Lecture 9: More OOP and Stacks and Queues

CS5001 / CS5003:
Intensive Foundations
of Computer Science

PDF of this presentation

Lecture 9: Grade report

- You should have received an email from me about a website I put together to show how you are doing in the course:
  - https://stanford.edu/~cgregg/cgi-bin/cs5001/
- I apologize for not having the grades updated before this.
- The handins site has some issues, so I decided to put the other site together. I'll update the grades as we finish them.
- The projected guess at your grade is just that -- a prediction. If you keep doing about as well as you have been, this is a good prediction of your likely final grade range (plus or minus grades are not predicted)
- In the email, you got a username and password -- I want to talk about how I created the passwords, and what that means for logging into the website -- this is actually a fascinating use of *cryptography*.

Lecture 9: Grade report

- I generated your passwords with the following very simple python program:

```
students = [
    ('benaghil', 'benaghil.h@husky.neu.edu'), ('bingbing', 'bi.yi@husky.neu.edu'),
    ('icabrera120', 'cabrera.i@husky.neu.edu'), ('datietuo', 'tiezhouduan@gmail.com'),
    ('bekezie', 'ekezie.b@husky.neu.edu'), ('jamesjfan', 'fan.ja@husky.neu.edu'),
    ('jiaofeng1129', 'feng.jiao@husky.neu.edu'), ('kimmyfeng', 'feng.le@husky.neu.edu'),
    ('elam', 'lam.es@husky.neu.edu'), ('lth110797', 'li.tianhu@husky.neu.edu'),
    ('wayneleo818', 'liu.jiawe@husky.neu.edu'), ('yimanliu', 'liu.yima@husky.neu.edu'),
    ('charlenelyu', 'lyu.yangh@husky.neu.edu'), ('mashaido', 'mashaido.r@husky.neu.edu'),
    ('j2massie', 'massie.j@husky.neu.edu'), ('ymichael', 'michael.y@husky.neu.edu'),
    ('draytonmoody', 'moody.d@husky.neu.edu'), ('whatcat', 'qiu.ziha@husky.neu.edu'),
    ('kamilahweeks', 'weeks.k@husky.neu.edu'), ('alexyang', 'yang.ale@husky.neu.edu'),
    ('kristinayin', 'yin.kr@husky.neu.edu'), ('cgregg', 'cgregg@northeastern.edu'),
    ]

animals = ['sheep', 'turkey', 'dog', 'duck', 'frog',
           'horse', 'coyote', 'rooster', 'pig', 'cow', 'bird', 'cat']

import random
import hashlib
def main():
    for name, email in students:
        animals_temp = animals[::]
        random.shuffle(animals_temp)
        pw = " ".join(animals_temp[:4])
        pw_hash = hashlib.sha256((name+pw).encode()).hexdigest()

        print(f'{name},{email},"{pw}",{pw_hash}')
if __name__ == '__main__':
    main()
```

- You should recognize everything in the program except, perhaps, **`random.shuffle`**, and **`hashlib.sha256`**

Lecture 9: Grade report

- I generated your passwords with the following very simple python program:

```
students = [
    ('benaghil', 'benaghil.h@husky.neu.edu'), ('bingbing', 'bi.yi@husky.neu.edu'),
  # ... removed for brevity
    ('kristinayin', 'yin.kr@husky.neu.edu'), ('cgregg', 'cgregg@northeastern.edu'),
    ]

animals = ['sheep', 'turkey', 'dog', 'duck', 'frog',
           'horse', 'coyote', 'rooster', 'pig', 'cow', 'bird', 'cat']

import random
import hashlib
def main():
    for name, email in students:
        animals_temp = animals[::]
        random.shuffle(animals_temp)
        pw = " ".join(animals_temp[:4])
        pw_hash = hashlib.sha256((name+pw).encode()).hexdigest()

        print(f'{name},{email},"{pw}",{pw_hash}')
if __name__ == '__main__':
    main()
```

Note that the hash is created from your username *and* password: this is called "salting" and protects against *rainbow tables*

- The **`random.shuffle`** function re-orders a list to a random order
- The **`hashlib.sha256`** function creates a *cryptographic hash* of a string. A cryptographic hash is a very long base-16 number that is very, very unlikely to be generated by any other string using the same function.

