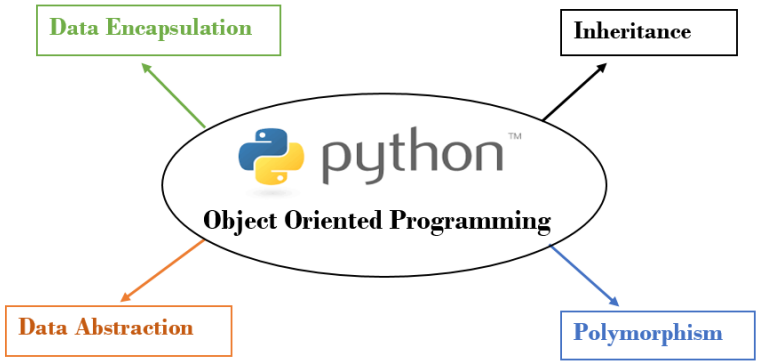


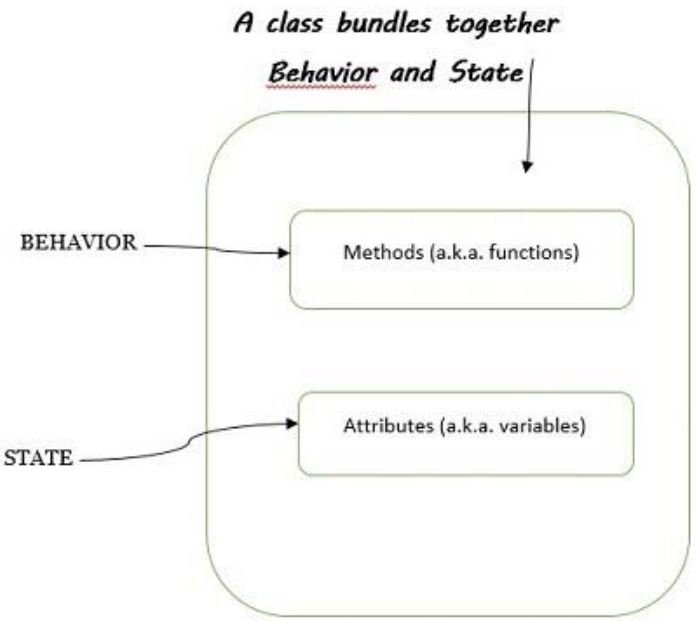
Object Oriented Programming

Object Oriented Programming

- Python是物件導向的語言。因此建立和使用類別與物件是非常容易的。
- OOP術語
  - 類別：用於定義表示使用者定義物件的一組屬性的原型。屬性是透過點符號存取的資料成員(類別變數和範例變數)和方法。
  - 類別變數：由類別的所有範例共用的變數。類別變數在類中定義，但在類別的任何方法之外。類別變數不像範例變數那樣頻繁使用。
  - 資料成員：儲存與類別及其物件相關聯的資料的類別變數或範例變數。
  - 函式過載：將多個行為分配給特定函式。 執行的操作因涉及的物件或引數的型別而異。
  - 範例變數：在方法中定義並僅屬於類別的當前範例的變數。
  - 繼承：將類別的特徵傳遞給從其衍生的其他類別。
  - 範例(實體、物件)：某個類別的單個物件。 例如，物件obj屬於Circle類別，它是Circle類別的範例。
  - 範例(實體)化：建立類別的範例。
  - 方法：在類別定義中定義的一種特殊型別的函式。
  - 物件：由其類別定義的資料結構的唯一範例。物件包括資料成員(類別變數和範例變數)和方法。
  - 運算子過載：將多個函式分配給特定的運算子。

Object Oriented Programming

- 類別(class)：使用者自建的資料型別
  - 類別：行為和狀態(屬性)
    - 一個類別將允許將物件的行為和狀態(屬性)綑綁在一起。觀察下圖以更好地理解：



Object Oriented Programming

- 類別(class)：使用者自建的資料型別
  - 討論類別時，以下幾點需要注意：
    - **行為(behavior)與函式相同**： 它是一段執行某些操作(或實現行為)的程式碼，
    - **狀態(state)與變數相同**：它是一個在類別中儲存值的地方。
    - 當宣告一個類別的**行為**和**狀態**時，它是一個類別中的**函式**和**變數**。

Object Oriented Programming

- 類別(class)：使用者自建的資料型別
  - 類別具有方法和屬性
    - 在Python中，建立方法定義了一個類別行為。方法是在一個類中定義的函式提供的OOP名稱。 歸納如下：
      - 類別函式：是方法的同義詞
      - 類別變數： 是名稱屬性的同義詞。
      - 類別：具有確切行為的範例的藍圖。
      - 物件：類別的一個範例，執行類中定義的功能。
      - 型別：表示範例所屬的類別
      - 屬性：任何物件值：object.attribute
      - 方法：類別中定義的「可呼叫屬性」

Object Oriented Programming

- 類別(class)：使用者自建的資料型別
  - 實例(instance)：類別產生的實體變數，與物件(object)同義
  - 屬性(attribute)：類別所使用的資料
    - 類別屬性(class attribute)：所有類別物件共享的資料
    - 實例屬性(instance attribute)：個別物件獨自擁有的資料
  - 方法(method)：定義於類別內的函式
    - 實例方法(instance method)：由類別物件所執行的方法
    - 類別方法(class method)：由類別本身所執行的方法
    - 靜態方法(static method)：可由類別或類別物件所執行的方法

Object Oriented Programming

- 建立類別
  - class語句建立一個新的類別定義。類別的名稱緊跟在class關鍵字之後，在類別的名稱之後緊跟冒號，如下：

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

  - 該類別有一個文件字串，可以通過ClassName.__doc__存取。
  - class_suite由定義類別成員、資料屬性和函式的所有元件語句組成。

Object Oriented Programming

- 建立類別
  - 以下是一個簡單的Python類別的例子：

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)
```

  - 變數empCount是一個類別變數，其值在此類別中的所有範例之間共用。這可以從類別或類別之外的Employee.empCount存取。
  - 第一個方法__init __()是一種特殊的方法，當建立此類別的新範例時，該方法稱為Python建構函式或初始化方法。
  - 宣告其他類別方法，如正常函式，但每個方法的第一個引數是**self**。Python將self引數新增到列表中，呼叫方法時不需要包含它。

Object Oriented Programming

- 建立範例物件
  - 要建立類別的範例，可以使用類名呼叫該類別，並傳遞其__init__方法接受的任何引數。

```
## This would create first object of Employee class
emp1 = Employee("Maxsu", 2000)
## This would create second object of Employee class
emp2 = Employee("Kobe", 5000)
```

- 存取屬性
  - 可以使用帶有物件的點(.)運算子來存取物件的屬性。類別變數將使用類別名稱存取如下：

```
emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)
```

Object Oriented Programming

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)

## This would create first object of Employee class
emp1 = Employee("Maxsu", 2000)
## This would create second object of Employee class
emp2 = Employee("Kobe", 5000)

emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)
```

```
Name :  Maxsu ,Salary:  2000
Name :  Kobe ,Salary:  5000
Total Employee 2
```

Object Oriented Programming

– 可以隨時新增，刪除或修改類別和物件的屬性：

```
emp1.salary = 7000  # Add an 'salary' attribute.
emp1.name = 'xyz'  # Modify 'age' attribute.
del emp1.salary  # Delete 'age' attribute.
```

– 如果不是使用普通語句存取屬性，可以使用以下函式：

- getattr(obj, name, [default])：存取物件的屬性。
- hasattr(obj, name)：檢查屬性是否存在。
- setattr(obj, name, value)：設定一個屬性。如果屬性不存在，那麼它將被建立。
- delattr(obj, name)：刪除一個屬性。

```
hasattr(emp1, 'salary')          # Returns true if 'salary' attribute exists
getattr(emp1, 'salary')          # Returns value of 'salary' attribute
setattr(emp1, 'salary', 7000)    # Set attribute 'salary' at 7000
delattr(emp1, 'salary')          # Delete attribute 'salary'
```

Object Oriented Programming

- 內建類別屬性
  - 每個Python類別保持以下內建屬性，並且可以像任何其他屬性一樣使用點運算子存取它們：
    - __dict__：字典儲存類別或物件所有的屬性資料
      - obj.__dict__：儲存實例屬性
      - class.__dict__：儲存類別屬性與類別方法
    - __doc__：代表定義於類別名稱後的字串(可跨列)，通常用來當成類別的說明文字。為類別屬性，全名為 class.__doc__
    - **__name__：類別名稱。**
    - __module__：定義類別的模組名稱。**此屬性在互動模式下的值為「main」。**
    - __bases__：一個包含基礎類別的空元組，按照它們在基礎類別列表中出現的順序。

