ECE326

PROGRAMMING LANGUAGES

Lecture 11 : Method Resolution Order

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Property

- Makes methods appear as data attributes
- Created using `property` built-in function

```
class Person:
    def __init__(self, first, last):
        self.first, self.last = first, last

    def get_full(self):
        return self.first + " " + self.last

    def set_full(self, value):
        self.first, self.last = value.split(" ", maxsplit=1)

    def del_full(self):
        del self.first
        del self.last

    full_name = property(get_full, set_full, del_full)
```

Property

```
class Person:
    … other definitions …
    full_name = property(get_full, set_full, del_full)

>> p = Person(“John”, “Doe”)
>> p.full_name
‘John Doe’
>> del p.full_name
>> p.first
AttributeError: ‘Person' object has no attribute 'first'
>> p.full_name = “Jane Smith”
>> p.last
‘Smith’
>> p.first
‘Jane’
```

Property

- Advice
  - Don’t use if the function performs expensive computation
    - The function is called every time you access the field
    - Property “hides” the fact that it’s actually a function call
- Use case
  - Interface change
    - E.g. the field was accessed directly. Now you want to change the way it is used or accessed
  - Make field read-only
    - Do not supply the setter and deleter

Property

- Example: throw an error if Celsius below absolute zero

```
class Temperature:
    def __init__(self, value):
        self.celsius = value     # this will call the property

    def get_celsius(self):
        return self._celsius

    def get_fahrenheit(self):
        return self._celsius * 9 / 5 + 32

    def set_celsius(self, value):
        if value < -273.15:
            raise ValueError(“Cannot go below 273.15 degrees C”)
        self._celsius = value

    celsius = property(get_celsius, set_celsius)
```

Multiple Inheritance

- Python supports multiple inheritance
  - Used extensively for mixins
  - Also for naturally complex relationships
- Review
  - Python has no pure virtual functions
  - All attributes in Python objects are virtual (can be overridden)
- All base classes in Python are virtual
  - No repeated base class issue
  - A subset of the diamond problem still exists

Multiple Inheritance

```
class Person:
    def __init__(self, name, age):
        self.name = name # print("In Person")
        self.age = age

class Student(Person):
    def __init__(self, id, **kwargs):
        self.id = id # print("In Student")
        super().__init__(**kwargs)

class Teacher(Person):
    def __init__(self, resume, **kwargs):
        self.resume = resume # print("In Teacher")
        super().__init__(**kwargs)

class TA(Student, Teacher):
    def __init__(self, name, age, id, resume):
        super().__init__(name=name, age=age, id=id, resume=resume)
```

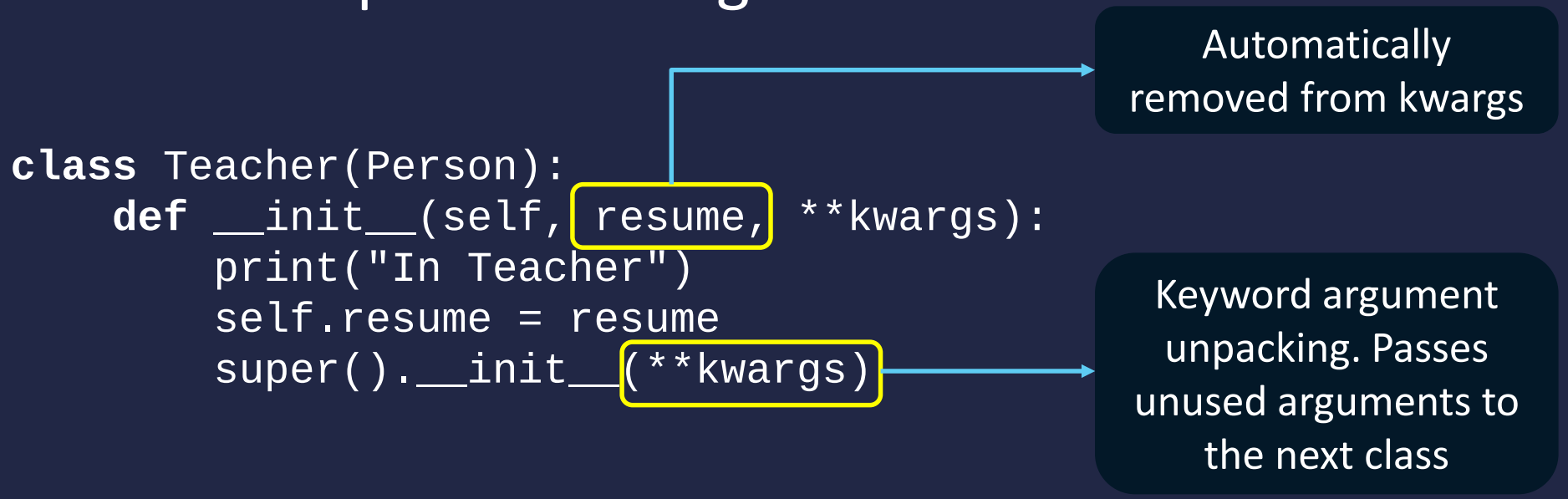
```
>> TA("jack",
.. 13, 48957257,
.. "hello world")

In Student
In Teacher
In Person
```

pass everything as keyword argument

Cooperative Inheritance

- Take the arguments you need and pass the rest on
  - Power of Python's variable keyword arguments
  - super() is used to chain the __init__ calls in different classes
- Do not use positional arguments



