OOP

Software Development & Python

Nick Levashov, 2022

girafe
ai

OOP main principles

Encapsulation

Polymorphism

Inheritance

Design principles and patterns

KISS, DRY, …

SOLID

Prototype, Singleton, …

OOP principles

Object

Object = Data + Code

OOP main principles

Encapsulation

Polymorphism

Inheritance

Encapsulation

bundling data and methods that work on that data within one unit; using internal members

girafe
ai

Encapsulation

```python
class PhoneNumber:
    def __init__(self, country_code: str, number: str):
        self.country_code = country_code
        self.number = number

    def beautify_number(self):
        return (
            f'+{self.country_code} ({self.number[:3]}) '
            f'{self.number[3:6]}-{self.number[6:8]}-{self.number[8:]}'
        )

    def get_number(self):
        return self.country_code + self.number
```

Encapsulation

```python
class PhoneNumber:
    def __init__(self, country_code: str, number: str):
        self.country_code = country_code
        self.number = number

    def beautify_number(self):
        return (
            f'+{self.country_code} ({self.number[:3]}) '
            f'{self.number[3:6]}-{self.number[6:8]}-{self.number[8:]}'
        )

    def get_number(self):
        return self.country_code + self.number


my_phone = PhoneNumber('7', '9012345678')
friend_phone = PhoneNumber('7', '9001112233')

print(my_phone.beautify_number(), 'calling', friend_phone.beautify_number())
```

Encapsulation

```python
class PhoneNumber:
    def __init__(self, country_code: str, number: str):
        self._country_code = country_code
        self._number = number

    def beautify_number(self):
        return (
            f'+{self._country_code} ({self._number[:3]}) '
            f'{self._number[3:6]}-{self._number[6:8]}-{self._number[8:]}'
        )

    def get_number(self):
        return self._country_code + self._number


my_phone = PhoneNumber('7', '9012345678')
friend_phone = PhoneNumber('7', '9001112233')

print(my_phone.beautify_number(), 'calling', friend_phone.beautify_number())
```

Information hiding

```python
class PhoneNumber:
    def __init__(self, country_code: str, number: str):
        self.__country_code = country_code
        self.__number = number

    def beautify_number(self):
        return (
            f'+{self.__country_code} ({self.__number[:3]}) '
            f'{self.__number[3:6]}-{self.__number[6:8]}-{self.__number[8:]}'
        )

    def get_number(self):
        return self.__country_code + self.__number


my_phone = PhoneNumber('7', '9012345678')
friend_phone = PhoneNumber('7', '9001112233')

print(my_phone.beautify_number(), 'calling', friend_phone.beautify_number())
```

Encapsulation

```
class PhoneNumber:
    def __init__(self, country_code: str, number: str):
        self._country_code = country_code
        self._number = number

    def beautify_number(self):
        return (
            f'+{self._country_code} ({self._number[:3]}) '
            f'{self._number[3:6]}-{self._number[6:8]}-{self._number[8:]}'
        )

    @property
    def number(self):
        return self._country_code + self._number


my_phone = PhoneNumber('7', '9012345678')
friend_phone = PhoneNumber('7', '9001112233')

print(my_phone.beautify_number(), 'calling', friend_phone.beautify_number())
```

Polymorphism

provision of a single interface to entities of different types

girafe
ai

Polymorphism types

- Duck typing

- Ad-hoc Polymorphism (Overloading)
- Inclusion Polymorphism (Subtyping)
- Coersion Polymorphism (Casting)
- Parametric Polymorphism (Early Binding)

Duck typing

If it walks like a duck and it quacks like a duck, then it must be a duck

Duck typing

```python
def load_new_data(container: list):
    for result in request_results():
        container.append(result)


class FakeList:
    def append(self, element):
        print('tricked you')


a, b = [], FakeList()
load_new_data(a)
load_new_data(b)
```

Ad-hoc Polymorphism (Overloading)

```python
from functools import singledispatch

@singledispatch
def fun(arg, verbose=False):
    if verbose:
        print("Let me just say,", end=" ")
    print(arg)

@fun.register
def _(arg: int, verbose=False):
    if verbose:
        print("Strength in numbers, eh?", end=" ")
    print(arg)

@fun.register
def _(arg: list, verbose=False):
    if verbose:
        print("Enumerate this:")
    for i, elem in enumerate(arg):
        print(i, elem)
```

Inclusion Polymorphism (Subtyping)

```python
class ListWithCounter(list):
    def __init__(self, *args, **kwargs):
        self._search_counter = 0
        super().__init__(*args, **kwargs)

    def __contains__(self, item):
        self._search_counter += 1
        return super().__contains__(item)
```

```python
>>> a, b = [1, 1, 2, 3, 3], ListWithCounter([1, 1, 2, 3, 3])
>>> print(set(a), set(b))
{1, 2, 3} {1, 2, 3}
```

Coercion Polymorphism (Casting)

```cpp
#include<iostream>
using namespace std;

class Integer {
    int val;
    public:
        Integer(int x) : val(x) {
    }
    operator int() const {
        return val;
    }
};

void display(int x) {
    cout << "Value is: " << x << endl;
}

int main() {
    Integer x = 50;
    display(100);
    display(x);
}
```

Parametric Polymorphism (Early Binding)

```
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}
```

Polymorphism types

- Duck typing

- Ad-hoc Polymorphism (Overloading)
- Inclusion Polymorphism (Subtyping)
- Coersion Polymorphism (Casting)
- Parametric Polymorphism (Early Binding)

Inheritance

mechanism of basing an object or class upon another object or class, retaining similar implementation

girafe
ai

Inheritance

```
class A:
    def __init__(self):
        self.a = 1
    def a_method():
        print('A method')


class B(A):
    def __init__(self):
        self.b = 2
    def b_method():
        print('B method')


>>> B().a_method()
```

Inheritance

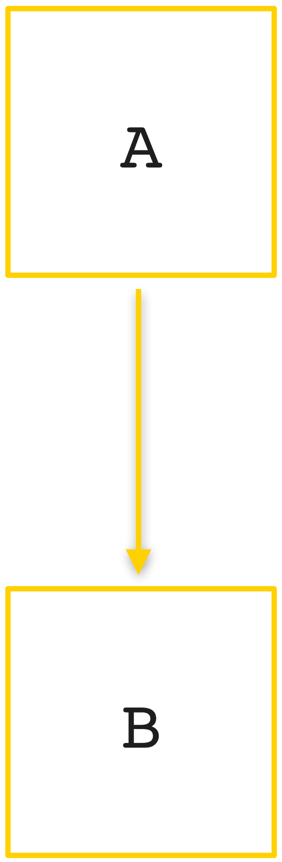
```
class A(object):
    def __init__(self):
        self.a = 1
    def a_method():
        print('A method')


class B(A):
    def __init__(self):
        self.b = 2
    def b_method():
        print('B method')


>>> B().a_method()
```

super()

```
class A:
    def __init__(self):
        self.a = 1
    def a_method():
        print('A method')


class B(A):
    def __init__(self):
        super().__init__()
        self.b = 2
    def b_method():
        print('B method')


>>> b = B()
>>> b.a_method()
>>> b.b_method()
>>> print(b.a, b.b)
```

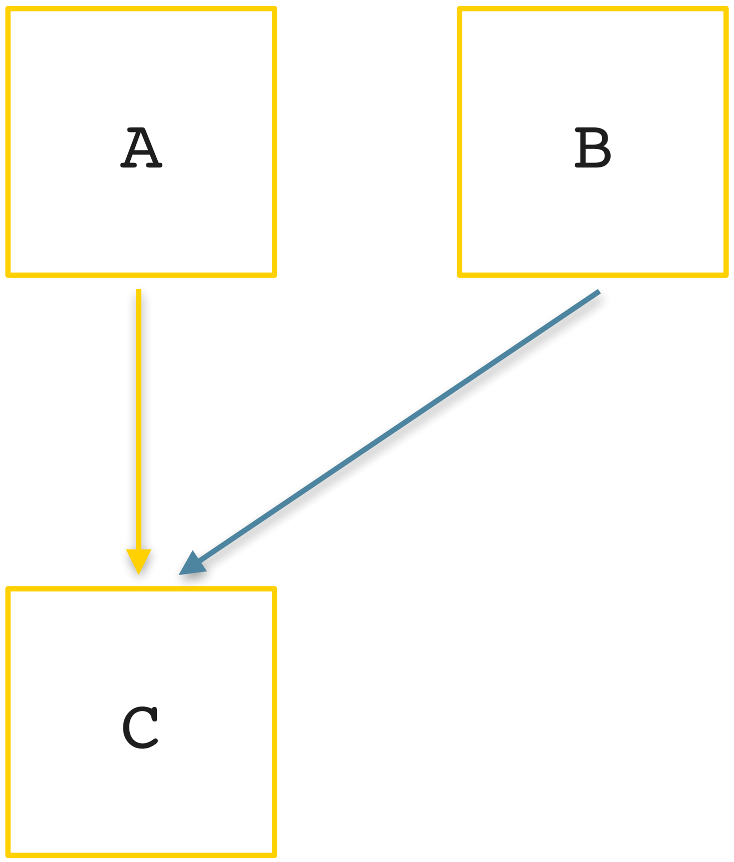
Single inheritance

```
class A: pass
class B(A): pass
```



