Object Oriented Programming in Python
Week 12

Course Recap & More

Prof. Rabih Neouchi

Python Built-in Classes

- **Iterable data structures**:
  - **strings** (immutable, ordered collection)
  - **lists** (mutable, ordered collection)
  - **tuples** (immutable, ordered collection)
  - **dictionaries** (mutable, unordered collection)
  - **sets** (mutable, unordered collection)

- **Mutable**: Internal state/content can be changed or modified once the object has been created
- **Ordered collection**:
  - Output of elements appears in the same order they are specified in initially
  - Can access specific elements using indexing

Python Built-in Classes

	Empty Constructors	Default Initialization	Built-in Functions	Built-in Operators
string	my_str = '' print(my_str)	my_str = 'Hi' print(my_str) Hi	len(), min(), max(), find(), index(), capitalize(), upper(), isupper(), lower(), islower(), strip(), lstrip(), rstrip(), split(), replace(), startsWith()	in, not in, del, +, *, <=, >= String slicing
list	my_list = [] my_list = list() print(my_list) []	my_list = [1,2,3] print(my_list) [1,2,3]	len(), min(), max(), index(), range(), list(), sum(), count(), remove(), append(), extend(), insert(), pop(), remove(), reverse(), sort()	in, not in, del, +, * List slicing
tuple	my_tuple = () my_tuple = tuple() print(my_tuple) ()	my_tuple = (1,2,3) print(my_tuple) (1,2,3)	len(), min(), max(), index(), sum(), count()	in, not in, del, +, *, <=, >= Tuple slicing
dictionary	my_dict = {} my_dict = dict() print(my_dict) {}	my_dict = {'Course':'MIS6382', 'School': 'JSOM'} my_dict = dict(Course='MIS6382', School='JSOM') print(my_dict) {'Course':'MIS6382', 'School': 'JSOM'}	len(), min(), max(), sorted(), get(), keys(), values(), items(), clear()	in, not in, del
set	my_set = set() print(my_set) set()	my_set = {'bananas', 'oranges', 'apples'} my_set = set(('bananas', 'oranges', 'apples')) print(my_set) {'bananas', 'oranges', 'apples'}	len(), clear(), add(), update(), discard(), remove(), copy(), union(), difference(), symmetric_difference(), intersection(), issubset(), issuperset()	in, not in, del, |, &, -, ^, <=, >=

Looping Structures

- **range()** function
- **for** loop
  - **Definite** loop: know upfront the number of times the loop will be executed
- **while** loop
  - **Indefinite** loop: don't know upfront how many times the loop will be executed
  - Exiting the loop depends on evaluating a conditional expression at every iteration
  - Make sure you control variables of conditional expression inside the body of the loop
  - **Beware** of infinite or dead loops
  - **Beware** of **while True**, or **while False** loops (the only way to exit is using a break statement!)
- **break** statement
  - Breaks control of execution to next statement following the loop
- **continue** statement
  - Breaks control of execution to the next iteration in the loop

File I/O

	Read Mode	Write Mode	Append Mode	with/as Construct

text files	f = open('file.txt', 'r')
	f.read()
	for line in f.readlines()
	  print(line, end = '')
	f.close()
		f = open('file.txt', 'w')
		f.write('text' + '\n')
		for i in f.range(11)
		  f.write('line: ' + str(i) + '\n')
		f.close()
			f = open('file.txt', 'a')
			f.write('text' + '\n')
			for i in f.range(11, 21)
			  f.write('line: ' + str(i) + '\n')
			f.close()
				with open('file.txt', 'r') as f:
				  statement(s)

binary files	import pickle fb = open('file.dat', 'rb') obj = pickle.load(fb) f.close()	import pickle obj = object() fb = open('file.dat', 'wb') pickle.dump(obj, fb) f.close()	import pickle obj1 = object() fb = open('file.dat', 'ab') pickle.dump(obj1, fb) f.close()	with open('file.dat', 'rb') as fb: statement(s)

