CSCI 134 Fall 2021:

Java Wrap-up & OOP Review

Dec 8, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
Announcements & Logistics

• **Lab 10 Selection Sort in Java**: due Wed/Thurs @ 10 pm

• **Final exam reminder**: Dec 18 @ 9:30 am

• Final exam will be cumulative: everything is fair game (including Java)
  • More weight on topics post-midterm topics
  • Will discuss more about this in Friday's wrap up lecture
  • Practice problems for final: will be released very soon

• Review session next week: **Wednesday, Dec 15, 7 - 8:30 pm**
  • Very informal, come ask questions

• **Course evals on Friday**: bring a laptop to class if possible
  • Please attend *your designated lecture section!*
  • Short wrap up (~30 min), end early for you to fill out evals in class
Last Time

• Discussed **loops** and **conditionals** in Java

• Python **for loops** are most similar to **for each loops** in Java

• A simple Java **for loop** explicitly requires starting condition, stopping condition, and steps in the header:

  
  ```java
  for i in range(10):
      print(i)
  ...
  for el in seq:
      print(el)
  ...
  ```

  ```java
  for (int i = 0; i < 10; i++) {
      System.out.println(i);
  }
  ...}
  ```

  ```java
  for (int i : myArray) {
      System.out.println(i);
  }
  ...}
  ```

• **for each** loop in Java

• Briefly discussed **methods** and **return** types in Java
Python vs Java: Check-in after Lab 10

• Curly braces, semicolons: what value do they add?

• Specifying data types at all times: how is it useful?
Python vs Java: Check-in after Lab 10

- **Curly braces, semicolons**: what value do they add?
  - Make the code more **maintainable** and **platform independent**!
  - White spaces, tabs, and line breaks are not stored consistently across computer architectures and operating systems
  - Converting a file from one system to another (say Windows to Mac) can change the white space
  - This would break a Python script; Java program might become unreadable but will still run!

- Specifying **data types** at all times: how is it useful?
  - In larger coding projects, not knowing the type of variables can make code harder to follow
  - This is why Python docstrings are so important!
Today

- Review **classes** and **objects** in more detail
  - A **class** vs an **instance** of the class (or an object of the class)
  - **Attributes** (called instance variables in Java) and slots
  - **Accessor** and **mutator** methods: getters, setters, and properties
  - **Scope**: public, private and protected (or _ and __ in Python)
- Note that the aforementioned topics are **language independent**!
  - We will look at them in both languages but the focus will be on reviewing the concepts and not the syntax!
Programming Language Features

- **Basic features:**
  - Data Types
  - Reading user input
  - Loops
  - Conditionals

- **Advanced topics:**
  - Classes
  - Interfaces
  - Collections
  - Graphical User Interface Programming
Classes and Objects

- Classes are blueprints for **objects** (or **instances**)
  - Collections of data (**variables and attributes**) and **methods** that act on those data
- We did not talk about Python **classes** until Lecture 21
  - Easy to ignore/forego this topic for simple examples in Python
- In Java, all code is defined within a class
  - We have to come to terms with **classes** and **methods** from Day 1
  - No such thing as a classless **module** or **function** in Java
- Support for classes are a feature of all **OOP languages**
  - Python and Java are both OOP languages
Classes and Objects

- Python is more consistent than Java with its OOP model and data types:
  - In Python, everything is an object: including ints, strings, functions, etc
  - Python types are implicit, can be queried using type
  - In Java, there are primitive types which are not objects (ints, doubles, booleans, chars etc) and "Object" versions of these types (Integer, Double, String, etc.)
    - Java forces explicit type declaration from the get go
- Why would we ever want to define our own classes?
  - Create our own “data types”
  - A way to bundle (or encapsulate) related data and methods for interacting with that data in an application-specific manner
Review: Object-Oriented Programming

Four major principles of OOP programming:

- **Abstraction** (data and procedural)
  - The main purpose of abstraction is hiding the unnecessary details from the users

- **Inheritance**
  - The ability for one object to take on the states, behaviors, and functionality of another object

