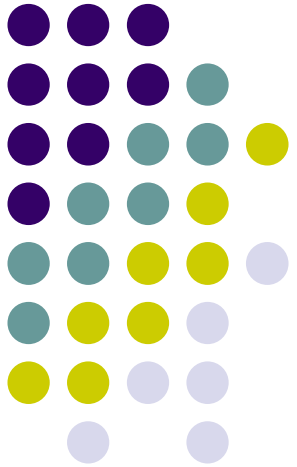
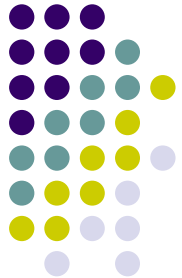
Python程式設計入門
類別

葉難

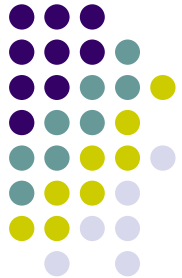


大綱



- 類別（class）：新式、舊式（古式）
- 型別（type）、類別（class）：3.x版合而爲一
- class述句
- 物件導向程式設計：封裝、繼承、重載
- 特殊方法、裝飾器、屬性（property）

各種語言構件，爲了...

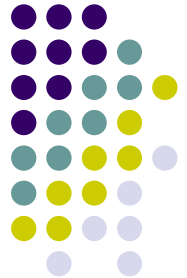


- 適切且清楚地表達意思
- 避免重複撰寫
- 開發「大型」軟體
- 維護：穩定（robust）性質
- 抽象化：黑盒子、共同介面

物件導向程式設計

- 封裝（encapsulation）：狀態與行爲
- 繼承（inheritance）：繼承階級架構
- 多型（polymorphism）：
  不同種類的個體擁有相同介面
  int與str都有「+」運算子
  list改寫（或重新實作）Sequence規定的介面
  函式可因參數型別不同而有不同行爲

class述句



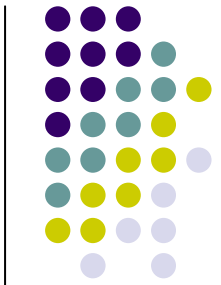
- 定義（建立）類別物件、指派名稱
- 語法，class擁有其範圍

```
class 類別名稱(父類別名...):
    述句...
```

