Lecture 03

- Command line
    - Links
    - Environment variables & `.env`
    - Shell scripts
    - `cron`
    - Compressing/decompressing with `tar`, `zip/unzip`
- Python
    - Data structures (lists, tuples, dictionaries, sets)

Python references

- O'Reilly catalog via UCSF library
    - Introducing Python
    - Python Crash Course
- Think Python
- Automate the Boring Stuff with Python
- Introduction to Python
- A Byte of Python

Requested information

- WSL: link to home in Windows - later in lecture
- PyCharm tutorials
  - Jetbrains' Learn PyCharm
  - Effective PyCharm Course ($$$)
  - Various YouTube videos

Review

- Paths, filename
- `pwd`, `ls`, `cd`
- `mv`, `cp`, `touch`
- `cat`, `head`, `tail`
- `grep`
- `|`, `>`

Review: Paths and Filenames

- **Paths**: Location of files/directories in the file system
  - Absolute path: Full path from root directory
    - `/Users/christopher/UCSF/DATASCI-217/03`
  - Relative path: Path relative to current location
    - `./UCSF/DATASCI-217/03`
    - **Note:** leading `./` represents the current working directory
- **Filename**: Name of a file, including its extension
  - Example: `script.py`, `data.csv`, `document.txt`

Review: `pwd`, `ls`, `cd`

- `pwd` (Print Working Directory) shows current directory path
  - Example: `$ pwd`
    Output: `/home/user/documents`
- `ls` (List) lists contents of a directory
  - Common options: `-l` (long format), `-a` (show hidden files)
  - Example: `$ ls -la`
- `cd` (Change Directory) navigates between directories
  - Example: `$ cd /home/user/downloads`

Review: `mv`, `cp`, `touch`

- `mv` (Move) - Moves or renames files/directories
  - Syntax: `mv [source] [destination]`
  - Example: `$ mv old_name.txt new_name.txt`
- `cp` (Copy) - Copies files/directories
  - Syntax: `cp [source] [destination]`
  - Example: `$ cp file.txt backup/file_copy.txt`
- `touch` - Creates empty files or updates timestamps
  - Example: `$ touch new_file.txt`

Review: `cat`, `head`, `tail`

- `cat` (Concatenate) - Displays content of file(s)
  - Example: `$ cat file.txt`
- `head` - Shows first few lines of a file (default: 10)
  - Example: `$ head -n 5 file.txt`
- `tail` - Shows last few lines of a file (default: 10)
  - Example: `$ tail -n 20 log_file.txt`

Review: `grep`

- `grep` (Global Regular Expression Print)
  - Searches for patterns in files
  - Syntax: `grep [options] pattern [file...]`
  - Example: `$ grep "error" log_file.txt`
  - Common options:
    - `-i` : Case-insensitive search
    - `-r` : Recursive search in directories
    - `-n` : Show line numbers

Review: Pipes `|` and Redirects `>`

- Pipe (`|`) - Sends output of one command as input to another
  - Example: `$ cat file.txt | grep "important"`
- Redirection (`>`) - Redirects output to a file
  - `>` : Overwrites existing content
  - `>>` : Appends to existing content
  - Examples:
    - `$ echo "Hello" > greeting.txt`
    - `$ ls -l >> file_list.txt`

Command line

- Links
- Environment variables & `.env`
- Shell scripts
- `cron`
- Compressing/decompressing (`tar`, `zip/unzip`)

Links

You will almost always want a soft link, `ln -s`

- Symbolic links (soft links)
  - Create: `ln -s target_path link_path`
  - Example: `ln -s /mnt/c/Users/YourName ~/windows_home`
- Hard links
  - Create: `ln target_path link_path`
  - Limitations: Same filesystem, no directories

Linking Windows home to WSL home:

```
ln -s /mnt/c/Users/YourName ~/windows_home
```

- Creates a shortcut, `windows_home` in `~` (your WSL home directory)
- The shortcut points to `/mnt/c/Users/YourName`, your Windows home directory

Environment Variables & .env Files

- Environment variables: Key-value pairs in OS environment
    - Access: `echo $VARIABLE_NAME`
    - Set temporary: `export VARIABLE_NAME=value`
    - Set permanent: Add to `~/.bashrc` or `~/.bash_profile`
- Use the `env` command to view the current environment
- Pipe to `grep` to filter for variables of interest

Using `.env` files for *secrets*

- `.env` files: Store environment variables for projects
  - Create: `touch .env`
  - Format: `VARIABLE_NAME=value`

NEVER COMMIT `.env` TO YOUR REPOSITORY!!!

