**MIDTERM NOTES**

**Week One: REVIEW**

Data Types:

* Numeric:
  + Int
  + Float
  + Complex
* Strings:
  + Str:
    - Define with x= ”string”
    - Index each character with []
    - Convert upper with “string”.upper()
    - Convert lower with “string”.lower()
    - Make into a list at a certain character point with.split(“HERE”)
    - Join a list into a string with ‘’.join([“hello”, “world”])
* Booleans:
  + Bool
* Collections:
  + List:
    - Defines with list() or []
    - Add them with .append
    - Access elements with list[0]
    - Change elements with list[0] = “new”
    - Concatenate with +
  + Tuples:
    - Defines with () or tuple()
    - Can not be altered once made (immutable)
    - Index with my\_tuple[0]
  + Set:
    - Defines with set() or {}
    - Entities will be made unique and unordered
    - Can be indexed by location
  + Dic:
    - Define with dict((1, “hello”), (2, “world”))
    - Or {1:”hello”, 2:”world”}
    - Index with dictionary[0]
    - Add values with dictionary[“newKey”] = “hi”
    - List keys with dict.keys()
    - List values with dict.values()
    - List key/values pairs with dict.items()

**Week Two: Modules, Venv, Best Practices:**

Create A Virtual Environment:

1. Make a folder and cd into it
2. python -m venv <NAME>
3. In Linux, start with: source name/bin/activate
4. (delete it by deleting the environment directory)

Code Formatting:

* Lowercase for variables and functions
* FakeCammelCase for Classes (title case?)
* Use UPPERCASE for constants
* To\_put\_a\_space\_use\_underscores
* “””Docstrings under methods and functions”””
* Organize code into functions and modules, remember to make the \_\_init\_\_.py when using packages and if \_\_name\_\_ == “\_\_main\_\_” in your modules

Pytest:

* Pytest --pdb
* Pytest -x <file>
* To install open a virtual environment and do pip install pytest
* The best practice is to cover 80% of the code with tests
* Don’t write more test code than application code
* Create a test file:
  + test\_\*.py
  + \*\_test.py
* Tests are functions whose names start with test\_
* Tests can be:
  + Regular functions outside of classes
  + Methods inside classes whose names start with a test
* Test files should be placed at the root of the folder
* Test examples:
  + The code (practice.py)

def add\_values(*a*, *b*): return *a*+*b*

* + The test (test\_practice.py)

import pytest

from practice import add\_values

def test\_adding():

result = add\_values(2,3)

assert result == 5

* Errors are red fs, passes are green dots
* Tests for a particular function should be stored in the same test function
* Assert is used to check if a value matches

**Week Three: Intro to Classes and Object Orientated Programming**

Keywords:

* Class: defines a new class
* Self: used to call methods of a class within a class
* Super: used to call upon methods and attributes of parents from the child class

Basic Definitions:

* Classes: define general categories, these are the custom data types that create the blueprint for an object
* Attributes: Values for a specific object
* Methods: behaviours (or capabilities) of a class
* Object or instance: A specific instance of a class
* State: The current values of the attributes in an object
* Modueliztaition: the objective of a model, describes an object (think init attributes)
* Encapsulation: The state of an object should only be changed from inside the object, and only methods should be able to tell it to alter its state (treat it as a black box)
  + Use \_ at the start of a variable to indicate that it is private
* Abstraction: The outline of a class from a higher level, the concept of an object, but separated from a specific instance
* Inheritance: Classes can take the classes of pre-existing classes and expand upon them
* Polymorphism: Classes can have the same method but perform slightly different results
* Getters: methods that return an attribute of an object
* Setters: A type of method that updates (changes) attribute of an object
* @property: a decorator that can be used on a method to make it behave as an attribute (read from it with object. method)
* @[method name].setter: a decorator that can be used to set a method as if it were an attribute
* Static Methods: do not get any reference to the class or object (the decorator for these is @staticmethod)
* Class Methods: receive a reference to the class when called directly from the class name (the decorator for these is @classmethod) These also receive a reference to their class as the implicit first argument (not self)
* Class Variables: Attributes that are common to all objects of the same class

Coding Structures:

Procedural Code:

Step by step operational code to be executed, often organized into functions (term 1)

Object-Oriented Code:

code that describes objects and what they can do

Both options have the same result but different advantages and disadvantages.

