Multiple Inheritance

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 4

Recap

- Ranges & Slicing
- Iterators
- Protocols
  - Sized
  - Container
- Python Functions
  - Positional Arguments
  - Default Arguments
  - Named Arguments
  - Variable List Arguments

Unpacking

Unpacking

We've learned about unpacking using the * asterisk

There's another form of unpacking that occurs implicitly without using the operator

Unpacking

Try this code:

```
>>> item1,item2 = (1, 2)
```

```
>>> print(item1)
```

```
>>> print(item2)
```

Unpacking

Try this code:

```
>>> item1,item2 = (1, 2)
>>> print(item1)
1
>>> print(item2)
2
```

Dynamically assigning each element of a sequence to a corresponding variable is known as **unpacking**.

Unpacking

Example of unpacking dictionary

```
default_character_data = {
        'name' : "Untitled",
        'level' : 4,
        'max_level' : 10,
        'weapon' : "Sword",
        'Gender' : "Male"
    }
for a_key, an_item in default_character_data.items():
    print(f"{a_key} - {an_item}")
```

Unpacking

Example of unpacking dictionary

```
{'name'  : "Untitled",
'level'  : 4,
'max_level' : 10,
'weapon' : "Sword",
'Gender' : "Male“}
```

```
for a_key, an_item in default_character_data.items():
    print(f"{a_key} - {an_item}")
```

Variable Keyword Arguments

Variable Keyword Arguments

Named Arguments are also known as **Keyword Arguments**.

Just like we can accept an arbitrary variable number of arguments using **`*args`**

`*args` is treated like a sequence type

We can accept an arbitrary variable number of keyword arguments using **`**kwargs`**

`**kwargs` is treated like a dictionary

Recall: Named Arguments

```
def my_func(a_list, a_num, a_string):
    print(a_list, a_num, a_string)


my_func(a_num=5, a_string='hello', a_list=['cat','dog'])
```

The position of parameters that are passed in doesn’t matter with named arguments. All that matters is explicitly assigning a value to a matching parameter name

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):
    for key, item in kwargs.items():
        print(key, item)

variable_keyword_args(port=21, host='localhost', debug=True)
```

Looks like we're passing in named arguments

Also notice how this looks like a dictionary without the {}

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):
    for key, item in kwargs.items():
        print(key, item)

variable_keyword_args(port=21, host='localhost', debug=True)
```

There are keys

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):
    for key, item in kwargs.items():
        print(key, item)

variable_keyword_args(port=21, host='localhost', debug=True)
```

There are keys, and **values**

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):
    for key, item in kwargs.items():
        print(key, item)

variable_keyword_args(port=21, host='localhost', debug=True)
```

****kwargs** interprets the key/value parameters passed in as a dictionary

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):
    for key, item in kwargs.items():
        print(key, item)

variable_keyword_args(port=21, host='localhost', debug=True)
```

kwargs is a dictionary, so in order to get the key/value pairs, we need to explicitly call the **items method**

Notice how we're using *unpacking* to get the key/value pairs from the dictionary

Variable Keyword Arguments

```
def variable_keyword_args(**kwargs):
    for key, item in kwargs.items():
        print(key, item)

variable_keyword_args(port=21, host='localhost', debug=True)
```

- Each key/value pair can be extracted using **key**, **item** variables
- Note that key and item are NOT special names, they could be replaced by x, y and the code behaves the same

variable_keyword_arguments.py

Multiple Inheritance

Multiple Inheritance

A VERY sensitive subject.

When one child class inherits from or more base classes and is able to use functionality from all the base classes.

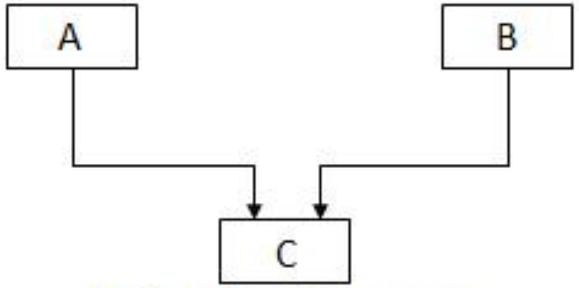
My Sagely Advice:

Avoid it if you can, there are alternatives and they are usually simpler to implement.

Many languages don’t support multiple Inheritance.

That being said, it is a feature and if it is needed, use it RESPONSIBLY. (Keep it simple and minimal)

Multiple Inheritance



Multiple Inheritance

```
Class C(A, B):
    # optional overridden functionality of A
    # optional overridden functionality of B
    # Implementation of C
    pass
```

All the rules of inheritance apply here.

Multiple inheritance is usually the tool of choice to use when implementing multiple interfaces (Remember the Interface Segregation Principle?)

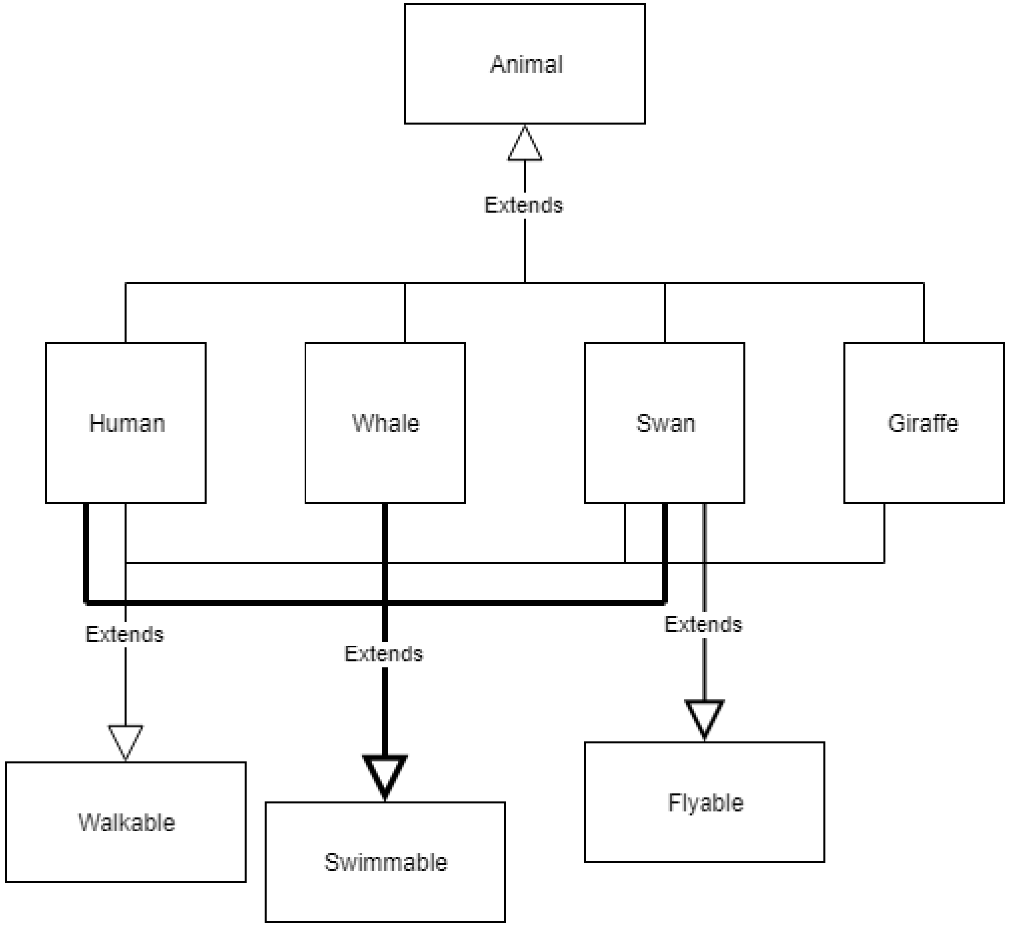
Rule of Thumb

As a rule of thumb, if you think you need multiple inheritance, you're probably wrong, but if you know you need it, you're probably right.

- Dusty Philips, Python3 Object-oriented Programming

RECAP: Interface Segregation Principle

