python 程式設計

第 十 講

類　　別

類別

- 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)：個別物件所擁有的私有資料
  - 程序(method)：定義於類別內的函式
    - 實例程序(instance method)：由類別物件所執行的程序
    - 類別程序(class method)：由類別本身所執行的程序
    - 靜態程序(static method)：可由類別或類別物件所執行的程序

分數類別 (一)

- 分數類別範例：物件建構與其字串表示方式
  - `__init__` 用來建構類別物件
  - `__str__` 用來產生物件的輸出字串型式

```
class Fraction :
    # 起始程序：用以建構物件
    def __init__ ( self , n = 0 , d = 1 ) :
        self.num , self.den = n , d
    #字串表示程序：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)
```

分數類別 (二)

- 以上兩程序皆是 **Fraction** 類別的實例程序
- 實例程序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__** 程序名稱前後皆有雙底線，此為 **python** 預設的程序名稱
- 物件的屬性與其個數可依問題自由設定
- **python** 並沒有如 **C++** 同等的私有成員，但若屬性或程序名稱是以一個底線起始的，在習慣上都被當成私有成員
- 程序名稱不能與屬性同名
- 程序名稱若有重複，則後來讀取的程序會取代先前的
- **一般來說，每當設計新類別時，都要先由這兩個程序開始設計**

分數類別 (三)

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)                 # 預設 分母為 1
c = Fraction(2,3)               # 分子 2 , 分母 3
d = Fraction.__init__(2,3) # 同上, 但直接使用起始程序名稱

# 自動使用 Fraction.__str__() 輸出物件對應的字串表示方式
print( a , b , c , d )
```

輸出:

```
0 4 2/3 2/3
```

❖ **為節省篇幅, 本章之後的分數類別範例都自動包含此兩個程序**

分數的一些實例程序 (一)

- 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)
```

分數的一些實例程序 (二)

- 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
```

  - **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
```

❖ **類別的起始程序不適用第二種型式，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
```

兩個特殊雙底線屬性 (一)

- **`__doc__`**: 代表定義於類別名稱後的字串（可跨行），通常被用來當成類別的說明文字。為類別屬性，使用 **`class.__doc__`**
- **`__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程序

兩個特殊雙底線屬性 (二)

```
class Fraction :
    """分數類別: num : 分子 den : 分母"""
    ...
```

執行:

```
print( Fraction.__doc__ )

a = Fraction(3,5)
print( a.__dict__ )                 # 印出 a 物件所儲存的資料

print( Fraction.__dict__ )          # 印出分數類別字典儲存內容
```

輸出:

```
分數類別: num : 分子   den : 分母
{'den': 5, 'num': 3}
{'__str__': <function Fraction.__str__ at 0x7f75f277cd08>,
 '__init__': <function Fraction.__init__ at 0x7f75f277c9d8>,
 ...(省略) }
```

兩個特殊雙底線屬性 (三)

- 可透過 **obj.__dict__** 字典直接更改物件內部屬性（不建議使用）

```
a = Fraction(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

類別內雙底線起始的名稱 (一)

- 無法為類別物件直接使用
- 可為類別內其他程序使用
- 名稱被自動加上 **`_ClassName`** 於其前

類別內雙底線起始的名稱 (二)

```
class Fraction :
    ...
    # 雙底線開始的程序名稱，不能由物件直接取用
    def __inverse( self ) :
        return Fraction(self.den,self.num)
    # 雙底線開始的程序名稱可由類別其他程序使用
    def inv( self ) :
        return self.__inverse()

a = Fraction(2,3)
# 錯誤，無此程序
print ( a.__inverse () )

# 正確，原雙底線程序名稱前被加上 _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程序與 C++ 設定的「私有成員」仍有差異

類別程序 (一)：類別共享的程序

- 類別程序以 **@classmethod** 獨立的標籤起始
- 類別程序的第一個參數為類別本身，通常以 **cls** 表示
- 類別程序使用 **class.method(...)** 來執行
- 類別程序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

類別程序 (二)

```
class Fraction :
    ...
    # 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"
```

類別程序 (三)

使用方式:

```
# 以下三個 Fraction 被自動設為類別程序的第一個參數
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() )
```

類別屬性與靜態程序

- 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
- 靜態程序：類別或物件共享的程序
  - 靜態程序以 `@staticmethod` 獨立的標籤起始
  - 靜態程序的第一個參數非物件或類別
  - 若程序不需取用物件或類別屬性，但「性質」上歸類為類別，則設計為靜態程序
  - 使用 `class.staticmethod(...)` 或 `obj.staticmethod(...)` 方式執行
  - 在實務上，靜態程序不常使用

類別範例 (一)

- 類別範例：簡單計程車里程計費
  - 類別屬性：計程車的里程計費資料適用所有計程車物件
  - 類別程序：根據駕駛距離計算距離
  - 靜態程序：提供類別計費資料

類別範例 (二)

```
class Taxi :

    # 類別屬性
    idis , udis , ifee , ufee = 1000 , 500 , 20 , 10

    # 實例程序
    def __init__( self , d = 0 ) : self.dis = d

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

    # 實例程序
    def  fee( self ) :
        return Taxi.charge(self.dis)

    # 實例程序
    def  __str__( self ) :
        return "距離: " + str(self.dis) + " m"
```

```
    # 靜態程序
    @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
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" )
```

類別範例 (三)

輸出:

```
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: 3800 m --> 80 NT
距離: 4000 m --> 90 NT
```

類別外部函式

- 函式若與類別/實例屬性或程序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 :
    ...
    # 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_frac() )
```

❖ 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，在歸類上 gcd 函式也可屬於分數， gcd 也可改寫為靜態程序

程序與函式的使用時機

■ 使用時機

名稱	語法	使用方式	使用時機
實例程序	`method(obj,...)`	`obj.method(...)` 或 `class(obj,...)`	需使用實例屬性
類別程序	`@classmethod` `method(cls,...)`	`cls.method(...)`	需使用類別屬性
靜態程序	`@staticmethod` `method(...)`	`obj.method(...)` 或 `class.method(...)`	不需使用到物件與類別屬性，但程序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

❖ 若程序同時需使用到實例屬性與類別屬性，則應設定成實例程序

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

```
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
print( a is b )                # False
```

檔案與模組 (一)

- 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，但使用所有 **foo** 模組定義的名稱前需加上 「**foo.**」
  - **import foo , bar , ...** : 一次併入 **foo.py**、**bar.py**、**...** 等多個檔案
  - **from foo import *** : 併入 **foo.py** 檔，但可直接使用 **foo** 模組定義名稱，不需加上「**foo.**」
  - **from foo import a** : 僅由 **foo.py** 檔案內併入 **a** 名稱於程式中，且不需加上「**foo.**」
  - **from foo import a , b , ...** : 同上，由 **foo.py** 檔案併入 **a , b , ...** ，且不需加上「**foo.**」

