Review – Collection of Data

- List: []
  - Ordered elements, mutable elements, mutable container
- Set: {} set()
  - Unordered elements, immutable elements, mutable container
- Tuple: ()
  - Ordered elements, mutable elements, immutable container
- Dictionary: {}
  - Unordered elements, immutable keys and mutable values, mutable container

File IO – Basics

Lecture 7

Hyung-Sin Kim

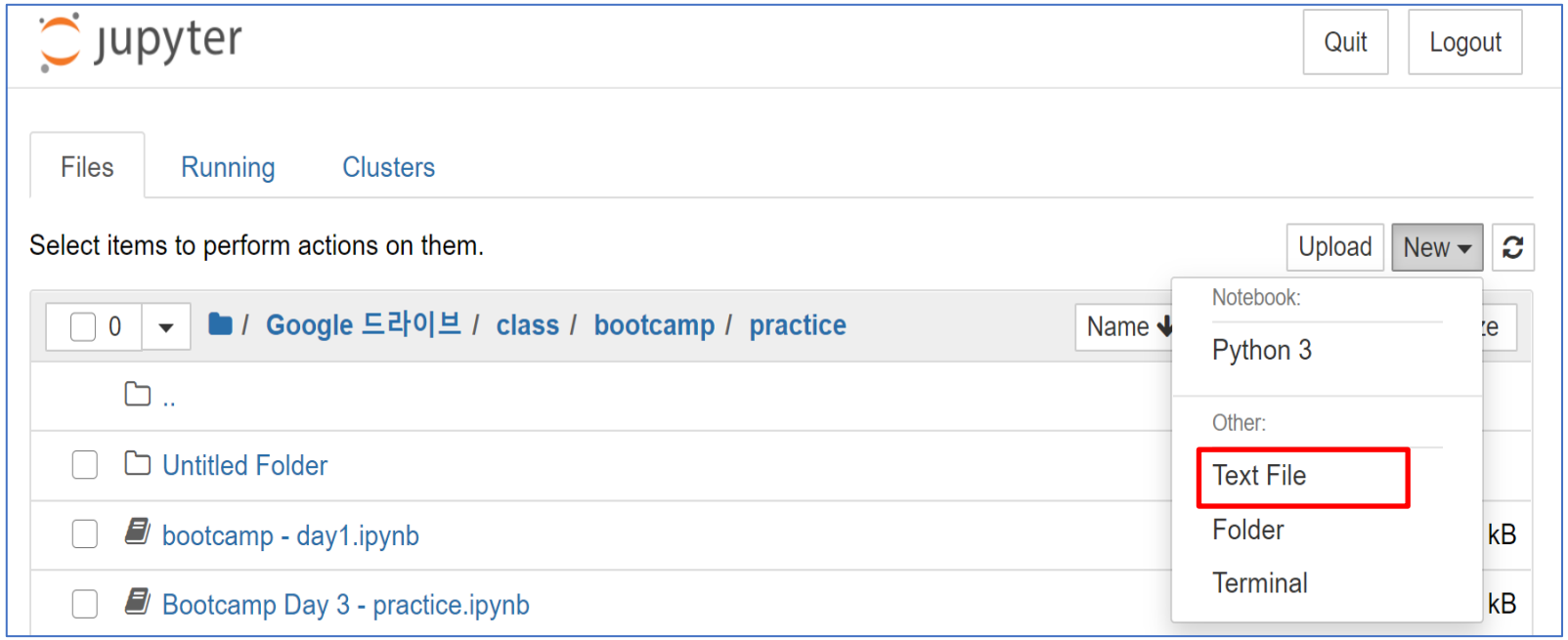
SNU Graduate School of Data Science

Introduction

- So far, we have made our own data by using various structures, string, list, tuple, set, and dictionary
- In the real world, there are lots and lots of data in files
- Once you can read and write data from/to files, you can work on real data!
- Among many kinds of files, text, music, video, ppt, word, etc., we will focus on **text files**

Opening a File – Do it!

- Make a directory, perhaps called “practice”
- In jupyter, select New->Text File and type the following:
  - First line of text
  - Second line of text
  - Third line of text



- Save this file in your current directory under the name **file_example.txt**

Opening a File – Do it!