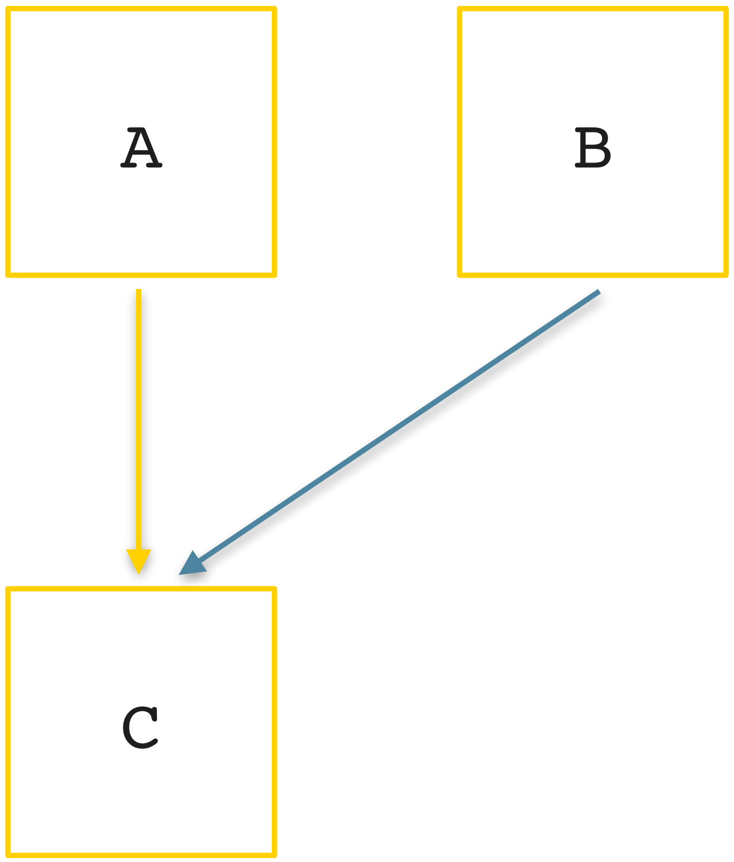
Multiple inheritance

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



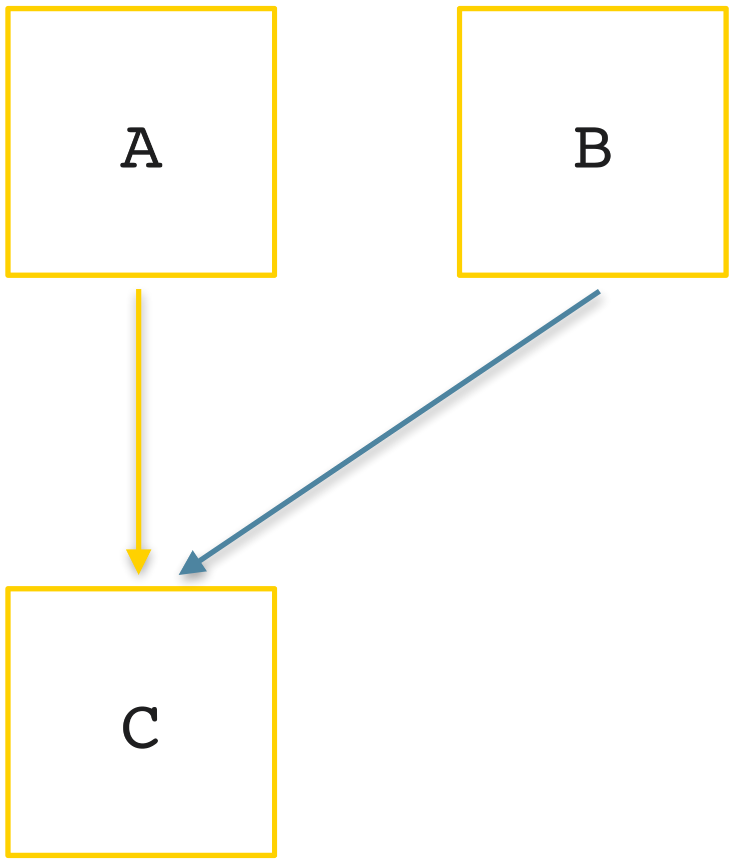
Multiple inheritance

```
class A:
    def __init__(self):
        self.a = 1

class B:
    def __init__(self):
        self.b = 2

class C(A, B):
    def __init__(self):
        B.__init__(self)
        A.__init__(self)
        self.c = 3

c = C()
print(c.a, c.b, c.c)
```



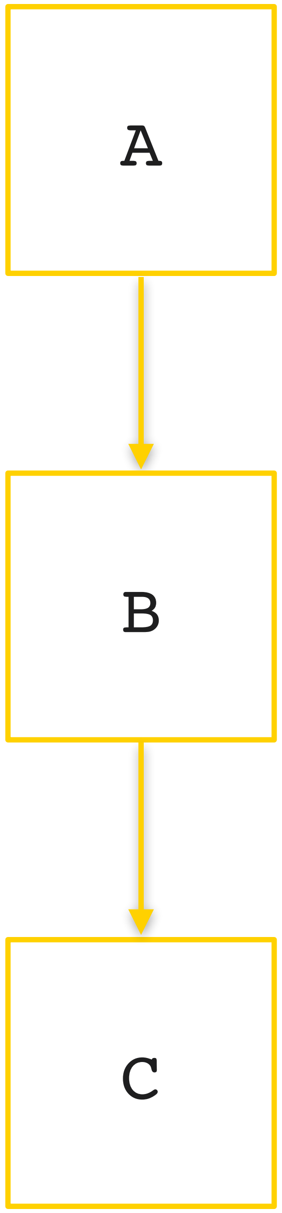
Multiple inheritance

```
class A:
    def __init__(self):
        super().__init__()
        self.a = 1

class B:
    def __init__(self):
        super().__init__()
        self.b = 2

class C(A, B):
    def __init__(self):
        super().__init__()
        self.c = 3

c = C()
print(c.a, c.b, c.c)
```



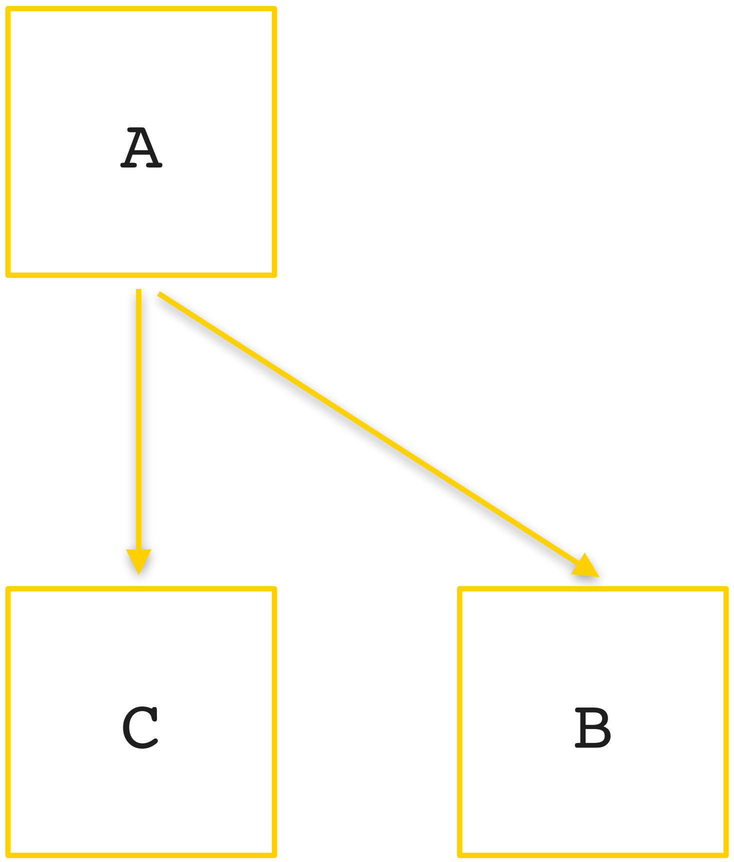
Multilevel inheritance

```python
class A: pass
class B(A): pass
class C(B): pass
```