Lecture 9: Grade report

- Here is the output of the program (with a different set of random passwords):

```
benaghil,benaghil.h@husky.neu.edu,"bird duck cow rooster",ef063785433b8b46db95a116750a0887207230507b3819ab2d61a666022d0bd0
bingbing,bi.yi@husky.neu.edu,"bird pig frog turkey",142af207a3ab506982b4f7c20d24395377df1b62e055b33b491c4c9b805b5b93
icabrera120,cabrera.i@husky.neu.edu,"rooster duck dog coyote",8f4814c1dd4dc99a6c69b1a5adedb82c85f1bed8b789589fd5aae2281db92021
datietuo,tiezhouduan@gmail.com,"frog duck bird cat",b2d2f802ddbcf9898654ce3ce26a6637b0c443eddc3ba4fd27a33e7e1b5a17ad
bekezie,ekezie.b@husky.neu.edu,"cat turkey sheep cow",ad64324f4200aa456a284cccd059b33393ddd3d2275d553894e00c9684a8f01f
jamesjfan,fan.ja@husky.neu.edu,"pig coyote bird rooster",e3ad46a2726c840c02b7915ffa33fd7cc0fe8dc46944843de3744ea9f129407a
jiaofeng1129,feng.jiao@husky.neu.edu,"frog dog cat cow",306c6f96559dc29149379b16bc75e4a5cd43fb5f3511fba4c30c0a985939fed7
kimmyfeng,feng.le@husky.neu.edu,"rooster bird coyote duck",2b30c61f04a61d6c79d82aa409f22826de3b9441a6839aaa012cf19c67242189
elam,lam.es@husky.neu.edu,"bird sheep duck dog",0eefcbb4e2c77658636fc4656e52ac89ac662479237733 93b05d5a8376daf3b9
lth110797,li.tianhu@husky.neu.edu,"rooster duck coyote sheep",7cde8cab235573517411720e561d69c7b3122d911c41a4392f33c74f534522cf
wayneleo818,liu.jiawe@husky.neu.edu,"coyote sheep duck horse",e11c18d3527e4bd1c9cd16705f51ee986f63323d9948a41bdbbecf7fa4aa3be1
yimanliu,liu.yima@husky.neu.edu,"frog cat rooster pig",83c5cf4ea66d5d858a40b95b8f9e078bfcf192046e78e6b950441e46653acdbc
charlenelyu,lyu.yangh@husky.neu.edu,"rooster turkey duck frog",d5726685638100652819cee7622e27d9d7baacd63514c2d03e5d9352752df470
mashaido,mashaido.r@husky.neu.edu,"turkey sheep duck bird",d6d56699142f810ad5e2a1521772288f94647bfb8cddc7a0eedea51c5e23b206
j2massie,massie.j@husky.neu.edu,"horse cow cat turkey",1a82a30794b9fd39f6bee43b0c515807fc239161eb3fbe7b60b7bb303b98a8ee
ymichael,michael.y@husky.neu.edu,"cow horse duck cat",f206511f58eab675c309a1c1f9939e38675f5e2ccf570c7697b3812e0b8edc96
draytonmoody,moody.d@husky.neu.edu,"turkey sheep cow frog",cc20d64378a9cf17683cf342732e07711771567b8a542dc6dc0641320b3045a7
whatcat,qiu.ziha@husky.neu.edu,"coyote horse sheep bird",2c1af87aed6dc02da751bbbd106e754f2684b4e17664518996aa108909a9db41
kamilahweeks,weeks.k@husky.neu.edu,"sheep dog horse cow",621e3c5858b1692fb989f9578ee5d51e1deceae578667d90206fdcedd4d3da9e
alexyang,yang.ale@husky.neu.edu,"pig duck frog dog",469e47e64f25b5a1b784a9a775ad2825fd28b258e36436cba89891add3e6964c
kristinayin,yin.kr@husky.neu.edu,"duck turkey dog bird",bf46cfaae3b598557c49f05086e1a257de9a9cffd6d4b895df0ba763d7820ab0
cgregg,cgregg@northeastern.edu,"frog bird rooster sheep",49b7d3afd676588614ca55ba49a99ec0bfb9a036422151740c33937752638b3b
```

- How many different passwords can we generate with twelve different animals? 12 choose 4, or 12 * 11 * 10 * 9 = 11,880 passwords
  - It is unlikely that two of you have the same password
- Here is one of the cryptographic hashes:
  - `49b7d3afd676588614ca55ba49a99ec0bfb9a036422151740c33937752638b3b`
- I store the usernames and hashes on the server, but not the passwords!