檔案與模組 (二)

- 使用 import foo 併入的 foo 模組名稱被存入 foo.__name__
- 範例:

以下 chars.py 檔案用來產生介於兩字元間的連續字元字串:

chars.py 檔案

```
class Chars :
    def __init__( self , c1 , c2 ) :
        n1 , n2 = ord(c1) , ord(c2)
        self.s = "".join([ chr(c) for c in range(n1,n2+1) ])
    def __str__ ( self ) : return self.s

# 印出 c1 到 c2 連續字元
def pchars(c1,c2) : print( Chars(c1,c2) )

# 回傳在 c1 到 c2 字串, 字元間有 sep 分開
def chars_sep(c1,c2,sep) : return sep.join(str(Chars(c1,c2)))
```

c1.py 取得 chars.py 程式檔:

c1.py 檔案

```
import chars

# 印出: abcde
print( chars.Chars('a','e') )

# 印出: abcde
chars.pchars('a','e')

# 印出: a--b--c--d--e
print( chars.chars_sep(c1,c2,'--') )
```

檔案與模組 (三)

c2.py 僅取用 chars.py 的 pchars 與 chars_sep 兩函式。

c2.py 檔案

```
# 僅由 chars.py 併入 pchars 與 chars_sep 兩函式
from chars import pchars , chars_sep

# 印出: abcde
pchars('a','e')

# 印出: a--b--c--d--e
print( chars_sep(c1,c2,'--') )

# 錯誤: 沒有併入 Chars 類別
print( Chars('a','e') )
print( chars.Chars('a','e') )
```

檔案與模組 (四)

- 若在 bar.py 檔內使用 from foo import a ，須留意由 foo.py 併入的名稱 a 可能會與 bar.py 檔內的同名稱重複，此時程式會保留最後出現的名稱設定，通常是 bar.py

c3.py 檔案

```
from chars import pchars
...

# 請留意: 以下函式名稱與 chars.pchars 一樣
def pchars(a,b) : print(a+b)

# 執行 c3.py 新的 pchars
pchars('a','e')          # 輸出: ae
```

__name__模組名稱字串 (一)

- 字串儲存模組名稱
- 檔案若為起始執行檔，則其 **__name__** 儲存 **"__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
```

__name__模組名稱字串 (二)

- 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__

__name__ 模組名稱字串 (三)

- 使用 **__name__** 條件式可分離測試程式區塊
  - 在開發程式時常需測試程式，為避免測試程式區塊與檔案其他程式區塊混雜一起，可用條件式檢查 **__name__** 值是否等於 **"__main__"** 來分離測試程式區塊

foo.py 檔

```
# A 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
    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，跳過 **B** 程式區塊

__name__ 模組名稱字串 (四)

- 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
...
pass

# 定義類似 C 語言的主函式
def main() :
    pass
if __name__ == "__main__" :
    main()
```

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

運算子覆載 (一)

- python 常用的運算子符號都有對應的函式名稱，例如：加號的對應名稱為 __add__。
  在使用上 p1 + p2 被改以 p1.__add__(p2) 方式執行
- 運算子程序都是實例程序，第一個參數都是類別物件
- 使用者可根據需要在類別內重新定義運算子

```
class Fraction :
    ....
    # 加法運算子
    def  __add__( self , frac ) :
        n = self.num * frac.den + self.den * frac.num
        d = self.den * frac.den
        return Fraction(n,d)

a , b = Fraction(2,3) , Fraction(5,2)

# 分數加法
c = a + b     # 也可寫成 c = a.__add__(b)

print( c )    # 印出 19/6
```

運算子覆載 (二)

- 以下為常用的運算子與其對應名稱
  - 基本運算子：+、-、*、．．．等
  - 複合運算子：+=、-=、*=、．．．等
  - 比較運算子：<、<=、>、>=、==、!= 等
  - 其他運算子：[]、del、in 等

運算子覆載 (三)

■ 基本運算子與複合運算子

運算子	運算式	函式運算
+	p1 + p2	p1.__add__(p2)
-	p1 - p2	p1.__sub__(p2)
*	p1 * p2	p1.__mul__(p2)
/	p1 / p2	p1.__truediv__(p2)
//	p1 // p2	p1.__floordiv__(p2)
**	p1 ** p2	p1.__pow__(p2)
%	p1 % p2	p1.__mod__(p2)
<<	p1 << p2	p1.__lshift__(p2)
>>	p1 >> p2	p1.__rshift__(p2)
&	p1 & p2	p1.__and__(p2)
|	p1 | p2	p1.__or__(p2)
^	p1 ^ p2	p1.__xor__(p2)
~	~p1	p1.__invert__()

運算子	運算式	函式運算
+=	p1 += p2	p1.__iadd__(p2)
-=	p1 -= p2	p1.__isub__(p2)
*=	p1 *= p2	p1.__imul__(p2)
/=	p1 /= p2	p1.__itruediv__(p2)
//=	p1 //= p2	p1.__ifloordiv__(p2)
**=	p1 **= p2	p1.__ipow__(p2)
%=	p1 %= p2	p1.__imod__(p2)
<<=	p1 <<= p2	p1.__ilshift__(p2)
>>=	p1 >>= p2	p1.__irshift__(p2)
&=	p1 &= p2	p1.__iand__(p2)
|=	p1 |= p2	p1.__ior__(p2)
^=	p1 ^= p2	p1.__ixor__(p2)

運算子覆載 (四)

- 比較運算子

運算子	運算式	程序運算
<	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)

- 下標運算子與包含

運算子	運算式	程序運算
[]	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)

運算子覆載 (五)

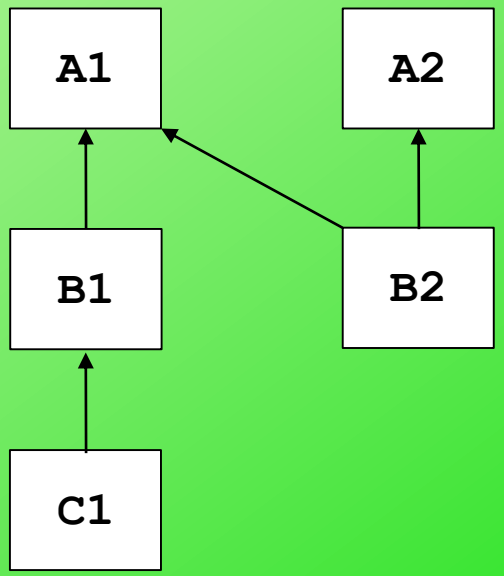
以下為一些分數的運算子覆載:

```
class Fraction :
    ...
    # p1*p2
    def __mul__ ( self , frac ) :
        d = self.den * frac.den
        n = self.num * frac.num
        return Fraction(n,d)

    # p1 += p2 利用加法運算子程序
    def __iadd__ ( self , frac ) :
        self = self + frac
        return self

    # 檢查 p1 < p2
    def __lt__ ( self , frac ) :
        if self.num*frac.den < self.den*frac.num :
           return True
        else :
           return False

    # 檢查 p1 == p2
    def __eq__ ( self , frac ) :
        if self.num*frac.den == self.den*frac.num :
            return True
        else :
            return False
```

類別繼承

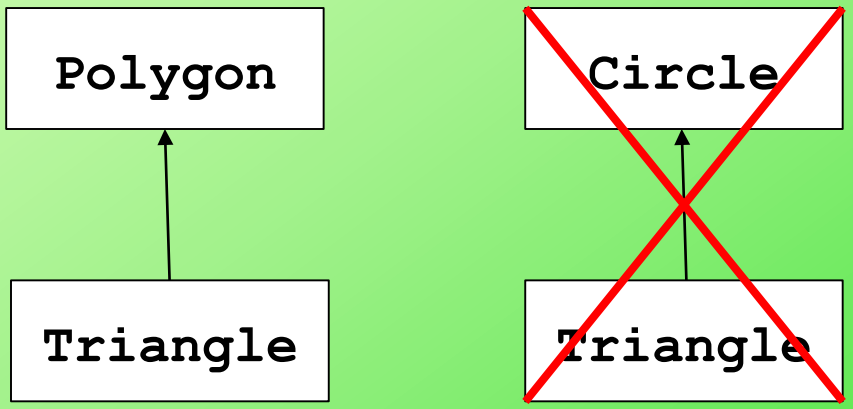
- 基礎類別(Base class)為被繼承類別或稱為超類別(superclass)、父類別(parent class)
- 衍生類別(Derived class)為繼承類別或稱為次類別(subclass)、子類別(child class)
- 類別架構: 透過類別繼承組合的類別關係

```
class A1 : pass
class A2 : pass

# 單一繼承
class B1(A1) : pass

# 多重繼承:  B2 --> A1 , A2
class B2(A1,A2) : pass

# 單一繼承:  C1 --> B1 ---> A1
class C1(B1) : pass
```



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
- 衍生類別程序也可使用衍生/基礎類別內的所有程序
- 衍生類別物件也可使用衍生/基礎類別內的所有程序



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

類別關係：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程序
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 繼承來 __str__ 程序。程式輸出：

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

基礎類別與多個衍生類別 (一)

- 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

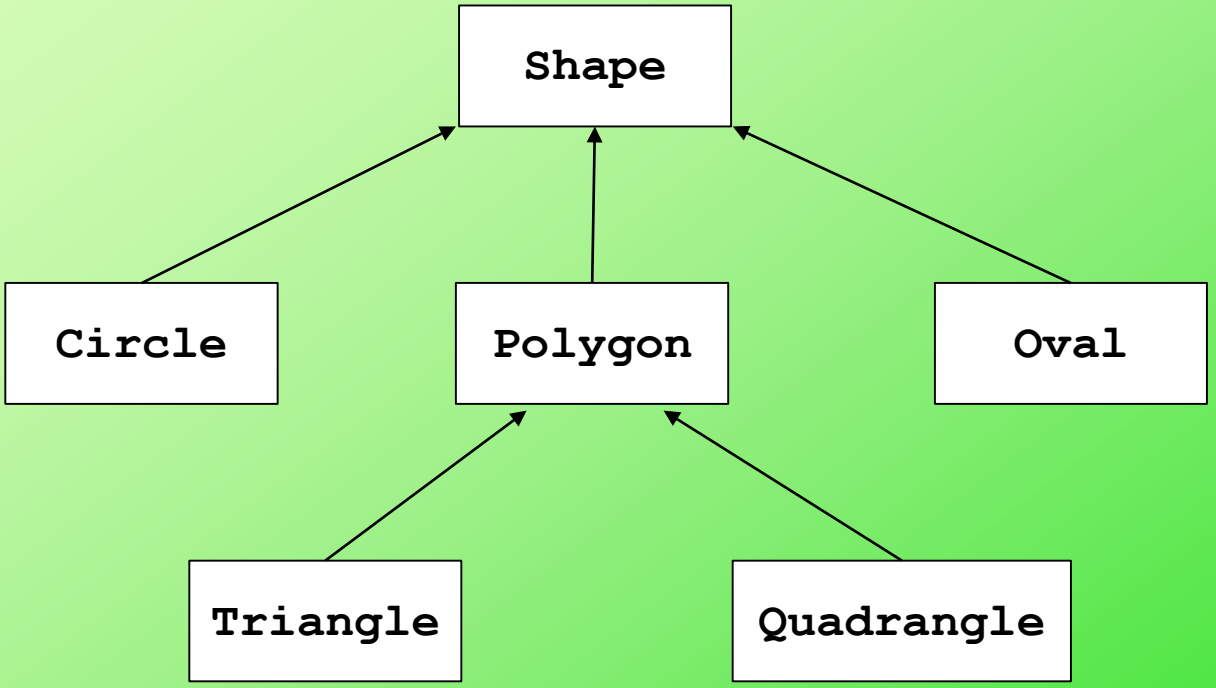


圖 1.4: Shape 類別架構

基礎類別與多個衍生類別 (一)

- 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程序
- **Shape 形狀類別架構: 參考圖 1.4**
  - **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 程序也可執行 Shape 類別的程序**
  - **Circle、Polygon、Oval 物件也可執行 Shape 類別的程序**
  - **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 程序也可執行 Polygon/Shape 類別的程序**
  - **Triangle 與 Quadrangle 物件也可執行 Polygon/Shape 類別的程序**

衍生類別與其物件 (一)

- 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程序/屬性
- 衍生類別物件使用程序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程序通常執行直屬基礎類別的起始設定程序
- 類別架構：多邊形、三角形、四邊形、矩形。參考圖 1.5

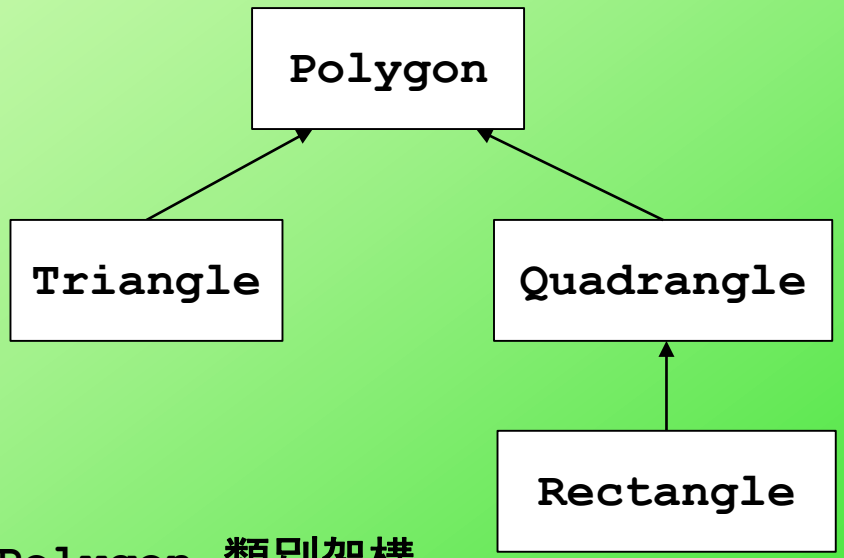


圖 1.5: Polygon 類別架構

衍生類別與其物件 (二)