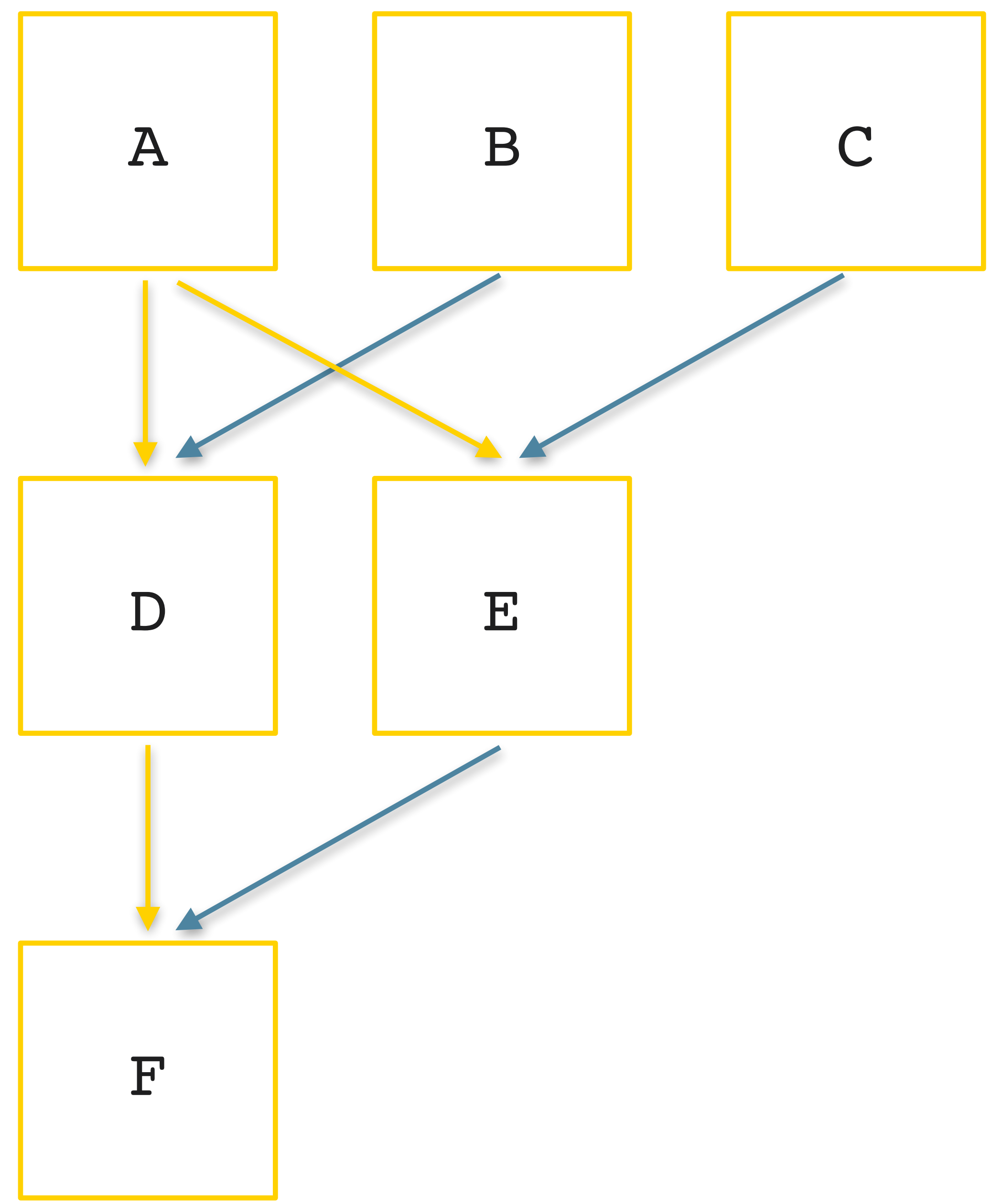
Hierarchical inheritance

```
class A: pass
class B(A): pass
class C(A): pass
```



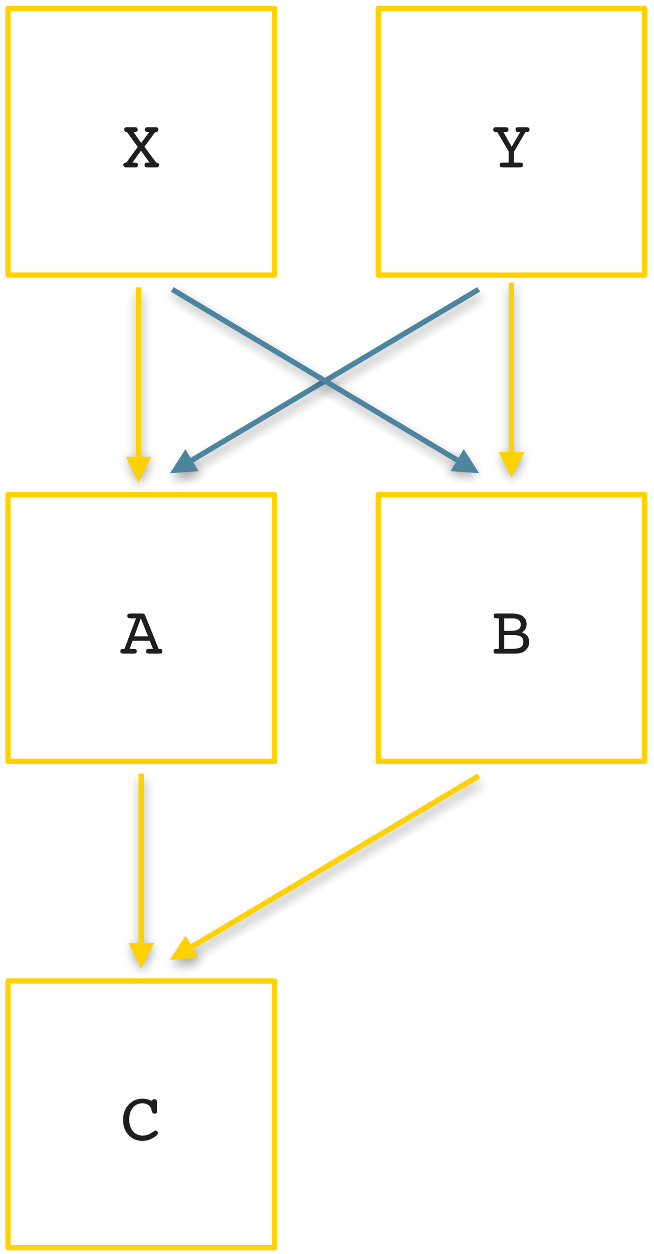
Hybrid inheritance

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



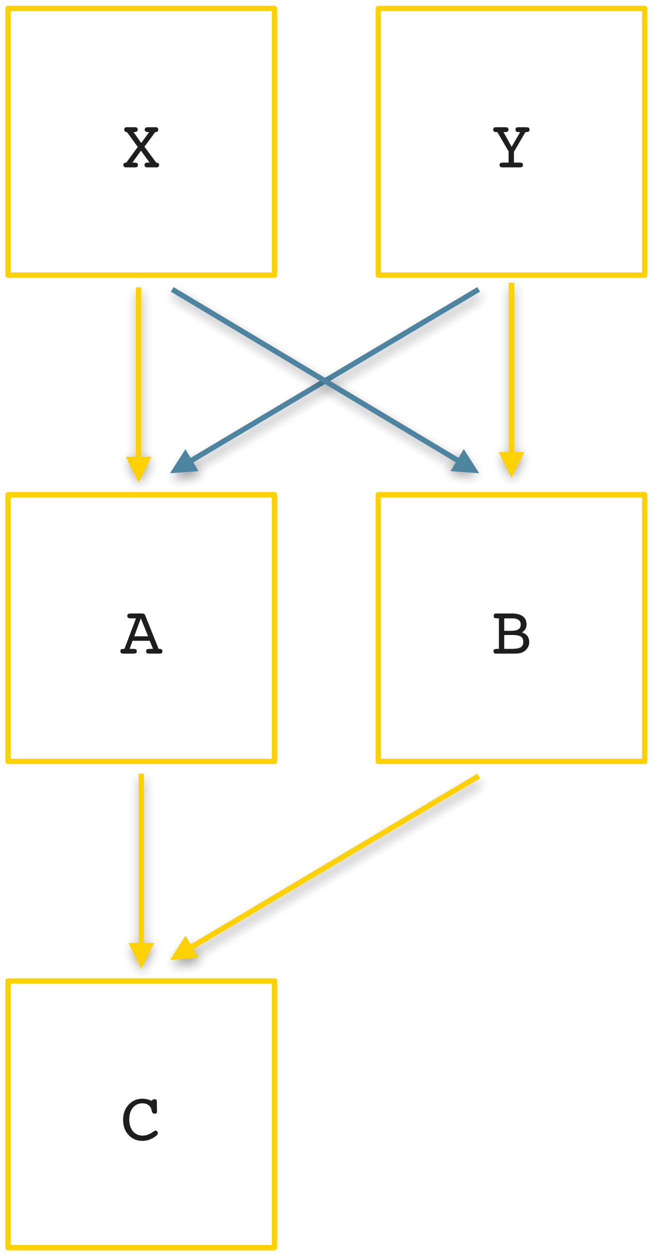
Hybrid inheritance (bad)