Object Oriented Programming

- 內建類別屬性

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)

emp1 = Employee("Maxsu", 2000)
emp2 = Employee("Bryant", 5000)
print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__ )
```

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {
   'displayCount': <function Employee.displayCount at 0x0160D2B8>,
   '__module__': '__main__', '__doc__': 'Common base class for all employees',
   'empCount': 2, '__init__':
   <function Employee.__init__ at 0x0124F810>, 'displayEmployee':
   <function Employee.displayEmployee at 0x0160D300>,
   '__weakref__':
   <attribute '__weakref__' of 'Employee' objects>, '__dict__':
   <attribute '__dict__' of 'Employee' objects>
}
```

Object Oriented Programming

- 銷毀物件(垃圾收集)
  - Python自動刪除不需要的物件(內建型別或類別範例)以釋放記憶體空間。Python定期回收不再使用的記憶體區塊的過程稱為垃圾收集。
  - Python的垃圾收集器在程式執行期間執行，當物件的參照計數達到零時觸發。物件的參照計數隨著指向它的別名數量而變化。
  - 當物件的參照計數被分配一個新名稱或放置在容器(列表，元組或字典)中時，它的參照計數會增加。當用**del**刪除物件的參照計數時，參照計數減少，其參照被重新分配，或者其參照超出範圍。當物件的參照計數達到零時，Python會自動收集它。

Object Oriented Programming

- 銷毀物件(垃圾收集)

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

  - 通常情況下，垃圾回收器會銷毀孤立實體並回收其空間。
  - 但是，類別可以實現呼叫**解構函式**的特殊方法__del__()，該方法在物件即將被銷毀時被呼叫。 此方法可能用於清理物件使用的任何非記憶體資源。

Object Oriented Programming

- 銷毀物件(垃圾收集)
  - 這個__del__()解構函式列印要被銷毀的物件的類別名稱：

```
pt1 = Point()
pt2 = pt1
pt3 = pt1
print (id(pt1), id(pt2), id(pt3));   # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

```
3083401324 3083401324 3083401324
Point destroyed
```

  - 理想情況下，應該在單獨的檔案中定義類別，然後使用import語句將其匯入主程式檔案。
  - 在上面的例子中，假定Point類別的定義包含在Point.py中，並且其中沒有其他可執行程式碼。

```
import Point
p1 = Point.Point()
```

Object Oriented Programming

- 封裝
  - 封裝是物件導向的基礎之一。OOP能夠以下列方式隱藏對開發人員有利的物件內部工作的複雜性 -
    - 簡化並使得在不知道內部結構的情況下使用物件變得容易理解。
    - 任何更改都可以很容易地管理。
  - 物件導向程式設計在很大程度上依賴於封裝。**術語封裝和抽象(也稱為資料隱藏)通常用作同義詞。它們幾乎是同義詞，因為抽象是通過封裝來實現的**。
  - 封裝提供了限制存取某些物件元件的機制，這意味著物件內部表示無法從物件定義的外部看到。存取這些資料通常透過特殊方法來實現的：**Getters和Setters**。
  - 這些資料儲存在範例屬性中，可以在類別以外的任何位置進行操作。為了保護它，**只能使用範例方法存取該資料。不應允許直接存取。**

Object Oriented Programming

- 封裝

```
class MyClass(object):
    def setAge(self, num):
        self.age = num

    def getAge(self):
        return self.age

## 範例化物件
zack = MyClass()
zack.setAge(45)
print(zack.getAge())
zack.setAge("Fourty Five")
print(zack.getAge())
```

```
45
Fourty Five
```

  - 只有在資料正確且有效的情況下，才能使用例外處理結構來儲存資料。正如上面所看到的，使用者對setAge()方法的輸入沒有限制。它可以是字串，數位或列表。因此，需要檢查上面的程式碼以確保儲存的正確性。

Object Oriented Programming

- 建構函式
  - 建構函式是一種特殊型別的方法(函式)，它在類別的實體化物件時被呼叫。建構函式通常用於初始化(賦值)給範例變數(物件)。建構函式還驗證有足夠的資源來使物件執行任何啟動任務。
- 建立一個建構函式
  - 建構函式是以雙底線(__)開頭的類別函式。建構函式的名稱是__init__()。
  - 建立物件時，如果需要，建構函式可以接受引數。當建立沒有建構函式的類別時，Python會自動建立一個不執行任何操作的預設建構函式。
  - **每個類別必須有一個建構函式，即使它只依賴於預設建構函式。**

Object Oriented Programming

- 建構函式
  - 建立一個名為ComplexNumber的類別，它有兩個函式__init__()函式來初始化變數，並且有一個getData()方法用來顯示數位。

```
class ComplexNumber:

    def __init__(self, r = 0, i = 0):
        """初始化方法"""
        self.real = r
        self.imag = i

    def getData(self):
        print("{0}+{1}j".format(self.real, self.imag))

if __name__ == '__main__':
    c = ComplexNumber(5, 6)
    c.getData()
```

```
5+6j
```

Object Oriented Programming

- 建構函式
  - 可以為物件建立一個新屬性，並在定義值時進行讀取。

```
class ComplexNumber:

    def __init__(self, r = 0, i = 0):
        """初始化方法"""
        self.real = r
        self.imag = i

    def getData(self):
        print("{0}+{1}j".format(self.real, self.imag))

if __name__ == '__main__':
    c = ComplexNumber(5, 6)
    c.getData()

    c2 = ComplexNumber(10, 20)

    # 試著賦值給一個未定義的屬性
    c2.attr = 120
    print("c2 = > ", c2.attr)

    print("c.attr => ", c.attr)
```

```
5+6j
c2 = >  120
Traceback (most recent call last):
  File "D:\test.py", line 23, in <module>
    print("c.attr => ", c.attr)
AttributeError: 'ComplexNumber' object has no attribute 'attr'
```

Object Oriented Programming

- 方法過載
  - 方法過載是指具有多個接受不同引數集的同名方法。
  - 給定單個方法或函式預設引數，可以指定自己的引數數量。根據函式定義，可以使用零個或一個引數來呼叫它。

```
class Human:
   def sayHello(self, name = None):
      if name is not None:
         print('Hello ' + name)
      else:
         print('Hello ')

#Create Instance
obj = Human()

#Call the method, else part will be executed
obj.sayHello()

#Call the method with a parameter, if part will be executed
obj.sayHello('Rahul')
```

```
Hello
Hello Rahul
```

Object Oriented Programming

- 函式也是物件
  - 可呼叫物件是一個物件可以接受一些引數，並可能返回一個物件。**函式是Python中最簡單的可呼叫物件**，但也有其他類似於類別或某些類別範例。
  - **Python中的每個函式都是一個物件**。物件可以包含方法或函式，但物件不是必需的函式。

```
def my_func():
    print('My function was called')

def second_func():
    print('Second function was called')

def another_func(func):
    print('The name: ', end=' ')
    print(func.__name__)
    print('The class:', end=' ')
    print(func.__class__)
    print("Now I'll call the function passed in")
    func()

another_func(my_func)
another_func(second_func)
```

```
The name:  my_func
The class: <class 'function'>
Now I'll call the function passed in
My function was called
The name:  second_func
The class: <class 'function'>
Now I'll call the function passed in
Second function was called
```

Object Oriented Programming

- 分數類別**範例**(**物件建構與其字串表示方式**)
  - __init__ 起始方法：用來建構類別物件，不需回傳(或回傳 None)
  - __str__ 字串表示方法：用來代表物件內容的字串

```
class Fraction :

    # 起始方法：用以建構物件，不需回傳
    def __init__( self , n = 0 , d = 1 ) :
        self.num , self.den = n , d

    #字串表示方法：設定物件的「字串」輸出樣式
    def __str__ ( self ) :
        if self.den == 1 :
            return str(self.num)
        else :
            return str(self.num) + '/' + str(self.den)
```

Object Oriented Programming

- 分數類別**範例**(**物件建構與其字串表示方式**)
  - 以上兩方法皆是 Fraction 類別的實例方法
  - 實例方法的第一個參數都是類別物件本身，通常以 self 表示
  - self 所儲存的資料為個別物件所獨有，稱為實例屬性
  - __init__與__str__ 方法名稱前後皆有雙底線，此為 python預設的方法名稱
  - 物件的屬性與數量可依問題自由設定
  - python 並沒有如 C++ 同等的私有成員，但若屬性或方法名稱是以一個底線起始，習慣上都被當成私有成員
  - **方法名稱不能與屬性同名**
  - **方法名稱若有重複，則後來讀取的方法會覆蓋先前的**
  - 每當設計新類別時，最好先由這兩個方法開始設計：起始方法用來產生物件，字串表示方法用來呈現物件屬性是否正確。

Object Oriented Programming

- 分數類別**範例**(**物件建構與其字串表示方式**)
  - 執行方式

