

The background features a collection of abstract geometric shapes, including triangles and thin lines, in dark blue, gold, and light gray, scattered across the white background.

物件導向

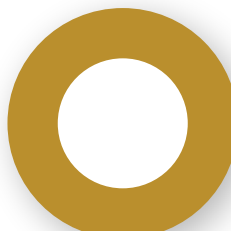

Object-oriented programming
OOP



簡介

物件導向(OOP)為使用物件的程式設計，物件是指將類別(class)實體化後的東西。

物件與類別的關係，就例如房屋藍圖(class)與房子(object)，需要先建立好房屋藍圖(class)後，再用此藍圖實際做出真正的房子(object)。

- 容易擴展
 - 可重複使用，減少撰寫重複的程式碼
 - 模組化，可將工作細分給不同團隊
- 
- 

目錄

1

類別與物件

Class & Object

2

繼承

Inheritance

3

多型

polymorphism

4

封裝

Encapsulation



PART

01



物件與類別

Class & Object



定義類別



class 類別名稱:

initializer

methods

● 第一種: 類別裡沒有建構子, 僅有方法

● 第二種: 類別裡有建構子及方法

```
class Animal:
    def callname(self):
        print("這是一隻動物!")
```

方法

```
animal1 = Animal()
animal1.callname()
```

第一種

```
class Animal:
    def __init__(self, name):
        self.name = name

    def callname(self):
        print(self.name + "是一隻動物!")

animal1 = Animal("獅子")
animal1.callname()
```

建構子

方法

第二種

```
class Animal:
    def __init__(self, name):
        self.x = name

    def callname(self):
        print(self.x + "是一隻動物!")

animal1 = Animal("獅子")
animal1.callname()
```

第二種

定義類別



self參數

self是參考物件本身的參數，使用 self參數可以存取類別中的物件成員。

- self.x與self.y的有效範圍為紅框處
- z的有效範圍為黃框處

```
class ClassName:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def m1(self, ...):
        self.y = 2
        z = 5
        ....

    def m2(self, ...):
        self.y = 10
        ....
```

建立物件



變數 = 類別名()

第一種: `animal1 = Animal()`

第二種: `animal1 = Animal("獅子")`

```
class Animal:
    def callname(self):
        print("這是一隻動物！")

animal1 = Animal()
animal1.callname()
```

```
class Animal:
    def __init__(self, name):
        self.name = name

    def callname(self):
        print(self.name + "是一隻動物！")

animal1 = Animal("獅子")
animal1.callname()
```

使用物件的方法



變數.方法()

第一種: `animal1.callname()`

第二種: `animal1.callname("獅子")`

```
class Animal:
    def callname(self):
        print("這是一隻動物！")

animal1 = Animal()
animal1.callname()
```

```
class Animal:
    def __init__(self, name):
        self.name = name

    def callname(self):
        print(self.name + "是一隻動物！")

animal1 = Animal("獅子")
animal1.callname()
```




PART

02



繼承

Inheritance



繼承

程式中會有父類別與子類別，子類別可擁有父類別的屬性(attribute)與方法(method)，主要是用來降低程式碼重複性。



單一繼承



多重繼承

繼承



單一繼承

- Animal類別有一個建構子與一個方法
- Dog類別繼承Animal類別
- 當x2去呼叫callName方法時，由於自己沒有這個方法，則會往上找父類別是否有此方法

```
class Animal:
    #建構
    def __init__(self, name):
        self.name = name
    #方法
    def callName(self):
        print(self.name + "是一隻動物！")

class Dog(Animal):
    def __init__(self, name):
        self.name = name

x1 = Animal("馬來貘")
x1.callName()

x2 = Dog("奧樂雞")
x2.callName()
```

繼承



多重繼承

- 繼承順序會影響輸出結果
- 搜尋方式會由自己這層開始找, 沒找到的話會往上找父類別

```
class A:
    def mymethod(self):
        print("爺爺喜歡打羽球")

class B(A):
    def mymethod(self):
        print("爸爸喜歡打高爾夫球")

class C(A):
    def mymethod(self):
        print("媽媽喜歡游泳")

class D(C, B):
    pass
```

D().mymethod()

媽媽喜歡游泳

```
class A:
    def mymethod(self):
        print("爺爺喜歡打羽球")

class B(A):
    def mymethod(self):
        print("爸爸喜歡打高爾夫球")

class C(A):
    def mymethod(self):
        print("媽媽喜歡游泳")

class D(B, C):
    pass
```

