

APSC-5984 Lab 4: File system

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

- APSC-5984 Lab 4: File system
 - 0. Overview
 - 1. Path
 - 1.1 Root directory
 - 1.2 Slash '/'
 - 1.3 Working directory (WD)
 - 1.4 Relative path vs. absolute path
 - 2. Path implementation
 - 2.1 Shell
 - 2.2 Python
 - 2.3 Exercise
 - 2.3.1 Create a base directory
 - 2.3.2 Change the WD
 - 2.3.3 Create a new directory
 - 2.3.4 Use relative path to change the WD
 - 2.3.5 Use relative path to create a new directory
 - 3. Interacting with files
 - 3.1 Create a file (mode `w`)
 - 3.1.1 Write to a file
 - 3.1.2 End of line (EOL)
 - 3.1.3 Close the file and `with` statement
 - 3.2 Append to a file (mode `a`)
 - 3.3 Read a file (mode `r`)
 - 4. String processing
 - 4.1 String slicing
 - 4.2 String split
 - 4.3 String replace
 - 4.4 Regular expression (RE)
 - 4.4.1 Find all numbers
 - 4.4.2 Find numbers that match a string pattern
 - 4.4.3 Greedy vs. non-greedy matching
 - 4.4.4 Exclusions
 - 4.4.5 Another example

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
```

Create a new directory `folder_1` in the current WD:

```
ls
# output: file_1.txt file_2.txt folder_1
mkdir folder_2
ls
# output: file_1.txt file_2.txt folder_1 folder_2
```

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.mkdir()`: create a new directory
- `os.path.join()`: join multiple path strings into a single path string

`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'
```

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

```
os.chdir('/home/niche')
os.listdir()
# output: ['file_1.txt', 'file_2.txt', 'folder_1']
os.mkdir('folder_2')
os.listdir()
# output: ['file_1.txt', 'file_2.txt', 'folder_1', 'folder_2']
```

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
```

2.3 Exercise

We will create a folder structure as follows:

```
exercise  
├── src  
│   ├── module_1  
│   ├── module_2  
│   └── module_3  
├── data  
│   └── trial_A  
│       ├── 2022_03_02  
│       ├── 2022_03_03  
│       └── 2022_03_04  
│   └── trial_B  
└── out
```

2.3.1 Create a base directory

Create a new directory named `exercise` and change the WD to the `exercise` directory.

2.3.2 Change the WD

Create a new directory named `src` in the `exercise` directory.

Change the WD to the `src` directory.

2.3.3 Create a new directory

Create new directories named `module_1`, `module_2`, and `module_3` in the `src` directory.

2.3.4 Use relative path to change the WD

Change the WD to the `exercise` directory.

2.3.5 Use relative path to create a new directory

Create new directories named `data`, `data/trial_A`, and `data/trial_B` in the `exercise` directory.

Change the WD to the `data/trial_A` directory.

Create new directories named `2022_03_02`, `2022_03_03`, and `2022_03_04` in the `exercise/data/trial_A` directory.

While in the `trial_A` directory, use relative path to create a new directory named `out` in the `exercise` directory.

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()
```

3.1.1 Write to a file

Let's try writing something in this file using the `write()` method:

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

And here is the file content:

```
cat file.txt
# Hello world!
# This is the second line
# This is the third line
```

3.1.2 End of line (EOL)

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.

Here is an example not including the EOL character:

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

The file content:

```
cat file.txt
# Hello world!This ia the second lineThis ia the third line
```

3.1.3 Close the file and `with` statement

We always want to use `close()` to release the file resource after we are done with the file. Let's see what happens if we do not use `close()`:

```
f1 = open('file.txt', 'w')
f1.write("Hello World! from f1\n")
```

Use shell to check the file content:

```
cat file.txt
```

The file will only be created but not written anything. This is because the file resource is not released until the program is terminated. Therefore, we need to use `close()` to let the computer know that we are done with the file and the content can be saved.

