APSC-5984 Lab 4: File system

Due: 2023-02-13 (Monday) 23:59:59

0. Overview

In this lab, you will learn basic Python commands to navigate the file system and manipulate files using the `os` library. Coupling the basic knowledge of interacting with the file system, you will be able to read and write data from/to files through Python.

1. Path

A path is a string that represents the location of a file or folder in the file system. Example path strings are `/home/niche/folder_1`.

1.1 Root directory

The root directory is the top-most directory in the file system. It is represented by a single slash `/` at the beginning of the path. In the example `/home/niche/folder_1`, the root directory contains `home` directory.`folder_1` directory.

1.2 Slash '/'

A slash `/` other than the root directory is used to separate the directories in a path and to represent the parent-child relationship between directories. For example, `home` directory is the parent directory of `niche` directory.

1.3 Working directory (WD)

The working directory (WD) is a directory where you are currently working in. You can also understand it as your current location in the file system.

1.4 Relative path vs. absolute path

An absolute path always starts with the root directory `/`, while a relative path always starts with the current WD without `/` at the beginning.

2. Path implementation

For example, let's structure the file system as follows:

```
root
└──── home
        └── niche
            ├── folder_1
            ├── file_1.txt
            └── file_2.txt
```

2.1 Shell

Common commands you will use:

- `ls`: list the files and folders in the current WD.
- `pwd`: print the current WD.
- `cd`: change the current WD.
- `mkdir`: create a new directory.
- `.`: an alias for the current WD.
- `..`: an alias for the parent directory of the current WD.
- Tilde sign `~` : an alias for the home directory of the current user.

Assume you are in the root directory `/`, you can use `pwd` to check your current location.

```
pwd
# output: /
```

Use `ls` to list the files and folders in the current WD.

```
ls
# output: home
```

If you want to change your current location to the `niche` directory:

```
cd home/niche
pwd
# output: /home/niche
```

Go back to the root directory `/`:

```
cd ../..
pwd
# output: /
```

`/home/<user_name>` is usually the home directory of the current user. You can use `~` to represent it.

```
cd ~
pwd
# output: /home/niche
```

2.2 Python

In Python, to interact with the file system, you need to import the `os` library.

```
import os
```

Here are the common Python `os` methods to interact with the file system:

- `os.getcwd()`: get the current WD
- `os.listdir()`: list the content of a directory (default: WD)
- `os.chdir()`: change the WD

`os.getcwd()` is equivalent to the `pwd` command in shell.

```
os.getcwd()
# output: '/home/niche'
```

`os.listdir()` is equivalent to the `ls` command in shell.

```
os.listdir()
# output: ['file_1.txt', 'file_2.txt', 'folder_1']
```

`os.chdir()` is equivalent to the `cd` command in shell.

```
os.chdir('/home/niche/folder_1')
os.getcwd()
# output: '/home/niche/folder_1'
```

`..` also works in Python.

```
os.chdir('../..')
os.getcwd()
# output: '/home'
```

The `os.path.join()` method is a convenient way to join multiple paths together. It is recommended to use over explicitly typing the path in a string is because it is OS-agnostic. For example, if you are using Windows, the path separator is `\` instead of `/`. Using `os.path.join()` will automatically adjust the path separator based on your OS.

```
os.path.join('home', 'niche', 'folder_1')
# output: 'home/niche/folder_1' if you are using Linux
# output: 'home\\niche\\folder_1' if you are using Windows
```

3. Interacting with files

Now that you know how to navigate the file system and basic knowledge of paths, we can start to interact with files in Python!

