The background is a gradient from dark red at the top to dark blue at the bottom. It features several faint, white, concentric circular patterns. Some of these circles have degree markings (40, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 260) and arrows indicating a clockwise direction. The main title is centered in the upper half of the image.

多媒體數據分析 與應用

張家瑋 博士

助理教授

國立臺中科技大學資訊工程系



PYTHON 是大量使用物件導向概念的語言

從 DATA FRAME 了解屬性與方法

```
import pandas as pd # 引用套件並縮寫為 pd

courses = ["A", "B", "C", "D", "E", "F"]

students = [48, 48, 54, 12, 50, 70]

school_dict = {"courses": courses,
               "students": students
               }

school_df = pd.DataFrame(school_dict)
print(school_df.dtypes) # school_df 有 dtypes 屬性
school_df.head(n = 3) # school_df 有 head() 方法
```

```
courses      object
students      int64
dtype: object
```

	courses	students
0	A	48
1	B	48
2	C	54

大綱

1. 程序導向與物件導向
2. Python的物件導向
3. 建立類
4. Self的重要性
5. 理解建構函式
6. 類變數和例項變數

程序導向與物件導向

- **程序導向**的程式設計把函式作為程式的基本單元。程式設計時，編寫一組一組的函式，然後一步一步按照順序的執行各個函式。通常為了簡化程式，將大塊函式通過切割成小塊函式來降低系統的複雜度。
- **物件導向(Object-oriented programming, OOP)**的程式設計把物件作為程式的基本單元，以物件概念設計程式時，每個物件都可以接收其他物件發過來的訊息，並處理這些訊息，計算機程式的執行就是一系列訊息在各個物件之間傳遞，各個物件呼叫相關的方法。

PYTHON的物件導向

- OOP程式設計是利用“類”和“物件”來建立各種模型來實現對真實世界的描述，使用物件導向程式設計的原因一方面是因為它可以使程式的維護和擴充套件變得更簡單，並且可以大大提高程式開發效率，另外，基於物件導向的程式可以使它人更加容易理解你的程式碼邏輯，從而使團隊開發變得更容易。
- 物件導向的幾大核心特性
 1. **Class 類**：一個類指相同事物相同特徵提取，把相同的屬性方法提煉出來定義在類中
 2. **Object 物件**：一個物件是類的例項，物件是具體的，類是抽象
 3. **封裝**：對外部世界隱藏物件的工作細節
 4. **繼承**：一個子類繼承基類的欄位和方法
 5. **多型**：對不同類的物件使用同樣的操作

建立類

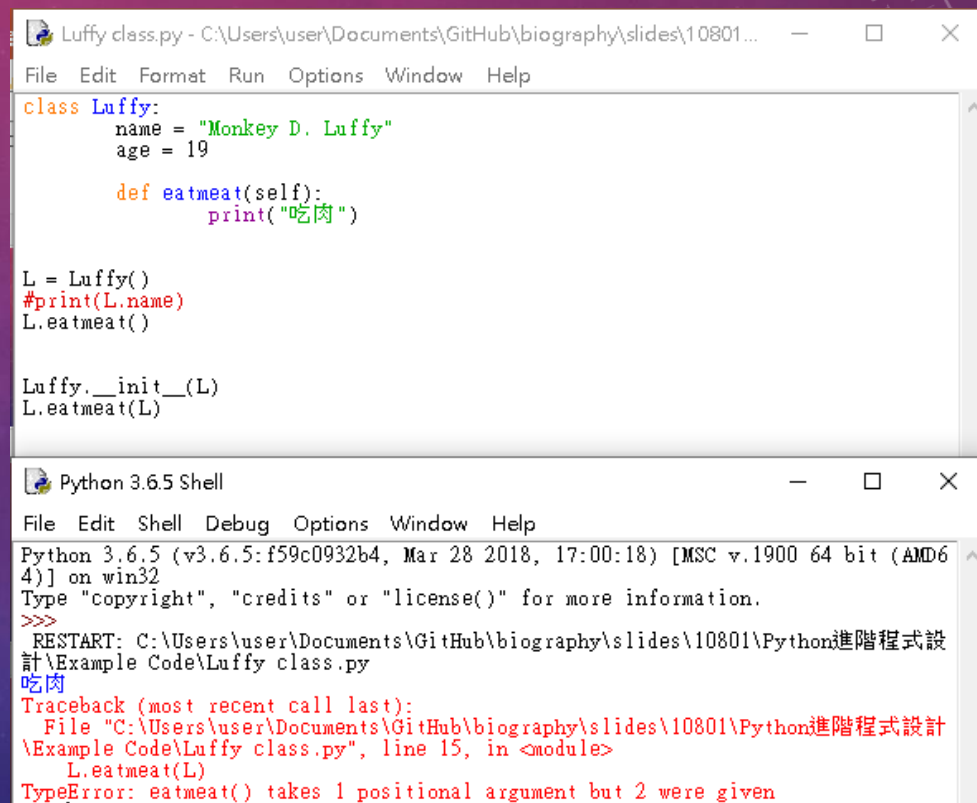
- Python中使用 **class** 關鍵字修飾類，類名一般採用首字母大寫，小圓號裡面表示繼承，所有類最後都繼承自object。
- 在類中我們可以定義屬性和方法，注意這個地方有一個叫法問題，方法其實和之前寫的函式一樣，但是在類中定義的稱為方法，兩個的區別在呼叫的時候，方法需要特定的物件，而函式不需要。

```
Luffy class.py
1 class Luffy:
2     name = "Monkey D. Luffy"
3     age = 19
4
5     def eatmeat(self):
6         print("吃肉")
7
8 L = Luffy()
9 print(L.name)
10 L.eatmeat()
```

