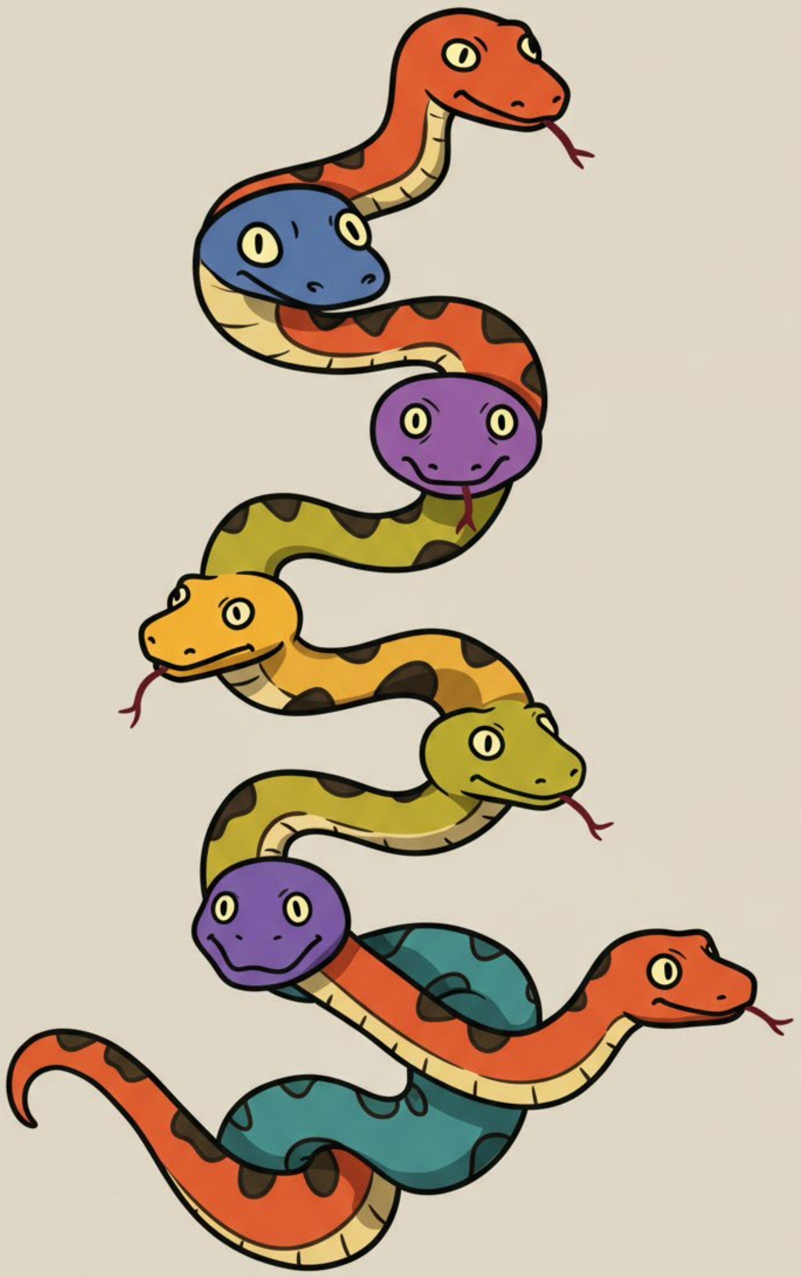
Week 2 – Data Structures and File Handling

CTD Python Essentials

list

- Mutable sequential data structure indexed with integers
- Implemented as dynamic arrays in contiguous memory
  - O(1) index and append unless resizing (O(n)) is needed
- Accessing via index
  - my_list[<index>] where index is zero based
  - my_list[<start>:<one-past>] returns a new list which is a slice
    - Omitting the first implies the beginning, omitting the last implies the end
    - [:] returns a shallow copy
    - Negative indices start from the end instead of the beginning
- Use len() to find the length of a list
  - Generic for any container in python
  - Or anything which implements __len__()



