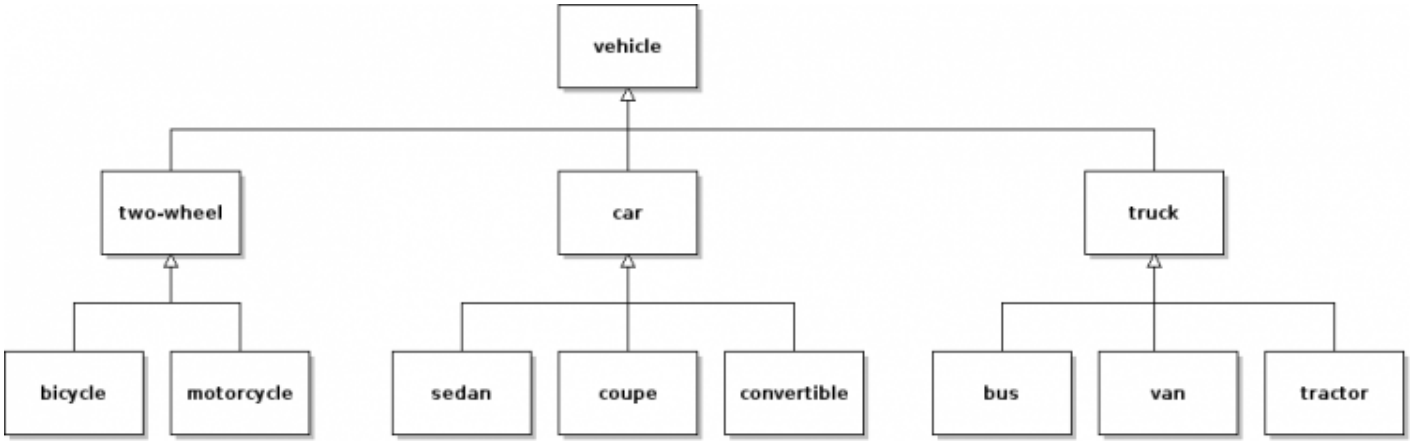
About Python IV

Advanced OO concepts

Any language that supports object orientation will let you create objects, either by instantiating them from classes or by building them from prototypes. Class-based object orientation becomes significantly more powerful when you are able to use **inheritance** and **polymorphism**. It is also a great convenience if you can use multiple **constructors**. Python permits all these.

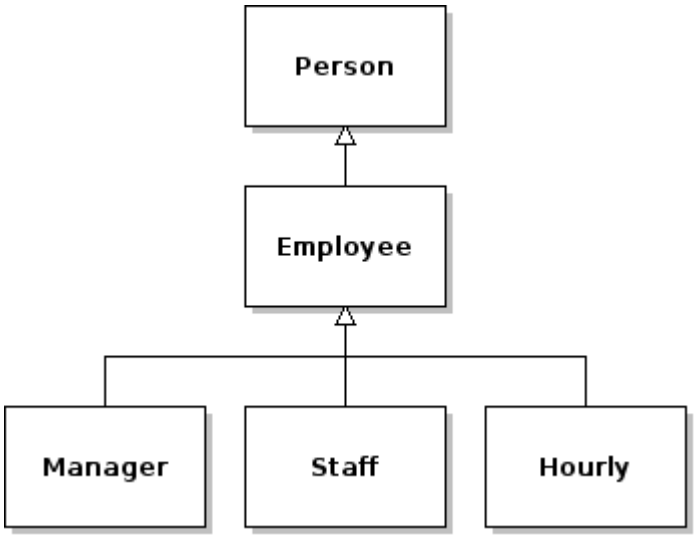
Inheritance

In the real world, we often create hierarchies of things. For example, a *vehicle* is “a machine that is used to carry people or goods from one place to another”.[1)] Based on this general concept, we may define a category of *two-wheel* vehicles that includes *bicycles* and *motorcycles*, a category called *car* that includes *sedans*, *coupes*, and *convertibles*, a *truck* category that includes *buses*, *vans*, *tractors*, and so on.

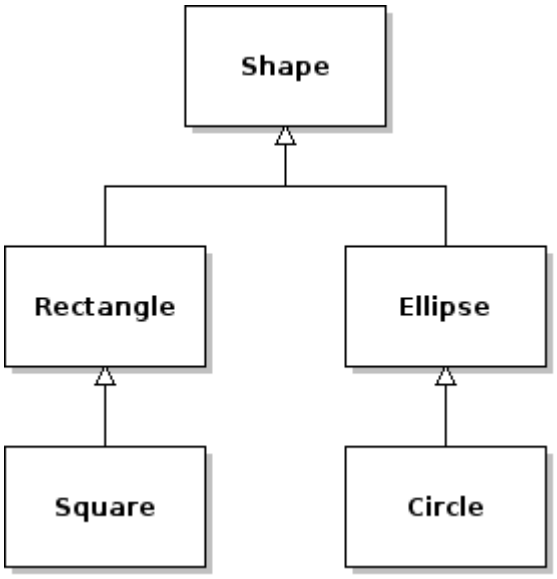


In these kinds of hierarchies, we typically start with a general class of things at the root of the tree, and all the other classes of things are more specialized versions of the general class of things.

This is the essence of inheritance in object oriented programming. For example, to solve a particular programming problem, we might define a `Person` class, and based on that definition we might then define an `Employee` class, and then based on that `Employee` class we might define `Manager`, `Staff`, and `Hourly` employee classes.



