National University of Singapore
School of Computing

CS2105 **Assignment 0** Semester 2 AY18/19

Submission Deadline

9th Feb 2019 (Saturday) **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 your later assignments.

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

Testing Your Programs

As announced in class, we **officially support Python 3 only**, and we strongly recommend you to use Python 3 for your assignments. All sample codes and skeleton programs are provided in Python 3 though we accept submission of C/C++ and Java programs as well. Programming languages other than Python 3, Java, and C/C++ are not allowed.

In addition, you are responsible to ensure that your programs run properly on **`sunfire`** server because **we will test and grade your programs on `sunfire`**. Unless stated otherwise for individual tasks, you are allowed to use libraries installed in public folders of sunfire (e.g. /usr/lib) only.

To test your programs, you may log on to **`sunfire`** using your **SoC UNIX id and password**.

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

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

b) If you forget your password, please reset it here:

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

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

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

To copy files to **`sunfire`**, use **`scp`**. The usage is as following (you should type the command in one line instead of two lines shown here):

```
scp -r <source_directory>
<SoC_UNIX_id>@sunfire.comp.nus.edu.sg:<target_directory>
```

d) To run a Python 3 script on **`sunfire`**, please use the following command:

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

For conenience, you may set a temporary shortcut for the full path of python3 program as follows:

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

This aliasing lasts for the current SSH session. You can then run a Python 3 script simply with

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

The sample runs that are given later require the **`python3`** aliasing to be set up.

e) If you are using a Windows machine, you may need to install an SSH client (e.g. “SSH Secure Shell Client”, downloadable from IVLE Workbin -> “SSH Secure Shell” folder) if your machine does not already have one installed. You may use the “SSH Secure File Transfer Client” software (bundled with “SSH Secure Shell Client”) to upload your programs to the **`sunfire`** server for testing. A brief guide on how to connect to **`sunfire`** and upload programs can be found in IVLE Files -> Assignments folder.

For your convenience, we provide a simple test suite. Each exercise has a simple test case. Please copy your programs, the three test cases and the folder named "tests" to **`sunfire`** and make sure your programs and three `.sh` files are in the same directory. To test your program for exercises 1, run the following command:

```
bash test-IPAddress.sh
```

You are advised to test your solutions thoroughly before submission. The test cases are provided for convenience only. Passing all tests does not guarantee that you will get full marks. You may assume all input data to your programs are valid and hence there is no need for you to perform input data validation.

If you have any question or encounter any problem with the steps above, please post your questions on IVLE forum or consult the teaching team.

Grading

Each program is worth 1 mark. Your programs will be graded automatically using grading scripts. Please make sure your programs behave exactly the same as the sample runs given in this document, as the grading scripts are unable to award partial marks. By default, we use the `python3` program installed in `/usr/local/Python-3.7/bin` on sunfire for grading.

We recommend that you use Python 3 for the assignments. If you use Java or C/C++, we will compile and run your program for grading, using the default compilers on sunfire (g++ 4.8.4 or java 9.0.4). Please make sure that your source file for a task has the same name as the Python script except the extension name. For example, if a task requires submission of “Checksum.py”, you should provide “Checksum.c”, “Checksum.cpp”, or “Checksum.java” respectively, depending on your programming language. The programming language is inferred from the extension name during grading. For a Java program, the class name should be consistent with the source file name, and please implement the **main()** method so that the program can be executed as a standalone process after compilation.

We will deduct 1 mark for every type of failure to follow instructions (e.g. wrong program name). Please take note of the following common issues:

1. Please 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.
2. Please do not output excessive messages not shown in sample runs (for example, debugging messages).

Program Submission

Please create a zip file containing your source files and submit it to the “Assignment_0_student_submission” folder of IVLE workbin. The file name should be **“<Student number>.zip”** where **<Student number>** is your matriculation number which starts with letter A. 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 try to find your actual source files if we detect this 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. Confirmed breach may result in zero mark for the assignment and further disciplinary action from the school.

For your information, about 40 plagiarism cases were caught last year.

Question & Answer

If you have any doubts on this assignment, please post your questions on IVLE 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 or give you necessary directions.

Exercise 1

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). Read an IP address in the bit-sequence form as a command-line argument and convert it to the dotted decimal format that we normally see. A 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, 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.