Lecture 9: Grade report

- Once I delete the original password file, it is impossible for me to get your password back! If a website sends you your password, then you know they are storing it, which means they have poor security (of course, I sent you your original passwords and haven't given you a way to change them...)

- How can we use this to authenticate you?
  1. In your browser, some Javascript generates the hash based on your password (the password never gets sent over the internet)
  2. Your username and hash are sent to the server
  3. The server also hashes your username+pw, just like in your browser
  4. If your username and the hash match up in the file above, you're authenticated!
- Does this have a flaw? Yes!
  - If someone gets a hold of the hashes and usernames, they can (with a little knowledge of Javascript) spoof your identity and see your grades.
  - (the implementation I threw together also has some other flaws, too...)
- The good news: if someone steals your hash, they might be able to log in, but they can almost certainly not recover your actual password without a lot of work.

Lecture 9: Last week's lab: the CalendarEvent class

The CalendarEvent class demonstrated how we can create a robust type in Python that we can use as an *object* in our programs.

```
class CalendarEvent:
    """
    Holds a calendar event
    """
    def __init__(self, start_date=None, duration=1, note='', reminder=False):
        if start_date:
            self.start_date = start_date
        else:
            # create a date to the nearest 1 hour ahead
            now = datetime.now()
            now = now.replace(minute=0, second=0, microsecond=0)
            now = now + timedelta(hours=1)

            self.start_date = now
        self.duration = duration  # in hours
        self.note = note
        self.reminder = reminder
```

The __init__ method is the first method called when we instantiate an object, e.g.,

```
new_event = CalendarEvent()
```

The __init__ method does the set up for the object, initializing the instance attributes, etc.

Lecture 9: Last week's lab: the CalendarEvent class

The __str__ function allows you to use print() with your class:

```
def __str__(self):
        """
        Returns a string representing the event
        :return: a string
        """
        return (
                f"From: {CalendarEvent.human_readable_date(self.start_date)}\n" +
                f"To:   "
                f"{CalendarEvent.human_readable_date(self.end_date())}\n" +
                f"Duration: {self.duration} hr\n" +
                f"Note: {self.note}\n" +
                f"Reminder: {self.reminder}\n"
        )
```

The __eq__ method allows you to use == between to instances:

```
    def __eq__(self, other):
        """
        Returns True if the date and duration are the same
        :return: True or False if the date and duration are the same
        """
        return self.start_date == other.start_date and self.duration == other.duratio
```

We decided that two CalendarEvents are equal if the date and the duration are the same.

