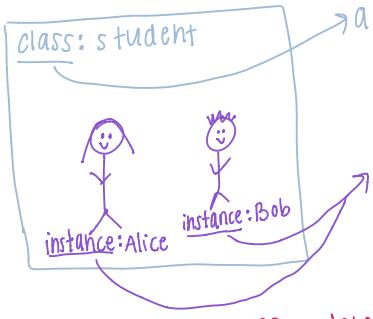
QUICK CONTENT REVIEW

- OOP (object oriented programming)
  - → allows us to treat data as objects

ex.)



→ a template for creating objects (if we know all students need a name, age, & school we can just create a template that we can use everytime we have a new student)

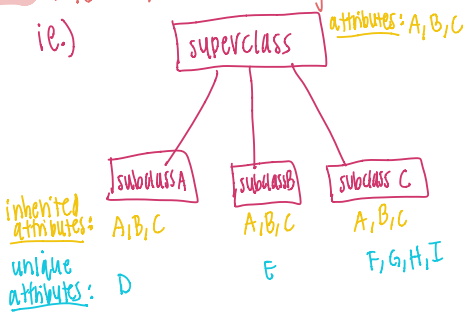
→ a single object created from the class (template is filled in)

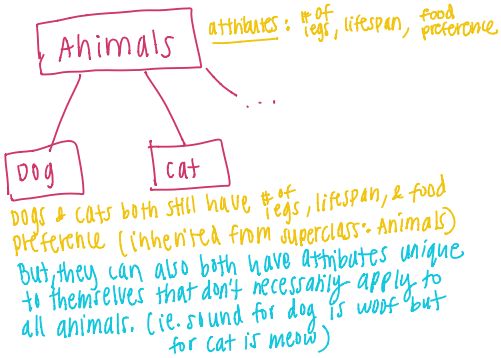
- instance attributes → details that all instances of a class have but that are different for each attribute (ex. names; both Alice & Bob are students but they have diff names)
- class attributes → property of the object that is shared by all instances of a class (ex. total # of students; this value is not different for Alice & Bob)
- method → smthg that all instances of a class can do (usually an action) (ex. all students can: do hw, go to oh, take exams, etc.)

→ if you need a template for smthg, make a **class** for it
  ↳ to use this template to make a new object, create an **instance** of the class
    ↳ everything that this object should be able to do can be added as a class **method**

- Inheritance
  - → if multiple classes share a lot of similar qualities, can just write a single **superclass** that contains all the qualities they had in common, and then each of the classes can just **inherit** the superclass to get access to all those qualities

ie.)





Dogs & cats both still have # of legs, lifespan, & food preference (inherited from superclass: Animals) But, they can also both have attributes unique to themselves that don't necessarily apply to all animals. (ie. sound for dog is woof but for cat is meow)

→ Hierarchal relationship
subclass is more specific version of superclass.

subclass is a superclass.

→ can always use this to test if hierarchy makes sense
- Dog is an animal ✓
- animal is a dog ✗

- iterators & generators
  - iterator: used to retrieve values contained in iterable one-by-one → commonly used to represent infinite sequences
  - generator: special type of function that uses 'yield' statements in place of 'return' statements

Keys

1. iter(): returns iterator object
2. next(): returns next value in iterable
3. 'stopIterationError': when you call 'next()' but all objects have already been iterated through
4. Any function with 'yield' in it returns a generator
5. generator returns an iterator (which is an iterable)
6. list(iterator): takes all items of an iterator & places them in a list

PRACTICE PROBLEMS

Email (Q2)

Tutorial:

We now want to write three different classes, `Server`, `Client`, and `Email` to simulate email. Fill in the definitions below to finish the implementation! There are more methods to fill out on the next page.

We suggest that you approach this problem by first filling out the `Email` *class, then fill out the* `register_client` *method of* `Server`*, then implement the* `Client` *class, and lastly fill out the send method of the* `Server` *class.*

collabedit

1

```
class Email:
    """Every email object has 3 instance attributes: the
    message, the sender name, and the recipient name.
    """
    def __init__(self, msg, sender_name, recipient_name):
```

3

```
class Client:
    """Every Client has instance attributes name (which is
    used for addressing emails to the client), server
    (which is used to send emails out to other clients), and
    inbox (a list of all emails the client has received).
    """
    def __init__(self, server, name):
        self.inbox = []
```

```
class Server:
    """Each Server has an instance attribute clients, which
    is a dictionary that associates client names with
    client objects.
    """
    def __init__(self):
        self.clients = {}
```

```
    def compose(self, msg, recipient_name):
        """Send an email with the given message msg to the
        given recipient client.
        """
```

4

```
    def send(self, email):
        """Take an email and put it in the inbox of the client
        it is addressed to.
        """
```

```
    def receive(self, email):
        """Take an email and add it to the inbox of this
        client.
        """
```

2

```
    def register_client(self, client, client_name):
        """Takes a client object and client_name and adds them
        to the clients instance attribute.
        """
```

general tips
- start with one class at a time
  - ↳ '__init__' is often the easiest one to start off with since we are just initializing all of our inputs
- read the comments very carefully!