```
class Walkable(abc.ABC):
    @abc.abstractmethod
    def walk(self):
        pass
class Swimmable
    @abc.abstractmethod
    def swim(self):
        pass
class Human(Animal, Walkable, Swimmable):
    def walk(self):
        # overridden walking code
    def swim(self):
        # overridden swimming code
```

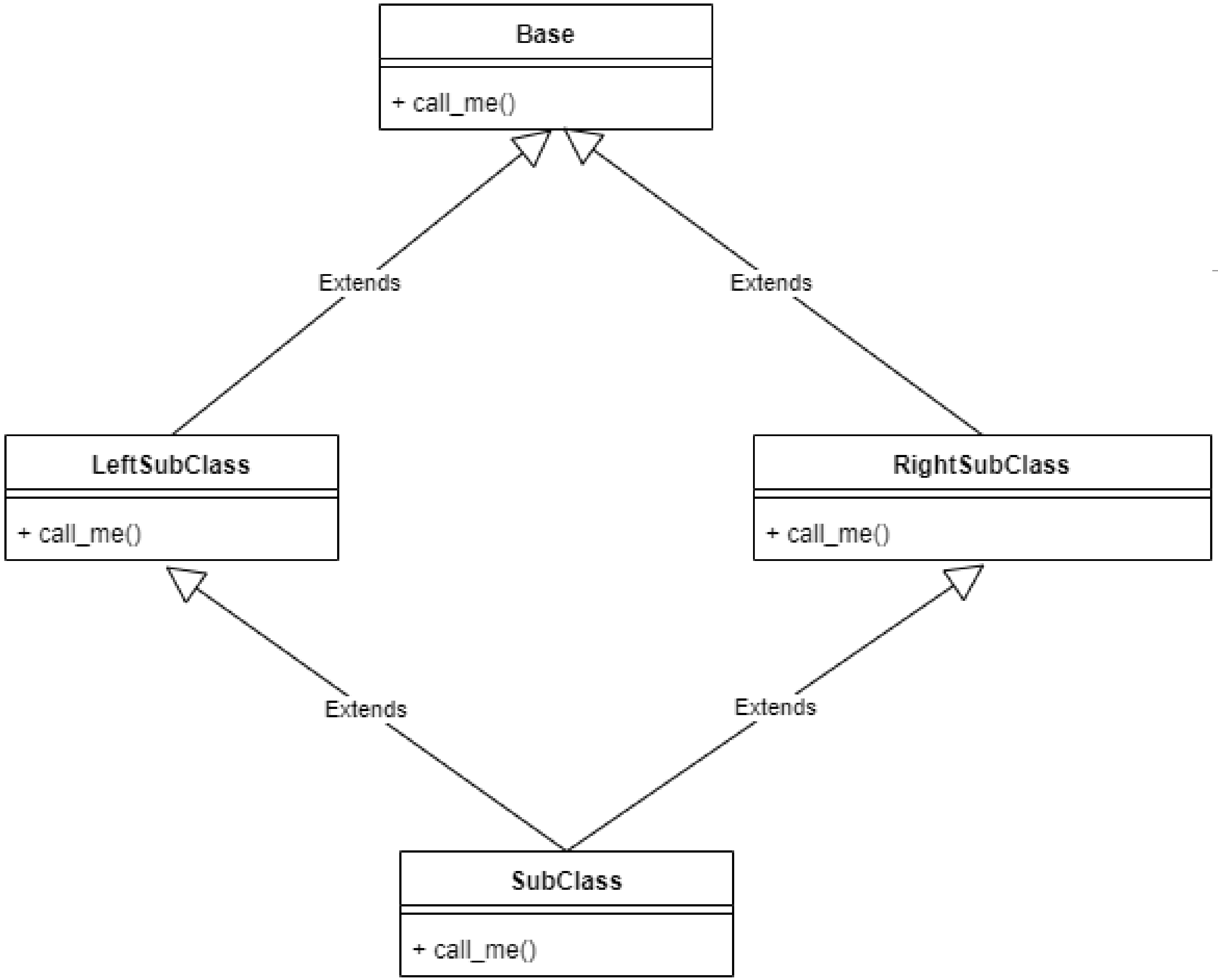


Instead of polluting the animal base class with additional functionality, we create separate interfaces (read: ABC) that handle different responsibilities.

This may seem difficult to maintain, but in fact it isn't. **This is only possible if the interfaces are well defined and have no overlap.**

The Diamond Problem

A SIMPLE TAKE AT WHY MULTIPLE INHERITANCE IS PROBLEMATIC

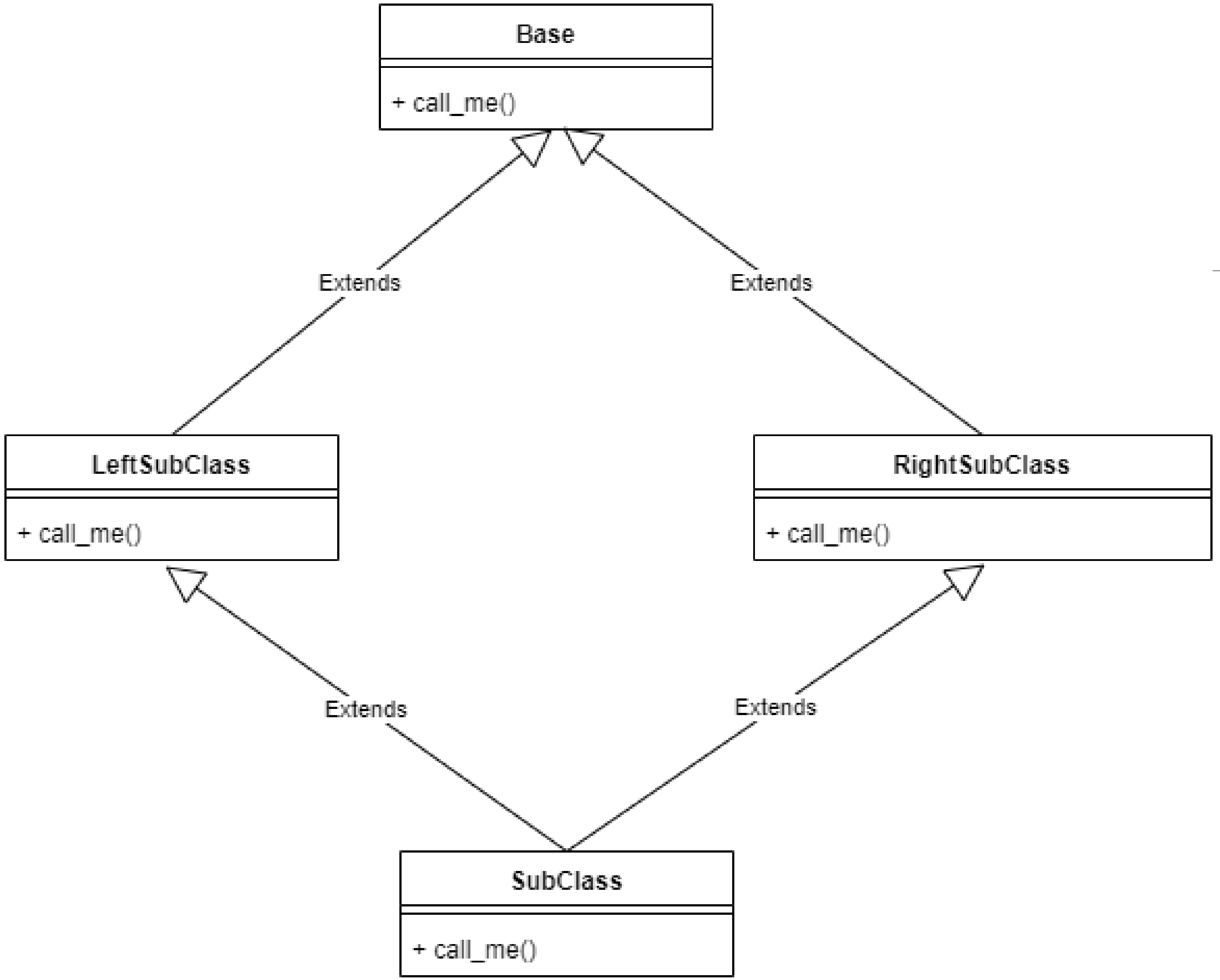


The Diamond Problem

At first glance this inheritance looks fine

Subclass inherits from 2 parents (LeftSubClass and RightSubClass). And Subclass also has a grandparent Base.

But if we look a little closer, some questions start to rise

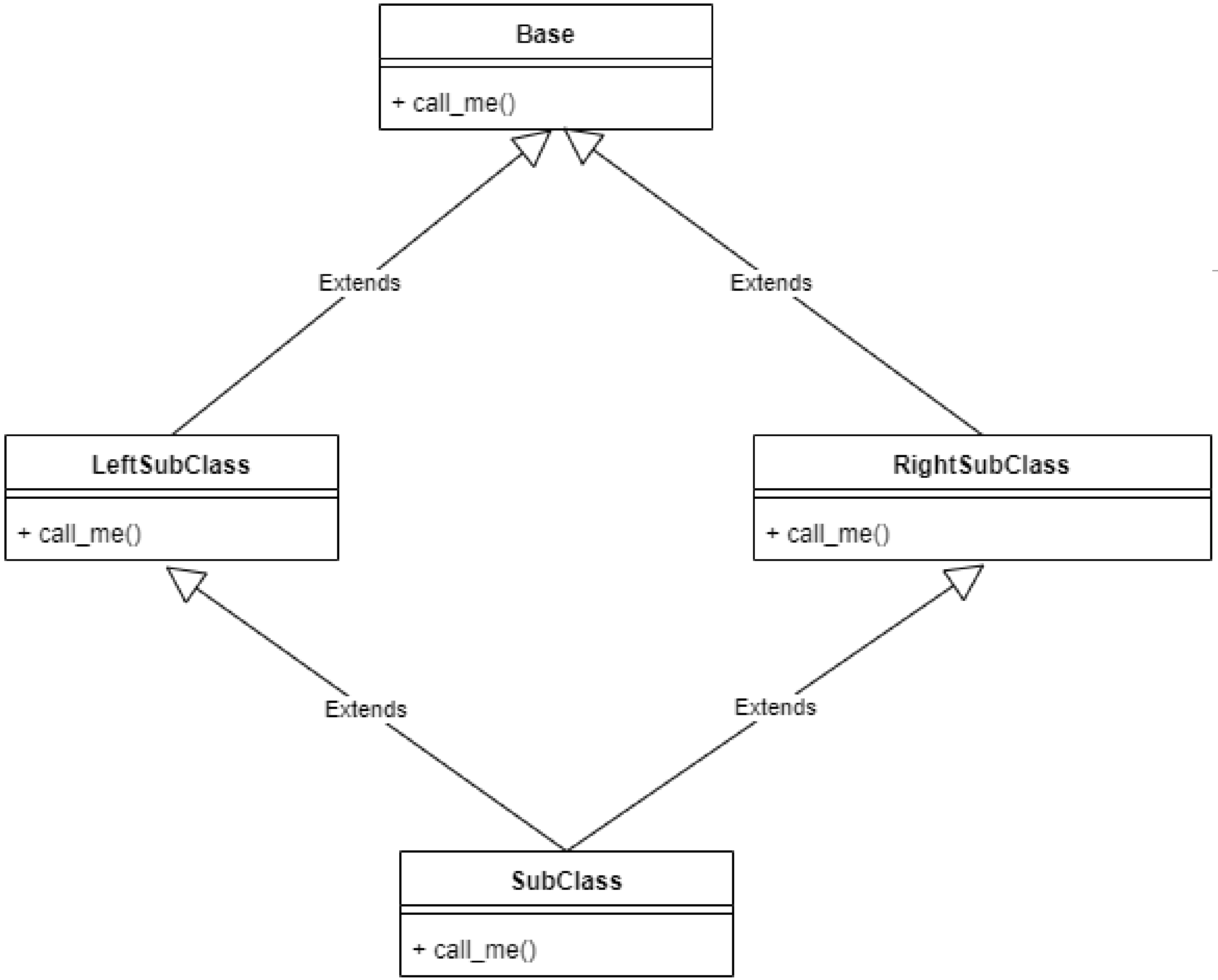


The Diamond Problem

SubClass inherits from LeftSubClass, which means it also inherits from Base

SubClass inherits from RightSubClass, which means it also inherits from a second Base?

Does that mean SubClass has two instances of Base as grandparents?



The Diamond Problem

One of the trickiest aspects of Multiple Inheritance is to determine the order in which methods are executed

All of the classes have a call_me() function

If SubClass invokes call_me() on its parent, which call_me() function is called?

- Is call_me() in LeftSubClass called?
- Is call_me() in RightSubClass called?
- Does it call both?
- Is call_me() in grandparent Base called?

The Diamond Problem

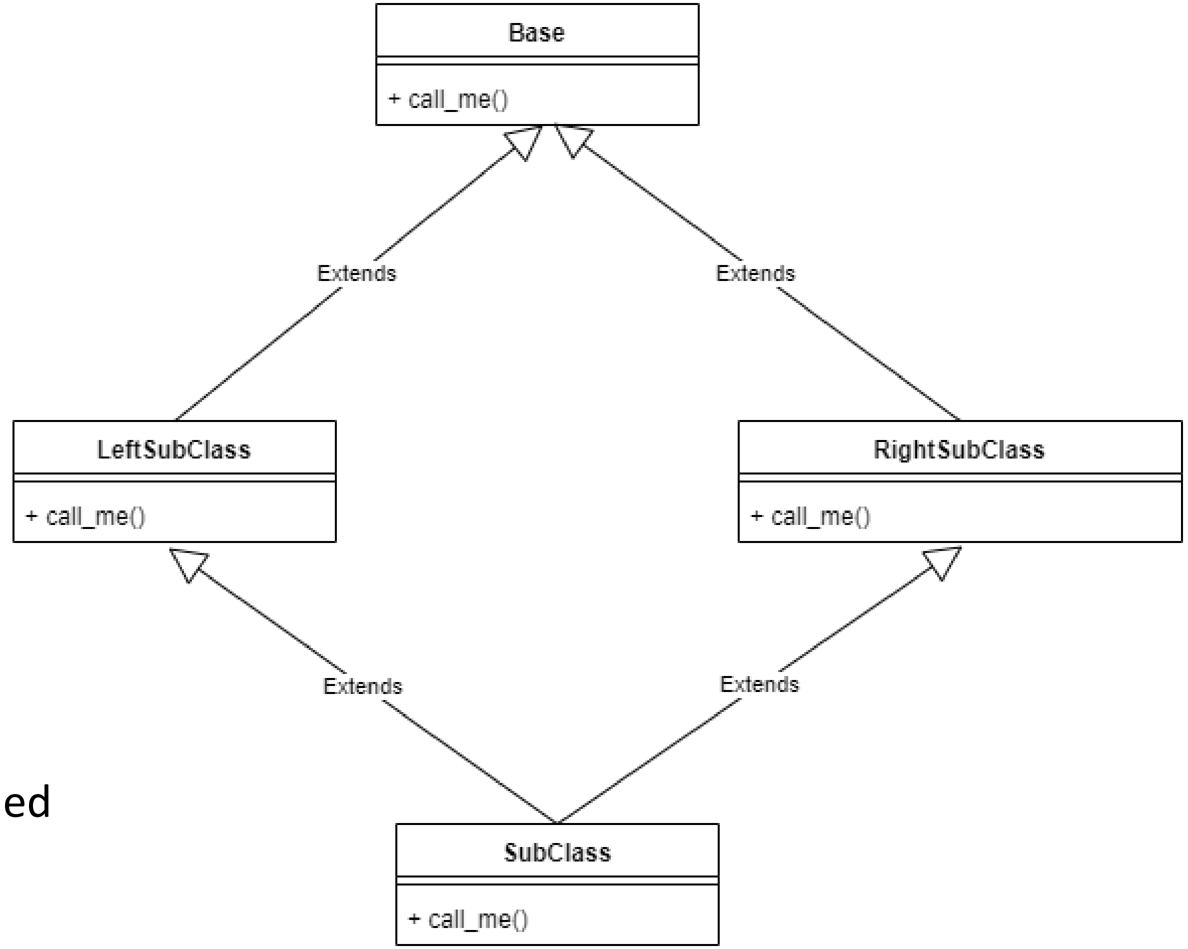
If we call the call_me method of a SubClass, in what order do we call the call_me methods of its super classes?

Let's take a look at some code!

multipleinheritance_diamond_problem.py

Notice in our code how it appears as though there are two Base grandparent classes if we call the parent's call_me() method directly?

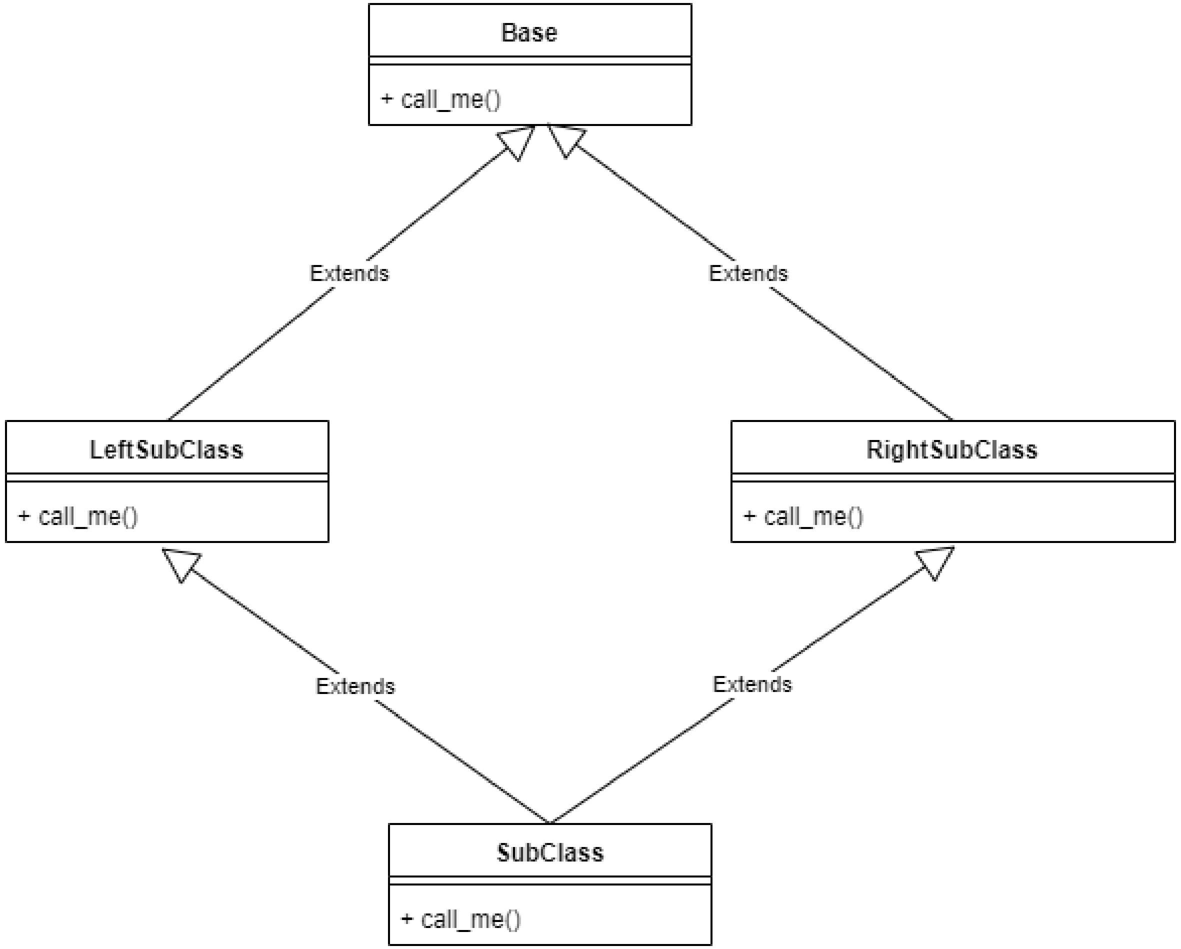