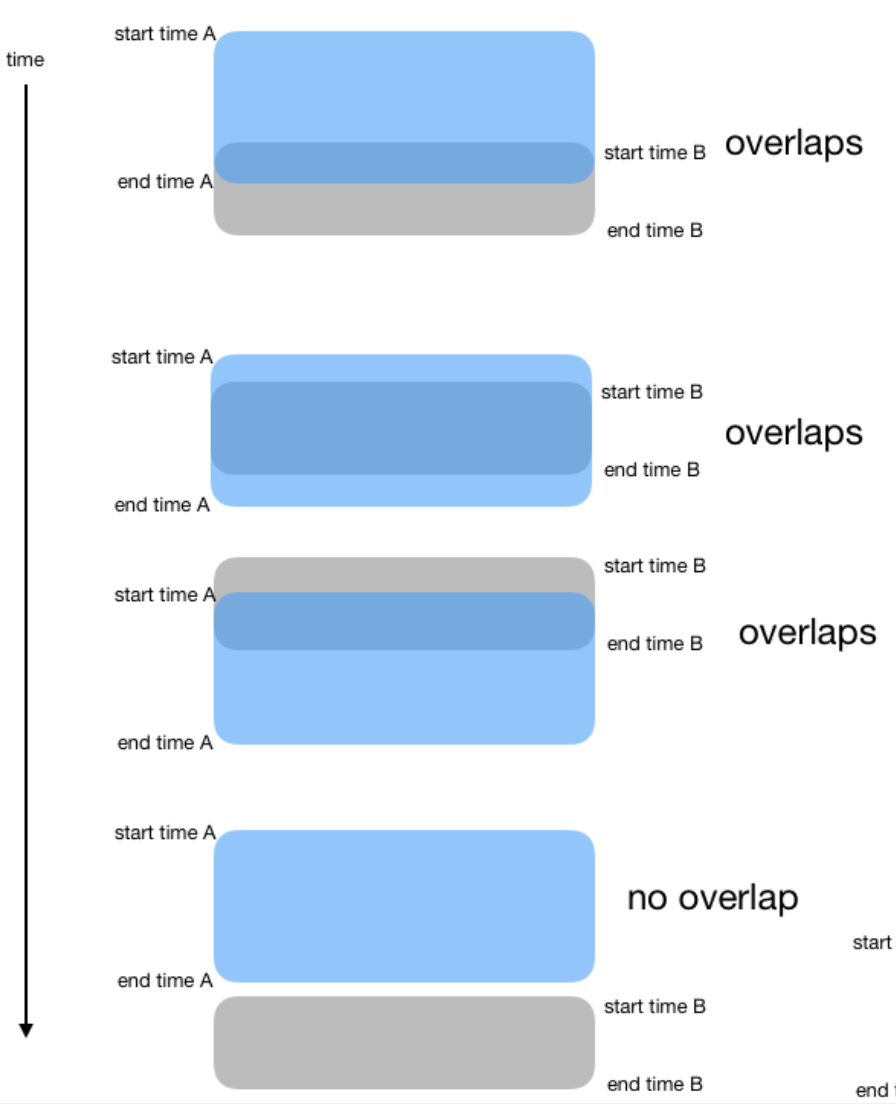
Lecture 9: Last week's lab: the CalendarEvent class

I had you create an overlaps() method, and said: *Two events are considered overlapping if the start date of the first is before the end date of the second, and if the start date of the second is before the end date of the first.*

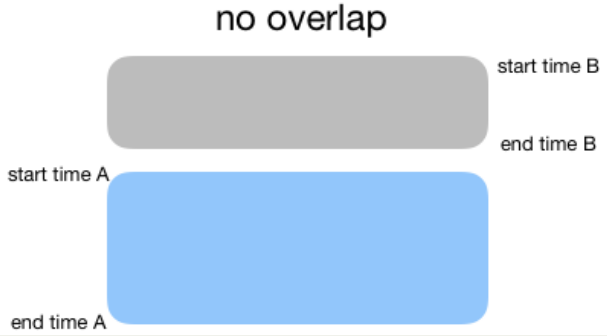
There was a diagram on the lab that hopefully allowed you to prove this to yourself.

```
    def overlaps(self, other):
        """
        Compares two events and determines if there is overlap.
        Two intervals do not overlap when the start date of the first is before
        the start date of the last, and the start date of the last is before the
        start date of the first.
        :param other: an CalendarEvent
        :return: True if there is overlap, False if not
        """
        return self.start_date < other.end_date() and other.start_date < self.end_date()
```

Lecture 9: Last week's lab: the CalendarEvent class



Two events are considered overlapping if the start date of the first is before the end date of the second, and if the start date of the second is before the end date of the first.



Lecture 9: More on OOP: Inheritance

- One of the more interesting aspects of object oriented programming is the idea that objects can *inherit* their properties from other objects. In other words, you can create a class that has, at its start, all the properties of another class. We call this a *derived* class, as it is derived from another class.
- Why do we like this? It is all about *software reuse*, which means that we can write something once and then re-use it for something else. This is often a good use of your time.
- We explicitly create an inherited type from another type as follows:

```
class NewType(OtherClass):
  ...
```

- We are going to look at a specific example, and you will use inheritance on your next assignment.
- The example is for a simple address book that keeps track of names and email addresses. We'll call the class Contact, and it keeps track of all contacts and initializes the names and addresses. This example was borrowed from Damian T. Gordon, Lecturer in Dublin Institute of Technology.

