ECE326

PROGRAMMING LANGUAGES

Lecture 12 : Method Resolution Order

Kuei (Jack) Sun

ECE

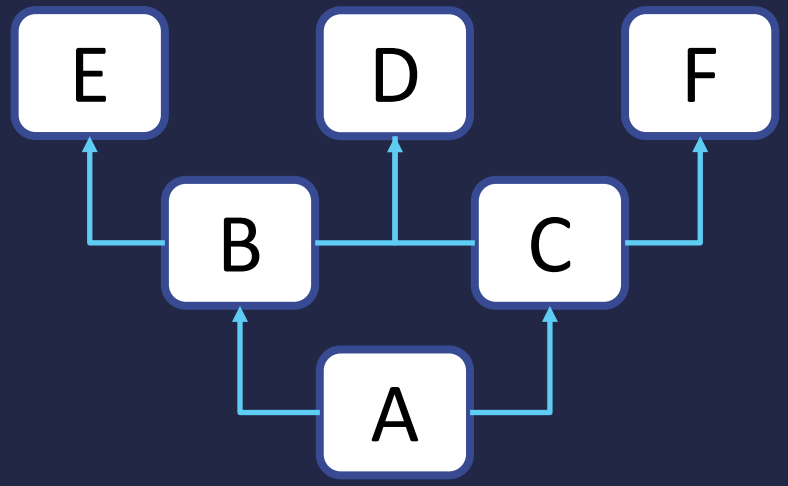
University of Toronto

Fall 2020

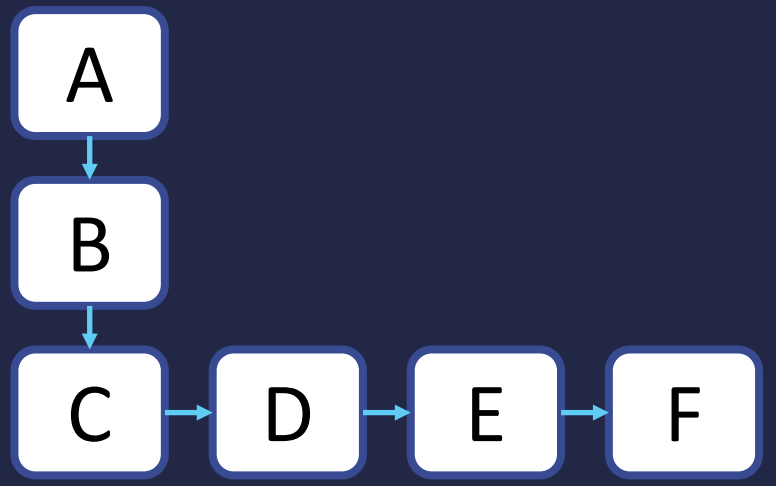
Method Resolution Order

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

Inheritance Hierarchy
(direct acyclic graph)



Method Resolution Order
(linearized)



Method Resolution Order

- Can have different implementation
  - Python uses C3 Linearization since Python 2.3

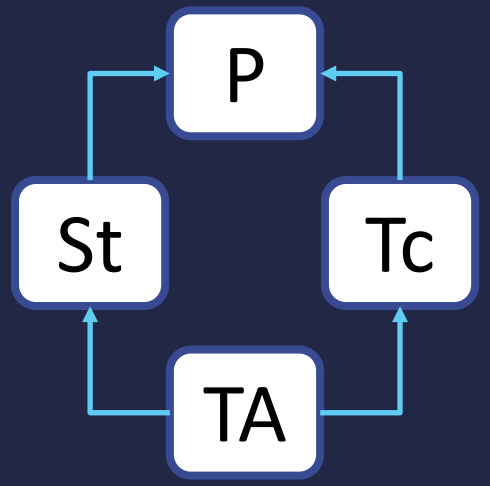
```
class Person:
    def vacation(self):
        return “Toronto Islands”

class Student(Person):
    pass

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

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

```
>> ta = TA()
>> 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 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
```

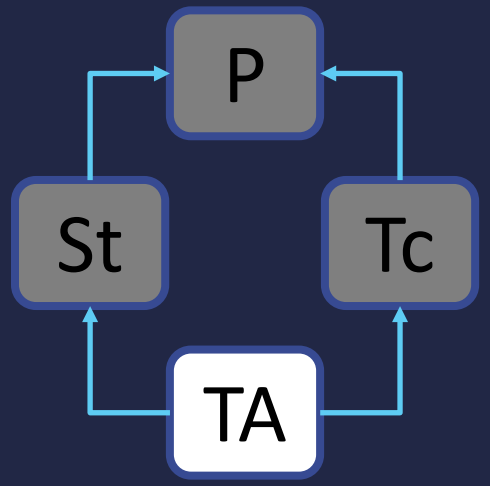


Depth First Search – Left to Right

- Naïve approach
  - Used by Python 2.1 and earlier