```
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 起始程序

    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 起始程序
```

衍生類別與其物件 (三)

```
# 矩形繼承自四邊形
class Rectangle(Quadrangle) :
    def __init__( self ) :
        Quadrangle.__init__(self)      # 執行 Quadrangle 起始程序

    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) )
```

- 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

執行類別架構的程序方式 (一)

- 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程序執行
- **self.method(args)**：依類別繼承順序由本類別起找尋程序執行
- **class.method(self,args)**：直接執行 **class.method(self,args)**，**class** 需在繼承順序內
- 類別架構：多邊形、三角形、四邊形、矩形。參考圖 **1.5**

執行類別架構的程序方式 (二)

```
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()
```

執行類別架構的程序方式 (三)

```
# 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="")
```

輸出:

```
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
```

計算近似顏色 (一)

有一 RGB 顏色檔如下:

```
LightBlue2 178 223 238
Yellow2 238 238 0
LightSalmon2 238 149 114
LightSalmon4 139 87 66
...
```

讀入 RGB 顏色檔，輸入某顏色，找出前十個靠近顏色，這裡所謂「靠近」是指以人眼辨視為基準的相近顏色。如果紅(red)、綠(gree)、藍(blue)都介於 [0,255] 之間，以下是用來計算兩顏色的距離公式 ΔC:

$$\Delta C = \sqrt{2\,\Delta r^2 + 4\,\Delta g^2 + 3\,\Delta b^2 + \bar{r}\,\frac{\Delta r^2 - \Delta b^2}{256}}$$

以上的 Δr、Δg、Δb 分別代表兩個紅、綠、藍的差距， r 則為兩個紅色的平均值。以黑白兩色為例，黑色 RGB 三色皆為 0 、白色則為 255 ，代入公式後兩色距離為 765 。

計算近似顏色 (二)

在程式中，每讀入一個顏色後，隨即產生一個顏色物件存入顏色字典內，以方便之後的比較，以下為部份輸出的結果。輸出的第二行，代表兩色距離，之後則為 **RGB** 顏色與名稱。

```
> Yellow
-->      ffff00 Yellow
 1: 44.9 eeee00 Yellow2
 2: 80.0 ffd700 Gold
 3: 111.9 eec900 Gold2
 4: 131.3 cdcd00 Yellow3
 5: 134.6 ffc125 Goldenrod
 6: 154.4 adff2f GreenYellow
 7: 157.6 b3ee3a OliveDrab2
 8: 160.3 eeb422 Goldenrod2
 9: 167.8 eead0e DarkGoldenrod2
10: 180.0 ffa500 Orange
```

```
> Green
--> 00ff00 Green
 1: 34.0 00ee00 Green2
 2: 100.0 00cd00 Green3
 3: 150.0 32cd32 LimeGreen
 4: 179.5 76ee00 Chartreuse2
 5: 181.3 66cd00 Chartreuse3
 6: 185.8 7cfc00 LawnGreen
 7: 190.4 7fff00 Chartreuse
 8: 203.0 00cd66 SpringGreen3
 9: 207.2 00ee76 SpringGreen2
10: 220.0 00ff7f SpringGreen
```

請留意，本題特別使用類別方式設計程式，但也可使用字典儲存顏色資料即能完成程式設計。

計算近似顏色 (三)

```
import math

class Color :
    "顏色類別儲存: 顏色名稱與 R G B 數值"

    nrgb = 3

    def __init__( self , cname , r , g , b ) :
        self.cname = cname
        self.rgb = [ int(x) for x in [ r , g , b ] ]

    # 輸出 16 進位表示的顏色
    def __str__( self ) :
        return ("{:0>2x}"*Color.nrgb).format( *(self.rgb) ) + " " + self.cname

    def name( self ) : return self.cname
    def red( self ) : return self.rgb[0]
    def green( self ) : return self.rgb[1]
    def blue( self ) : return self.rgb[2]

    #兩顏色的距離
    def distance_from( self , color ) :
        dr , dg , db = [ self.rgb[i] - color.rgb[i] for i in range(Color.nrgb) ]
        ravg = ( self.red() + color.red() ) / 2
        return math.sqrt( 2*dr**2 + 4*dg**2 + 3*db**2 + ravg*(dr**2-db**2)/256 )
```

計算近似顏色 (四)

```
if __name__ == "__main__" :
    colors = {}
    with open("rgb.dat") as infile :
        for line in infile :
            n , *rgb = line.split()
            colors[n] = Color(n,*rgb)