*在類中定義方法的要求，就是第一個引數必須時self * 7

SELF 的重要性

- 不僅是__init__需要self作為第一個引數，類中定義的所有方法都需要。
- 類程式碼設計為在所有物件例項間共享，**self**可以幫助標示要處理哪個物件例項的資料。



The screenshot shows a Python IDE with two windows. The top window, titled 'Luffy class.py', contains the following code:

```
class Luffy:
    name = "Monkey D. Luffy"
    age = 19

    def eatmeat(self):
        print("吃肉")

L = Luffy()
#print(L.name)
L.eatmeat()

Luffy.__init__(L)
L.eatmeat(L)
```

The bottom window, titled 'Python 3.6.5 Shell', shows the execution of the code. It displays a restart message and a traceback for a `TypeError`:

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:\Users\user\Documents\GitHub\biography\slides\10801\Python進階程式設計\Example Code\Luffy class.py
吃肉
Traceback (most recent call last):
  File "C:\Users\user\Documents\GitHub\biography\slides\10801\Python進階程式設計\Example Code\Luffy class.py", line 15, in <module>
    L.eatmeat(L)
TypeError: eatmeat() takes 1 positional argument but 2 were given
```


理解建構函式

- `__init__()` 方法是一個特殊的方法，在物件例項化的時候呼叫（`init` 表示初始化的意思，是 `initialization` 的簡寫）注意前後兩個下劃線不可以省略，這個方法也叫構造方法。
- 在定義類的時候，如果沒有顯示定義一個 `__init__()` 方法，程式預設呼叫一個無參數的 `__init__()` 方法，但是要注意，一個類中只定義一個建構函式，編寫多個例項化的時候會呼叫最後一個。

```
Luffy class.py
1 class Luffy:
2     def __init__(self, dream):
3         print("class 初始化的時候，執行__init__")
4         self.dream = dream
5
6     name = "Monkey D. Luffy"
7     age = 19
8
9     def eatmeat(self):
10        print("吃肉", self.dream)
11
12
13 L = Luffy("One Piece")
14 L.eatmeat()
```

類變數和例項變數

```
ClassInstance.py
1 class Luffy:
2     def __init__(self):
3         self.name="Luffy"
4         name = "Monkey D. Luffy"
5         age = 19
6
7 L = Luffy()
8 print(L.name)
9 print(Luffy.name)
```

```
ClassInstance.py
1 class Luffy:
2     def __init__(self):
3         self.name="Luffy"
4         name = "Monkey D. Luffy"
5         age = 19
6
7 L = Luffy()
8
9 L.name = "777"
10 Luffy.name = "666"
11
12 print(L.name)
13 print(Luffy.name)
14
15 LL = Luffy()
16 print(LL.name)
17 print(Luffy.name)
```



GO THROUGH **EXAMPLES**

根據類別建立物件(OBJECT)

當類別 `Course` 被定義完成，就可以使用 `Course()` 當作建構子 (Constructor) 建立物件。

