Inheritance

Announcements

Announcements

- Midterm grades to be released Wed 3/19 (tentative)

- OH schedule updates
  - All 9 am OH canceled starting today Mon 3/17
  - More updates to come (stay tuned on Ed)

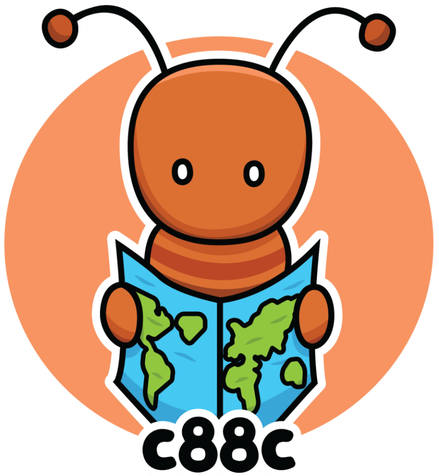
- **[EC opportunity]** Mid-semester survey to be released soon
  - If 75% or more of the class fills out the survey by Mon 3/31, everyone will receive 1 pt of EC

- Feedback: go.c88c.org/rebecca-lec or scan QR code

Inheritance Basics

Why inheritance?

- The world can be represented by objects, and objects can be related
- DRY: Don’t Repeat Yourself
- Ants project
- Fun fact: Our mascot is named InheritAnt!



Inheritance syntax

```
class <class_name>(<superclass_name>):
    <suite>
```

Examples:

```
class Dog(Animal):
    …

class Cat(Animal):
    …
```

Terminology

- superclass = parent class = base class
- subclass = child class

Overriding methods and attributes

```
class Animal:
    def __init__(self, name):
        self.name = name

    def make_noise(self):
        print(f'{self.name} made a noise!')

class Dog(Animal):
    def __init__(self, name, owner):
        super().__init__(name)
        self.owner = owner

    def make_noise(self):
        print('Woof!')

class Cat(Animal):
    def __init__(self, name, owner):
        super().__init__(name)
        self.owner = owner

    def make_noise(self):
        print('Meow!')
```

```
>>> animal = Animal('Bessie')
>>> animal.name
'Bessie'
>>> animal.owner
AttributeError: 'Animal' object has no attribute 'owner'
>>> animal.make_noise()
Bessie made a noise!
>>> dog = Dog('Boba', 'Upasana')
>>> dog.name
'Boba'
>>> dog.owner
'Upasana'
>>> dog.make_noise()
Woof!
>>> cat = Cat('Rigatoni', 'Andie')
>>> cat.make_noise()
Meow!
>>> Cat.make_noise()
TypeError: make_noise() missing 1 required positional argument:
'self'
>>> Cat.make_noise(cat)
Meow!
```

Overriding methods and attributes

```
class Animal:
    def __init__(self, name):
        self.name = name

    def make_noise(self):
        print(f'{self.name} made a noise!')

class Dog(Animal):
    def __init__(self, name, owner):
        super().__init__(name)
        self.owner = owner

    def make_noise(self):
        print('Woof!')

class Cat(Animal):
    def __init__(self, name, owner):
        super().__init__(name)
        self.owner = owner

    def make_noise(self):
        print('Meow!')
```

Q: What additional superclass might we want to make to avoid repeating ourselves?

A: Pet class that inherits from Animal and includes an owner attribute. Then Dog and Cat can inherit from Pet!

Lookup

Lookup rules

Instance variable lookup

1. Lookup name in instance
2. Lookup name in class that instance belongs to
3. Lookup in parent class, if one exists (recursively)
4. Error if still not found

Class variable lookup

1. Lookup in class
2. Look up in parent class, if one exists (recursively)
3. Error if still not found

Lookup exercise

```
class A:
    foo = 0
    def __init__(self, foo, bar):
        self.foo = foo + A.foo
        A.foo += 1
        self.bar = bar

class B(A):
    foo = 5
    def __init__(self, bar):
        super().__init__(B.foo, bar)


>>> first = A(2, 3)
>>> first.foo
2
>>> first.bar
3
>>> A.foo
1
>>> second = A(2, 3)
>>> second.foo
3
>>> second.bar
3
>>> A.foo
2

>>> third = B(2, 3)
TypeError: __init__() takes 2 positional arguments but 3 were
given
>>> third = B(3)
>>> third.foo
7
>>> third.bar
3
>>> B.foo
5
>>> A.foo
3
>>> third.foo = 100
>>> third.foo
100
>>> B.foo
5
```

type vs. isinstance

```
class C:
    pass

class D(C):
    pass
```

```
>>> first = C()
>>> type(first)
<class '__main__.C'>
>>> second = D()
>>> type(second) == D
True
>>> isinstance(first, C)
True
>>> isinstance(second, C)
True
>>> isinstance(second, D)
True
```

Applications / System Design

Inheritance vs. Composition

Inheritance: *is-a* relationship

Composition: *has-a* relationship

Let’s design Spotify!