In Python, we can use `with` statement to automatically close the file

```
with open('file.txt', 'w') as f1:
    f1.write("Hello World! from f1\n")
```

Check the file content again:

```
cat file.txt
# Hello World! from f1
```

Here is another case without using `close()` properly where you have more than two file streams opened at the same time:

```
# create the first file stream
f1 = open('file.txt', 'w')
f1.write("--- F1 ---\n")
# create the second file stream
f2 = open('file.txt', 'w')
f2.write("--- F2F2F2 ---\n")
# close the first file stream
f1.close()
```

Check the file content:

```
cat file.txt
# --- F1 ---
```

Close the second file stream:

```
# close the second file stream
f2.close()
```

Check the file content again:

```
cat file.txt
# --- F2F2F2 ---
```

You will find that the content of the file is overwritten by the second file stream.

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 first create a file named `file.txt`:

```
f = open('file.txt', 'w')
f.write('Hello world!\n')
```



```
f.write('This is the second line\n')
f.write('This is the third line\n')
f.close()
```

And we can show the content in Shell:

```
cat file.txt
# Hello world!
# This is the second line
# This is the third line
```

Now, let's append some content to the file:

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

Check the file content:

```
cat file.txt
# Hello world!
# This is the second line
# This is the third line
# This is the fourth line
# This is the fifth line
```

You should notice that the first three lines were not overwritten by the new content.

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 convenient when we want to process the content line by line. Hence, we can use the `readlines()` method:

```
f = open('file.txt', 'r')
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 common string processing methods.

4.1 String slicing

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

```
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"]
```

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 each filename.

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 down each string to smaller pieces of information such as trials and treatments. We noticed that they were separated by the underscore `_`. Hence, we can use the `split()` method that takes the separator (i.e., `_`) as the argument to split the string:

```
filename = file_list[0] # '2022/09/08_trialA_trt1.txt'  
filename.split('_')  
# output: ['2022/09/08', 'trialA', 'trt1.txt']
```

We can apply the same logic to the first substring `2022/09/08` to extract detailed date information:

```
filename = file_list[0] # '2022/09/08_trialA_trt1.txt'  
elements = filename.split('_')  
print(elements)  
# output: ['2022/09/08', 'trialA', 'trt1.txt']  
date = elements[0]  
print(date)  
# output: '2022/09/08'  
  
yyyymmdd = date.split("/")  
print(yyyymmdd)  
# output: ['2022', '09', '08']  
  
year = yyyymmdd[0]  
month = yyyymmdd[1]  
day = yyyymmdd[2]  
print("File name = " + filename)  
print("Year = " + year)  
print("Month = " + month)
```

```
print("Day = " + day)
# output:
# File name = 2022/09/08_trialA_trt1.txt
# Year = 2022
# Month = 09
# Day = 08
```

4.3 String replace

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

```
filename = file_list[0] # '2022/09/08_trialA_trt1.txt'
elements = filename.split('_')
print(elements)
# output: ['2022/09/08', 'trialA', 'trt1.txt']
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)

Regular expression (RE) is a more flexible pattern recognition method than simple string slicing or replacement. 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
<code>.</code>	Matches any character except newline.
<code>^</code>	Matches the start of the string.
<code>\$</code>	Matches the end of the string or just before the newline at the end of the string.
<code>*</code>	Matches 0 or more repetitions (greedy) of the preceding RE.
<code>+</code>	Matches 1 or more repetitions (greedy) of the preceding RE.
<code>?</code>	Matches 0 or 1 (greedy) of the preceding RE.
<code>*?</code>	Matches 0 or more repetitions (non-greedy) of the preceding RE.
<code>+?</code>	Matches 1 or more repetitions (non-greedy) of the preceding RE.
<code>??</code>	Matches 0 or 1 (non-greedy) of the preceding RE.
<code>\w</code>	Matches any alphanumeric character including the underscore.
<code>\W</code>	Matches any non-alphanumeric character.
<code>\d</code>	Matches any numeric digit.
<code>\D</code>	Matches any non-numeric digit.
<code>\s</code>	Matches any whitespace character.
<code>\S</code>	Matches any non-whitespace character.
<code>[]</code>	Matches any character in the brackets.
<code>{m}</code>	Matches exactly m copies of the previous RE.
<code>{m,n}</code>	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 RE works.