- **Encapsulation**
  - The bundling of data, along with the methods that operate on that data, into a single unit

- **Polymorphism**
  - Using a single type entity (method, operator or object) to represent different types in different scenarios (e.g., operator overloading)
Methods vs Functions

Methods (Python and Java)

• Always defined **within a class**
• Are called using **dot notation** on a specific **instance** of the containing class
• A method is implicitly passed a reference to the object on which it is invoked (**self** in Python, **this** in Java)
• A method can optionally manipulate **parameters**
• A method may or may not **return** a value
• A method can operate on the **attributes/instance variables** that are defined within the containing class

Functions (Python only)

• Stand-alone logical blocks of code that are **defined outside of a class**
• Once defined, a function can be called from anywhere in the program (by importing if in a separate module)
• A function definition specifies **parameters** (input that is passed to the function when it is called). If parameters are passed, they need to be passed **explicitly**
• A function may perform an action (e.g. print or modify), and/or return a value (or implicitly return None)
self Parameter Review

• In **Python**, method **definitions** have **self explicitly** defined as the first parameter (and we use this variable inside the method body)

• But **we don’t pass the self parameter explicitly** when we invoke the methods!

• This is because whenever we call a method on an object, the object itself is **implicitly** passed as the first parameter

• Methods are like **object-specific functions** and this lets us access the object’s attributes via the methods directly
def plainFunction():
    print("I am a classless function!")

class TestClass:
    def sayHi(self, name):
        return "Hello " + name

if __name__ == "__main__":
    # create an instance of the TestClass class
    test = TestClass()

    # call sayHi() method on test
    print(test.sayHi("CS134"))

    # call plainFunction, which is not part of class
    plainFunction()
```java
public class TestClass {
    public String sayHi(String name) {
        return "Hello " + name;
    }
}

public static void main (String args[]) {
    // create an instance TestClass
    TestClass test = new TestClass();

    // invoke the method sayHi
    System.out.println(test.sayHi("CS134"));
}
```
Data Attributes or Instance Variables

• Classes keep track of relevant state in instance variables (Java) or attributes (Python)

• In Python, attributes should be stored in __slots__
  • Attributes in __slots__ (list of strings) are explicitly specified
  • Prevents attributes from being added dynamically

• In Java, instance variables are typically defined at the top of the class before all methods
  • Instance variables are accessible to all methods of the class

• RULE OF THUMB: Make all attributes private (or protected)
  • In Python, this means using "_" or "__" and in Java we say “private”
  • Only accessed via accessor (getter) and mutator (setter) methods
## Scope Review

<table>
<thead>
<tr>
<th>Private</th>
<th>Protected</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Python:</strong> Double leading underscore (__) in name of variable or method</td>
<td><strong>Python:</strong> Single leading underscore (_) in name of variable or method</td>
<td><strong>Python:</strong> No leading underscore in name of variable or method</td>
</tr>
<tr>
<td><strong>Java:</strong> Use the keyword <code>private</code></td>
<td><strong>Java:</strong> Use the keyword <code>protected</code></td>
<td><strong>Java:</strong> Use the keyword <code>public</code></td>
</tr>
</tbody>
</table>
| Private methods and variables/attributes are *not accessible from outside* of the containing class | Protected methods and variables/attributes should only be accessed by subclasses | Public methods and variables/attributes can be *freely used outside of the class*

These access rules are actually enforced in Java; are more of a convention in Python
Methods and Data Abstraction

• Users are given access to data attributes only through methods in OOP.

• Manipulating attributes/instance variables should only be done via:
  
  • **accessor (getter) methods:** provide “read-only” access to the class attributes/instance variables (return value).
  
  • **mutator (setter) methods:** let us modify the values of class attributes/instance variables (do not return).

• In Python these methods are also known as procedural attributes.

• Using getters and setters enforces **data abstraction**.
  
  • Methods provide a **public interface** to attribute values.
  