`.env` Files and Environment Variables

A robust function for setting environment variables:

```
load_env () {
    set -o allexport # enable the "allexport" option
    source $1        # set env var's from .env file
    set +o allexport # disable the "allexport" option
}

# Usage
load_env /path/to/.env
```

Loading `.env` files in Python

- Loading in Python:

  ```python
  import os
  from dotenv import load_dotenv
  load_dotenv()
  # Or if the file has a different name
  load_dotenv(<FILENAME>)

  variable_from_env = os.getenv('SOME_VARIABLE_NAME')
  ```

Setting default editor to nano

1. Open your shell configuration file:

   ```
   nano ~/.bashrc
   ```

2. Add this line at the end of the file:

   ```
   export EDITOR=nano
   ```

3. Save and exit (Ctrl+X, then Y, then Enter)

4. Reload the configuration:

   ```
   source ~/.bashrc
   ```

Shell Scripts

- Text files containing shell commands, one per line
- May use variables and flow control (different in `bash` vs `python`)
- First line: Shebang (`#!/bin/bash`) indicates the interpreter
- Make executable: `chmod +x script.sh`
- Run:
  - `./script.sh`
  - `bash script.sh`

Example shell script

```bash
#!/bin/bash
echo "Hello, World!"
for i in {1..5}; do
    echo "Count: $i"
done
```

Shell vs Environment Variables

- Shell variables are local to the current shell, set within the script or as part of the command to run the script:
    - `variable_name=value`
    - `variable_name=value bash script.sh`
- Environment variables are available to the current shell and its child processes, set inside or outside the script with:
    - `export variable_name=value`

Making scripts executable with `chmod`

Changes file **mod**e (permissions). Syntax: `chmod [options] mode file`

- Symbolic: `u` (user), `g` (group), `o` (others), `a` (all)
  - `+` (add), `-` (remove), `=` (set exactly)
  - `r` (read), `w` (write), `x` (execute)
- Numeric (advanced): 3-digit number (4=read, 2=write, 1=execute)
  - `XYZ` Digits: owner → group → everyone
  - Sum numbers to combine: read + write = 4 + 2 = 6

Tip: Use `ls -l` to view current permissions

`chmod` Examples

You are not expected to remember all of these! Look them up when needed until familiar

- Make executable for everyone:
  `chmod +x script.sh` (equivalent to `chmod a+x script.sh`)
- Give execute permission to owner:
  `chmod u+x script.sh`
- Set read & write for owner, read for others:
  `chmod 644 file.txt`
- Give all permissions to everyone (use cautiously):
  `chmod 777 file.txt`

Passing Arguments to Shell Scripts

- Access arguments with $1, $2, etc.
- $0 is the script name
- $# is the number of arguments

Example script (save as `greet.sh`):

```bash
#!/bin/bash
echo "Hello, $1! The weather is $2 today."
echo "This script name is $0"
echo "You passed $# arguments"
```

Usage: `./greet.sh Alice sunny`

Schedule Recurring Tasks with `cron`

- Edit crontab: `crontab -e`
- Syntax: `* * * * * command_to_execute`
  - Minute (0-59)
  - Hour (0-23)
  - Day of month (1-31)
  - Month (1-12)
  - Day of week (0-7, 0 or 7 is Sunday)
- `*` is (again) a wildcard that matches all

NOTE: `cron` doesn't run if the computer is off

`crontab` Examples

1. Every 15 minutes: `*/15 * * * * /path/to/script.sh`
2. Every day at 2:30 AM: `30 2 * * * /path/to/script.sh`
3. Every Monday at 9:00 AM: `0 9 * * 1 /path/to/script.sh`
4. First day of each month at midnight: `0 0 1 * * /path/to/script.sh`
5. Every weekday at 6:00 PM: `0 18 * * 1-5 /path/to/script.sh`
6. Hourly during business hours: `0 9-17 * * 1-5 /path/to/script.sh`

Check your `crontab` at https://crontab.guru

Logging Cron Job Output

- Redirect output to a file for debugging and monitoring
- Use `>>` to append, `>` to overwrite

Example crontab entry:

```
0 2 * * * /path/to/script.sh >> /path/to/logfile.log 2>&1
```

- `>> /path/to/logfile.log`: Append stdout to file
- `2>&1`: Redirect stderr to stdout (advanced)

Check logs: `tail -f /path/to/logfile.log`

Compressing and Decompressing

1. `tar`, `tar.gz`, `tgz`
    - Create: `tar -cvf archive.tar files`
    - Extract `.tar`: `tar -xvf archive.tar`
    - Compress (gzip): `tar -czvf archive.tar.gz files`
    - Extract `.tar.gz`: `tar -xzvf archive.tar.gz -C destination_dir`
2. `zip`
    - Compress: `zip archive.zip files`
    - Extract: `unzip archive.zip`
    - Compress directory: `zip -r archive.zip directory`

LIVE DEMO!!!