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



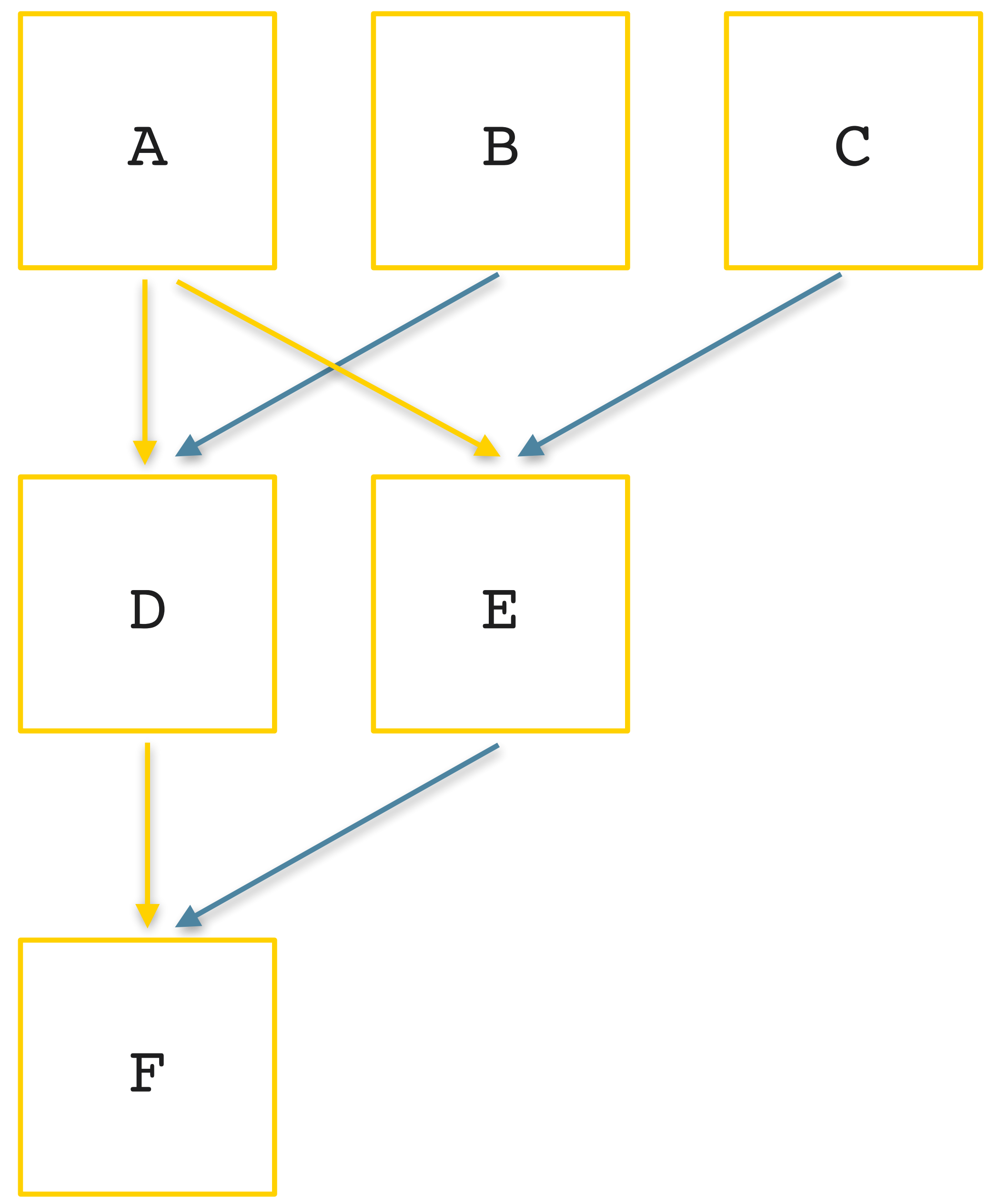
Hybrid inheritance (bad)

