INHERITANCE AND ASYMPTOTICS 8

DATA C88C

October 26, 2022

1 Inheritance

1.1 Introduction

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`:

```
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.

1.2 Questions

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

```
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
```

Solution: 4

```
>>> b.a
```

Solution: 3

```
>>> f.garply()
```

Solution: `AttributeError: 'Foo'object has no attribute 'baz'`

```
>>> b.garply()
```

Solution: 3

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

Solution: 9

```
>>> f.baz = lambda val: val * val
>>> f.garply()
```

Solution: `16`

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):
```

Solution:

```
        Pet.__init__(self, name, owner)
        self.lives = lives
```

```
    def talk(self):
        """ Print out a cat's greeting.
        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """
```

Solution:

```
        print(self.name + ' says meow!')
```

```
    def lose_life(self):
        """Decrements a cat's life by 1. When lives reaches
        zero, 'is_alive' becomes False.
        """
```

Solution:

```
        if self.lives > 0:
            self.lives -= 1
            if self.lives == 0:
                self.is_alive = False
        else:
            print("This cat has no more lives to lose :(")
```

Video walkthrough

3. 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 ________________________: # Fill me in!
```

Solution:
```
class NoisyCat(Cat):
```

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

Solution:
```
        Cat.__init__(self, name, owner, lives)
```

No, this method is not necessary because NoisyCat already inherits Cat's __init__ method

```
    def talk(self):
        """Talks twice as much as a regular cat.
        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """
```

Solution:
```
        Cat.talk(self)
        Cat.talk(self)
```

Video walkthrough

2 Asymptotics

2.1 Introduction

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

input	function call	return value	number of operations
1	square(1)	$1 \cdot 1$	1
2	square(2)	$2 \cdot 2$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
100	square(100)	$100 \cdot 100$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	square(n)	$n \cdot n$	1

- `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.

input	function call	return value	number of operations
1	factorial(1)	$1 \cdot 1$	1
2	factorial(2)	$2 \cdot 1 \cdot 1$	2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
100	factorial(100)	$100 \cdot 99 \cdots 1 \cdot 1$	100
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	factorial(n)	$n \cdot (n-1) \cdots 1 \cdot 1$	n

2.2 Guidelines

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.

2.3 Questions

1. What is the runtime of the following function?

```
def one(n):
    if 1 == 1:
        return None
    return n
```

a. $\Theta(1)$ b. $\Theta(logn)$ c. $\Theta(n)$ d. $\Theta(n^2)$ e. $\Theta(2^n)$

Solution: $\Theta(1)$ - the function always returns None, because 1 == 1 is always True. And even if it was a false statement, the function would just return n. So since the runtime of the function doesn't change with respect to the size of the input, it is constant time.

2. What is the runtime of the following function?

```
def two(n):
    for i in range(n):
        print(n)
```

a. $\Theta(1)$ b. $\Theta(logn)$ c. $\Theta(n)$ d. $\Theta(n^2)$ e. $\Theta(2^n)$

Solution: $\Theta(n)$ - the function iterates n times; if n increases by 1, the function loops 1 additional time. Therefore there is a linear relationship between the input size and runtime.

3. What is the runtime of the following function?

```
def three(n):
    while n > 0:
        n = n // 2
```

a. $\Theta(1)$ b. $\Theta(logn)$ c. $\Theta(n)$ d. $\Theta(n^2)$ e. $\Theta(2^n)$

Solution: $\Theta(logn)$ - The function continues to loop as long as n $>$ 0. Inside the while loop, we divide n by 2 every loop. So to get the function to loop one additional time, we need to double our original input size. This is a logarithmic relationship between input size and runtime.

4. What is the runtime of the following function?

```
def four(n):
    for i in range(n):
        for j in range(i):
            print(str(i), str(j))
```

a. $\Theta(1)$ b. $\Theta(logn)$ c. $\Theta(n)$ d. $\Theta(n^2)$ e. $\Theta(2^n)$

Solution: d. $\Theta(n^2)$ - The outer loop loops through every number from 0 to n. The inner loop loops corresponding to the outer loop. So the total number of loops from the inner loop looks like this: 0 + 1 + 2 + 3 + 4 ... + n. This is the summation of the first n natural numbers = n(n + 1)/2, which asymptotically is $\Theta(n^2)$.

5. What is the runtime of the following function?

```
def five(n):
    if n <= 0:
        return 1
    return five(n - 1) + five(n - 2)
```

a. $\Theta(1)$ b. $\Theta(logn)$ c. $\Theta(n)$ d. $\Theta(n^2)$ e. $\Theta(2^n)$

Solution: e. $\Theta(2^n)$ - Draw out the tree of recursive calls. You should see that every node branches out into 2 more nodes. Since the base case returns when n $<=$ 0, and each recursive call subtracts 1 or 2 from n, the height of our tree is n. We're branching out by a factor of 2 each layer for n layers – that means we'll have 2^n nodes in our tree of recursive calls. Each 'node' represents 1 'unit of work' as all the function does is return something. So 1 unit of work across 2^n nodes is 2^n total.

6. What is the runtime of the following function?

```
def six(n):
    if n <= 0:
        return 1
    return six(n//2) + six(n//2)
```

a. $\Theta(1)$ b. $\Theta(logn)$ c. $\Theta(n)$ d. $\Theta(n^2)$ e. $\Theta(2^n)$

Solution: c. $\Theta(n)$ - Draw out the tree of recursive calls. You should see that every node branches out into 2 more nodes. Since the base case returns when n $<=$ 0, and each recursive call divides n by 2, the height of our tree is logn (by the same logic as three(n): if we want one additional layer in our tree, our original input has to be doubled, which is a logarithmic relationship). We're branching out by a factor of 2 each layer for logn layers – that means we'll have $2^{logn} = n$ nodes in our tree of recursive calls. Each 'node' represents 1 'unit of work' as all the function does is return something. So 1 unit of work across n nodes is n total.