- 預設繼承自「object」（3.x版）

```
class Deck():
    pass
```

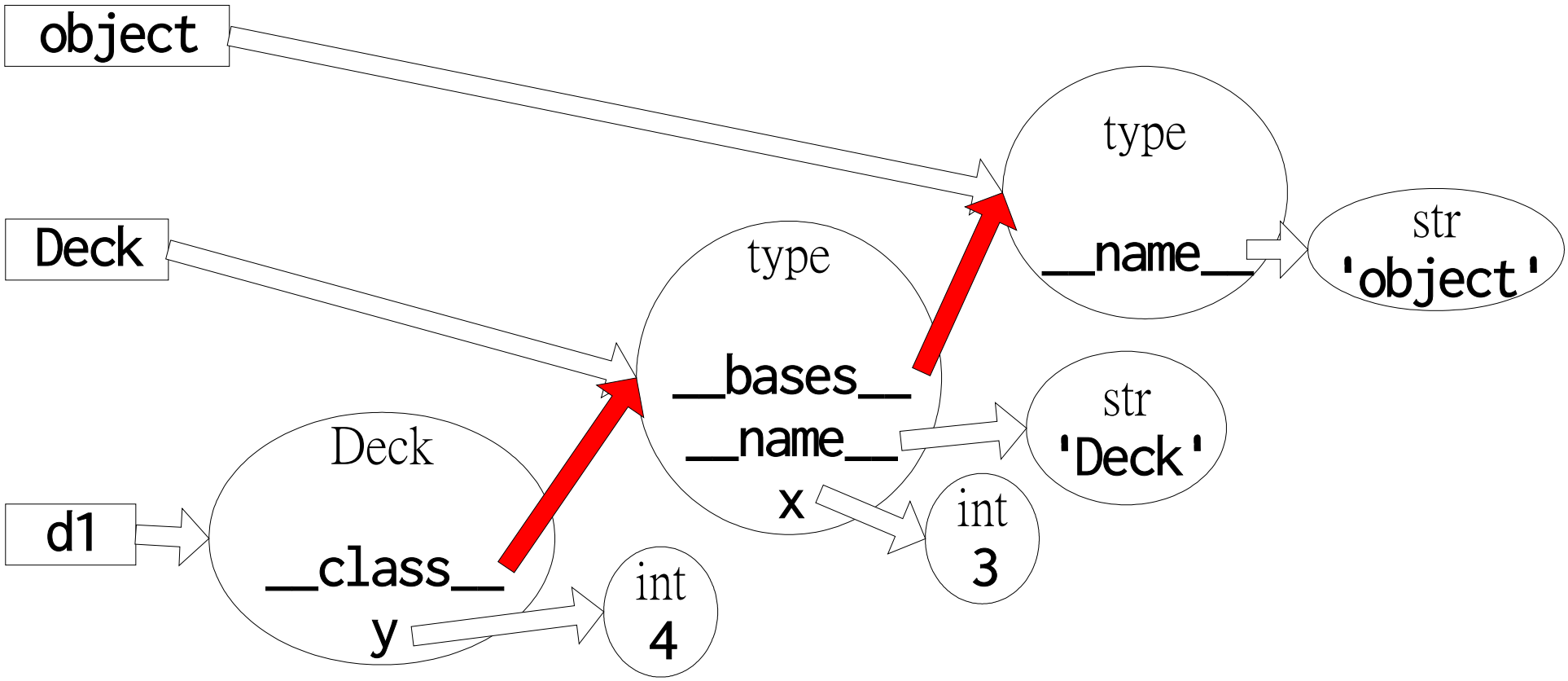
類別也是物件



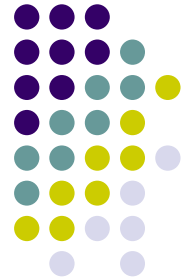
- 預設繼承自object，繼承了__call__、__new__、__str__等基本能力
- Deck是「類別」物件

```
    # 建構式（constructor）
d1 = Deck()             # d1也是物件，爲了區分
print(d1)               # 稱爲實體（instance）
Deck.x = 3          # 屬性項（attribute）
d1.y = 4            # 屬性項（attribute）
print(Deck.x, d1.y)
```

示意圖

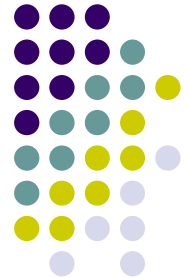

object
Deck
d1
type
__name__
str
'object'
type
__bases__
__name__
x
str
'Deck'
int
3
Deck
__class__
y
int
4

建構式（constructor）



- 型別object，已提供__call__（可被呼叫者）、__new__（配置記憶體與其他事務）
- 使用class述句定義類別時，需提供__init__（初始化方法），其參數self是新建立的實體，然後由你提供實體變數，也就是產生名稱指向物件

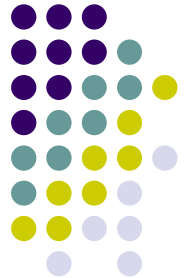
建立實體



- `__init__`：初始化方法，爲個別實體建立實體變數（instance variable）
- 方法（method）與`self`

```
class Person():
    def __init__(self, name, age):
        self.name = name    # 實體變數
        self.age = age      # 實體變數

    def say_hello(self):     # 方法
        print('Hello, I am ' + self.name)
```

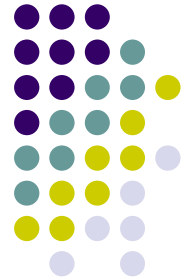
實體的方法



- 方法其實就是函式，但提供語法支援

```
p1 = Person('Amy', 25)

Person.say_hello(p1)
# 上下寫法意義相同
p1.say_hello()
```

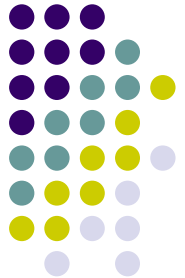
class述句擁有其範圍？



- 裡頭的函式，並未包含外圍範圍

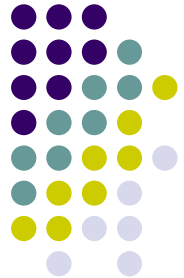
```
class Person():
    HI_STR = 'Hi, I am '   # 類別變數（或靜態變數）
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hi(self):      # 方法
        print(Person.HI_STR + self.name)
```