We still don't know which parent's call_me() function is called



The Diamond Problem

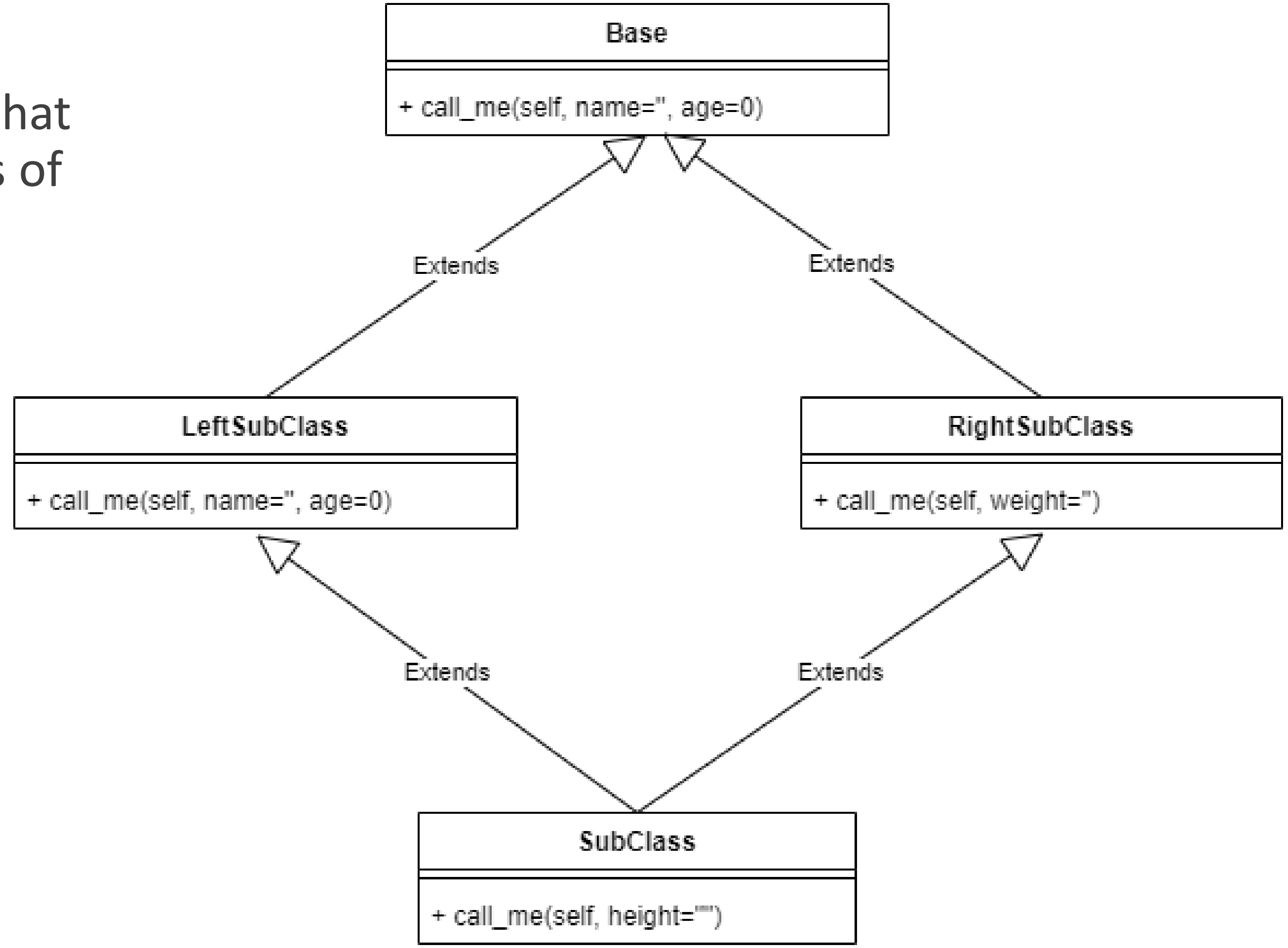
The **super()** keyword executes the methods in the parent class by first resolving the **Method Resolution Order**! (More on that in a bit)

multipleinheritance_diamond_solution.py



The Diamond Problem

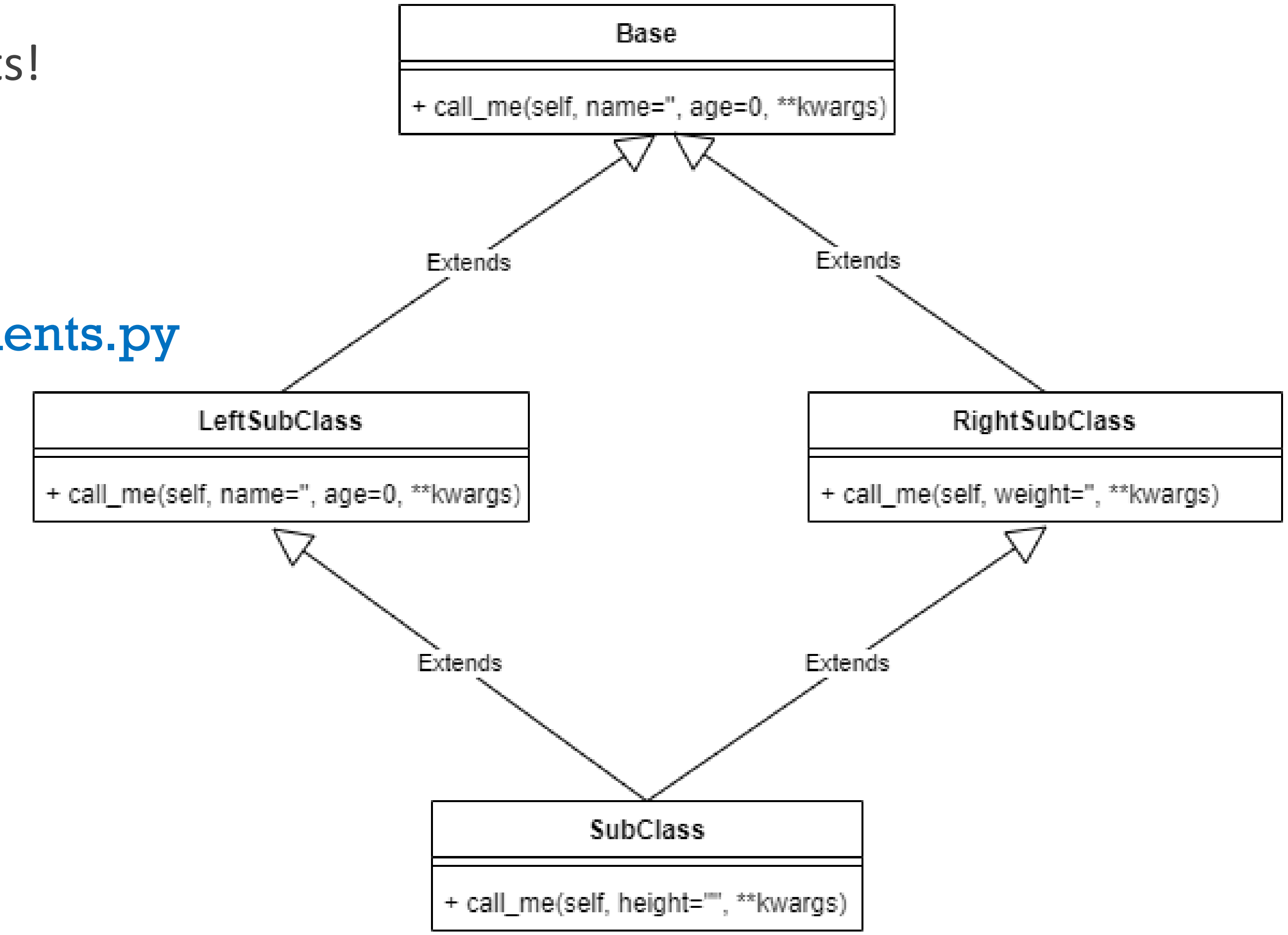
Great we figured out how to get around the problem of which method is called first. But what if all the call_me() methods took different sets of arguments?



The Diamond Problem

<u>The Answer:</u> Variable Keyword Argument Lists! We use the power of method overloading in python to pass a dictionary of a number of different argument types.

multipleinheritance_super_different_arguments.py



Method Resolution Order

A SIMPLE TAKE AT WHY MULTIPLE INHERITANCE IS PROBLEMATIC

Method Resolution Order

As we’ve seen in the previous example, finding which parent comes first in multiple inheritance is tricky.

Given a class C in a **complicated multiple inheritance** hierarchy, it’s difficult to determine the order that methods are overridden

- Ie: to specify order of the ancestors of C

The list of ancestors of class C, including the class itself, ordered from nearest ancestor to furthest is called the **class precedence list** or **the linearization of C**

https://www.python.org/download/releases/2.3/mro/

Method Resolution Order

With single inheritance, if C is a subclass of B, and B is a subclass of A, then the **linearization of class C is [C, B, A, Object]**

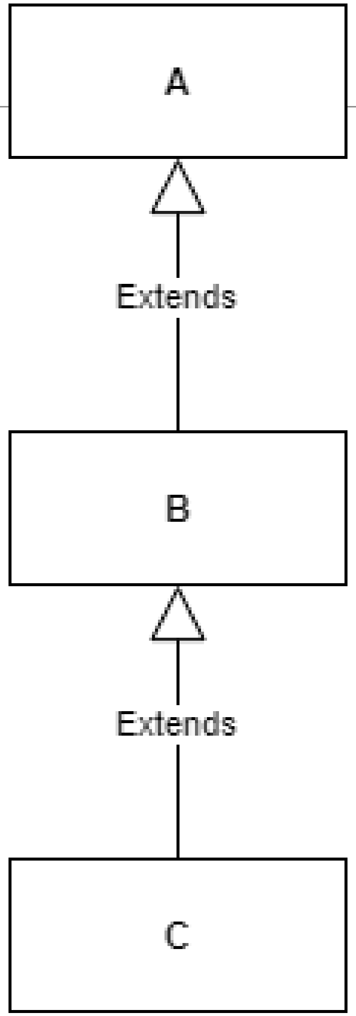
Linearizing single inheritance is easy, however with multiple inheritance, it gets more difficult

It's more difficult to create a linearization that respects **local precedence ordering** and **monotonicity** (more on this soon!)

The **Method Resolution Order** (MRO) is the set of rules that we follow to construct this linearization

In Python, "MRO of C" and "linearization of class C" are synonymous

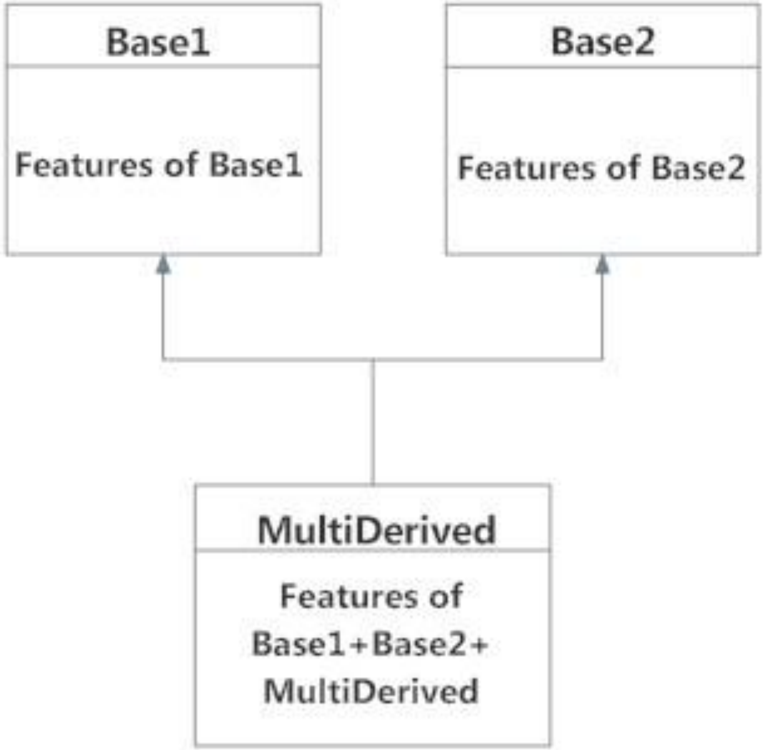
- The MRO of C in this case is [C, B, A, Object]



https://www.python.org/download/releases/2.3/mro/

Method Resolution Order