```
# 自動使用 Fraction._ _init_ _ 產生物件
a = Fraction() # 預設 分子為 0 ,分母為 1
b = Fraction(4) # 分子 4，分母預設為 1
c = Fraction(2,3) # 分子 2 ，分母 3

#自動使用 Fraction._ _str_ _ 列印物件所對應的「字串」
print( a , b , c)
```

```
0 4 2/3
```

Object Oriented Programming

- 分數類別**範例**(**類別的實例方法**)
  - 第一個參數為物件

```
class Fraction :
    # 設定分子與分母
    def set_val( self , n , d = 1 ) :
        self.num , self.den = n , d

    # 取得分子 分母

    def get_num( self ) :
        return self.num

    def get_den( self ) :
        return self.den

    # 計算分數倍數
    def mul( self , m ) :
        return Fraction(self.num*m,self.den)
```

Object Oriented Programming

- 分數類別**範例**(**類別的實例方法**)
  - 兩種執行方式
    - obj.method(arg2,arg3,...)：**obj 被自動設定為第一個參數(arg1)**

```
a = Fraction() # a = 0 參考第 1 頁
a.set_val(2,5) # 重新設定為 2/5

print( a.get_num() ) # 印出 a 的分子 2
a.set_val() # 錯誤，少了分子參數
```

    - 物件屬性也可直接透過屬性名稱更改數值，**但不建議使用**

```
b = Fraction()
b.num , b.den = 3, 4 # 直接設定 b 物件分子與分母
print( b ) # 印出：3/4
```

Object Oriented Programming

- 分數類別**範例**(**類別的實例方法**)
  - 兩種執行方式
    - class.method(obj,arg2,arg3,...)

```
a = Fraction() # a = 0
Fraction.set_val(a,3,7) # 重新設定為 3/7
print( a.mul(2) ) # 輸出 6/7
```

    - 類別的起始方法不適用第二種型式，不能直接使用以下型式建構全新物件

```
# 以下 b 名稱皆為第一次使用
Fraction.__init__(b,3,7) # 錯誤，b 未定義
Fraction(b,3,7) # 錯誤，b 未定義
```

Object Oriented Programming

- 分數類別**範例**(**內建類別屬性**)
  - 可透過 obj._ _dict_ _ 字典直接更改物件內部屬性，**但不建議使用**

```
a = Fraction(3,5) # a 為 3/5
a.num = 4 # 等同 a._ _dict_ _['num'] = 4
a.__dict__['den'] = 7 # 等同 a.den = 7
print(a) # 印出 4/7
```

  - **以 obj._ _dict_ _ 方式任意修改物件屬性不利程式開發**

Object Oriented Programming

- 分數類別**範例**(**雙底線起始的屬性或方法**)
  - 類別物件無法直接使用
  - 名稱前加上 _ClassName 則可為類別物件取用
  - **可直接給類別其他方法使用**

```
# 雙底線開始的方法名稱，不能由物件直接取用
def __inverse( self ) :
    return Fraction(self.den,self.num)

# 雙底線開始的方法名稱可由類別其他方法使用
def inv( self ) :
    return self.__inverse()
```

Object Oriented Programming

- 分數類別**範例**(**雙底線起始的屬性或方法**)

```
a = Fraction(2,3)
# 錯誤，無此方法
print ( a.__inverse () )
# 正確，原雙底線方法名稱前被加上 _Fraction
print ( a._Fraction__inverse () )
# 正確
print ( a.inv() ) # 輸出：3/2
```

  - 類別內雙底線起始的屬性或方法與 C++ 設定的「**私有**成員」仍有差異

Object Oriented Programming

- 分數類別**範例**(**類別方法：類別共享的方法**)
  - 類別方法為 @classmethod 裝飾器(decorator)，以 @ 起始
  - 類別方法的第一個參數為類別本身，通常以 cls 表示
  - 類別方法使用 class.method(...) 來執行
  - 類別方法經常用來**定義不同型式的物件產生方式**，使用 return 回傳類別物件

Object Oriented Programming

- 分數類別**範例**(**類別方法：類別共享的方法**)

```
# 由字串轉換來的分數
@classmethod
def fromstr( cls , fstr ) :
    if fstr.isdigit() :
        num , den = int(fstr) , 1
    else :
        num , den = map( int , fstr.split('/') )
    return cls(num,den)

# 帶分數型式
@classmethod
def mixed_fraction( cls , a = 0 , n = 0 , d = 1 ) :
    num , den = a * d + n , d
    return cls(num,den)

# 分數資料說明
@classmethod
def data_doc( cls ) :
    return "num:分子 , den:分母"
```

Object Oriented Programming

- 分數類別**範例**(**類別方法：類別共享的方法**)
  - 使用方式：

```
# 以下三個 Fraction 被自動設為類別方法的第一個參數
a = Fraction.fromstr("5")
b = Fraction.fromstr("4/7")
c = Fraction.mixed_fraction(2,3,4)
# 印出：5 4/7 11/4
print( a , b , c )
# 印出：num:分子 , den:分母
print( Fraction.data_doc() )
```

  - 裝飾器(decorator)為包裝函式的函式，用來擴充被包裝函式的功能。
  - @classmethod 將在其後定義的方法當成預設的 classmethod 方法的參數送入執行

Object Oriented Programming

- 分數類別**範例**(**類別屬性與靜態方法**)
  - 類別屬性：類別各物件所共用的資料
    - 類別屬性是**物件共用**的，非個別物件的屬性
    - 使用 class.attribute 方式執行
  - 靜態方法：**類別或物件共享**的方法
    - 靜態方法為 @staticmethod 裝飾器
    - 靜態方法的第一個參數非物件或類別
    - 若方法不需取用物件或類別屬性，但「性質」上歸類為類別，則設計為靜態方法
    - 使用 class.staticmethod(...) 或 obj.staticmethod(...) 方式執行
    - 在實務上，靜態方法不常使用

Object Oriented Programming

- 簡單計程車里程計費(**類別屬性與靜態方法**)
  - 類別屬性：計程車的里程計費資料適用所有計程車物件
  - 類別方法：根據駕駛距離計算距離
  - 靜態方法：提供類別計費資料

Object Oriented Programming

- 簡單計程車里程計費(**類別屬性與靜態方法**)

```python
class Taxi :
    # 類別屬性
    idis , udis , ifee , ufee = 1000 , 500 , 20 , 10

    # 實例方法
    def __init__( self , d = 0 ) :
        self.dis = d

    # 類別方法
    @classmethod
    def charge( cls , dis ) :
        if dis < cls.idis :
            return cls.ifee
        else :
            return cls.ifee + cls.ufee * (1+(dis-cls.idis) //cls.udis)

    # 實例方法
    def fee( self ) :
        return Taxi.charge(self.dis)

    # 實例方法
    def __str__( self ) :
        return "距離: " + str(self.dis) + " m"

    # 靜態方法
    @staticmethod
    def fee_rule() :
        return """idis : 初始里程 udis : 單位里程ifee : 初始費用 ufee : 單位里程費用"""

# 程式碼由此開始執行
taxies = [ Taxi(200*i) for i in range(5,21) ]
print( Taxi.fee_rule() )

for car in taxies :
    # car.fee() 與 Taxi.charge(car.dis) 相同
    print( car , "-->" , car.fee() , "NT" )
    #print( car , "-->" , Taxi.charge(car.dis), "NT" )
```

```
idis : 初始里程 udis : 單位里程ifee : 初始費用 ufee : 單位里程費用
距離: 1000 m --> 30 NT
距離: 1200 m --> 30 NT
距離: 1400 m --> 30 NT
距離: 1600 m --> 40 NT
距離: 1800 m --> 40 NT
距離: 2000 m --> 50 NT
距離: 2200 m --> 50 NT
距離: 2400 m --> 50 NT
距離: 2600 m --> 60 NT
距離: 2800 m --> 60 NT
距離: 3000 m --> 70 NT
距離: 3200 m --> 70 NT
距離: 3400 m --> 70 NT
距離: 3600 m --> 80 NT
距離: 3800 m --> 80 NT
距離: 4000 m --> 90 NT
```

Object Oriented Programming

- 類別的外部函式
  - 函式若與類別/實例屬性或方法無關則可定義在類別外部
  - 類別外部的函式可給檔案內其它程式碼所使用