問題



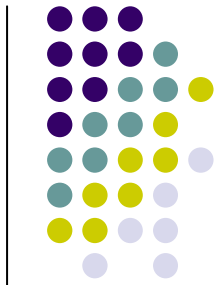
```
class MyClass():
    def __init__(self): self.x = 3
    def foo(self):
        self.x += 1;   print(self.x)
    def bar(tiger, n):
        tiger.x += n;  print(tiger.x)
c = MyClass()
c.foo(); MyClass.foo(c)     # ?
c.bar(-2)                   # ?
MyClass.bar()               # ?
c.foo(c)                    # ?
```

特殊方法



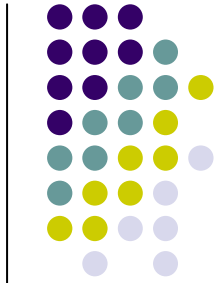
- `__init__`：初始化方法
- `__str__`：以易讀易懂的字串表達物件
- `__repr__`：以Python直譯器可看懂的字串表達物件，一般來說，執行此字串可建立出物件
- `__len__`：容器類物件，「長度」觀念
- `__iter__`、`__next__`：可迭代者、迭代器
- `__format__`、`__hash__`、`__bool__`、`__bytes__`、等等

__str__



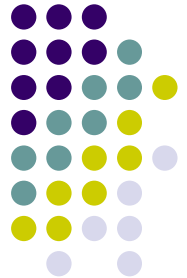
```
class Person():
      def __str__(self):
          return '<Person> %s, age %d' %
                                (self.name, self.age)

p1 = Person('Amy', 25)
print(p1, p1.__str__(), str(p1))

# <Person> Amy, age 25
```

__repr__



```
class Person():
    def __repr__(self):
        return "Person('%s', %d)" %
                            (self.name, self.age)

p1 = Person('Amy', 25)
exec( 'p3 = ' + repr(p1) )
# p3 = Person('Amy', 25)
print(p3)
```

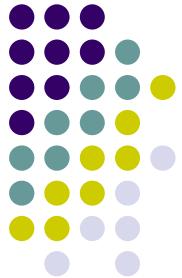
__iter__、__next__



- Iterable抽象型別：__iter__

```
class MyDeck():

    def __iter__(self):
        return MyDeckIterator(self)

    # 沒有 __next__
```

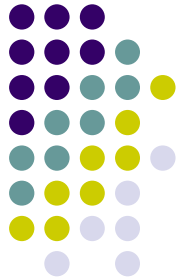
__iter__、__next__



- Iterator抽象型別：__next__

```
class MyDeckIterator():

 def __init__(self, deck):
        self.deck = deck

    def __next__(self):
        ...省略...
        ...耗盡時，引發異常StopIteration

    def __iter__(self): return self
```

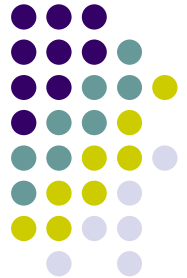
__call__



- 實體被當做函式呼叫時
- x(arg1, arg2, ...)等同於x.__call__(arg1, arg2, ...)
- 範例：Fibmemo.py

```
class Fibmemo():
    def __init__(self):
        self.memo = {0: 0, 1: 1}

    def __call__(self, n):
        if n not in self.memo:
            self.memo[n] = self(n-1)+self(n-2)
        return self.memo[n]
```

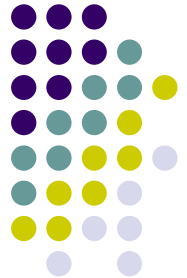
文脈（context）管理協定



- __enter__、__exit__與with述句
- 範例：Fibmemo.py

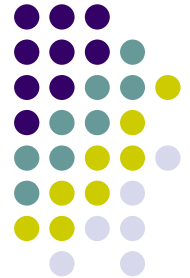
```
def __enter__(self):
    self.memo = {0: 0, 1: 1}
    return self

def __exit__(self, exc_type,
                        exc_value, traceback):
    self.memo = {0: 0, 1: 1}
    return False # 不壓抑異常
```