To convert binary numbers to decimal numbers, remember that both are positional numerical systems, whereby the first 8 positions of the binary systems are:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Therefore, 00000011 = (0*128) + (0*64) + (0*32) + (0*16) + (0*8) + (0*4) + (1*2) + (1*1)

= 3

Name your program as **`IPAddress.py`**.

Two sample runs are shown below, with user input highlighted in **blue**.

Sample run #1:

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

This line executes your Python 3 program. It's not part of the output.

Sample run #2:

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

Exercise 2

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 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 shown below.

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

Note: you may encounter an out-of-memory error if you attempt to load a huge file into the memory as above. However, in this exercise we assume the file we process is not that big.

(To C/C++ users: you can use the **crc32()** function from **zlib** as well, and we will link your program with zlib using **-lz**.)

(To Java users: you can import and use the **java.util.zip.CRC32** class.)

Sample run:

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

Exercise 3

Write a program **XorCat.py** that reads a binary file in a streaming manner, modifies every byte by an xor operation with a given byte, and outputs the results to standard output. Here "streaming" means that you should use a (small) byte buffer to read and process the file buffer-by-buffer until encountering end of file, instead of working on the entire file content at once. In this way, the memory usage is fully in control, and the program can thus handle very large files without problems. The name "XorCat" means that this program is similar to the **cat** utility on unix, plus the xor feature. Your program should take in two command-line arguments **`<src>`** and **`<mask>`**, where **`<src>`** is the path to the file and **`<mask>`** is an integer in the range [0, 255] (i.e. the range of an unsigned byte) that is used as one operand for xor. **`<src>`** should be placed in the same folder as **XorCat.py**.

Python 3 exposes the low-level file APIs from the operating system through its IO interfaces. A file **open()**'ed for binary reading keeps the current reading position in the file. The position is initialized to 0, i.e. the beginning, when the file is opened. When the file is read (partially), the position progresses so that the next call to read the file can return the data starting at the end of the already read part. **readinto()** is a useful method of the file object for this task. It **read**s file data starting at the current position **into** a given fixed-size buffer, and returns the number of bytes actually read (see https://docs.python.org/3/library/io.html#io.BufferedIOBase.readinto for details). When there are not enough bytes remaining in the file to fill the buffer, all of the remaining data are read into the buffer, and the returned number is smaller than the buffer size. If the returned number is 0, we know that the end of file is encountered. Below is a code snippet to read a binary file buffer by buffer with buffer size being 1024 bytes:

```
buffer = bytearray(1024)  # allocate the buffer
with open("test.jpg", "rb") as f:
    numBytesRead = f.readinto(buffer)
    while numBytesRead > 0:
        # process the bytes in buffer here, up to
        # numBytesRead bytes
        numBytesRead = f.readinto(buffer)
    # end of file encountered
```

Since bytes in a **bytearray** are mutable, to apply xor operation on every byte in a **bytearray**, we can simply iterate through all bytes and do the operation for each byte:

```
# i is an iterating index for buffer
buffer[i] = (buffer[i] ^ mask) # "^" is the xor operator
```

To write binary data to standard output (stdout), you should use **sys.stdout.buffer.write()** instead of **sys.stdout.write()**, as the latter is in text mode. An example would be:

```
sys.stdout.buffer.write(buffer)
```

stdout can be treated as a write-only file, and it preserves the order of sequential writes of data. Therefore, you can simply do read-process-write in the main loop of your program.

Note that an opened file needs to be closed manually after usage, unless the **with** statement is used (as above), because the file is automatically closed when the entire block of **with** statement is finished. Therefore, it is recommended to use **with** statements when operating on files.

Sample run #1:

```
python3 XorCat.py test.jpg 210 > test.jpg.xor210
cmp test.jpg.xor210 test-jpg-xor210-ans
```

(Note 1: if you run the first command without the trailing "**> test.jpg.xor210**", the binary content would be output to the terminal which results in unreadable garbled outputs. The trailing part saves the standard output of your program into the specified file instead of showing the data to the terminal.)
(Note 2: the **cmp** command **comp**ares two binary files, and it outputs nothing if the two files have exactly the same content.)

Sample run #2:

```
python3 XorCat.py test-IPAddress.sh 123 > test-IP.sh.xor123
```