Object-Oriented code tends to be:

* Easier to reuse
* More bloated for small projects
* extendable/easily changeable
* Better for larger projects
* Groups together the data and the functions that operate on that data into one place
* Promotes the modularization of code
* Isolates code from each other

Objects:

* Often found in the real world and are independent of other objects they share a class with
* Objects may be very complex, they can even contain other objects
* Can be used to describe non-tangle things (a bank account, a game)
* Classes are written with:
  + Names (identity):
    - Written as singular with a capital letter
  + Attributes:
    - Most objects have many
    - Some are red only, some are changed
    - These are also called properties, characteristics, state, fields, variables…
  + Methods (behaviours):
    - Verb+Noun structure
    - Methods are like functions, but they only apply to objects made from that class
    - Often used to get or set something (generally internal class attributes)
* Objects are not classes and classes are not objects
* Objects are the substance created from the outline of a class
* Terms to remember for OOP:
* Class:
  + A blueprint for objects
* Methods:
  + Class-specific built-in functions
* Attributes:
  + Variables that hold data that are part of a class
* Object:
  + A specific instance of a class
* Inheritance:
  + Means by which a class can inherit capabilities from another
* Composition:
  + Building complex objects out of other objects

**Week Four: UML and Relations**

Composition: “has a”

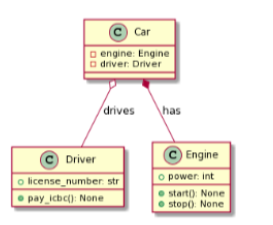
Two interrelated objects that can not exist without one another (car and engine)

Drawn as a solid filled-in square.

Aggregation: “use a”

Two objects that are connected but exist independently (car and driver)

Drawn as a not filled in square connecting two classes.



Polymorphism:

we don’t care about the types of objects when applying methods

General UML notes:

* - means a private attribute
* + means a public one
* Composition: written as a filled-in square connection from the dependent to the one it is dependent on (engine is dependent on the car, the car has an engine)
* Aggregation: written as a hollow square
* Extention: written as an arrow where the child points at the parent

Inheritance:

The child duplication of a class that adds upon the parent.

To get access to the qualities of a parent class use super()

isinstance(thing,parent) will check if the thing is that parent’s child

type(thing) will only return the direct class of the thing

Example:

class Pet:

def \_\_init\_\_(*self*, *name*):

*self*.name = *name*

def say\_hello(*self*):

print(f"{*self*.name} say hi!")

class Cat(Pet):

def sleep(*self*):

print("zzzzzzzzzz")

class Dog(Pet):

def \_\_init\_\_(*self*, *name*):

print("In the dog zone")

super().\_\_init\_\_(*name*)

def do\_trick(*self*):

print(f"{*self*.name} play dead")

def say\_hello(*self*):

print("Woof Woof")

super().say\_hello()

* Get a parent argument value attribute with super().\_\_init\_\_(thing)
* Get a parent method with super().method\_name()
* Attributes or methods from the parent can be overwritten by the child
* Private variables are written with an underscore \_infront, eg self.\_age

UML Implementation in Python:

Composition:

* Example: a hand with five fingers
* Objects from other classes are usually created when the \_\_init\_\_ happens
* For this the fingers are part of the hand and do not exist independently, so it is a composition

class Finger:

def flick(*self*):

print("OH NO!!!")

class Hand:

def \_\_init\_\_(*self*):

*self*.\_fingers = [Finger() for \_ in range(5)]

def thumbs\_up(*self*):

*self*.\_fingers[0].flick()

my\_hand = Hand()

my\_hand.thumbs\_up()

Aggregation:

* Example: the car and the driver

class Driver:

def \_\_init\_\_(*self*, *name*):

*self*.name = *name*

class Car:

def \_\_init\_\_(*self*, *driver*):

*self*.driver = *driver*

#now you can associate the two objects when creating your car

sam = Driver("Sam")

my\_car = Car(sam)

my\_car = Car(*driver*=sam) #same thing as the line above

Decorators:

* A specific type of function that returns a wrapper function
* When a function is wrapped, only the wrapper is technically available
* Write decorators by putting @function\_name above the function name
* Wrapper functions trigger after the main function