補充：**with**述句



```
with open(...) as fin:
    with open(...) as fout:
        ...

#### 上下寫法相同

with open(...) as fin, open(...) as fout:
    ...
```

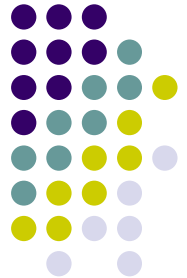
運算子重載（**overloading**）



- 「+」對應「__add__」
  「x + y」等同於「x.__add__(y)」
- 「and」對應「__and__」
- 「<」對應「__lt__」

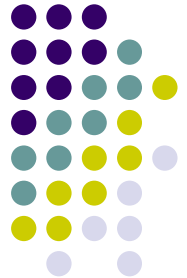
- 「x - y」，若x沒有__sub__且y是另一種型別，則會呼叫「y.__rsub__(x)」

範例：Card



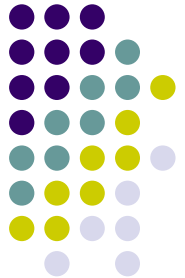
```
class Card():
    suits = {'spade':4, 'heart':3, 'diamond':2, 'club':1}
    ranks = {'A':14, '2':15, '3':3, '4':4, '5':5,
             '6':6, '7':7, '8':8, '9':9, '10':10,
             'J':11, 'Q':12, 'K':13}
    def __lt__(self, other):
        rank_diff = Card.ranks[self.rank] -
                                Card.ranks[other.rank]
        if rank_diff < 0:
            return True
        elif rank_diff == 0:
            if Card.ranks[self.rank] <
                                 Card.ranks[other.rank]:
                return True

        return False
```

裝飾器



- staticmethod：靜態方法，不會收到self，猶如一般函式
- classmethod：類別方法，第一個參數會是類別，慣例以cls爲名
- property：屬性，讀、寫、刪除；支援「obj.attr」語法

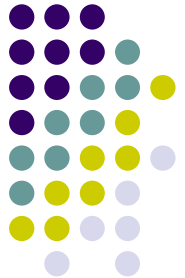
staticmethod



- 靜態方法，不會收到self，猶如一般函式

```
class MyClass():
    @staticmethod
    def foo(arg1, arg2):
        return arg1 + arg2

x = MyClass()
print(x.foo(3, 4))          # 印出7
print(MyClass.foo(5, 6))    # 印出11
```

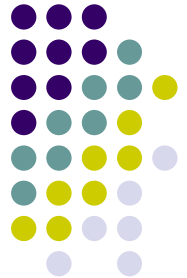
classmethod



- 類別方法，第一個參數會是類別，慣例以cls爲名

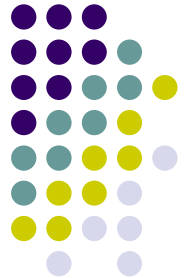
```
def __new__(cls, args):
        self = super().__new__(cls)
####
class MyStream():
    @classmethod
    def from_file(cls, filename):
        ...省略...
    @classmethod
    def from_socket(cls, sock):
        ...省略...
```

property



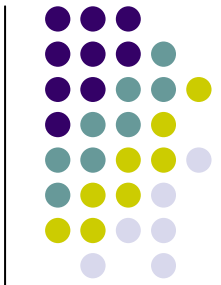
```
class Person():
    def __init__(self, name, birthyear):
        self._name = name
        self._birthyear = birthyear

    def get_birthyear(self):      # 讀
        return self._birthyear

    def set_birthyear(self, v):   # 寫
        self._birthyear = v

    def del_birthyear(self):      # 刪除
        del self._birthyear

    birthyear = property(get_birthyear, set_birthyear,
    # 屬性                                      del_birthyear)
```

@property，常用寫法



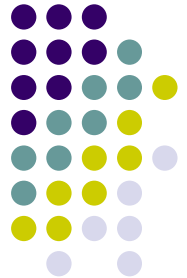
```
@property                # 讀
def birthyear(self):  return self._birthyear

@birthyear.setter        # 寫
def birthyear(self, v):  self._birthyear = v

@birthyear.deleter       # 刪除
def birthyear(self):  del self._birthyear
```

