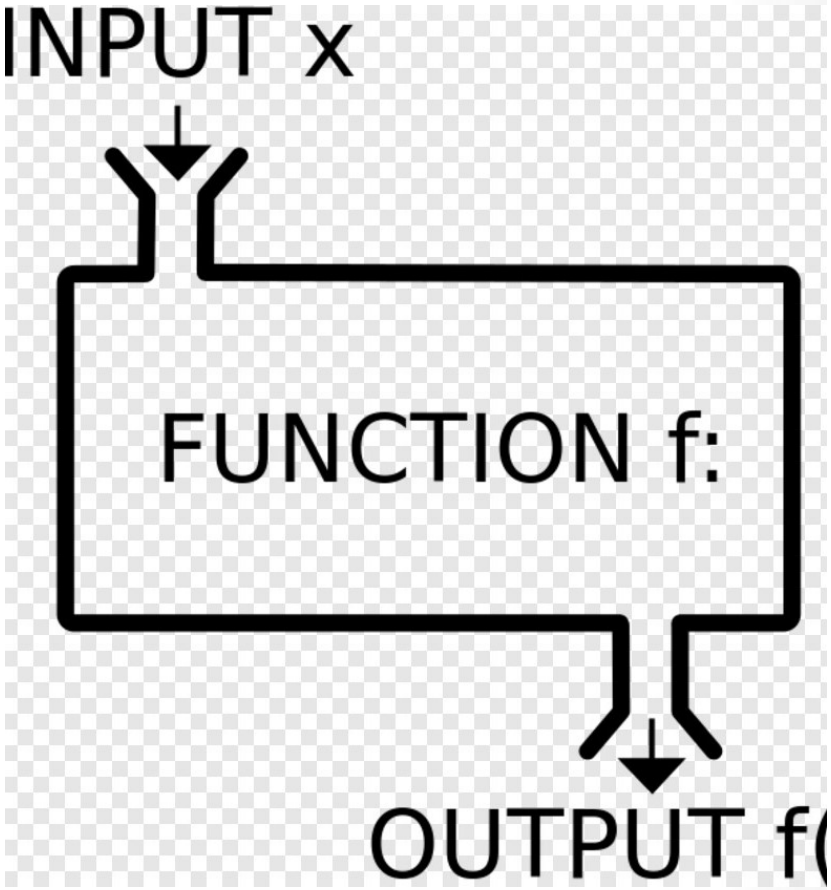
# FUNCTION AND IMPORTING PACKAGES




Chih-Chung Hsu (許志仲)
Institute of Data Science
National Cheng Kung University
https://cchsu.info

# Variable Scope

- nonlocal and global 兩個修飾子，分別運用在全域與半全域

```
1   ##  variable  scop
2
3   x  =  'global  variable'
4
5   def  example():
6       global  x
7       y  =  'local  variable'
8       x  =  x  *  2
9       print(x)
10      print(y)
11
12  example()
```

```
global variableglobal variable
local variable
```

```
1   ##  variable  scop
2
3   x  =  'global  variable'
4
5   def  example():
6       x='local  variable'
7       print('inner-',x)
8       def  nl():
9           nonlocal  x
10          x  =  'inner  variable'
11          print('inner2-',  x)
12
13      nl()
14      print('after  call  func-',  x)
15
16  example()
17  print('global-',  x)
```

```
inner- local variable
inner2- inner variable
after call func- inner variable
global- global variable
```

# closure 函式

- 類似前一節的內部函式，在函式內動態建立一個函式，回傳該函式，稱作closure。
- 使用 closure 的好處是可以看到外部函式的變數，讓程式碼集中在外部函式內，而非宣告定義在外部函式之外，避免程式碼分散不易閱讀。

- Definition on WiKi
  - *a **closure** (also **lexical closure** or **function closure**) is a technique for implementing lexically scoped name binding in a language with first-class functions XDD*

# Closure

```
1   global_var  =  100
2   def  convert(a=10):
3       local_var  =  10
4       def  weighting(var):
5           print('var=',  var)
6           return  var
7       local_var  =  weighting(local_var)
8
9   convert()
10  print(global_var)
11  #  print(local_var)
```

```
var= 10
100
```

- 傳統func:
  - y is a local var of convert., glob is a global variable

```
1   global_var  =  100
2   def  convert(a=10):
3       local_var  =  10
4       def  weighting():
5           print(local_var  +  global_var)
6
7       return  weighting
8
9   c=convert()
10  c()
11  #  print(local_var)
```

```
110
```

- Closure
  - Return subfunction…
    - Local_var is preserved in weighting function!!