List Methods

- Lists supports lots of methods and operations
  - Common sequence operations
  - Mutable sequence operations
- Explicit conversion via list()

```python
# selected list methods

# Adding Elements

my_list = [1, 2, 3] # initial value for each operation
my_list.append(4)  # my_list will be [1, 2, 3, 4]
my_list.insert(1, 1.5)  # my_list will be [1, 1.5, 2, 3]
my_list.extend([4, 5])  # my_list will be [1, 2, 3, 4, 5]

# removing Elements

my_list = [1, 2, 3, 2]
my_list.remove(2)  # my_list will be [1, 3, 2]

my_list = [1, 2, 3] # initial value for each operation
item = my_list.pop()  # item will be 3, my_list will be [1, 2]
item = my_list.pop(0)  # item will be 1, my_list will be [2, 3]
my_list.clear() # my_list will be []

# Changing order in place

my_list = [3, 1, 2] # initial value for each operation
my_list.sort()  # my_list will be [1, 2, 3]
my_list.sort(reverse=True)  # my_list will be [3, 2, 1]
my_list.reverse()  # my_list will be [2, 1, 3]

# searching lists

my_list = [1, 2, 3]
idx = my_list.index(2)  # idx will be 1
my_list = [1, 2, 3, 2]
count = my_list.count(2)  # count will be 2
```

tuple

- Tuples are immutable sequential data structures
- Implemented as a fixed sized array in contiguous memory
  - More efficient than lists (no resizing mechanism)
- Support all of the common sequence operations
  - Can be treated like lists except for operations which make changes in place (mutate)
- Tuples can be used as dict keys and can be added to sets
  - lists can not

dict

- A mutable mapping data structure
  - key/value pairs
  - Associative array
- Implemented as hash tables
  - Lookup is efficient
  - Constant time lookup when there are no collisions
  - Unlikely worst case of O(n) when everything collides
- These are the [methods and operations](#) supported by dicts
- Key lookup and assignment using my_dict[<key>]
- Keys must be [hashable](#)
  - Immutable and can be compared to other objects
- Iterating
  - The dict itself provides an iterator over keys
  - The values() methods provides an iterator over values
  - The items() method provides an iterator over (<key>, <value>) tuples
- Size via the len() generic function
- remove everything using the clear() method
- shallow copy using the copy() method

```python
# selected dict methods

my_dict = {'name': 'Alice', 'age': 25}

# dict.get(key[, default])
value = my_dict.get('city', 'Unknown') # value is 'Unknown'

# update - add items from another dict in place
updates = {'age': 26, 'city': 'New York'} # overwrites age
my_dict.update(updates)
my_dict.update(country='USA', occupation='Engineer') # can use keyword args too
# {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA', 'occupation': 'Engineer'}

# dict.pop(key[, default]) # raises KeyError if missing and no default
job = my_dict.pop('occupation')
# my_dict now contains {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}

# dict.setdefault(key[, default])
# returns the value for key if it exists
# sets the key with the default if it doesn't
my_dict = {'name': 'Alice', 'age': 25}
city_value = my_dict.setdefault('city', 'Unknown') # added 'city' value as 'Unknown' and returns
name_value = my_dict.setdefault('name', 'No Name') # name value is 'Alice', my_dict unchanged

# dict.fromkeys(iterable, value=None)
# initialize a dict from an iterable containing keys
# set to a default or None if not supplied
new_dict = dict.fromkeys(['a', 'b', 'c'], 42) # {'a': 42, 'b': 42, 'c': 42}

# del is a statement, not a method or function
my_dict = {'name': 'Alice', 'age': 25}
del my_dict['age'] # my_dict is now {'name': 'Alice' }
```

Iterables and Iterators

- An Iterable is any Python object capable of returning its elements one at a time
  - lists, tuples, strings, dictionaries, and sets
  - Anything which implements __iter__()
  - Can be used in a for loop
- An Iterator is a lazy data structure which generates values one at a time as needed rather than all up front.
  - Efficient for very large data streams
  - Can be used in a for loop
  - range() returns an iterator
  - iter() converts a sequence to an iterator
  - dict.items() returns an iterator over key/value tuples
  - map() and filter() return iterators
  - Anything which implements __iter__() and __next__() is an iterator