**Week Five: Reading JSON**

Python - JSON mapping

* Dict = object
* List/tuple = array
* Str = string
* Int, long, float = number
* None = null

Parsing JSON into Python:

* This is the reverse of the above with some changes:
* Object = dict
* Array = list
* String = str
* Integer number = int
* Floating-point = float
* True, false = True, False
* Null = None

JSON: Javascript Object Notation

Format with alt+shift+f

Pros:

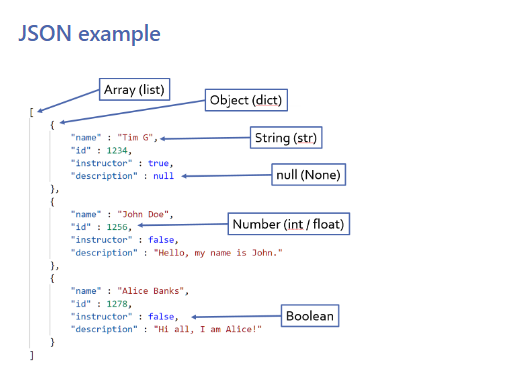
* Lightweight
* Language independent
* Easy to read/write
* Flat (text format)

Cons:

* Very flexible - no validation
* No comments
* Hard to debug

Structure:

* Key-values pairs for objects
* Keys are strings
* Values can be:
  + Strings
  + Numbers (int floats)
  + Objects (another JSON nested block)
  + Arrays
  + Special (booleans, null)



Definitions:

* Serialization: creating JSON from data
* Deserialization: Creating data JSON

Using JSON in python:

* Import json
* Open and read from the file

import json

with open("trivia.json") as fp:

data = json.load(fp)

* Load: reads from a json file
* Loads: reads from a json string
* dump(obj, file): writes to a file
* dumps(obj): this writes a string (can store as a variable)
* To format:
  + json.dumps(object, indent=4)

JSON Exception Handling:

* Import the JSONDecodeError function
* From json import JSONDecodeError
* Example:
* try:
* Data = json.load(jsonFile)
* Except JSONDecodeError as err: print(“json error”)
* This is best practice for data we do not have control over

**WEEK SIX: Using Magic Methods~**

Magic methods are inclosed in \_\_ (double underscores, sometimes called dunder methods). we do not call them, they call themselves.

Example of a class with magic method functions:

class Person:

def \_\_init\_\_(*self*, *name*):

*self*.name = *name*

def \_\_str\_\_(*self*):

return f"Hello, my name is {*self*.name}"

def \_\_repr\_\_(*self*):

return f"<Person: {*self*.name}>"

p = Person("shawn")

Note that \_\_repr\_\_ is called when the object is viewed in the terminal, and \_\_str\_\_ is called when the object is printed

Magic Method Index:

* \_\_str\_\_(self): For print(instance), or str(instance)
* \_\_len\_\_(self): Allows len(instance)
* \_\_getitem\_\_(self,key): Allows for instance[] (this can be a slice)
* \_\_call\_\_(self): Allows to use instance()
* \_\_contains\_\_(self, other): Allows to test something in instance expression
* \_\_iter\_\_(self): Allows to use instance as an iterator
* \_\_next\_\_(self): Returns the next value in the iteration (see above)
* (note that for math methods, only the instance on the left is important here)
* \_\_add\_\_(self, right): Using instance + something
* \_\_sub\_\_(self, right): Using instance - something
* \_\_mul\_\_(self, right): Using instance \* something
* \_\_pow\_\_(self, right): Using instance \*\* something
* \_\_mod\_\_(self, right): Using instance % something
* \_\_eq\_\_(self, right): Using instance == something
* \_\_lt\_\_(self, right): Using instance < something. Needed for sorting
* \_\_le\_\_(self, right): Using instance <= something
* \_\_gt\_\_(self, right): Using instance > something

Packing/Unpacking Arguments:

* The syntax is \*args (for an iterable) and \*\*kwargs (for a dictionary)
* Examples:
* For an iterable:

def function(\**args*): print(*args*) #prints out ('yes', 2)

function("yes", 2)

* For a dictionary:

def function(\*\**kwargs*): print(*kwargs*) #prints out {'question': 'yes', 'number': 2}

function(*question*= "yes", *number*= 2)