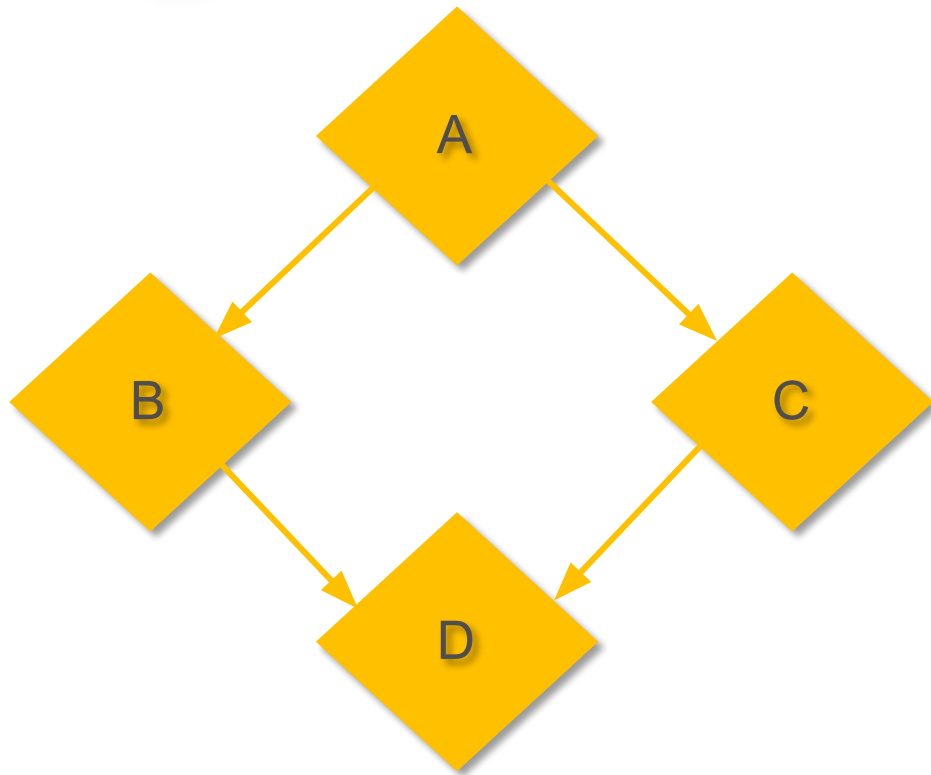
D().mymethod()

爸爸喜歡打高爾夫球

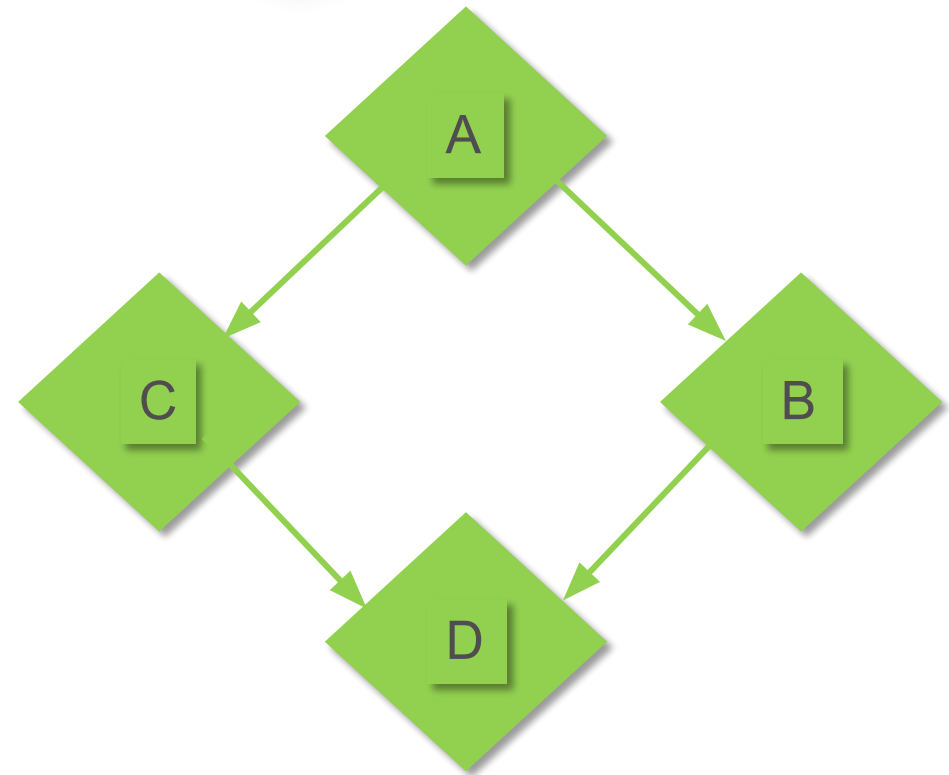
繼承



多重繼承狀況一



多重繼承狀況二



繼承



super

- EmailPerson繼承person
- EmailPerson除了繼承父類別的建構子, 還另外新增一個參數email, 故此時需要用super來繼承父類別的屬性

```
class Person():  
    def __init__(self, name):  
        self.name = name  
  
class EmailPerson(Person):  
    def __init__(self, name, email):  
        super().__init__(name)  
        self.email = email  
  
x1 = EmailPerson("mark", "mark@gmail.com")  
  
print(x1.name)      → mark  
print(x1.email)     → mark@gmail.com
```



PART

03



多型

polymorphism



多型

呼叫相同名稱的方法，但會得到不同的結果。



不同類別有相同方法名稱



多型是指方法的多型，並非屬性的多型



[補充資料一](#)



