Object-Oriented Programming

Formal Inheritance Rules

Inheritance occurs on attribute name reference—the *object.name* lookup at the heart of object-oriented code—whenever *object* is derived from a class. It differs in classic and new-style classes, although typical code often runs the same in both models.

Classic classes: DFLR

In classic classes (the default in 2.X), for name references, inheritance searches:

1. The *instance*
2. Then its *class*
3. Then all its class's *superclasses*, depth-first and then left-to-right

The first occurrence found along the way is used. This order is known as *DFLR*.

This reference search may be kicked off from either an instance or a class; attribute *assignments* normally store in the target object itself without search; and there are special cases for `__getattr__()` (run if the lookup failed to find a name) and `__setattr__()` (run for all attribute assignments).

New-style classes: MRO

Inheritance in new-style classes (the standard in 3.X and an option in 2.X) employ the *MRO*—a linearized path through a class tree, and a nested component of inheritance, made available in a class's `__mro__` attribute. The MRO is roughly computed as follows:

1. List all the classes that an instance inherits from using the classic class's DFLR lookup rule, and include a class multiple times if it's visited more than once.
2. Scan the resulting list for duplicate classes, removing all but the last (rightmost) occurrence of duplicates in the list.

The resulting MRO sequence for a given class includes the class, its superclasses, and all higher superclasses up to and including the implicit or explicit `object` root class at the top of the tree. It's ordered such that each class appears before its parents, and multiple parents retain the order in which they appear in the `__bases__` superclass tuple.

Because common parents in *diamonds* appear only at the position of their *last* visitation in the MRO, lower classes are searched first when the MRO list is used later by attribute inheritance (making it more breadth-first than depth-first in diamonds only), and each class is included and thus visited just *once*, no matter how many classes lead to it.

The MRO ordering is used both by inheritance (ahead) and by the `super()` call—a built-in function that always invokes a *next* class on the MRO (relative to the call point), which might not be a superclass at all, but can be used to dispatch method calls throughout a class tree visiting each class just once.

Example: non-diamonds

```
class D:       attr = 3            #   D:3   E:2
class B(D):    pass                #   |     |
class E:       attr = 2            #   B     C:1
class C(E):    attr = 1            #    \   /
class A(B, C): pass                #      A
X = A()                            #      |
print(X.attr)                      #      X

# DFLR = [X, A, B, D, C, E]
# MRO  = [X, A, B, D, C, E, object]
# Prints "3" in both 3.X and 2.X (always)
```

Example: diamonds

```
class D:       attr = 3            #   D:3   D:3
class B(D):    pass                #   |     |
class C(D):    attr = 1            #   B     C:1
class A(B, C): pass                #    \   /
X = A()                            #      A
print(X.attr)                      #      |
                                   #      X
# DFLR = [X, A, B, D, C, D]
# MRO  = [X, A, B, C, D, object] (retains last D only)
# Prints "1" in 3.X, "3" in 2.X (or "1" if D(object))
```

New-Style inheritance algorithm

Depending on class code, new-style inheritance may involve descriptors, metaclasses, and MROs as follows (name sources in this procedure are attempted in order, either as numbered or per their left-to-right order in "or" conjunctions):

To look up an attribute name:

1. From an *instance* `I`, search the instance, its class, and its superclasses, as follows:
   a) Search the `__dict__` of all classes on the `__mro__` found at `I`'s `__class__`.
   b) If a data descriptor was found in step *a*, call its `__get__()` and exit.
   c) Else, return a value in the `__dict__` of the instance `I`.
   d) Else, call a nondata descriptor or return a value found in step *a*.
2. From a *class* `C`, search the class, its superclasses, and its metaclasses tree, as follows:
   a) Search the `__dict__` of all metaclasses on the `__mro__` found at `C`'s `__class__`.
   b) If a data descriptor was found in step *a*, call its `__get__()` and exit.
   c) Else, call a descriptor or return a value in the `__dict__` of a class on `C`'s own `__mro__`.
   d) Else, call a nondata descriptor or return a value found in step *a*.
3. In both rule 1 and 2, *built-in* operations (e.g., expressions) essentially use just step *a* sources for their implicit lookup of method names, names, and `super()` lookup is customized.

In addition, method `__getattr__()` may be run if defined when an attribute is not found; method `__getattribute__()` may be run for every attribute fetch; and the implied `object` superclass provides some defaults at the top of every class and metaclass tree (that is, at the end of every MRO).

