# Acquisition



Acquisition [1] is a mechanism that allows objects to obtain attributes from their environment. It is similar to inheritence, except that, rather than traversing an inheritence hierarchy to obtain attributes, a containment hierarchy is traversed.

The ExtensionClass. release includes mix-in extension base classes that can be used to add acquisition as a feature to extension subclasses. These mix-in classes use the context-wrapping feature of ExtensionClasses to implement acquisition. Consider the following example:

```
import ExtensionClass, Acquisition

class C(ExtensionClass.Base):
  color='red'

class A(Acquisition.Implicit):

  def report(self):
    print self.color

a=A()
c=C()
c.a=A()

c.a.report() # prints 'red'

d=C()
d.color='green'
d.a=a

d.a.report() # prints 'green'

a.report() # raises an attribute error
```

The class A inherits acquisition behavior from Acquisition.Implicit. The object, a, "has" the color of objects c and d when it is accessed through them, but it has no color by itself. The object a obtains attributes from it's environment, where it's environment is defined by the access path used to reach a.

## 1.1 Acquisition wrappers

When an object that supports acquisition is accessed through an extension class instance, a special object, called an acquisition wrapper, is returned. In the example above, the expression c.a returns an acquisition wrapper that contains references to both c and a. It is this wrapper that performs attribute lookup in c when an attribute cannot be found in a.

Aquisition wrappers provide access to the wrapped objects through the attributes aq_parent, aq_self, aq_base. In the example above, the expressions:

'c.a.aq_parent is c'

and:

'c.a.aq_self is a'

both evaluate to true, but the expression:

'c.a is a'

evaluates to false, because the expression c.a evaluates to an acquisition wrapper around c and a, not a itself.

The attribute aq_base is similar to aq_self. Wrappers may be nested and aq_self may be a wrapped object. The aq_base attribute is the underlying object with all wrappers removed.

## 1.2 Acquisition Control

Two styles of acquisition are supported in the current ExtensionClass release, implicit and explicit aquisition.

### 1.2.1 Implicit acquisition

Implicit acquisition is so named because it searches for attributes from the environment automatically whenever an attribute cannot be obtained directly from an object or through inheritence.

An attribute may be implicitly acquired if it's name does not begin with an underscore, _.

To support implicit acquisition, an object should inherit from the mix-in class Acquisition.Implicit.

### 1.2.2 Explicit Acquisition

When explicit acquisition is used, attributes are not automatically obtained from the environment. Instead, the method aq_aquire must be used, as in:

```
print c.a.aq_acquire('color')
```

To support explicit acquisition, an object should inherit from the mix-in class Acquisition.Explicit.

### 1.2.3 Controlled Acquisition

A class (or instance) can provide attribute by attribute control over acquisition. This is done by:

- subclassing from Acquisition.Explicit, and
- setting all attributes that should be acquired to the special value: Acquisition.Acquired. Setting an attribute to this value also allows inherited attributes to be overridden with acquired ones.

  For example, in:

  ```
  class C(Acquisition.Explicit):
    id=1
    secret=2
    color=Acquisition.Acquired
    __roles__=Acquisition.Acquired
  ```

  The only attributes that are automatically acquired from containing objects are color, and __roles__. Note also that the __roles__ attribute is acquired even though it's name begins with an underscore. In fact, the special Acquisition.Acquired value can be used in Acquisition.Implicit objects to implicitly acquire selected objects that smell like private objects.

### 1.2.4 Filtered Acquisition

The acquisition method, aq_acquire, accepts two optional arguments. The first of the additional arguments is a "filtering" function that is used when considering whether to acquire an object. The second of the additional arguments is an object that is passed as extra data when calling the filtering function and which defaults to None.

The filter function is called with five arguments:

- The object that the aq_acquire method was called on,
- The object where an object was found,
- The name of the object, as passed to aq_acquire,
- The object found, and
- The extra data passed to aq_acquire.

If the filter returns a true object that the object found is returned, otherwise, the acquisition search continues.

For example, in:

```
from Acquisition import Explicit

class HandyForTesting:
   def __init__(self, name): self.name=name
   def __str__(self):
      return "%s(%s)" % (self.name, self.__class__.__name__)
   __repr__=__str__

class E(Explicit, HandyForTesting): pass

class Nice(HandyForTesting):
   isNice=1
   def __str__(self):
      return HandyForTesting.__str__(self)+' and I am nice!'
   __repr__=__str__

a=E('a')
a.b=E('b')
a.b.c=E('c')
a.p=Nice('spam')
a.b.p=E('p')

def find_nice(self, ancestor, name, object, extra):
   return hasattr(object,'isNice') and object.isNice

print a.b.c.aq_acquire('p', find_nice)
```

The filtered acquisition in the last line skips over the first attribute it finds with the name p, because the attribute doesn't satisfy the condition given in the filter. The output of the last line is:

```
spam(Nice) and I am nice!
```

## 1.3 Acquisition and methods

Python methods of objects that support acquisition can use acquired attributes as in the report method of the first example above. When a Python method is called on an object that is wrapped by an acquisition wrapper, the wrapper is passed to the method as the first argument. This rule also applies to user-defined method types and to C methods defined in pure mix-in classes.

Unfortunately, C methods defined in extension base classes that define their own data structures, cannot use aquired attributes at this time. This is because wrapper objects do not conform to the data structures expected by these methods.

## 1.4 Acquiring Acquiring objects

Consider the following example:

```
from Acquisition import Implicit

class C(Implicit):
   def __init__(self, name): self.name=name
   def __str__(self):
      return "%s(%s)" % (self.name, self.__class__.__name__)
   __repr__=__str__

a=C("a")
a.b=C("b")
a.b.pref="spam"
a.b.c=C("c")
a.b.c.color="red"
a.b.c.pref="eggs"
a.x=C("x")

o=a.b.c.x
```

The expression o.color might be expected to return "red". In earlier versions of ExtensionClass, however, this expression failed. Acquired acquiring objects did not acquire from the environment they were accessed in, because objects were only wrapped when they were first found, and were not rewrapped as they were passed down the acquisition tree.

In the current release of ExtensionClass, the expression "o.color" does indeed return "red".

When searching for an attribute in o, objects are searched in the order x, a, b, c. So, for example, the expression, o.pref returns "spam", not "eggs". In earlier releases of ExtensionClass, the attempt to get the pref attribute from o would have failed.

If desired, the current rules for looking up attributes in complex expressions can best be understood through repeated application of the __of__ method: a.x

and by keeping in mind that attribute lookup in a wrapper is done by trying to lookup the attribute in the wrapped object first and then in the parent object. In the expressions above involving the __of__ method, lookup proceeds from left to right.

Note that heuristics are used to avoid most of the repeated lookups. For example, in the expression: a.b.c.x.foo, the object a is searched no more than once, even though it is wrapped three times.

[1] Gil, J., Lorenz, D., Environmental Acquisition--A New Inheritance-Like Abstraction Mechanism OOPSLA '96 Proceedings, ACM SIG-PLAN, October, 1996