```
# Try this code:
class Base1:
    pass
class Base2:
    pass
class MultiDerived (Base1, Base2):
    pass

print(MultiDerived.mro()) # or MultiDerived.__mro__
```



[<class '__main__.**MultiDerived**'>, <class '__main__.**Base1**'>, <class '__main__.**Base2**'>, <class **'object'**>]

MRO: Rules

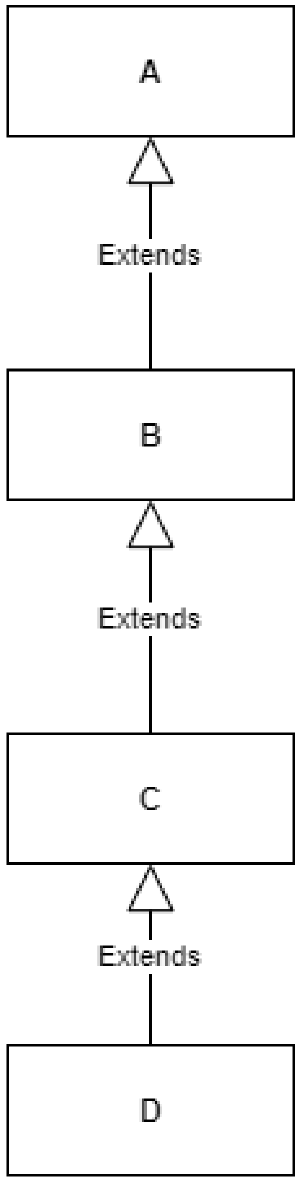
MRO must respect **local precedence ordering** and also provide **monotonicity**.

It ensures that a class always appears before its parents

Monotonicity:

If B Precedes A in the Linearization of C, then B **HAS TO** precede A in the Linearization of any child class of C.

Linearization(C) = [C, **B, A**, object]
Linearization(D) = [D, C, **B, A**, object]



MRO: Rules

Local Precedence ordering:

The order in a class that we're inheriting from multiple parents

This means that a class C(B1, …, Bn) will have in its MRO, the base classes (B1, …, Bn) in the order that we specified

```
class F:
    remember2buy = "Milk"
class E(F):
    remember2buy = "Eggs"
class G(F,E):
    pass
```

Local precedence of:

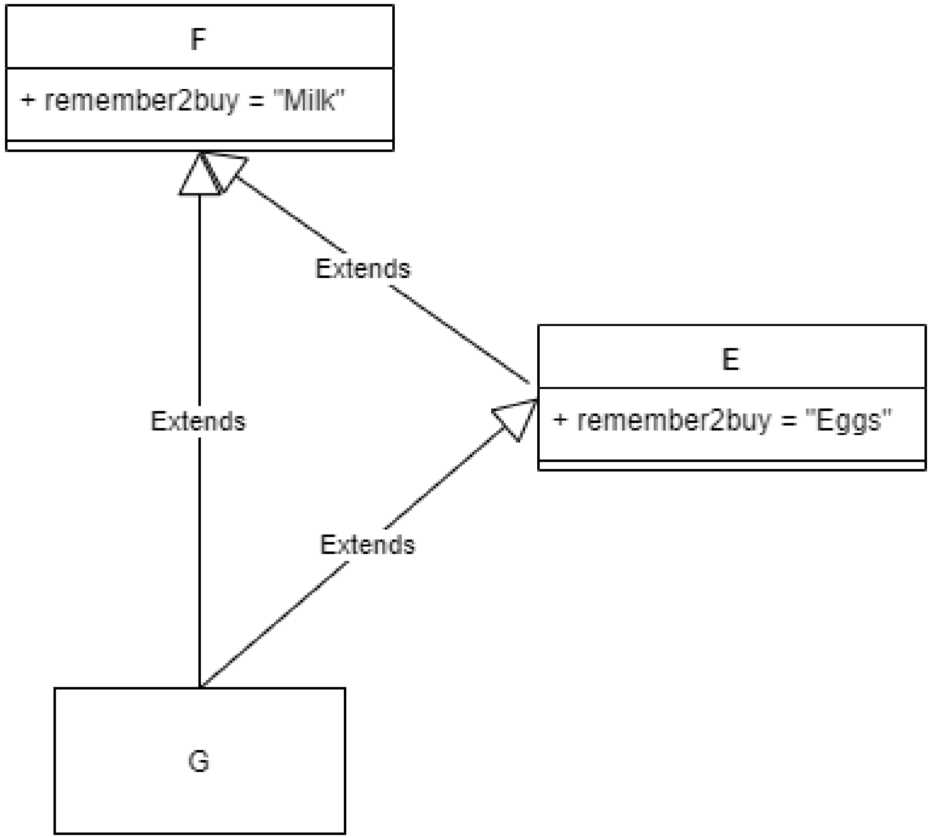
- class F is the implicit (object)
- class E is (F)
- class G is (F, E) #we'll see why this local precedence order is bad soon

MRO: Rules

Local Precedence ordering:

Let’s see how this looks visually

```
class F:
    remember2buy = “Milk”

class E(F):
    remember2buy = “Eggs”

class G(F,E):
    pass
```



MRO: Rules

As per the class G(F,E) declaration we evaluate in the following order:

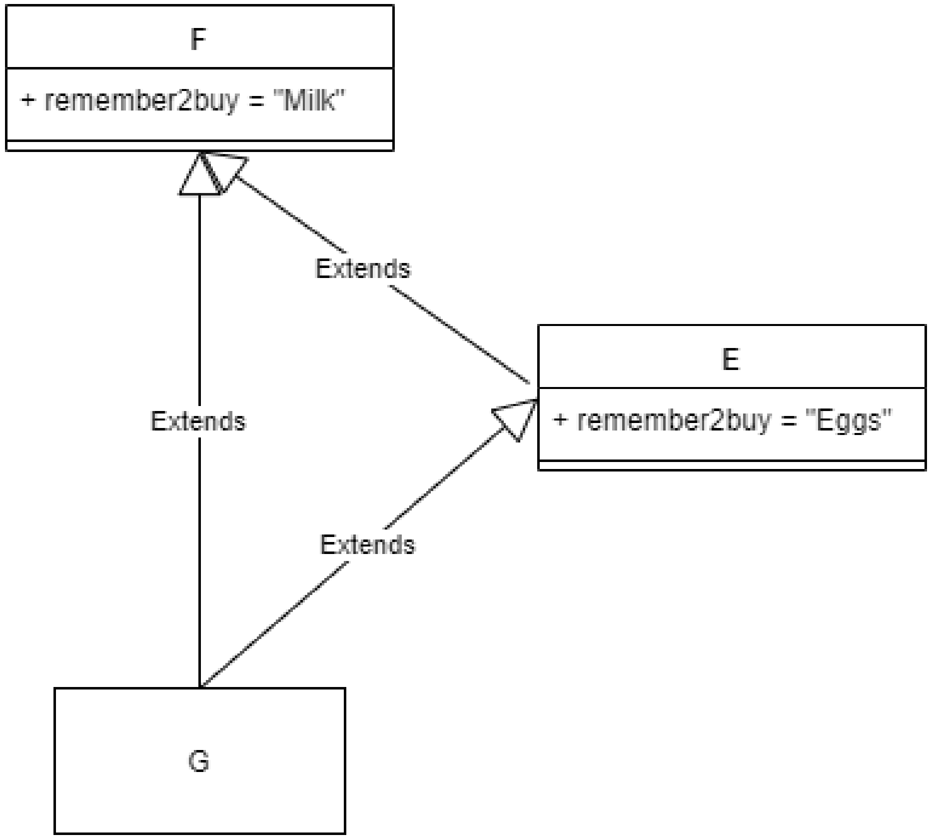
G – Evaluate G, F, and E

F – Evaluate F, Object

E – Evaluate E, F , This causes F to be evaluated again! Not a valid inheritance hierarchy! Python3 will throw a:

`TypeError: Cannot create a consistent method resolution order (MRO) for bases X, Y`

