Programming Principles in Python (CSCI 503)

Files, Scripts, and Modules

Dr. David Koop

Regular Expressions

- AKA regex
- A syntax to better specify how to decompose strings
- Look for patterns rather than specific characters
- Metacharacters: `. ^ $ * + ? { } [ ] \ | ( )`
  - Repeat, one-of-these, optional
- Character Classes: `\d` (digit), `\s` (space), `\w` (word character), also `\D`, `\S`, `\W`
- Digits with slashes between them: `\d+/\d+/\d+`
- Usually use raw strings (no backslash plague): `r'\d+/\d+/\d+'`

Regular Expression Examples

-
  ```
  s0 = "No full dates here, just 02/15"
  s1 = "02/14/2021 is a date"
  s2 = "Another date is 12/25/2020"
  s3 = "April Fools' Day is 4/1/2021 & May the Fourth is 5/4/2021"
  ```
- `re.match(r'\d+/\d+/\d+',s1) # returns match object`
- `re.match(r'\d+/\d+/\d+',s2) # None!`
- `re.search(r'\d+/\d+/\d+',s2) # returns 1 match object`
- `re.search(r'\d+/\d+/\d+',s3) # returns 1! match object`
- `re.findall(r'\d+/\d+/\d+',s3) # returns list of strings`
- `re.finditer(r'\d+/\d+/\d+',s3) # returns iterable of matches`

Substitution

- Do substitution in the middle of a string:
- `re.sub(r'(\d+)/(\d+)/(\d+)',r'\3-\1-\2',s3)`
- All matches are substituted
- First argument is the regular expression to **match**
- Second argument is the **substitution**
  - \1, \2, … match up to the **captured groups** in the first argument
- Third argument is the **string** to perform substitution on
- Can also use a **function**:
-
```
to_date = lambda m:
f'{m.group(3)}-{int(m.group(1)):02d}-{int(m.group(2)):02d}'
re.sub(r'(\d+)/(\d+)/(\d+)', to_date, s3)
```

Reading Files

- Use the `open()` method to open a file for reading
  - `f = open('huck-finn.txt')`
- Usually, add an `'r'` as the second parameter to indicate read (default)
- Can iterate through the file (think of the file as a collection of lines):

  -
    ```
    f = open('huck-finn.txt', 'r')
    for line in f:
        if 'Huckleberry' in line:
            print(line.strip())
    ```
- Using `line.strip()` because the read includes the newline, and print writes a newline so we would have double-spaced text
- Closing the file: `f.close()`

Remember Encodings (Unicode, ASCII)?

- Encoding: How things are actually stored
- ASCII "Extensions": how to represent characters for different languages
  - No universal extension for 256 characters (one byte), so…
  - ISO-8859-1, ISO-8859-2, CP-1252, etc.
- Unicode encoding:
  - UTF-8: used in Python and elsewhere (uses variable # of 1—4 bytes)
  - Also UTF-16 (2 or 4 bytes) and UTF-32 (4 bytes for everything)
  - Byte Order Mark (BOM) for files to indicate endianness (which byte first)

Encoding in Files

-
  ```
  all_lines = open('huck-finn.txt').readlines()
  all_lines[0] # '\ufeff\n'
  ```
- `\ufeff` is the UTF Byte-Order-Mark (BOM)
- Optional for UTF-8, but if added, need to read it

-
  ```
  a = open('huck-finn.txt', encoding='utf-8-sig').readlines()
  a[0] # '\n'
  ```
- No need to specify UTF-8 (or ASCII since it is a subset)
- Other possible encodings:
  - cp1252, utf-16, iso-8859-1

Assignment 4

- Assignment covers strings and files
- Reading & writing data to files
- Deals with characters and formatting

Parsing Files

- Dealing with different formats, determining more meaningful data from files
- txt: text file
- csv: comma-separated values
- json: JavaScript object notation
- Jupyter also has viewers for these formats
- Look to use libraries to help possible
  - `import json`
  - `import csv`
  - `import pandas`
- Python also has pickle, but not used much anymore

Comma-separated values (CSV) Format

- Comma is a field separator, newlines denote records

  -
    ```
    a,b,c,d,message
    1,2,3,4,hello
    5,6,7,8,world
    9,10,11,12,foo
    ```
- May have a header (`a,b,c,d,message`), but not required
- No type information: we do not know what the columns are (numbers, strings, floating point, etc.)
  - Default: just keep everything as a string
  - Type inference: Figure out the type to make each column based on values
- What about commas in a value? → double quotes

JavaScript Object Notation (JSON)

- A format for web data
- Looks very similar to python dictionaries and lists
- Example:

  -
    ```
    {"name": "Wes",
     "places_lived": ["United States", "Spain", "Germany"],
     "pet": null,
     "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
                  {"name": "Katie", "age": 33, "pet": "Cisco"}] }
    ```
- Only contains literals (no variables) but allows null
- Values: strings, arrays, dictionaries, numbers, booleans, or null
  - Dictionary keys must be strings
  - Quotation marks help differentiate string or numeric values

Python csv module

- Help reading csv files using the csv module

