Lecture 28: Conclusion

CS 61A - Summer 2024
Raymond Tan

Final Review

Object-Oriented Programming

[Concept] Object-Oriented Programming

Object-oriented programming focuses on distinguishing objects using two aspects:

- their characteristics -> class/instance variables
  - Class variables apply to all instances of the object; however, they can be overridden
  - Instance variables only apply when the object has been instantiated
- what they do -> methods
  - Instance methods vs class methods

[Concept] Class vs. Instance

- Class variables/methods apply to the abstract concept of the instance and possibly to the actual instance itself
    - In other words, even if you don’t instantiate the dog class, it is generally known that a dog can bark via a class method called bark().
    - When you have an initial impression of an object, those characteristics can usually be written as class variables

- Instance methods/variables apply to individual instances of the object. These only apply after instantiation has taken place.

[Concept] Inheritance & Overriding

- It is possible to create subclasses of a parent class.
    - For example, you can make a child class called Chihuahua which inherits from the parent class Dog.
    - In order to inherit, use the .super() method.

- What if you want the child class to be different in some way from the parent class?
    - That's totally possible! You can override a parent class variable/method by redefining it within the child class

    - When the variable/method is called, the redefined method/variable for the child will be used instead of the parent's method/variable.

```
class Dog:
    legs = 4
    def __init__(self, breed):
        self.breed = breed
    def bark(self):
        print ("Woof!")


class Chihuahua(Dog):
    def __init__(self):
        super().__init__("Chihuahua")
    def bark(self):
        print ("Yeet!")
```

[Concept] Str vs Repr

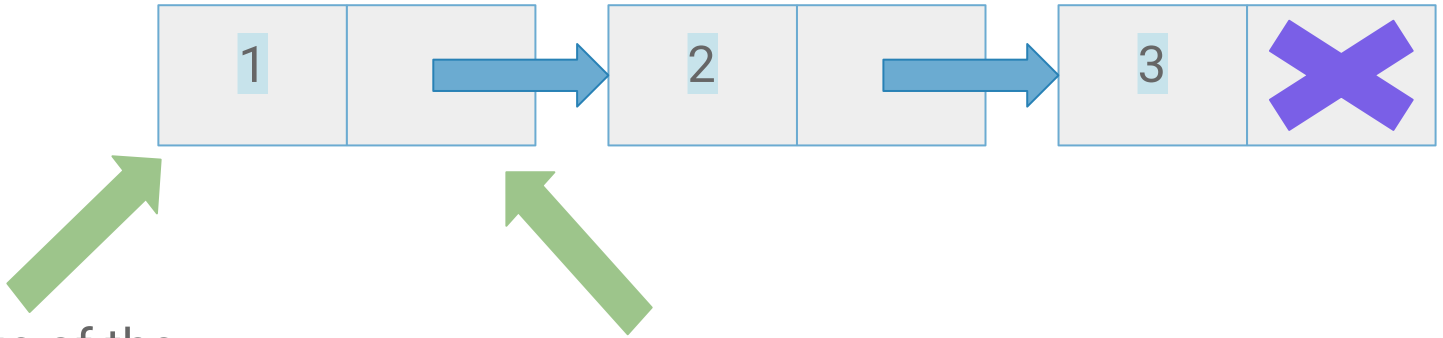
- __str__ and __repr__ both return Python expressions.
- Their difference lies in what they're intended for:
    - __repr__ is not meant for the client. It's made to return an accurate representation of a Python object for the purposes of the programmer/machine.
    - __str__ is meant to be displayed to the client.
- str(obj) calls obj.__str__ and repr(obj) calls obj.__repr__, respectively
- print(obj) calls str

Practice: Fall 2020 MT2 Q3

Linked Lists

[Concept] Linked List

```
Link(1, Link(2, Link(3)))
```



Value of the `first` is the number 1

Value of the `rest` is a pointer to another Linked List

Practice: Spring 2017 MT 1 Q5

Practice: Lab 12 Q12 (Trees)