3.1 Create a file (mode `w`)

To create a file, we can use the `open()` method. The `open()` method takes two arguments: the path to the file and the mode. The path just works as the way we learned in the previous section, it can be either absolute or relative path. The mode is a string that specifies how you want to interact with the file. The most common modes are:

- `r`: read-only mode
- `w`: write-only mode
- `a`: append-only mode

We will go through each mode in the following section. Let's start with the `w` mode. The `w` mode will create a new file if the file does not exist, or overwrite the existing file if the file already exists. We can create a file named `file.txt` in the current WD using the following code:

```
f = open('file.txt', 'w')
f.close()
```

We always need to use `close()` to release the file resource after we are done with the file. Otherwise, the file will be locked and you will not be able to access it. Let's try writing something in this file:

```
f = open('file.txt', 'w')
f.write('Hello world!\n')
f.write('This ia the second line\n')
f.write('This ia the third line\n')
f.close()
```

It is noteworthy that the `write()` method will not automatically add an end of line (EOL) character (`\n`) at the end of the line. Therefore, we need to add it manually.

If you open the file `file.txt` in a text editor, you should see the following content:

```
Hello world!
This ia the second line
This ia the third line
```

3.2 Append to a file (mode `a`)

The `a` mode is similar to the `w` mode, except that it will append the content to the end of the file instead of overwriting the existing content. Let's try appending something to the file `file.txt`:

```
f = open('file.txt', 'a')
f.write('This ia the fourth line\n')
f.close()
```

Check the file again to see if the content is appended as expected.

3.3 Read a file (mode `r`)

The `r` mode is used to read the content of a file. There are two ways to read the content of a file:

- `read()` : read the entire content of the file.
- `readlines()`: store each line of the file as an element in a list.

Let's take a look of the first method `read()`:

```
f = open('file.txt', 'r')
content = f.read()
print(content)
f.close()
# output: 'Hello world!\nThis ia the second line\nThis ia the third
line\n'
```

You might notice that the entire file content, including the EOL characters `\n`, is stored in the variable `content`. This approach works fine when ones want to access the raw information, but it is not covenient when we want to process the content line by line. Hence, we can use the `readlines()` method:

```
f = open('file.txt', 'r')s
lines = f.readlines()
print(lines)
f.close()
# output: ['Hello world!\n', 'This ia the second line\n', 'This ia the
third line\n']
```

The file contents are split into a list of lines automatically. You can use `len()` method to count how many lines are there in the file:

```
len(lines)
# output: 3
```

4. String processing

String processing is an essential skill to extract meaningful information from a text-based file. In this section, we will go through some common string processing methods.

4.1 String slicing

We have introduced the data type `string` and the related operators the previous lab. In this section, you will apply the operation to a list of strings. First, let's create an example list:

```
file_list = [
            "2022/09/08_trialA_trt1.txt",
            "2022/09/15_trialA_trt2.txt",
            "2022/09/28_trialB_trt1.txt",
            "2022/09/30_trialB_trt2.txt",
            "2022/10/01_trialC_trt1.txt",
            "2022/10/08_trialC_trt2.txt"]
```

In this example, you are provided with a list of file names. You might soon notice that the file names are coded in the following format: `YYYY/MM/DD_trialX_trtY.txt`. In this practice, we want to extract the date, trial, and treatment information from the provided list.

We know that the file extension names are always occupied the last three characters of the file. We can use the indexing operation we learned earlier to extract the extension name:

```
file_list[0]
# output: '2022/09/08_trialA_trt1.txt'
```

```
file_list[0][-3:]
# output: 'txt'
```

The slicing index `-3:` means we want every character from the third last character to the end of the string. To generate a list of the extension names, you can use a `for` loop:

```
extension_list = []
for file in file_list:
    extension_list.append(file[-3:])
print(extension_list)
# output: ['txt', 'txt', 'txt', 'txt', 'txt', 'txt']
```

In this example, the `file` variable is assigned to each element (file name) in the list `file_list`. Then we use the `append()` method to add the extracted information (extension name) to the target list `extension_list`.