```
class F:
    remember2buy = “Milk”
class E(F):
    remember2buy = “Eggs”
class G(F,E):
    pass
```

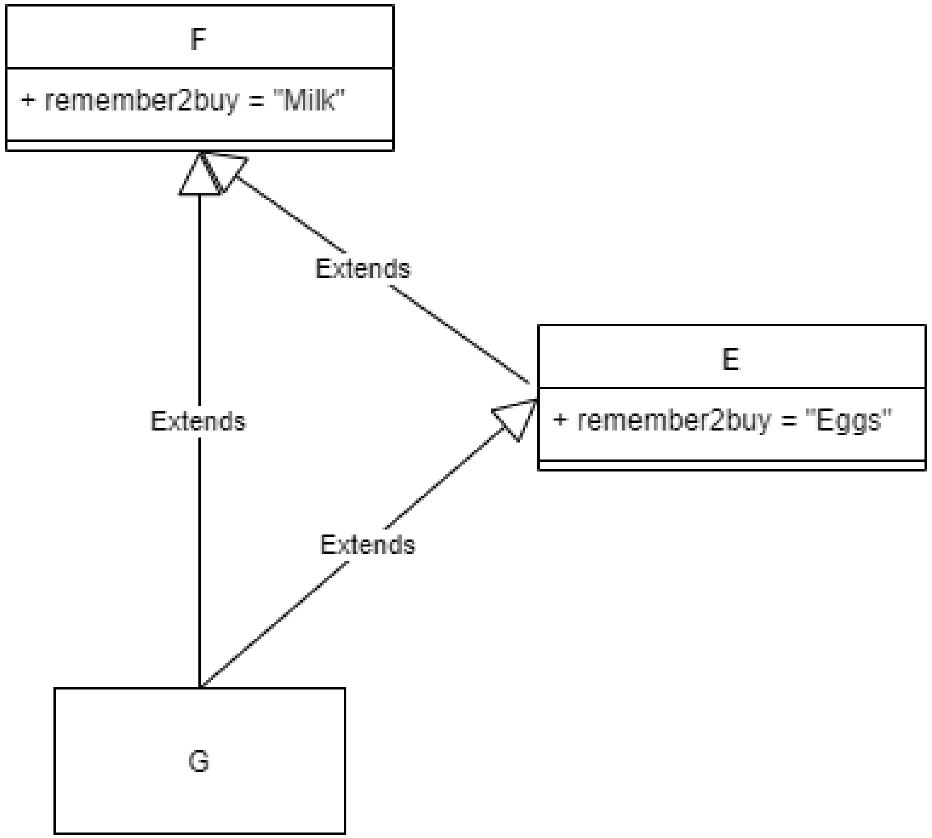


MRO: Rules

Linearization:

G = [G, F, E, F, object] ← Fails monotonicity. F is a parent of E, it can not appear before E

```
class F:
    remember2buy = "Milk"

class E(F):
    remember2buy = "Eggs"

class G(F,E):
    pass
```



MRO: Rules - Solution

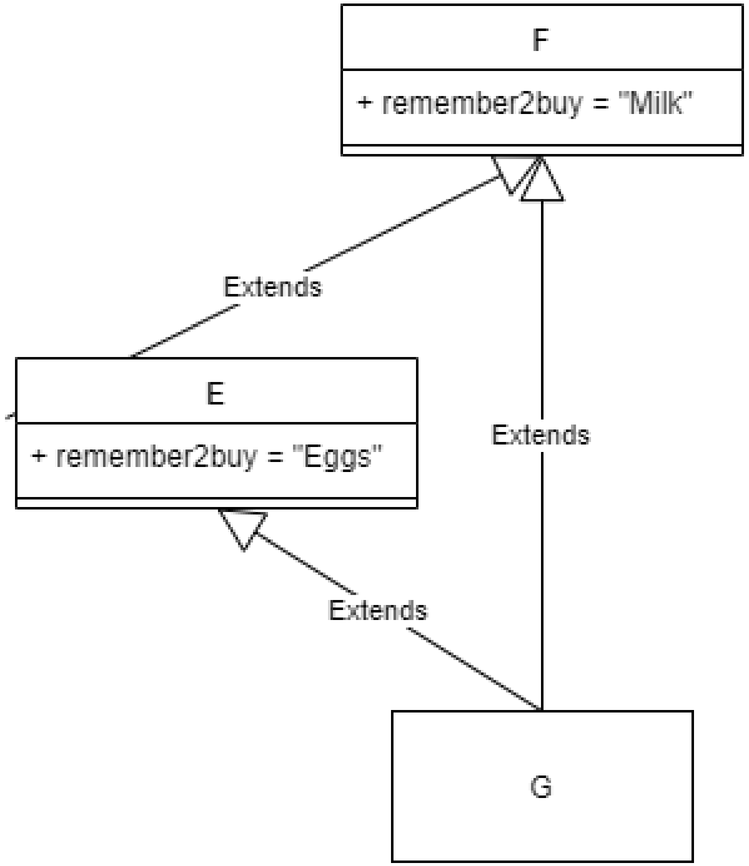
We change the order in which class G inherits E and F. According to the new class G(E, F) declaration, we evaluate in the following order:

G – Evaluate G, E, and F

E – Evaluate E, F

F – Evaluate F , Object

```
class F:
    remember2buy = "Milk"

class E(F):
    remember2buy = "Eggs"

class G(E,F):
    pass
```

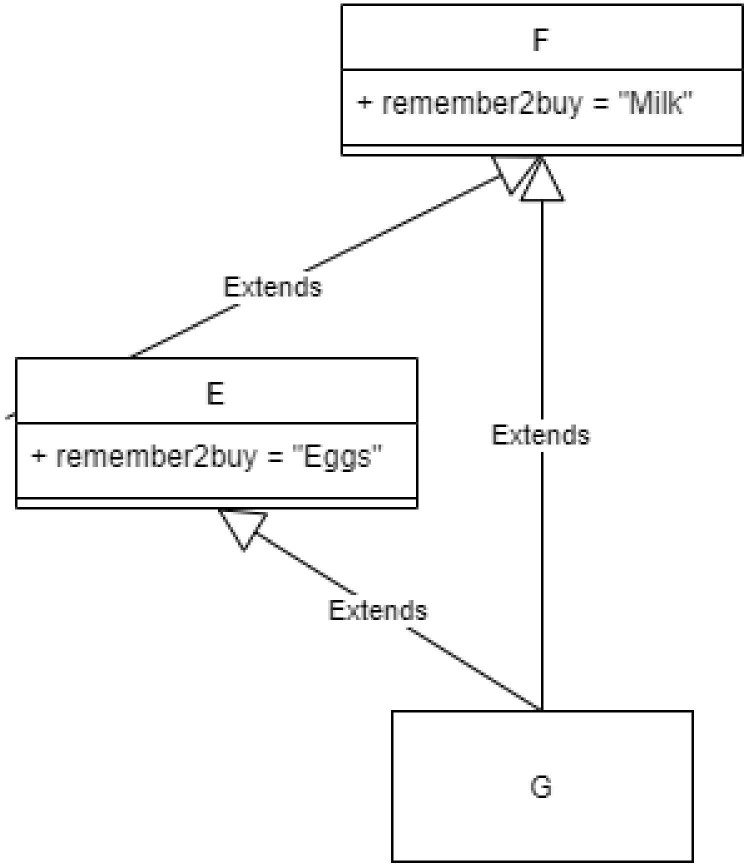


MRO: Rules - Solution

Solution:

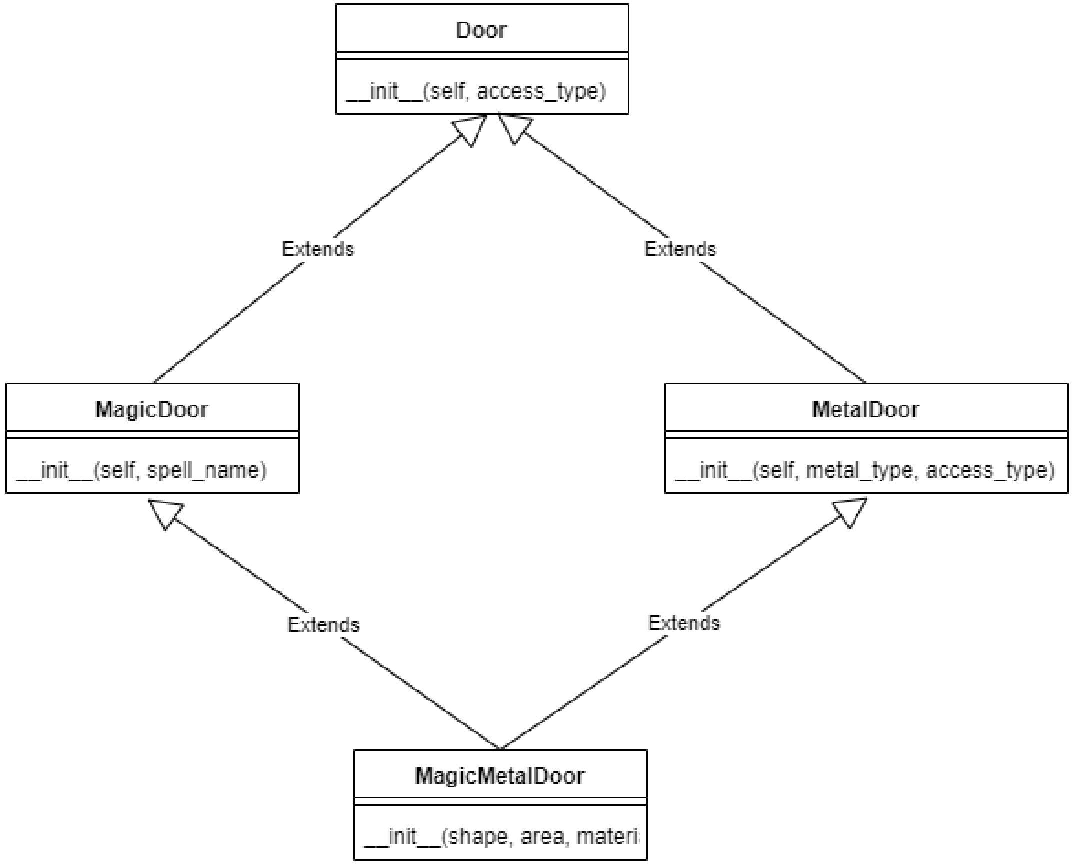
Linearization(G) = [G, E, F, object]

