# SI 506: Lecture 12

## TOPICS

1. The `main()` function
2. Docstrings
3. Opening a file (ye olde way / recommended way)
4. Working with CSV files

## Vocabulary

- **Docstring**. String literal that appears as the first statement in a function, class, or module. The docstring provides a terse description of an object's purpose, attributes, and behavior. The docstring is assigned to an object's **doc** attribute and is available via introspection.
- **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.
- **Flow of execution**. The order in which statements in a program are executed. Also referred to as *control flow*.
- **UTF-8**. UTF-8 is a variable-width character encoding that uses one to four one-byte (8-bit) code units to represent individual characters. The encoding encompases the older US-ASCII character set as well nearly all Latin-script alphabets as well as IPA extensions, Greek, Cyrillic, Coptic, Armenian, Hebrew, Arabic, Syriac, Thaana, N'Ko alphabets and most Chinese, Japanese, and Korean characters. UTF-8 is the dominant encoding used on the Web.

## Data

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

## 1.0 The main() function: controlling the flow of execution

Python features *two* file execution modes. Code in a file can be executed as a script from the command line or the code can be *imported* into another Python file in order to access its definitions and statements.

If a Python file is executed from the command line the Python interpreter will run the file under the special name of `__main__` rather than the program's actual file name (e.g., `lecture_XX.py`). We refer to such a file as a *script* or a program.

Given this naming behavior we can choose the program's entry point and control the program's execution flow by directing the Python interpreter to call the `main()` function *first* in order to execute the statements defined in its code block.

## 1.1 Why `main()`?

Employing a `main()` function to manage your program's flow of execution separates the code you write to manage a program's work flow from the code you write to perform specific tasks (e.g., functions). This encourages code modularization and, by relying on function calls to perform specific tasks, helps to eliminate code duplication.

An important side benefit is that with the work flow code restricted to `main()` the other definitions and statements comprising the file (e.g., functions, classes, constants) can be imported as a module into another Python module without triggering the code located in `main()`. This can occur because module code imported from one Python file into another Python file is known by the Python interpreter by the module's actual file name and not by the name `__main__` as is the case with scripts run from the command line.

💡 We will cover modules, module imports, and execution modes in more detail *after* the midterm.

## 2.0 Docstrings

The Python documentation string or docstring is a string literal that is positioned as the first statement in a function. The docstring provides a short summary of the function's expected behavior, including details regarding defined parameters (required and optional) and return value, if any. The Python interpreter assigns the string to the special "dunder" `__doc__` object attribute and is available via introspection. Docstrings can also be assigned to modules, classes, and class methods.

There are two forms of docstrings: single line and multi-line statements. Single line Docstrings are reserved for describing obvious behaviors. For example, the built-in `len()` function is described with a single line docstring:

```
>>> len.__doc__
'Return the number of items in a container.'
```

The docstrings in functions and other objects that you will encounter in this course resemble a specially formatted multiline strings bounded by triple quotation marks (`"""`).

The docstring format we use is as follows:

```
"""Short description outlining the purpose and expected behavior of the
function. Between one
and five sentences should suffice to describe the function in all its
glory.

Parameters:
    < name > (< type >): Terse description of the parameter.
    . . . [Repeat for each parameter, required and optional]

Returns:
    < type >: Terse description of the return value. If no value is
explicitly returned use `None`.
"""
```

The teaching team will make increasing use of docstrings in both lectures, labs, problem sets and lab exercises, the midterm, and the last assignment in order to describe a function's purpose, define parameters and indicate an explicit return value, if any.

## 3.0 Opening files

## 3.1 Opening a file (ye olde way)

You can use the built-in `open()` function to access a file and return a file object (also known as 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.

```
filepath = './umsi-faculty.txt' # relative path
# filepath = 'umsi-faculty.txt' # alternative

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

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

## 3.2 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` statement provides a context manager that closes the file automatically once all statements within the `with` code block are executed, even if errors occur.

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

## 3.3 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 | Mode | 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 |
| '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 |

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

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

## 3.4 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

```python
with open(filepath, '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(filepath, '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(filepath, '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 defines three parameters:

- `filepath` (`str`): path to the file
- `encoding` (`str`): specifies the encoding used to *decode* (i.e., read) the file. Default = `utf-8`.
- `strip` (`bool`): 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. Default = `True`.

```
def read_file(filepath, encoding='utf-8', strip=True):
    """Read text file line by line. Remove whitespace and trailing newline
    escape character.

    Parameters:
        filepath (str): path to file
        encoding (str): name of encoding used to decode the file.
        strip (bool): remove white space, newline escape characters

    Returns
        list: list of lines in file
    """
    with open(filepath, 'r', encoding=encoding) 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

            # return [line.strip() for line in file_obj] # list
comprehension (single line)
        else:
            return file_obj.readlines() # list
```

