Inheritance 1

- *Inheritance* allows us to reuse the code of an existing class B in creating a new class C
- Let’s recap how the attribute lookup worked for classes
- When looking for an attribute, the lookup procedure starts with the instance dictionary, and continues with the class attributes
- If both fail, then the attribute is searched from the base classes and, recursively, from their base classes
So, it may look like we access an attribute of a class C, when in reality we are accessing the attribute of its base class B.

In this case we say that the class C *inherits* the attribute from its base class B.

If we have attributes with the same name in both the class and its base class, the attribute of the base class is hidden.

We say that the class C *overrides* the attribute of the base class B.

Terminology: B is a *base class* and C is a *derived class*.
Inheritance 3

Example:

class B(object):
    def f(self):
        print "Executing B.f"
    def g(self):
        print "Executing B.g"

class C(B):
    def g(self):
        print "Executing C.g"

x=C()
x.f()  # inherited from B
x.g()  # overridden by C
Relation of base and derived classes

- A derived class is sometimes also called a *subclass* and the base class is called *super class*
- The inheritance relation of two classes B and C can be tested with function `issubclass`:
  
  
  ```python
  issubclass(C,B)==True but issubclass(B,C)==False
  ```

- Function `isinstance(obj, cls)` allows us to test whether an instance has type `cls` or has an *ancestor* class of type `cls`

- Let’s create instances `x=C()` and `y=B()`

- Now we have `isinstance(x,B)==isinstance(x,C)==isinstance(y,B)==True`

- But `isinstance(y,C)==False`
Inheritance hierarchy

- object should be a base class or an ancestor class of every other class
- This means that `isinstance(x, object)==True` for all instances `x`
Use of inheritance

- By deriving from an existing class we can modify and/or extend its behaviour, without touching the original class.

- For example, if we want to add one method to a list class, we can use inheritance. Therefore we have to only code the part that has changed and reuse the rest of the code of type list.

- Another use of inheritance is to create conceptual hierarchies. For instance, later we will learn about the exception hierarchy of Python.

- Third use would be to use classes to create interfaces. There can be several classes that have same interface (that is, they offer the same attributes), but their behaviour or implementation can be very different. This allows changing a part of your program with minimal changes required elsewhere in the code.
A class can have several base classes:

```python
class Derived(Base1, Base2, ..., Basen):
```

This is called *multiple inheritance*, which is sometimes useful but it also creates problems.

Your class can usually only inherit from one standard type at the same time.

For instance, if you try to derive from both `list` and `dict` the result would be ambiguous.

Another problem is when the inheritance hierarchy is not tree-shaped.
The class A is now an ancestor of D through both B and C classes.

class A(object):
    def f(self):
        print "A"

class B(A):
    pass

class C(A):
    def f(self):
        print "C"

class D(B, C):
    pass
x=D()
x.f()  # what will this print
Will the call `x.f()` refer to `A.f` or `C.f`?

The order in which base classes are visited is called *Method Resolution Order (MRO)*.

Normally the lookup proceeds in left-to-right, depth-first order.

But in the diamond case we would meet `A.f` before `C.f`.

Extra rule: if a base class can be reached through many paths, the right-most path is chosen.

Therefore we will first see `C.f` which hides `A.f`.
Referring to base class attributes

- If in the definition of the method `C.f` we need to call the corresponding method of class `A`, we can use the fully qualified call `A.f(...)`
- This is called *delegation*
- It is useful, for instance, when you want to call the `__init__` method of the base class from the `__init__` of the derived class to initialise the base class attributes
Class-level methods 1

- Python provides two types of class-level methods. That is, methods that don’t need to refer to any instance; they don’t have self as the first parameter
- Usually all functions defined inside a class body are assumed to be normal methods
- A normal method can be turned into a static method by calling the built-in type staticmethod:

```python
class C(object):
    def f():
        print "Executing static method"
        f=staticmethod(f)

x=C()
C.f()  # can be called without an instance
x.f()  # But also works through an instance
```
Another class-level method, in addition to static method, is called *class method*

It can be created with the type call `classmethod`

A class method will have a special first parameter, which refers to the class of the method. This parameter is usually named `cls`

```python
class C(object):
    def f(cls):
        print("Executing class method of class %s" % cls)
        f=classmethod(f)
    x=C()
    C.f()  # can be called without an instance
    x.f()  # But also works through an instance
```
Descriptors are a way to control the access to attributes

Instead of using a normal attribute, we can use an instance of a descriptor class in place of the attribute

When we access the descriptor attribute, we are actually calling certain methods of the descriptor.