[補充資料二](#)

多型



三個類別皆有callName方法



x1、x2、x3分別呼叫
callName方法皆有不同結果

```
class Animal:
    #建構
    def __init__(self, name):
        self.name = name
    #方法
    def callName(self):
        print(self.name + "是一種動物！")

class Dog(Animal):
    def callName(self):
        print(self.name + "是一隻狗")

class Cat(Animal):
    def callName(self):
        print(self.name + "是一隻貓")

x1 = Animal("獅子")
x1.callName()
x2 = Dog("小白")
x2.callName()
x3 = Cat("小花")
x3.callName()
```

獅子是一種動物！

小白是一隻狗

小花是一隻貓

多型

```
1 class A{
2     void hello(){
3         System.out.print("沒有參數");
4     }
5
6     void hello(int i){
7         System.out.print("有1個int參數");
8     }
9
10    void hello(int i, int j){
11        System.out.print("有2個int參數");
12    }
13
14    void hello(int i, String s){
15        System.out.print("有1個int、1個字串參數");
16    }
17
18    void hello(String s) {
19        System.out.print("有1個字串參數");
20    }
21 }
22
23 public class unit3_6 {
24     Run | Debug
25     public static void main(String[] args) {
26         A a1 = new A();
27         a1.hello();
28         System.out.println();
29         a1.hello(5,3);
30         System.out.println();
31         a1.hello(10, "apple");
32     }
33 }
```

```
class Animal:
    #建構
    def __init__(self, name):
        self.name = name
    #方法
    def callName(self):
        print(self.name + "是一種動物!")

class Dog(Animal):
    def callName(self):
        print(self.name + "是一隻狗")

class Cat(Animal):
    def callName(self):
        print(self.name + "是一隻貓")

x1 = Animal("獅子")
x1.callName()
x2 = Dog("小白")
x2.callName()
x3 = Cat("小花")
x3.callName()
```

Python不允許相同方法名稱、不同參數形式



PART

04



封装

Encapsulation



封裝

物件中包含本身操作所需要的資訊，此物件不需要依賴其他物件就可以完成操作。



封裝



私有屬性



私有方法

封裝



封裝

- x1.name是以封裝的方式程式，可以直接讓外部使用
- x1.age則是以外部給予參數的方式呈現，這邊是使用公開方法

```
class Info:

    def __init__(self):
        self.name = "mark"
        self.age = 18

    def my_age(self, age):
        self.age = age

    def phonenumber(self):
        print("0910123456")

x1 = Info()
x1.my_age(30)

print("我的名字：" + x1.name)
print("我的年齡：" + str(x1.age))
x1.phonenumber()
```

我的名字：mark
我的年齡：30
0910123456

封裝



私有屬性

- 將建構子的self.name修改為 self.__name
- 要將屬性調整為private, 則在屬性前增加 __

```
class Info:
    def __init__(self):
        self.__name = "mark"
        self.age = 18

    def my_age(self, age):
        self.age = age

    def phonenumber(self):
        print("0910123456")
```

```
x1 = Info()
x1.my_age(30)

print("我的名字：" + x1.name)
print("我的年齡：" + str(x1.age))
x1.phonenumber()
```

AttributeError: 'Info' object has no attribute 'name'

- 若要呼叫私有屬性, 需使用 __類別名__私有屬性

```
class Info:
    def __init__(self):
        self.__name = "mark"
        self.age = 18

    def my_age(self, age):
        self.age = age

    def phonenumber(self):
        print("0910123456")
```

```
x1 = Info()
```

```
print("我的名字：" + x1._Info__name) 我的名字：mark
```

封裝



私有方法

- 在方法前增加 `__`
- 若要呼叫私有方法, 需使用 `__類別名__私有方法`

```
class Info:

    def __init__(self):
        self.__name = "mark"
        self.age = 18

    def my_age(self, age):
        self.age = age

    def __phonenumber(self):
        print("0910123456")
```

```
x1 = Info()
x1.__phonenumber()
x1._Info__phonenumber()
```

`AttributeError: 'Info' object has no attribute '__phonenumber'`

0910123456

封裝



Get/Set

- 分別在方法中新增get與set方法
- 但取用方法時太過複雜，故有另一種寫法

```
class Person:

    def __init__(self, age):
        self._age = age

    def get_age(self):
        return self._age

    def set_age(self):
        if(isinstance(self,str)):
            self._age = int(self.age)
        elif(isinstance(self,int)):
            self._age = age

p = Person("18")
p.set_age()
print (p.get_age())
```


封裝

- ✓ property 定義屬性
- ✓ 方法名.setter 設定屬性
- ✓ 方法名.deleter 刪除屬性

```
class Person:

    def __init__(self, age):
        self._age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self):
        if(isinstance(self,str)):
            self._age = int(age)
        elif(isinstance(self,int)):
            self._age = age

    @age.deleter
    def age(self):
        del self._age

p = Person("18")
print (p.age)
```