Alternative (Bad)

```
class Person:
    def __init__(self, name, age):
        … set name and age …

class Student(Person):
    def __init__(self, name, age, id):
        self.id = id # print("In Student")
        Person.__init__(name, age)

class Teacher(Person):
    def __init__(self, name, age, resume):
        self.resume = resume # print("In Teacher")
        Person.__init__(name, age)

class TA(Student, Teacher):
    def __init__(self, name, age, id, resume):
        Student.__init__(name, age, id)
        Teacher.__init__(name, age, resume)
```

```
>> TA(“joey",
.. 21, 63490483,
.. “good bye")

In Student
In Person
In Teacher
In Person
```

Person’s constructor got called twice…

Method Resolution Order

- The order in which attributes are looked up
  - super() knows which __init__ to call next

```
class Person:
    def vacation(self):
        return “Toronto Islands”

class Student(Person):
    pass

class Teacher(Person):
    def vacation(self):
        return “Caribbean Islands”

class TA(Student, Teacher):
    pass
```

```
>> ta = TA("jack", 13,
48957257, "hello world")
>> ta.vacation()
Caribbean Islands
>> TA.__mro__
(<class '__main__.TA'>,
<class '__main__.Student'>,
<class '__main__.Teacher'>,
<class '__main__.Person'>,
<class 'object'>)
```

Depth First Search – Left to Right

- Naïve approach
  - Used by Python 2.1 and earlier

```
class TA(Student, Teacher):
    pass
```

- MRO using DFS-LR: (TA, Student, *Person*, Teacher)
- Not what we intuitively expect
  - Teacher is more specialized than Person
- Huge problem for Python 2.2
  - `object` becomes the base class of all classes

Refinement

- Python 2.2
  - Still uses depth-first search, left to right
  - Delete duplicate if it shows up again later

```
# pseudo code for new method resolution order
def mro_v2(cls):
    mro = [ cls ]
    for parent in cls.parents:
        mro = [ c for c in mro if c not in parent.__mro__ ]
        mro.extend(parent.__mro__)
    return mro
```

- Now order is (TA, Student, Teacher, Person, object)
- However…

Local Precedence Ordering

- Order in which parent classes are inherited
- The new MRO does not honor this ordering

```
class Person:
    def vacation(self):
        return “Toronto Islands”

class Student(Person):
    def vacation(self):
        return “Queen’s Park”

class PartTime(Person, Student):
    pass
```

Under new algorithm, the MRO is:
(PartTime, Student, Person)

```
>> PartTime(…).vacation()
Queen’s Park
```

Q: But Student is more specialized than Person?
A: Yes, as such, this inheritance hierarchy is *ambiguous.*

Monotonicity

- Given that C_1 and C_2 are part of the inheritance hierarchy of C, then if C_1 precedes C_2 in the linearization of C, then C_1 must precedes C_2 in the linearization of any subclass of C.
  - Method Resolution Order (MRO) is the set of rules that constructs the linearization of class.
  - Hierarchy that fails this criteria is *ambiguous* for C
- Python 2.3 and later will raise TypeError if it detects ambiguous hierarchy

Ambiguous Hierarchy

- Using Python 2.2 MRO algorithm

```
class X: pass
class Y: pass
class A(X, Y): pass
class B(Y, X): pass
class C(A, B): pass

L[A] = (A, X, Y, o)
L[B] = (B, Y, X, o)
L[C] = (C, merge(L[A] + L[B]))
L[C] = (C, merge(A, X, Y, o, B, Y, X, o))
L[C] = (C, A, B, Y, X, o)

# this violates monotonicity because in L[C], Y comes before
# X but in L[A], X comes before Y!
```

Python 2.2 used to accept ambiguous hierarchy, which can lead to subtle bugs because the resolution ordering *changes* depending on whether A is a subclass of C or not.

```
>> a = A()  # resolution order changes
>> c = C()  # when A is a subclass of C
```

C3 Linearization

- An algorithm designed for Dylan programming language
- Maintains local precedence ordering and monotoncity
- Used by many languages
  - E.g., Python, Perl, …etc
- The linearization of C is the sum of C plus the merge of the linearizations of the parents and the list of the parents.
- $L[C(B_1 \dots B_N)] = (C, \text{merge}(L[B_1] \dots L[B_N], B_1 \dots B_N))$

Terminology

- Head
  - The first element of the list (i.e. linearization)
    - E.g. 5 is the head of the list [5, 2, 3, 7]
- Tail
  - The remaining elements of the list (not head)
    - E.g. [2, 3, 7] is the tail of the list [5, 2, 3, 7]
- L[C]
  - The linearization of the class C
- Base case: L[object] = object