```
class F:
    remember2buy = “Milk”

class E(F):
    remember2buy = “Eggs”

class G(E, F):
    pass
```



MRO: What would the MRO be?

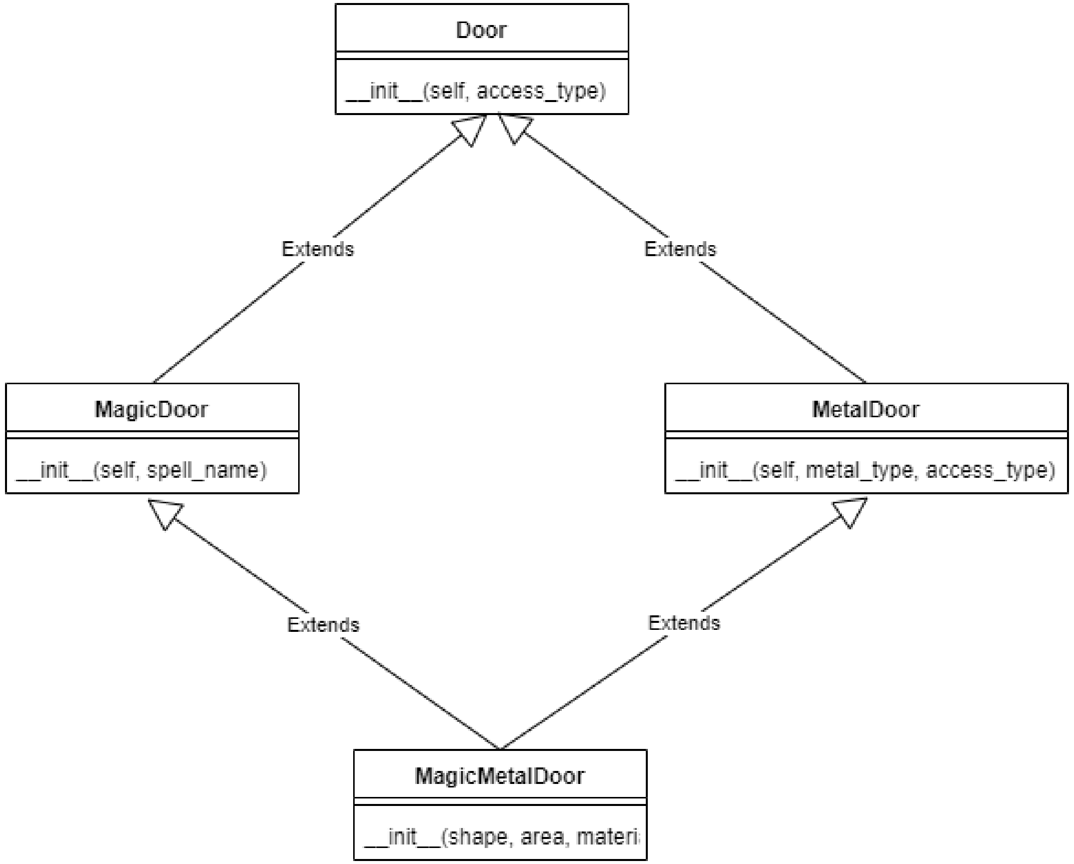
What would the MRO of
MagicMetalDoor(MagicDoor, MetalDoor)
Be?



MRO: What would the MRO be?

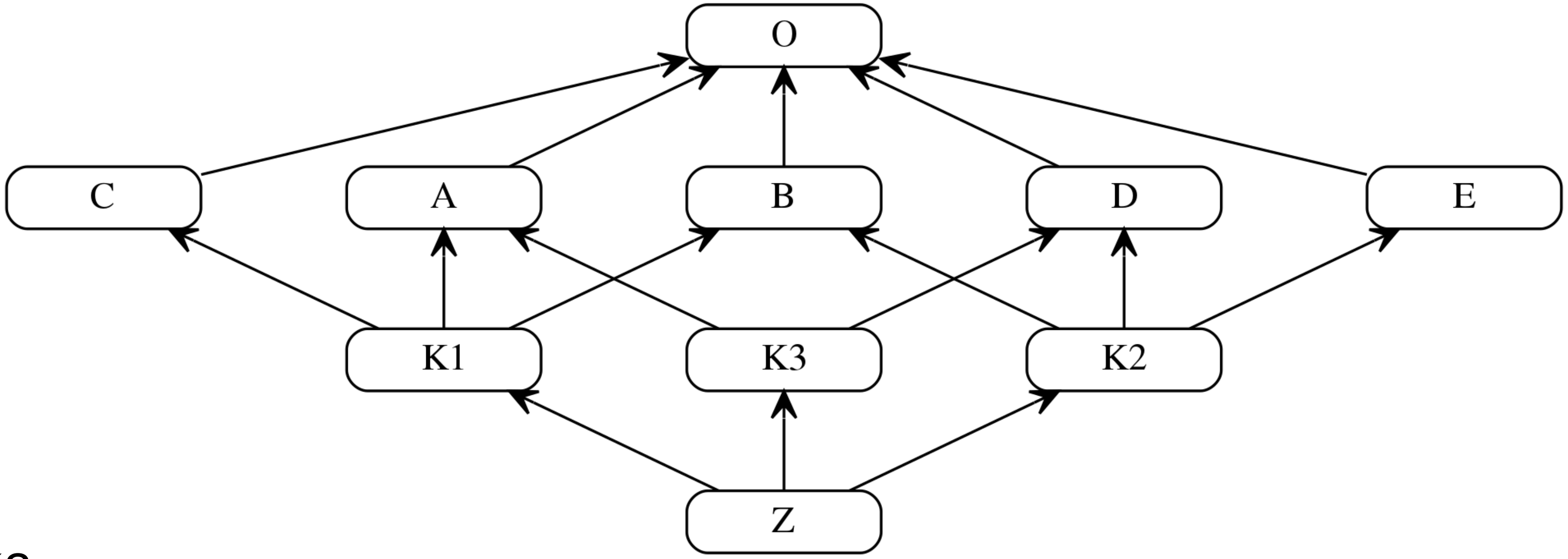
What would the MRO of
MagicMetalDoor(MagicDoor, MetalDoor)
Be?

Linearization(MagicMetalDoor)
= [MagicMetalDoor,
MagicDoor,
MetalDoor,
Door,
Object]

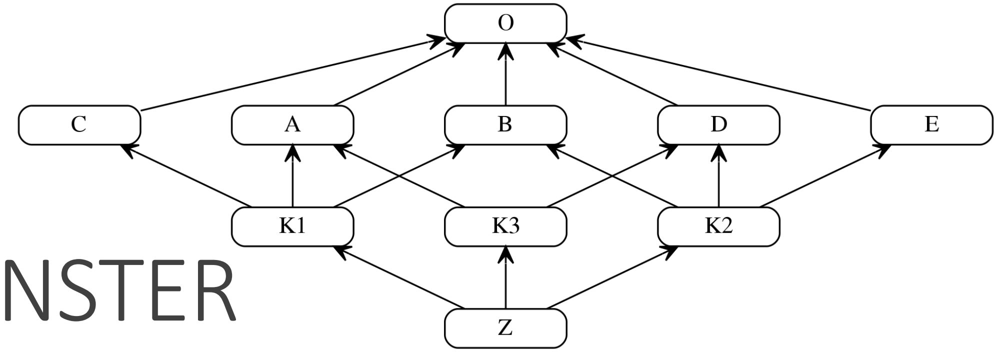


Try finding out the MRO of this MONSTER

class O
class A extends O
class B extends O
class C extends O
class D extends O
class E extends O
class K1 extends A, B, C
class K2 extends D, B, E
class K3 extends D, A
class Z extends K1, K2, K3



https://en.wikipedia.org/wiki/C3_linearization



Try finding out the MRO of this MONSTER

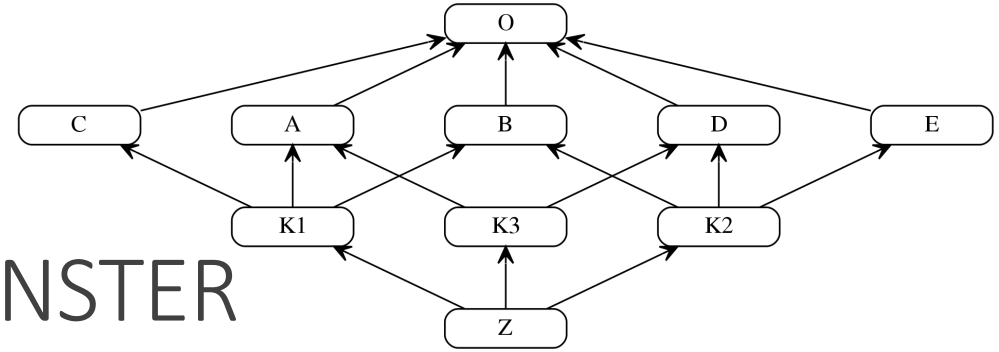
L(O) = [O]
L(A) = [A,O]
L(B) = [B,O]
L(C) = [C,O]
L(D) = [D,O]
L(E) = [E,O]
L(K1) = [K1, A, B, C, D]
L(K2) = [K2, D, B, E, O]
L(K3) = [K3, D, A, O]

class Z extends K1, K2, K3 #original python class code

L(Z) = [Z] + merge(L(K1), L(K2), L(K3), [K1, K2, K3])

What this is saying in regular English is "To find the linearization of Z, we must start with the list Z and merge it with the linearization of K1, K2, K3, along with the immediate parents K1, K2, K3"

https://en.wikipedia.org/wiki/C3_linearization



Try finding out the MRO of this MONSTER

L(O) = [O]
L(A) = [A,O]
L(B) = [B,O]
L(C) = [C,O]
L(D) = [D,O]
L(E) = [E,O]
L(K1) = [K1, A, B, C, D]
L(K2) = [K2, D, B, E, O]
L(K3) = [K3, D, A, O]

L(Z) = [Z] + merge(L(K1), L(K2), L(K3), [K1, K2, K3])

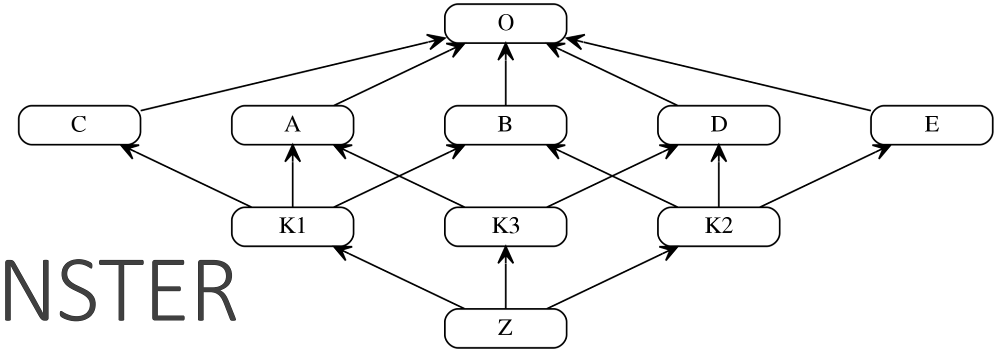
L(Z) – this is our notation for indicating the **L**inearization of **(Z)**
[Z] – this is the list we'll be adding to as we merge classes. Notice how we start with the class Z itself in the list

L(K1), L(K2), L(K3) – these are the linearizations of K1, K2, K3
[K1, K2, K3] – these are the immediate parents of Z

Notice how we must include the linearization of any parents along with the parents themselves

https://en.wikipedia.org/wiki/C3_linearization

Try finding out the MRO of this MONSTER

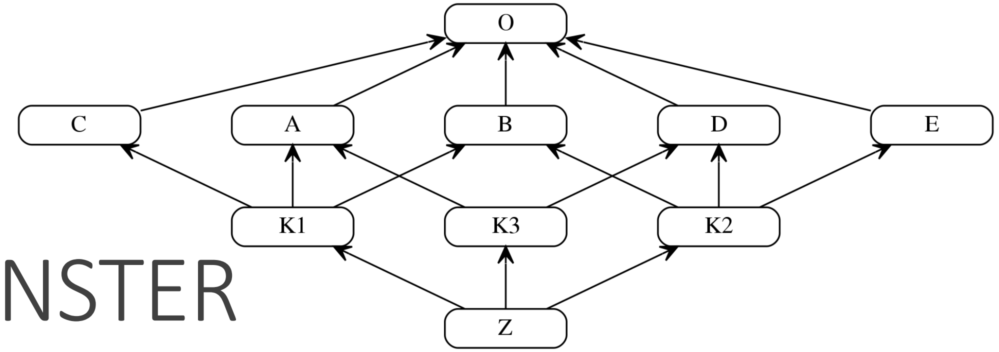


L(O) = [O]
L(A) = [A,O]
L(B) = [B,O]
L(C) = [C,O]
L(D) = [D,O]
L(E) = [E,O]
L(K1) = [K1, A, B, C, D]
L(K2) = [K2, D, B, E, O]
L(K3) = [K3, D, A, O]

L(Z) = [Z] + merge(L(K1), L(K2), L(K3), [K1, K2, K3])

Step 1: Expand the linearization of the parents
Notice the reference of the previous linearizations on the left?
L(K1), L(K2), L(K3) are already done for us, so let's put them in our equation

https://en.wikipedia.org/wiki/C3_linearization



Try finding out the MRO of this MONSTER

L[Z] = [Z] + merge([K1, A, B, C, O], [K2, D, B, E, O], [K3, D, A, O], [K1, K2, K3])

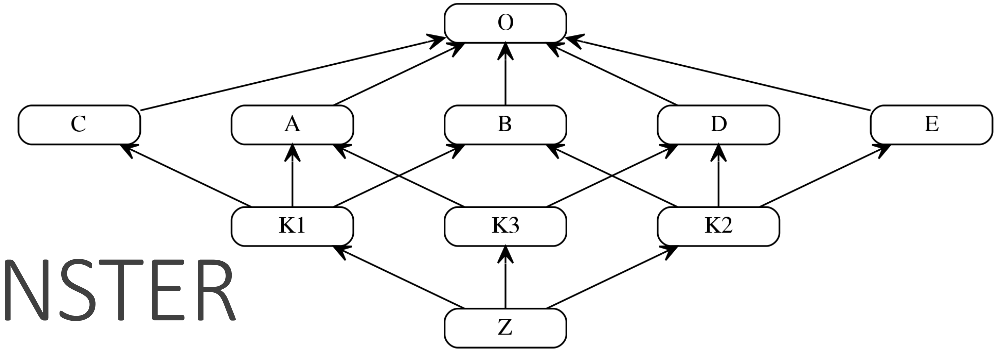
Step 2: Begin merging
Before merging we need to follow an algorithm
There's a concept of a head and tail in each class list

[K1, A, B, C, O]

Head is the first element in the list [K1
Tail is everything after the head A, B, C, O]

https://en.wikipedia.org/wiki/C3_linearization

Try finding out the MRO of this MONSTER



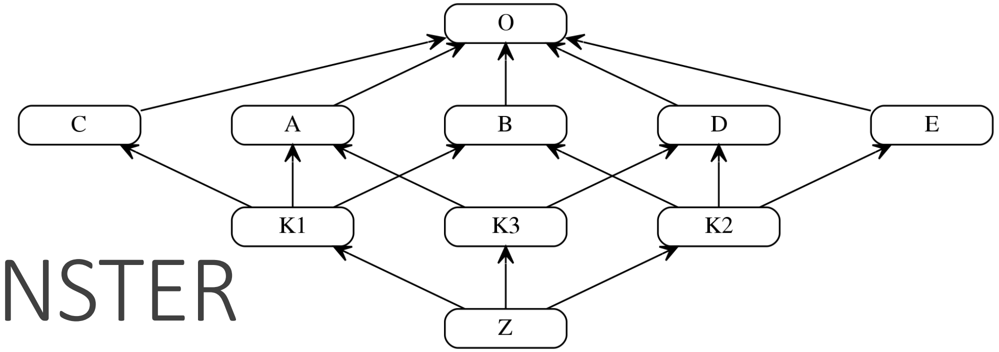
L[Z] = [Z] + merge([**K1**, A, B, C, O], [K2, D, B, E, O], [K3, D, A, O], [K1, K2, K3])

Step 2: Begin merging

The algorithm is as follows:

- Pick the head ***(K1)*** of the list, beginning with the first list *([K1, A, B, C, O])*
- If this head does not exist in the **tail** of any other list, add it to the linearization [**Z**] and remove it from all the other lists *([K2, D, B, E, O])*
- If this head DOES exist in the **tail** of any other list, pick the head of the next list and repeat the previous check
- If we go through all the lists and can't find a head that doesn't exist in a tail, MRO fails

https://en.wikipedia.org/wiki/C3_linearization

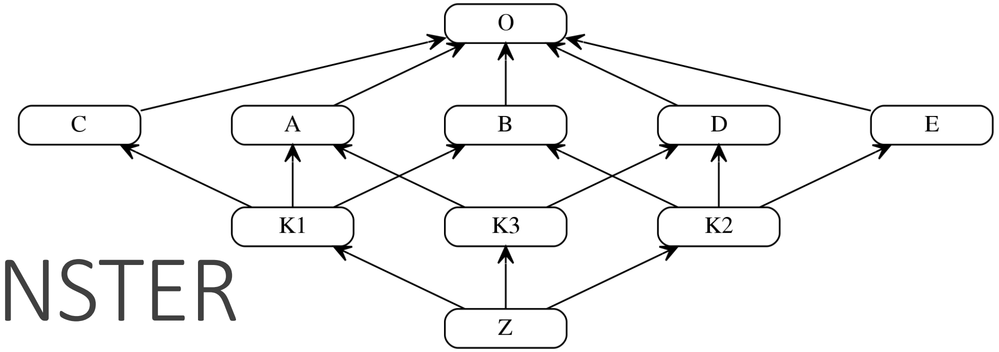


Try finding out the MRO of this MONSTER

L[Z] = [Z] + merge([**K1**, A, B, C, O], [K2, D, B, E, O], [K3, D, A, O], [K1, K2, K3])

Step 2: Begin merging
Following the algorithm, we pick K1 as the head
K1 does not exist in any of the other lists' tails, so we add it to the list [Z]

https://en.wikipedia.org/wiki/C3_linearization

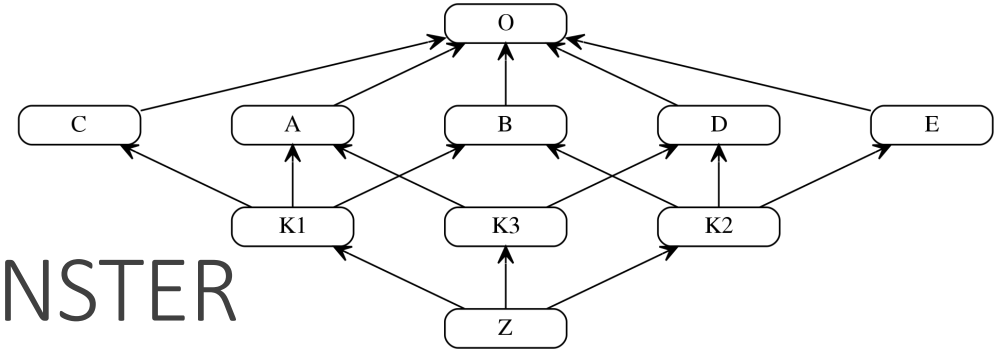


Try finding out the MRO of this MONSTER

L[Z] = [Z, K1] + merge([~~K1~~, A, B, C, O], [K2, D, B, E, O], [K3, D, A, O], [~~K1~~, K2, K3])

Step 2: Begin merging