set

- A mutable collection datatype which models sets
- Useful for uniquifying a collection
- Objects added to sets must be hashable
- Construction and explicit conversion using set()
  - Can also use e.g. {‘a’, ‘b’, ‘c’} as a set literal
- Use the add() method to add items
- Check membership using in or not in
- Use remove() to remove an item
  - discard() removes an items but does not check whether it exists
- The copy() method creates a shallow copy
- The clear() method creates empties the set
- These are the methods and operations supported by sets

```
# selected set operations

# intersection
a = {1, 2, 3}
b = {2, 3, 4}
print(a.intersection(b))  # {2, 3}
print(a & b)              # {2, 3}
# difference
print(a.difference(b))  # {1}
print(a - b)            # {1}
# symmetric difference
print(a.symmetric_difference(b))  # {1, 4}
print(a ^ b)                      # {1, 4}

# In place updates
# intersection_update
x.intersection_update(y) # {2, 3}
# difference_update
x.difference_update(y) # {1}
# symmetric_difference_update
x.symmetric_difference_update(y) # {1, 4}
# update (like union but modifies in place)
x = {1, 2, 3}
x.update([3, 4, 5]) # {1, 2, 3, 4, 5}

# Subset, Superset, and Disjoint Checks
a = {1, 2}
b = {1, 2, 3}
c = {4, 5}
a.issubset(b)   # True
a <= b          # True
b.issuperset(a) # True
b >= a          # True
a.isdisjoint(c) # True (since they have no elements in common)
```

File Handling

- The open function is used to create a file object
  - Sometimes referred to as a file handle or file descriptor
  - Establishes a connection with the operating system for input/output operations.
  - Important to close() a file object when you are done
    - For example write data is not necessarily saved to disk until it is closed
  - Typical modes
    - ‘r’ read from a file
    - ‘w’ write to a file, overwrites an existing content
    - ‘a’ append to a file, write after existing content
    - Defaults to a text file. Binary and buffered binary are also supported
  - Common methods
    - read() reads the whole file, readline() reads single lines
    - Typically the iterator is used : for line in file_object:
      - Does not strip newlines, use strip() for that
    - write() – does not include a newline (add ‘\n’)
- The with statement is usually used for file operations since it automatically closes open file objects on exit
- It’s important to trap exceptions for file operations
- Formatted strings: f’string content {<variable-value-inserted>} more string content}’

CSV and JSON

- The python standard library provides modules for conveniently accessing a variety of standard file formats.
- CSV is frequently used for import and export of spreadsheets and databases
- JSON data is also well supported
- Later in the class, we will use pandas DataFrames
  - These use their own methods to access csv and json

Modules and Libraries

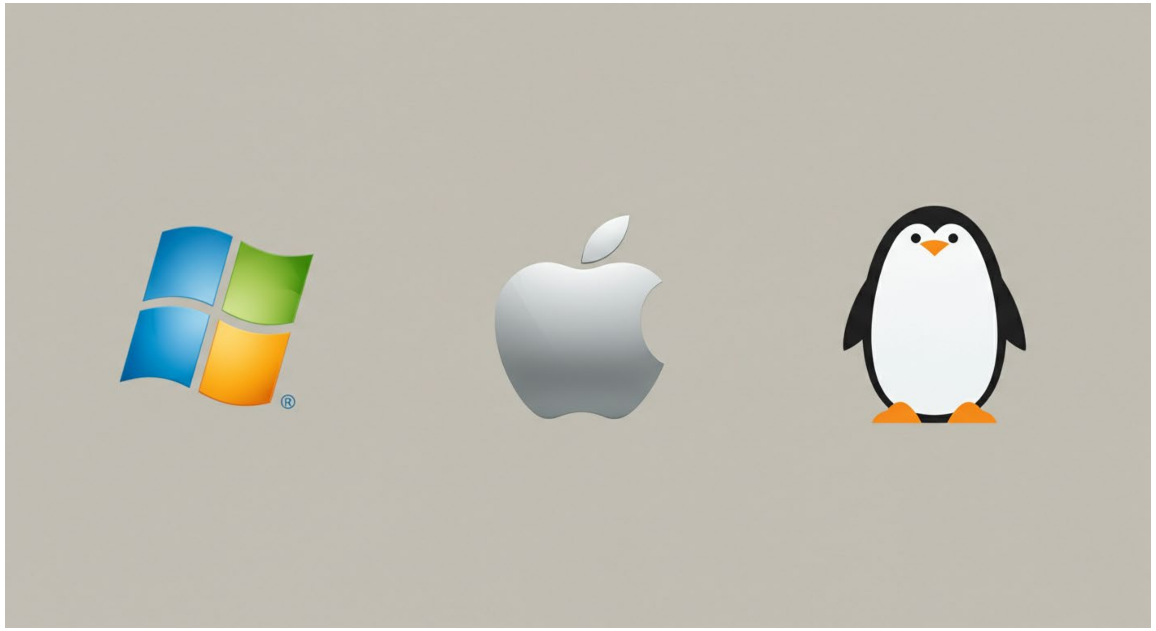
- Python provides a convenient module system for managing name spaces
  - Break programs up into multiple files
  - Avoid accidental name collisions
- Usage
  - import <module-name> # file name with no quotes or .py suffix
  - import <module-name> as <shorter-name>
  - To access names inside the module, qualify it with the module name
    - <module-name>.<name-in-module>
- Module search order
  1. Built in
  2. Local files
  3. The PYTHONPATH environment variable
  4. site-packages for the local installation
- For more complex projects you can create python packages
  - Standard structure for project management
  - Can manage multiple hierarchical modules together