Python

- Data structures (lists, tuples, dictionaries, sets)
- Mention of list comprehensions

Lists

- Ordered, mutable collection of items
- Created with square brackets `[]` or `list()`
- Can contain mixed data types

```python
fruits = ['apple', 'banana', 'cherry']
numbers = [1, 2, 3, 4, 5]
mixed = [1, 'two', 3.0, [4, 5]]
```

List Operations

- Indexing: `fruits[0]` # 'apple'
- Slicing: `fruits[1:3]` # ['banana', 'cherry']
- Appending: `fruits.append('date')`
- Extending: `fruits.extend(['elderberry', 'fig'])`
- Removing: `fruits.remove('banana')` or `del fruits[1]`

List Operation Examples

`nums=[0, 1, 2, 3, 4, 5]`

- Indexing:
  - First element: `nums[0]` # 0
  - Second-to-last: `nums[-2]` # 4
- Slicing:
  - First three: `nums[:3]` # [0, 1, 2]
  - All but first: `nums[1:]` # [1, 2, 3, 4, 5]
  - Last three: `nums[-3:]` # [3, 4, 5]
  - Reverse: `nums[::-1]` # [5, 4, 3, 2, 1, 0]

List Operation Examples II

- Length: `len(nums)` # 6
- Min/Max/Sum: `min(nums)`, `max(nums)`, `sum(nums)` # 0, 5, 15
- Sum: `sum(nums)` # 15
- Append: `nums.append(6)` # Adds 6 to the end
- Insert: `nums.insert(2, -1)` # Adds -1 at index 2, shifting rest right
- Remove: `nums.remove(3)` # Removes first occurrence of 3
- Pop: `nums.pop()` # Remove and return the last element, 5
- Pop: `nums.pop(0)` # Remove and return the first element, 0

`len()` in Loops and Slicing

```python
# In loops
for i in range(len(my_list)):
    print(my_list[i])

# In slicing
# Note: `//` is "floor division", i.e., divide and round down
mid = len(my_string) // 2
first_half = my_string[:mid]
```

Generating lists with `range()`

- Creates a sequence of numbers
- Commonly used in for loops
- Syntax: `range(start, stop, step)`

Tip: `range()` is memory-efficient, generating values on-the-fly

`range()` Examples

```python
# 0 to 4
list(range(5))  # [0, 1, 2, 3, 4]

# 2 to 7
list(range(2, 8))  # [2, 3, 4, 5, 6, 7]

# 1 to 10, counting by 2
list(range(1, 11, 2))  # [1, 3, 5, 7, 9]

# Counting backwards
list(range(10, 0, -1))  # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Strings as Lists

Strings in Python are sequences of characters and can be treated similarly to lists in many ways

```
my_string = "Hello, World!"
# Access individual characters
print(my_string[0])   # Output: H
print(my_string[-1])  # Output: ! (last character)

# Use slices
print(my_string[0:5])    # Output: Hello
print(my_string[::-1])   # Output: !dlroW ,olleH (reversed)
```

Dictionaries

- Unordered collection of key-value pairs
- Created with curly braces `{}` or `dict()`
- Keys must be unique and immutable

```python
person = {'name': 'Bob', 'age': 25, 'job': 'Developer'}
scores = dict(math=90, science=85, history=88)
print(person{'job'}) # 'Developer'
print(scores{math})  # 90

# Can start with an empty dict and add/remove keys as needed
mylist = {}
mylist{'key'} = 'value'
```

Dictionary Operations

- Accessing values: `person['name']` or `person.get('name')`
- Adding/updating: `person['email'] = 'bob@example.com'`
- Removing keys: `del person['age']` or `person.pop('age')`
- Getting all keys: `person.keys()`
- Getting all values: `person.values()`
- Getting all items: `person.items()`
    - `items` are key-value pairs

Checking for Key Existence

Using `in`

```python
my_dict = {'b': 1, 'a': 2, 'c': 3}
# Using `in`
if 'a' in my_dict: print("Key 'a' exists") # True!
```

Using `.get()` to retrieve a value and (optionally) return a default value if the key doesn't exist