4.2 String split

Now, we want to break each string to extract other information such as trials and treatments. We noticed that they are separated by the underscore `_`. We can use the `split()` method that takes the separator (i.e., `_`) as the argument to split the string:

```
filename = file_list[0]
filename.split('_')
# output: ['2022/09/08', 'trialA', 'trt1.txt']
```

We can apply the same logic to the first element of the string to extract date information:

```
filename = file_list[0]
elements = filename.split('_')
date = elements[0]
print(date)
# output: '2022/09/08'
yyyymmdd = date.split("/")
print(yyyymmdd)
# output: ['2022', '09', '08']
```

4.3 String replace

From the split result, we can see that the treatment information is not in the format we wnat. Eech substring `trt1.txt` still tails with the file extension name `.txt`.

```
filename = file_list[0]
elements = filename.split('_')
```

```
trt = elements[2]
print(trt)
# output: 'trt1.txt'
```

We have two strategies to remove the file extension name. The first one is to use a slicing operation:

```
new_trt = trt[:-4]
print(new_trt)
# output: 'trt1'
```

Or, we can use the `replace()` method to replace the file extension name with an empty string:

```
new_trt = trt.replace('.txt', '')
print(new_trt)
# output: 'trt1'
```

4.4 Regular expression (RE)

If the string pattern is more complicated than position-based slicing or replacement, you want to use regular expression to implement a more flexible pattern recognition. We will introduce the `re` module that provides a set of methods to implement regular expression.

As always, we need to import the `re` module first:

```
import re
```

The `re` module provides a set of methods to implement regular expression. In this lab, we will only use `re.findall()` method, which returns a list of all the non-overlapping matches in the string. The syntax of the method is:

```
re.findall(pattern, string)
```

The `pattern` is the regular expression pattern, and the `string` is the string to be searched. So, what is a regular expression pattern? Here is a table of some common regular expression patterns:

Pattern	Description
`.`	Matches any character except newline.
`^`	Matches the start of the string.
`$`	Matches the end of the string or just before the newline at the end of the string.
`*`	Matches 0 or more (greedy) repetitions of the preceding RE.

Pattern	Description
`+`	Matches 1 or more (greedy) repetitions of the preceding RE.
`?`	Matches 0 or 1 (greedy) of the preceding RE.
`\w`	Matches any alphanumeric character including the underscore.
`\W`	Matches any non-alphanumeric character.
`\d`	Matches any numeric digit.
`\D`	Matches any non-numeric digit.
`\s`	Matches any whitespace character.
`\S`	Matches any non-whitespace character.
`[]`	Matches any character in the brackets.
`{m}`	Matches exactly m copies of the previous RE.
`{m,n}`	Matches from m to n (inclusive) copies of the previous RE.

It is toally normal to get confused by the table above when you first see it. Don't worry, we will go through some examples to understand how to use regular expression.

Here is an example:

```
filelines = [
    "File name: 2022/09/08_trialA_trt1.txt",
    "treatment A: (1, 3) and (2, 4)",
    "treatment B: (5, 1, 8)",
    "treatment C: (2, 4), (1, 9, 8)"]
```

The data `filelines` was intentionally made to be a little bit messy. We want to extract the treatment information in each pair of parenthesis.

```
trts = []
for line in filelines:
    value = re.findall(r"\([\d\s,]*\)", line)
    trts.append(value)
print(trts)
# output: [[], ['(1, 3)', '(2, 4)'], ['(5, 1, 8)'], ['(2, 4)', '(1, 9,
8)']]
```

Let's break down the RE pattern:

- `\(` and `\)`: matches a left parenthesis and a right parenthesis, respectively. The back slash `\` is used to escape the special character `(` and `)`.

- `[\d\s,]`: Since the treatment information is a list of numbers separated by commas, we can use the square brackets `[]` to match any character in the brackets. The pattern `[\d\s,]` means:
    - `\d`: matches any digit.
    - `\s`: matches any whitespace.
    - `,`: matches a comma.
- `*`: Since there is no specific number of digits, we can use the `*` to match 0 or more repetitions of the preceding RE.