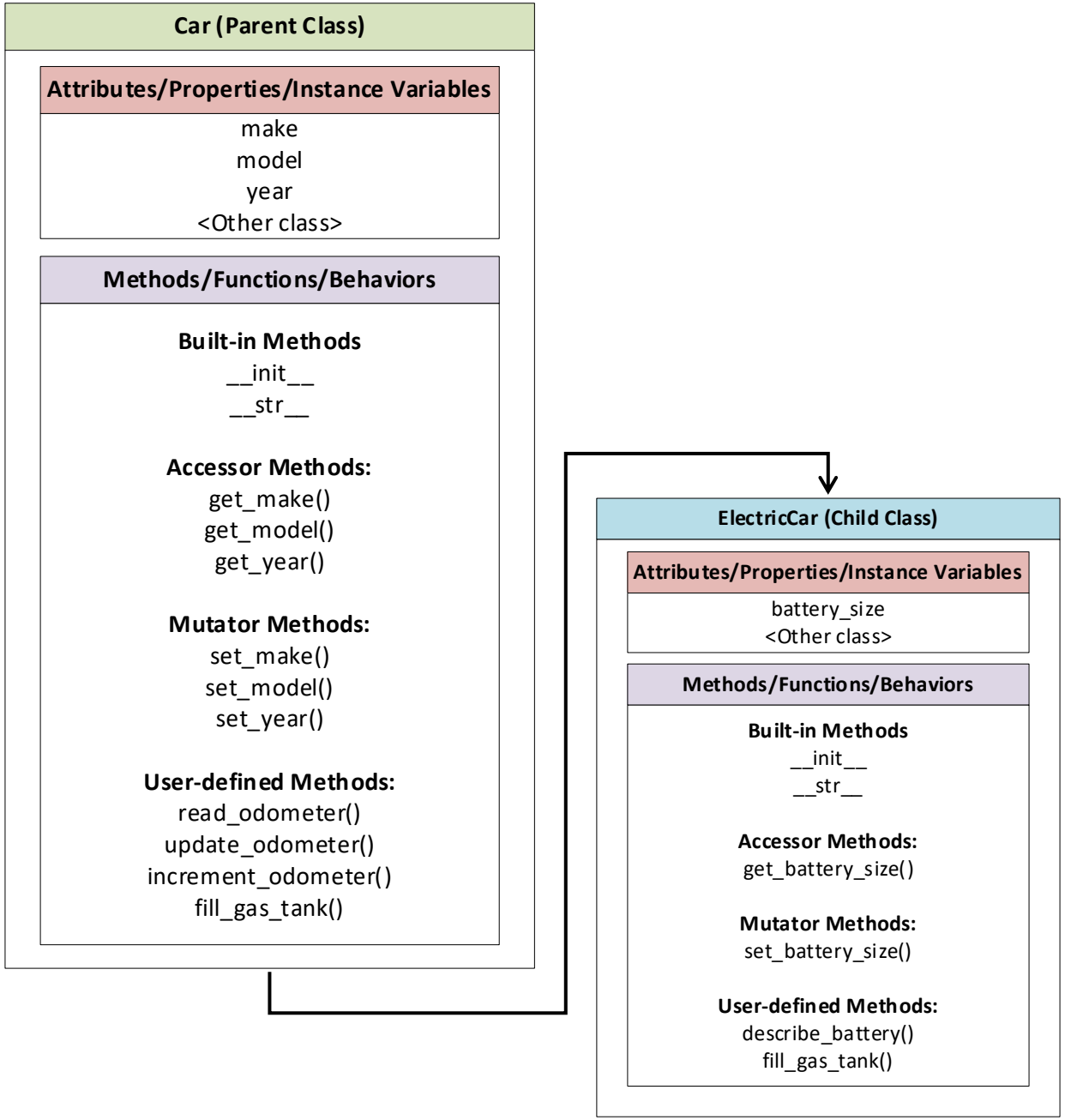
- Default mode is '**r**'
- Other file methods: seek(), tell()
- **with/as** construct:
  - Indented statement(s) within block
  - Ensures **file handle** is **automatically closed** upon exiting construct scope

Exception Handling

- **Idea**: Handle any thing that can go wrong during execution
- **Objective**: Allow your code to fail gracefully
- **Built-in** Exceptions: Exceptions are thrown as they occur (Python specific)
- **User-defined** Exceptions: Raise your own exceptions (logic specific)
- **try/except/[else/finally]** blocks
  - Sandwich code that may throw exception(s) inside the **try** block
  - Catch thrown/raised exceptions inside the **except** block
  - If no exceptions are thrown/raised inside try block, statements are executed in the **else** block
  - In all cases, statements are always executed in the **finally** block
- Can handle **multiple** exceptions **separately**
  - **Multiple except** blocks: Executed in the order they are written
- All Exceptions (built-in and user-defined) inherit from the **Exception** (generic) class

Python User-Defined Classes

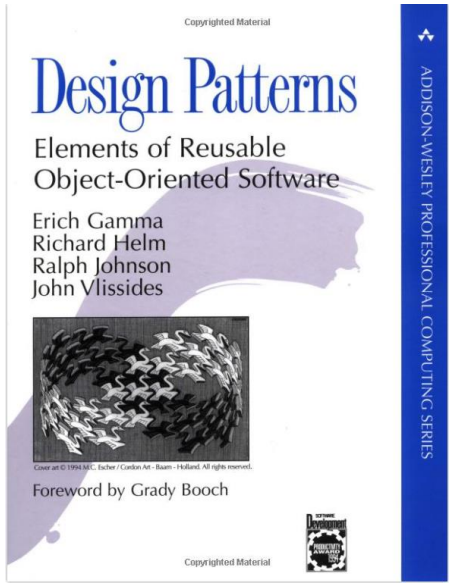
- A **class** is a **abstract blueprint** of an entity
- An **object** is a **single instance** of a class
- **Properties**/Attributes/Instance Variables
- **Methods**/Functions/Behaviors
  - Built-in Methods
  - Accessor Methods
  - Mutator Methods
  - User-defined Methods

