SI 506: Lecture 11

TOPICS

1. `os.path`
2. Opening a file (ye olde way / recommended way)
3. Working with CSV files

Vocabulary

This week

- **Deep copying**. For a given mutable object (e.g., `list`) constructs a new compound object and recursively *copies* into it objects found in the original.
- **File Object**. An object that provides a file-oriented application programming interface (API) to a either a text file, binary file (e.g., image file), or a buffered binary file. File objects include read and write methods for interacting with a file stored locally or remotely.
- **Nested Loop**. A `for` or `while` loop located within the code block of another loop.
- **Shallow copying**. For a given mutable object (e.g., `list`) constructs a new compound object but inserts *references* (rather than copies) into it of objects found in the original.
- **Tuple packing**. Assigning items to a tuple.
- **Tuple unpacking**. Assigning tuple items to an equal number of variables in a single assignment.

Previous

- **Argument**. A value passed to a function or method that corresponds to a parameter defined for the function or method.
- **Boolean**. A type (`bool`) or an expression that evaluates to either `True` or `False`.
- **Built-in Function**. A function defined by the Standard Library that is always available for use.
- **Caller**. The initiator of a function call.
- **Conditional Statement**. A statement that determines a computer program's *control flow* or the order in which particular computations are to be executed.
- **Dictionary**. An associative array or a map, wherein each specified value is associated with or mapped to a defined key that is used to access the value.
- **Expression**. An accumulation of values, operators, and/or function calls that return a value. `len(< some_list >)` is considered an expression.
- **f-string**. Formatted string literal prefixed with `f` or `F`.
- **Function**. A defined block of code that performs (ideally) a single task. Functions only run when they are explicitly called. A function can be defined with one or more *parameters* that allow it to accept *arguments* from the caller in order to perform a computation. A function can also be designed to return a computed value. Functions are considered "first-class" objects in the Python eco-system.
- **Immutable**. Object state cannot be modified following creation. Strings are immutable.
- **Iterable**. An object capable of returning its members one at a time. Both strings and lists are examples of an iterable.
- **Iteration**. Repetition of a computational procedure in order to generate a possible sequence of outcomes. Iterating over a `list` using a `for` loop is an example of iteration.

- **Method**. A function defined by and bound to an object. For example the `str` type is provisioned with a number of methods including `str.strip()`.
- **Mutable**. Object state can be modified following creation. Lists are mutable.
- **Operator**. A symbol for performing operations on values and variables. The assignment operator (`=`) and arithmetic operators (`+`, `-`, `*`, `/`, `**`, `%`, `//`).
- **Parameter**. A named entity in a function or method definition that specifies an argument that the function or method accepts.
- **Scope**. The part of a script or program in which a variable and the object to which it is assigned is visible and accessible.
- **Sequence**. An ordered set such as `str`, `list`, or `tuple`, the members of which (e.g., characters, elements, items) can be accessed.
- **Slice**. A subset of a sequence. A slice is created using the subscript notation `[]` with colons separating numbers when several are given, such as in `variable_name[1:3:5]`. The bracket notation uses slice objects internally.
- **Statement**. An instruction that the Python Interpreter can execute. For example, assigning a variable to a value such as `name = 'arwhyte'` is considered a statement.
- **Tuple**. An ordered sequence that cannot be modified once it is created.

Data

- **umsi-faculty.txt**. List of UMSI faculty (last name, first name) as of January 2021.
- **umsi-faculty.txt**. List of UMSI faculty (first name, last name) as of January 2021.
- **resnick-publications.csv**. Select list of Professor Paul Resnick's academic publications, 1996-2019.

1.0 Using `os.path` to create paths

The standard library's `os.path` module includes a number of useful functions for constructing pathnames out of strings. The installed `os.path module` is always the `path` module suitable for the operating system Python is running on (e.g., macOS, Windows 10), and therefore usable for local paths.

```python
# Absolute path to directory in which *.py is located.
abs_path = os.path.dirname(os.path.abspath(__file__))

# Current working directory
cwd = os.getcwd()

# Construct macOS and Windows friendly paths
faculty_path = os.path.join(abs_path, 'umsi-faculty.txt')
resnick_path = os.path.join(abs_path, 'resnick-publications.csv')
```

💡 an alternative library for working with path objects is the `pathlib` module, introduced as part of the Python 3.4 release.

