National University of Singapore
School of Computing

CS2105 **Assignment 0** Semester 2 AY20/21

Submission Deadline

31st Jan 2021 (Sunday) **11:59pm** sharp. 1 point penalty will be imposed on late submission (Late submission refers to submission or re-submission after the deadline).

Objectives

This is a warm up assignment to familiarize you with programming skills that will be useful for later assignments.

This programming assignment is worth 3 marks. All the work in this assignment shall be completed **individually**.

Grading

We accept submission of Python 3, Java, or C/C++ programs. We recommend you to use **Python 3** for your assignments though, as all sample codes or skeleton programs are provided in Python 3. Programming languages other than Python 3, Java, and C/C++ are not allowed.

In addition, we will test and grade your programs on the `sunfire` server. Please make sure that your programs run properly on `sunfire` and not only on your own system. By default, we use the `python3` program installed in folder `/usr/local/Python-3.7/bin` on `sunfire` for grading. Unless stated otherwise for individual tasks, you are allowed to use libraries installed in public folders of `sunfire` (e.g. `/usr/lib`) only.

Each program is worth 1 mark. Your programs will be graded automatically using scripts. Please make sure that your programs behave exactly the same as described in this document, because the grading scripts are unable to award partial marks.

A set of testing scripts is released to you in the assignment package. These test scripts cover common cases that your programs are expected to handle. During actual grading, we will use extra cases to test your programs. Hence, passing all the released test cases does not guarantee that you will get full marks. In addition, we will detect fraudulent cases such as hard-coding answers within programs and deduct marks accordingly.

If you choose to use Java or C/C++, we will compile and run your program for grading using the default compilers on `sunfire` (`java 9.0.4` installed in `/usr/local/java/jdk/bin`, or `gcc 4.8.4` installed in `/usr/local/gcc-4.8/bin`). The programming language is inferred from the extension name of your source code during grading. Therefore, please use extension names to indicate the language of your programs. For exam-

ple, if a task requires submission of `Checksum.py`, you should provide `Checksum.java`, `Checksum.c`, or `Checksum.cpp`, respectively, depending on your programming language. For a Java program, the class name should be consistent with the source file name, and please implement the static `main()` method so that the program can be executed as a standalone process after compilation.

We will deduct 1 mark for failure to follow instructions. Please take note of the following common issues:

- Make sure that your programs have correct names.
- Make sure that every line of your output is properly ended with a line break (i.e. "\n" character). In particular, your program should end the output with a line break.
- Do not output irrelevant messages that are not shown in sample runs (for example, debugging messages).

Testing Your Programs

To test your programs, please use your SoC UNIX ID and password to log on to `sunfire` first:

1. If you don't have your SoC UNIX account, please create it here:

   https://mysoc.nus.edu.sg/~newacct

2. If you forget your SoC password, please reset it here using your NUSNET ID and password:

   https://mysoc.nus.edu.sg/~myacct/resetpass.cgi

3. If you are using a UNIX-like system (e.g. Linux, Mac, or Cygwin on Windows), SSH should be available from the command line and you may simply type:

   ```
   ssh <SoC-UNIX-ID>@sunfire.comp.nus.edu.sg
   ```

4. To copy files to `sunfire` on command line, use `scp`. The usage is as following:

   ```
   scp -r <source-folder> <SoC-UNIX-ID>@sunfire.comp.nus.edu.sg:<target-folder>
   ```

5. After logging on to `sunfire`, you can run a Python 3 script with the following command:

   ```
   /usr/local/Python-3.7/bin/python3 <path-to-script>
   ```

   For convenience, you can set a shortcut to python3 as follows:

   ```
   alias python3=/usr/local/Python-3.7/bin/python3
   ```

   This shortcut is temporary, and it lasts for the current SSH session only. To make it permanent, you can run the following command **once** to store the shortcut into the configuration file:

   ```
   echo alias python3=/usr/local/Python-3.7/bin/python3 >> ~/.bash_profile
   ```

   With the shortcut, you can then run a Python 3 script simply with

   ```
   python3 <path-to-script>
   ```

6. If you are using a Windows machine, you may need to install an SSH client (e.g. "SSH Secure Shell Client", available in LumiNUS Files -> "SSH Secure Shell" folder) if your machine does not have one installed. You may use the "SSH Secure File Transfer Client" software (bundled with "SSH Secure Shell Client") to upload your programs to `sunfire` for testing.