Keyboard input

- The built in function input([<prompt>]) is used for keyboard input
- It prompts the user with the optional prompt string and returns a string with the newline removed

Interacting with the OS

- The os module provides a portable way to access operating system information
  - os.**chdir**(path) # change the working directory
  - os.**getcwd**() # get the current working directory
  - os.**getenv**(*key, default=None*) # get an environment variable
- The sys module is used to access system specific information
  - sys.**argv** # list of command line arguments where argv[0] is the script name
- The argparse module provide a feature rich way to specify and capture command line arguments
  - Use it for all but the most simple cases
  - Generates help messages



Command line scripts

```python
#!/usr/bin/env python3

import argparse

# Most of the script is implemented in separate functions
def calculate(number, multiplier):
    "compute a multiplication result (num * multiplier)"
    return number * multiplier

# handle command line arguments
def main():
    parser = argparse.ArgumentParser(
        description="A simple command-line calculator."
    )
    # Required positional argument: a number
    parser.add_argument("number", type=int, help="An integer number to be used in calculation")

    # Optional argument: multiplier with a default value of 2
    parser.add_argument("-m", "--multiplier", type=int, default=2, help="An optional multiplier (default is 2)")

    # Optional flag: uppercase
    parser.add_argument("-u", "--uppercase", action="store_true", help="Print the result in uppercase")

    args = parser.parse_args()

    # Perform a simple calculation
    result = calculate(args.number, args.multiplier)
    result_str = f"The result is {result}"

    # Print result
    if args.uppercase:
        print(result_str.upper())
    else:
        print(result_str)

    exit(0)

if __name__ == "__main__":
    main()
```

Virtual Environments

- Python provides a convenient way to manage package dependencies using virtual environments
- Specific interpreter, modules, libraries and binaries for a given project
- Standard practice is not to add the venv to the versioning system
  - Instead reproduce it as needed (manage requirements.txt version)
    - pip install -r requirements.txt # list of packages and revision numbers
- Steps
  - python -m venv .venv # the last argument can be any file path
  - source .venv/bin/activate # or source .venv/Scripts/activate on Windows
- Make sure you see (.venv) in your prompt

```
Owner@Galatea MINGW64 ~/code/ctd/python/jboa/me/python_homework (lesson1)
$ ls
README.md  assignment1/  assignment2/  assignment3/  csv/  db/  examples/  foo  load_db.py  requirements.txt

Owner@Galatea MINGW64 ~/code/ctd/python/jboa/me/python_homework (lesson1)
$ source .venv/Scripts/activate
(.venv)
Owner@Galatea MINGW64 ~/code/ctd/python/jboa/me/python_homework (lesson1)
$
```

Q&A
and
Demo