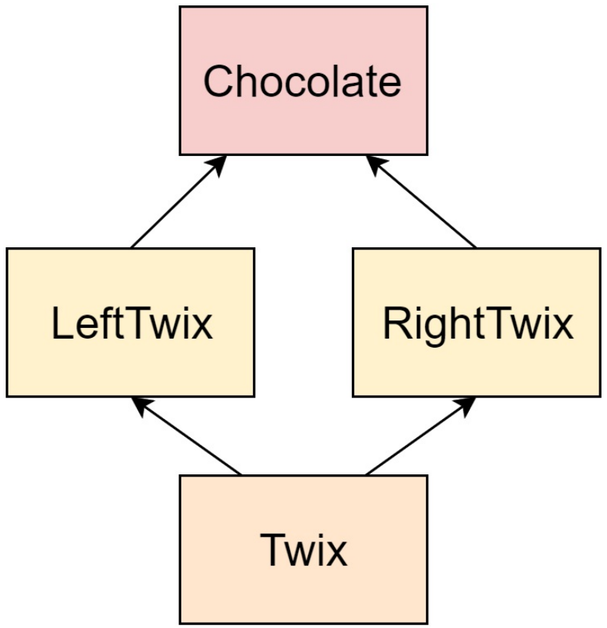
The curious case of __mro__

Method resolution order in multiple inheritance using C3 linearization



Inheritance in Python

- Python supports **multiple inheritance**, i.e. a class can be derived from more than one base classes
- In multiple inheritance, the features of all the base classes are inherited into the derived class
- The syntax for multiple inheritance is similar to single inheritance

In [1]:
```
class A:
    pass

class B:
    pass

class C(A,B):
    pass

print(C.__bases__)
```

```
(<class '__main__.A'>, <class '__main__.B'>)
```

What is MRO?

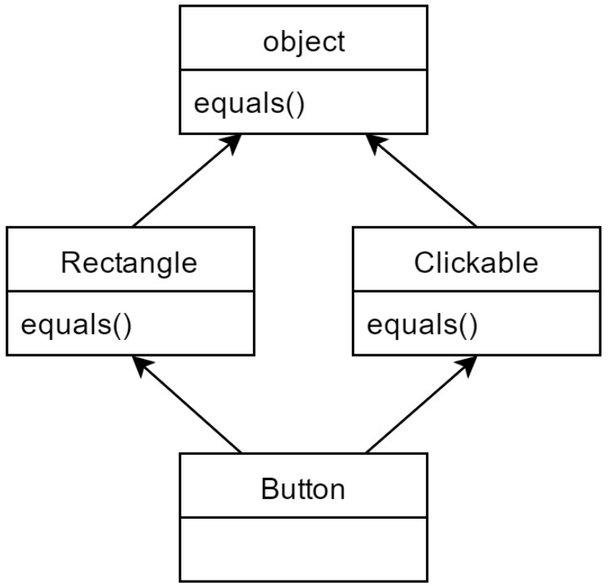
- MRO stands for Method Resolution Order
- MRO is the order in which base classes in the inheritance hierarchy are searched when looking for a method
- All Python versions after 2.3 use the *C3 linearization* algorithm to determine this order
- Not all classes admit a linearization. There are cases, in complicated hierarchies, where it is not possible to derive a class such that its linearization respects all the desired properties.

Check this article on python.org for more info. Part of this presentation also based on this tutorial.

Why is this useful?

- Given a class C, in the case of single inheritance hierarchy, if C is a subclass of C1, and C1 is a subclass of C2, then the linearization of C is simply the list [C,C1,C2].
- However, in a complicated multiple inheritance hierarchy, it is a non-trivial task to specify the order in which methods are overridden, i.e. to specify the order of the ancestors of C.
- We need to be able discover **deterministically** the order of classes for method calls in the inheritance chain.
- The list of the ancestors of a class C, including the class itself, ordered from the nearest ancestor to the furthest, is called the *class precedence list* or the *linearization* of C.
- The Method Resolution Order (MRO) is the set of rules that construct the linearization. In the Python literature, the idiom the *MRO* of the class C is also used as a synonymous for the *linearization* of C.