The methods for reading, setting, and deleting an attribute are `__get__`, `__set__`, and `__delete__`, respectively
Descriptors 1

The following descriptor class makes sure that we don’t assign a negative value to the corresponding attribute:

class Nonnegative(object):
    def __get__(self, x, C):
        print "Executing get"
        return x._v
    def __set__(self, x, value):
        print "Executing set"
        if value < 0:
            raise ValueError
        x._v=value
Let’s make the attribute `a` for a class `C` using the Nonnegative descriptor and then try to use it:

```python
class C(object):
    _v = 2
    v = Nonnegative()
x = C()
y = C()
x.v = 3  # will execute the `__set__` method
y.v = 5
x.v  # will execute the `__get__` method
y.v
x.v = -2 # will cause an error
```

Properties

- A *property* is an attribute of an instance that triggers special function calls when the attribute is read, set or deleted.
- One can create properties with the type call `property(fget, fset, fdel, doc)` and binding the return value to an attribute name.
- The functions `fget`, `fset` and `fdel` will be called when corresponding operations, reading, setting, or deleting the attribute are tried.
- Properties are actually more user friendly versions of descriptors: we don’t have to create a separate descriptor class.
Creating a property

An example of creating a property `a` for the class `C`:

```python
class C(object):
    def __init__(self, value):
        self.a_ = value
    def seta(self, value):
        if value < 0:
            raise ValueError
        self.a_ = value
    def geta(self):
        return self.a_
    a = property(geta, seta)
```
Accessing a property

Let’s try to access the attribute a of an instance x:

```python
>>> x=C(2)
>>> x.a
2
>>> x.a=5
>>> x.a
5
>>> x.a = -2  # this will cause an error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "property.py", line 8, in seta
    def geta(self):
        ValueError
```
Properties with synthetic attributes 1

- An example of an attribute whose value is computed when needed

```python
class Rectangle(object):
    def __init__(self, w, h):
        self.width = w
        self.height = h

    def getarea(self):
        print "Executing getarea"
        return self.width * self.height

area = property(getarea,
                doc="Computes the area on the fly")
```
Properties with synthetic attributes 2

Let’s try to access this attribute:

```python
>>> r=Rectangle(4,5)
>>> r.area
20
>>> Rectangle.area.__doc__
'Computes the area on the fly'
```
We have already encountered one *special method*, namely the `__init__` method.

- This method sets the instance attributes to some initial value.
- Its first parameter is the `self`, and the subsequent parameters are the ones that were passed to the call of the class.
- The `__init__` method should return no value.
- In the following slides the main general purpose special methods are introduced.
- They are executed when certain operations on objects are performed.
Objects and classes

General purpose special methods 1

- In the following, C is a class and x and y are its instances
- `__hash__` returns an int value, with the following requirement: `x == y` implies `x.__hash__() == y.__hash__()`. The value is used in storing objects in dictionaries and sets. The instances x and y must be immutable
- A class with `__call__` method makes its instances callable. I.e. the call `x(a, b, ...)` will result in calling this special method with the given parameters
- The method `__del__` gets called when the corresponding instance gets deleted
- Method `__new__` is used to control the creation of new instances. It can be used, for example, to create classes that have only one instance.
The method `__str__` is called when the print statement needs to print the value of an instance. It returns a string. The print-format expression calls this for conversion `%s`.

The method `__repr__` is called when the interactive interpreter prints the value of an evaluated expression, and when the conversion `%r` for print-format expression is used. Returns a canonical representation string that (at least in theory) can be used to recreate the original object.

Special methods `__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, and `__ne__` get called when the corresponding operators `x==y`, `x>=y`, `x>y`, `x<=y`, `x<y`, and `x!=y` are used.