What is inheritance?

- Inheritance is the mechanism for creating a class that modifies or specializes an existing class.
- The existing class is the base class or superclass.
- The new class is the derived class or subclass.
- The subclass inherits all attributes of the superclass (data and methods).
- The subclass can redefine (override) anything it inherits.

Recall our Account class

```python
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
    def deposit(self, amt):
        self.balance += amt
    def withdraw(self, amt):
        self.balance -= amt
    def inquiry(self):
        return self.balance
```

A class Derived from Account

```python
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0, 4) == 1:
            return self.balance * 1.25
        else:
            return self.balance
```

20% of the time, return an artificially high balance to “help” the account holder overdraw their account which increases fees paid to Wells Fargo the bank.
How does it work?

An access (dot operator) of an attribute (data or method) for an instance of EvilAccount, looks first in EvilAccount. If the item isn’t found there, Python looks in the superclass, Account.

It doesn’t stop there. If the item isn’t found in Account, Python looks in Account’s superclass. This continues until reaching object, the root of the class tree.

Adding to EvilAccount

```python
class EvilAccount(Account):
    def __init__(self, name, balance, ef):
        Account.__init__(self, name, balance)
        self.evil_factor = ef
    def inquiry(self):
        if random.randint(0, 4) == 1:
            return self.balance * self.evil_factor
        else:
            return self.balance
```

Create an EvilAccount object

```python
checking2 = EvilAccount('Jane', 10000, 1.1):
checking2.inquiry()
checking2.deposit(100)
checking2.inquiry()
```

EvilerAccount

```python
class EvilerAccount(EvilAccount):
    def deposit(self, dep):
        self.withdraw(5)         # convenience fee
        EvilAccount.deposit(dep)
```

Explicit invocation of superclass deposit method
**EvilerAccount**

A more general version:
```python
class EvilerAccount(EvilAccount):
    def deposit(self, dep):
        self.withdraw(5)  # convenience fee
        super(EvilerAccount, self).deposit(dep)
```

You don't have to explicitly name the superclass.

---

**Multiple Inheritance**

**What is it?**

Allows a class to inherit features from more than one superclass.

**Why do we need it?**

Some types exhibit characteristics of multiple other types so extending them both is desirable/necessary.

---

**The Details**

- `super(cls, instance)` allows attribute lookup in base classes.
  - `cls` is the current class
  - `instance` is an instance of `cls`
  - It uses normal lookup rules

- **Python3 difference alert:**
  - Python3 uses the far nicer syntax: `super().attribute`

---

**Example**

An F22 Raptor is a fighter jet.
It has some characteristics of a plane
It has some characteristics of a weapon

Plane alone doesn't describe it because not all planes are weapons.

Weapon alone doesn't describe it because not all weapons are planes.
Multiple Inheritance:

Consider these two new classes related to Account:

```python
class DepositCharge(object):
    fee = 5
    def deposit_fee(self):
        self.withdraw(self.fee)

class WithdrawCharge(object):
    fee = 3
    def withdraw_fee(self):
        self.withdraw(self.fee)
```

Multiple Inheritance:

Let's use these new classes:

```python
d = MoreEvilerAccount('Tim', 100000000, 1.2)
d.deposit(10)
d.inquiry()
```

Multiple Inheritance:

Now we create a new account type:

```python
class MoreEvilerAccount(EvilAccount, DepositCharge, WithdrawCharge):
    def deposit(self, amt):
        self.deposit_fee()
        super(MoreEvilerAccount, self).deposit(amt)
    def withdraw(self, amt):
        self.withdraw_fee()
        super(MoreEvilerAccount, self).withdraw(amt)
```

Multiple Inheritance:

Hmm, this is subtle:

```python
d = MoreEvilerAccount('Tim', 100000000, 1.2)
d.deposit_fee()  # DepositCharge.deposit_fee() fee is $5
```

What's going on here?

```python
d = MoreEvilerAccount('Tim', 100000000, 1.2)
d.deposit_fee()  # DepositCharge.deposit_fee() fee is $5
```

WithdrawCharge.withdraw_fee fee is...? $5 !??
**Explanation**

- Fee is a class variable. It is defined twice: once in DepositCharge and once in WithdrawCharge.
- The value in DepositCharge was used in both calls. Why? Because the order of classes listed in the class definition for MoreEvilerAccount defines a priority. DepositCharge is listed before WithdrawCharge so it has higher priority.

**Polymorphism**

**What is it?**

Within the context of inheritance, ability to use an instance without regard to its type.

**Why do we need it?**

To simplify implementations of subclasses: there is no need to override every attribute of superclass A in subclass B since instances of B can directly access attributes in A.

**Polymorphism**

**How does it work?**

It is handled entirely by the attribute lookup process.

Consider class C that is a subclass of B. B is, in turn, a subclass of A.

Let c be an instance of C. In a reference to c.attr, attr is searched in:

- instance c
- class C
- class B
- class A
- class object