The diamond problem



- In the above example the `Button` class inherits two differerent implementations of `equals()`
- It has no implementation of the operation of its own
- When `button.equals()` is called, it is unknown which implemetation - from `Rectangle`, `Clickable` or `object` will be used

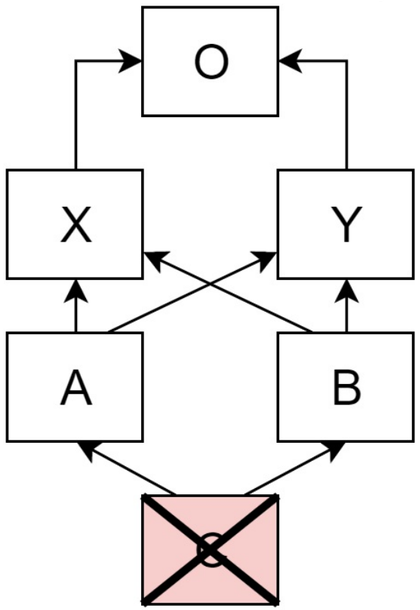
C3 Linearization

- First introduced in the Dylan language
- Algorithm based on 3 important properties (this is how the name *C3* is derived)

    1. Consistent extended precedence graph (MRO is determined based on structure of the inheritance graph)
    2. Preserving local precedence ordering (no class will appear before any of its subclasses)
    3. Monotonicity

Monotonicity

- An MRO is monotonic when the following is true:
  - If C1 precedes C2 in the linearization of C then C1 precedes C2 in the linearization of any subclass of C.
- Consider: `class X(O), class Y(O), class A(X,Y), class B(Y,X), class C(B,A)`
  - Based on monotonicity it is **not** possible to derive a new class C from A and B since X precedes Y in A, but Y precedes X in B, therefore the method resolution order would be ambiguous in C (XY breaks monotonicity with B, YX breaks monotonicity with A).



Definition and notation

- Notation
  - C1 C2 … CN indicates the list of classes [C1,C2,…,CN]
  - The *head* of the list is its first element: head = C1
  - The *tail* is the rest of the list: tail = C2 … CN
  - The sum of the lists [C] + [C1,C2,…,CN] = C + (C1 C2 … CN) = C C1 C2 … CN
- Consider a class C in a multiple inheritance hierarchy, with C inheriting from the base classes B1, B2, …, BN:
  - The linearization of C is the sum of C plus the merge of linearizations of the parents and the list of the parents
  - L[C(B1 … BN)] = C + merge(L[B1],…,L[BN], B1 … BN)
- Example: L[Y(X1 X2 X3)] = Y + merge(L[X1],L[X2],L[X3], X1 X2 X3)

Computing merge

Consider a simple merge example: merge(DO,EO,DE) = DEO

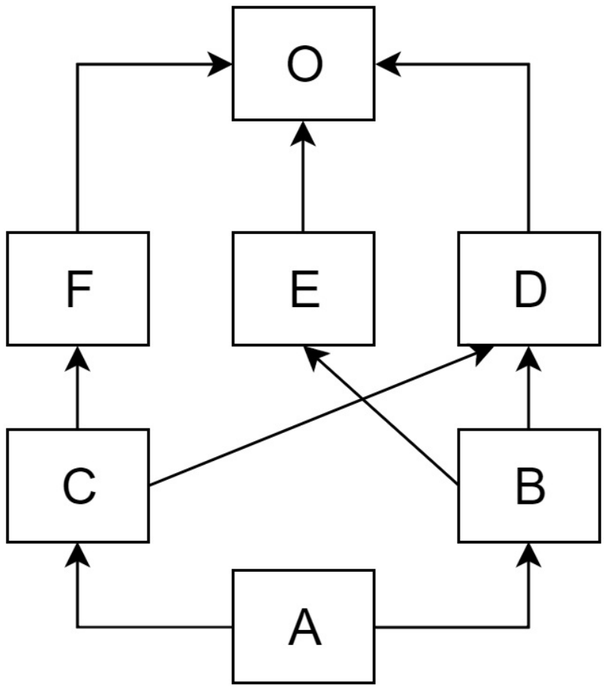
1. Select the first head of the lists which does not appear in the tail of any of the other lists.
    - A good head may appear as the first element in multiple lists at the same time, but it is forbidden to appear anywhere else.
2. Remove the selected element from all the lists where it appears as a head and append to the output list.
3. Repeat the operation of selecting and removing a good head to extend the output list until all remaining lists are exhausted.
4. If at some point no good head can be selected, because the heads of all remaining lists appear in any one tail of the lists, then the merge is impossible to compute due to cyclic dependencies in the inheritance hierarchy and no linearization of the original class exists.

Properties of merge

- Three important considerations when computing merge:
  1. The merge of several sequences is a sequence that contains **each** of the the elements of the input sequence
     - All elements within the input lists *DO*, *EO* and *DE* are present in the merged result *DEO*.
  2. An element that appears in more than once of the input sequences appears **only once** in the output sequence
     - *D*, *E* and *O* appear in more than on input sequence, but the result has only one instance of each.
  3. If two elements appear in the same input sequence, their order in the output sequence is the same as their order in the input sequence.
     - In the input sequence, D precedes both O and E; E precedes O. The same ordering is maintained in the merged output.