# Closure Function

- 好，那麼為什麼有需要寫這麼複雜的 Closure?

```
##  Closure  -  why

list1  =  []
for  i  in  range(3):
    def  myfunc(a):
        return  i+a
    list1.append(myfunc)

for  f  in  list1:
    print(f(1),  end=',')
print()
```

- Guess what?
  - Output will be [>> 3,3,3,]

# Closure Function

- 大功告成，你把uncertainty 的variable i給保留下來了!!

```
1   ##  Colsure-why-2
2   list1  =  []
3  for  i  in  range(3):
4      def  myfunc(i):
5          def  wrapper(a):
6              return  i  +  a
7          return  wrapper
8      list1.append(myfunc(i))
9
10 for  f  in  func_list:
11     print(f(1),  end=',')
```

```
1, 2, 3,
```

# Closure 函式

| | | |
|---|---|---|
| 1 | def hello(msg): | |
| 2 | def say(hi): | |
| 3 | return hi+msg | |
| 4 | return say | Hello,Claire |
| 5 | x=hello('Claire') | Hi,Fiona |
| 6 | y=hello('Fiona') | |
| 7 | print(x('Hello,')) | |
| 8 | print(y('Hi,')) | |

- 第1 到4 行：定義函式hello，輸入參數msg，在函式內定義內部函式say，輸入參數hi，內部函式say 用於回傳參數hi( 函式say 的參數) 串接參數msg( 函式hello 的參數) 的字串( 第2 到3 行)，最後回傳函式物件say。

# 程式解說

- 第5 行：呼叫函式hello 以「Claire」為輸入，將回傳的函式物件指定給變數x。

- 第6 行：呼叫函式hello 以「Fiona」為輸入，將回傳的函式物件指定給變數y。
- 第7行：呼叫函式x以「Hello,」為輸入，將回傳的字串利用函式print顯示在螢幕上。
- 第8行：呼叫函式x以「Hi,」為輸入，將回傳的字串利用函式print顯示在螢幕上。

# Decorator( 裝飾器)

- 函式debug 允許輸入一個函式func1，在函式debug 內定義另一個函式func2，函式func2 接收傳入函式func2 的位置引數args 與關鍵字引數kwargs，顯示函式func1的函式名稱、說明文件到螢幕上

```
def debug(func1):
    def func2(*args, **kwargs):
        print(' 正在執行函式 ', func1.__name__)
        print(' 函式的說明文件爲 ', func1.__doc__)
        print(' 位置引數 ', args)
        print(' 關鍵引數 ', kwargs)
        return func1(*args, **kwargs)
    return func2
```

# Decorator( 裝飾器)

- 接著顯示傳入函式func2 的位置引數與關鍵字引數在螢幕上，回傳函式func1 以位置引數args 與關鍵字引數kwargs 為輸入的結果，最後函式debug 執行結束回傳函式func2。

```
def debug(func1):
    def func2(*args, **kwargs):
        print(' 正在執行函式 ', func1.__name__)
        print(' 函式的說明文件爲 ', func1.__doc__)
        print(' 位置引數 ', args)
        print(' 關鍵引數 ', kwargs)
        return func1(*args, **kwargs)
    return func2
```