```
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
```

MRO: TA

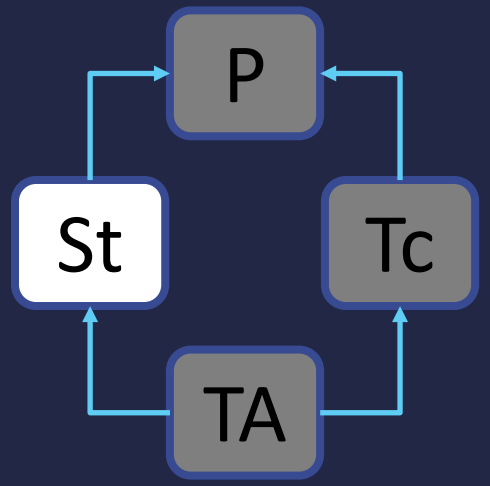


Depth First Search – Left to Right

- Naïve approach
  - Used by Python 2.1 and earlier

```
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
```

MRO: TA
Student

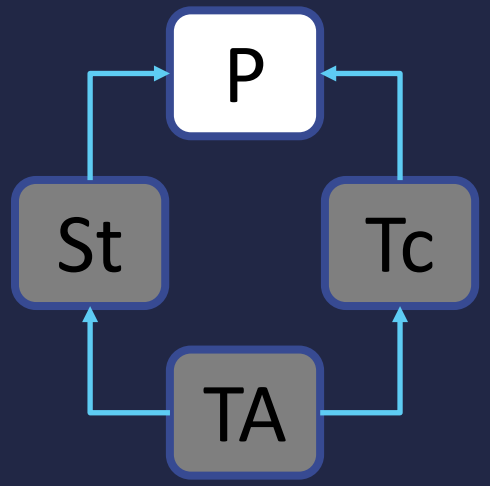


Depth First Search – Left to Right

- Naïve approach
  - Used by Python 2.1 and earlier

```
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
```

MRO: TA
Student
Person

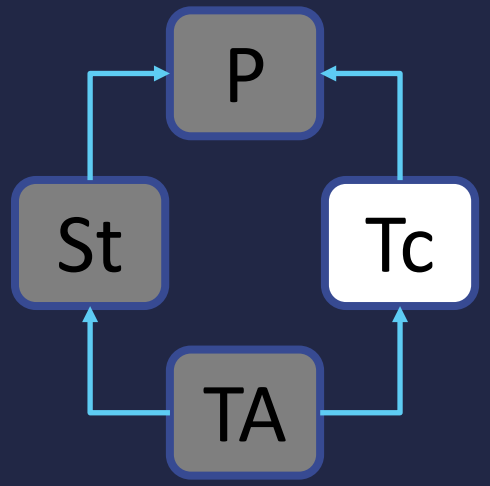


Depth First Search – Left to Right

- Naïve approach
  - Used by Python 2.1 and earlier

```
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
```

MRO:
TA
Student
Person
Teacher



Depth First Search – Left to Right

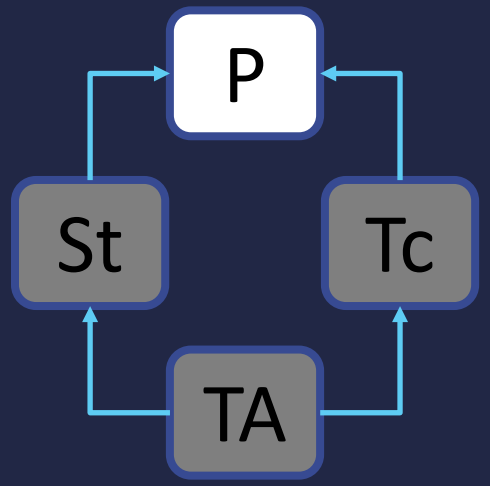
- Naïve approach
  - Used by Python 2.1 and earlier

```
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
```

Person already exists, so don’t add it again

MRO:

TA
Student
Person
Teacher
~~Person~~



Depth First Search – Left to Right

- Naïve approach
  - Used by Python 2.1 and earlier
- 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 earlier duplicate if it shows up again later

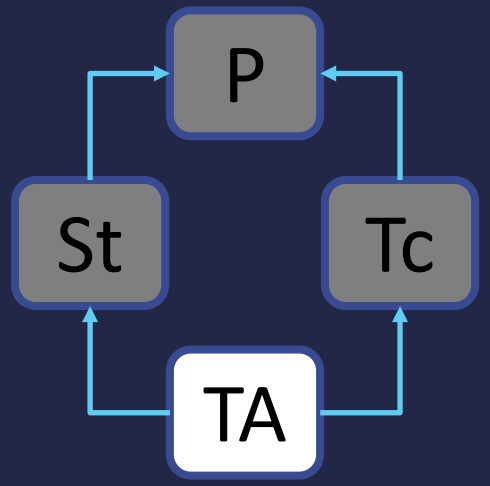
```
# pseudo code for new method resolution order
def mro_v2(cls):
    mro = [ cls ]
    for parent in cls.__bases__:
        for c in parent.__mro__:
            if c in mro:
                mro.remove(c)
        mro.extend(parent.__mro__)
    return mro
```

Refinement

- Python 2.2
  - DFS-LR, remove earlier duplicates