Lecture 9: More on OOP: Inheritance

```
class Contact:
    contacts_list = []

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.contacts_list.append(self)
```

- The **`contacts_list`** variable is a *class attribute*, meaning that it is shared between all of the instances in the Contact class. Note that we call it with the **`Contact.contacts_list`** dot-notation.

Lecture 9: More on OOP: Inheritance

- Let's say that our contact list is for a company, and some of the contacts are suppliers we can order from, and some are staff members of the company.
- If we want to add an order for a contact, we should make it so that we can only order from suppliers and not staff members.
- We can create an inherited class, called Supplier, to our program:

```
class Supplier(Contact):
    def order(self, order):
        print(f"The order will send {order} to {self.name}")
```

- Now we have a **`Supplier`** class that has all of the properties of the **`Contact`** class built-in, with the addition of the **`order`** method.

Lecture 9: More on OOP: Inheritance

- Let's add some testing code to see if it works:

```
def main():
    c1 = Contact("Vera StaffMember", "VeraStaffMember@MyCompany.com")
    c2 = Contact("Mac StaffMember", "MacStaffMember@MyCompany.com")
    s1 = Supplier("Ely Supplier", "ElySupplier@Supplies.com")
    s2 = Supplier("Kristina Supplier", "KristinaSupplier@Supplies.com")

    print(f"Our Staff Members are: {c1.name} and {c2.name}")
    print(f"Their email addresses are: {c1.email} and {c2.email}")
    print(f"Our Suppliers are: {s1.name} and {s2.name}")
    print(f"Their email addresses are: {s1.email} and {s2.email}")
```

- Output:

```
Our Staff Members are: Vera StaffMember and Mac StaffMember
Their email addresses are: VeraStaffMember@MyCompany.com and MacStaffMember@MyCompany.com
Our Suppliers are: Ely Supplier and Kristina Supplier
Their email addresses are:ElySupplier@Supplies.com and KristinaSupplier@Supplies.com
```

Lecture 9: More on OOP: Inheritance

- We can order from suppliers:

```
def main():
    c1 = Contact("Vera StaffMember", "VeraStaffMember@MyCompany.com")
    c2 = Contact("Mac StaffMember", "MacStaffMember@MyCompany.com")
    s1 = Supplier("Ely Supplier", "ElySupplier@Supplies.com")
    s2 = Supplier("Kristina Supplier", "KristinaSupplier@Supplies.com")

    print(f"Our Staff Members are: {c1.name} and {c2.name}")
    print(f"Their email addresses are: {c1.email} and {c2.email}")
    print(f"Our Suppliers are: {s1.name} and {s2.name}")
    print(f"Their email addresses are: {s1.email} and {s2.email}")

    s1.order("Bag of sweets")
    s2.order("Boiled eggs")
```

- Output:

```
Our Staff Members are: Vera StaffMember and Mac StaffMember
Their email addresses are: VeraStaffMember@MyCompany.com and MacStaffMember@MyCompany.com
Our Suppliers are: Ely Supplier and Kristina Supplier
Their email addresses are:ElySupplier@Supplies.com and KristinaSupplier@Supplies.com
The order will send Bag of sweets to Ely Supplier
The order will send Boiled eggs to Kristina Supplier
```

Lecture 9: More on OOP: Inheritance

- But we can't order from staff:

```
def main():
    c1 = Contact("Vera StaffMember", "VeraStaffMember@MyCompany.com")
    c2 = Contact("Mac StaffMember", "MacStaffMember@MyCompany.com")
    s1 = Supplier("Ely Supplier", "ElySupplier@Supplies.com")
    s2 = Supplier("Kristina Supplier", "KristinaSupplier@Supplies.com")

    print(f"Our Staff Members are: {c1.name} and {c2.name}")
    print(f"Their email addresses are: {c1.email} and {c2.email}")
    print(f"Our Suppliers are: {s1.name} and {s2.name}")
    print(f"Their email addresses are: {s1.email} and {s2.email}")

    c1.order("Bag of sweets")
    c2.order("Boiled eggs")
```