Scheme

[Concept] Scheme Overview Part 1

- Scheme is a functional programming language that consists of expressions, which can be:
  - Primitive expressions: a value that evaluates to itself
    - Ex: 3 evaluates to 3, #t evaluates to #t
  - Combinations: either a call expression or a special form
    - Ex: (operator operand1 operand2 …)
- We see several types of combinations in Scheme
  - Call Expressions: include an operator and 0 or more operands
    - (+ 1 3) or (quotient 10 2)
  - Special Forms : A combination that is not a call expression
    - (if 1 3 5) or (define x 10)

[Concept] Scheme Overview Part 2

- How do we evaluate expressions?
  1. Evaluate the operator
  2. If the operator is NOT a special form: Evaluate the operands, apply the operator to the operands
  3. If the operator IS a special form, evaluate based on the special form's evaluation rules
- Scheme does not use iteration!
  - We will be using recursion instead!

[Concept] Scheme Special Forms Part 1

- Special forms are pre defined language features
  - `if`
    - `(if <predicate> <do if predicate is true> <do if predicate is false>)`
  - `cond`
    - `(cond (<Pred1> <Expr1>) (<Pred2> <Expr2>) .....  (else <expr>))`
    - Goes through each Pred until one is true and if it is it returns the corresponding expression
    - Similar to if, elif, and else statement in python
  - `define`
    - `(define <name> <expression>)`
    - Assigns the value of an expression to a name
    - `(define (<name> [param] ...) <body>)`
    - Defines a function based on a body
    - Note this returns the name of the function not the function itself unlike Lambda which returns a function

[Concept] Scheme Special Forms Part 2

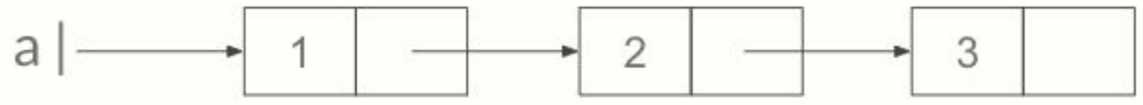
- `and`
  - `(and <expr1> <expr2>...)`
  - Evaluated the expressions in order until it reaches a false value and returns the false value, or returns the last expression
- `or`
  - `(or <expr1> <expr2>..)`
  - Evaluated the expressions in order until it reaches a true expression and returns the true expression. If there are no true expressions it returns the last expression
- `let`
  - `(let [binding] <body>)`
- `begin`
  - `(begin <expression>..)`
  - Evaluates each expression in order
- `append`
  - `(<append <list1> <list2>)`
  - Returns a NEW list of list1 followed by list2

[Concept] Scheme Lists Part 1

- Scheme lists are similar to the linked list data structure we saw in Python!
- There are two parts to a scheme list:
  - **`car`:** this is the first part of the list (similar to `Link.first`)
  - **`cdr`:** this is the rest of the list (similar to `Link.rest`)
- **Constructing Scheme Lists**
  - There are three ways to create Scheme lists:
  - Usings **`cons`**:
    - `(cons <first> <rest>)`
    - Takes two arguments!
  - Usings **`list`**
    - `(list <elem1> <elem2> <elem3> …)`
    - Add as many elements as you want
    - These elements are evaluated
  - Using **`quote`**
    - `‘(<elem1> <elem2> <elem3> ….)`
    - We do not evaluate the elements in this list

[Concept] Scheme Lists Part 2

Scheme	Python	Value/Representation in Scheme
(define a (cons 1 (cons 2 (cons 3 nil))))	a = Link(1, Link(2, Link(3, Link.empty)))	
(car a)	a.first	1
(cdr a)	a.rest	(2 3)
(cadr a), or (car (cdr a))	a.rest.first	2
(cddr a), or (cdr (cdr a))	a.rest.rest	(3)