- Open a python cell and type this program:
  - file = open(‘file_example.txt’, ‘r’)
  - contents = file.read()
  - file.close()
  - print(contents)

- Do you see something? ☺

Opening a File

- file = open(<<file_name>>, mode)
  - Open a file and return an object that knows how to get information from the file
  - This object also keeps track of **current location**: how much you've read and which part of the file you are about to read next
    - **File cursor**: The marker that keeps track of the current location in the file (initially at the beginning of the file and moving to the end as you read or write data)
  - Mode: **'r'** is for reading, **'w'** is for writing, and 'a' for appending
- file.read()
  - Read the entire file, make the contents into a string, and return the string
- file.close()
  - Release all resources associated with the open file object

Opening a File – With Statement

- A common programming pattern
  - (1) Get access to a resource, (2) do something with the resource, and (3) tidy up and release the resource
  - open() – access, read() – do something, close() – release
  - <u>But it is possible for close() to **not be executed** due to some errors</u>

- A safer way is to implement same function by using '**with**' statement
  - with open('file_example.txt', 'r') as file:
  - contents = file.read()
  - print(contents)

  - Resource is released when the program leaves the 'with' statement

File Path

- **Absolute** path: If file is in the following absolute path
  - /User/hskim/Desktop/practice/file_example.txt
- **Relative** path: Starting from the <u>current</u> working directory
  - If you are at /User/hskim/Desktop, just type "practice/file_example.txt"
  - If you are at /User/hskim/Desktop/temp, type "../practice/file_example.txt"
    - ../ lets you go up!

- Some functions about directory in module "os"
  - >>> import os
  - >>> os.getcwd() # show the current working directory
  - >>> os.chdir("/User/hskim/Desktop")

Reading Contents in Files

- When you want to read **characters** and put them into a single string
  - with open(“file_example.txt”, 'r’) as file:
  - contents = file.**read**()
    - Read from the current cursor location to the end of file, the file cursor moves to the end of file
  - contents = file.read(10)
    - Read 10 characters from the current cursor location, the file cursor moves to the 11-th character
  - If you want to read what you have already read again, close and open the file again
- When you want to make a list of strings containing **individual lines**
  - lines = file.**readlines**()
    - Output example: ['First line of text.\n’, 'Second line of text.\n’, ‘Third line of text.\n’]
    - Output contains whitespace characters. To remove them, you need to you strip method: **lines.strip()**
  - Now that you have a list of strings, you can do **reversed**(lines) or **sorted**(lines)

Reading Contents in Files

- When you want to do something for a **single** line
  - with open(“file_example.txt”, ‘r’) as file:
  - line = file.**readline**()
  - <<do something with the single line>>

- When you want to **repeat** something for every line
  - with open(“file_example.txt”, ‘r’) as file:
  - for line in file:
  - <<do something with the single line>>

Reading Contents in Files

- When you work on a file over Internet!
  - import urllib.request
  - url = "https://robjhyndman.com/tsdldata/ecology1/hopedale.dat"
  - with urllib.request.urlopen(url) as file:
  - for line in file:
  - line = line.strip()
  - **line = line.decode("utf-8")**
  - print(line)
- Since there are so many data types in a file in the Internet, you don't know if the file has text or music…
  - Python's read or readline methods return a **bytes** type, instead of string
  - You need to decode bytes type to use it as string

Reading Contents in Files

- Bit
  - To a computer, information is nothing but bits (0 or 1)
  - All data (characters, sounds, and pixels) are represented as sequences of bits

- Byte
  - A larger unit of data - 8 bits
  - Programming languages interpret bytes for users to think them as integers, strings, functions, and documents…

- Since Byte is the fundamental data unit, regardless of data types, Python reads information on a webpage as bytes

Writing Files

- with open(‘file_example.txt’, ‘w’) as output_file:
- output_file.write(‘Programming for Data Science’)
  - Write characters to a file and returns the number of characters written
  - In the write mode (‘w’), write from the beginning of the file

- with open(‘file_example.txt’, ‘a’) as output_file:
- output_file.write(‘ is fun…?’)
  - In the append mode (‘a’), write from the end of the file

- You need to manually add “\n” to change the line

Reading and Writing Together

- with open(‘file_input.txt’, ‘r’) as input_file, open(‘file_output.txt, ‘w’) as output_file:
- <<read input_file, do something, and write to output_file>>