    while True :
        color_name = input("> ")
        if color_name in colors :
            color = colors[color_name]
            print( "-->" + " "*6 , color )
            # 計算與其他顏色的距離組合
            cpair = [ ( colors[x] , color.distance_from(colors[x]) )
                      for x in colors if x != color_name ]
            # 印出十個最相近的顏色
            i = 1
            for c , d in sorted( cpair , key=lambda p : ( p[1] , p[0].name() ) ) :
                print("{:>2}: {:>5.1f} {:}".format(i,d,c))
                if i == 10 : break
                i += 1
            print()
```

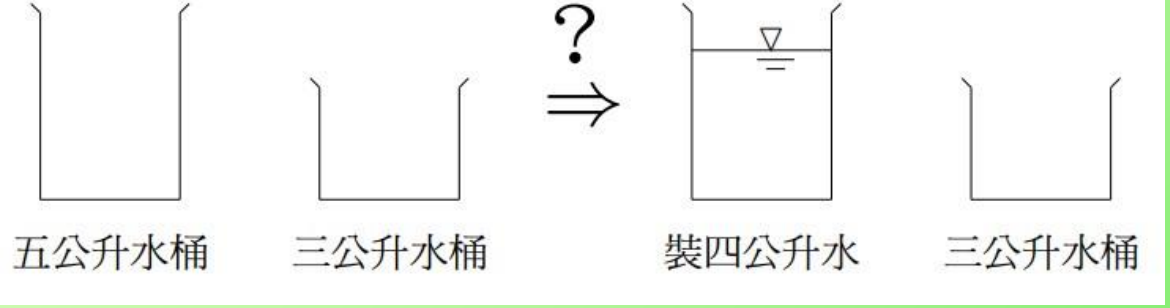
水桶倒水模擬 (一)

大小兩個水桶，裝滿容量分別是 5 公升與 3 公升，水桶沒有刻度。現要設計程式模擬水桶裝水、倒水使得經過一些裝倒水後，大水桶可得 4 公升的水。

由於水桶沒有水位刻度，每次裝水都會裝到水桶滿。當將 a 水桶倒向 b 水桶時，最多到 b 水桶裝滿為止。程式裡利用覆載 >> 運算子執行 a >> b 模擬 a 水桶倒向 b 水桶的動作。倒水動作會同時改變 a 與 b 兩水桶的水位，但 b 水桶是以 immutable 方式傳入 >> 運算子程序，程序內的水位修正並不會影響原傳入的 b 水桶參數。

為了一併更改 a 與 b 兩水桶的水位資料，遂以 return a , b 方式同時回傳修正後的物件。

水桶倒水模擬 (二)



此程式執行後輸出：

```
a , b 兩水桶容量分別為 5 公升與 3 公升 :
   a: 水桶高: 5 ,  水位高: 0   b: 水桶高: 3 ,  水位高: 0

> a 先裝滿水後倒向 b :
   a: 水桶高: 5 ,  水位高: 2   b: 水桶高: 3 ,  水位高: 3

> b 倒光後, a 再倒水到 b :
   a: 水桶高: 5 ,  水位高: 0   b: 水桶高: 3 ,  水位高: 2

> a 先裝滿水後倒向 b :
   a: 水桶高: 5 ,  水位高: 4   b: 水桶高: 3 ,  水位高: 3
```

水桶倒水模擬 (三)

```
class Bucket :

    # 設定初始水桶與水位高
    def __init__( self , h , w = 0 ) :
        self.bucket_height , self.water_height = h , w

    def __str__( self ) :
        return ( "水桶高: " + str(self.bucket_height) + " ,  " +
                 "水位高: " + str(self.water_height) )

    # 裝滿水
    def fill_water( self ) :
        self.water_height = self.bucket_height

    # 水倒光
    def empty_bucket( self ) :
        self.water_height = 0
```

```
    # 將水倒向 foo
    def pour_to( self , foo ) :

        remain = foo.bucket_height - foo.water_height

        if self.water_height >= remain :
            foo.water_height = foo.bucket_height
            self.water_height -= remain
        else :
            foo.water_height += self.water_height
            self.water_height = 0

        return self , foo

    # 使用 A >> B 模擬 A 水桶倒向 B 水桶動作
    def __rshift__( self , foo ) :
        return self.pour_to(foo)
```

水桶倒水模擬 (四)

```
if __name__ == '__main__' :

    a , b = Bucket(5,0) , Bucket(3,0)

    print( "> a , b 兩水桶容量分別為 5 公升與 3 公升 : " )
    print( " a: " + str(a) , " b: " + str(b) )
    print()

    # a 先裝滿水後倒向 b
    a.fill_water()
    a , b = a >> b
    print( "> a 先裝滿水後倒向 b : " )
    print( "  a: " + str(a) ,"  b: " + str(b) )
    print()

    # b 倒光後, a 再倒水到 b
    b.empty_bucket()
    a , b = a >> b
    print( "> b 倒光後, a 再倒水到 b : " )
    print( "  a: " + str(a) ,"  b: " + str(b) )
    print()

    # a 先裝滿水後倒向 b
    a.fill_water()
    a , b = a >> b
    print( "> a 先裝滿水後倒向 b : " )
    print( "  a: " + str(a) ,"  b: " + str(b) )
```

分數類別的運算子覆載 (一)

- 設定分數類別與覆載相關運算子，使得兩分數物件可利用 +、-、*、 / 運算子執行加減乘除四則運算。

  同時也可以執行 +=、-=、*=、/= 與六個分數間的比較運算。

分數程式執行後輸出：

```
num:分子 , den:分母
f1 = 2/6
f2 = 2/4
f3 = 5/3

2/6 + 2/4 = 5/6
2/6 - 2/4 = -1/6
2/6 * 2/4 = 1/6
2/6 / 2/4 = 2/3

f3 += f1 ---> f3 = 2

f1 < f2
f1 != f2
```

分數類別的運算子覆載 (二)

```
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 :

    """分數類別: num : 分子  den : 分母"""

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

    # 使用字串設定分數
    @classmethod
    def  fromstr( cls , fstr ) :
        if fstr.isdigit() :
            n , d = int(fstr) , 1
        else :
            n , d = map( int , fstr.split('/') )

        return cls(n,d)
```

分數類別的運算子覆載 (三)

```
# 使用帶分數設定分數
@classmethod
def  mixed_frac( cls , a = 0 , num = 0 , den = 1 ) :
    n , d = a * den + num , den
    return cls(n,d)

# 回傳最簡分數
def simplest_frac( self ) :
    g = gcd(self.num,self.den)
    return Fraction(self.num//g,self.den//g)

# 重新設定分數
def set_val( self , n = 0 , d = 1 ) :
    self.num , self.den = n , d

# 取得分子與分母
def get_num( self ) : return self.num
def get_den( self ) : return self.den
```

分數類別的運算子覆載 (四)

```
# 分數加法
def __add__( self , frac ) :
    n = self.num * frac.den + self.den * frac.num
    d = self.den * frac.den
    return Fraction(n,d).simplest_frac()

# 分數減法
def __sub__( self , frac ) :
    n = self.num * frac.den - self.den * frac.num
    d = self.den * frac.den
    return Fraction(n,d).simplest_frac()

# 分數乘法
def __mul__( self , frac ) :
    n = self.num * frac.num
    d = self.den * frac.den
    return Fraction(n,d).simplest_frac()