  • Attributes remain part of the **private implementation**.
Specifying Python Annotations with @

- **Annotations @**: Python provides a rich collection of *syntactic notes* that can change how code is interpreted, called *annotations*, typically prefixed with the at-sign (@).

- **@property annotation**: Used for *getter* methods.
  - Lets us treat a procedural attribute (a getter method) as a data attribute

- **@var.setter annotation**: Used for *setter* methods.
  - Lets us treat a procedural attribute (a setter method) as a data attribute. Must have corresponding @property method defined.

- We are still calling methods, but using Python annotations, we can **omit the parentheses**!

**Note**: Java does not use annotations!
# originally in lec 27

class LinkedList:
    #"""Implements our own recursive list data structure"""
    __slots__ = ['_value', '_rest']

    def __init__(self, v=None, r=None):
        # call setters for _value and _rest
        self.setValue = v
        self.setRest = r

    @property
    def getRest(self):
        return self._rest

    @property
    def getValue(self):
        return self._value

    @getRest.setter
    def setRest(self, val):
        self._rest = val

    @getValue.setter
    def setValue(self, val):
        self._value = val

Private attributes

Calling **setter methods**.
Notice lack of ()

public getter method for _rest

public getter method for _value

public setter method for _rest

public setter method for _value
```python
# originally in lec 37

class LinkedList:
    """Implements our own recursive list data structure"""
    __slots__ = ['_value', '_rest']

def __init__(self, value=None, rest=None):
    # call setters for value and rest
    self._value = value
    self._rest = rest

@property
def rest(self):
    return self._rest

@property
def value(self):
    return self._value

@rest.setter
def rest(self, val):
    self._rest = val

@value.setter
def value(self, val):
    self._value = val
```

Private attributes

Calling setter methods. Notice lack of ()

Same code using a more typical naming convention!

public getter method for _rest

public getter method for _value

public setter method for _rest

public setter method for _value
```java
public class LinkedList {
    private String value;
    private LinkedList rest;

    public LinkedList(String val) {
        this.setValue(val);
        this.setRest(null);
    }

    public LinkedList(String val, LinkedList other) {
        this.setValue(val);
        this.setRest(other);
    }

    public String getValue() {
        return this.value;
    }

    public LinkedList getRest() {
        return this.rest;
    }

    public void setRest(LinkedList r) {
        this.rest = r;
    }

    public void setValue(String v) {
        this.value = v;
    }
}
```

- **Private** instance variables
- Notice that `rest` is of type `LinkedList`. Recursion!
- Constructors, ignore for now
- Calling setter methods
- **Public** getter method for value
- **Public** getter method for rest
- **Public** setter method for rest
- **Public** setter method for value
public class LinkedList {
    private String value;
    private LinkedList rest;

    public LinkedList(String val) {}  // Constructors, ignore for now!

    public LinkedList(String val, LinkedList other) {
        setValue(val);
        setRest(other);
    }

    public String getValue() {
        return value;
    }

    public LinkedList getRest() {
        return rest;
    }

    public void setRest(LinkedList r) {
        rest = r;
    }

    public void setValue(String v) {
        value = v;
    }
}

Same code without an explicit use of this

Calling setter methods

public getter method for value

public getter method for rest

public setter method for rest

public setter method for value
Special Methods & Operator Overloading

• Classes in Python and Java define several “special” methods
  • **Python**: `__init__`, `__str__`, `__repr__`, `__eq__`
  • **Java**: `constructor(s)`, `toString()`, `equals()`

• Python has many more due to **operator overloading**
  • Operator overloading means we redefine common operations (like addition + or using list notation `[ ]` for access) for our data type
    • `__add__`, `__getitem__`, `__setitem__`, `__contains__`
    • Many more!