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

with open('./umsi-faculty-v3.txt', '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(filepath, '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" # restore \n

# filepath = os.path.join(abs_path, 'umsi-faculty-v4.txt')
with open('./umsi-faculty-v4.txt', '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(filepath, 'r') as file_obj: # open in read mode
    data = file_obj.readlines() # returns a list

with open('./umsi-faculty-v5.txt', '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 defines four parameters:

- `filepath` (`str`): path to target file (if file does not exist it will be created)
- `data` (`list` | `tuple`): sequence to be written to the target file
- `encoding` (`str`): encoding used to *encode* (i.e., write) the file. Default = `utf-8`.
- `newline` (`bool`): specifies whether the newline escape sequence (`\n`) should be appended to each string element in the passed in `data` list. Default = `True`.

```
def write_file(filepath, data, encoding='utf-8', 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 | tuple): sequence of strings comprising the content to
be written to the
                             target file
        encoding (str): name of encoding used to encode the file.
        newline (bool): add newline escape sequence to line

    Returns:
        None
    """
    with open(filepath, 'w', encoding=encoding) 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
write_file('./umsi-faculty-v6.txt', surnames)
```

## 4.0 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`.

   ```
   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.

## 4.1 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.

```
import csv
```

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

```
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 user-defined function `read_csv()`. The function defines four parameters:

- `filepath` (`str`): location of the CSV file to be read.
- `encoding` (`str`): encoding used to *decode* (i.e., read) the file. Default = `utf-8`.
- `newline` (`str`): replacement value for newline `'\n'` or `'\r\n'` (Windows) character sequences. Default = `''` (blank).

- delimiter (str): delimiter that separates each row value. Default = ','.

The function employs the 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 you can loop over it in order to access each "row" element. Doing so allows each row in the reader to be appended to the data list. Once the for loop finishes its work, the data list is returned to the caller.

```
def read_csv(filepath, encoding='utf-8', newline='', 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.

    WARN: If a byte order mark (BOM) is encountered at the beginning of
the first line of decoded
    text, call < read_csv > and pass 'utf-8-sig' as the < encoding >
argument.

    WARN: If newline='' is not specified, newlines '\n' or '\r\n' embedded
inside quoted fields
    may not be interpreted correctly by the csv.reader.

    Parameters:
        filepath (str): The location of the file to read
        encoding (str): name of encoding used to decode the file
        newline (str): specifies replacement value for newline '\n'
                       or '\r\n' (Windows) character sequences
        delimiter (str): delimiter that separates the row values

    Returns:
        list: nested "row" lists
    """
    with open(filepath, 'r', encoding=encoding, newline=newline) as
file_obj:
        data = []
        reader = csv.reader(file_obj, delimiter=delimiter)
        for row in reader:
            data.append(row)

        return data
```

## 4.2 The csv module: csv.writer()

To read the contents of a *.csv file we can call the csv.writer() function, conveniently located within the user-defined function write_csv(). The function defines five parameters:

- filepath (str): path to target file (if file does not exist it will be created)
- data (list | tuple): sequence to be written to the target file

- `headers` (`list` | `tuple`): optional header row list or tuple
- `encoding` (`str`): encoding used to *encode* (i.e., write) the file. Default = `utf-8`.
- `newline` (`str`): replacement value for newline `'\n'` or `'\r\n'` (Windows) character sequences. Default = `''` (blank).

Similar to `read_csv()` the function employs 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.

❗ The `write_data` function 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.

```python
def write_csv(filepath, data, headers=None, encoding='utf-8', newline=''):
    """
    Writes data to a target CSV file. Column headers are written as the
first
    row of the CSV file if optional headers are specified.

    WARN: If newline='' is not specified, newlines '\n' or '\r\n' embedded
inside quoted
    fields may not be interpreted correctly by the csv.reader. On
platforms that utilize
    `\r\n` an extra `\r` will be added.

    Parameters:
        filepath (str): path to target file (if file does not exist it
will be created)
        data (list | tuple): sequence to be written to the target file
        headers (seq): optional header row list or tuple
        encoding (str): name of encoding used to encode the file
        newline (str): specifies replacement value for newline '\n'
                       or '\r\n' (Windows) character sequences

    Returns:
        None
    """
    with open(filepath, 'w', encoding=encoding, newline=newline) as
file_obj:
        writer = csv.writer(file_obj)
        if headers:
            writer.writerow(headers)
            for row in data:
                writer.writerow(row)
        else:
            writer.writerows(data)
```

## 4.3 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:

```
filepath = './resnick-publications.csv'
publications = read_csv(filepath)

print(f"\n3.0: Total publications (rows) = {len(publications)}")

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

print(f"\n3.0: Total elements (columns) = {len(headers)}")

# 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)

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