Practice: Fall 2018 Final Q5

Tail Calls

- A tail call is a call expression in a **tail context**:
    - The last body sub-expression in a `lambda` expression (or procedure definition)
    - Examples (non-exhaustive list):
        - Sub-expressions 2 & 3 in a tail context `if` expression
        - All non-predicate sub-expressions in a tail context `cond`
        - The last sub-expression in a tail context `and`, `or`, `begin`, or `let`

[Practice] Tail Recursion

- Write a tail-recursive function that calculates the length of a scheme list
- Normal implementation (not tail-call optimized):

```
(define (length s)
    (if (null? s) 0
    (+ 1 (length (cdr s)) ) )
```

[Practice] Tail Recursion

- Tail-optimized version
- Notice how we made a nested function so we can keep an intermediate result to return at the base case (stored in the variable n)
- Since we're updating n after each call, we don't need to store previous frames

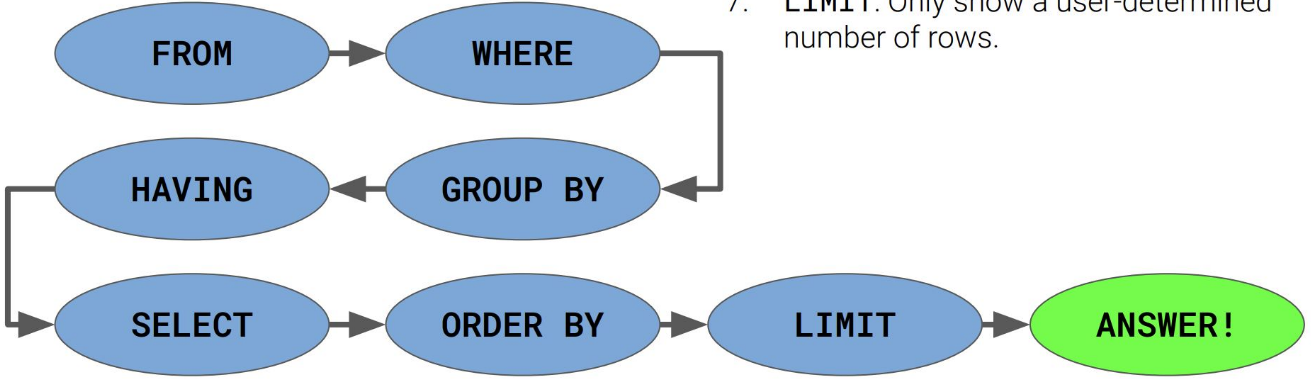
```scheme
(define (length-tail s)
    (define (length-iter s n)
        (if (null? s) n
        (length-iter (cdr s) (+ 1 n)) ) )
    (length-iter s 0) )
```

SQL

[Concept] Query Structure & Order of Operations

```
SELECT <column/expression list>
FROM <table>
[WHERE <condition>]
[GROUP BY <column(s)>]
[HAVING <condition>]
[ORDER BY <column(s)> [DESC/ASC]]
[LIMIT <number of rows>];
```

1. `FROM`: Retrieve the tables.
2. `WHERE`: Filter the rows.
3. `GROUP BY`: Make groups.
4. `HAVING`: Filter the groups.
5. `SELECT`: Aggregate into rows and get specific columns.
6. `ORDER BY`: Sort by certain columns (optionally ascending/descending, default is ascending).
7. `LIMIT`: Only show a user-determined number of rows.



