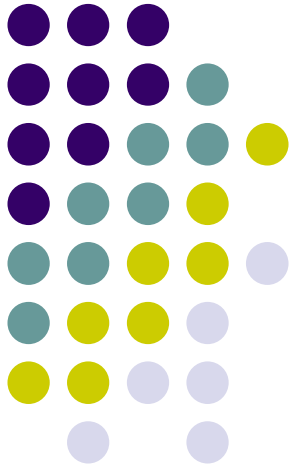
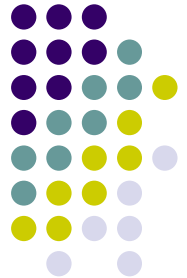
Python程式設計入門
函式(2/4)

葉難



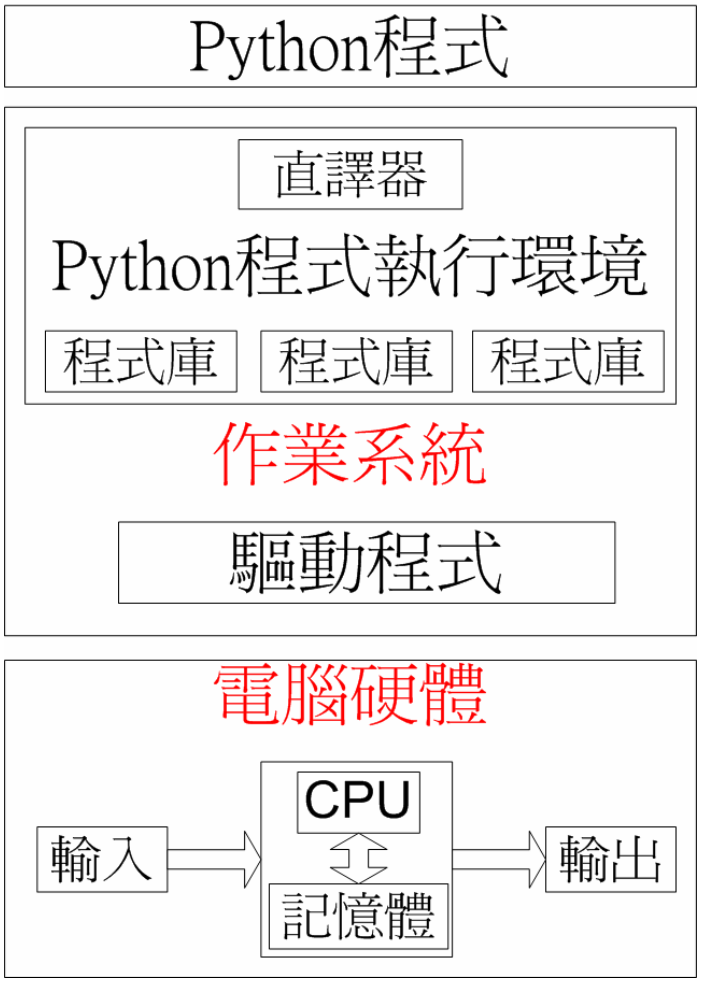
大綱



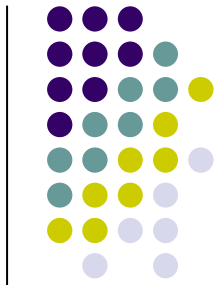
- 函式定義與呼叫，def述句和lambda運算式
- 參數傳遞
- 範圍，命名空間，環境模型
- 遞迴（recursion）
- 高階函式（higher-order function）
- 裝飾器（decorator）
- 產生器（generator）
- 函數式程式設計

抽象模型

- 把下層視爲黑盒子
- 以少數概念規則、解釋底下各種複雜現象與行爲
- 系統在腦袋裡的想像模樣
- Python程式語言的抽象模型：電腦如何執行程式是一回事、你如何理解程式是另一回事
- 因間隔導致上下不一致：錯誤理解

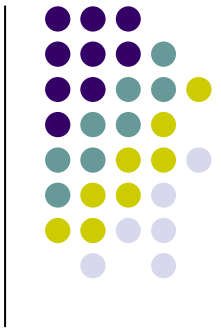


範圍（scope）：LEGB



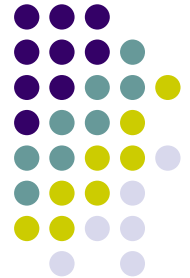
- 順序：
  區域（函式）、
  外圍（閉包）、
  全域（模組）、
  內建
- Local、
  Enclosing、
  Global、
  Builtin