# Decorator( 裝飾器)

- 使用上可以輸入一個函式物件到裝飾器，使用變數接收裝飾器所回傳的函式物件，輸入到裝飾器的函式名稱與接收裝飾器所回傳的函式名稱，可以使用相同函式名稱

```
def debug(func1):
    def func2(*args, **kwargs):
        print(' 正在執行函式 ', func1.__name__)
        print(' 函式的說明文件爲 ', func1.__doc__)
        print(' 位置引數 ', args)
        print(' 關鍵引數 ', kwargs)
        return func1(*args, **kwargs)
    return func2
```

# Decorator( 裝飾器)

- 如以下範例都使用add，但輸入的函式物件add 與回傳的函式物件 add 是指向不同的函式物件，也就是經由裝飾器可以將函式物件轉換成另一個函式物件。

# Decorator( 裝飾器) Example

```
def add(a, b):
    ' 回傳 a 加 b 的結果 '
    return a+b
add = debug(add)
print(add(1, b=2))
```

上述程式執行結果如下。

```
正在執行函式 add
函式的說明文件爲 回傳 a 加 b 的結果
位置引數 (1,)
關鍵引數 {'b': 2}
3
```

# Decorator( 裝飾器)

- 也可以利用「@」簡化裝飾器的使用步驟，在需要裝飾器的函式的上一行，使用「@」串接裝飾器名稱，就可以達成使用裝飾器改變下一行所定義的函式。

```
@debug
def add(a, b, c):
    ' 回傳 a+b+c 的結果 '
    return a+b+c
print(add(1, 2, c=3))
```

# Decorator( 裝飾器): @

- 程式執行結果如下

```
正在執行函式 add
函式的說明文件爲 回傳 a+b+c 的結果
位置引數 (1, 2)
關鍵引數 {'c': 3}
6
```

# Decorator( 裝飾器)

```
1  def debug(func1):
2      def func2(*args, **kwargs):
3          print(' 正在執行函式 ', func1.__name__)
4          print(' 函式的說明文件爲 ', func1.__doc__)
5          print(' 位置引數 ', args)
6          print(' 關鍵引數 ', kwargs)
7          return func1(*args, **kwargs)
8      return func2
9  def add(a, b):
10     ' 回傳 a 加 b 的結果 '
11     return a+b
12 add = debug(add)
13 print(add(1, b=2))
14 @debug
15 def add(a, b, c):
16     ' 回傳 a+b+c 的結果 '
17     return a+b+c
18 print(add(1, 2, c=3))
```

```
正在執行函式 add
函式的說明文件爲 回傳 a 加 b 的結果
位置引數 (1,)
關鍵引數 {'b': 2}
3
正在執行函式 add
函式的說明文件爲 回傳 a+b+c 的結果
位置引數 (1, 2)
關鍵引數 {'c': 3}
6
```

# An example for Decorator

```
1   def  print_func_name(func):
2       def  wrap():
3               print("Now  use  function  '{}'".format(func.__name__))
4               func()
5       return  wrap
6
7   def  add():
8           print("add  !!!")
9
10  def  sub():
11          print("substracting!!")
12
13  if  __name__  ==  "__main__":
14          print_func_name(add)()
15          print_func_name(sub)()
16
```

```
Now use function 'add'
add !!!
Now use function 'sub'
substracting!!
```

# Syntax Candy for Decorator

```
1   def  print_func_name(func):
2       def  wrap():
3               print("Now  use  function  '{}'".format(func.__name__))
4               func()
5       return  wrap
6   @print_func_name
7   def  add():
8           print("add  !!!")
9   @print_func_name
10  def  sub():
11          print("substracting!!")
12
13  if  __name__  ==  "__main__":
14          add()
15          sub()
16
```

```
Now use function 'add'
add !!!
Now use function 'sub'
substracting!!
```

# So what's the application?

- Decorator:
  - 除了原有的Function，我們附加於上的新function，也可以在其中新增新功能!!
    - 有點像是繼承舊有的功能，然後可以自訂新功能!! (物件導向會教)

- 所以呢?
  - 例如原本功能沒有很齊全，你要加強功能
  - 例如原本功能你要把輸出做包裝、做修改
  - 例如你不想讓別人知道原本function call是哪一個…
  - 寫寫看範例