Practice: Fall 2023 Final Q8

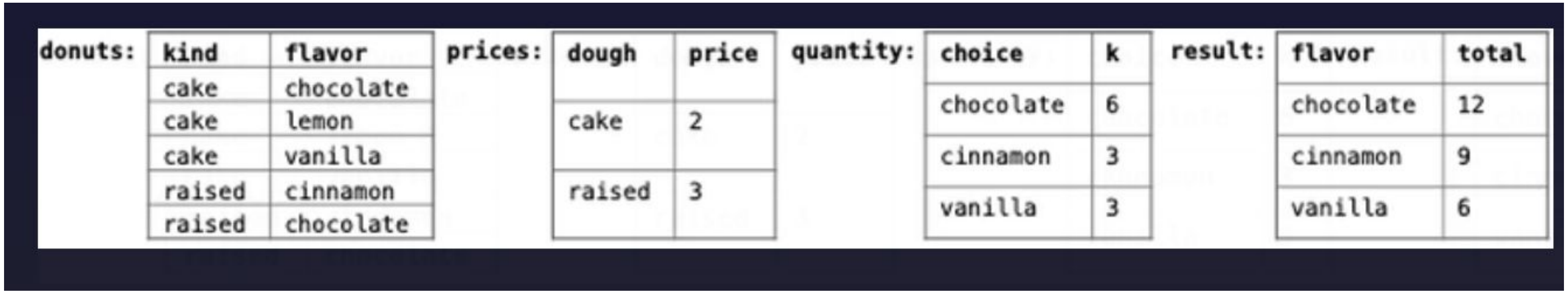
8. **(6.0 points) Cheap Donuts**

There are three tables in a database:

- The `donuts` table has a row for each menu option at a donut shop. There are columns for the `kind` (string) of dough and `flavor` (string). For example, there is one row for chocolate cake donuts (although the store may have many such donuts, they only have one menu option for this kind & flavor combination).
- The price of a donut depends only on the dough. The `prices` table has a row for each kind of dough. The `dough` (string) column contains the kind; the `price` (number) column is for one donut made from that kind of dough.
- Your friends only care about the flavor, not the kind of dough. The `quantity` table contains one row for every flavor your friends want. The `choice` (string) column is the flavor they want and the `k` (number) column is the number of donuts of that flavor they want.

Create a table with two columns, `flavor` (string) and `total` (number) with one row for each flavor your friends want. The `total` column contains the **least expensive** total cost of buying `k` donuts of that flavor, where `k` is the number your friends want.

The rows of the result can appear in any order. Here is an example, but complete the query so that it would work even if the contents or number of rows were different.



donuts:

kind	flavor
cake	chocolate
cake	lemon
cake	vanilla
raised	cinnamon
raised	chocolate

prices:

dough	price
cake	2
raised	3

quantity:

choice	k
chocolate	6
cinnamon	3
vanilla	3

result:

flavor	total
chocolate	12
cinnamon	9
vanilla	6

Break

61a wrapped

Paradigms

We started with imperative programming which used statements to change the environment of a program

We moved onto functional programming which used functions instead of statements to achieve things

We discussed object oriented programming which let us create our own objects

Finally, we learned about declarative programming where we told the computer the output we wanted

Concepts

- Iteration
- Data Structures
- Recursion
- Environments
- Mutability
- Efficiency
- Abstraction

Throwback – What is 61A?

- A course about managing complexity
  - Mastering abstraction
  - Programming paradigms
- An introduction to programming
  - Full understanding of Python fundamentals
  - Combining multiple ideas in large projects
  - How computers interpret programming languages
- Different languages: Scheme & SQL
- A challenging course that will demand a lot of you

Look at how far you've come

We went from learning how to evaluate expressions like

```
add(add(6, mul(4, 6)), mul(3, 5))
```

and now you've built an interpreter for Scheme in Python!

in LESS than 8 weeks

What comes next?