Q: What are some objects we might want to define?

A: User, Artist, Song, Playlist, Album, etc.

Let’s design Spotify!

Q: How are these objects related to each other?

A:

- An Artist *is a* User
- An Artist *has many* Songs
- A Playlist *has many* Songs
- An Album *is a* Playlist
- A User *has many* Playlists
- An Artist *has many* Albums
- etc.

Demo: Design Spotify

Multiple Inheritance

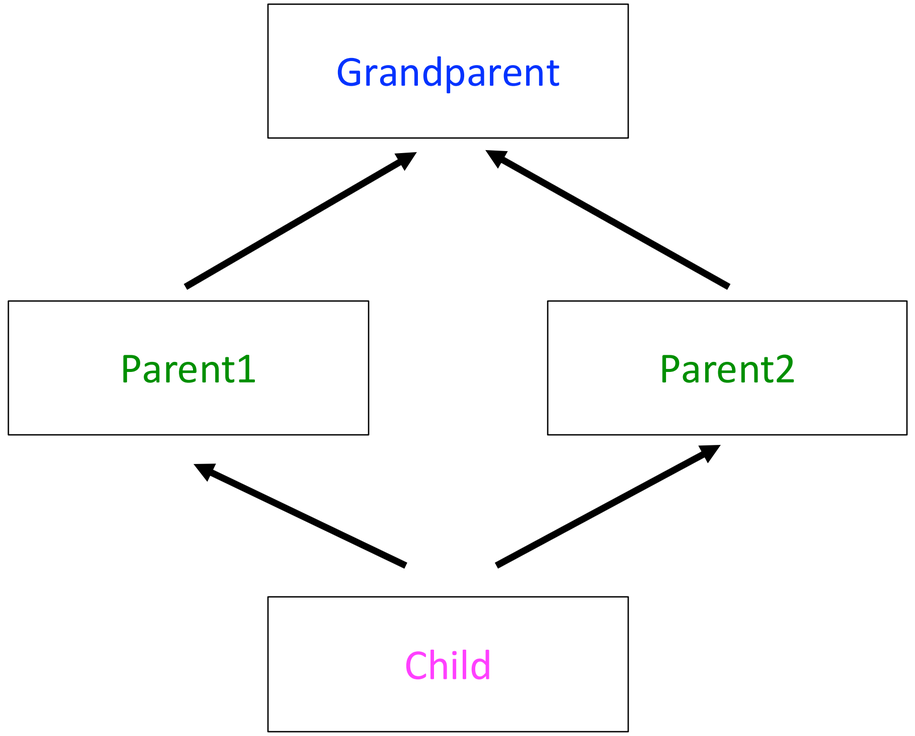
Method Resolution Order (MRO) with diamond inheritance

```
class Grandparent:
    def where_am_i(self):
        print('In grandparent')

class Parent1(Grandparent):
    def where_am_i(self):
        super().where_am_i()
        print('In parent 1')

class Parent2(Grandparent):
    def where_am_i(self):
        super().where_am_i()
        print('In parent 2')

class Child(Parent1, Parent2):
    def where_am_i(self):
        super().where_am_i()
        print('In child')
```



Python looks up attributes/methods from:

1. Current class
2. Parent classes, from left to right
3. Grandparent class

Method Resolution Order (MRO) with diamond inheritance

```
class Grandparent:
    def where_am_i(self):
        print('In grandparent')

class Parent1(Grandparent):
    def where_am_i(self):
        super().where_am_i()
        print('In parent 1')

class Parent2(Grandparent):
    def where_am_i(self):
        super().where_am_i()
        print('In parent 2')

class Child(Parent1, Parent2):
    def where_am_i(self):
        super().where_am_i()
        print('In child')
```

Python looks up attributes/methods from:

1. Current class
2. Parent classes, from left to right
3. Grandparent class

```
>>> g = Grandparent()
>>> p1 = Parent1()
>>> p2 = Parent2()
>>> c = Child()
>>> g.where_am_i()
In grandparent
>>> p1.where_am_i()
In grandparent
In parent 1
>>> p2.where_am_i()
In grandparent
In parent 2
>>> c.where_am_i()
In grandparent
In parent 2
In parent 1
In child
>>> Child.mro()
[<class '__main__.Child'>,
<class '__main__.Parent1'>,
<class '__main__.Parent2'>,
<class '__main__.Grandparent'>,
<class 'object'>]
```