Or, we might define a general `Shape` class. Then based on the `Shape` class we might define `Rectangle` and `Ellipse` classes. Then based on the `Rectangle` class we might define a `Square` class (a square is rectangle with equal height and width), and based on the `Ellipse` class we might define a `Circle` (a circle is an ellipse with zero eccentricity).



Generalization and specialization

In both of the above cases, the classes closest to the root of the tree are more general than the classes toward the bottom. A `Square` is a special kind of `Rectangle` and a `Rectangle` is a special kind of `Shape`. An `Hourly` employee is a special kind of `Employee`, and an `Employee` is a special kind or `Person`. For this reason, we often say that inheritance defines *"is a"* relationships.

Class heirarchies

We call trees of classes like the above **class heirarchies**. The class at the root of the tree is a **base class**, and classes that inherit from a base classes are **derived classes**.[2)]

Inheritance and code reuse

One of the advantages of implementing classes with inheritance is **code reuse**. Almost all class-based object oriented languages let you define derived classes *without having to re-write the base class code*. The idea is that you write the base class code *once*. Then when you write the derived classes, you add only whatever new code is required for the derived class or new definitions for old code that must be **overridden**.

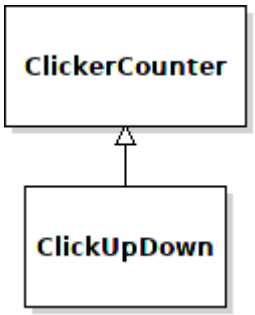
Since you do not need to rewrite the code that is common to both base and derived classes, you end up writing less code. But even more important, when you fix a bug in the base class, it automatically propagates to the derived classes.

Inheritance and polymorphism

Another advantage of using inheritance is that with statically typed languages (such as C++ and Java), it facilitates **polymorphism** (discussed below). Dynamically typed languages (like Python) behave polymorphically almost by definition.

Inheritance in Python

To demonstrate the use of inheritance in Python, we are going to create a specialized version of `ClickerCounter` called `ClickUpDown`.



In addtion to the two buttons found on a `ClickerCounter` (for incrementing and resetting the count), a `ClickUpDown` object will add another that resets the count. In terms of a Python model, a `ClickUpDown` is identical to a `ClickerCounter` except that it has an additional method: `click_down`.

Here is the base class:

```
# base class definition
class ClickerCounter():
    def __init__(self):
```

```
        self.count = 0

    # accessor for count
    def get_count(self):
        return self.count

    # click the counter
    def click(self):
        self.count = self.count + 1

    # reset the count
    def reset(self):
        self.count = 0
```

And here is the derived class:

```
# derived class definition
class ClickUpDown(ClickerCounter):
    # click down the counter
    def clickdown(self):
        self.count = self.count - 1
```

That's it!

Using the new class:

```
b = ClickUpDown()       # instantiate a ClickUpDown object
b.click()               # 1
b.click()               # 2
b.click()               # 3
b.clickdown()           # should be 2
print b.get_count()

a = ClickerCounter()    # instantiate a ClickerCounter object
a.click()               # 1
a.click()               # 2
a.click()               # 3
a.clickdown()           # can't do that!
print a.get_count()
```

Polymorphism

In statically typed programming languages, the behavior of an object bound to a variable might be:
a) the behavior associated with the class of the variable used to reference the object, or
b) the behavior associated with the class of the object to which the variable is bound.

We call b) above **polymorphism**. In other words, when a language behaves polymorphically, the behavior of objects is based on the *class to which the object belongs*—not to the *class of the variable by which you are accessing the object*.

Polymorphism in Python

Because Python is dynamically typed, Python is polymorphic without any extra effort on the part of the programmer. When variables are bound to objects, the type of the variable changes—therefore the behavior is always determined by the object's class.

```
a = ClickUpDown()       # a now references a ClickUpDown object
a = ClickerCounter()    # a now references a ClickerCounter object
```

In fact, it's hard to make Python behave non-polymorphically.

Parameterized constructor example

Let's now add a `click_limit` feature to our `ClickerCounter`. When the user clicks past the `click_limit`, the count will automatically reset to zero. By default, the `click_limit` will be 100,000. Let us also let the user set the `click_limit` when she or he instantiates a `ClickerCounter`.

To do this, we will need to add an instance variable to store the `click_limit`, add some logic to the `click` method, and add a parameterized constructor:

```
class ClickerCounter():
    # parameterized constructor
    def __init__(self, click_limit = 100000):
        self.count = 0
        self.click_limit = click_limit

    # accessor for count
    def get_count(self):
        return self.count

    # click the counter
    def click(self):
        if self.count < self.click_limit:
            self.count = self.count + 1
        else:
            self.reset()

    # reset the count
    def reset(self):
        self.count = 0
```

To use it:

```
a = ClickerCounter(3)       # make a clicker-counter that counts up to 3
a.click()
a.click()
a.click()
print a.get_count()         # should be 3
a.click()                   # should go from 3 to 0
print a.get_count()         # should be 0
```


1)

“Merriam-Webster's Learner's Dictionary.” Merriam-Webster's Learner's Dictionary. http://www.learnersdictionary.com/search/vehicle (accessed February 10, 2011).

2)

Synonyms for **base class** are **superclass** and **parent class**. Synonyms for **derived class** are **subclass** and **child class**. You should generally stick to one pair—(base class:derived class), (superclass:subclass), or (parent class:child class).