```
class Course:
    '''這是一個叫做 Courses 的類別''' # Doc string
    def __init__(self, course, students):
        self.course = course
        self.students = students

# 根據 Course 類別建立一個物件 A
a = Course("A", 48)
print(a)
```

```
<__main__.Course object at 0x000001E82FC9F7F0>
```


使用物件的屬性(ATTRIBUTE)

在物件名稱後面使用，接屬性名稱就可以使用。

```
class Course:
    '''這是一個叫做 Course 的類別''' # Doc string
    def __init__(self, course, students):
        self.course = course
        self.students = students

# 根據 Course 類別建立一個物件 A
a = Course("A", 48)

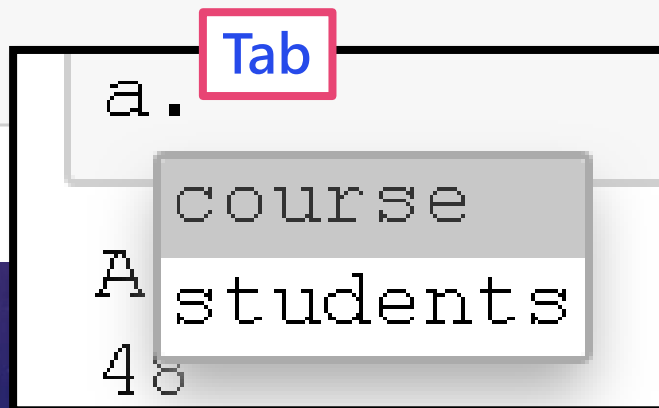
# 印出 a 的兩個屬性
print(a.course)
print(a.students)

# 印出 a 的類別 doc string
print(a.__doc__)
```

A

48

這是一個叫做 Course 的類別



使用物件的屬性(ATTRIBUTE)

```
['_class_',  
 '_delattr_',  
 '_dict_',  
 '_dir_',  
 '_doc_',  
 '_eq_',  
 '_format_',  
 '_ge_',  
 '_getattr_',  
 '_gt_',  
 '_hash_',  
 '_init_',  
 '_init_subclass_',  
 '_le_',  
 '_lt_',  
 '_module_',  
 '_ne_',  
 '_new_',  
 '_reduce_',  
 '_reduce_ex_',  
 '_repr_',  
 '_setattr_',  
 '_sizeof_',  
 '_str_',  
 '_subclasshook_',  
 '_weakref_',  
 'course',  
 'students']
```

```
class Course:  
    '''這是一個叫做 Course 的類別''' # Doc string  
    def __init__(self, course, students):  
        self.course = course  
        self.students = students  
  
# 根據 Course 類別建立一個物件 A  
a = Course("A", 48)  
  
# 印出 a 的兩個屬性  
print(a.course)  
print(a.students)  
  
# 印出 a 的類別 doc string  
print(a.__doc__)  
dir(a)
```

定義方法 (METHOD)

```
class Course:
    '''這是一個叫做 Course 的類別''' # Doc string
    def __init__(self, course, students):
        self.course = course
        self.students = students
    def print_info(self):
        print(self.course, "課程有", self.students, "位學生修習！")

# 根據 Course 類別建立一個物件 A
a = Course("A", 48)

# 根據 Course 類別建立一個物件 B
b = Course("B", 48)

# 使用 a 的 print_info() 方法
a.print_info()

# 使用 b 的 print_info() 方法
b.print_info()
```

A 課程有 48 位學生修習！

B 課程有 48 位學生修習！

繼承 (INHERITANCE)

```
class Course:
    '''這是一個叫做 Course 的類別''' # Doc string
    def __init__(self, course, students):
        self.course = course
        self.students = students
    def print_info(self):
        print(self.course, "課程有", self.students, "位學生修習！")

class Homework(Course):
    '''
    這是一個叫做 Homework 的類別。
    Homework 繼承 Course 類別，她新增了一個 print_hw() 方法
    '''
    def print_hw(self):
        print(self.course, "門課程預計會有", self.students * 30, "份作業！")