# 分數除法
def __truediv__( self , frac ) :
    n = self.num * frac.den
    d = self.den * frac.num
    return Fraction(n,d).simplest_frac()
```

分數類別的運算子覆載 (五)

```
# 分數 +=
def __iadd__( self , frac ) :
    self = self + frac
    return self

# 分數 -=
def __isub__( self , frac ) :
    self = self - frac
    return self

# 分數 *=
def __imul__( self , frac ) :
    self = self * frac
    return self

# 分數 /=
def __itruediv__( self , frac ) :
    self = self / frac
    return self
```

分數類別的運算子覆載 (六)

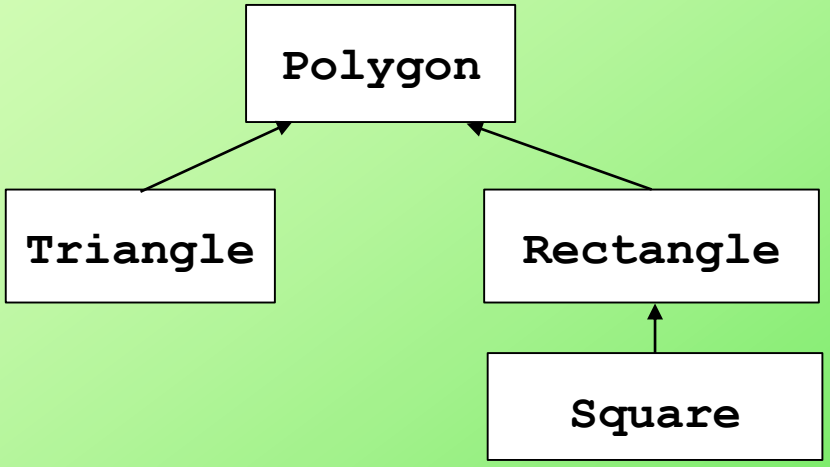
```
# <
def __lt__( self , frac ) :
    return True if self.num*frac.den < self.den*frac.num else False

# ==
def __eq__( self , frac ) :
    return True if self.num*frac.den == self.den*frac.num else False

# >= != <= >
def __ge__( self , frac ) : return not ( self < frac )
def __ne__( self , frac ) : return not ( self == frac )
def __le__( self , frac ) : return ( self < frac or self == frac )
def __gt__( self , frac ) : return not ( self <= frac )

# 分數
@classmethod
def data_doc( cls ) : return "num:分子 , den:分母"

# 輸出分數字串表示分式
def  __str__ ( self ) :
    if self.den == 1 :
        return str(self.num)
    else :
        return "/".join( [ str(self.num) , str(self.den) ] )
```

分數類別的運算子覆載 (七)

```
if __name__ == "__main__" :

    f1 = Fraction(2,6)
    f2 = Fraction.fromstr("2/4")
    f3 = Fraction.mixed_frac(1,2,3)

    print( Fraction.data_doc() )

    print( "f1 =" , f1 )
    print( "f2 =" , f2 )
    print( "f3 =" , f3 )
    print()

    print( f1 , "+" , f2 , '=' , f1+f2 )
    print( f1 , "-" , f2 , '=' , f1-f2 )
    print( f1 , "*" , f2 , '=' , f1*f2 )
    print( f1 , "/" , f2 , '=' , f1/f2 )
    print()

    f3 += f1
    print( "f3 += f1 ---> f3 =" , f3 )

    print()
    print( "f1 < f2" if f1 < f2 else "f1 >= f2" )
    print( "f1 == f2" if f1 == f2 else "f1 != f2" )
```

平面點的多邊形類別架構 (一)

今有由平面點構成的 **Polygon** 多邊形類別架構如下:



多邊形類別架構下的所有點都被儲存在 **Polygon** 基礎類別，**Rectangle** 長方形另存兩邊長，**Square** 正方形則儲存一邊長。每個類別物件都可旋轉，旋轉後回傳各自的類別物件，並沒有統一回傳類別。在類別架構的設計原則下，多邊形基礎類別儲存的屬性或程序都是整個類別架構的所有類別物件共用的。基礎類別是整個類別架構下所有類別的共享類別，各個衍生類別所儲存的屬性或建立的程序則是衍生類別間的個別差異。

平面點的多邊形類別架構 (二)

程式執行後輸出：

```
1 四邊形(0.0,0.0)->(2.0,0.0)->(2.0,3.0)->(0.0,1.0)  面積：4.0
2 三角形(0.0,0.0)->(2.0,0.0)->(2.0,3.0) 內接圓半徑：0.70  面積：3.0
3 長方形(0.0,0.0)->(0.0,2.0)->(-3.0,2.0)->(-3.0,0.0) 兩邊長：2.0 3.0 面積：6.0
4 正方形(2.0,3.0)->(7.0,3.0)->(7.0,8.0)->(2.0,8.0) 邊長：5.0  面積：25.0

> 旋轉 90 度：
1 四邊形(0.0,0.0)->(0.0,2.0)->(-3.0,2.0)->(-1.0,0.0)
2 三角形(0.0,0.0)->(0.0,2.0)->(-3.0,2.0) 內接圓半徑：0.70
3 長方形(0.0,0.0)->(-2.0,0.0)->(-2.0,-3.0)->(-0.0,-3.0) 兩邊長：2.0 3.0
4 正方形(-3.0,2.0)->(-3.0,7.0)->(-8.0,7.0)->(-8.0,2.0) 邊長：5.0
```

平面點的多邊形類別架構 (三)

```
import math

cno = "零一二三四五六七八九十"

# 平面向量點
class Point :

    def __init__( self , x = 0 , y = 0 ) : self.x , self.y = x , y

    def __str__( self ) :
        return "({:>3.1f},{:>3.1f})".format(self.x,self.y)

    def get_x( self ) : return self.x
    def get_y( self ) : return self.y

    def __add__( self , pt ) :
        return Point(self.x+pt.x,self.y+pt.y)

    def __sub__( self , pt ) :
        return Point(self.x-pt.x,self.y-pt.y)

    # 點到原點長度
    def len( self ) : return math.sqrt(self.x**2+self.y**2)

    # 旋轉點, 輸入角度
    def rotate( self , ang ) :
        rang = ang*math.pi/180
        x = self.x * math.cos(rang) - self.y * math.sin(rang)
        y = self.x * math.sin(rang) + self.y * math.cos(rang)
        return Point(x,y)
```

平面點的多邊形類別架構 (四)

```
# 多邊形基礎類別
class Polygon :
    def __init__( self , pts ) : self.pts , self.n = pts , len(pts)
    def name( self ) : return cno[self.n] + "邊形"
    def __str__( self ) : return self.name() + self.pts_str()