2.0 Opening files

2.1 Opening a file (ye olde way)

You can use the built-in `open()` function to open a file and return a file object or a file handle. At a minimum you must pass a file path to `open()` as an argument.

The following example illustrates how to open a file in "read" mode, return a file object, call its `read()` method and return a string of the text file's content.

❗ Note that you `must` call the file object's `close()` method in order to close the connection to the open file. This is a best practice because it both frees up system resources and ensures that any file changes not yet accessible due to buffering are made available.

```
faculty_path = './umsi-faculty.txt'

file_obj = open(faculty_path) # open
data = file_obj.read() # returns a single string
file_obj.close() # close (REQUIRED)
```

💡 you can include a `size` argument of type `int` to the `read()` method in order to limit the number of bytes to return.

2.2 Opening a file (the recommended way)

The `with` statement (PEP 343) is a control-flow structure that helps to factor out `try/except` statements as well as ensure that "clean up" actions such as closing an open file object occurs without the need to call the necessary file object `close()` explicitly.

💡 Recommended practice is to use the `with` statement when opening a file because the `with` statements context manager closes the file automatically once all statements within the `with` code block are executed, even if errors occur.

```
with open(faculty_path) as file_obj: # open
    data = file_obj.read() # returns a single string
```

2.3 File opening modes

You can specify the *mode* by which the built-in `open()` function opens a file. For SI 506 only the "read" (`r`) and "write" (`w`) modes will be employed for opening text, CSV, and JSON files. That said you should familiarize yourself with the other available modes, noting too that Python can work with binary content such as images and PDF files.

Character	Description	SI 506 (in scope)
'r'	open for reading (default); equivalent to `rt`	**Yes**
'w'	open for writing, truncating the file first	**Yes**
'a'	open for writing, appending to the end of the file if it exists	No

Character	Description	SI 506 (in scope)
'x'	open for exclusive creation, failing if the file already exists	No
'b'	binary mode (e.g., image, PDF), contents returned as bytes objects; `rb` = read binary; `wb` = write binary	No
't'	text mode (default)	No
'+'	open for updating (reading and writing)	No

```
with open(faculty_path, 'r') as file_obj: # open in read mode
    data = file_obj.read()

print(f"\n2.2 read mode ('r')\n{data}")

filepath = './umsi-faculty-v2.txt'
# filepath = os.path.join(abs_path, 'umsi-faculty-v2.txt')
with open(filepath, 'w') as file_obj: # open in write mode
    file_obj.write(data) # writes string to file
```

2.4 Read methods: `read()`, `readline()`, `readlines()`

The example above introduced the `read()` method, which, by default, reads the file object in its entirety and returns as a single string. In contrast, the `readline()` method reads a single line of text. You can call `readline()` n-times in order to return successive lines of text. You can also pass a `size` argument of type `int` to the `readline()` method in order to limit the number of characters to be returned

```
with open(faculty_path, 'r') as file_obj: # open in read mode
    data = file_obj.readline()
    # data += file_obj.readline() # UNCOMMENT: call n times but not
efficient
    # data += file_obj.readline() # UNCOMMENT: call n times but not
efficient
    # data += file_obj.readline() # UNCOMMENT: call n times but not
efficient
```

A more useful file object method is `readlines()`. The `readlines()` method returns a list of strings corresponding to each line in the file object.

❗ Note that each string returned includes a trailing *newline* escape sequence `\n`.

```
with open(faculty_path, 'r') as file_obj: # open in read mode
    data = file_obj.readlines() # returns list; elements include trailing
'\n'
```

❗ You are limited to calling a file object's `read()` method or `readlines()` method *once* after opening a connection to a file.

```
with open(faculty_path, 'r') as file_obj: # open in read mode
    data = file_obj.read()
    data_lines = file_obj.readlines() # WARN: does not execute
```

Given that opening a file is a common operation let's define a function to perform the task.

Given a valid `filepath`, the function `read_file` below opens a file, returns a file object, and retrieves the content as a list before returning it to the caller.