Following the algorithm, we pick K1 as the head
K1 does not exist in any of the other lists’ tails, so we add it to the list [Z]
After adding K1 to [Z] we need to remove K1 from all the other lists

https://en.wikipedia.org/wiki/C3_linearization



Try finding out the MRO of this MONSTER

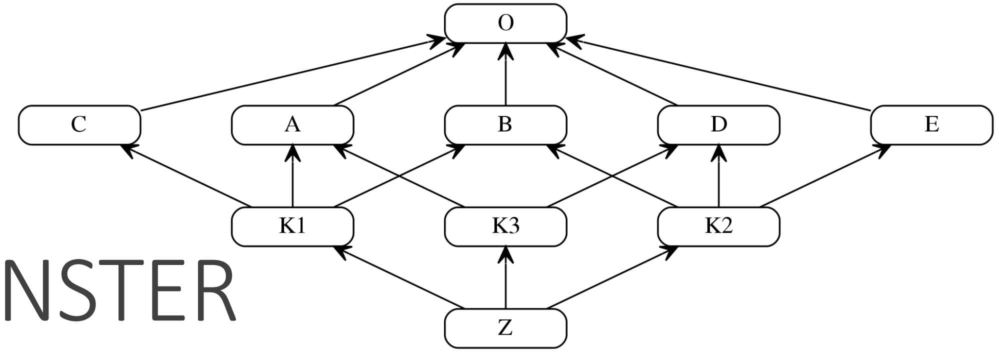
L[Z] = [Z, K1] + merge([A, B, C, O], [K2, D, B, E, O], [K3, D, **A**, O], [K2, K3])

Repeat Step 2
Pick head from the first list in merge
We pick A
Does A exist in the tails of the other lists?
YES! It exists in the tail of [K3, D, **A**, O]

Skip to the next list

https://en.wikipedia.org/wiki/C3_linearization

Try finding out the MRO of this MONSTER



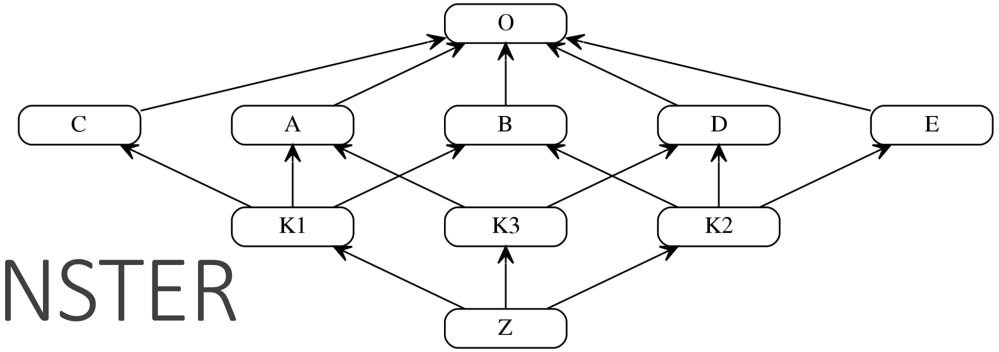
L[Z] = [Z, K1] + merge([A, B, C, O], [K2, D, B, E, O], [K3, D, A, O], [K2, K3])

Repeat Step 2
Pick head from the next list in merge
We pick K2
Does K2 exist in the tails of the other lists?
NO! Add it to [Z, K1]

Remove all K2 in the other lists

https://en.wikipedia.org/wiki/C3_linearization

Try finding out the MRO of this MONSTER

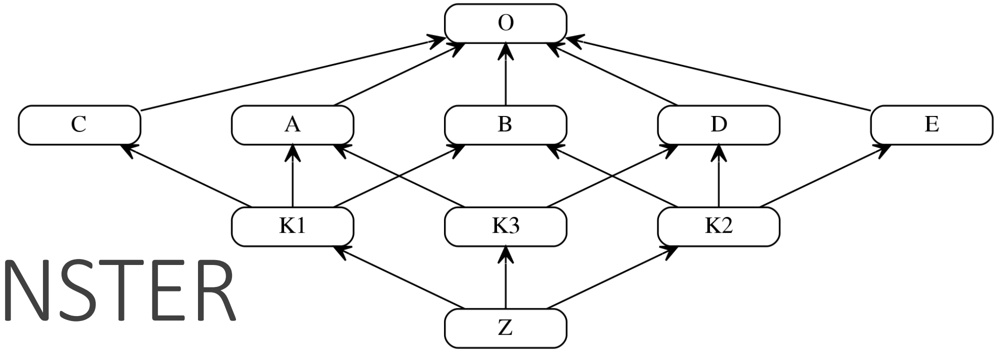


L[Z] = [Z, K1, K2] + merge([A, B, C, O], [D, B, E, O], [K3, D, <u>**A**</u>, O], [K3])

Repeat Step 2
Pick head from the next list in merge
We pick A
Does A exist in the tails of the other lists?
YES!

Skip to the next list

https://en.wikipedia.org/wiki/C3_linearization



Try finding out the MRO of this MONSTER

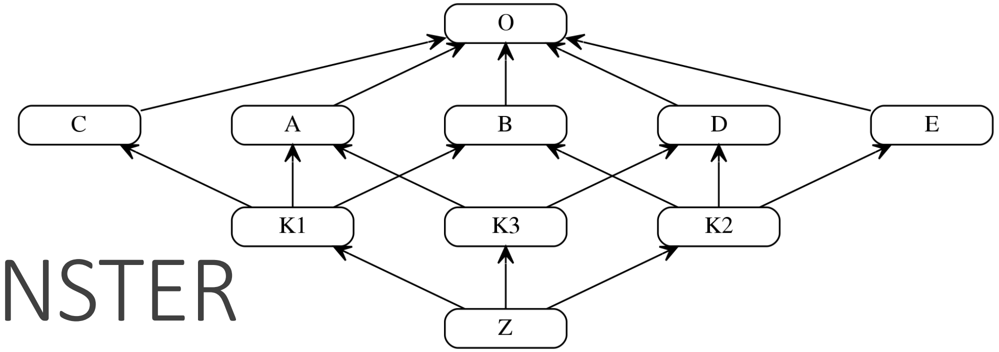
L[Z] = [Z, K1, K2] + merge([A, B, C, O], [D, B, E, O], [K3, **D**, A, O], [K3])

Repeat Step 2
Pick head from the next list in merge
We pick D
Does D exist in the tails of the other lists?
YES!

Skip to the next list

https://en.wikipedia.org/wiki/C3_linearization

Try finding out the MRO of this MONSTER



L[Z] = [Z, K1, K2] + merge([A, B, C, O], [D, B, E, O], [K3, D, A, O], [K3])

Repeat Step 2

Pick head from the next list in merge

We pick K3

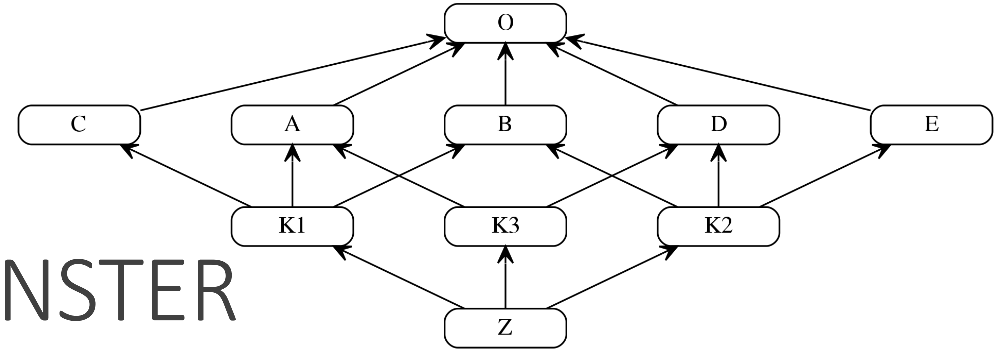
Does K3 exist in the tails of the other lists?

NO! Add to [Z, K1, K2]

Remove all K3 in the other lists

https://en.wikipedia.org/wiki/C3_linearization

Try finding out the MRO of this MONSTER



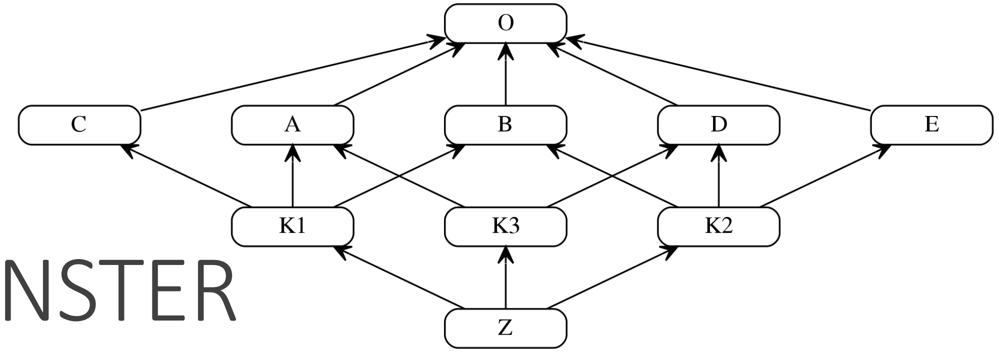
L[Z] = [Z, K1, K2, K3] + merge([A, B, C, O], [D, B, E, O], [D, A, O])

Keep repeating to get the following

= [Z, K1, K2, K3] + merge([A, B, C, O], [D, B, E, O], [D, A, O])	// fail A, select D
= [Z, K1, K2, K3, D] + merge([A, B, C, O], [B, E, O], [A, O])	// select A
= [Z, K1, K2, K3, D, A] + merge([B, C, O], [B, E, O], [O])	// select B
= [Z, K1, K2, K3, D, A, B] + merge([C, O], [E, O], [O])	// select C
= [Z, K1, K2, K3, D, A, B, C] + merge([O], [E, O], [O])	// fail O, select E
= [Z, K1, K2, K3, D, A, B, C, E] + merge([O], [O], [O])	// select O
= [Z, K1, K2, K3, D, A, B, C, E, O]	// done

https://en.wikipedia.org/wiki/C3_linearization

Try finding out the MRO of this MONSTER



L(O) = [O]
L(A) = [A,O]
L(B) = [B,O]
L(C) = [C,O]
L(D) = [D,O]
L(E) = [E,O]
L(K1) = [K1, A, B, C, D]
L(K2) = [K2, D, B, E, O]
L(K3) = [K3, D, A, O]

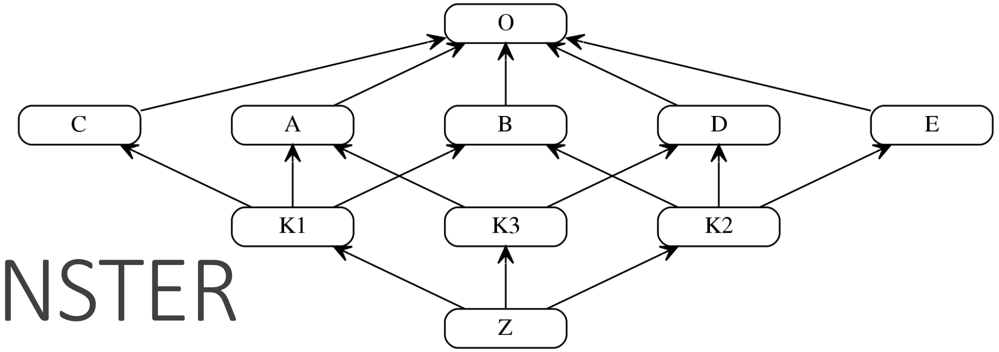
L[Z] = [Z, K1, K2, K3, D, A, B, C, E, O]

This final result may look strange because we might've expected
L[Z] = [Z, K1, K2, K3, A, B, C, D, E, O]

However the order of linearization completely depends on the inheritance order of the previous classes

Notice how K2, and K3 inherited D first then other classes? This is what caused the somewhat strange linearization

https://en.wikipedia.org/wiki/C3_linearization



Try finding out the MRO of this MONSTER

L(O) = [O]
L(A) = [A,O]
L(B) = [B,O]
L(C) = [C,O]
L(D) = [D,O]
L(E) = [E,O]
L(K1) = [K1, A, B, C, D]
L(K2) = [K2, D, B, E, O]
L(K3) = [K3, D, A, O]

L[Z] = [Z, K1, K2, K3, D, A, B, C, E, O]

However, our linearization algorithm ensures monotonicity, and local precedence

Local precedence ordering
- Class Z inherits from K1, K2, K3 before others classes

Monotonicity
- The inheritance order of K1, K2, K3's parents are maintained in the subclass Z

https://en.wikipedia.org/wiki/C3_linearization

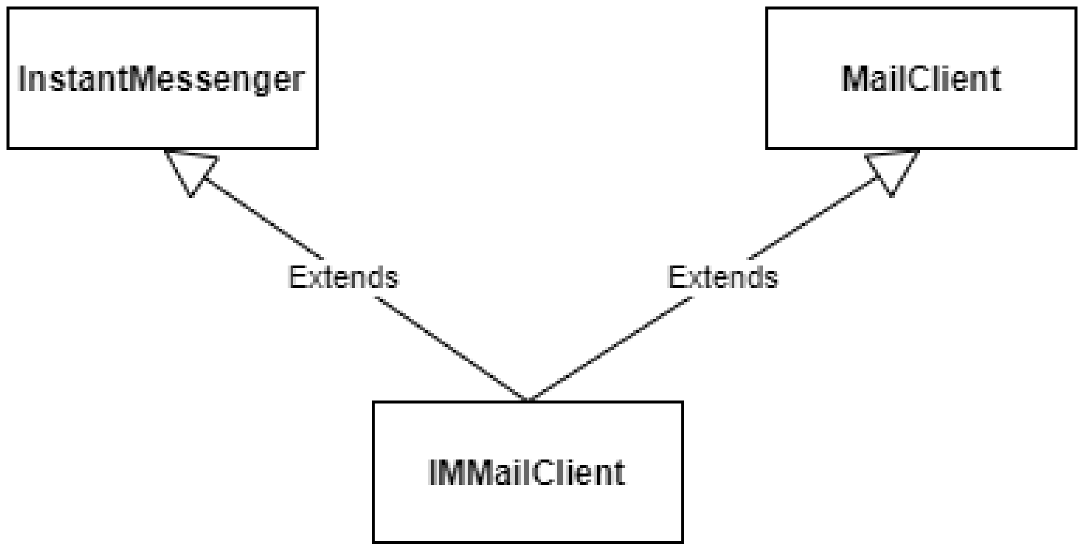
Mix-Ins

Mixins

Mixins are a special type of a **<u>SuperClass</u>** object which is not meant to be instantiated. Instead, it is meant to be Inherited.

These inherit and encapsulate behaviours and attributes from two completely different classes.

We then inherit from a mixin to create classes that can exhibit behaviours of both the base classes.



Mixins