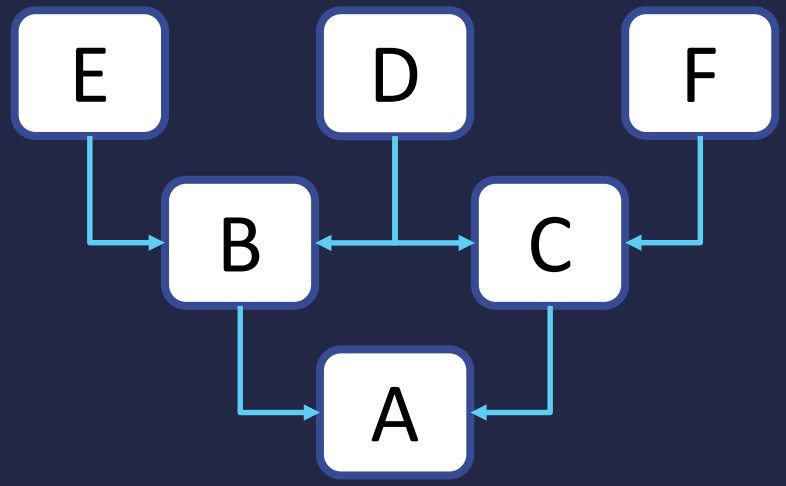
C3 Linearization

- Good head
  - A class that does not exist in the tail of any other lists
- Algorithm
  - For each list in local precedence order, remove the head from the merge *if* it is a good head.
    - Otherwise try the next list
  - Repeat until all classes are removed or there is no good head
    - In latter case, merge is not possible
    - Error will be raised for ambiguous hierarchy

C3 Linearization

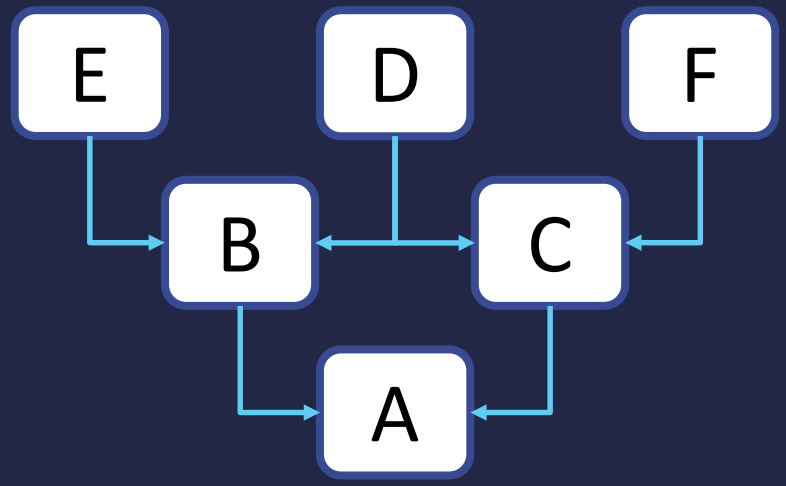
```
class F: pass
class E: pass
class D: pass
class C(D,F): pass
class B(D,E): pass
class A(B,C): pass

L[F] = (F, o)
L[E] = (E, o)       L[D] = (D, o)
L[C] = (C, merge((D, o), (F, o), (D, F))
# D is a good head, remove it from all lists and move it out
L[C] = (C, D, merge((o), (F, o), (F))
# o is not a good head, but F is
L[C] = (C, D, F, merge((o), (o))
# o is now a good head, and we're done
L[C] = (C, D, F, o)
```



C3 Linearization

```
class F: pass
class E: pass
class D: pass
class C(D,F): pass
class B(D,E): pass
class A(B,C): pass
```




```
L[F] = (F, o)
L[E] = (E, o)                    L[D] = (D, o)
L[C] = (C, D, F, o)              L[B] = (B, D, E, o)
L[A] = (A, merge((B, D, E, o), (C, D, F, o), (B, C))
L[A] = (A, B, merge((D, E, o), (C, D, F, o), (C))
L[A] = (A, B, merge((D, E, o), (C, D, F, o), (C))
L[A] = (A, B, C, merge((D, E, o), (D, F, o))
L[A] = (A, B, C, D, merge((E, o), (F, o))
L[A] = (A, B, C, D, E, merge((o), (F, o))
L[A] = (A, B, C, D, E, F, merge((o), (o))
L[A] = (A, B, C, D, E, F, o)
```

Ambiguous Hierarchy

```
class X: pass
class Y: pass
class A(X, Y): pass
class B(Y, X): pass
class C(A, B): pass

L[X] = (X, o)
L[Y] = (Y, o)
L[A] = (A, merge((X, o), (Y, o)))
L[A] = (A, X, Y, o)
L[B] = (B, Y, X, o)
L[C] = (C, merge((A, X, Y, o), (B, Y, X, o), (A, B))
L[C] = (C, A, merge((X, Y, o), (B, Y, X, o), (B))
L[C] = (C, A, B, merge((X, Y, o), (Y, X, o))

# Uh-oh, cannot continue. Neither X or Y are good heads
```