範圍（scope）



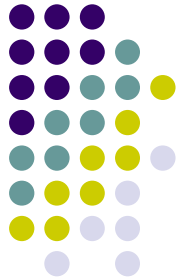
- 物件的存活時間：沒名稱指向它時，變成垃圾，會被系統自動回收
- 第一次指派：名稱在某範圍內第一次出現在指派述句左邊，會在該範圍內建立名稱
- 同名：不同範圍內可存在相同的名稱
- Python採用靜態範圍（static scoping），又稱語彙範圍（lexical scoping）：靜態分析程式碼，便可判斷名稱會屬於哪個範圍

重要範例（釐清基本觀念）



- scope_basic.py
- 規則：
  - 名稱第一次指派的地方，決定該名稱所屬範圍
  - 名稱決議程序，找出「名稱指向哪個物件」的過程，依照順序：區域、外圍、全域、內建範圍
  - 不同範圍的名稱若同名，內層會掩蓋住外一層的
  - 函式的區域範圍，各自獨立

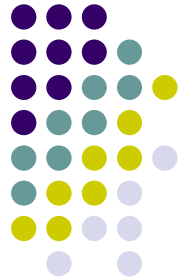
問題（local_foo.py）



- 請問底下程式執行結果爲何？

```
x = 3

def foo():
    x += 10
    print(x)

foo()
print(x)
```

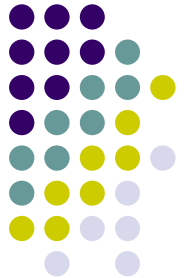
問題（local_bar.py）



- 請問底下程式執行結果爲何？

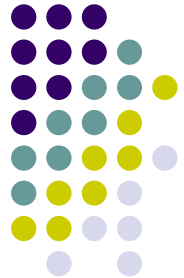
```
y = 100

def bar(x):
    print(y)
    y = x + 10
    return y

bar(3)
```

不好的寫法



```
def f(x):
    global y      # 宣告y爲全域名稱
    y = x

# print(y)        # 若執行會出錯，y尚未定義

f(3)              # 呼叫後，才有y、才指向物件
print(y)          # 印出3
```

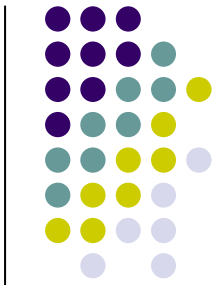
抽象模型



- 工廠模型：函式如同一座工廠
- 環境模型：函式如同設計圖，每次呼叫都會蓋出一座工廠；嗯，還要更複雜...

- 範圍（scope）：靜態概念
- 命名空間（namespace）：動態概念
- 環境（environment）：一串命名空間

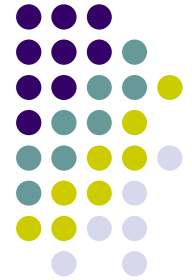
工廠模型



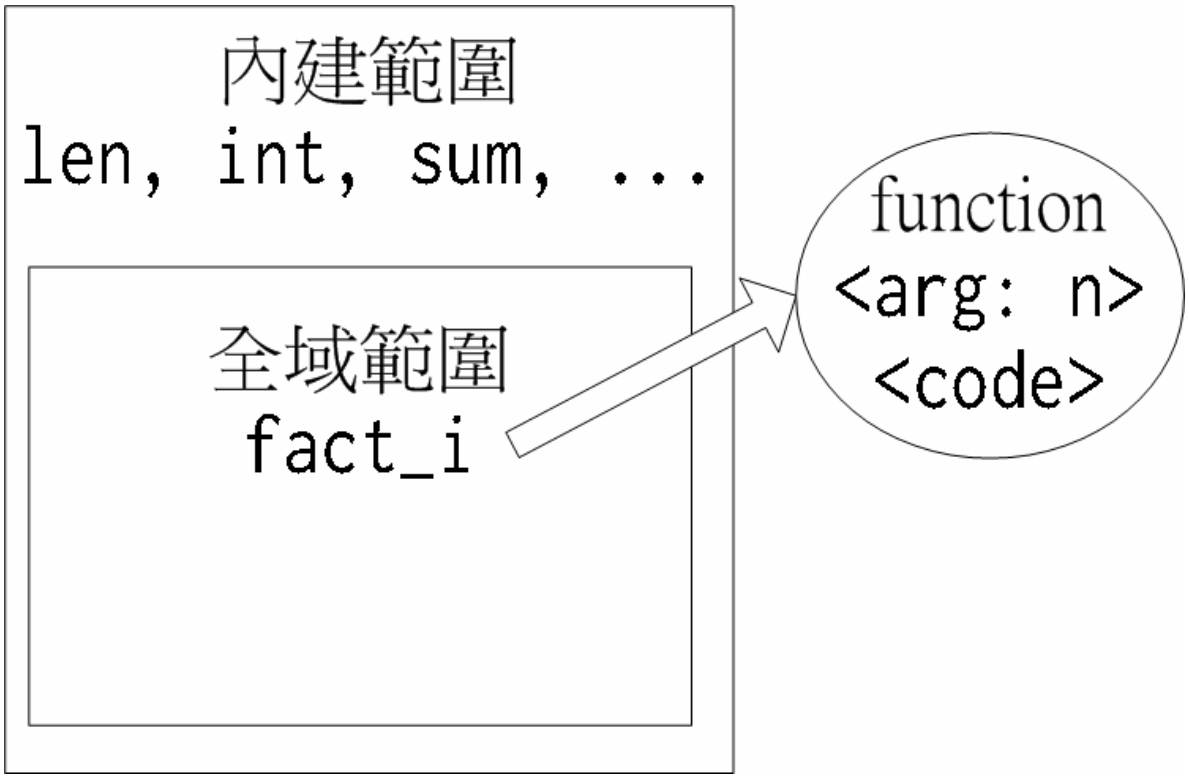
- 階乘（迭代形式）：n! = n * (n-1) * (n-2) ... * 1

```
def fact_i(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

x = fact_i(3)
y = fact_i(7)
```