The function also defines an optional parameter named `strip` that specifies whether or not individual lines in the file object are returned "as is" or are stripped of leading/trailing whitespace as well as the newline escape sequence `\n` removed.

```
read_file(filepath, strip=True):
    """Read text file line by line. Remove whitespace and trailing newline
    escape character.

    Parameters:
        filepath (str): path to file
        strip (bool): remove white space, newline escape characters

    Returns
        list: list of lines in file
    """

    with open(filepath, 'r', encoding='utf-8') as file_obj:
        if strip:
            data = []
            for line in file_obj:
                # data.append(line) # includes trailing newline '\n'
                data.append(line.strip()) # strip leading/trailing
whitespace including '\n'
            return data
        else:
            return file_obj.readlines() # list
```

💡 the built-in `open()` function's optional `encoding` parameter specifies the encoding used to *decode* (when opening the file) or *encode* (when writing content to a file). UTF-8 is a variable width character encoding that builds on the older ASCII character coding standard. It uses one to four one-byte (8-bit) code units to represent characters. UTF-8 is the most common encoding used on the Web.

2.5 Write methods: `write()`, `writelines()`

To write data to a file call the built-in `open()` function in "write" (`w`) mode. The file object that is returned includes both a `write()` method and a `writelines()` method. Call the `write()` method when working with a string.

```
with open(faculty_path, 'r') as file_obj: # open in read mode
    data = file_obj.read() # returns a single multiline string

filepath = './umsi-faculty-v3.txt'
# filepath = os.path.join(abs_path, 'umsi-faculty-v3.txt')
with open(filepath, 'w') as file_obj: # open in write mode
    file_obj.write(data)
```

Call the `writelines()` method when working with a sequence. In the example below the data is retrieved from the file, then each name is reversed (last name, first name -> first name last name), and the results written to a text file.

```
with open(faculty_path, 'r') as file_obj: # open in read mode
    data = file_obj.readlines() # returns a list

# Reverse names
for i in range(len(data)):
    name = data[i].strip().split(', ') # strip \n
    data[i] = f"{name[1]} {name[0]}\n" # restor \n

# WARN: does not update string
# for faculty_member in data:
#     name = faculty_member.strip().split(', ') # strip \n
#     faculty_member = f"{name[1]} {name[0]}\n" # does not update string
element

filepath = './umsi-faculty-v4.txt'
# filepath = os.path.join(abs_path, 'umsi-faculty-v4.txt')
with open(filepath, 'w') as file_obj: # open in write mode
    file_obj.writelines(data)
```

If you require fine-grain control over what is written to file when working with a sequence, you can call the built-in `open()` function in "write" (`w`) mode and then loop over the list and pass each (modified) element to the file object's `write()` method.

In the example below the data is retrieved from the file, each faculty member's name string is split, and each surname together with a trailing newline escape character (`\n`) is written to a text file.

```
with open(faculty_path, 'r') as file_obj: # open in read mode
    data = file_obj.readlines() # returns a list

filepath = './umsi-faculty-v5.txt'
# filepath = os.path.join(abs_path, 'umsi-faculty-v5.txt')
with open(filepath, 'w') as file_obj: # open in write mode
    for row in data:
        # Surname only
        # file_obj.write(row.split(', ')[0]) # WARN: lose trailing `n`
        file_obj.write(f"{row.split(', ')[0]}\n") # add `n`
```

❗ Note that the `write()` method does not add a newline escape sequence to the string passed to it.

Given that writing content to a file is a common task that could occur multiple times in a program, we should migrate the write operation to a function.

Given a valid `filepath`, the `write_file` function below writes the passed in `data` to the target file.

The function includes an optional parameter named `newline` to control whether or not a newline escape sequence (`\n`) should be appended to each string element in the passed in `data` list.

```
def write_file(filepath, data, newline=True):
    """Write content to a target file encoded as UTF-8. If optional
newline is specified
    append each line with a newline escape sequence (`\n`).

    Parameters:
        filepath (str): path to target file (if file does not exist it
will be created)
        data (list): list of strings comprising the content to be written
to the target file
        newline (bool): add newline escape sequence to line

    Returns:
        None
    """

    with open(filepath, 'w', encoding='utf-8') as file_obj:
        if newline:
            for line in data:
                file_obj.write(f"{line}\n") # add newline
        else:
            file_obj.writelines(data) # write sequence to file
```

We can then refactor our code to utilize the `read_file` and `write_file` functions:

```
# Get data
data = read_file(filepath)

# Access surnames first before calling function
surnames = []
for row in data:
    surnames.append(row.split(', ')[0]) # trailing \n not required

# Write surnames to file
filepath = './umsi-faculty-v6.txt'
write_file(filepath, surnames)
```

3.0 Working with CSV files

A comma-separated values (CSV) file is a common data interchange format used to represent tabular data. It is a delimited text file that utilizes a comma (`,`) typically to separate individual values. Other delimiters include pipes (`|`) or tabs, though use of the latter is usually referred to as a tab-delimited values (TSV) file.