```
# 計算兩數的 gcd
def gcd( a , b ) :
    a , b = abs(a) , abs(b)
    if a > b :
        return gcd(a%b,b) if a%b else b
    else :
        return gcd(b%a,a) if b%a else a

class Fraction :
    def __init__ ( self , n = 0 , d = 1 ) :
        self.num , self.den = n , d

    #字串表示方法：設定物件的「字串」輸出樣式
    def __str__ ( self ) :
        if self.den == 1 :
            return str(self.num)
        else :
            return str(self.num) + '/' + str(self.den)
    # 計算最簡分數
    def simplest_form( self ) :
        g = gcd(self.num,self.den)
        return Fraction(self.num//g,self.den//g)

a = Fraction(16,28)
# 印出：4/7
print( a.simplest_form() )
```

gcd 函式與分數屬性並無直接關係，但分數運算經常用到 gcd 函式，gcd 函式可設定為分數方法，歸類為分數類別的靜態方法

Object Oriented Programming

- 使用時機

名稱	語法	使用方式	使用時機
實例方法	`def method(obj,...)`	`obj.method(...)` 或 `class(obj,...)`	需使用實例屬性
類別方法	`@classmethod` `def method(cls,...)`	`cls.method(...)`	需使用類別屬性
靜態方法	`@staticmethod` `def method(...)`	`obj.method(...)` 或 `class.method(...)`	不需用到物件與類別屬性，但方法在性質上與類別相關
外部函式	`fn(...)`	`fn(...)`	不需使用物件與類別屬性

  - 如果方法同時需使用到實例屬性與類別屬性，則應設定成**實例方法**

Object Oriented Programming

- 物件指定
  - 指定代表原物件多了個名稱
  - 指定後兩物件為相同物件直到其中一物件變成新物件

```
class Fraction :
    def __init__ ( self , n = 0 , d = 1 ) :
        self.num , self.den = n , d

    #字串表示方法：設定物件的「字串」輸出樣式
    def __str__ ( self ) :
        if self.den == 1 :
            return str(self.num)
        else :
            return str(self.num) + '/' + str(self.den)

    # 設定分子與分母
    def set_val( self , n , d = 1 ) :
        self.num , self.den = n , d

a = b = Fraction(3,4) # a 與 b 為同一個物件
b.set_val(5,6) # a 與 b 都是 5/6
print( a is b ) # True
a = Fraction(1,2) # a 為 1/2 , b = 5/6，a 為新物件
print( a is b ) # False
```

Object Oriented Programming

- 資料隱藏(Private變數)
  - 物件的屬性在類別定義之外可能或不可見。需要使用雙下劃線字首命名屬性，然後**這些屬性將不會直接對外部可見**。

```
class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print (self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
print (counter.__secretCount)
```

```
1
2
Traceback (most recent call last):
   File "test.py", line 12, in <module>
      print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Object Oriented Programming

- 資料隱藏(Private變數)
  - 透過內部更改名稱來包含類別名稱來保護這些成員。如果將最後一行替換為以下，那麼是可以存取object._className__attrName等屬性：

```
print (counter._JustCounter__secretCount)
```

```
class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print(self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
#print(counter.__secretCount)
print(counter._JustCounter__secretCount)
```

```
1
2
2
```

Object Oriented Programming

- 存取權限控制
  - 在物件導向設計中, 資料與程式碼被封裝在類別裡面, 目的是可限制外部程式碼對類別成員的存取權限, 以提升軟體之強固性 (robustness) 與隱蔽性 (privacy), 存取權限分成三類：
    - 公開 (public)：**不限制存取, 外部程式碼也可自由存取, 也會被子類別繼承(regular_lower_case)**
    - 保護 (protected)：**只限類別內可存取, 也會被子類別繼承(_single_leading_underscore)**
    - 私有 (private)：**只限類別內可存取, 不會被子類別繼承(__ double_leading_underscore)**
  - 因為 Python 管控存取權限是透過對成員的名稱做特定的規範，所以 Python 類別的成員在預設情況下都是公開的，非公開之成員是對名稱做如下之特別處理：
    - 保護 (protected)：以**一個底線**開頭 (protected)，例如 _name 或 _showInfo()
    - 私有 (private)：以**兩個底線**開頭 (private) ，例如 __name 或 __showInfo()

Object Oriented Programming

- 存取權限控制
  - **不過 Python 的保護其實只是掩耳盜鈴，防君子不防小人，其所謂的保護只是著眼於一般人較少會用底線開頭來為成員命名，從而被直接存取的機會較少而已**，例如：

```
class MyClass():
    def __init__(self, a="hello", b="world"):      # 設定參數之預設值
        self._a=a                   # 被保護的屬性
        self.__b=b                # 私有的屬性
    def showInfo(self):
        print("a =", self._a, "b =", self.__b)     # 透過 self 物件存取其屬性

myobj=MyClass()
print(myobj._a)      # 外部程式碼仍可存取被保護的屬性

myobj._a=123         # 被保護的屬性可以被更改
print(myobj._a)

myobj.showInfo()     # 屬性 _a 真的被改變了
```

  - 可見**只要知道屬性名稱是單底線開頭就可以自由地存取它，資料並沒有真正被隱藏**。

Object Oriented Programming

- 存取權限控制
  - **反觀雙底線開頭的屬性則會被隱藏，外部程式碼無法直接存取它，只能透過公開的方法取得其值**:

```
print(myobj.__b)      # 外部程式碼無法存取私有的屬性
myobj.showInfo()      # 私有屬性只能透過公開方法取得
```

  - 如果**直接改變私有屬性之值, 雖然不會出現錯誤, 但其實並沒有真的被改變** :

```
myobj.__b=456         # 企圖更改私有屬性之值
print(myobj.__b)      # 檢視似乎值被改變了

myobj.showInfo()       # 呼叫公開方法 showInfo() 顯示並沒有被改變
```

Object Oriented Programming

- 存取權限控制
  - 雖然私有變數無法被外部存取，必須透過公開的物件方法，但可否像存取屬性那樣而非呼叫函式? 可以的，只要利用 @property 修飾器將此物件方法轉成屬性模式即可。
  - 具體作法是定義一個傳回私有變數值的公開物件方法，其名稱為私有變數去掉前面的雙底線，然後在此物件方法前面添加 @property 修飾器將此方法變身為屬性：

```
class MyClass():
    def __init__(self, a="Hello"):
        self.__a=a

    @property                   # 修飾器, 將方法 a() 修飾為物件屬性
    def a(self):
        return self.__a         # 傳回私有屬性之 值

myobj=MyClass()

print(dir(myobj))               # 檢視物件內容
```

```
['_MyClass__a', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'a']
```

  - 此類別定義了一個與私有變數 __a 與同名的物件方法 a()，其內容只是單純地傳回私有屬性之值 (唯讀)，然後在 a() 方法前面加上 @property 修飾器，這樣外部程式碼就可以用 .a 屬性讀取私有屬性 __a 之值，而非呼叫 a() 方法了。

Object Oriented Programming

- 存取權限控制
  - 可見除了真實名稱為 _MyClass__a 的私有變數 __a 外，還多了一個 a 方法，但它已被綁定到屬性，故要以屬性的方式存取，而不是呼叫方法, 例如：

```
myobj.a                    # 讀取私有屬性值
#myobj.a()                 # 被修飾器綁定為屬性後無法被呼叫 (not callable)
#del myobj.a               # 無法刪除私有屬性
#myobj.a="World"           # 無法設定私有屬性之值
```

  - **但只能唯讀，不可更改，也不可刪除**

Object Oriented Programming

- 存取權限控制**範例**

```
class GetSet(object):

   instance_count = 0 # public

   __mangled_name = 'no privacy!' # special variable

   def __init__(self, value):
      self._attrval = value # _attrval is for internal use only
      GetSet.instance_count += 1

   @property
   def var(self):
      print('Getting the "var" attribute')
      return self._attrval

   @var.setter
   def var(self, value):
      print('setting the "var" attribute')
      self._attrval = value

   @var.deleter
   def var(self):
      print('deleting the "var" attribute')
      self._attrval = None

cc = GetSet(5)
cc.var = 10 # public name
print(cc.var)
print(cc._attrval)
print(cc._GetSet__mangled_name)
del cc.var
```

```
setting the "var" attribute
Getting the "var" attribute
10
10
no privacy!
deleting the "var" attribute
```