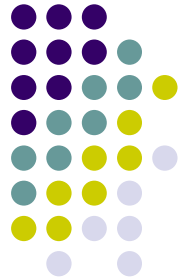
工廠模型想像圖（1/5）



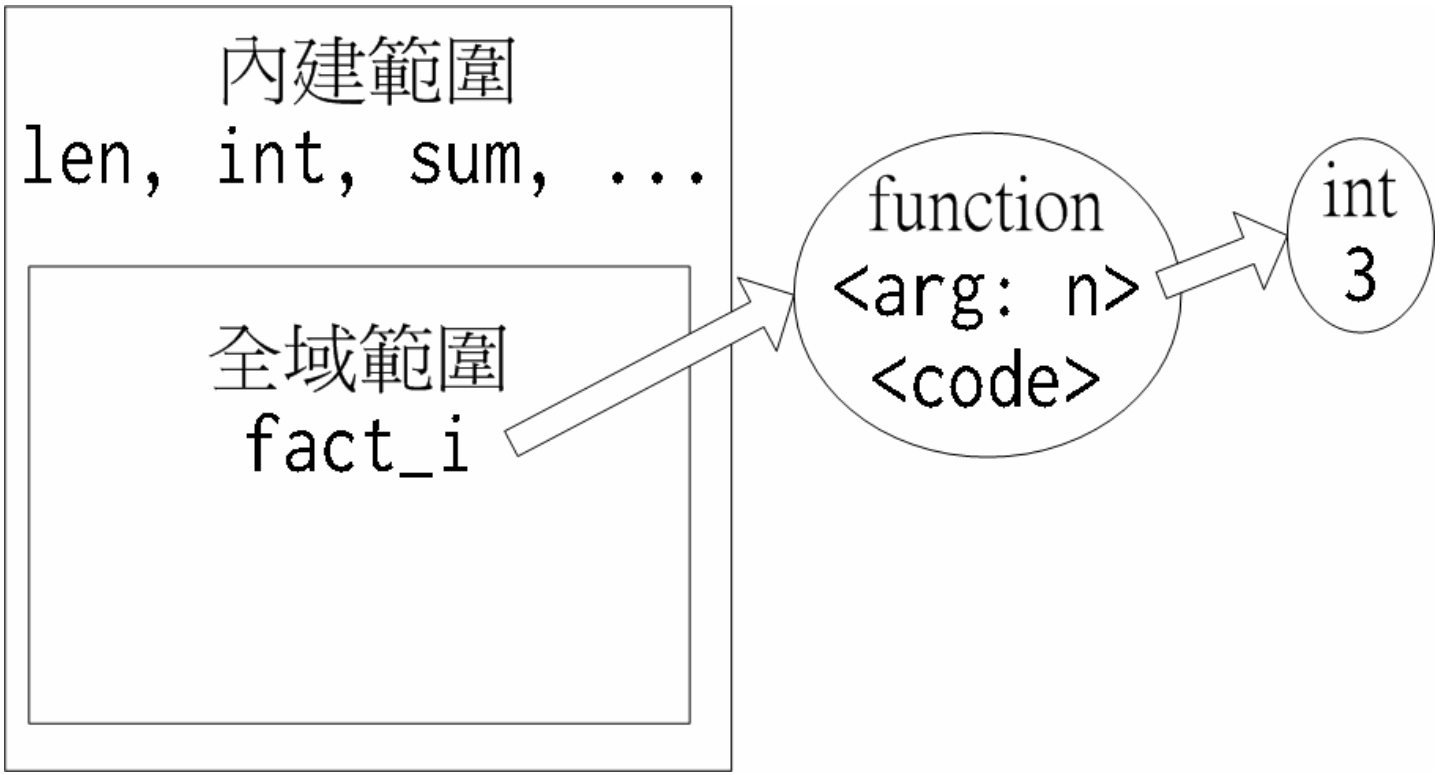
- fact_i函式定義完成後



工廠模型想像圖（2/5）

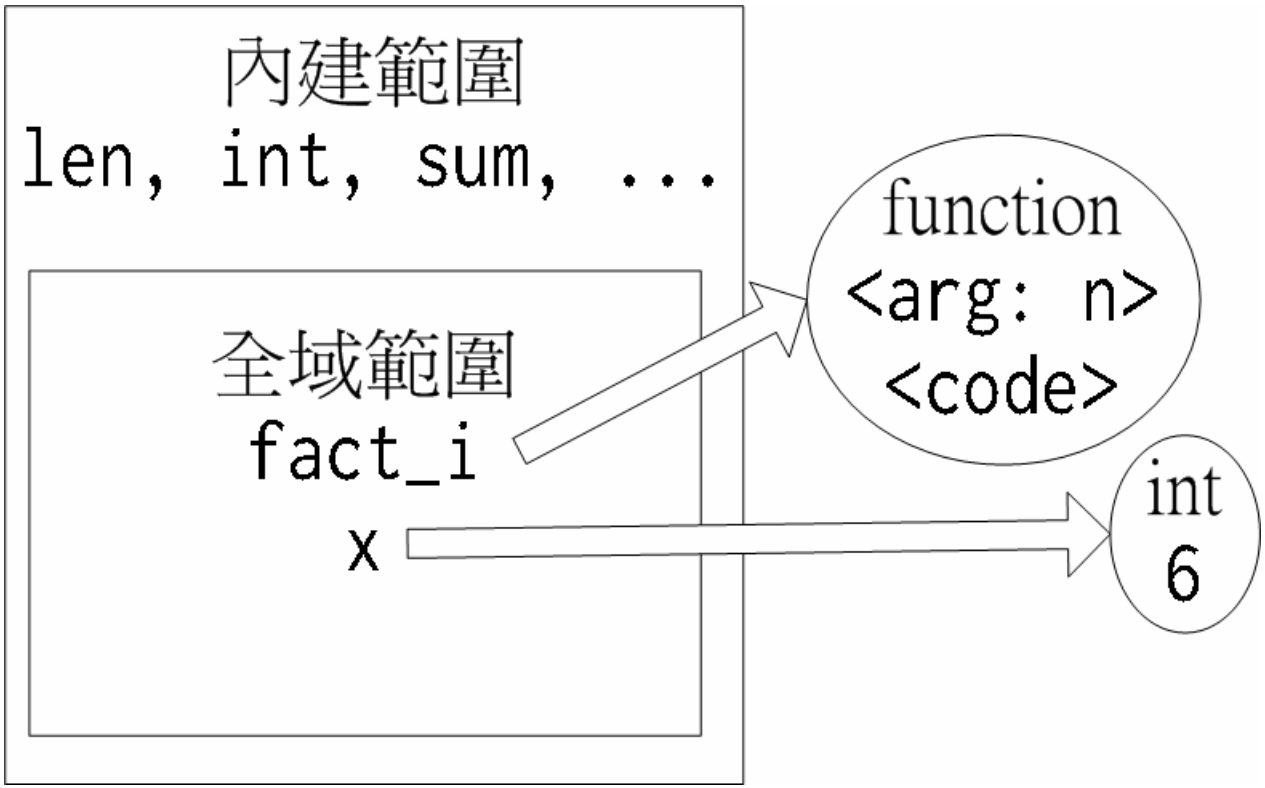


