Abstract Data Types, Interfaces, Generics
Recall:
- Abstract Data Types (ADTs)
  - A collection of data and operations on that data
- Data Structure
  - How we choose to implement an ADT

Abstract Classes
Generics
Multiple Inheritance and Interfaces
Comparing Stack and Queue

class StackOfStrings:
    def __init__(self):
        self.first = None

    def push(self, s):
        node = Node()
        node.item = s;
        if self.first == None:
            self.first = node
        else:
            node.next = self.first
            self.first = node

    def pop(self):
        if self.first == None:
            throw("Stack is empty!")
        result = self.first.item
        self.first = self.first.next
        return result

    def toString(self):
        result = ""
        current = self.first
        while current != None:
            result += current.item
            result += " 
            current = current.next
        return result

    def isEmpty(self):
        return self.first == None

class QueueOfStrings:
    def __init__(self):
        self.first = None
        self.last = None

    def enqueue(self, s):
        node = Node()
        node.item = s
        node.next = None
        if self.last != None:
            self.last.next = node
            self.last = node
        if self.first == None:
            self.first = node

    def dequeue(self):
        if self.first == None:
            throw("Queue is empty!")
        result = self.first.item
        self.first = self.first.next
        if self.first == None:
            self.last = None
        return result

    def toString(self):
        result = ""
        current = self.first
        while current != None:
            result += current.item
            result += " 
            current = current.next
        return result

    def isEmpty(self):
        return self.first == None

Repeated code is evil!
from abc import ABC, abstractmethod

class AbstractDataType(ABC):
    def __init__(self):
        super().__init__()
        self.first = None
        self.last = None

    @abstractmethod
    def add(self, item):
        pass

    @abstractmethod
    def remove(self):
        pass

    def toString(self):
        result = ""
        current = self.first
        while current != None:
            result += current.item
            result += " 
            current = current.next
        return result

    def isEmpty(self):
        return self.first == None
Comparing New Stack and Queue

class StackOfStrings(AbstractDataType):
    def __init__(self):
        super().__init__()
    def add(self, item):
        self.push(item)
    def remove(self):
        return self.pop()
    def push(self, s):
        node = Node()
        node.item = s;
        if self.first == None:
            self.first = node
        else:
            node.next = self.first
            self.first = node
    def pop(self):
        if self.first == None:
            throw("Stack is empty!")
        result = self.first.item
        self.first = self.first.next
        return result

class QueueOfStrings(AbstractDataType):
    def __init__(self):
        super().__init__()
    def add(self, item):
        self.enqueue(item)
    def remove(self):
        return self.dequeue()
    def enqueue(self, s):
        node = Node()
        node.item = s
        node.next = None
        if self.last != None:
            self.last.next = node
            self.last = node
        if self.first == None:
            self.first = node
    def dequeue(self):
        if self.first == None:
            throw("Queue is empty!")
        result = self.first.item
        self.first = self.first.next
        if self.first == None:
            self.last = None
        return result
Generics

- Our data structures are still tied to whatever type of item they hold:
  - Stacks that hold strings
  - Stacks that hold floating point numbers
  - Stacks that hold Positions

```python
class Node:
    # Strings
    def __init__(self):
        self.item = ""
        self.next = None

class Node:
    # Floats
    def __init__(self):
        self.item = 0.0
        self.next = None

class Node:
    # Positions
    def __init__(self):
        self.item = None
        self.next = None
```
Generics

- In Python, this is not as big an issue
  - Python uses “dynamic typing”
    - The type of data is determined at runtime
- In other languages, this is an issue
  - The data type of input parameters and return values must be specified ahead of time
    - So, when you push an item on a stack, or pop an item from a stack, Java or C++ will expect it to be of a certain type
  - We can use a generic data type in these languages
  - Give it a name, like a variable, and then use that in place of the data type name
public class StackOfStrings
{
    private class Node
    {
        private String item;
        private Node next;
    }
    private Node first = null;

    public boolean isEmpty()
    {
        return (first == null);
    }

    public void push(String s)
    {
        Node node = new Node();
        node.item = s;
        node.next = first;
        first = node;
    }

    public String pop()
    {
        if (first == null)
            throw new RuntimeException("Stack is empty!");
        String result = first.item;
        first = first.next;
        return result;
    }
    ...
}

public class StackOfDoubles
{
    private class Node
    {
        private double item;
        private Node next;
    }
    private Node first = null;

    public boolean isEmpty()
    {
        return (first == null);
    }

    public void push(double s)
    {
        Node node = new Node();
        node.item = s;
        node.next = first;
        first = node;
    }

    public double pop()
    {
        if (first == null)
            throw new RuntimeException("Stack is empty!");
        double result = first.item;
        first = first.next;
        return result;
    }
    ...
}
public class Stack<E> {
    private class Node {
        private E item;
        private Node next;
    }
    private Node first = null;

    public void push(E s) {
        Node node = new Node();
        node.item = s;
        node.next = first;
        first = node;
    }

    ...
public class Stack<Item> {
    private class Node {
        private Item item;
        private Node next;
    }
    private Node first = null;
    public void push(Item s) {
        Node node = new Node();
        node.item = s;
        node.next = first;
        first = node;
    }
    ...