# 根據 Homework 類別建立一個物件 A
a = Homework("A", 48)

# 使用 a 的 print_hw() 方法
a.print_hw()

# 檢查 a 是否還擁有 print_info() 方法
a.print_info()
```

A 門課程預計會有 1440 份作業！
A 課程有 48 位學生修習！

在繼承時使用 SUPER()

```
class OnlyCourse:
    '''這是一個叫做 OnlyCourse 的類別''' # Doc string
    def __init__(self, course):
        self.course = course

class Course(OnlyCourse):
    '''這是一個叫做 Course 的類別''' # Doc string
    def __init__(self, course, students):
        super().__init__(course)
        self.students = students
    def print_info(self):
        print(self.course, "課程有", self.students, "位學生修習！")

# 根據 Course 類別建立一個物件 a
a = Course("A", 48)

# 印出 a 的兩個屬性
print(a.course)
print(a.students)
print(a.print_info())
```

```
A
48
A 課程有 48 位學生修習！
None
```

在繼承時改寫方法 (OVERRIDE)

```
class Course:
    '''這是一個叫做 Course 的類別''' # Doc string
    def __init__(self, course, students):
        self.course = course
        self.students = students
    def print_info(self):
        print(self.course, "課程有", self.students, "位學生修習！")

class Homework(Course):
    '''
    這是一個叫做 Homework 的類別。
    Homework 繼承 Course 類別，她新增了一個 print_hw() 方法
    '''
    def print_hw(self):
        print(self.course, "門課程預計會有", self.students * 30, "份作業！")
    def print_info(self):
        print(self.course, "課程有", self.students, "位學生修習！(我是重複的那一個)")

# 根據 Homework 類別建立一個物件 A
a = Homework("A", 48)

# 使用 a 的 print_hw() 方法
a.print_hw()

# 檢查 a 是否還擁有 print_info() 方法
a.print_info()
```

A 門課程預計會有 1440 份作業！
A 課程有 48 位學生修習！(我是重複的那一個)

大綱

1. 類的訪問許可權
2. 靜態方法、例項方法、類方法
3. 繼承
4. 多型
5. 封裝
6. 解構函式
7. 類中特殊成員

類的訪問許可權

- 在類中我們定義自己的屬性和方法，通過例項化後的物件，可以在外部進行呼叫，但是我們也可以對屬性和方法的訪問許可權進行設定，讓外界無法訪問。
- 在python中例項的變數名以__開頭，就會變成私有變數（ private ）外部不能訪問。

類的訪問許可權

```
class Luffy:
    def __init__(self,dream):
        print("類初始化的時候執行__init__")
        self.__dream=dream
    __name="蒙奇·D·路飛" #加__字首變成私有變數
    age=10
    def eatmeat(self):
        print("吃肉", self.__dream) #私有變數在外部無法訪問，內部可以訪問
```

```
l = Luffy("夢想是找到傳說中的One Piece，成為海賊王") #例項化
l.eatmeat() #呼叫類的方法
print(l.age)
```

類初始化的時候執行__init__
吃肉 夢想是找到傳說中的One Piece，成為海賊王
10

```
print(l.__name) #報錯 'Lufei' object has no attribute 'name'
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-12-7cf82b4b8a08> in <module>()
----> 1 print(l.__name) #報錯 'Lufei' object has no attribute 'name'

AttributeError: 'Luffy' object has no attribute '__name'
```

類的訪問許可權

- 如果希望類中的變數都改為私有變數，那麼類中的變數就都無法被外部訪問了。
- 如果想要對這些變數進行操作，就只能透過方法來解決。

類的訪問許可權

```
class Luffy:
    def __init__(self,dream):
        print("類初始化的時候執行__init__")
        self.__dream=dream

    name="蒙奇·D·路飛"
    __age=10
    #加__字首變成私有變數
    #通過方法或者私有變數返回

    def getage(self):
        return self.__age
        #通過方法對私有變數進行賦值操作，並可以進行資料安全驗證

    def setage(self,age):
        if age<0:
            return "太小了"
        else:
            self.__age = age
            return "ok"

    def eatmeat(self):
        print("吃肉", self.__dream)
        #私有變數在外部無法訪問，內部可以訪問
```