- 執行fact_i(3)的時候



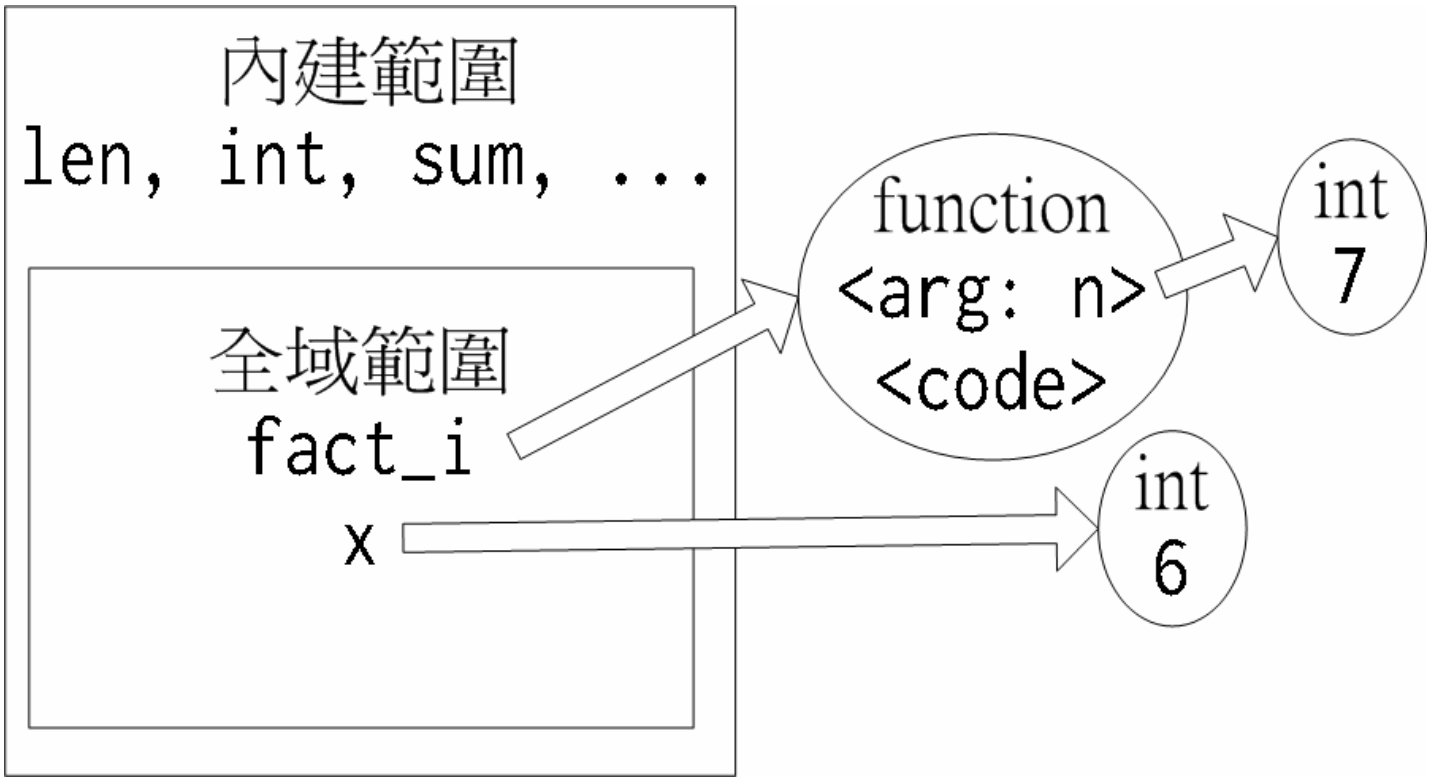
工廠模型想像圖（3/5）

- x = fact_i(3)執行完畢



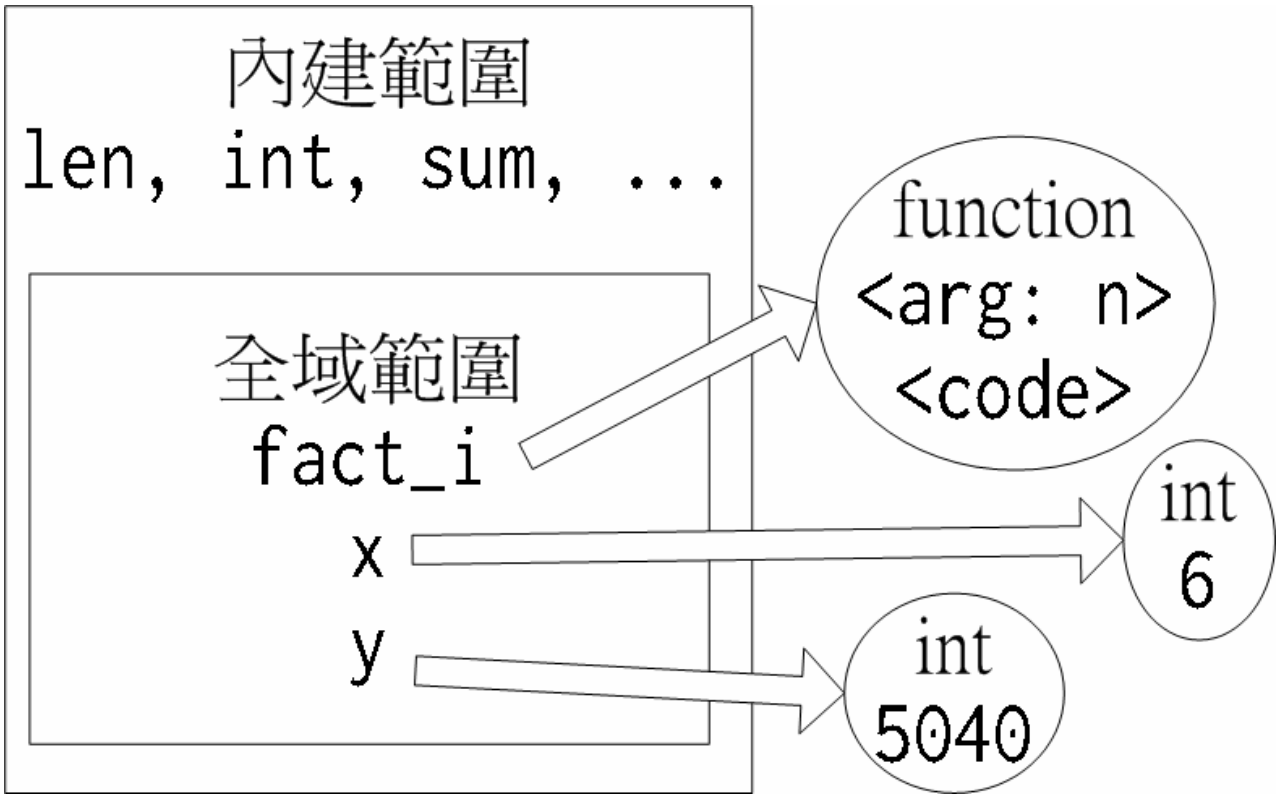
工廠模型想像圖（4/5）

- 執行fact_i(7)的時候

