Relationships Between Classes

Thomas Schwarz, SJ

Relationships between Classes

- An address class
  - Write a __str__ dunder for indian addresses

```
class Address:
    def __init__(self, country,  city, street,
                 number, postal, state, apartment = None):
        self.country = country
        self.city = city
        self.street = street
        self.postal = postal
        self.state = state
        self.number = number
        self.apartment = apartment
```

Relationships between Classes

- A class Name
  - Wide worldwide variety of names
    - What do we need?
      - Fields: First Name, Last Name, Middle Names, Patronym, Matronym, Titles, …
      - Method: Legal name (short)
      - Method: Legal name (complete)
      - Method: Address

Relationships between Classes

- A class can include another class
  - E.g. Class Person
    - Has fields Name (a class) and Address (a class)
      - We can use all methods defined on the components

```
class Person:
   def __init__(self, name, address, ID):
      self.name = name
      self.address = address
   def postal_label(self):
      return str(self.name)+'\n'+str(self.address)
```

Relationships between Classes

- One class can be a specialization of another class
  - E.g. Employee is a specialization of Person
    - Every Employee is a person
      - “is-a” relationship
    - But not every person is an employee
  - Distinguish this from the “has-a” relationship

Inheritance

- “Is-a” relationship
    - Captured in OO through *Inheritance*

Inheritance

"We started to push on the inheritance idea as a way to let novices build on frameworks that could only be assigned by experts"

- -Alan Kay: The Early History of Smalltalk

Inheritance Example

- Playing cards
  - Have suit and rank

```
class Card:
    def __init__(self, suite, rank):
        self.suite = suite
        self.rank = rank
    def __str__(self):
        return "({:2s},{:2s})".format(
                             self.suite[:2],
                             self.rank[:2])
    def __repr__(self):
        return '[Card' + str(self)+']'
```

Inheritance

- To inherit from a class, just add the name of the base class in parenthesis

```
class BlackjackCard(Card):
```

Inheritance

- To initialize a derived class, usually want to call the initializer of the base class

```
values = {'ace':11, '2':2, '3':3, '4':4, '5':5, '6':6, '7':7, '8':8,
          '9':9, '10':10, 'jack':10, 'queen':10, 'king':10}

class BlackjackCard(Card):
    def __init__(self, suite, rank):
        super().__init__(suite, rank)
        self.value = values[rank]
        self.softvalue = 1 if rank=='ace' else self.value
    def __str__(self):
        return "{} of {} with value {}({})".format(
            self.rank,
            self.suite,
            self.value,
            self.softvalue
            )
```

Inheritance

- Notice:
  - All methods in the base class are still available and attributes
  - But we can also override them

```
def __hash__(self):
        return super().__hash__()^self.softvalue
```

Calling base class function

Inheritance

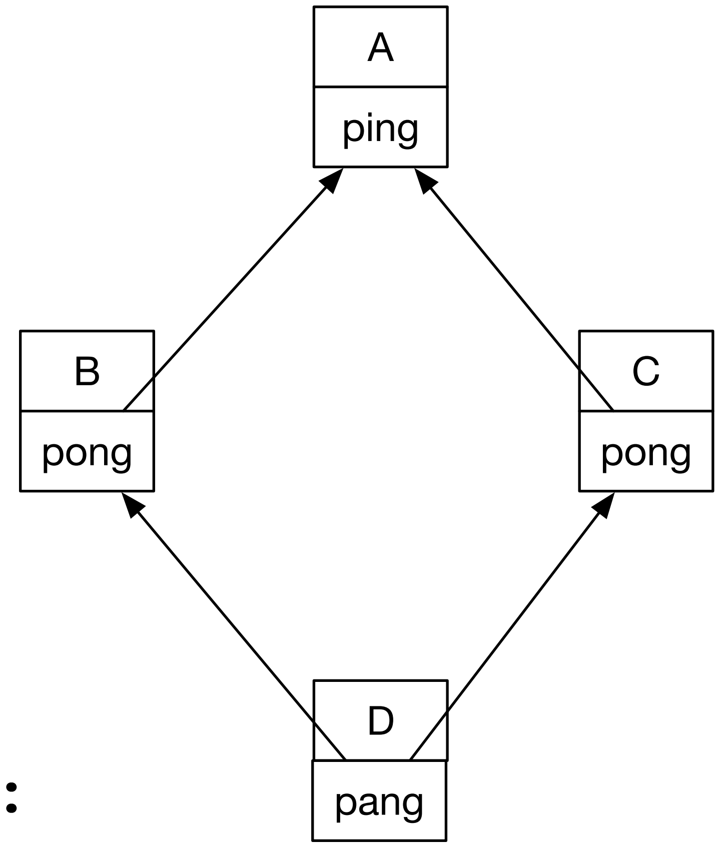
- Multiple inheritance
  - Allowed but tricky
  - **Diamond Problem**

```
class A:
   def ping(self):
      print('ping')

class B:
   def pong(self):
      print('pong')

class C:
   def pong(self):
      print('PONG')
```

```
class D(B,C):
   def ping(self):
      super().ping()
   def pang(self):
      super().ping()
      super().pong()
      C.pong(self)
```