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

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



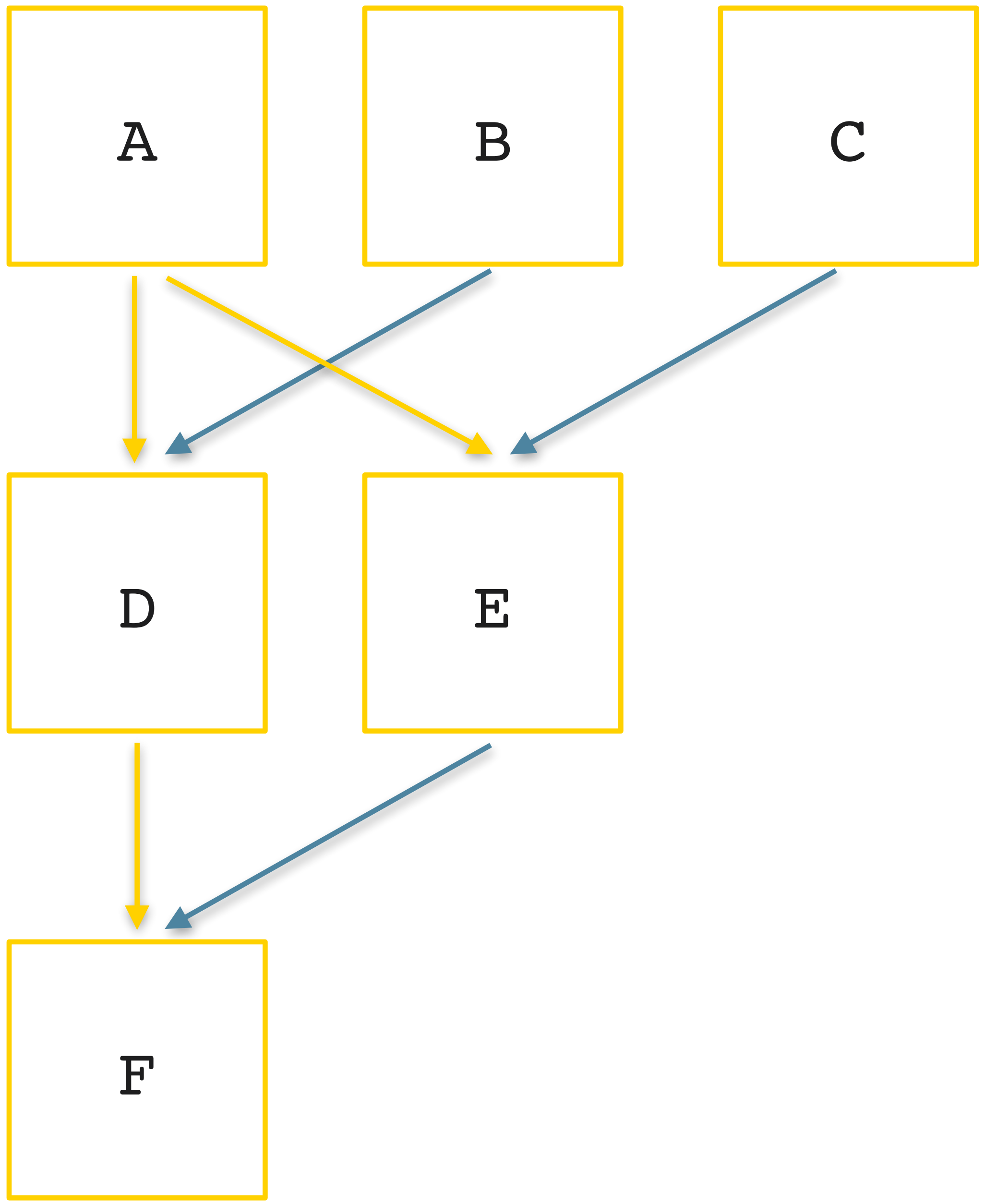
Method Resolution Order

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



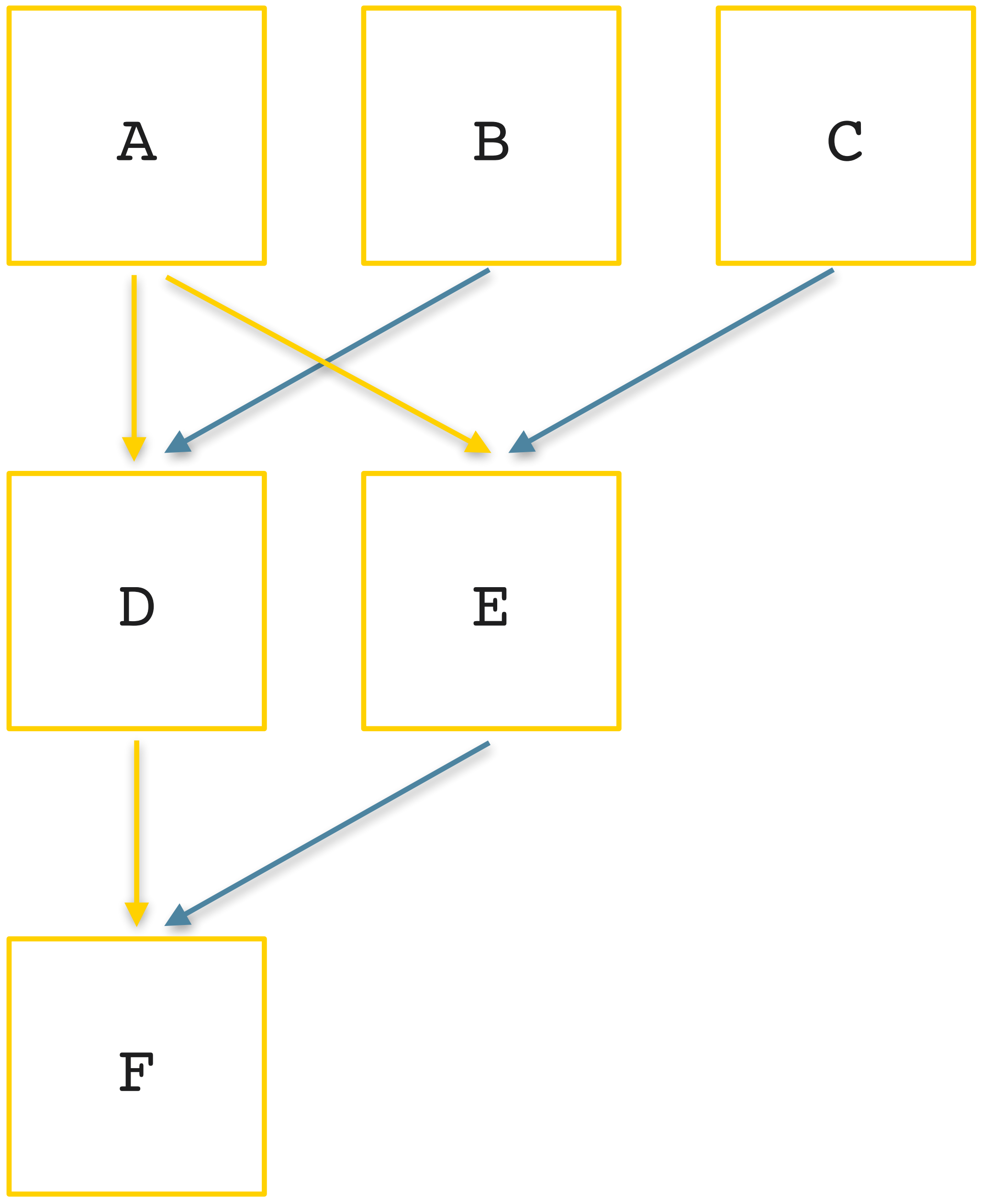
Method Resolution Order

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

>>> F.__mro__
(<class '__main__.F'>,
<class '__main__.D'>,
<class '__main__.E'>,
<class '__main__.A'>,
<class '__main__.B'>,
<class '__main__.C'>,
<class 'object'>)
```



Method Resolution Order

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

L[A] = AO
L[B] = BO
L[C] = CO
L[D] = DABO
L[E] = EACO
L[F] = FDEABCO
```



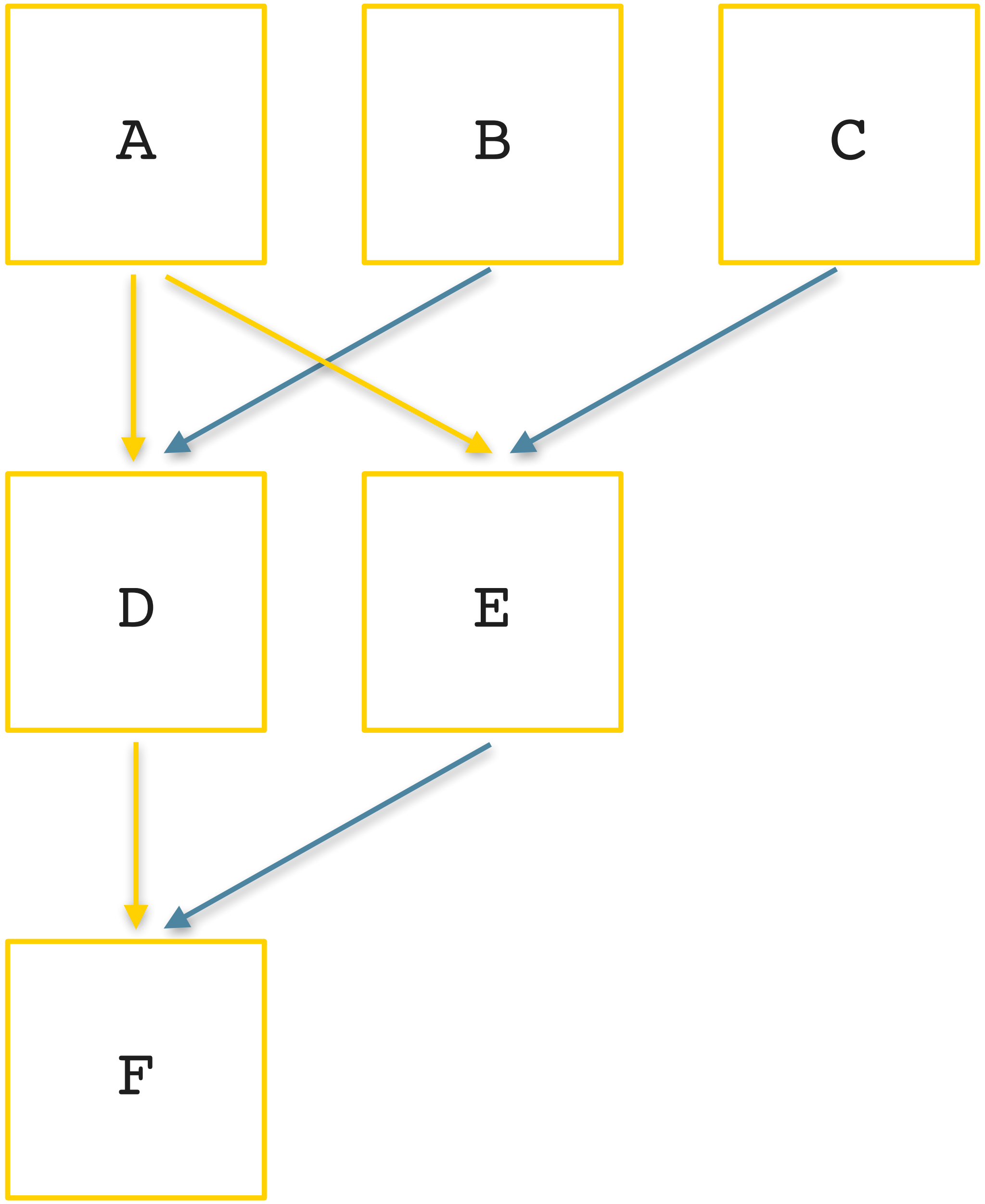
C3 linearization

Local Precedence Order

Monotonicity

C3 MRO

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

L[A] = AO
L[B] = BO
L[C] = CO
L[D] = DABO
L[E] = EACO
L[F] = FDEABCO
```



C3 MRO

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

L[A] = ?
L[B] = ?
L[C] = ?
L[D] = ?
L[E] = ?
L[F] = ?
```

C3 MRO

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

L[A] = AO
L[B] = BO
L[C] = CO
L[D] = ?
L[E] = ?
L[F] = ?
```

```
L[D] =
= D + merge(L[A], L[B], AB) =
= D + merge(AO, BO, AB) =
= D + A + merge(O, BO, B) =
= D + A + B + merge(O, O) =
= D + A + B + O =
= DABO