Object Oriented Programming

- 檔案與模組
  - 每個 python 檔案自成一個模組(module)，**模組名稱為去除副檔名的檔案名稱**
  - 模組內定義的物件、函式、類別自成一個使用區域
  - import 可將其他模組併入使用
    - import foo ：

    併入 foo.py 檔案於程式內，使用所有 foo 模組定義的名稱前**需加上**「foo.」

    - from foo import * ：

    併入 foo.py 檔，可直接使用 foo 模組定義名稱，**不需加上**「foo.」

    - from foo import a ：

    僅併入 foo.py 檔的 **a** 於程式內，使用時**不需加上**「foo.」

    - from foo import a , b , ... ：

    僅併入 foo.py 檔的 **a , b** , ... 於程式內，使用時**不需加上**「foo.」

    - import foo , bar , ... ：

    一次併入 foo.py、bar.py、... 等多個檔案

Object Oriented Programming

- __name__模組名稱字串
  - 字串儲存模組名稱
  - **檔案若為起始執行檔，則 __name__ 字串自動設定為 "__main__"**
  - 操作範例：
    - bar.py 檔使用 import foo 將 foo.py 檔併入程式內：

bar.py 檔案

```
import foo
print( __name__ )          # 輸出: __main__
print( foo.__name__ )      # 輸出: foo
```

Object Oriented Programming

- __name__模組名稱字串
  - 操作範例：
    - 定義兩個檔案 foo.py 與 bar.py

foo.py 檔案

```
def foo() : print( "foo" )

print( "foo:" , __name__ )
```

bar.py 檔案

```
import foo
print( "bar:" , foo.__name__ )
print( "bar:" , __name__ )
```

    - 分別執行 foo.py 與 bar.py 得到以下輸出結果：

執行	foo.py	bar.py
輸出	foo: __main__	foo: foo bar: foo bar: __main__

Object Oriented Programming

- __name__模組名稱字串
  - __name__ 可用來建構程式測試區塊
    - **在程式開發階段常需測試程式，為避免測試用的程式與已有的程式區塊混雜在一起，難以區分，可藉由檢查 __name__ 值是否等於"__main__" 分離測試程式為獨立一區**

foo.py 檔案

```
# A 程式區塊 : 開發完成程式區塊
...
pass
if __name__ == "__main__" :        # 判斷 foo.py 是否為起始執行檔
    # B 程式區塊: 測試用程式區塊
    pass
```

- 當 foo.py 為起始執行檔案時，會執行 B 程式區塊
- 當 foo.py 不是起始執行檔案時，則跳過 B 程式區塊

Object Oriented Programming

- __name__模組名稱字串
  - 可搭配自行定義的 main 函式達到類似 C 語言主函式 main() 的效果

foo.py 檔

```
# A 程式區塊 : 開發完成程式區塊
...
pass
# 定義類似 C 語言的主函式
def main() :
    pass
if __name__ == "__main__" :
    # B 程式區塊: 測試用程式區塊
    main()
```

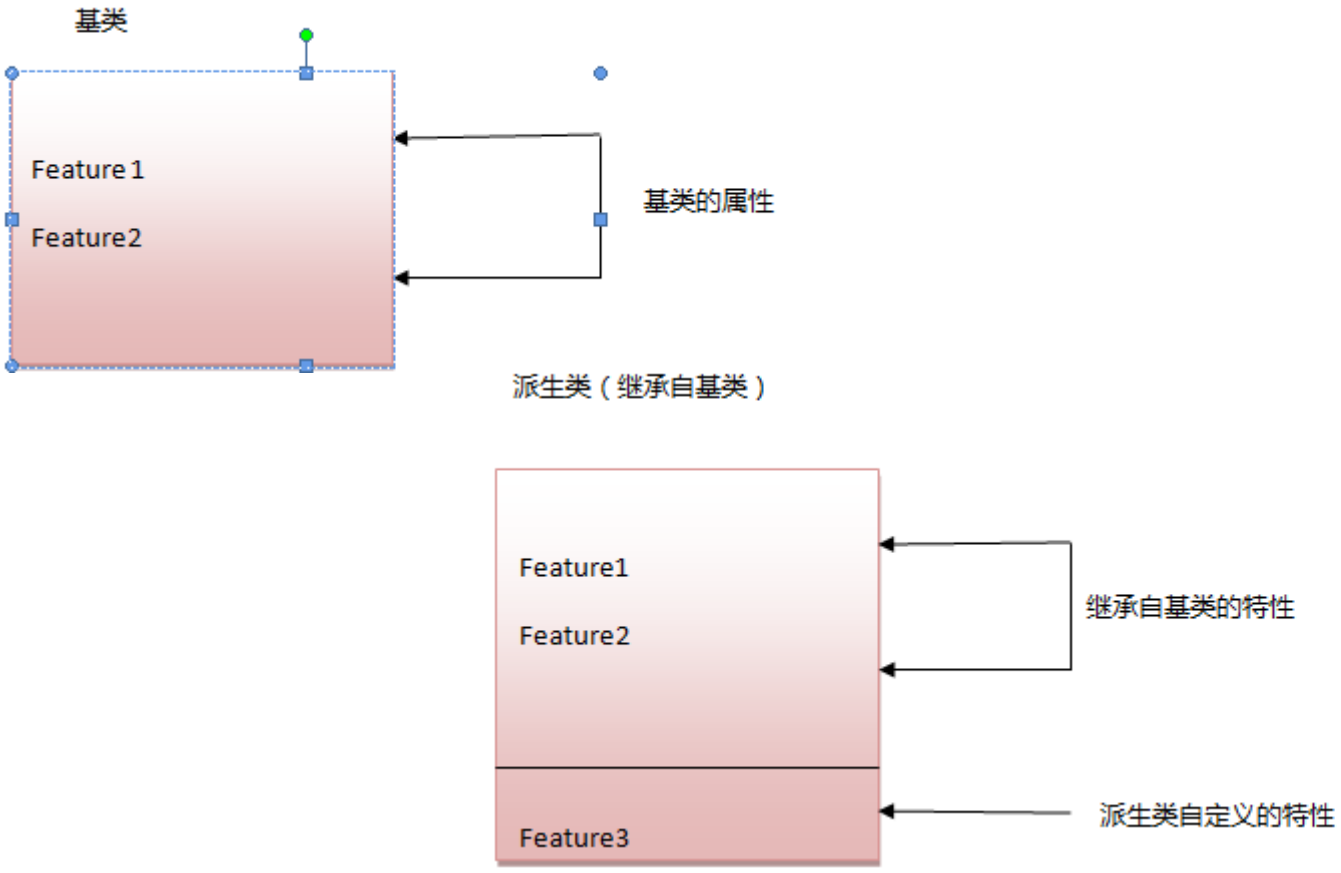
  - 當 foo.py 為起始執行檔案時，會執行 main() 函式開始執行
  - 當 bar.py 檔 import foo 時，可在 bar.py 檔內使用 foo.main() 執行 foo 模組的 main() 函式

Object Oriented Programming

- 繼承
  - 繼承用於指定一個類別將從其父類別獲取其大部分或全部功能。它是物件導向程式設計的一個特徵。物件導向程式設計的一個主要優勢是**重用**。繼承是實現這一目標的機制之一。
  - 這是一個非常強大的功能，方便使用者對現有類別進行幾個或多個修改來建立一個新的類別。新類別稱為子類別或衍生類別，從其繼承屬性的主類稱為基礎類別或父類別。
  - **繼承允許先建立一個通用類別或基礎類別，然後再將其擴充套件為更專門化的類別。**
  - 使用類別繼承不用從頭開始構建程式碼，可以通過在新類別名**後面的括號**中列出父類別來從一個預先存在的類別衍生它來建立一個類別。子類別繼承其父類別的屬性，可以像子類別中一樣定義和使用它們，並向其新增新功能。它有助於程式碼的可重用性。
  - 在物件導向的術語中，當類別X擴充套件類別Y時，則Y別稱為超級/父/基礎類別，X稱為子類別/子/衍生類別。這裡需要注意一點，只有資料欄位和非專用方法才能被子類別存取。私有資料欄位和方法只能在類別中存取。