```
l=Luffy("夢想是找到傳說中的One Piece，成為海賊王")    #例項化
l.eatmeat()        #呼叫類的方法
l.setage(100)#對私有變數年齡進行設定
print(l.getage()) #獲取私有變數年齡
```

類初始化的時候執行__init__
吃肉 夢想是找到傳說中的One Piece，成為海賊王
100

類的訪問許可權

- 既然變數可以，那方法是不是能設定為私有方法呢？

```
class Luffy:
    def __init__(self, dream):
        print("類初始化的時候執行__init__")
        self.__dream=dream

    name="蒙奇·D·路飛"
    __age=10
    #加__字首變成私有變數
    #通過方法或者私有變數返回

    def getage(self):
        return self.__age
        #通過方法對私有變數進行賦值操作，並可以進行資料安全驗證

    def setage(self, age):
        if age<0:
            return "太小了"
        else:
            self.__age = age
            return "ok"

    def eatmeat(self):
        print("吃肉", self.__dream)
        #私有變數在外部無法訪問，內部可以訪問

    def __getdream(self):#定義一個私有方法
        print(self.__dream) #class外界無法訪問，內部可以同self.__getdream()訪問
```

```
l.__getdream()
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-11-3f451af235db> in <module>()
----> 1 l.__getdream()

AttributeError: 'Luffy' object has no attribute '__getdream'
```


靜態方法、例項方法、類方法

- 例項方法隱含的引數為類例項 `self`
- 類方法隱含的引數為類本身 `cls`
 - 類方法無法訪問例項變數，但可以訪問類變數
 - 類方法更類似Java物件導向中的靜態方法
- 靜態方法沒有隱含引數
 - 類例項也可以直接呼叫靜態方法。
 - 靜態方法無法訪問例項變數
 - 靜態方法有點像函式工具庫的作用

靜態方法、例項方法、類方法

```
class Bird:
    name="lalal"

    def fly(self):
        print("fly")
        print(self) #輸出: <__main__.Bird object at 0x00000000022032E8>

    @classmethod #類方法(由@classmethod裝飾的方法)
    def eat(cls):#類方法的第一個引數都是類物件而不是例項物件
        #類方法可以通過類或它的例項來呼叫
        print("eat")
        print(cls) #輸出:<class '__main__.Bird'>

    @staticmethod #靜態方法(由@staticmethod裝飾的方法)
    def sleep(): #不需要定義例項即可使用靜態方法。另外，多個例項共享此靜態方法。
        #使用了靜態方法，就不能再使用self
        # print(name) #靜態方法不能訪問類變數和例項變數會報錯
        print("sleep")

b = Bird() # Bird.fly(b) 傳遞self進去，不傳遞只是一個語法糖

b.fly() #例項方法只能被例項物件呼叫

#可以被類或類的例項物件呼叫
b.eat()
b.sleep()

#可以被類或類的例項物件呼叫
Bird.eat()
Bird.sleep()
```

```
fly
<__main__.Bird object at 0x000001EE76A63E80>
eat
<class '__main__.Bird'>
sleep
eat
<class '__main__.Bird'>
sleep
```

靜態方法、例項方法、類方法

- 靜態方法：無法訪問類屬性、例項屬性，相當於一個相對獨立的方法，跟類其實沒什麼關係，換個角度來講，其實就是放在一個類的作用域裡的函式而已。
- 類方法：可以訪問類屬性，但無法訪問例項屬性。上述的變數name，在類裡是類變數，在例項中又是例項變數，所以容易混淆。

繼承

- 繼承的語法，在定義好的類小括號裡面寫上繼承的類名，此時，被繼承的類稱為父類或者基類，繼承的類的稱為子類或派生類。

```
#父類
class Dragon:
    name="蒙奇·D·龍"
    def Getdream(self):
        print("推翻世界政府，建立和諧，自由，平等，充滿夢想的世界。")
#子類
class Luffy(Dragon):
    pass

l=Luffy() #例項化子類
l.Getdream() #從父類繼承來的
```

推翻世界政府，建立和諧，自由，平等，充滿夢想的世界。

- Luffy是子類，Dragon是父類，子類可以繼承父類非私有的所有屬性和方法

繼承

- Python支援多重繼承，在小括號裡面可以通過都好分隔寫多個父類的名稱，需要注意的是當多個父類的時候，python會從左到右搜尋。
- 子類可繼承多個父類，同時獲得多個父類的所有非私有功能