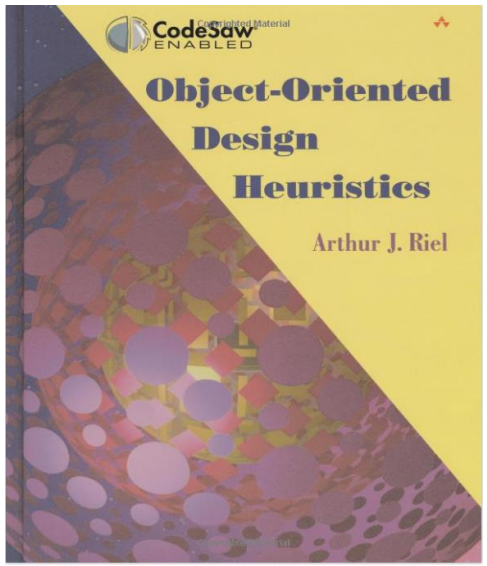


OOP Principles

- **Encapsulation** (data hiding)
  - Hide instance variables and expose them only through accessor and mutator methods
- **Inheritance**
  - Transitive, ‘is-a’ relationship between a super (parent) and a sub (child) class
  - Sub-class is a **specialized** (more improved version) of the super-class
  - All Properties and Methods of the super-class are inherited by the sub-class
  - All classes (built-in and user-defined) inherit from the **object** (generic) class
- **Polymorphism**
  - Ability of an object to behave differently at run time depending on the type of the object itself
  - Polymorphic functions
  - Polymorphic types

Further OOP Readings





- Gang of Four (GoF) 23 Design Patterns
- References:
  - https://refactoring.guru/design-patterns
  - https://sourcemaking.com/design_patterns
  - https://springframework.guru/gang-of-four-design-patterns/
  - https://github.com/nitinmuteja/GOFDesignPatterns/

Purpose	Design Pattern	Aspect(s) that can vary
Creational	Abstract Factory	families of product objects
	Builder	how a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class
Structural	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Facade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location
Behavioral	Chain of Responsibility	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed, traversed
	Mediator	how and which objects interact with each other
	Memento	what private information is stored outside an object, and when
	Observer	number of objects that depend on another object; how the dependent objects stay up to date
	State	states of an object
	Strategy	an algorithm
	Template Method	steps of an algorithm
	Visitor	operations that can be applied to object(s) without changing their class(es)

Visualization

- **pandas**
  - fast, powerful, flexible and easy to use open source data analysis and manipulation
- **numpy**
  - Numbers and scientific computing with Python
  - Distributed, and sparse array libraries
- **matplotlib**
  - Comprehensive library for creating static, animated, and interactive visualizations in Python
  - https://matplotlib.org/
- **seaborn**
  - Built on top of matplotlib
  - https://seaborn.pydata.org/
  - https://pypi.org/project/seaborn/

Modeling Tip

- If you use R or Python:
  - **Interpreted** (not compiled) languages
  - **Avoid** using **loops** (for & while statements) with these languages
  - Always **think** of your data as a **data frame**, or table (with **rows** & **columns**)
  - For every operation you want to do to your data, there is a **fast** library that would help you achieve (at either the **row** or **column** level) what you need done on the data **without** the need to loop
  - E.g. **pandas**, **numpy** packages in Python
  - E.g. **dplyr** package in R

Course Evaluation

- Please fill out & complete the evaluations. You can do so on your phones, tablets, or laptops.
- I do value your (honest, constructive and actionable) feedback. It helps me improve future iterations of the course.
- Your responses are anonymous. I do not receive any information on student specific responses.
- **Note**: Course Evaluation is **optional**. Submitting a course evaluation will result in a **1% bonus** of the total final grade.

Final Exam

- Multiple choice questions.
- 35 points: 50 questions @ 0.7 point each.
- Calculator maybe needed.
- Duration: 2 hours.
- Time: 05/07/2024 – 05/10/2024 (UTD Testing Center Hours).
- Location: UTD Testing Center (Must Register).
- I will not ask you to write Python Code.
- **Notes**: 3 pages double sided (Handwritten or printed). Will be collected after the test.
- **No make-up exams (no exceptions)!**
- **Tip**: Do not memorize. Have a good rest prior the exam, so that you are able to generalize!

Congratulations!

- You are officially a **junior software (data) engineer**! (if you grasped the material discussed in this course)
- Wish you sincerely the best of luck with your careers ahead
- Available for help & future career advice!