繼承



- duck typing（鴨子型別）
- subtyping（子類別繼承）：子類別繼承父類別所有的屬性項（介面）
- super()、私有名稱、抽象型別
- 「物件.屬性項」：屬性項搜尋程序

super()，做好初始化動作



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

class B(A): # B繼承自A，A繼承自object
    def __init__(self, z):
        # A.__init__(self, 0, 0)
        # super(B, self).__init__(0, 0)
        super().__init__(0, 0)
        self.z = z
```

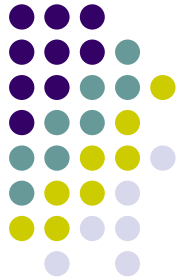
覆寫（overwrite）、重載（overload）

```
class Animal():
    def shout(self): print('Animal shout')

class Dog(Animal):
    def shout(self): print('wan wan')

class Cat(Animal):
    def shout(self): print('meow')
```

私有名稱「__name」

```
class A():
    def __init__(self):
        self.x = 3
        self.__pi = 3.14    # 私有名稱
                            # 會被改成_A__pi
class B(A):
    def bar(self):
        # return self.__pi   # 出錯
        return self._A__pi    # OK
```

collections.abc

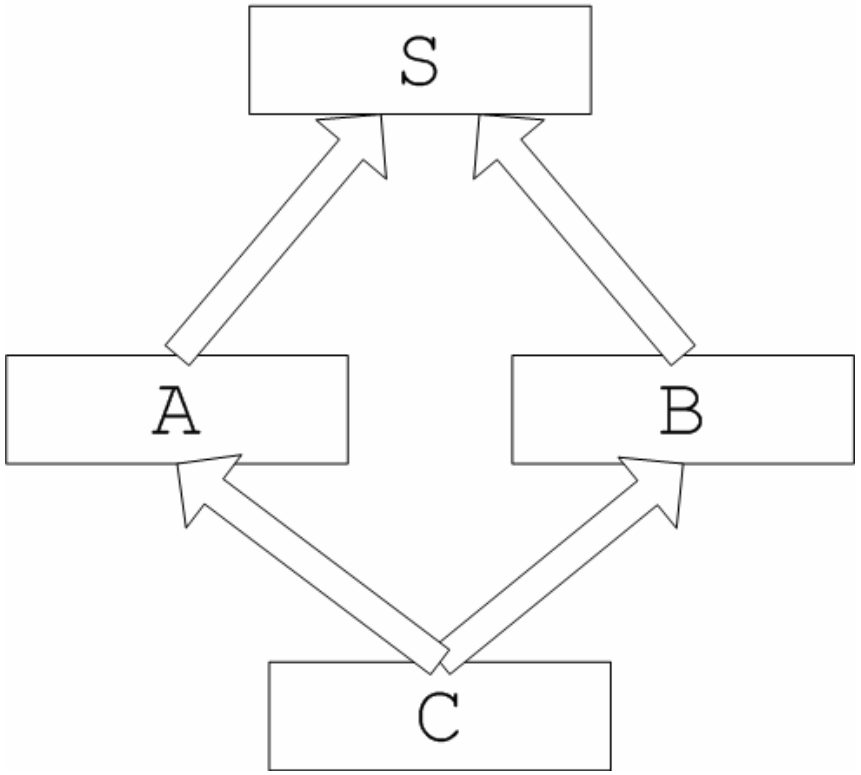


- Container、Sized、Iterable、Iterator、Sequence、Mapping、等等
- 若繼承自這些抽象型別，必須實作其介面，否則出錯
- 例如Sequence，需實作`__getitem__`、`__len__`、`__contains__`、`__iter__`、`__reversed__`、`index`、`count`等介面

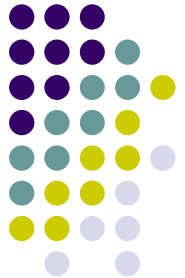
多重繼承

- 沒有眾人皆滿意的機制，總是會有例外