```
#父類
class Dragon:
    name="蒙奇·D·龍"
    def Getdream1(self):
        print("推翻世界政府，建立和諧，自由，平等，充滿夢想的世界。")

class Shanks:
    name="香克斯"
    def Getdream2(self):
        print("平衡新世界的勢力。")

class Luffy(Dragon,Shanks):
    pass

l=Luffy() #例項化子類
print(l.name) #當訪問的屬性兩個父類中都有定義的時候以第一個為主
l.Getdream1() #從父類Dragon繼承來的
l.Getdream2() #從父類Shanks繼承來的
```

蒙奇·D·龍

推翻世界政府，建立和諧，自由，平等，充滿夢想的世界。

平衡新世界的勢力。

多型

- 多型意味著即使不知道變數所引用的物件型別是什麼，也能對物件進行操作，多型會根據物件的不同而表現出不同的行為。
- 多型我們不用對具體的子型別進行了解，到底呼叫哪一個方法，在執行的時候會由該物件的確切型別決定，使用多型，我們只管呼叫，不用管細節。

```
#父類
class Dragon:
    name="蒙奇·D·龍"
    def Getdream(self):
        print("推翻世界政府，建立和諧，自由，平等，充滿夢想的世界。")

#子類
class Luffy(Dragon):
    name="蒙奇·D·路飛"
    def Uniqueskills(self):
        print("三檔")

#子類
class Ace(Dragon):
    name="艾斯"
    def Uniqueskills(self):
        print("火拳")

#定義父類作為引數，所有的子類都可以傳參進去
def Show(R):
    R.Uniqueskills()
    R.Getdream()

Show(Luffy()) #輸出三檔
Show(Ace()) #輸出火拳
```

三檔

推翻世界政府，建立和諧，自由，平等，充滿夢想的世界。

火拳

推翻世界政府，建立和諧，自由，平等，充滿夢想的世界。

封裝

- 從學習函式以來都在提及封裝的概念，封裝我們可以理解為，不用管具體的實現細節，直接呼叫即可，就像我們看電視，完全不用管電視是怎麼播放的，只需要按下按鈕可以觀看即可

解構函式

- 當使用del 刪除物件時，會呼叫他本身的解構函式，另外當物件在某個作用域中呼叫完畢，在跳出其作用域的同時解構函式也會被呼叫一次，這樣可以用來釋放記憶體空間。
- 解構函式：在例項釋放、銷燬的時候執行、通常用於做一些收尾工作，如關閉一些資料

```
class Luffy:
    def __init__(self):
        self.name="路飛"
    def __del__(self):
        print("掛了")
```

```
L = Luffy()
```

```
print(L.name)
```

```
路飛
```

```
L.__del__()
```

```
掛了
```

```
print(L.name)
```

```
路飛
```

```
L = None
```

```
掛了
```

```
print(L.name)
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-7-4713d913834d> in <module>()
----> 1 print(L.name)
```

```
AttributeError: 'NoneType' object has no attribute 'name'
```


類中特殊成員

```
class BigBrid:
    '''描述類的資訊''' #BigBrid.__doc__ 可以看到
    name="bb"
    def eat(self):
        print("吃")
    def __str__(self):
        return 'lidao'

b=BigBrid()
b.name="lala"
print(b.__doc__) #輸出：描述類的資訊
print(b.__module__)#輸出：__main__
print(b.__class__)#輸出：<class '__main__.BigBrid'>
#__init__ 構造方法，通過類建立物件時，自動觸發執行
#__del__ 析構方法，當物件在記憶體中被釋放時，自動觸發執行。
print(BigBrid.__dict__) #獲取類的成員
print(b.__dict__) #獲取 物件b 的成員 輸出：{'name': 'lala'}
#__str__ 如果一個類中定義了__str__方法，那麼在列印 物件 時，預設輸出該方法的返回值。
print(b) #輸出 lidao
```

The background features a vertical gradient from red at the top to blue at the bottom, overlaid with a field of small white dots. On the left side, there are several white circular and semi-circular graphic elements. A prominent circular scale with degree markings (40, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 260) and arrows is visible. Other elements include concentric circles, dashed lines, and curved arrows, some of which are partially cut off by the frame.

THANK YOU