CS 61B – Data Structures and Algorithms

CS 61C – Computer Architecture and Introduction to Hardware

CS 70 – Discrete Math and Probability

Data 100 – Principles of Data Science

CS 16X – Systems; Lower level courses that look at Security, Operating Systems, Programming Languages, Compilers, and more

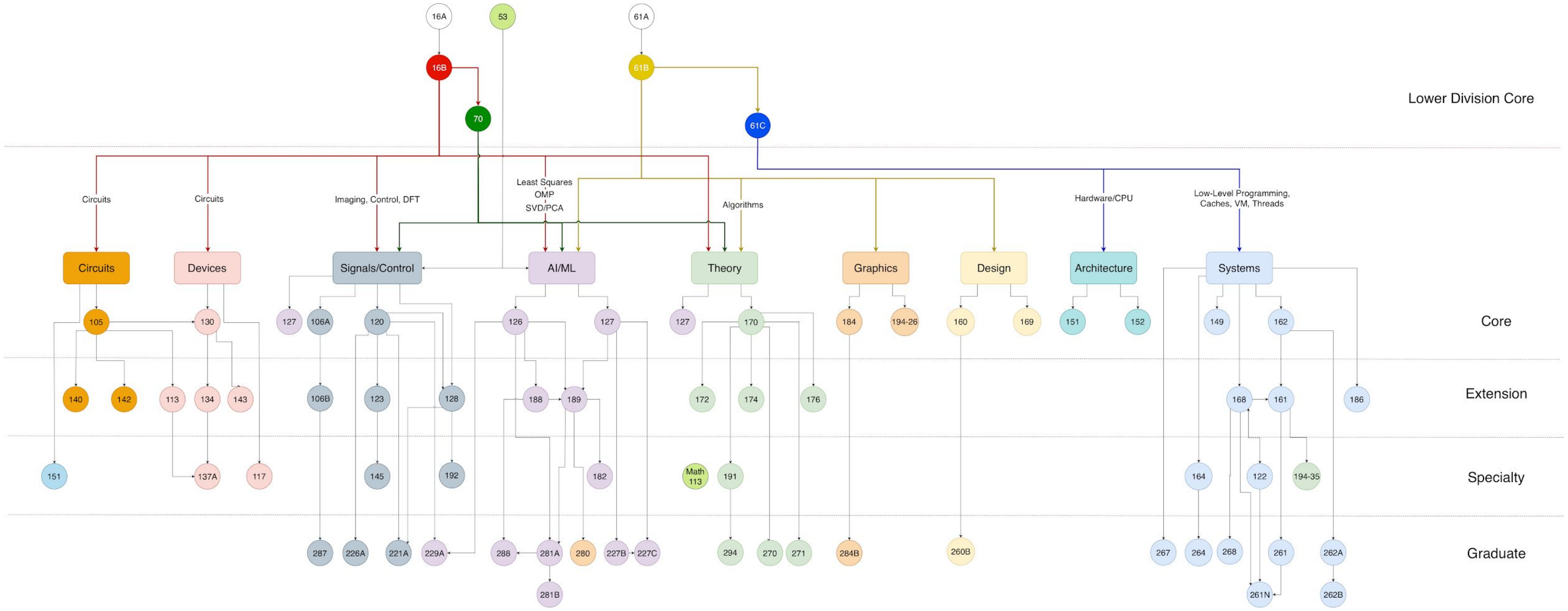
CS 17X – Theory

CS 18X – Various Applications; AI, ML, Graphics, neural networks, and other higher level ideas

CS 194 – Special Topics (these are really cool!)

Course Map

Check out https://hkn.eecs.berkeley.edu/courseguides for course guides!



Learn more Python + other languages!

We didn’t teach you everything about Python, there is a lot more!

Once you learn 1 language, other languages become easier to learn

Courses at Berkeley are taught in a wide range of languages, such as Python, C, Go, Java, Rust, etc

Practice

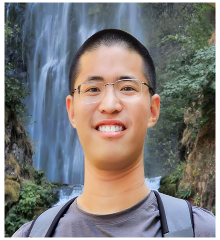
HackerRank + LeetCode are common sites students use to practice coding questions for Software Engineering interviews

Hackathons + Personal Projects are a good way to experiment

There’s a massive open source community in the world! Build something and put it out there

Most important, have fun!

Special thanks to the awesome TAs and Tutors of the course!



Thank you and Good Luck!!!