```
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
```

MRO: TA

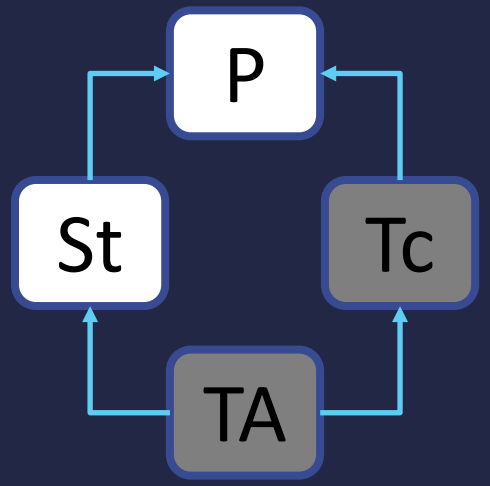


Refinement

- Python 2.2
  - DFS-LR, remove earlier duplicates

```
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
```

MRO: TA
Student
Person



Refinement

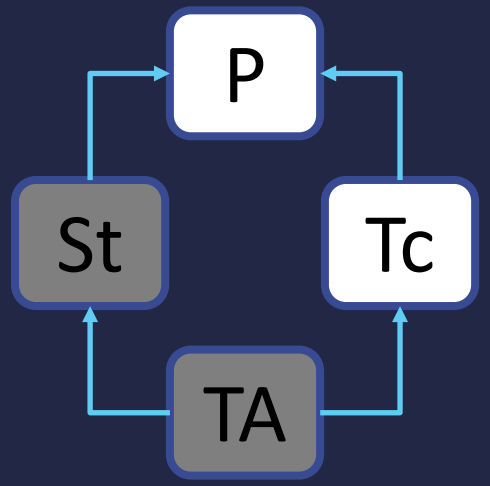
- Python 2.2
  - DFS-LR, remove earlier duplicates

```
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
```

Person already exists, so remove previous one

MRO:

TA
Student
~~Person~~
Teacher
Person



Refinement

- Python 2.2
  - Still uses depth-first search, left to right
  - Delete earlier duplicate if it shows up again later
- 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 accepted ambiguous hierarchy, which led to subtle bugs because the resolution order *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 monotonicity
- 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

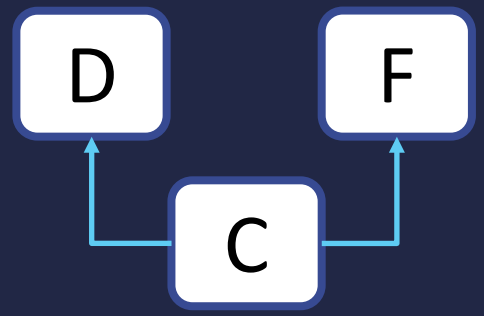
C3 Linearization

- 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

- The linearization of C is the sum of C plus the merge of the *linearizations of the parents* and the *list of the parents*.

```
class F: pass
class D: pass
class C(D,F): pass
```




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

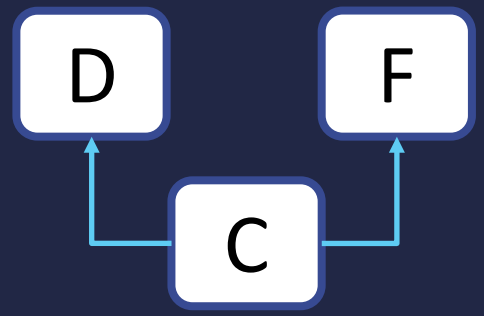
List of the parents

Linearizations of the parents, L[D] and L[F]

C3 Linearization

- The linearization of C is the sum of C plus the merge of the *linearizations of the parents* and the *list of the parents*.

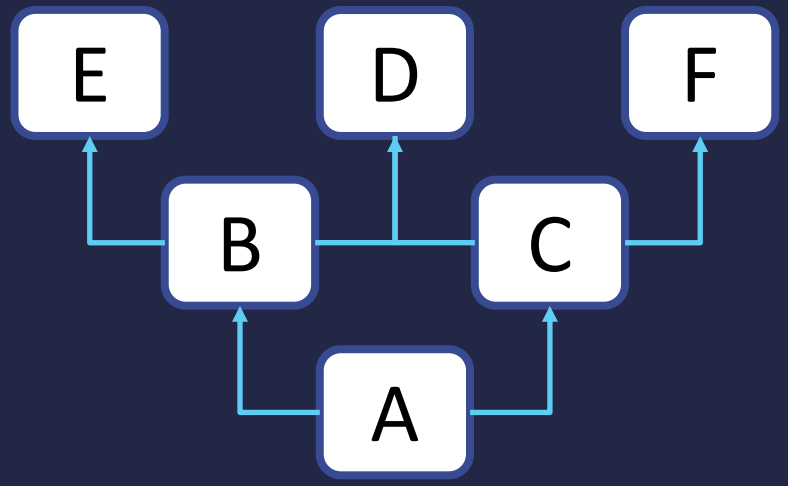
```
class F: pass
class D: pass
class C(D,F): pass
```




```
L[F] = (F, 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, 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), (X, Y)))
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
```