Object Oriented Programming

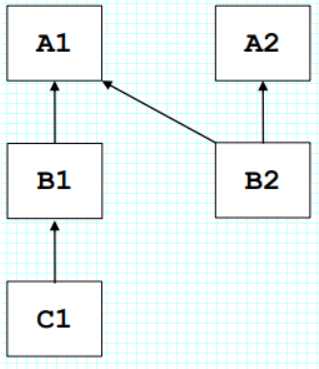
- 繼承



Object Oriented Programming

- 繼承
  - 基礎類別(Base class)為被繼承類別或稱為超類別(superclass)、父類別(parent class)
  - 衍生類別(Derived class)為繼承類別或稱為次類別(subclass)、子類別(child class)
  - 類別架構：透過類別繼承組合的類別關係

```
class A1 : pass
class A2 : pass
# 單一繼承: B1 繼承自 A1, A1 為父類別,
            B1 為子類別
class B1(A1) : pass
# 多重繼承: B2 同時繼承自 A1 與 A2
class B2(A1,A2) : pass
# 單一繼承: C1 繼承自 B1
class C1(B1) : pass
```



Object Oriented Programming

- 繼承語法
  - 衍生類別被宣告為很像它們的父類別，然而，繼承的基礎類別的列表在類別名稱之後給出：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

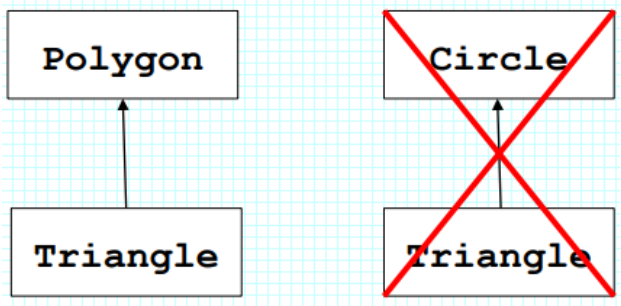
```
class BaseClass:
   Body of base class
class DerivedClass(BaseClass):
   Body of derived class
```

  - 引數說明
    - 必須在包含衍生類別定義的範圍中定義名稱BaseClassName。還可以使用其他任意表示式代替基礎類別名稱。當在另一個模組中定義基礎類別時要指定模組的名稱。

```
class DerivedClassName(modulename.BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Object Oriented Programming

- 繼承(**基礎類別與衍生類別**)
  - 衍生類別物件可視同基礎類別物件
  - 衍生類別物件擁有基礎類別物件的所有性質
  - 衍生**類別方法**也可使用衍生/基礎類別內的所有方法
  - 衍生**類別物件**也可使用衍生/基礎類別內的所有方法



Object Oriented Programming

- 繼承(**類別繼承的好處**)
  - 重複使用已有程式
  - 簡化未來程式開發
  - 較好的程式延伸性
  - 降低維護開發費用
  - 類別使用相同介面
  - 方便建立**程式庫**

Object Oriented Programming

- 繼承例子

```
class Date(object):
    def get_date(self):
        return "2018-06-30"

class Time(Date):
    def get_time(self):
        return "09:09:09"

dt = Date()
print("Get date from Date class: ", dt.get_date())

tm = Time()
print("Get time from Time class: ", tm.get_time())
print("Get date from class by inheriting or calling Date class method: ", tm.get_date())
```

```
Get date from Date class:  2018-06-30
Get time from Time class:  09:09:09
Get date from class by inheriting or calling Date class method:  2018-06-30
```

  - 首先建立了一個名為Date的類別，並將該物件作為引數傳遞，之後建立了另一個名為time的類別，並將Date類稱為引數。透過這個呼叫，可以存取Date類別中的所有資料和屬性。正因為如此，建立的Time類別物件tm中獲取父類別中get_date方法。
  - Object.Attribute查詢層次結構
    - 物件
    - 當前類別
    - 該類繼承的任何父類別

Object Oriented Programming

- 繼承例子

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print ("Calling parent constructor")

   def parentMethod(self):
      print ('Calling parent method')

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class
   def __init__(self):
      print ("Calling child constructor")

   def childMethod(self):
      print ('Calling child method')

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Object Oriented Programming

- 繼承例子(**多邊形與三角形**)

```
cno = "零一二三四五六七八九"

# 平面點類別
class Point :
    def __init__( self , x = 0 , y = 0 ) : self.x , self.y = x , y
    def __str__( self ) : return "({},{})".format(self.x,self.y)

# 多邊形類別
class Polygon :
    def __init__( self , pts ) : self.pts = pts
    def __str__( self ) : return " ".join( [ str(pt) for pt in self.pts ] )
    def name( self ) : return cno[len(self.pts)] + "邊形"

# 三角形：使用多邊形的起始設定方法
class Triangle(Polygon) :
    def __init__( self , p1 , p2 , p3 ) : Polygon.__init__(self,[p1,p2,p3])
    def name( self ) : return "三角形"

if __name__ == "__main__" :
    # 定義四個點與兩個物件
    p1 , p2 , p3 , p4 = Point(0,0) , Point(1,0) , Point(0,2) , Point(-2,2)
    poly , tri = Polygon([p1,p2,p3,p4]) , Triangle(p1,p2,p3)
    for foo in [ poly , tri ] : print(foo.name(),str(foo))
```

  - 程式最後一行列印 Triangle 物件表示字串時，因 Triangle 並無設定，即使用由 Polygon 父類別
  - 繼承來 __str__ 方法。程式輸出：

```
四邊形 (0,0) (1,0) (0,2) (-2,2)
三角形 (0,0) (1,0) (0,2)
```

Object Oriented Programming

- 繼承
  - 以類似的方式，可以從多個父類別來構建一個新的類別，如下所示：

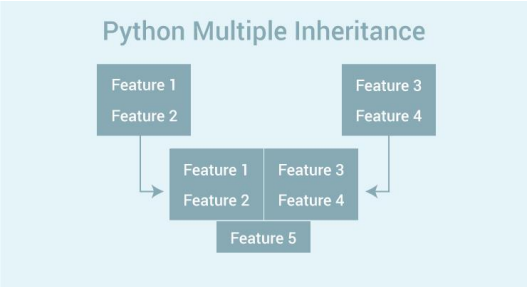
```
class A:        # define your class A
.....

class B:         # define your calss B
.....

class C(A, B):   # subclass of A and B
.....
```

  - 可以使用issubclass()或isinstance()函式來檢查兩個類別和範例之間的關係。
    - issubclass(sub，sup)：
      - 如果給定的子類別(sub)確實是超類別(super)的子類別返回True。
    - isinstance(obj，Class)：
      - 如果obj是類別Class的一個物件或者是類別的一個子類別的物件則返回True。

Object Oriented Programming

- 繼承(**多重繼承**)
  - Python也支援部分的多重繼承形式。一個類別如果要繼承多個基礎類別的話，其形式如下：

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```



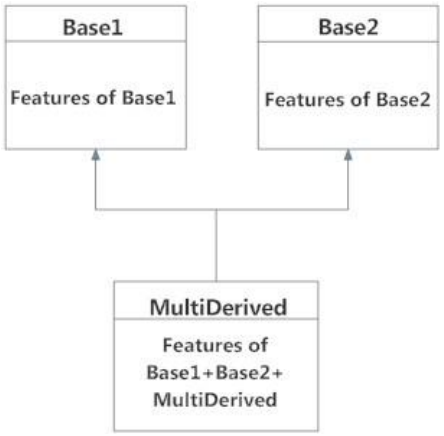
  - 唯一的規則是：當尋找一個attribute的定義時要如何尋找。其規則是先深，而後由左至右(depth-first, left-to-right)。所以當要找一個在子類別 DerivedClassName 裡面的attribute卻找不到時，會先找 Base1 ，然後沿著 Base1 的所有基礎類別尋找，如果找完還沒有找到的話再找 Base2 及其基礎類別，依此類推。
  - Python仰賴程式設計師們的約定俗成的習慣來避免可能的名稱衝突。例如一個眾所週知多重繼承的問題，**如果一個類別繼承了兩個基礎類別，這兩個基礎類別又分別繼承了同一個基礎類別**。也許很容易就了解在這樣的情況下到底會是什麼狀況，(**這個instance將會只有一個單一共用基礎類別的"instance variables"或是data attributes**)。

Object Oriented Programming

- 繼承(多重繼承範例)

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

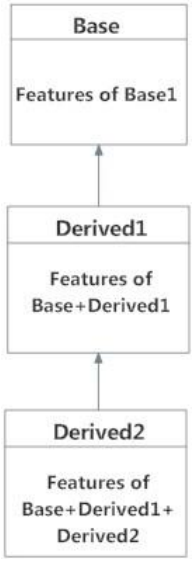
  - MultiDerived衍生自Base1和Base2類別。MultiDerived類別從Base1和Base2繼承。



Object Oriented Programming

- 繼承(**多層繼承**)
  - 另一方面，也可以繼承一個衍生類的形式。這被稱為多層(級)繼承。它可以在Python中有任何的深度(層級)。在多層繼承中，基礎類別和衍生類別的特性被繼承到新的衍生類中。

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```



Derived1衍生自Base，Derived2衍生自Derived1。

Object Oriented Programming

- 繼承(MRO：**方法解析順序**)
  - 每個類別都衍生自類別：object。它是Python中最基礎的型別。
  - 所以在技術上，所有其他類別，無論是內建還是使用者定義，都是衍生類別，所有物件都是物件(object)類別的範例。