Before running the test scripts, please upload your programs along with the `test` folder from the package to `sunfire`. Make sure that your programs and the `test` folder are in the same directory. Then, you can run `bash test/<Exercise Name>.sh` for testing. For example, to test your program for Exercise 1, run the following command:

```
bash test/IPAddress.sh
```

By default, the script runs through all test cases. You can also choose to run a certain test case by specifying the case number in the command:

```
bash test/IPAddress.sh 3
```

To stop a test, press and optionally hold `Ctrl-c` if pressing once does not exit the test.

If you use Java or C/C++, we will compile your program **each time** you run the testing script. The compiled code is stored in a temporary folder and removed after testing, in order to avoid affecting files under your own folders.

You may assume that all input data to your programs are valid. Hence, there is no need to perform input data validation in your programs. If you have any question or encounter any problem with the steps above, please post your questions on LumiNUS forum or consult the teaching team.

Program Submission

Please create a **zip** file containing your source files and submit it to the `Assignment_0_student_submission` folder of LumiNUS Files. The file name should be `<Matric Number>.zip` where `<Matric Number>` is your matriculation number which starts with letter `A`, and the file format must be .zip. An example file name would be `A0165432X.zip`. All file names, including the zip file and all source files within the zip, are **case-sensitive**. In addition, your zip file should not contain any folders or subfolders or irrelevant files such as test.jpg, although we always try to find your actual source files during grading.

You are not allowed to post your solutions to any publicly accessible site on the Internet.

Plagiarism Warning

You are free to discuss this assignment with your friends. But, ultimately, you should write your own program. We employ zero-tolerance policy against plagiarism. If a suspicious case is found, student would be asked to explain his/her code to the evaluator in face. A confirmed breach may result in an F grade for the entire module and further

disciplinary action from the school.

Question & Answer

If you have any doubts on this assignment, please post your questions on LumiNUS forum or consult the teaching team. We are not supposed to debug programs for you, and we provide support for language-specific questions on a best-effort basis only. The intention of Q&A is to help clarify misconceptions and give you necessary directions.

Exercise 1 – IPAddress

Before you start, you may read the following online article on how to use command-line argument: https://docs.python.org/3/library/sys.html#sys.argv.

An IP address is a sequence of 32 bits (a bit is either 1 or 0). Your task is to write a program that reads an IP address in the bit-sequence form from a command-line argument, converts it to the dotted decimal format that we normally see, and then prints the result. The dotted decimal format of an IP address is formed by grouping 8 bits at a time and converting the binary representation into decimal representation.

For example, an IP address 00000011100000001111111111111110 will be converted into dotted decimal format as: 3.128.255.254. This is because:

1. the 1st 8 bits 00000011 will be converted to 3,
2. the 2nd 8 bits 10000000 will be converted to 128,
3. the 3rd 8 bits 11111111 will be converted to 255, and
4. the last 8 bits 11111110 will be converted to 254.

Name your program as `IPAddress.py`.

Two sample runs are given below, with command line inputs highlighted in blue. Note that all sample runs require the `python3` shortcut when running on `sunfire`.

Sample run #1:

```
$ python3 IPAddress.py 00000011100000001111111111111110
3.128.255.254
```

Sample run #2:

```
$ python3 IPAddress.py 11001011100001001110010110000000
203.132.229.128
```

Exercise 2 – Checksum

Checksum can be used to detect if data is corrupted during network transmission (e.g. a bit flips from 0 to 1). Write a program `Checksum.py` to calculate the CRC-32 checksum for a file `<src>` entered as command-line argument. File `<src>` should be placed in the same folder as `Checksum.py`.

You may use the `crc32()` function from the `zlib` library to calculate the CRC-32 checksum (see https://docs.python.org/3/library/zlib.html#zlib.crc32 for details. Remember to import this library before use!). Firstly, you will need to read all the bytes from a file and store them into a `bytes` object. You should open the file with **b**inary **r**eading mode by using **"rb"** as the mode argument of `open()`. Then, you can call the `read()` method of the file object and get the entire file content in a `bytes` object. Now you can get the checksum by directly calling `crc32()` with the file data, and finally print this unsigned 32-bit checksum.

An example code snippet is given below.

```
with open("test.jpg", "rb") as f:
  bytes = f.read()
checksum = zlib.crc32(bytes)
```

Note: in practice, you may encounter an out-of-memory error if you attempt to load a huge file entirely into the memory. In this exercise, however, we limit the maximum file size to 10MB so that you can do so safely without triggering such errors.

If you use Java, you may want to import the `java.util.zip.CRC32` class. If you use C/C++, you can call the `crc32()` function of `zlib` as well, and we will link your program to `zlib` using `-lz` during grading.