  -
    ```
    import csv
    with open('persons_of_concern.csv', 'r') as f:
        for i in range(3): # skip first three lines
            next(f)
        reader = csv.reader(f)
        records = [r for r in reader] # r is a list
    ```
- or

  -
    ```
    import csv
    with open('persons_of_concern.csv', 'r') as f:
        for i in range(3): # skip first three lines
            next(f)
        reader = csv.DictReader(f)
        records = [r for r in reader] # r is a dict
    ```

Writing Files

- `outf = open("mydata.txt", "w")`
- If you open an existing file for writing, you wipe out the file's contents. If the named file does not exist, a new one is created.
- Methods for writing to a file:
  - `print(<expressions>, file= outf)`
  - `outf.write(<string>)`
  - `outf.writelines(<list of strings>)`
- If you use write, no newlines are added automatically
  - Also, remember we can change print's ending: `print(…, end=", ")`
- Make sure you close the file! Otherwise, content may be lost (buffering)
- `outf.close()`

With Statement: Improved File Handling

- With statement does "enter" and "exit" handling:
- In the previous example, we need to remember to call `outf.close()`
- Using a with statement, this is done automatically:

  -
    ```
    with open('huck-finn.txt', 'r') as f:
        for line in f:
            if 'Huckleberry' in line:
                print(line.strip())
    ```
- This is important for **writing** files!

  -
    ```
    with open('output.txt', 'w') as f:
        for k, v in counts.items():
            f.write(k + ': ' + v + '\n')
    ```
- Without `with`, we need `f.close()`

Context Manager

- The with statement is used with contexts
- A context manager's **enter** method is called at the beginning
- …and **exit** method at the end, even if there is an exception!

-
  ```
  outf = open('huck-finn-lines.txt','w')
  for i, line in enumerate(huckleberry):
      outf.write(line)
      if i > 3:
          raise Exception("Failure")
  ```

-
  ```
  with open('huck-finn-lines.txt','w') as outf:
      for i, line in enumerate(huckleberry):
          outf.write(line)
          if i > 3:
              raise Exception("Failure")
  ```

Context Manager

- The with statement is used with contexts
- A context manager's **enter** method is called at the beginning
- …and **exit** method at the end, even if there is an exception!
- ~~`outf = open('huck-finn-lines.txt','w')`~~

```
~~for i, line in enumerate(huckleberry):
      outf.write(line)
      if i > 3:
          raise Exception("Failure")~~
```

- `with open('huck-finn-lines.txt','w') as outf:`

```
      for i, line in enumerate(huckleberry):
          outf.write(line)
          if i > 3:
              raise Exception("Failure")
```

Reading & Writing JSON data

- Python has a built-in `json` module

  -
    ```
    with open('example.json') as f:
        data = json.load(f)
    ```

  -
    ```
    with open('example-out.json', 'w') as f:
        json.dump(data, f)
    ```
- Can also load/dump to strings:
  - `json.loads`, `json.dumps`

Reading Binary Data

- Add a `'b'` to the open call to specify binary mode
  - `f = open('data.bin', 'rb')`
- You can read the data using the methods we have seen before, but…
  - …data comes back as a **byte string** (prefixed with `b`)
  - …you cannot read (all of) it nicely because it is not text-encoded!
  - `b'@\t!\xf9\xf0\x1b\x86n\x00\x00\x03HHello \xf0\x9f\x99\x82'`
  - Can create your own byte strings: `b'\x80\x00\x00\x00'`
- Need to know what this data is in order to read it!
  - Need a **byte-by-byte** description
  - 8 bytes are double (float), next 4 are an int (32-bit), last 10 are char* (string)

The struct module

- The `struct` module allows us to pack and unpack binary data
  - `pack`: python data to byte string
  - `unpack`: byte string to python data
- Need to specify format using format strings (different from string format!)
  - These are generally C types (not python types!)
  - Examples: `char` (`c`), `int` (`i`), `double` (`d`), `char*` (`s`)
  - Can specify any sequence of these characters but should match the data
  - For strings (`char*`), prefix the number of bytes
- Example: `struct.unpack('di10s')`

Endianness

- The order that multi-byte data is stored in memory
- **Big-endian**: most significant byte is first (left to right)
- **Little-endian**: most significant byte is last (right to left)
- Format string prefix: use < for little-endian and > for big-endian
- Defaults to **native** (whichever your machine uses, most are little-endian)
- Changes values a lot:
  - `struct.unpack('>i', b'\x01\x00\x00\x00')[0] # 16777216`
  - `struct.unpack('<i', b'\x01\x00\x00\x00')[0] # 1`
- Example: `struct.unpack('>di10s')`
  - `(3.14159, 840, b'Hello \xf0\x9f\x99\x82')`

String Decoding

- Strings are read as byte strings, not Unicode strings!
  - `b'Hello \xf0\x9f\x99\x82'`
- Need to decode them based on the character encoding
- Default encoding is utf-8, but can specify others
  - `b'Hello \xf0\x9f\x99\x82'.decode('ascii') # error`
  - `b'Hello \xf0\x9f\x99\x82'.decode() # 'Hello 🙂'`

Writing Binary Data

- First, `struct.pack`, basically the reverse of unpack
  - `d_out = struct.pack(f'di{len(vs)}s', vf, vi, vs)`
  - Uses default endianness (little-endian)
- Need to encode strings: `vs.encode()`
  - `len(vs) # 7`
  - `len(vs.encode()) # 10 # length changes`
- Problems:
  - Data is too big for the specified format (e.g. int)
  - String length (when to stop reading?)

Variable-length Data

- Python int has methods for converting to/from bytes
  - To write, need number of bytes, can calculate minimum number
    - `num_bytes = (big_int.bit_length + 7) // 8`
    - `big_int_bytes = big_int.to_bytes(num_bytes)`
  - Inverse: `int.from_bytes(f.read(num_bytes))`
- Problem: **How many** bytes to read?
- Solution: Write that number with the data
  - `f.write(struct.pack('i', num_bytes))`
  - `f.write(big_int_bytes)`
  - When reading, first read integer number of bytes and read that many bytes

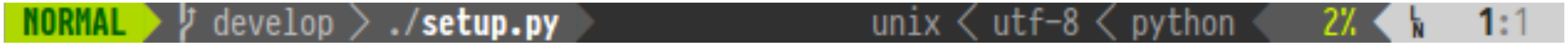
Variable-length Data

- Strings have the same problem
- Can use the same approach as integers
- Also, could read byte-by-byte until we reach **null terminator** (`'\0'`)
- Easy to write: `f.write(bstr + b'\0')`
- More involved to read:

  -
    ```
    bstr = b''
    while (b := f.read(1)) != b'\0':
        bstr += b
    bstr
    ```
- Remember to encode/decode!

Command-Line Interfaces

Command Line Interfaces (CLIs)

- Prompt:
  - `$`
  - 
    NORMAL develop ./setup.py unix utf-8 python 2% 1:1
- Commands
  - `$ cat <filename>`
  - `$ git init`
- Arguments/Flags: (options)
  - `$ python -h`
  - `$ head -n 5 <filename>`
  - `$ git branch fix-parsing-bug`

Command Line Interfaces

- Many command-line tools work with stdin and stdout
  - `cat test.txt # writes test.txt's contents to stdout`
  - `cat # reads from stdin and writes back to stdout`
  - `cat > test.txt # writes user's text to test.txt`
- Redirecting input and output:
  - `<` use input from a file descriptor for stdin
  - `>` writes output on stdout to another file descriptor
  - `|` connects stdout of one command to stdin of another command
  - `cat < test.txt | cat > test-out.txt`

Python and CLIs

- Python can be used as a CLI program
  - Interactive mode: start the REPL
    - `$ python`
  - Non-interactive mode:
    - `$ python -c <command>`: Execute a command
    - `$ python -m <module>|<package>`: Execute a module
- Python can be used to create CLI programs
  - Scripts: `python my_script.py`
  - True command-line tools: `./command-written-in-python`

Interactive Python in the Shell

- Starting Python from the shell
  - `$ python`
- `>>>` is the Python interactive prompt

  -
    ```
    >>> print("Hello, world")
    Hello, world
    ```

  -
    ```
    >>> print("2+3=", 2+3)
    2+3= 5
    ```
- This is a REPL (Read, Evaluate, Print, Loop)

Interactive Python in the Shell

- `...` is the continuation prompt

-
  ```
  >>> for i in range(5):
  ...     print(i)
  ...
  ```
- Still need to indent appropriately!
- Empty line indicates the suite (block) is finished
- This isn't always the easiest environment to edit in

Ending an Interactive Session

- `Ctrl-D` ends the input stream
  - Just as in other Unix programs
- Another way to get normal termination
  - `>>> quit()`
- `Ctrl-C` interrupts operation
  - Just as in other Unix programs

Interactive Problems

- But standard interactive Python doesn't save programs!
- IPython does have some magic commands to help
  - `%history`: prints code
  - `%save`: saves a file with code
  - These are most useful outside the notebook, but you can type them in the notebook, too
- However, it is nice to be able to edit code in files and run it, too

Module Files

- A **module file** is a text file with the `.py` extension, usually name.py
- Python source on Unix is expected to be in UTF-8
- Can use any text editor to write or edit…
- …but an editor that understands Python's spacing and indentation helps!
- Contents looks basically the same as what you would write in the cell(s) of a notebook
- There are also ways to write code in multiple files organized as a package, will cover this later

Scripts, Programs, and Libraries

- Often, interpreted ~ scripts and compiled code ~ programs/libraries
  - Python does compile **bytecode** for modules that are imported
- Modifying this usual definition a bit
  - Script: a one-off block of code meant to be run by itself, users **edit the code** if they wish to make changes
  - Program: code meant to be used in different situations, with **parameters** and **flags** to allow users to customize execution without editing the code
  - Library: code meant to be called from other scripts/programs
- In Python, can't always tell from the name what's expected, code can be both a library and a program