- Output:

```
Our Staff Members are: Vera StaffMember and Mac StaffMember
Their email addresses are: VeraStaffMember@MyCompany.com and MacStaffMember@MyCompany.com
Our Suppliers are: Ely Supplier and Kristina Supplier
Their email addresses are: ElySupplier@Supplies.com and KristinaSupplier@Supplies.com
Traceback (most recent call last):
  File "/Users/tofer/Dropbox/NE/CS5001/PycharmProjects/AddressBook/address-book.py", line 34, in <module>
    main()
  File "/Users/tofer/Dropbox/NE/CS5001/PycharmProjects/AddressBook/address-book.py", line 29, in main
    c1.order("Bag of sweets")
AttributeError: 'Contact' object has no attribute 'order'
```

Lecture 9: More on OOP: Inheritance

- Let's make our class better -- let's add a method to search for a contact. Where would we put that? It probably makes sense to associate it with the **`contact_list`** itself.
- Let's do something interesting:

```
class ContactList(list):
    def search(self, name):
        """Return any search hits"""
        matching_contacts = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts
```

  - We have now built a new class based on a **`list`**!

Lecture 9: More on OOP: Inheritance

- We can now change our original **`Contact`** class:

```
class Contact:
    contacts_list = ContactList()

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.contacts_list.append(self)
```

- We can test it:

```
def main():
    c1 = Contact("Vera StaffMember", "VeraStaffMember@MyCompany.com")
    c2 = Contact("Mac StaffMember", "MacStaffMember@MyCompany.com")
    s1 = Supplier("Ely Supplier", "ElySupplier@Supplies.com")
    s2 = Supplier("Kristina Supplier", "KristinaSupplier@Supplies.com")

    print(f"Our Staff Members are: {c1.name} and {c2.name}")
    print(f"Their email addresses are: {c1.email} and {c2.email}")
    print(f"Our Suppliers are: {s1.name} and {s2.name}")
    print(f"Their email addresses are: {s1.email} and {s2.email}")

    search_name = input("Who would you like to search for? ")
    print([c.name for c in Contact.contacts_list.search(search_name)])
```

Lecture 9: More on OOP: Inheritance

```
def main():
    c1 = Contact("Vera StaffMember", "VeraStaffMember@MyCompany.com")
    c2 = Contact("Mac StaffMember", "MacStaffMember@MyCompany.com")
    s1 = Supplier("Ely Supplier", "ElySupplier@Supplies.com")
    s2 = Supplier("Kristina Supplier", "KristinaSupplier@Supplies.com")

    print(f"Our Staff Members are: {c1.name} and {c2.name}")
    print(f"Their email addresses are: {c1.email} and {c2.email}")
    print(f"Our Suppliers are: {s1.name} and {s2.name}")
    print(f"Their email addresses are: {s1.email} and {s2.email}")

    search_name = input("Who would you like to search for? ")
    print([c.name for c in Contact.contacts_list.search(search_name)])
```

- Output

```
Our Staff Members are: Vera StaffMember and Mac StaffMember
Their email addresses are: VeraStaffMember@MyCompany.com and MacStaffMember@MyCompany.com
Our Suppliers are: Ely Supplier and Kristina Supplier
Their email addresses are: ElySupplier@Supplies.com and KristinaSupplier@Supplies.com
Who would you like to search for? Ely
['Ely Supplier']
```

Lecture 9: More on OOP: Overriding and super

- So far, we've used inheritance to add new behavior.
- But, we can also *change* behavior.
- Let's look at **`Supplier`** again, and change how **`order`** works.

```
47  class SupplierCheck(Supplier):
48      def order(self, order, balance):
49          if balance < 0:
50              print("This customer is in debt.")
51          else:
52              print(f"The order will send our {order} to {self.name}")
53
```

- Notice that PyCharm puts the little blue circle with a red arrow on line 48 above -- this means that we have changed a behavior that the original class had.

