# Homework 08

Re-submit Assignment

**Due** Mar 25 by 10pm **Points** 100 **Submitting** a file upload

## Assignment Description

## Part 1: Date Arithmetic Operations

`date_arithmetic()-> Tuple[datetime, datetime, int]`

Date arithmetic is far more difficult than you might imagine. For example, what is the date three days after Feb 27, 2020? It's not Feb 30. Did February 2020 have 28 or 29 days? Given two arbitrary dates, how many days are between those two dates?

Write a function `date_arithmetic` to use Python's `datetime` module to answer the following questions:

1. What is the date three days after *Feb 27, 2020*?
2. What is the date three days after *Feb 27, 2019*?
3. How many days passed between *Feb 1, 2019* and Sept 30*, 2019*?

***IMPORTANT***: The function should return a `tuple` with three values in the following order:

1. An instance of class `datetime` representing the date three days after *Feb 27, 2020*.
2. An instance of class `datetime` representing the date three days after *Feb 27, 2019*.
3. An `int` representing the number of days between *Feb 1, 2019* and *Sept 30, 2019*

Here below I have some pseudo code as a hint:

```
from datetime import datetime, timedelta

def date_arithmetic() -> Tuple[datetime, datetime, int]:
    """ Code segment demonstrating expected return values. """
    three_days_after_02272020: datetime = # your code goes here for calculation
    three_days_after_02272019: datetime = # your code goes here for calculation
    days_passed_02012019_09302019: int = # your code goes here for calculation

    return three_days_after_02272020, three_days_after_02272019, days_passed_01012019_09302019
```

Note that the variable assignments are not mandatory, as long as you follow the PEP8 naming convention.

You may just call the appropriate `datetime` methods to answer the questions.

## Part 2: field separated file reader

`file_reader(str, int, str, bool) -> Iterator[List[str]]`

Reading text files with a fixed number of fields, separated by a specific character, is a common task. For example, .CSV files (comma separated values) are common in Windows applications while Unix data files are typically separated by the '|' (vertical bar) character.

Write a **generator** function `file_reader()` to read field-separated text files and yield a tuple with all of the values from a single line in the file on each call to `next()`. Your generator **MUST** meet the following requirements:

- You must implement a generator that reads the file **one line at a time,** splits the line into fields on the separator, and yields a tuple with each of the values in that line. You should **NOT** read the entire file into memory.
- The generator definition should include following parameters:
  - `path: str` - the path of the file to be read (Do not hardcode the file path because we'll reuse this function many times in future assignments)
  - `fields: int` - the number of fields expected in each line
  - `sep: str` - an ***optional*** parameter to specify the field separator that *defaults to comma (`','`)* (that wasn't intended as an emoji…)
  - `header: bool`- an ***optional*** boolean parameter that *defaults to* `False` to specify if the first line in the file is a header line
- The generator should raise a `FileNotFound` exception if the specified file can't be opened for reading along with a meaningful message to help the reader to understand the problem
- The generator should raise a `ValueError` exception if a line in the file doesn't have exactly the expected number of fields, e.g. "ValueError: 'foo.txt' has 3 fields on line 26 but expected 4". **The exception message should include:**
  - **the file name,** e.g. 'foo.txt'
  - **the line number in the file where the problem occurred**
  - **the number of fields found in the line**
  - **the number of fields expected in the line**
- The *n* fields from each line should be yielded as a `list of strings`
- If the call to the generator specified a header row, then the first call for data should return the second row from the file, skipping over the header row after checking that the header row has the expected number of fields in the header

Here's a sample file, student_majors.txt, that includes a header row and three fields, separated by '|': CWID, name, and major:

```
CWID|Name|Major
123|Jin He|Computer Science
234|Nanda Koka|Software Engineering
345|Benji Cai|Software Engineering
```

Here's sample code using the generator to read a vertical bar separated file with a header row and 3 fields:

```
for cwid, name, major in file_reader(path, 3, sep='|', header=True):
    print(f"cwid: {cwid} name: {name} major: {major}")
```

**Note**: Be sure to include automated tests with at least one test file, and include your test file in your submission.

**Hint**: The code is much shorter than the description of the requirements

# Part 3: Scanning directories and files

`class FileAnalyzer`

Python is a great tool for automating a number of tasks, including scanning and summarizing the files in a file system.

Write a Python class, `FileAnalyzer` that given a directory name, searches that directory for Python files (i.e. files ending with `.py`). For each .py file in the directory, open each file and calculate a summary of the file including:

- the file name
- the total number of lines in the file
- the total number of characters in the file
- the number of Python functions (lines that begin with 'def ', including methods inside class definitions)
- the number of Python classes (lines that begin with 'class ')

Generate a summary report with the directory name, followed by a tabular output that looks similar to the following:

The class `FileAnalyzer` should have **at least** these attributes and these methods listed below:

- **Attributes**:
  - `self.files_summary: Dict[str, Dict[str, int]]`, a Python dictionary that stores the summarized data for the given file path. The **key**s of dictionary will be the **filename** of a python file, the **value** of each key will be a `dict` as well. The value dictionary will have 4 key-value pairs that represent 4 data points we collect for each file. The structure of `self.files_summary` will look like below:

```
{
   'filename_0.py': {
       'class': # number of classes in the file
       'function': # number of functions in the file
       'line': # number of lines in the file
       'char': # number of characters in the file
       }, ...
}
```

- **Methods**:
  - `self.analyze_files(self) -> None`: a method that populate the summarized data into `self.files_summary`. Note that you **MUST NOT** pass any argument to this method, the given directory path is better stored as an attribute - there will be no naming requirement for this variable but I personally use `self.directory`.
  - `self.pretty_print(self) -> None`: a method that print out the pretty table from the data stored in the `self.files_summary`. Note that you **MUST NOT** pass any argument, other than self, to this method.

**Note**: Your should execute the `self.analyze_files` in the `self.__init__` but you **MUST NOT** execute the `self.pretty_print` in the `self.__init__.`

```
Summary for /Users/jrr/Documents/Stevens/810/Assignments/HW08_test
+-----------------------------------------------------------------------------------+---------+-----------+-------+------------+
|                                     File Name                                     | Classes | Functions | Lines | Characters |
+-----------------------------------------------------------------------------------+---------+-----------+-------+------------+
|  /Users/jrr/Documents/Stevens/810/Assignments/HW08_test/0_defs_in_this_file.py    |    0    |     0     |   3   |     57     |
|         /Users/jrr/Documents/Stevens/810/Assignments/HW08_test/file1.py           |    2    |     4     |   17  |    183     |
|   /Users/jrr/Documents/Stevens/810/Assignments/HW08_test/HW02-fractions-jrr.py    |    1    |     12    |  147  |    4674    |
|    /Users/jrr/Documents/Stevens/810/Assignments/HW08_test/HW02_3-fractions.py     |    2    |     15    |  151  |    4647    |
| /Users/jrr/Documents/Stevens/810/Assignments/HW08_test/HW03-fractions-example.py  |    2    |     8     |   78  |    2435    |
|     /Users/jrr/Documents/Stevens/810/Assignments/HW08_test/HW03-fractions.py      |    2    |     27    |  207  |    6355    |
|     /Users/jrr/Documents/Stevens/810/Assignments/HW08_test/HW04-fractions.py      |    2    |     30    |  242  |    7831    |
|   /Users/jrr/Documents/Stevens/810/Assignments/HW08_test/HW08-CodeAnalyzer.py     |    2    |     6     |  112  |    3625    |
+-----------------------------------------------------------------------------------+---------+-----------+-------+------------+
```

Keep in mind that you may see a file, but you may not be able to read it. In that case, raise a `FileNotFound` exception if the specified file can't be opened for reading

Your program should use exceptions to protect the program against unexpected events.

The **PrettyTable module** (https://pypi.python.org/pypi/PTable/0.9.0) provides an easy way to create tables for output. You should use PrettyTable to format your table. See the lecture notes for an example.

You'll need to think about how to test this program with unittest. You do not need to automatically verify the output from the table but you should validate the various numeric values, e.g. number of classes, functions, lines, and characters. Your automated test should validate the counts for each file in the test directory.

**Hints**:

1. Python's `os` module has several helpful functions, e.g. `os.listdir(directory)` lists all of the files in the specified directory. Note that the file names returned by `os.listdir` do **NOT** include the directory name.
2. `os.chdir(directory)` will change the current directory to the specified directory when your program runs.
3. Be sure to handle the cases where the user enters an invalid directory name
4. Test your code against the sample files in Canvas
5. Be sure to use unittest to test your program and follow the PEP-8 coding standards.

# Deliverable

# File Structure

In this assignment, you will need to **separate your code into two files** - one for code logic and one for unit test. It's always a good practice to separate the code and the test.

You are going to have **two .py files** for submission:

1. `HW08_FirstName_LastName.py`
2. `HW08_Test_FirstName_LastName.py`

In `HW08_FirstName_LastName.py` you should have following code:

```
""" HW08 implementation file layout template"""

def date_arithmetic() -> Tuple[datetime, datetime, int]:
    """ Your docstring should go here for the description of the function."""
    pass # implement your code here
```

```
def file_reader(path, fields, sep=',', header=False) -> Iterator[Tuple[str]]:
    """ Your docstring should go here for the description of the function."""
    pass # implement your code here

class FileAnalyzer:
    """ Your docstring should go here for the description of the class."""
    def __init__(self, directory: str) -> None:
        """ Your docstring should go here for the description of the method."""
        self.directory: str = directory # NOT mandatory!
        self.files_summary: Dict[str, Dict[str, int]] = dict()

        self.analyze_files() # summerize the python files data

    def analyze_files(self) -> None:
        """ Your docstring should go here for the description of the method."""
        pass # implement your code here

    def pretty_print(self) -> None:
        """ Your docstring should go here for the description of the method."""
        pass # implement your code here
```

In `HW08_Test_FirstName_LastName.py` you will define your test cases.

If you have any concerns, please do not hesitate to reach out.