Inheritance

- Method Resolution for d.pong( ):
  - First look in the current class
  - Then look into B
  - Then look into C
  - Then look into A
- Implemented via __mro__, which lists the classes in a certain order
- Can avoid ambiguity by giving explicit class names in the invocation

```
class D(B,C):
   def ping(self):
      super().ping()
   def pang(self):
      super().ping()
      super().pong()
      C.pong(self)
```

Inheritance

- Multiple inheritance can be used
  - Can use inheritance to define an interface:
    - A base class that requires that certain methods are implemented
  - Then multiple inheritance is fine

Classes through Special Methods

- Can find all attributes of an instance defined using __dict__ or dir :

```
>>> c=Card('heart', 'king')
>>> c.__dict__
{'suite': 'heart', 'rank': 'king'}
```

```
>>> dir(c)
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__retr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'rank',
'suite']
```

Classes through Special Methods

- Equality versus Identity
  - Default evaluation for == looks at location of storage
    - Can get storage location with object.__repr__( )
    - Or in most Python implementation, with id

```
>>> id(d)
140299613922544
>>> object.__repr__(d)
'<__main__.Card object at 0x7f9a0ca664f0>'
>>> hex(id(d))
'0x7f9a0ca664f0'
```

Classes through Special Methods

- Equality versus Identity
    - This is usually not the behavior we want
        - Equality means all attributes are equal
        - Need to define __eq__ in your class

```
class Card:
    def __eq__(self, other):
        return self.suite==other.suite and self.rank==other.rank
```

```
>>> d=Card('heart', 'king')
>>> c=Card('heart', 'king')
>>> d==c
True
```

Classes through Special Methods

- Equality versus Identity
  - We can still compare for identity with **is**

```
>>> d is c
False
```

Classes through special methods

- Identity, equality, equality of names are all different concepts
  - As the following excerpt will show

Classes through Special Methods

‘You are sad,’ the Knight said in an anxious tone: ‘let me sing you a song to comfort you.’
‘Is it very long?’ Alice asked, for she had heard a good deal of poetry that day.
‘It’s long,’ said the Knight, ‘but very, *very* beautiful. Everybody that hears me sing it—either it brings the *tears* into their eyes, or else—’
‘Or else what?’ said Alice, for the Knight had made a sudden pause.
‘Or else it doesn’t, you know. The name of the song is called “*Haddocks’ Eyes*.”’
‘Oh, that’s the name of the song, is it?’ Alice said, trying to feel interested.
‘No, you don’t understand,’ the Knight said, looking a little vexed. ‘That’s what the name is *called*. The name really *is* “*The Aged Aged Man*.”’
‘Then I ought to have said “That’s what the *song* is called”?’ Alice corrected herself.
‘No, you oughtn’t: that’s quite another thing! The *song* is called “*Ways and Means*”: but that’s only what it’s *called*, you know!’
‘Well, what *is* the song, then?’ said Alice, who was by this time completely bewildered.
‘I was coming to that,’ the Knight said. ‘The song really *is* “*A-sitting On A Gate*”: and the tune’s my own invention.’

Classes through special methods

- See:
    - Name of the name
    - Name
    - Call
    - Identity

Classes through Special Methods

- We cannot make cards into elements of sets without making them hashable

```
>>> seta = {c}
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    seta = {c}
TypeError: unhashable type: 'Card'
```

-

Classes through Special Methods

- Need to declare a method __hash__ and a method __eq__

    -
      ```
      class Card:
          def __hash__(self):
              return hash(self.suite)*hash(self.rank)
      ```
    - Now it works

```
>>> c = Card('heart', 'king')
>>> seta = {c}
>>> c in seta
True
```

Classes through Special Methods

- But to do this, we should make cards immutable
    - Right now, we can just say

```
c.rank = 'ace'
```

- Strategy: declare the components private
- Create a getter function
    - Which we do by using a property generator