ANSWER:

```
class Email:
    """Every email object has 3 instance attributes: the
    message, the sender name, and the recipient name.
    """
    def __init__(self, msg, sender_name, recipient_name):
        self.msg = msg
        self.sender_name = sender_name
        self.recipient_name = recipient_name
```

same structure for most '__init__' methods - initialize all instance variables (values passed in as parameters)

```
class Server:
    """Each Server has an instance attribute clients, which
    is a dictionary that associates client names with
    client objects.
    """
    def __init__(self):
        self.clients = {}

    def send(self, email):
        """Take an email and put it in the inbox of the client
        it is addressed to.
        """
        receiver = email.recipient_name
        rec_client = self.clients[receiver]
        rec_client.receive(email)
```

find the correct client object

use its 'receive' method

```
    def register_client(self, client, client_name):
        """Takes a client object and client_name and adds them
        to the clients instance attribute.
        """
        self.clients[client_name] = client
```

```
class Client:
    """Every Client has instance attributes name (which is
    used for addressing emails to the client), server
    (which is used to send emails out to other clients), and
    inbox (a list of all emails the client has received).
    """
    def __init__(self, server, name):
        self.inbox = []
        self.name = name
        self.server = server
        self.server.register_client(self, self.name)

    def compose(self, msg, recipient_name):
        """Send an email with the given message msg to the
        given recipient client.
        """
        email = Email(msg, self.name, recipient_name)
        self.server.send(email)
```

again just initializing (filling in the template for this object)

this client object

if client is going to use this server they need to register

make an email object

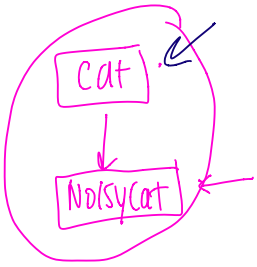
use my server to send it

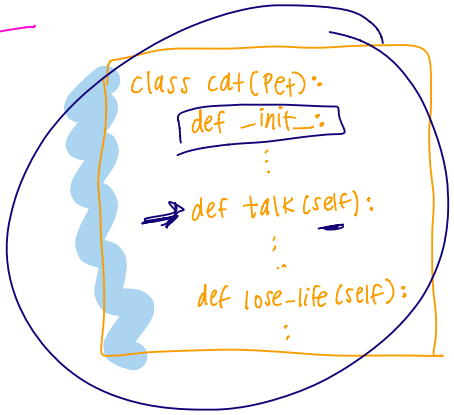
```
    def receive(self, email):
        """Take an email and add it to the inbox of this
        client.
        """
        self.inbox.append(email)
```

just add it onto inbox list

Cat (Q4) ~WARM UP~

2.2 **Tutorial:** More cats! Fill in this implemention of a class called `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot – twice as much as a regular `Cat`!






```
class NoisyCat(Cat): # Fill me in!

    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?
        super().__init__(self, name, owner, lives)

    def talk(self):
        """Talks twice as much as a regular cat.

        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """
        super().talk(self)
        super().talk(self)
```

SOLUTION:

```
class NoisyCat(Cat): # Fill me in!
    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?
        super().__init__(self, name, owner, lives)

    def talk(self):
        """Talks twice as much as a regular cat.

        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """
        super().talk()
        super().talk()
```

name of class: NoisyCat
inheriting: Cat (since basic template is same, we just want to make small changes for NoisyCats specifically)

not necessary cuz NoisyCat is already inheriting __init__ of Cat
↳ only necessary if we need to add smthg not already in Cat's __init__

alternative way to do

```
cat.talk(self)
cat.talk(self)
```

since NoisyCat inherits Cat, it has access to all Cat methods but must specify its calling a Cat method by doing: Cat.method_name(NoisyCat object)

self refers to this NoisyCat

Merge (Q7)

4.2 Write a generator function merge that takes in two infinite generators a and b that are in increasing order without duplicates and returns a generator that has all the elements of both generators, in increasing order, without duplicates

```
def merge(a, b):
    """
    >>> def sequence(start, step):
    ...     while True:
    ...         yield start
    ...         start += step
    >>> a = sequence(2, 3) # 2, 5, 8, 11, 14, ...
    >>> b = sequence(3, 2) # 3, 5, 7, 9, 11, 13, 15, ...
    >>> result = merge(a, b) # 2, 3, 5, 7, 8, 9, 11, 13, 14, 15
    >>> [next(result) for _ in range(10)]
    [2, 3, 5, 7, 8, 9, 11, 13, 14, 15]
    """
```

- generator returns an iterator
- think of the infinite generators as sequences (you can start off by just imagining lists to get the basic intuition of the problem first!)

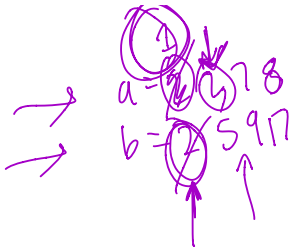
optional breakout rooms!

- Remember the "new sequence" is constructed by the yield statements.
  - each time we call next() on the iterator returned by merge(a,b) we go up to the first yield statement (output one element of sequence) & then stop until next time next() is called (we resume right where we left off last time)

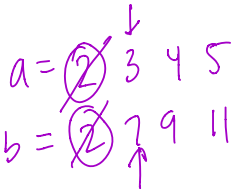
SOLUTION:

```
def merge(a,b):
    first_a = next(a)
    first_b = next(b)
    while True:
        if first_a == first_b:
            yield first_a
            first_a, first_b = next(a), next(b)
        elif first_a < first_b:
            yield first_a
            first_a = next(a)
        else:
            yield first_b
            first_b = next(b)
```

- can think of a & b as sequences so these are the values of both
- essentially means we just keep going till we reach an error (OK); reach end of a & b (infinite sequence)
- if first element of a is same as first element of b, we only want one of them to appear in new sequence (no duplicates) & can now move to next 2 elements in both sequences to compare
- if first element of a is smaller, we want that element to come before the larger first element of b in new sequence, so "add" first element of a to new sequence and then get next value in sequence a to compare to first element of b
- first element of b is smaller, same as case above except w/ a & b reversed



c = 2



→ c = 2