L[E] = ... = EACO
```

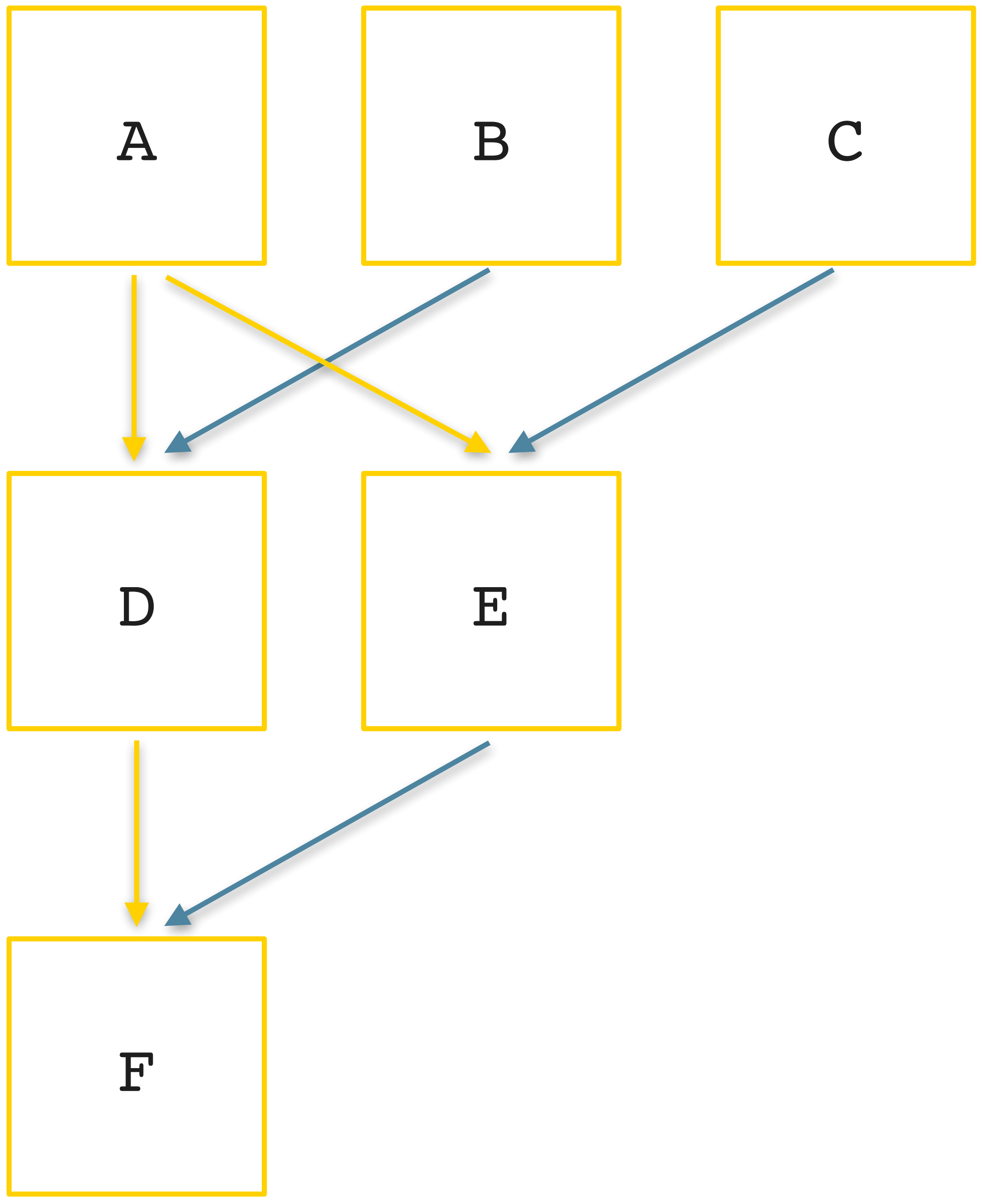
C3 MRO

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

L[A] = AO
L[B] = BO
L[C] = CO
L[D] = DABO
L[E] = EACO
L[F] = ?
```

```
L[F] =
= F + merge(L[D], L[E], DE) =
= F + merge(DABO, EACO, DE) =
= F + D + merge(ABO, EACO, E) =
= F + D + E + merge(ABO, ACO) =
= F + D + E + A + merge(BO, CO) =
= F + D + E + A + B + C + O =
= FDEABCO
```