```
# 菱形
class S(): pass
class A(S): pass
class B(S): pass
class C(A, B): pass
# 順序是C A B S object
```

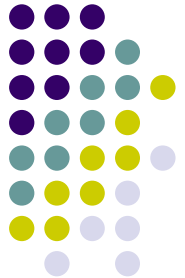


MRO



- 「物件.屬性項」：屬性項搜尋程序，又稱「MRO（method resolution order）」
- 先到實體與實體的類別裡尋找，之後根據類別定義列出的父類別順序，從左到右依序搜尋，

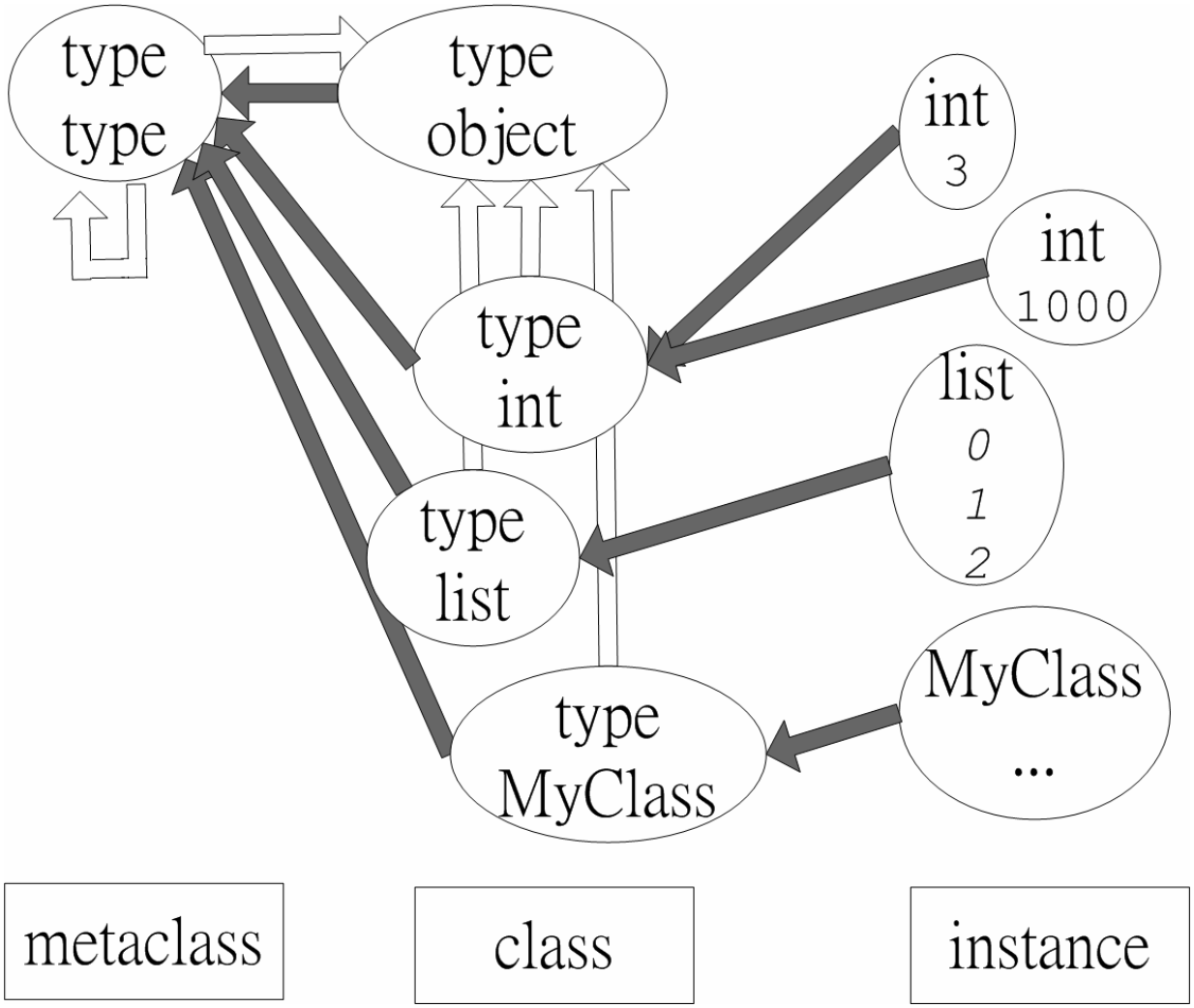
問題（mi.py）



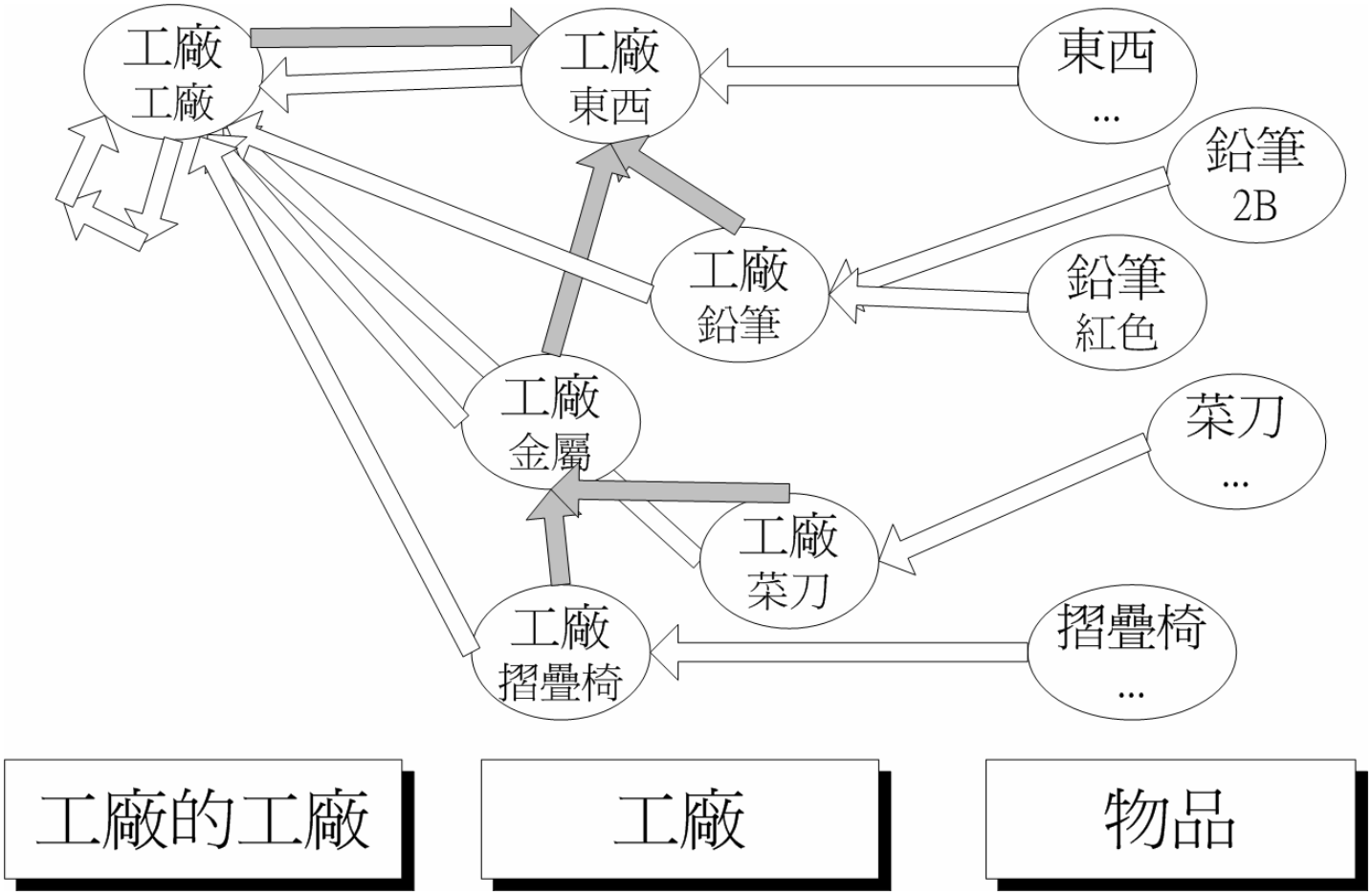
- 如何決定順序

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

# Z X Y ...之後是A還是B？
```

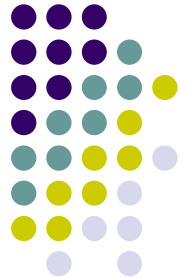
後設類別（**metaclass**）



譬喻



模組abc （Abstract Base Class）



```
from abc import ABCmeta

class MyIterable(metaclass=ABCmeta):

    @abstractmethod
    def __iter__(self):
        ...
    @classmethod
        def __subclasshook__(cls, C):
        ...
```

Q&A