```
# Output: True
print(issubclass(list,object))

# Output: True
print(isinstance(5.5,object))

# Output: True
print(isinstance("Hello",object))
```

Object Oriented Programming

- 繼承(MRO：**方法解析順序**)
  - 在多繼承方案中，在當前類別中首先搜尋任何指定的屬性。如果沒有找到，搜尋繼續進入父類別，深度優先，再到左右的方式，而不需要搜尋相同的類別兩次。
  - 所以在MultiDerived類別的例子中，搜尋順序是[MultiDerived，Base1，Base2，object]。此順序也稱為MultiDerived類別的線性化，用於查詢此順序的一組規則稱為方法解析順序(MRO)。
  - MRO必須防止區域優先排序，並提供單調性。它確保一個類別總是出現在其父類別之前，並且在多個父類別的情況下，該順序與基礎類別的數組相同。
  - 一個類別的MRO可以被看作是__mro__屬性或者mro()方法。前者傳回一個數組，而後者傳回一個列表。

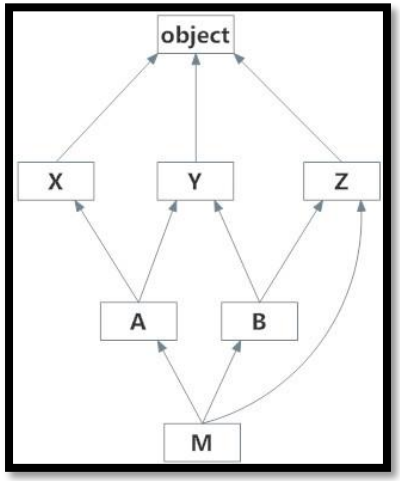
```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass

print(MultiDerived.__mro__)
print(MultiDerived.mro())
```

```
(<class '__main__.MultiDerived'>, <class '__main__.Base1'>, <class '__main__.Base2'>, <class 'object'>)
[<class '__main__.MultiDerived'>, <class '__main__.Base1'>, <class '__main__.Base2'>, <class 'object'>]
```

Object Oriented Programming

- 繼承(MRO：**方法解析順序**)

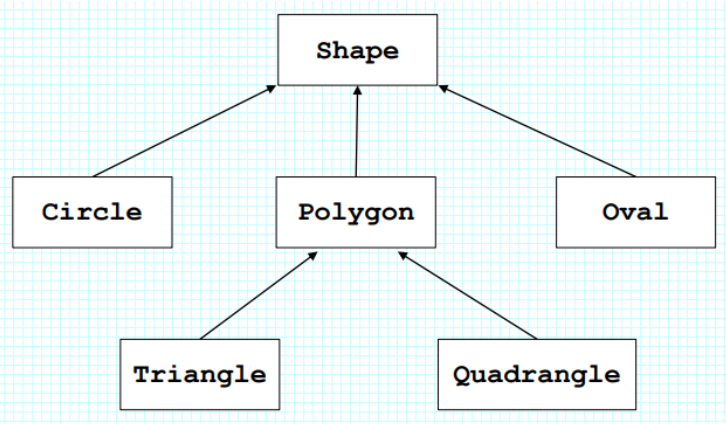
```
class X:
    pass
class Y:
    pass
class Z:
    pass

class A(X,Y):
    pass
class B(Y,Z):
    pass

class M(B,A,Z):
    pass

print(M.mro())
```



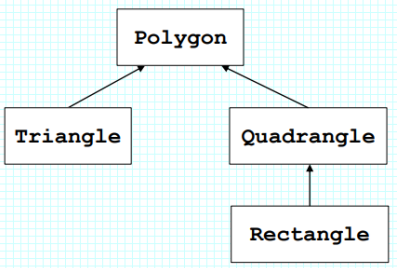

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.X'>, <class '__main__.Y'>, <class '__main__.Z'>, <class 'object'>]
```

Object Oriented Programming

- 繼承(**基礎類別與多個衍生類別**)
  - **基礎類別是所有衍生類別的交集類別**
  - **每個衍生類別物件可視同基礎類別物件**
  - **同層衍生類別間有著個別差異**
  - **同層衍生類別各有著不同的類別屬性與方法**
  - Shape 形狀類別架構：
    - Shape 類別為 Circle、Polygon、Oval 的交集類別
    - Circle、Polygon、Oval 物件可視同 Shape 物件
    - Circle、Polygon、Oval 方法也可執行 Shape 類別的方法
    - Circle、Polygon、Oval 物件也可執行 Shape 類別的方法
    - Polygon 類別為 Triangle 與 Quadrange 的交集類別
    - Triangle 與 Quadrangle 物件可視同 Polygon/Shape 物件
    - Triangle 與 Quadrangle 方法也可執行 Polygon/Shape 類別的方法
    - Triangle 與 Quadrangle 物件也可執行 Polygon/Shape 類別的方法



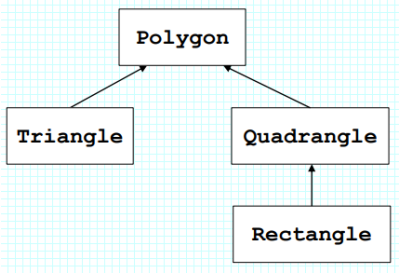
Object Oriented Programming



- 繼承(**衍生類別與其物件**)
  - 每個衍生類別物件可視同基礎類別物件使用
  - 衍生類別物件可自由使用本身與基礎類別的方法/屬性
  - 衍生類別物件使用方法是由物件類別起依繼承順序向上直至基礎類別
  - 衍生類別物件屬性包含所有繼承來各個基礎類別的屬性
  - 衍生類別的**起始設定方法**通常執行直屬基礎類別的起始設定方法
  - 類別架構：多邊形、三角形、四邊形、矩形。

Object Oriented Programming

- 繼承(衍生類別與其物件)




```
cno = "零一二三四五六七八九"

# 基礎類別：多邊形類別
class Polygon :

    def __init__( self , n ) :
        self.npt = n

    def __str__( self ) :
        return cno[self.npt] + "邊形"

# 三角形繼承自多邊形
class Triangle(Polygon) :

    def __init__( self ) :
        Polygon.__init__(self,3) # 執行 Polygon 起始方法

    def __str__( self ) : return "三角形"

# 四邊形繼承自多邊形
class Quadrangle(Polygon) :

    def __init__( self ) :
        Polygon.__init__(self,4) # 執行 Polygon 起始方法

class Rectangle(Quadrangle) :
    def __init__( self ) :
        Quadrangle.__init__(self) # 執行 Quadrangle 起始方法
    def __str__( self ) : return "矩形"

if __name__ == "__main__" :
    # 四個不同圖形
    shapes = [ Polygon(5) , Triangle() , Quadrangle() , Rectangle() ]
    # 輸出：五邊形 三角形 四邊形 矩形 共四列
    for shape in shapes :
        print( shape ) # 等同 print( str(shape) )
```

```
五邊形
三角形
四邊形
矩形
```

由於四邊形並無 __str__，使用繼承來的 Polygon.__str__ 列印 Quadrange 物件
Polygon 類別架構內各類別的邊數都存於 Polygon 類別內

Object Oriented Programming

- 繼承(**執行類別架構的方法的方式**)
  - 類別繼承順序：包含本身類別、父類別、···、最頂層的基礎類別
  - super().method(args)：依類別繼承順序由父類別起找尋方法執行
  - self.method(args)：依類別繼承順序由本類別起找尋方法執行
  - class.method(self, args)：直接執行，class 需在繼承順序內