Keep in mind the following when working with or creating CSV files:

1. If a value in a CSV file includes a delimiter (e.g., a comma), the value is usually surrounded by double quotation marks (`""`).
2. The first row in a CSV file is often a designated "header" row that contains a list of the column names (or headers) that describe the following data. This is recommended practice that helps to make the CSV file self-documenting. But you need to exclude the row when working with the actual data.
3. Occasionally the first character in a CSV file is a byte order mark (BOM). You can filter it out by changing the built-in `open()` function's optional encoding value to `utf-8-sig`.

   ```python
   with open("path_to_a_csv_file.csv", encoding="utf-8-sig") as fileobj:
       # Retrieve data
   ```

4. You can save Excel spreadsheets and export Google sheets as CSV files.
5. The VS Code marketplace features an extension called Rainbow CSV that you can install in order to make viewing a CSV file a more pleasant experience.

3.1 The `csv` module: `csv.reader()`

The Python Standard library includes a `csv` module that simplifies working with CSV files. In order to use the `csv` module you must *import* it into your program. This is done by adding an `import` statement to your code located at the top of your file.

```python
import csv
```

Once imported the `csv` module and its objects and object methods are referenced using dot notation:

```python
reader = csv.reader(fileobj, delimiter=delimiter)
```

To read the contents of a `*.csv` file we can call the `csv.reader()` function, conveniently located within the function `read_csv()`. The function itself accepts a file path to the CSV file as well as an optional delimiter if the CSV file is delimited with a character other than a comma such as a pipe (`|`).

The function uses a `with` statement and the built-in `open()` function to open the file and return a file object. A `reader` object is then created by calling the `csv.reader()` function and passing to it the `file_obj` and the `delimiter` as arguments. The reader object is an *iterable* (e.g., has members that can be accessed) and we can loop over it in order to access each "row" element. Doing so allows us to append each `row` in the `reader` to the `data` list. Once the `for` loop finishes its work, the `data` list is returned to the caller.

```
def read_csv(filepath, delimiter=','):
    """
    Reads a CSV file, parsing row values per the provided delimiter.
Returns a list
    of lists, wherein each nested list represents a single row from the
input file.

    Parameters:
        filepath (str): The location of the file to read.
        delimiter (str): delimiter that separates the row values

    Returns:
        list: contains nested "row" lists
    """

    with open(filepath, 'r', newline='', encoding='utf-8') as file_obj:
        data = []
        reader = csv.reader(file_obj, delimiter=delimiter)
        for row in reader:
            data.append(row)

        return data
```

3.2 The `csv` module: `csv.writer()`

We can use the `csv.writer()` function to write data to a target CSV file. The function `write_csv()` accepts a file path, the data to be written to the file, and an optional headers list of column name strings as arguments.

Similar to `read_csv()` the function uses a `with` statement and the built-in `open()` function to open the file and return a file object. A `writer` object is then created by calling the `csv.writer()` function and passing to it the `file_obj` as an argument.

If headers are passed in, a header row is written by calling the `writer.writerow()` method and passing to it the `headers` list. Then each row in `data` is written to the file. If no headers are provided the `data` list is passed directly to the `writer.writerows()` method (which accepts a sequence as an argument) and the data is written to the new file by a batch process.

```
def write_file(filepath, data, newline=True):
    """Write content to a target file encoded as UTF-8. If optional
newline is specified
    append each line with a newline escape sequence (`\n`).

    Parameters:
        filepath (str): path to target file (if file does not exist it
will be created)
        data (list): list of strings comprising the content to be written
to the target file
        newline (bool): add newline escape sequence to line

    Returns:
        None
    """

    with open(filepath, 'w', encoding='utf-8') as file_obj:
        if newline:
            for line in data:
                file_obj.write(f"{line}\n") # add newline
        else:
            file_obj.writelines(data) # write sequence to file
```

❗ `write_data` requires that data passed to it is either a `list` or a `tuple`. If working with strings, each string must be placed in a `list` otherwise the `csv_writer` will parse the string as a sequence, treating each character as a seperate data element and separating each with a comma when writtent to the CSV file.