4.4.1 Find all numbers

Example sentence from the USDA web page

(<https://www.aphis.usda.gov/aphis/ourfocus/animalhealth/animal-disease-information/cattle-disease-information/cattle-surveillance>)

From 2006 to 2015, the BSE Ongoing Surveillance Program tested approximately 40,000 samples per year.

Assign the sentence to a string variable and use RE to extract numbers:

```
string = "From 2006 to 2015, the BSE Ongoing Surveillance Program tested  
approximately 40,000 samples per year."  
re.findall(r"\d+", string)  
# output: ['2006', '2015', '40', '000']
```

This example shows how to find all numbers in a string. We use `\d` to match any numeric digit, and `+` after that to match one or more copies of the preceding RE (i.e., `\d`).

However, it is not an ideal result if we only want to extract the year information. We know that the year information is always a four-digit number. Hence, we can explicitly specify the number using `{m}`:

```
re.findall(r"\d{4}", string)  
# output: ['2006', '2015']
```

4.4.2 Find numbers that match a string pattern

Here is another example sentence

(https://www.aphis.usda.gov/animal_health/animal_diseases/brucellosis/downloads/fpa-val-rpt.pdf)

In fact, these samples had an average reading of 254 mP at the Texas laboratory compared to the average negative control reading of 88 mP. The FP also clearly identified the four sero-negative samples (samples 1, 7, 17, and 19) as negative.

If you want to extract each mP read, simply using `\d+` will not work as you will also get numbers about the sample id (e.g., 1, 7, 17, and 19).

```
re.findall(r"\d+", string)  
# output: ['254', '88', '1', '7', '17', '19']
```

Here you want to add additional string pattern to narrow down the search:

```
re.findall(r"\d+ mP", string)  
# output: ['254 mP', '88 mP']
```

or you can only keep the 3-digit number:

```
re.findall(r"\d{3} mP", string)
# output: ['254 mP']
```

4.4.3 Greedy vs. non-greedy matching

Let's take a look of this paragraph

The FP for B. abortus relies on an O-polysaccharide from B. abortus that is covalently linked with a fluorescein isothiocyanate tracer molecule. Infected cattle were those from which B. abortus had been cultured from milk or tissues. Serum samples were also assayed from B. abortus strain 19 vaccinated cattle at various times post vaccination. The FP test was shown to accurately classify uninfected cattle as negative, infected cattle as positive, and strain 19 vaccinated cattle as negative.

Say we want to extract every sentence that contains the word "B. abortus". We can design a RE pattern like this:

```
re.findall(r".*B\. abortus.*\.", string)
# output: B. abortus relies on an ... cattle as negative.
```

Break down the RE pattern:

- `B\. abortus`: match the word "B. abortus"
- `.*`: match any character until it reaches a period `.`
- `\.`: match the period `.`

A `\` before the period `.` called an escape character. It is because `.` is a special character that represent any character in RE, and we want to match the period character itself.

This search is a greedy search, as it will match as many characters as possible; such search will return the whole paragraph instead of sentences. To fix this, we can use a non-greedy search by adding a `?` after the `*`:

```
re.findall(r".*?B\. abortus.*?\.", string)
# output [
# 'The FP for B. abortus relies on an O-polysaccharide from B.',
# ' abortus that is covalently linked with a fluorescein isothiocyanate
tracer molecule. Infected cattle were those from which B. abortus had been
cultured from milk or tissues.',
# ' Serum samples were also assayed from B. abortus strain 19 vaccinated
cattle at various times post vaccination.']
```

4.4.4 Exclusions

Ok, it seems closer to what we want. However, The first sentence that contains two "B. abortus" words confuses our search. We can further specify the context of the period . by excluding B. from the search:

```
re.findall(r".*?)B. abortus.*?[^B]\.", string)
# output: [
# 'The FP for B. abortus relies on an O-polysaccharide from B. abortus
# that is covalently linked with a fluorescein isothiocyanate tracer
# molecule.',
# ' Infected cattle were those from which B. abortus had been cultured
# from milk or tissues.',
# ' Serum samples were also assayed from B. abortus strain 19 vaccinated
# cattle at various times post vaccination.']
```

4.4.5 Another 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 (e.g., (1, 3) and (1, 9, 8)):

```
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.