Lecture 9: More on OOP: Overriding and super

- When we test it, we get different behaviors for different types of suppliers:

```
def main():
    s1 = Supplier("Ely Supplier", "ElySupplier@Supplies.com")
    s2 = Supplier("Kristina Supplier", "KristinaSupplier@Supplies.com")
    s3 = SupplierCheck("Anne Supplier", "AnneSupplier@Supplies.com")
    s4 = SupplierCheck("Mary Supplier", "MarySupplier@Supplies.com")

    s1.order("Bag of sweets")
    s2.order("Boiled eggs")
    s3.order("Coffee", 23)
    s3.order("Wine", -12)
```

- Output:

```
The order will send Bag of sweets to Ely Supplier
The order will send Boiled eggs to Kristina Supplier
The order will send our Coffee to Anne Supplier
This customer is in debt.
```

Lecture 9: More on OOP: Overriding and super

- Overriding works well, but we can do better. Right now, there is still some duplicated code:

```
class Supplier(Contact):
    def order(self, order):
        print(f"The order will send {order} to {self.name}")


class SupplierCheck(Supplier):
    def order(self, order, balance):
        if balance < 0:
            print("This customer is in debt.")
        else:
            print(f"The order will send our {order} to {self.name}")
```

- So, it would be better if we could only make the check in the SupplierCheck function, but have the common part remain the same. We can! We will use the *super* class (see next slide).

Lecture 9: More on OOP: Overriding and super

- Overriding works well, but we can do better. Right now, there is still some duplicated code:

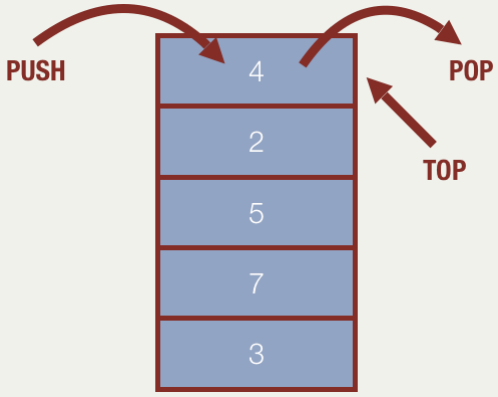
```
class Supplier(Contact):
    def order(self, order):
        print(f"The order will send {order} to {self.name}")


class SupplierCheck(Supplier):
    def order(self, order, balance):
        if balance < 0:
            print("This customer is in debt.")
        else:
            super().order(order)
```

- Now, we have the similar code in **`Supplier`**, and **`SupplierCheck`** calls its parent (**`super`**) to call the similar code.

Lecture 9: Stacks

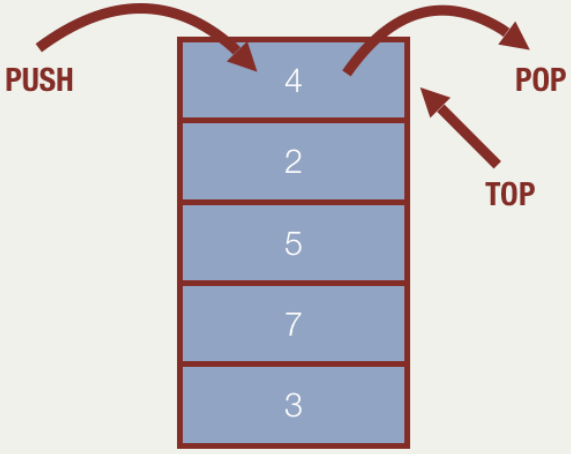
- So far in this course, we have talked about lists, tuples, dictionaries, etc., and now we can start talking about some more interesting *data types*. The first is a *stack*. A stack has the following behaviors and functions:
- **`push(value)`** (or **`add(value)`**) - place an entity onto the top of the stack
- **`pop()`** (or **`remove()`**) - remove an entity from the top of the stack and return it
- **`top()`** (or **`peek()`**) - look at the entity at the top of the stack, but don't remove it
- **`is_empty()`** - a boolean value, true if the stack is empty, false if it has at least one element. (note: a runtime error occurs if a pop() or top() operation is attempted on an empty stack.)

Why do we call it a "stack?" Because we model it using a stack of things:



Lecture 9: Stacks

- The **`push`**, **`pop`**, and **`top`** operations are the only operations allowed by the stack, and as such, only the top element is accessible. Therefore, a stack is a "Last-In-First-Out" (LIFO) structure: the last item in is the first one out of a stack.



- Despite the stack's limitations (and indeed, because of them), the stack is a very frequently used data structure. In fact, most computer architectures implement a stack at the very core of their instruction sets — both **`push`** and **`pop`** are *assembly code* instructions.

Lecture 9: Queues

...

Lecture 9: Queues

...

Lecture 9: Falling Particles Assignment (out tomorrow)

...