Sample run:

```
$ python3 Checksum.py test/test.jpg
3237218320
```

Exercise 3 – PacketExtr

In this exercise, you are going to write a program, `PacketExtr.py`, to read consecutive "packets" from the `stdin` stream, to filter out packets with the valid packet type, and to extract them with data payloads in a *responsive* manner and output to `stdout`.

We define a custom format of packets for this exercise. A packet consists of a text-based header and a binary data payload following the header. The header is a string formatted as "`Type:␣<PacketType>,␣Size:␣<size>B`", where ␣ represents a white-space character, `<PacketType>` is a 6 bytes field representing the type of the packet, and `<size>` is a decimal integer representing the number of bytes of the binary data following the header. The header is ended with the "`B`" character, and the payload immediately follows the header without any byte (such as "\n") in between. For example, "`Type:␣CS2105,␣Size:␣1024B`" is a complete and valid header.

There could be many different types of packets, but the program is required to only process and output the packets of type "CS2105".

The binary payload can contain any byte, not limited to printable characters. Therefore, the payload should not be treated as string data in your program. This packet format clearly defines the boundary between header and binary data within a packet, and also the boundary between consecutive packets. It ensures that all information can be parsed correctly over a data stream.

Packets are fed to your program sequentially through `stdin`, until End-Of-File (EOF) is encountered. It is guaranteed that all packets have correct formats and correct payload sizes. Your program should be **responsive** to the input in the sense that upon receiving a full packet, it outputs the payload of the packet to `stdout`. Unlike Exercise 2, the program cannot read all data once and process them in batch.

It is recommended that your program reads and writes data in binary mode instead

of the default text mode. For interactive I/O, `read1()` should be used instead of the ordinary `read()`, and `flush()` should be called after each `write()` to avoid delays caused by buffering. Both `read()` and `read1()` accept one argument which is the maximum number of bytes to read. The main difference is that, `read()` returns only when the specified number of bytes are read or End Of File (EOF) is encountered, while `read1()` returns immediately upon new data stream in and may return fewer bytes than specified. You can learn more about the details at https://docs.python.org/3/library/io.html#io.BufferedReader and https://docs.python.org/3/library/io.html#io.BufferedWriter. The following code shows how to do binary I/O on `stdin` and `stdout`:

```
import sys
# read **at most** 5 bytes from stdin
data = sys.stdin.buffer.read1(5)
# write data to stdout and flush immediately
sys.stdout.buffer.write(data)
sys.stdout.buffer.flush()
```

Here, the `data` object is of `bytes` class. Python 3 programs can operate on binary data using `bytes` objects. This class is very similar to `str`, the string class. For example, both classes have functions `find()`, `split()`, and also slice operators for range access (e.g. `a[0:10]`). Details about this class can be found at https://docs.python.org/3/library/stdtypes.html#bytes-and-bytearray-operations.

The following shows how to manipulate `bytes` objects:

```
# prepend b to the '...' expression to form a bytes object
# instead of str
pos = data.find(b'x')
if pos >= 0:
  # if byte 'x' is found in data
  part1 = data[0:pos+1]  # this is similar to str slicing
  part2 = data[pos+1:]   # slice until the end of data
```

To convert a `bytes` object to `str`, your program can call the `decode()` method of `bytes`. In this exercise, there is no need to worry about text encoding, as we only use basic ASCII characters in headers.

Finally, to detect EOF on `stdin`, your program can check the length of the `bytes` object read from `stdin`. If your program expects to read more data from `stdin` but receives a zero-length `bytes` object, it means that there is no more data on `stdin` and EOF is encountered.

When testing your program on command line, you can feed the contents of a file to `stdin` of your program using file redirection (<), instead of typing into the terminal. You can also use another type of redirection (>) to save the output of your program to a file rather than let it print to the terminal. For example, the following line feeds the file

`input.data` to the program and save its output to `output.data`:

```
python3 PacketExtr.py < input.data > output.data
```

You can then compare binary contents of `output.data` and the given reference output by running:

```
cmp output.data ref-output.data
```

By default, `cmp` outputs all differences found between the two files. Hence, no output means that the two files are identical.

During grading, we limit the size of each packet to 1MB and the size of all packets to 10MB. In addition to testing correctness, we will test responsiveness by setting a **one-second timeout** after feeding your program a full packet. That is, if your program does not output the packet payload on time, we will deem your program as unresponsive. This timeout is sufficient for our test data sizes. We reiterate that your program should not do batch processing, as interactive processing is one of the key points of this exercise.

Sample run:

```
$ python3 PacketExtr.py < test/packets-a.in > run-a.out
$ cmp test/packets-a.out run-a.out
```