```
class Base1:
    def sayHi(self):
        print('Hi’)

class Base2:
    def sayWorld(self):
        print('World’)

class MultiDerived (Base1, Base2):
    def __init__(self):
        self.sayHi()
        self.sayWorld()

m = MultiDerived()
```

```
Output:
Hi
World
```

Interface Segregation Principle

Interface Segregation Principle

Multiple Inheritance is a great way to implement different Interfaces from parent Abstract Base Classes (ABC's)

Some rules to guide you:

- Make sure the interfaces don't overlap
- Avoid putting implementation in the ABC's unless absolutely necessary. Emulate Java interfaces. Treat these as pure abstractions.
- You can use mixins to encapsulate different interfaces to create a common base class to inherit from. For the purpose of this course this isn't necessary or expected.

```
class Walkable(abc.ABC):

    @abc.abstractmethod
    def walk(self):
        pass

class Swimmable(abc.ABC)

    @abc.abstractmethod
    def swim(self):
        pass

class Human(Walkable, Swimmable):

    def walk(self):
        # overridden walking code

    def swim(self):
        # overridden swimming code
```

Multiple Inheritance Alternatives

Alternatives?

Questions to ask yourself:

- Do I need inheritance? Will this code be re-used a lot? If not, I can just define it separately in the child class.
- Can I restructure my base classes?
- Can I use composition instead?

That’s it for Week 4!

Start working on Assignment 1!