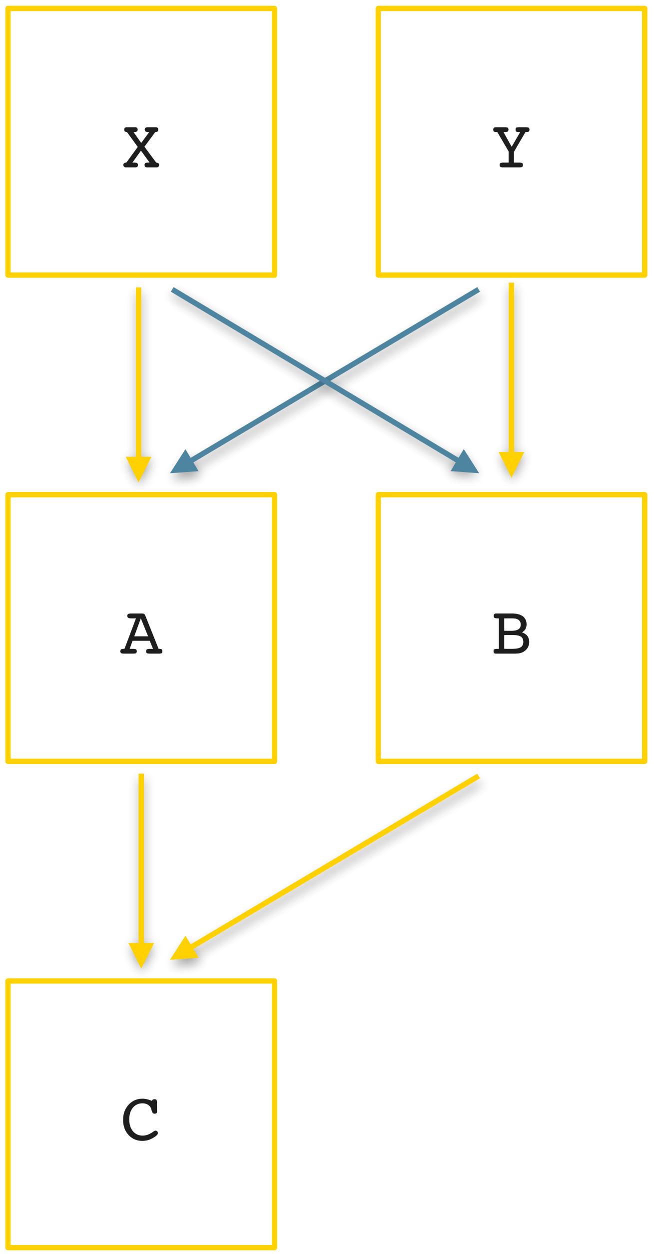
C3 MRO

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

L[A] = AO
L[B] = BO
L[C] = CO
L[D] = DABO
L[E] = EACO
L[F] = FDEABCO
```



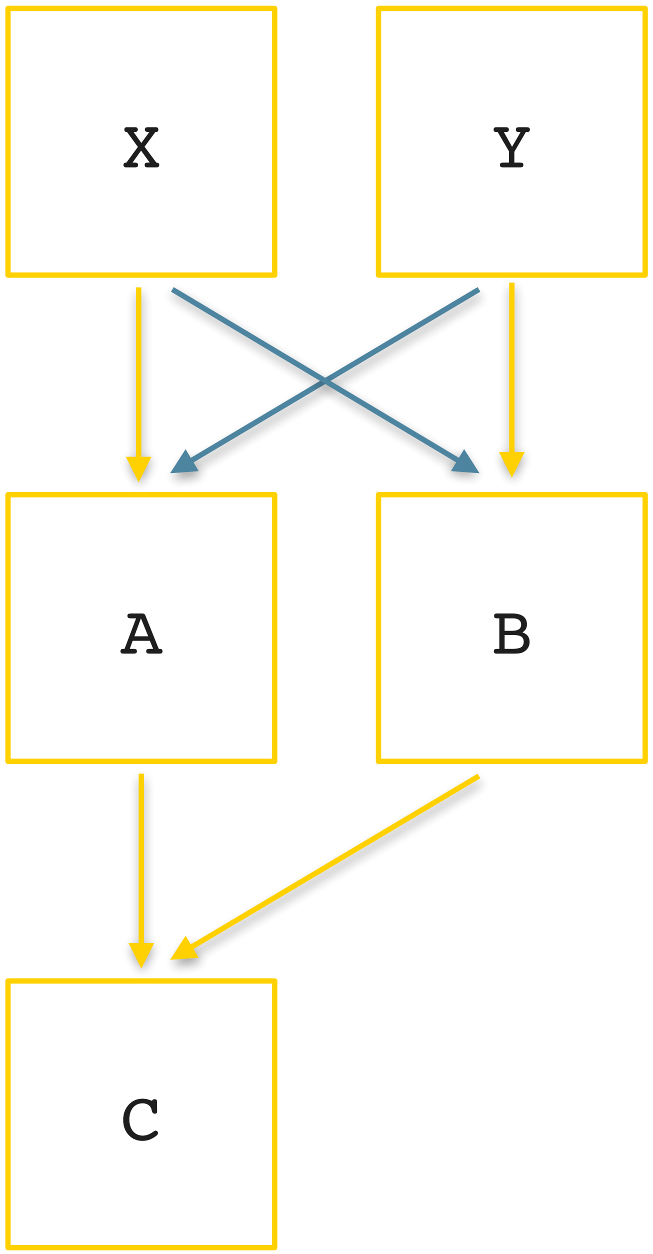
C3 MRO (bad)

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



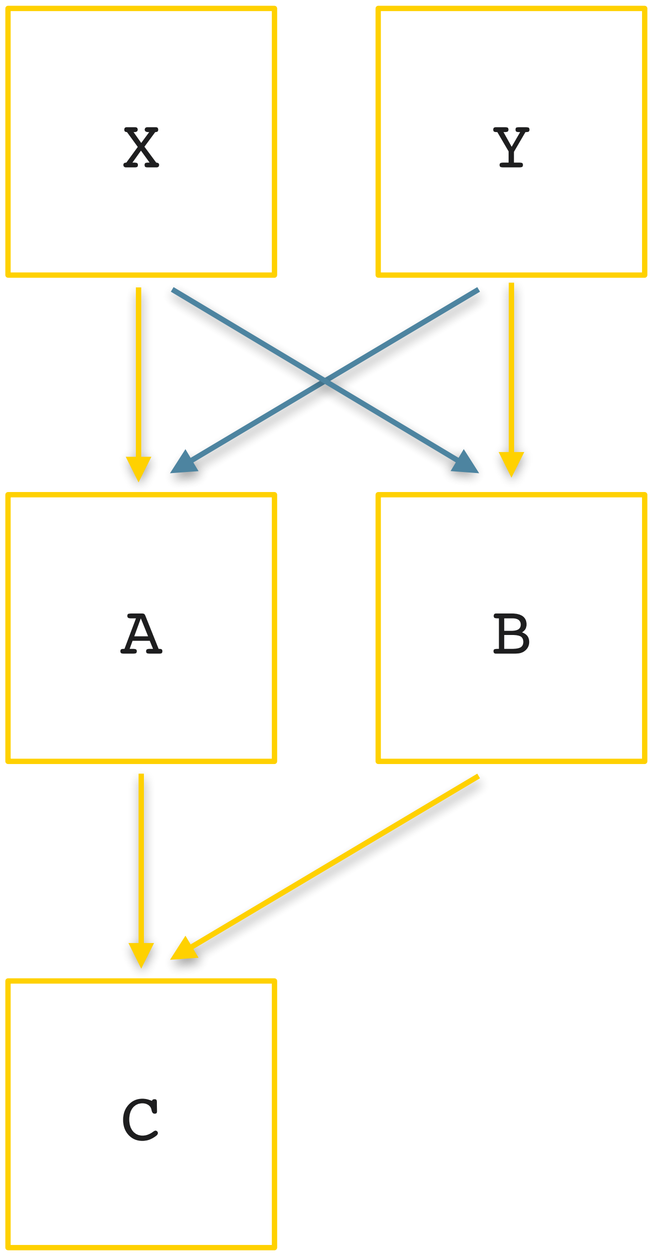
C3 MRO (bad)

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

L[X] = XO
L[Y] = YO
L[A] = ?
L[B] = ?
L[C] = ?
```



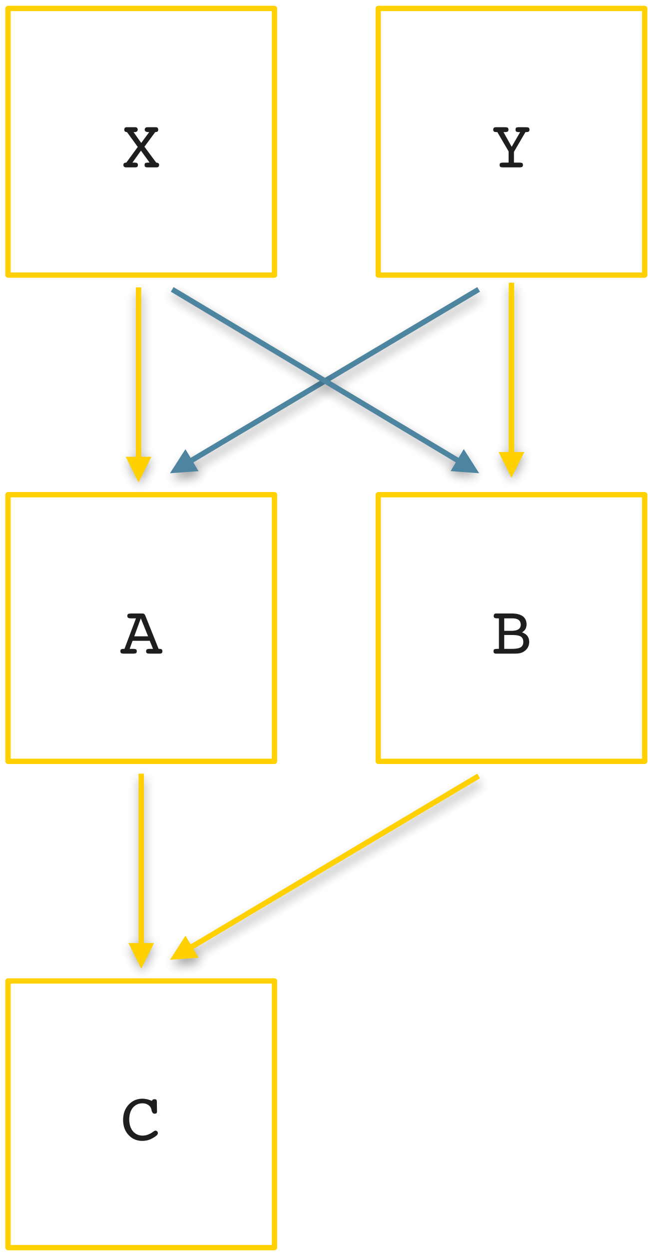
C3 MRO (bad)

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

L[X] = XO
L[Y] = YO
L[A] = ?
L[B] = ?
L[C] = ?

L[A] =
= A + merge(L[X], L[Y], XY) =
= A + merge(XO, YO, XY) =
= A + X + merge(O, YO, Y) =
= A + X + Y + merge(O, O) =
= A + X + Y + O = AXYO
```



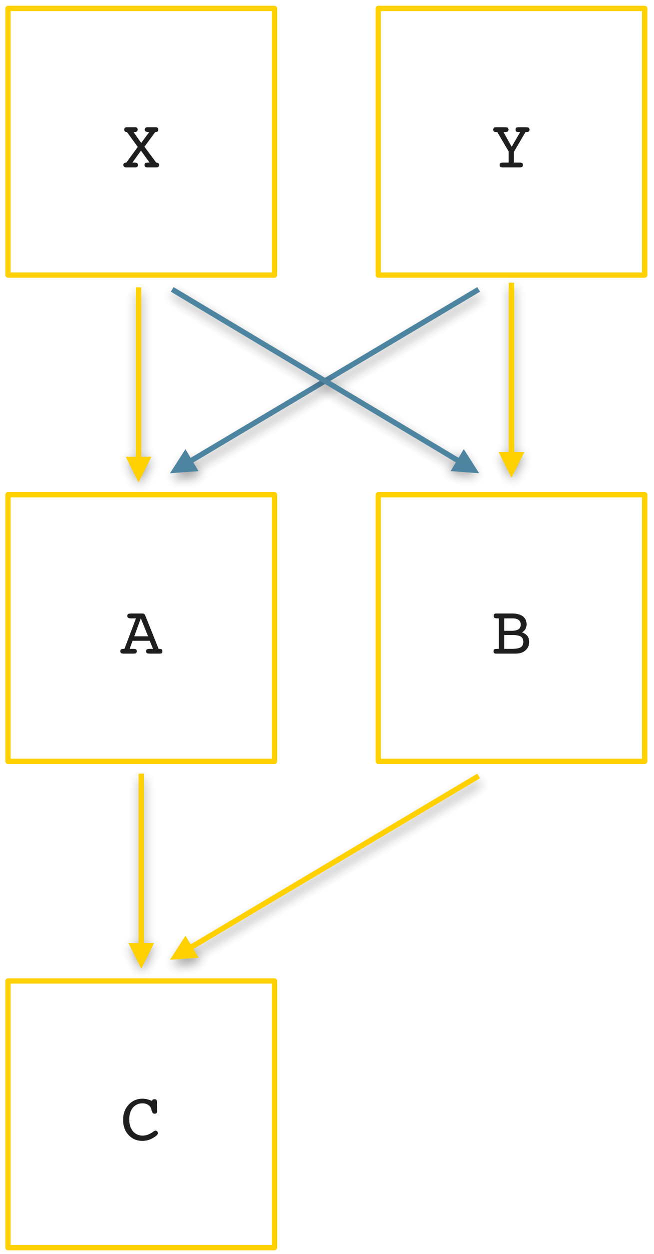
C3 MRO (bad)

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

L[X] = XO
L[Y] = YO
L[A] = AXYO
L[B] = BYXO
L[C] = ?
```