3.3 Example: read/write CSV files

With the csv read/write functions implemented we can read "source" CSV files, manipulate and/or analyze the data returned, and write the results of our work to one or more "target" CSV files.

If, for example, we wanted to return a list of publications coauthored by Professor Resnick on recommender systems—Resnick is considered a pioneer in the development of such systems—and then write the list to a file we could do the following:

```
# 1.0 Read CSV file and retrieve data
filepath = './input/resnick-publications.csv'
publications = read_csv(filepath)

# 2.0 Get headers
headers = publications[0] # header row

# 3.0 Filter title on "recommender"; accumulate results
recommender_publications = []
for publication in publications[1:]:
    if 'recommender' in publication[headers.index('title')].lower():
        recommender_publications.append(publication)

# 4.0 Write CSV file
filepath = './output/resnick-recommender_publications.csv'
write_csv(filepath, recommender_publications, headers)
```

4.0 Nested loops

What if we needed to identify all publications that Professor Resnick coauthored with another member of the UMSI faculty? The "authors" data in `resnick-publications.csv` is a string that we will need to split as the following example illustrates:

"Moser, Carol; Schoenebeck, Sarita Yardi; Resnick, Paul"

We can look up UMSI faculty members after reading `umsi-faculty.txt` in an attempt to obtain matches on each publication's list of coauthors but we will need to implement a *nested loop* in order to accomplish the task.

A nested loop refers to a loop located within the code block of another loop. During each iteration of the "outer" loop, the "inner" loop will execute, completing all its iterations (unless terminated early) *prior* to the outer loop commencing its next iteration, if any.

```
for < element > in < sequence >:
    # indented block
    for < element > in < element (itself a sequence) >
        < statement A >
        < statement B >
        ...
```

❗ Be sure to ignore Paul Resnick when matching coauthors against the UMSI faculty list; otherwise you will end up returning all publications.

The following example illustrates the pattern.

```
faculty = read_file(faculty_path)

# Normalize the strings (all lower case)
for i in range(len(faculty)):
    faculty[i] = faculty[i].lower() # mutates list element string value

# for member in faculty:
#     member = member.lower() # WARN: this does not mutate the underlying
string

print(f"4.0: UMSI faculty (lower case) sample\n{faculty[:3]}")

# Find UMSI coathured publications
umsi_coauthor_publications = []
for publication in publications[1:]:
    authors = publication[headers.index('authors')].split('; ')
    authors.remove('Resnick, Paul') # ignore Paul
    for author in authors:
        if author.lower() in faculty:
            umsi_coauthor_publications.append(publication)
            break # one match is all we need

# Write CSV file
```

```
filepath = './output/umsi_coauthor_publications.csv'
write_csv(filepath, umsi_coauthor_publications, headers)
```

If instead you choose as your outer loop the list of UMSI faculty rather than the list of publications you will need to implement two nested loops as well as guard against appending duplicate publications to `umsi_coauthor_publications` in cases where a publication features multiple UMSI coauthors. The solution is less efficient than the previous solution.

```
# Loop over faculty, publications, authors (less efficient)
umsi_coauthor_publications.clear()
for member in faculty:
    if member == 'resnick, paul':
        continue
    for publication in publications[1:]:
        authors = publication[headers.index('authors')].split('; ')
        for author in authors:
            if author.lower() == member and publication not in
umsi_coauthor_publications:
                umsi_coauthor_publications.append(publication)
                break # one match is all we need

# Write CSV file (order of publications determined by faculty loop)
filepath = './umsi_coauthor_publications-v2.csv'
write_csv(filepath, umsi_coauthor_publications, headers)
```

💡 Note use of a compound `if` statement featuring the logical operator `and`. Both conditions *must* evaluate to `True` for a publication to be appended to the accumulator list. The second condition (`publication not in umsi_coauthor_publications`) prevents duplicate publications from being appended to the accumulator list.

```
if author.lower() == member and publication not in
umsi_coauthor_publications:
    umsi_coauthor_publications.append(publication)
```