E is for Element, but nothing special about E. Any Java identifier can be used such as Item. It just needs to be used consistently in the declaration of the generic class.
C. Multiple Inheritance

Example Website: https://www.python-course.eu/python3_multiple_inheritance.php
class Clock

__init__(int hours, int minutes, int seconds)
    # Construct a new Clock

set_Clock(int hours, int minutes, int seconds)
    # Set the clock to a new time

string __str__()
    # Return a string representation of time

tick()
    # Advance the clock by one second

class Calendar

__init__(int day, int month, int year)
    # Construct a new Calendar with day, month and year

boolean leapyear(int year)
    # Determine if a year is a leap year

set_Calendar(int day, int month, int year)
    # Set the calendar to a new date

string __str__()
    # Return a string representation of date

advance()
    # Advance the calendar by one day

class ClockCalendar(Clock, Calendar)

__init__(int hours, int minutes, int seconds, int day, int month, int year)
    # Construct a new ClockCalendar

string __str__()
    # Return a string representation of date and time

tick()
    # Advance the ClockCalendar by one second
ClockCalendar UML

Clock
- hours
- minutes
- seconds
+ __init__()
+ set_Clock()
+ __str__()
+ tick()

Calendar
- days
- months
- years
+ __init__()
+ leap_year()
+ set_Calendar()
+ __str__()
+ advance()

ClockCalendar
+ __init__()
+ tick()
+ __str__()
class Clock(object):
    def __init__(self, hours, minutes, seconds):
        self.set_Clock(hours, minutes, seconds)

    def set_Clock(self, hours, minutes, seconds):
        if type(hours) == int and 0 <= hours and hours < 24:
            self._hours = hours
        else:
            raise TypeError("Hours have to be integers between 0 and 23!")
        if type(minutes) == int and 0 <= minutes and minutes < 60:
            self.__minutes = minutes
        else:
            raise TypeError("Minutes have to be integers between 0 and 59!")
        if type(seconds) == int and 0 <= seconds and seconds < 60:
            self.__seconds = seconds
        else:
            raise TypeError("Seconds have to be integers between 0 and 59!")

    def __str__(self):
        return "{0:02d}:{1:02d}:{2:02d}".format(self._hours, self.__minutes, self.__seconds)

    def tick(self):
        if self.__seconds == 59:
            self.__seconds = 0
        if self.__minutes == 59:
            self.__minutes = 0
        if self._hours == 23:
            self._hours = 0
        else:
            self._hours += 1
        else:
            self.__minutes += 1
        else:
            self.__seconds += 1
class Calendar(object):
    months = (31,28,31,30,31,30,31,31,30,31,30,31)
    date_style = "American"

    @staticmethod
    def leapyear(year):
        if not year % 4 == 0:
            return False
        elif not year % 100 == 0:
            return True
        elif not year % 400 == 0:
            return False
        else:
            return True

    def __init__(self, d, m, y):
        self.set_Calendar(d, m, y)

    def set_Calendar(self, d, m, y):
        if type(d) == int and type(m) == int and type(y) == int:
            self.__days = d
            self.__months = m
            self.__years = y
        else:
            raise TypeError("d, m, y have to be integers!")

    def __str__(self):
        if Calendar.date_style == "British":
            return "{0:02d}/{1:02d}/{2:4d}".format(self.__days, self.__months, self.__years)
        else:
            return "{0:02d}/{1:02d}/{2:4d}".format(self.__months, self.__days, self.__years)

    def advance(self):
        max_days = Calendar.months[self.__months - 1]
        if self.__months == 2 and Calendar.leapyear(self.__years):
            max_days += 1
        if self.__days == max_days:
            self.__days = 1
            if self.__months == 12:
                self.__months = 1
                self.__years += 1
            else:
                self.__months += 1
        else:
            self.__days += 1
class CalendarClock(Clock, Calendar):
    def __init__(self, day, month, year, hour, minute, second):
        Clock.__init__(self, hour, minute, second)
        Calendar.__init__(self, day, month, year)

    def tick(self):
        previous_hour = self._hours
        Clock.tick(self)
        if (self._hours < previous_hour):
            self.advance()

    def __str__(self):
        return Calendar.__str__(self) + ', ' + Clock.__str__(self)
C. Multiple Inheritance: The Diamond Problem
class A:
    def __init__(self):
        print("A.__init__")

class B(A):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()

class D(B,C):
    def __init__(self):
        print("D.__init__")
        super().__init__()

if __name__ == "__main__":
    print("Instantiating a D object:")
    d = D()
    print("Instantiating a C object:")
    c = C()
    print("Instantiating a B object:")
    b = B()
    print("Instantiating an A object:")
    a = A()
The Diamond Problem, Larger Context
Java Approach

- Allow single inheritance only
- Any other characteristics come from “Interfaces”
  - Totally abstract
  - Include method signatures (but not code)
  - No attributes included
// Interface for a shape that has a border that can be a different color and has a variable pen thickness.
public interface Bordered {
    Color getBorderColor();
    void setBorderColor(Color color);
    double getBorderThickness();
    void setBorderThickness(double thickness);
}

A class adds implements Bordered to the class declaration.
The class must then implement the four methods in interface Bordered.

public class CircleBorder extends Circle implements Bordered

public class RectangleBorder extends Rectangle implements Bordered
Summary

- **Recall:**
  - Abstract Data Types (ADTs)
    - A collection of data and operations on that data
  - Data Structure
    - How we choose to implement an ADT
- Abstract Classes
- Generics
- Multiple Inheritance and Interfaces
Download the Hierarchy.py file from the website and run it. You may want to draw out the abstraction hierarchy represented by the code.

Try making the class Pet a subclass of LivingThing and re-run the code

Annotate the code in a header comment to explain the output that results from running the code both ways.

Submit your solution to the Moodle Activity 3 dropbox for today. You get 1 point for turning something in, another for turning in something with a reasonable explanation.