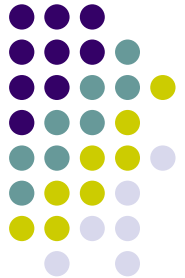


工廠模型想像圖（5/5）

- y = fact_i(7)執行完畢



遞迴



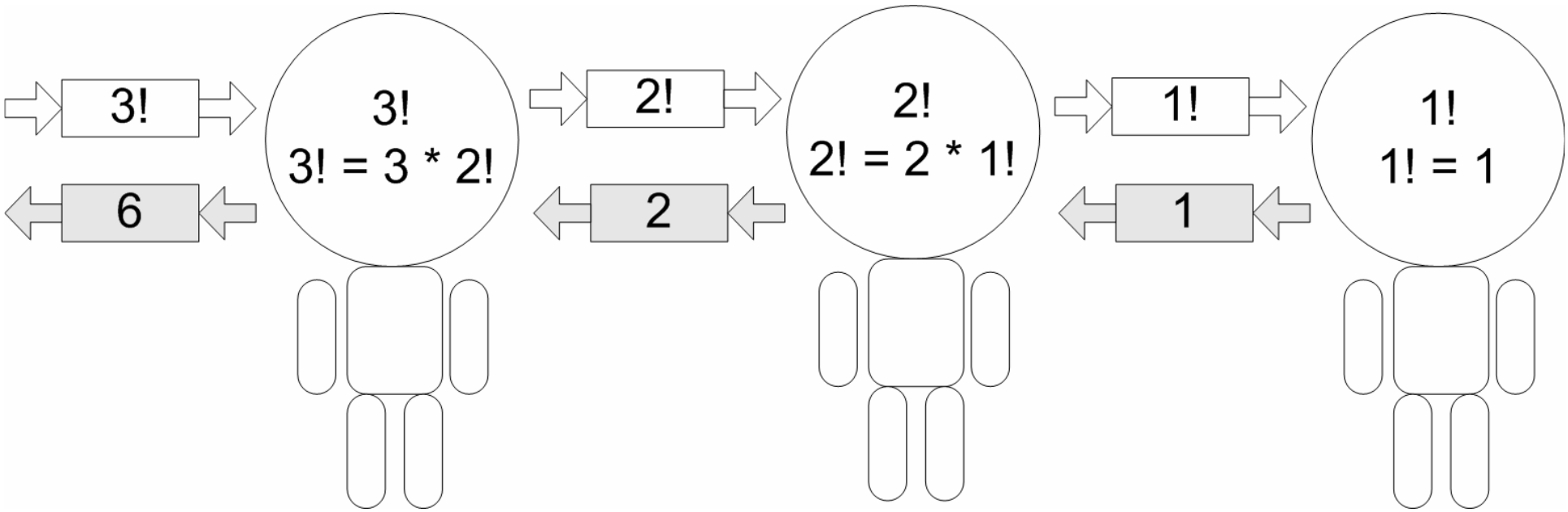
- 階乘定義（遞迴形式）

  n! = n * (n-1)!　　縮減問題

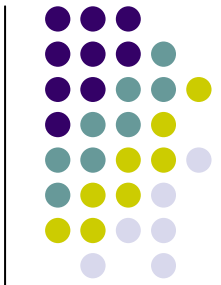
  0! = 1! = 1　　終止條件

```
def fact_r(n):
    if n == 0 or n == 1:              # 終止條件
        return 1
    else:
        return n * fact_r(n-1)        # 縮減問題
```