As *special cases*, built-in operations skip name sources as described in rule *3*, and the `super()` built-in function precludes normal inheritance. For objects returned by `super()`, attributes are resolved by a special context-sensitive scan of a limited portion of a class's MRO only, choosing the first descriptor or value found along the way, instead of running full inheritance (which is used on the `super` object itself only if this scan fails); see `super()` in "Built-in Functions".

To assign an attribute name:

A subset of the lookup procedure is also run for attribute assignments:

- When applied to an *instance*, such assignments essentially follow steps *a* through *c* of rule *1*, searching the instance's class tree, although step *b* calls `__set_()` instead of `__get__()`, and step *c* stops and stores in the instance instead of attempting a fetch.
- When applied to a *class*, such assignments run the same procedure on the class's metaclass tree: roughly the same as rule *2*, but step *c* stops and stores in the class.

The `__setattr__()` method still catches all attribute assignments as before, although it becomes less useful for this method to use the instance `__dict__` to assign names, as some new-style extensions such as slots, properties,

and descriptors implement attributes at the class level—a sort of "virtual" instance data mechanism. Some instances might not have a `__dict__` at all when slots are used (an optimization).

New-style precedence and context

New-style inheritance procedures effectively impose precedence rules on the foundational operation of name resolution, which may be thought of as follows (with corresponding steps of the inheritance algorithm in parentheses):

For instances, try:

1. Class-tree data descriptors (*1b*)
2. Instance-object values (*1c*)
3. Class-tree nondata descriptors (*1d*)
4. Class-tree values (*1d*)

For classes, try:

1. Metaclass-tree data descriptors (*2b*)
2. Class-tree descriptors (*2c*)
3. Class-tree values (*2c*)
4. Metaclass-tree nondata descriptors (*2d*)
5. Metaclass-tree values (*2d*)

Python runs at most one (for instances) or two (for classes) tree searches per name lookup, despite the presence of four or five name sources. This may happen in addition to selecting a context-specific starting point per the MRO, for code using the new-style `super()` built-in function.

See also "Methods for Descriptors" and "Metaclasses" for their subjects; "Operator Overloading Methods" for usage details of `__setattr__()`, `__getattr__()`, and `__getattribute__()`; and Python's *object.c* and *typeobject.c* source code files, which host the implementations of instances and classes, respectively (in Python's source code distribution).

...from later related section...

Built-in Functions

`super([type [, object]])`

> Returns the superclass of *`type`*. If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(object, type)` must be true. If the second argument is a type, `issubclass(object, type)` must be true. This call works for all classes in 3.X, but only for new-style classes in Python 2.X, where *`type`* is also not optional.
>
> In 3.X only, calling `super()` without arguments in a class method is implicitly equivalent to `super(containing-class, method-self-argument)`. Whether implicit or explicit, this call form creates a bound proxy object that pairs the *`self`* instance with access to the calling class's location on the MRO of *`self`*'s class. This proxy object is usable for later superclass attribute references and method calls. See also "New-style Classes: MRO" for more on MRO ordering.
>
> Because `super()` always selects a *next* class on the MRO—the first class following the calling class having a requested attribute, whether it is a true superclass or not—it can be used for method call routing. In a *single-inheritance* class tree, this call may be used to refer to parent superclasses generically without naming them

explicitly. In *multiple-inheritance* trees, this call can be used to implement cooperative method-call dispatch that propagates calls through a tree.

The latter usage mode, cooperative method-call dispatch, may be useful in diamonds, as a conforming method call chain visits each superclass just once. However, `super()` can also yield highly implicit behavior which for some programs may not invoke superclasses as expected or required. The `super()` method dispatch technique generally imposes three requirements:

- *Anchors*: the method called by `super()` must exist—which requires extra code if no call-chain anchor is present.
- *Arguments*: the method called by `super()` must have the same argument signature across the entire class tree—which can impair flexibility, especially for implementation-level methods like constructors.
- *Deployment*: every appearance of the method called by `super()` but the last must use `super()` itself—which can make it difficult to use existing code, change call ordering, override methods, and code self-contained classes.

Because of these constraints, calling superclass methods by explicit superclass name instead of using `super()` may in some cases be simpler, more predictable, or required. For a superclass *`S`*, the explicit and traditional form *`S`*`.method(self)` is equivalent to the implicit `super().method()`. See also “New-Style inheritance algorithm” for more on the `super()` attribute lookup special case; case; instead of running full inheritance, its result objects scan a context-dependent tail portion of a class tree’s MRO, selecting the first matching descriptor or value.