C3 MRO (bad)

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

L[X] = XO
L[Y] = YO
L[A] = AXYO
L[B] = BYXO
L[C] = ?

L[C] =
= C + merge(L[A], L[B], AB) =
= C + merge(AXYO, BYXO, AB) =
= C + A + merge(XYO, BYXO, B) =
= C + A + B + merge(XYO, YXO)
```



Method Resolution Order (C3)

The Python 2.3 Method Resolution Order

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

OOP main principles

Encapsulation

Polymorphism

Inheritance

Design principles and patterns

girafe
ai

Design principles

- KISS (Keep it simple, silly)

- DRY (Don't repeat yourself)
- WET (Write everything twice)

- YoYo problem

Design principles
The Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

SOLID

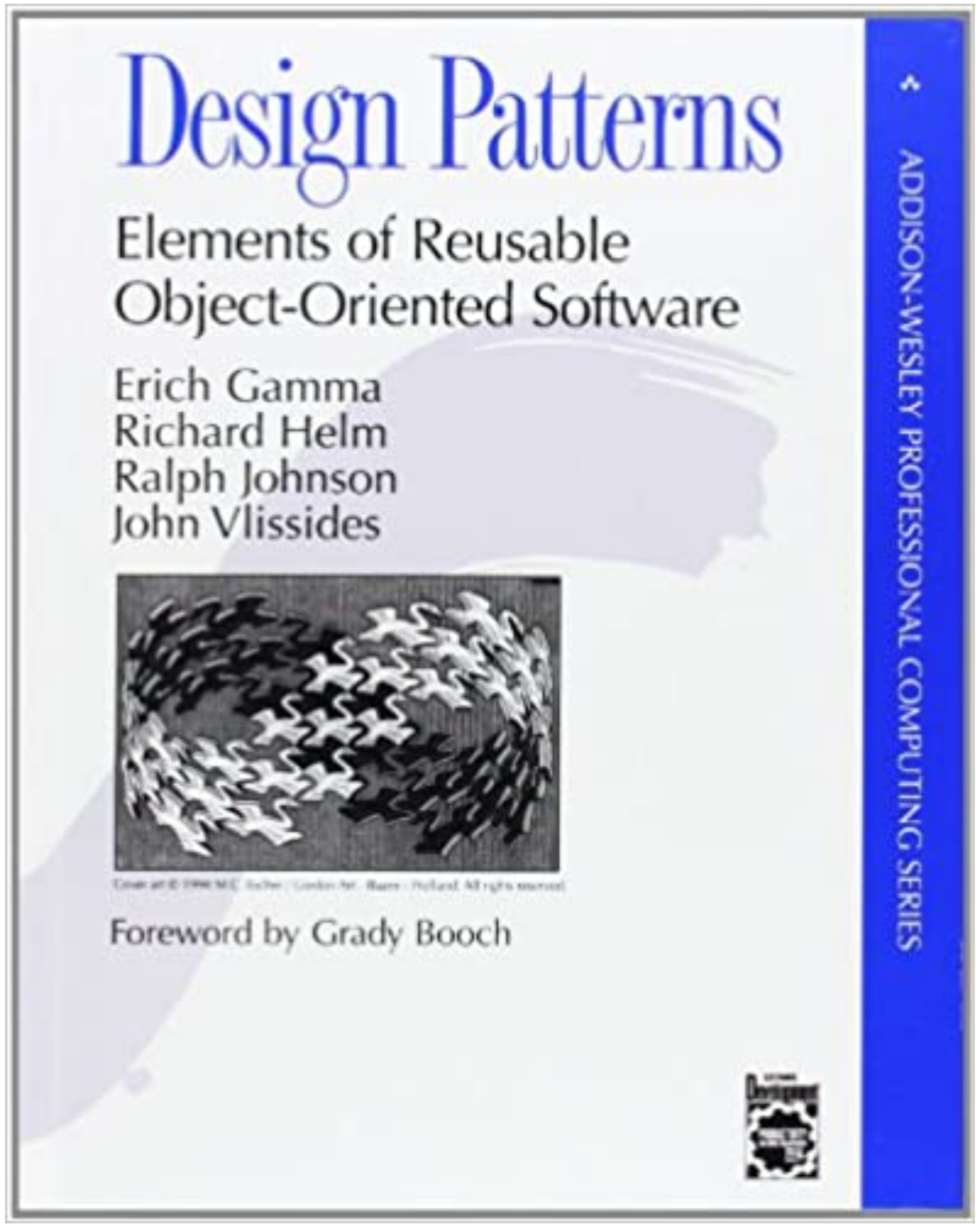
The Single-responsibility principle	There should never be more than one reason for a class to change
The Open–closed principle	Software entities should be open for extension, but closed for modification
The Liskov substitution principle	Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it
The Interface segregation principle	Many client-specific interfaces are better than one general-purpose interface
The Dependency inversion principle	Depend upon abstractions, not concretions

SOLID

SOLID in Python (Examples)

https://www.pythontutorial.net/python-oop/python-single-responsibility-principle/

Design patterns



Design patterns

- Creational patterns
- Structural patterns
- Behavioral patterns
- Concurrency patterns

Almost done

girafe
ai

girafe
ai

metaclasses

Singleton

```python
class MySingleton:
    def __init__(self):
        self.i = 1

ms1 = MySingleton()
ms1.i = 10

ms2 = MySingleton()
print(ms2.i)  # TODO 10
```

Singleton

```python
class Singleton:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not isinstance(cls._instance, cls):
            cls._instance = object.__new__(cls, *args, **kwargs)
        return cls._instance

class MySingleton(Singleton):
    def __init__(self):
        self.i = 1


ms1 = MySingleton()
ms1.i = 10

ms2 = MySingleton()
print(ms2.i)  # 1
```

metaclasses

```
>>> type(5)
<class 'int'>

>>> type(type(5))
<class 'type'>

>>> type(type(type(5)))
<class 'type'>
```

metaclasses

```
type(name: str, bases: tuple, attrs: dict, **kwds)
```

metaclasses

```
type(name: str, bases: tuple, attrs: dict, **kwds)

class MyClass: pass
MyClass = type('MyClass', (), {})
```

metaclasses

```
type(name: str, bases: tuple, attrs: dict, **kwds)

class MyClass: pass
MyClass = type('MyClass', (), {})


class MyClass(X, Y):
    a = 1
    def set_a(self):
        self.a = 10

def set_a(self):
    self.a = 10
MyClass = type('MyClass', (X, Y), {'a': 1, 'set_a': set_a})
```

metaclasses

```
def my_metaclass(name: str, bases: tuple, attrs: dict, **kwds):
    """Metaclass to add uppercase version of methods"""
    upper_attrs = {attr.upper(): value for attr, value in attrs.items()}
    attrs |= upper_attrs
    return type(name, bases, attrs, **kwds)


class MyClass(metaclass=my_metaclass):
    def print_100(self):
        print(100)


mc = MyClass()
mc.print_100()
mc.PRINT_100()
```

Singleton using metaclass

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class MySingleton(metaclass=Singleton):
    def __init__(self):
        self.i = 1


ms1 = MySingleton()
ms1.i = 10

ms2 = MySingleton()
print(ms2.i)  # 10
```

metaclasses

Post about Python metaclasses

https://stackoverflow.com/a/6581949

girafe
ai