階乘（遞迴）想像圖

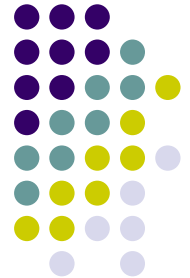


環境模型



- 命名空間（namespace）：可儲存名稱與綁定關係（以及指向的物件）
- 環境（environment）：一串命名空間
  - 有範圍者，執行時會產生出命名空間
  - 串聯適當的上層命名空間，變成新的執行「環境」
  - 在「環境」裡，執行程式

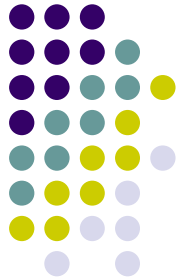
內建與全域（模組）範圍



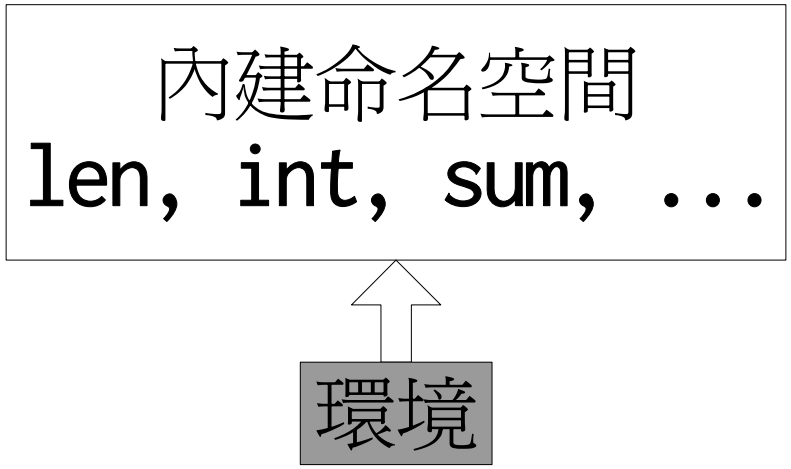
- 匯入模組時，會在新環境裡執行該模組的程式碼

```
                    # 執行程式檔（模組）之前
r = 3               #
import math         # 匯入：第一部分
                    # 匯入：第二部分

x = math.pi * r**2      # 回到此處繼續執行
```