• Java does not support operator overloading
  • But it does support **method overloading** (same method, different parameters)
Initializing an Object

• When creating a new instance of a class in Python or Java, we have to initialize the values of the attributes/instance variables

  • **Python:** `__init__` method
  
  • **Java:** Constructor(s)

• These special methods are **automatically called** when you create an instance of the class

  • **Python:** `board = BoggleBoard()`
  
  • **Java:** `BoggleBoard board = new BoggleBoard()` (notice the use of `new`)

• Let’s look at how this works for our **LinkedList**
Python

class LinkedList:
    """Implements our own recursive list data structure"""
    __slots__ = ['_value', '_rest']

    def __init__(self, value=None, rest=None):
        # call setters for value and rest
        self.value = value
        self.rest = rest

Java

public class LinkedList {
    private String value;
    private LinkedList rest;

    public LinkedList(String val) {
        setValue(val);
        rest = null;
    }

    public LinkedList(String val, LinkedList other) {
        setValue(val);
        setRest(other);
    }

Java does not allow us to specify “default” values for parameters, so we need to define multiple constructors with the same name (method overloading).

Constructors have no return type and are the same name as the class.
String Representation of an Object

- It is often convenient to be able to print a string “version” of an instance of a class
  - Very helpful when debugging
- Python and Java both provide special methods for this
  - Python: __str__ and __repr__
  - Java: toString()
- For __str__ and toString(), we can choose how the objects of the class are printed
- For __repr__ (Python only, since Java is not interactive), we want to generate a string that would allow us to recreate the object
Python

def __str__(self):
    return "value : " + str(self.value) + ", next : (" + str(self.rest) + ")"

def __repr__(self):
    return "LinkedList('{}',{})".format(self.value, self.rest)

>>> from linkedlist import *
>>> myList = LinkedList("a")
>>> myList
LinkedList('a',None)
>>> print(myList)
value : a, next : (None)

Java

public String toString() {
    return "value : " + getValue() + ", next : (" + getRest() + ")";
}

LinkedList myList, myList2;
myList = new LinkedList("a");
System.out.println("myList: " + myList);

bash-3.2$ java LinkedList
myList: value : a, next : (null)
Comparing Objects

• Often convenient to compare two instances of a class
• We have to decide if we want to compare their values or identities
• Comparing values: determining if the data contained in two separate instances of a class is the same (e.g., two lists that contain the same values)
  • Python: `==` operator (`__eq__` special method, operator overloading)
  • Java: `equals()` method
• Comparing identities: determining if two instances are actually the same? (Do they reside in the same place in memory?)
  • Python: `is` operator (cannot be overloaded!)
  • Java: `==` operator
Python

def __eq__(self, other):
    # If both lists are empty
    if self.rest is None and other.rest is None:
        return True
    # If both lists are not empty, then data of current elements must match, # and same should be recursively true for rest of the elements
    elif self.rest is not None and other.rest is not None:
        return self.value == other.value and self.rest == other.rest
    # If one of the lists is empty and other is not
    else:
        return False

Java

public boolean equals(LinkedList other) {
    if (getRest() == null && other.getRest() == null) {
        return true;
    } else if (getRest() != null && other.getRest() != null) {
        boolean val = getValue().equals(other.getValue());
        boolean r = getRest().equals(other.getRest());
        return val && r;
    } else {
        return false;
    }
}
Other Useful Methods

• **Testing membership** - we often want to know if a specific item or value exists in our data structure
  
  • **Python:** in operator (`__contains__` special method)
  
  • **Java:** contains() method

• **Computing length** - we often want to know the length or size of a data structure
  
  • **Python:** len function (`__len__` special method)
  
  • **Java:** length() method

• For our LinkedList implementations, all of these operations/methods will be recursive
Other Useful Methods

Python

def __len__(self):
    # base case: i'm last item
    if self.rest is None:
        return 1
    else:
        return 1 + len(self.rest)

def __contains__(self, val):
    if self.value == val:
        return True
    elif self.rest is None:
        return False
    else:
        return val in self.rest

Java

class Node {
    int value;
    Node rest;
}

public int length() {
    if (getRest() == null) {
        return 1;
    } else {
        return 1 + getRest().length();
    }
}

public boolean contains(String search) {
    if (getValue().equals(search)) {
        return true;
    } else if (getRest() == null) {
        return false;
    } else {
        return getRest().contains(search);
    }
}