Compute the linearization:

class A(B,C), class B(D,E) class C(D,F), class D(O), class E(O), class F(O), class O



C3 computing example

L[C(B1 ... BN)] = C + merge(L[B1],...,L[BN], B1 ... BN)

```
L[O] = O
L[D] = D + merge(L[O],O) = D + merge(O,O) = DO
L[E] = EO, L[F] = FO
L[B] = B + merge(L[D],L[E],DE)
    = B + merge(DO,EO,DE)
    = B + D + merge(O,EO,E)
    = B + D + E + merge(O,O)
    = BDEO
L[C] = C + merge(L[D],L[F],DF)
    = C + merge(DO,FO,DF)
    = CDFO
L[A] = A + merge(L[B],L[C],BC)
    = A + merge(BDEO,CDFO,BC)
    = A + B + merge(DEO,CDFO,C)
    = A + B + C + merge(DEO,DFO)
    = A + B + C + D + merge(EO,FO)
    = A + B + C + D + E + merge(O,FO)
    = A + B + C + D + E + F + merge(O,O)
    = ABCDEFO
```

In [2]:

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

from inspect import getmro
print(getmro(A))
print(A.__mro__)
```

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

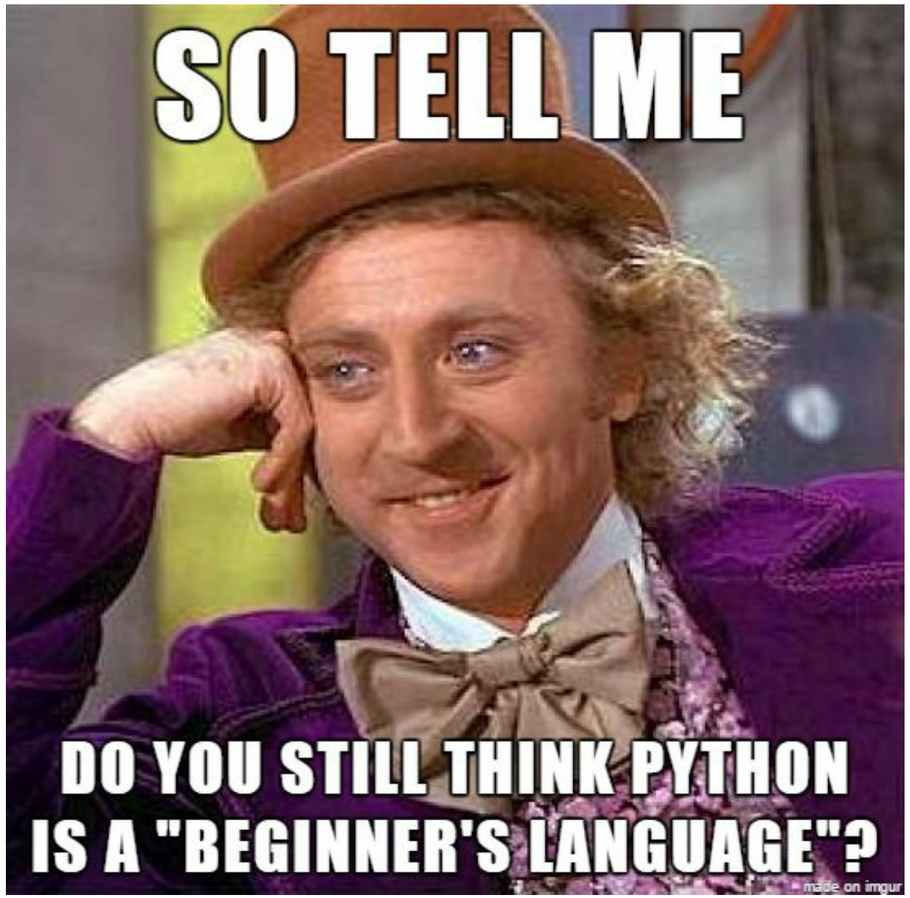
In [3]:

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

print(E.__mro__)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-b4c716884832> in <module>()
      3 class C(A,B): pass
      4 class D(B,A): pass
----> 5 class E(C,D): pass
      6
      7 print(E.__mro__)

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


SO TELL ME
DO YOU STILL THINK PYTHON
IS A "BEGINNER'S LANGUAGE"?
made on imgur