Object Oriented Programming

- 繼承(**執行類別架構的方法方式**)

```
五邊形：五個邊，內角和 540 度
三角形：三個邊，內角和 180 度，有內心、外心、垂心、重心、旁心
四邊形：四個邊，內角和 360 度
矩形：四個邊，內角和 360 度 ，兩對邊等長，四個角皆為直角
```

```python
cno = "零一二三四五六七八九"

# 繼承順序：Polygon
class Polygon :
    def __init__( self , n ) :
        self.npt = n
    def name( self ) :
        return cno[self.npt] + "邊形"
    def total_angle( self ) :
        return 180*(self.npt-2)
    def property( self ) :
        return "{}個邊，內角和 {} 度".format(cno[self.npt],self.total_angle())
    def __str__( self ) :
        return self.name()

# 繼承順序：Triangle、Polygon
class Triangle(Polygon) :
    def __init__( self ) :
        super().__init__(3)
    def name( self ) :
        return "三角形"
    def property( self ) :
        return ( super().property() + "，有內心、外心、垂心、重心、旁心" )
    def __str__( self ) :
        return self.name()

# 繼承順序：Quadrangle、Polygon
class Quadrangle(Polygon) :
    def __init__( self ) :
        Polygon.__init__(self,4)
    # 可省略
    def __str__( self ) :
        return self.name()

# 繼承順序：Rectangle、Quadrangle、Polygon
class Rectangle(Quadrangle) :
    def __init__( self ) :
        super().__init__()
    def name( self ) :
        return "矩形"
    def property( self ) :
        return ( Polygon.property(self) + " ，兩對邊等長，四個角皆為直角" )
    def __str__( self ) :
        return self.name()
if __name__ == "__main__" :
    shapes = [ Polygon(5) , Triangle() , Quadrangle() , Rectangle() ]
    for shape in shapes :
        print(shape,'：',shape.property(),sep="")
```

Object Oriented Programming

- 覆寫方法
  - 可以隨時覆寫父類別的方法。覆寫父方法的一個原因是：可能希望在子類別中使用特殊或不同的方法功能。

```
class Parent:        # define parent class
   def myMethod(self):
      print ('Calling parent method')

class Child(Parent): # define child class
   def myMethod(self):
      print ('Calling child method')

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

```
Calling child method
```

Object Oriented Programming

- 基本方法覆寫
  - 下表列出了可以在自己的類別中覆蓋的一些**通用**方法：

編號	方法	描述	呼叫範例
1	**__init__ (self [,args...])**	建構函式(帶任意可選引數)	obj = className(args)
2	__del__(self)	解構函式，刪除一個物件	del obj
3	__repr__(self)	可評估求值的字串表示	repr(obj)
4	**__str__(self)**	可列印的字串表示	str(obj)
5	__cmp__ (self, x)	物件比較	cmp(obj, x)

Object Oriented Programming

- 運算子覆寫
  - 假設已經建立了一個Vector類別來表示二維向量。當使用加號(+)運算子執行運算時，它們會發生什麼？ 很可能Python理解不了想要做什麼。
  - 但是，可以在類別中定義__add__方法來執行向量加法，然後將按照期望行為那樣執行加法運算。

```
class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)
```

```
Vector(7,8)
```

Object Oriented Programming

- 運算子覆寫
  - 以下為常用的運算子與其對應名稱
    - 基本運算子：+、-(減)、-(負)、*、‧‧‧等
    - 複合運算子：+=、-=、*=、‧‧‧等

運算子	運算式	函式運算	運算子	運算式	函式運算
+	p1 + p2	p1.__add__(p2)	+=	p1 += p2	p1.__iadd__(p2)
-	p1 - p2	p1.__sub__(p2)	-=	p1 -= p2	p1.__isub__(p2)
*	p1 * p2	p1.__mul__(p2)	*=	p1 *= p2	p1.__imul__(p2)
/	p1 / p2	p1.__truediv__(p2)	/=	p1 /= p2	p1.__itruediv__(p2)
//	p1 // p2	p1.__floordiv__(p2)	//=	p1 //= p2	p1.__ifloordiv__(p2)
**	p1 ** p2	p1.__pow__(p2)	**=	p1 **= p2	p1.__ipow__(p2)
%	p1 % p2	p1.__mod__(p2)	%=	p1 %= p2	p1.__imod__(p2)
-	-p1	p1.__neg__()			
<<	p1 << p2	p1.__lshift__(p2)	<<=	p1 <<= p2	p1.__ilshift__(p2)
>>	p1 >> p2	p1.__rshift__(p2)	>>=	p1 >>= p2	p1.__irshift__(p2)
&	p1 & p2	p1.__and__(p2)	&=	p1 &= p2	p1.__iand__(p2)
|	p1 | p2	p1.__or__(p2)	|=	p1 |= p2	p1.__ior__(p2)
^	p1 ^ p2	p1.__xor__(p2)	^=	p1 ^= p2	p1.__ixor__(p2)
~	~p1	p1.__invert__()			

Object Oriented Programming

- 運算子覆寫
  - 以下為常用的運算子與其對應名稱
    - 比較運算子：<、<=、>、>=、==、!= 等

運算子	運算式	方法運算
<	p1 < p2	p1.__lt__(p2)
<=	p1 <= p2	p1.__le__(p2)
>	p1 > p2	p1.__gt__(p2)
>=	p1 >= p2	p1.__ge__(p2)
==	p1 == p2	p1.__eq__(p2)
!=	p1 != p2	p1.__ne__(p2)

    - 其他運算子：[]、del、in 等

運算子	運算式	方法運算
[]	p1[i]	p1.__getitem__(i)
[]	p1[i]=x	p1.__setitem__(i,x)
del []	del p1[i]	p1.__delitem__(i)
in	x in p1	p1.__contains__(x)

Object Oriented Programming

- 例外(Exceptions)也可以是類別
  - 使用者自訂的exception不用只是被限定於只是字串物件而已，它們現在也可以用類別來定義了。使用這個機制的話，就可以**創造出一個可延伸的exception的階層**了。
  - 有兩個新的有效的(語意上的)形式現在可以用來當作**引發exception**的敘述：
    1. raise Class, instance
    2. raise instance
  - 在第一個形式裡面，instance 必須是 Class 這個類別或其子類別的一個instance。第二種形式其實是底下這種形式的一個簡化：

    raise instance.__class__, instance

Object Oriented Programming

- 例外(Exceptions)也可以是類別
  - 所以現在在except的語句裡面就可以使用字串物件或是類別都可以了。一個在exception子句裡的類別可以接受一個是該類別的exception，或者是該類別之子類別的exception。(相反就不可以了，一個except子句裡如果用的是子類別，就不能接受一個基礎類別的exception) ，例如：下面的程式碼就會依序的印出B, C, D來：

```
class B(BaseException):
    pass

class C(B):
    pass

class D(C):
    pass

for c in [B,C,D]:
    try:
        raise c()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

  - 值得注意的是，如果上面的例子裡的except子句次序都掉轉的話(也就是 "except B" 是第一個)，這樣子印出來的就是B, B, B，也就是只有第一個可以接受的except子句被執行而已。
  - 當一個沒有被處理到的exception是一個類別時，所印出來的錯誤信息會包含其**類別的名稱**，**然後是(：)**，然後是這個instance用**內建的__str()__ 函式轉換成的字串**。

Object Oriented Programming

- 多型(多種形狀)
  - 多型是Python中類別定義的一個重要特性，可以在類別或子類別中使用通用命名方法。這允許功能**在不同時間使用不同型別的實體**。所以，它提供了靈活性和鬆散耦合，以便程式碼可以隨著時間的推移而擴充套件和輕鬆維護。
  - 這允許函式使用任何這些多型類別的物件，而不需要知道跨類別的區別。
  - 多型性可以通過繼承進行，子類別使用基礎類別方法或覆蓋它們。

Object Oriented Programming

- 多型(多種形狀)
  - 使用之前的繼承範例理解多型的概念，並在兩個子類別中新增一個名為show_affection的常用方法：
    - 從這個例子可以看到，它指的是一種設計，其中不同型別的物件可以以相同的方式處理，或者更具體地說，兩個或更多的類別使用相同的名稱或通用介面，因為同樣的方法(下面的範例中的show_affection) 用任何一種型別的物件呼叫。

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

    def eat(self, food):
        print('%s is eating %s , '%(self.name, food))

class Dog(Animal):

    def fetch(self, thing):
        print("{0} wags {1}".format(self.name, thing))

    def show_affection(self):
        print("{0} wags tail ".format(self.name))

class Cat(Animal):

    def swatstring(self):
        print('%s shreds the string! ' % (self.name))

    def show_affection(self):
        print("{0} purrs ".format(self.name))

d = Dog('Ranger')
c = Cat("Meow")

d.show_affection()
c.show_affection()
```

所以，所有的動物都表現出喜愛(show_affection)，都不太相同。「show_affection」行為因此具有多型性，因為它根據動物的不同而採取不同的行為。 因此，抽象的「動物」概念實際上並不是「show_affection」，而是特定的動物(如狗和貓)具有動作 show_affection 的具體實現。

```
Ranger wags tail
Meow purrs
```

Q & A