly language** which is assembled into binary **machine code**.

# Types of high-level languages

There are many high-level languages, but only a handful see common use today.

This course will teach you the basics of programming in **Python**, a language created by Guido van Rossum. Python was released in 1991 and has continually been expanded since then. We will use Python 3.x (the x here is a number between 4 and 6, so you might use Python 3.5 or 3.6.) There is a dialect called Python 2 (2.7x) which is slightly different, so make sure you’re using some version of Python 3.

Van Rossum is a fan of the TV show *Monty Python’s Flying Circus*, by the way.

So what does Python code look like?

# Hello World – Python 3.x

**print (**"Hello world!"**)**

Python is a lot easier to read than assembly. While it’s still not English, it uses commands that are close enough to guess at. To print “It’s only a flesh wound”, you can write:

print (“It’s only a flesh wound”)

To ask someone what their favorite color is (and save it in memory in a location called favorite), you can write:

favorite = input(“What is your favorite color?”)

In this course we will learn a few Python commands at a time, and build on them in each module to write increasingly complex programs.

# **Part 2 – Assignment**

# Writing your own “Hello World”

You know the Python code needed, so now you need to get that information into Python.

We will use the tool **IDLE** (for Integrated DeveLopment Environment) to write Python programs. IDLE is installed by default when you install Python. (For more information, see the Appendices under “Further Reading” in Module 1.) You don’t run python.exe itself – you launch IDLE and let it take care of the behind the scenes details.

IDLE has two modes. At launch it starts a **Python shell**, an interactive window where you can type Python code and it will interpret what you type as soon as you press Enter.

**TO DO:** Try this now. Launch IDLE and type this print statement into the window.

Here’s a sample output:

Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32

Type "copyright", "credits" or "license()" for more information.

>>> **print ('Hello world')**

Hello world

>>>

The >>> symbols are the IDLE prompt, which tells you IDLE is waiting for your input. My input is in **bold**.

While we’re here, let’s try a few more commands.

**TO DO:** Type these commands, pressing Enter after each.

2+2

10 / 3

Here’s what my IDLE shell looks like at this point:

>>> **print ('Hello world')**

Hello world

>>> **2+2**

4

>>> **10 / 3**

3.3333333333333335

**TO DO:** You should **take a screenshot** of your IDLE shell window with your conversation with Python now. Follow these steps:

1. With your IDLE Shell window selected, press **Alt + PrintScreen.** This will screenshot just the IDLE window and not your entire desktop. You can also use the Windows Clipping tool or another method you feel comfortable with.
2. Open MS Word and paste into a new document. Your screenshot will appear in the document.
3. Save this document with a filename in the format **M1T1\_lastname.docx** (replacing “lastname” with your last name). You should save the file somewhere you can easily find it when it’s time to submit your work, for example on your flash drive in a directory named CTI110 or on the desktop.

You may have noticed that sometimes I put spaces between commands, and sometimes I didn’t. Either will work here. Your instructor may go into more detail, but basically some people find code easier to read one way or the other.

By the way, you can use either single quotes or double quotes around “Hello World”, as long as they match. We’ll explain further in Module 2.

# From The Shell to a Program

The shell is useful for finding out how Python works, and it’s where you carry out most of your conversations with it. But you’ll need to use a different part of IDLE in order to write and save full programs.

**TO DO:** From the IDLE window, select **File -> New File**.

This will open a new window. This is an **editor window**, where you’ll write longer programs for IDLE to run for you. This window works like a standard text editor.

Outside of the shell, python needs a **source file** in order to interpret and run a Python program. These files end with a .py extension. Let’s put something in your file and then save it.

**TO DO:** Type the following code into your IDLE editor window:

# M1T1

# Hello World

# My Name

# The Current Date

Replace “My Name” and “The Current Date”. This is a **comment header block** and all your programs must include them. Correct filenames and comment blocks are necessary to receive full credit for any assignment.

Now save your file with **File -> Save.** Notice where IDLE wants to save your file by default. It’s likely not where you would ever find it again, so first navigate to where you’re saving files. Preferably this is a **CTI110** folder on your flash drive. For a filename use **M1T1\_lastname**, with your own last name.

**TO DO:** Confirm that the M1T1\_lastname.py file and your screenshot of the IDLE shell are both in the same location.

Now you have a Python program with a correct header, but it doesn’t do anything. Below the comment block, add a new line:

print("Hello World")

Unlike the shell, there is no >>> prompt.

# Running the Program

**TO DO:** From the editor menu, select **Run -> Run Module**. (A module is a .py file.) You will see a dialog titled **Save Before Run or Check**, telling you that your file has changed since you last saved it. Select **OK**.

The shell window will reappear with new output. Mine looks like this:

============ RESTART: C:/Users/username/CTI110/m1t1\_mylastname.py ============

Hello World

>>>

The RESTART line indicates that Python is running a new program. The next line is the program output, and the >>> prompt indicates that the program has finished running. (Nothing is in bold here, because I didn’t type anything into the shell this time.)

As you work on programs you’ll go back and forth between your editor window and the shell. It can be helpful to line them up side by side so that you can see both – in the long run this will save you a lot of fumbling between windows.

# Submitting Your Work

In order to get full credit for the assignment, you’ll need to attach both files (your screenshot and the .py file) and submit the assignment to Blackboard.

You can attach any number of files before you press Submit, and some assignments will require you to include multiple files in this way. Ask your instructor for assistance if you have issues submitting the assignment.