Python classes can implement a useful abstraction technique known as inheritance. To illustrate this concept, consider the following Dog and Cat classes.

class Dog:
    def __init__(self, name, owner):
        self.is_alive = True
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says woof!")

class Cat:
    def __init__(self, name, owner, lives=9):
        self.is_alive = True
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says meow!")

Notice that because dogs and cats share a lot of similar qualities, there is a lot of repeated code! To avoid redefining attributes and methods for similar classes, we can write a single
superclass from which the similar classes inherit. For example, we can write a class called Pet and redefine Dog as a subclass of Pet:

```python
class Pet:
    def __init__(self, name, owner):
        self.is_alive = True   # It's alive!!
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name)

class Dog(Pet):
    def talk(self):
        print(self.name + ' says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class is a more specific version of the other, e.g. a dog is a pet. Because Dog inherits from Pet, we didn’t have to redefine __init__ or eat. However, since we want Dog to talk in a way that is unique to dogs, we did override the talk method.
2 Questions

1. Assume these commands are entered in order. What would Python output?

```python
>>> class Foo:
...     def __init__(self, a):
...         self.a = a
...     def garply(self):
...         return self.baz(self.a)

>>> class Bar(Foo):
...     a = 1
...     def baz(self, val):
...         return val

>>> f = Foo(4)
>>> b = Bar(3)
>>> f.a

>>> b.a

>>> f.garply()

>>> b.garply()

>>> b.a = 9
>>> b.garply()

>>> f.baz = lambda val: val * val
>>> f.garply()
```
2. Below is a skeleton for the Cat class, which inherits from the Pet class. To complete the implementation, override the __init__ and talk methods and add a new lose_life method.

Hint: You can call the __init__ method of Pet to set a cat’s name and owner.

class Cat(Pet):
    def __init__(self, name, owner, lives=9):

        def talk(self):
            """ Print out a cat's greeting."

            >>> Cat('Thomas', 'Tammy').talk()
            Thomas says meow!
            """

        def lose_life(self):
            """Decrements a cat's life by 1. When lives reaches zero, 'is_alive'
            becomes False."
            """
When we talk about the efficiency of a function, we are often interested in the following: as the size of the input grows, how does the runtime of the function change? And what do we mean by “runtime”?

- **square(1)** requires one primitive operation: \( \ast \) (multiplication). **square(100)** also requires one. No matter what input \( n \) we pass into **square**, it always takes one operation.

<table>
<thead>
<tr>
<th>input</th>
<th>function call</th>
<th>return value</th>
<th>number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>square(1)</td>
<td>1 \cdot 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>square(2)</td>
<td>2 \cdot 2</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>square(100)</td>
<td>100 \cdot 100</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( n )</td>
<td>square(( n ))</td>
<td>( n \cdot n )</td>
<td>1</td>
</tr>
</tbody>
</table>

- **factorial(1)** requires one multiplication, but **factorial(100)** requires 100 multiplications. As we increase the input size of \( n \), the runtime (number of operations) increases linearly proportional to the input.

<table>
<thead>
<tr>
<th>input</th>
<th>function call</th>
<th>return value</th>
<th>number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>factorial(1)</td>
<td>1 \cdot 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>factorial(2)</td>
<td>2 \cdot 1 \cdot 1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>factorial(100)</td>
<td>100 \cdot 99 \cdot ... \cdot 1 \cdot 1</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( n )</td>
<td>factorial(( n ))</td>
<td>( n \cdot (n - 1) \cdot ... \cdot 1 \cdot 1 )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Here are some general guidelines for finding the order of growth for the runtime of a function:

- If the function is recursive or iterative, you can subdivide the problem as seen above:
  - Count the number of recursive calls/iterations that will be made in terms of input size \( n \).
  - Find how much work is done per recursive call or iteration in terms of input size \( n \).

  The answer is usually the product of the above two, but be sure to pay attention to control flow!

- If the function calls helper functions that are not constant-time, you need to take the runtime of the helper functions into consideration.
• We can ignore constant factors. For example, \( \Theta(1000000n) = \Theta(n) \).

• We can also ignore lower-order terms. For example, \( \Theta(n^3 + n^2 + 4n + 399) = \Theta(n^3) \). This is because the \( n^3 \) term dominates as \( n \) gets larger.
1. What is the runtime of the following function?
   ```python
   def one(n):
       if 1 == 1:
           return None
       return n
   ```
   a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

2. What is the runtime of the following function?
   ```python
   def two(n):
       for i in range(n):
           print(n)
   ```
   a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

3. What is the runtime of the following function?
   ```python
   def three(n):
       while n > 0:
           n = n // 2
   ```
   a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

4. What is the runtime of the following function?
   ```python
   def four(n):
       for i in range(n):
           for j in range(i):
               print(str(i), str(j))
   ```
   a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

5. What is the runtime of the following function?
   ```python
   def five(n):
       if n <= 0:
           return 1
       return five(n - 1) + five(n - 2)
   ```
   a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

6. What is the runtime of the following function?
   ```python
   def five(n):
       if n <= 0:
           return 1
       return five(n//2) + five(n//2)
   ```
   a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)