    # 座標點連接
    def pts_str( self ) :
        return "->".join( [ str(pt) for pt in self.pts ] )

    # 旋轉座標點
    def rotate_pts( self , ang ) :
        return [ pt.rotate(ang) for pt in self.pts ]

    # 多邊形面積
    def area( self ) :
        a , n = 0 , self.n
        for i in range(n) :
            a += ( self.pts[i].get_x() * self.pts[(i+1)%n].get_y() -
                   self.pts[i].get_y() * self.pts[(i+1)%n].get_x() )
        return abs(a)/2

     # 周長
     def perimeter( self ) :
         s = 0
         for i in range(self.n) :
             s += (self.pts[(i+1)%self.n]-self.pts[i]).len()
         return s

     # 旋轉
     def rotate( self , ang ) : return Polygon(self.rotate_pts(ang))
```

平面點的多邊形類別架構 (五)

```
# 三角形繼承多邊形
class Triangle(Polygon) :

    # 三點座標
    def __init__( self , pts ) : super().__init__(pts)

    @classmethod
    def from_pts( cls , pt1 , pt2 , pt3 ) : return cls([pt1,pt2,pt3])

    def name( self ) : return "三角形"

    def __str__( self ) :
        return ( self.name() + self.pts_str() +
                 " 內接圓半徑: " + "{:<5.2f}".format(self.icircle_rad() ) )

    # 內接圓半徑
    def icircle_rad( self ) : return 2*super().area()/super().perimeter()

    # 旋轉
    def rotate( self , ang ) : return Triangle(super().rotate_pts(ang))
```

平面點的多邊形類別架構 (六)

```
# 多邊形繼承多邊形
class Rectangle(Polygon) :

    # 三個相鄰點得矩形座標
    def __init__( self , pt1 , pt2 , pt4 ) :
        pt3 = pt2 + ( pt4 - pt1 )
        self.len1 , self.len2 = (pt2-pt1).len() , (pt4-pt1).len()
        super().__init__([pt1,pt2,pt3,pt4])

    @classmethod
    def from_pt_len( cls , pt1 , len1 , len2 , ang=0 ) :
        pt2 = pt1 + Point(len1,0).rotate(ang)
        pt4 = pt1 + Point(0,len2).rotate(ang)
        return cls(pt1,pt2,pt4)

    def name( self ) : return "長方形"

    def __str__( self ) :
        return ( self.name() + self.pts_str() +
                 " 兩邊長: " + "{:} {:}".format(self.len1,self.len2) )

    def rotate( self , ang ) :
        pts = super().rotate_pts(ang)
        return Rectangle(pts[0],pts[1],pts[3])
```

平面點的多邊形類別架構 (七)

```
# 正方形繼承長方形
class Square(Rectangle) :

    # 兩相鄰點得方形座標
    def __init__( self , pt1 , pt2 ) :
        pt4 = pt1 + (pt2-pt1).rotate(90)
        self.len = (pt2-pt1).len()
        super().__init__(pt1,pt2,pt4)

    @classmethod
    def from_pt_len( cls , pt1 , len , ang=0 ) :
        pt2 = pt1 + Point(len,0).rotate(ang)
        return cls(pt1,pt2)

    def name( self ) : return "正方形"

    def __str__( self ) :
        return ( self.name() + super().pts_str() +
                 " 邊長: " + "{:}".format(self.len) )

    def rotate( self , ang ) :
        pts = super().rotate_pts(ang)
        return Square(pts[0],pts[1])
```

平面點的多邊形類別架構 (八)

```
if __name__ == "__main__" :

    pt1 , pt2 , pt3 , pt4 = Point() , Point(2,0) , Point(2,3) , Point(0,1)

    gs = [ Polygon([pt1,pt2,pt3,pt4]) ,
           Triangle.from_pts(pt1,pt2,pt3) ,
           Rectangle.from_pt_len( pt1 , 2 , 3 , 90 ) ,
           Square.from_pt_len( pt3 , 5 ) ]

    for i , g in enumerate(gs) :
        print(i+1,g," 面積: "+str(g.area()))