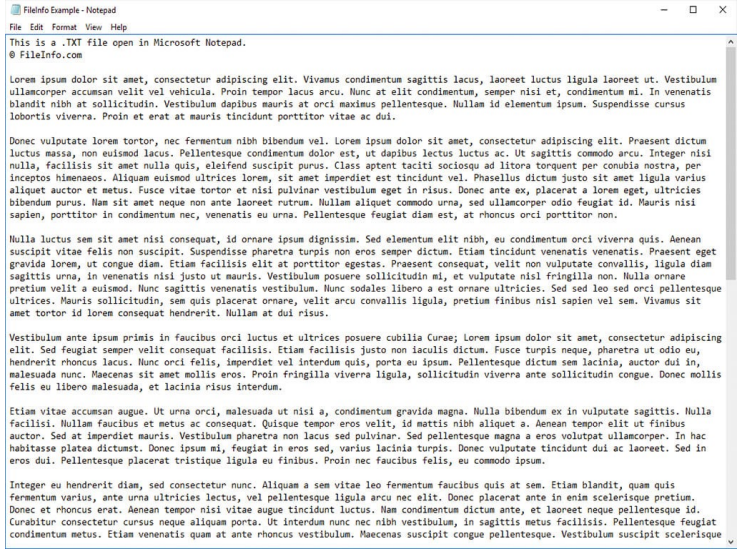
File IO – Reading Techniques

Lecture 7

Hyung-Sin Kim

SNU Graduate School of Data Science

Now let's extract data from a file and organize them as data structures we've learned!



List
Set
Tuple
Dictionary

Reading Techniques – Whitespace-delimited

- A file example for student grading
  - Inhoe 2021-1111 A+
  - Jaewook 2021-2222 A+
  - …

- A line can contain many different information, which are divided by **whitespace**

One line has **three different types** of information:
name, number, and grade

We want to handle each information separately!

Reading Techniques – Whitespace-delimited

- Whitespace handling

```
with open('file_grade.txt', 'r') as students:
        process_file(students)

def process_file(input_file: TextIO) -> None:
        line = input_file.readline()
        while line:
                line = line.strip()
                value = line.split()
                names.append(value[0])
                numbers.append(value[1])
                grades.append(value[2])
                line = input_file.readline()
```

A separate function for processing a file

from typing import TextIO

Remove "\n" at the end of the line

Break a line into a list of words
['inhoe', '2020-1111', 'A+']

Reorganize the words as three different lists

One more step into the dirty real world

Reading Techniques – Skipping the Header

- A file example for student grading
  - # This file is for a GSDS course "computing foundations for data science."
  - # This file is created by Hyung-Sin Kim
  - # This file contains students' grades.
  - Inhoe 2021-1111 A+
  - Jaewook 2021-2222 A+
  - …

- Many files include **header** to describe what they are, which is not data

We do not want to read the header!

Reading Techniques – Skipping the Header

- A solution for skipping the header

```
def process_file(input_file: TextIO) -> None:
        line = skip_header(input_file)        # The first useful line after skipping header
        while line:
            line = line.strip()
            value = line.split()
            names.append(value[0])
            numbers.append(value[1])
            grades.append(value[2])
            line = input_file.readline()

def skip_header(input_file: TextIO) -> str:
        line = input_file.readline()
        while line.startswith('#'):
            line = input_file.readline()
        return line
```

Reading Techniques – Skipping the Header

- A solution that does not work
  - def process_file(input_file: TextIO) -> None:
  - skip_header(input_file) # Skip header
  - <<Do something useful with real data>>

  - def skip_header(input_file: TextIO) -> None:
  - while line.startswith('#'):
  - input_file.readline()

  - When skip_header ends…

Cursor is at start of the second useful line

```
# This file is for a GSDS course "computing foundations for data science."
# This file is created by Hyung-Sin Kim
# This file contains students' grades.
Inhoe 2021-1111 A+
Jaewook 2021-2222 A+
```

One more step into the dirty real world

Reading Techniques – Handling Missing Values

- A file example for student grading
  - # This file is for a GSDS course "computing foundations for data science."
  - # This file is created by Hyung-Sin Kim
  - # This file contains students' grades.
  - Inhoe 2021-1111 A+
  - -
  - Jaewook 2021-2222 A+
  - …

- Data in your file are not guaranteed to be ideal. There can be many **missing values** and **typos**

We do not want to process the weird values!

Reading Techniques – Handling Missing Values

- Ignore missing values

```
def process_file(input_file: TextIO) -> None:
        line = skip_header(input_file)        # The first useful line after skipping header
        while line:
            line = line.strip()

            if line != '-':
                value = line.split()
                names.append(value[0])
                numbers.append(value[1])
                grades.append(value[2])

            line = input_file.readline()
```

Summary

- Files can be read, written to, and added to
- Contents are commonly stored in list of **strings**
- Three common stages for reusability: Input, processing, and output stages
- To make the functions usable by different types of readers, the reader is opened outside the function and passed as an argument
- TextIO is used in type annotations to indicate an open file

Thanks!