```python
# Using `.get()`
my_dict.get('d', 0)  # Returns 0 if 'd' doesn't exist
```

Sets

- Unordered collection of unique elements
- Created with curly braces `{}` or `set()`
- Useful for removing duplicates and set operations

```python
fruits = {'apple', 'banana', 'cherry'}
numbers = set([1, 2, 2, 3, 3, 4])  # {1, 2, 3, 4}
```

Set Operations

- Adding: `fruits.add('date')`
- Removing: `fruits.remove('banana')` or `fruits.discard('banana')`
- Union: `set1 | set2` or `set1.union(set2)`
- Intersection: `set1 & set2` or `set1.intersection(set2)`
- Difference: `set1 - set2` or `set1.difference(set2)`

Tuples

Constants, most often used to return results from functions

- Ordered, **immutable** collection of items
- Created with parentheses `()` or `tuple()`
- May contain heterogeneous data

```
coordinates = (10, 20)
person = ('Alice', 30, 'Engineer')
```

Tuple Operations

- Indexing: `person[0]` # 'Alice'
- Slicing: `person[1:]` # (30, 'Engineer')
- Unpacking: `name, age, job = person`
- **Cannot modify tuples after creation**

Nested Data Structures

- Combining lists, dictionaries, and other structures
- Common in real-world, multi-dimensional data

```python
# Mixed types
users = [
    {"name": "Alice", "age": 30, "skills": ["Python", "SQL"]},
    {"name": "Bob", "age": 25, "skills": ["JavaScript", "HTML"]}
]
print(users[0]["skills"][0])  # Output: Python

# Multi-dimensional
coordinate = [[0,5], [3,1], [4,2]]
print(coordinate[2][0]) # 4
# [4,2] is the 2nd index on outer list, 4 is the 0th index on [4,2]
```

Getting `sorted()`

Two built-in methods for sorting lists: `sort()` and `sorted()`

- `list.sort()` Sorts the list in-place, **modifying the original list**
    - Usage: `my_list.sort()`
    - Optional arguments:
        - `reverse=True` for descending order
        - `key` function for custom sorting (semi-advanced)
- `sorted()` Returns new sorted list, **leaving the original unchanged**
    - Usage: `sorted_list = sorted(my_list)`
    - Same optional arguments as `sort()`

Sorting Lists

- Use `list.sort()` (in-place) or `sorted()` (returns new list)
- Basic usage: `sorted(my_list)` or `my_list.sort()`
- Options: `reverse=True` for descending order, `key` for custom sorting

```python
numbers = [3, 1, 4, 1, 5, 9, 2]
print(sorted(numbers))  # [1, 1, 2, 3, 4, 5, 9]
print(sorted(numbers, reverse=True))  # [9, 5, 4, 3, 2, 1, 1]
```

Sorting Dictionaries

Dictionaries are unordered, but can sort by `key` or, with effort, `value`

```
my_dict = {'a': 2, 'b': 1, 'c': 3}

# Sorting by key, returns list of keys
sorted_keys = sorted(my_dict) # ['a', 'b', 'c']

# Sort by values, returns list of items, in reverse order
# Note: all values must be the same type, e.g., string or int
sorted_by_value = sorted(my_dict.items(), key=lambda x: x[1], reverse=True))
# [('c', 3), ('a', 2), ('b', 1)]

# Can also convert into a dict
sorted_by_value = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))
# {'c': 3, 'a': 2, 'b': 1}
```

Brief mention: List Comprehensions (advanced)

This is more advanced, but you are likely to come across it in the wild. I'll show an example in the demo.

- Syntax: `[expression for item in iterable if condition]`

```python
squares = [x**2 for x in range(10)]
even_squares = [x**2 for x in range(10) if x % 2 == 0]
```

Brief mention: `all()` (advanced)

`all()` is a built-in Python function for combining a list of booleans

- It returns `True` if all elements are true (or if the list is empty)
- Syntax: `all(list_of_booleans)`

Example:

```python
print(all([True, True, True]))  # Output: True
print(all([True, False, True]))  # Output: False
print(all([]))  # Output: True (vacuously true)
```

Summary

- Lists: Ordered, mutable collections
- Tuples: Ordered, immutable collections
- Dictionaries: Key-value pairs
- Sets: Unordered collections of unique elements
- List comprehensions: Concise way to create lists

LIVE DEMO!!!

Assignment

https://classroom.github.com/a/bTwHLV-s