    print("\n> 旋轉 90 度: ")
    for i , g in enumerate(gs) : print(i+1,g.rotate(90))
```

函式求根類別架構 (一)

- 在微積分中有幾個簡單的數值方法可用來求得函式的根，例如：二分逼近法、Secant 法與牛頓迭代法，本例題將透過求根類別架構來設計這三種求根子類別。在進入實際程式設計階段前，以下先簡單介紹三種求根方法：
  - 二分逼近法(bisection method)是微積分中間值定理的直接應用，假若函數的根 r ∈ (a,b)，滿足 f(a)f(b) < 0，讓 c 為 a 與 b 的中點，則可檢查 f(a)f(c) < 0 或是 f(c)f(b) < 0 來決定根是在 (a,c) 之間或是在 (c,b) 之間，如此就縮小一半區間範圍，重複此步驟直到 f(c) 逼近 0。
  - Secant 法是利用靠近根的兩個點 a 與 b，求得穿越 (a,f(a)) 與 (b,f(b)) 的直線 L，再計算直線 L 與 X 軸的交點 c 來估算根 r。若 f(c) 不接近 0，則讓新的 a、b 兩點為舊的 b、c 兩點，重複迭代直到 f(c) 逼近 0。以下為迭代公式：

$$c = b - f(b)\frac{b - a}{f(b) - f(a)}$$

函式求根類別架構 (二)

- Newton 法利用根的近似點 x_1 來計算 $(x_1, f(x_1))$ 切線與 x 軸的交點 x_2 為近似根。若 $f(x_2)$ 不接近 0，讓新的 x_1 為 x_2 ，重複迭代直到 $f(x_2)$ 逼近 0，Newton 法迭代公式如下：

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

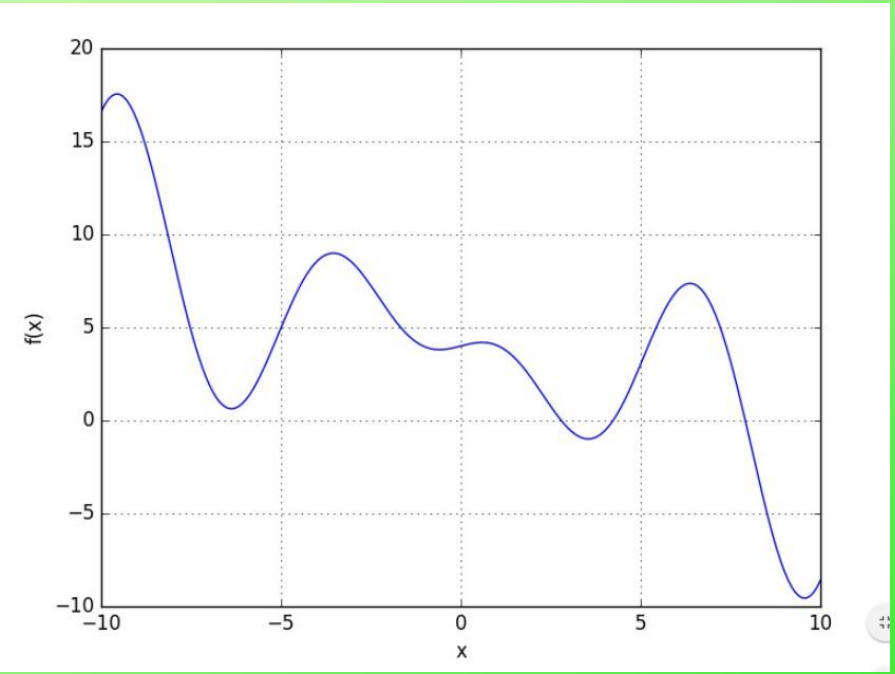
以上方法求得的近似根 c 或 x_2 都要測試函數值的絕對值是否小於預設的誤差值。這三個求根方法以 Newton 法收斂最快，如果起始點 x_1 在根附近，則 Newton 法通常很快就會逼近根，而簡單的二分逼近法則是一種最慢的求根方法。

在程式設計上，三種方法都是求根法，可設計 Root_Finding 為其共同繼承的基礎類別，三種方法為其衍生類別。基礎類別是三種子類別的交集，三個方法的共同資料屬性可存於基礎類別，例如：預設的函數誤差值上限與收斂迭代次數。衍生類別間的個別差異即是各自求根演算法，定義於各類別之中。

函式求根類別架構 (三)

數值求根方法通常要由根的近似值起始運算才能收斂，根的近似值可透過函數圖形大約得知。在程式中，我們也設計一簡單函式 **plot_fn** 來畫出函式的圖形。函數是定義於 **fn** 函式，牛頓法所用的微分函數則是使用簡單的中間值微分公式。

下圖為程式範例所求解 **f(x) = xcos(x) - 5sin($\frac{x}{10}$) + 4 的函數圖形**



函式求根類別架構 (四)

以下為使用三種求根方式的運算結果，其中二分逼近法的起始區間為 [2,3]； Secant 法的兩個起點分別是 x=2 與 x=3 ；牛頓法的起始點則為 x=2。

```
Bisection method :[32] 2.7915574 6.465e-11
Secant method    :[ 5] 2.7915574 7.654e-12
Newton method    :[ 4] 2.7915574 0.000e+00
```

輸出的每一列包含求根法的名稱、收斂時迭代的次數、近似根、近似根的函數值。由結果可看出 Newton 法的迭代次數最少，近似根也最精確。

函式求根類別架構 (五)

```
from math import *
import pylab

# 函式
def fn( x ) :
    return x * cos(x) - 5*sin(x/10) + 4

# 計算微分
def df( x , h=1.e-7 ) :
    return ( fn(x+h) - fn(x-h) ) / (2*h)

# 在 [a,b] 間畫圖, npts 為點數
def plot_fn( a , b , npts = 100 ) :

    dx = ( b - a ) / (npts-1)
    xs = [ a + i * dx for i in range(npts) ]
    ys = [ fn(x) for x in xs ]

    pylab.figure(facecolor='white')
    pylab.grid()
    pylab.xlabel("x")
    pylab.ylabel("f(x)")
    pylab.plot( xs , ys )
    pylab.show()
```

函式求根類別架構 (六)

```
# 定義求根基礎類別
class Root_Finding :

    def __init__( self , err = 1.e-10 ) :
        self.n , self.err = 0 , err

    # 設定收斂誤差
    def set_err( self , err ) : self.err = err

    # 迭代次數
    def iter_no( self ) : return self.n
```

函式求根類別架構 (七)

```
# 二分逼近法
class Bisection(Root_Finding) :

    def __init__( self , err = 1.e-10 ) :
        super().__init__(err)

    def __str__( self ) : return "Bisection method"

    # 二分求根法: 在 [a,b] 區間內求根
    def find_root( self , a , b ) :

        fa , fb , n = fn(a) , fn(b) , 1

        while True :
            c = ( a + b ) / 2
            fc = fn(c)

            if abs(fc) < self.err : break

            if fa * fc < 0 :
                b , fb = c , fc
            elif fb * fc < 0 :
                a , fa = c , fc

        n += 1

        self.n = n
        return c
```

函式求根類別架構 (八)

```
# Secant 法
class Secant(Root_Finding) :

    def __init__( self , err = 1.e-10 ) :
        super().__init__(err)

    def __str__( self ) : return "Secant method"

    # Secant 求根法: 由 a , b 兩點起始
    def find_root( self , a , b ) :

        fa , fb , n = fn(a) , fn(b) , 1

        while True :
            c = b - fb * (b-a) / (fb-fa)
            fc = fn(c)
            if abs(fc) < self.err : break

            a , b = b , c
            fa , fb = fb , fc
            n += 1

        self.n = n
        return c
```

函式求根類別架構 (九)

```
# 牛頓迭代法
class Newton(Root_Finding) :

    def __init__( self , err = 1.e-10 ) :
        super().__init__(err)

    def __str__( self ) : return "Newton method"

    # 牛頓求根: 由點 x1 起始
    def find_root( self , x1 ) :

        n = 1
        while True :
            x2 = x1 - fn(x1)/df(x1)
            if abs(fn(x2)) < self.err : break
            x1 , n = x2 , n+1

        self.n = n
        return x2
```

函式求根類別架構 (十)

```
if __name__ == '__main__' :

    # 畫函式圖形
    plot_fn(-10,10,200)

    foo = Bisection()
    rt = foo.find_root(2,3)
    print( "{:<17}:[{:>2}] {:>9.7f} {:>10.3e}".format( str(foo) ,
           foo.iter_no() , rt , fn(rt) ) )

    foo = Secant()
    rt = foo.find_root(2,3)
    print( "{:<17}:[{:>2}] {:>9.7f} {:>10.3e}".format( str(foo) ,
           foo.iter_no() , rt , fn(rt) ) )

    foo = Newton()
    rt = foo.find_root(2)
    print( "{:<17}:[{:>2}] {:>9.7f} {:>10.3e}".format( str(foo) ,
           foo.iter_no() , rt , fn(rt) ) )
```