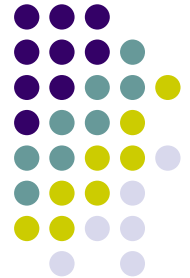
內建與全域（模組）範圍想像圖（1/7）



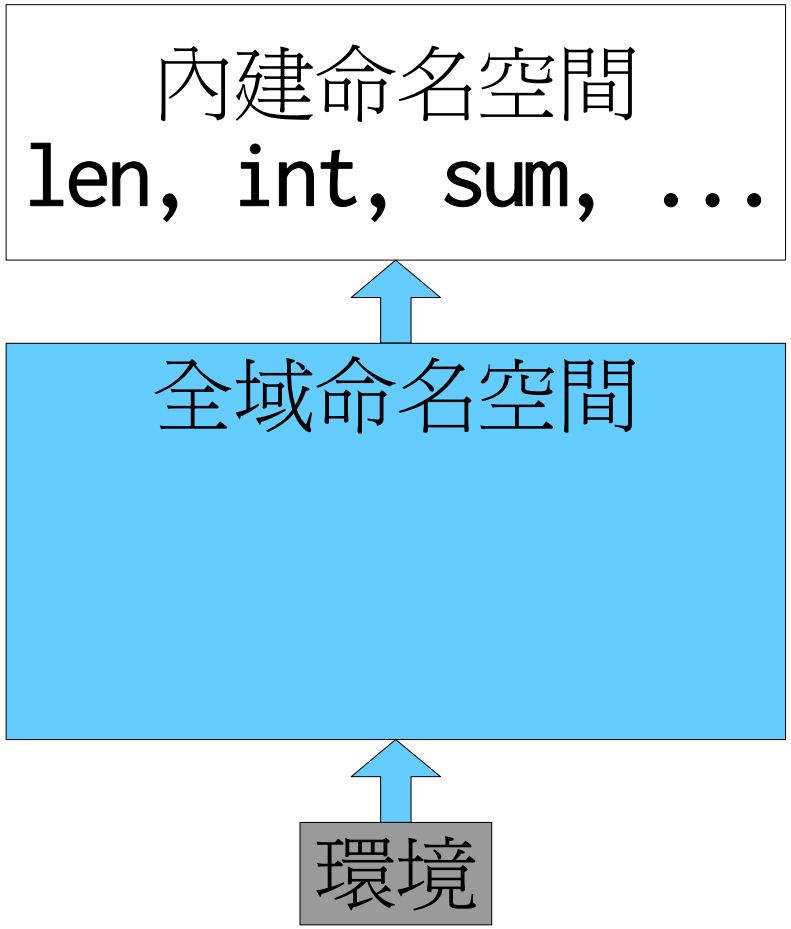
- 執行主程式檔（主模組）之前



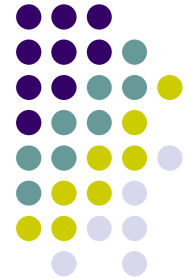
內建與全域（模組）範圍想像圖（2/7）



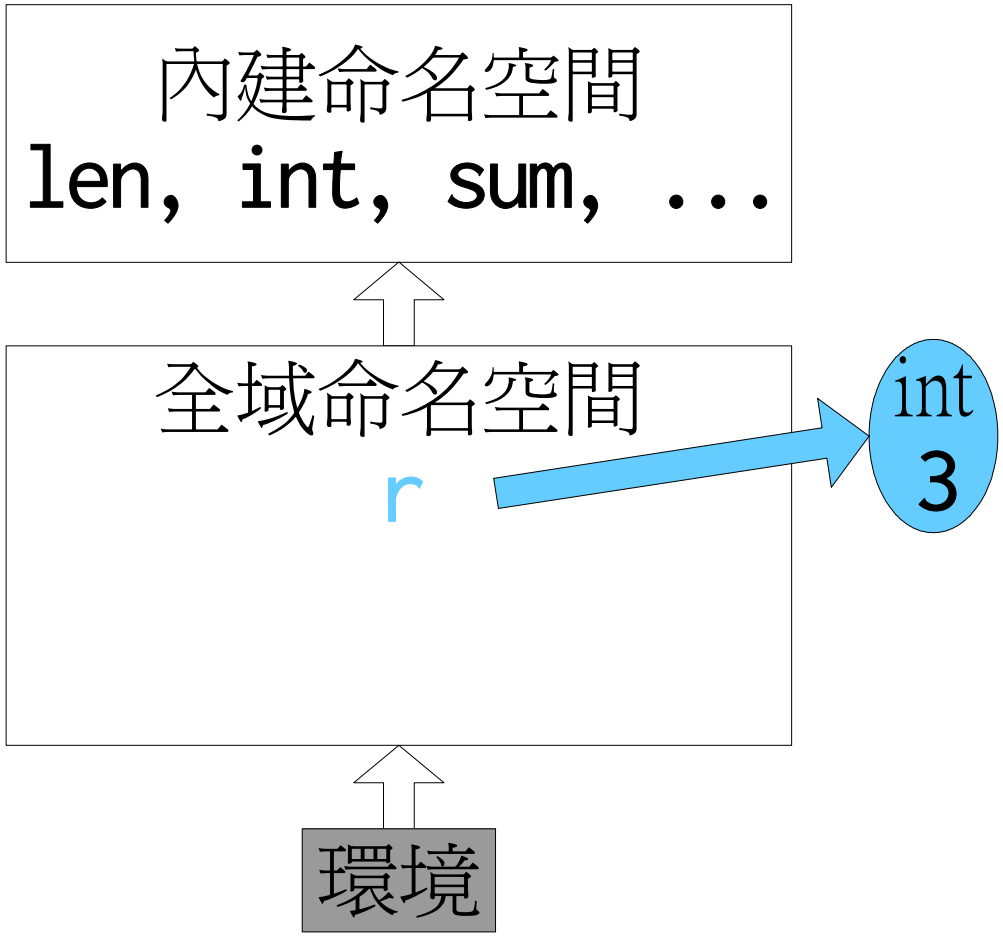
- 執行主程式檔（模組）



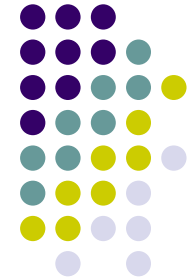
內建與全域（模組）範圍想像圖（3/7）



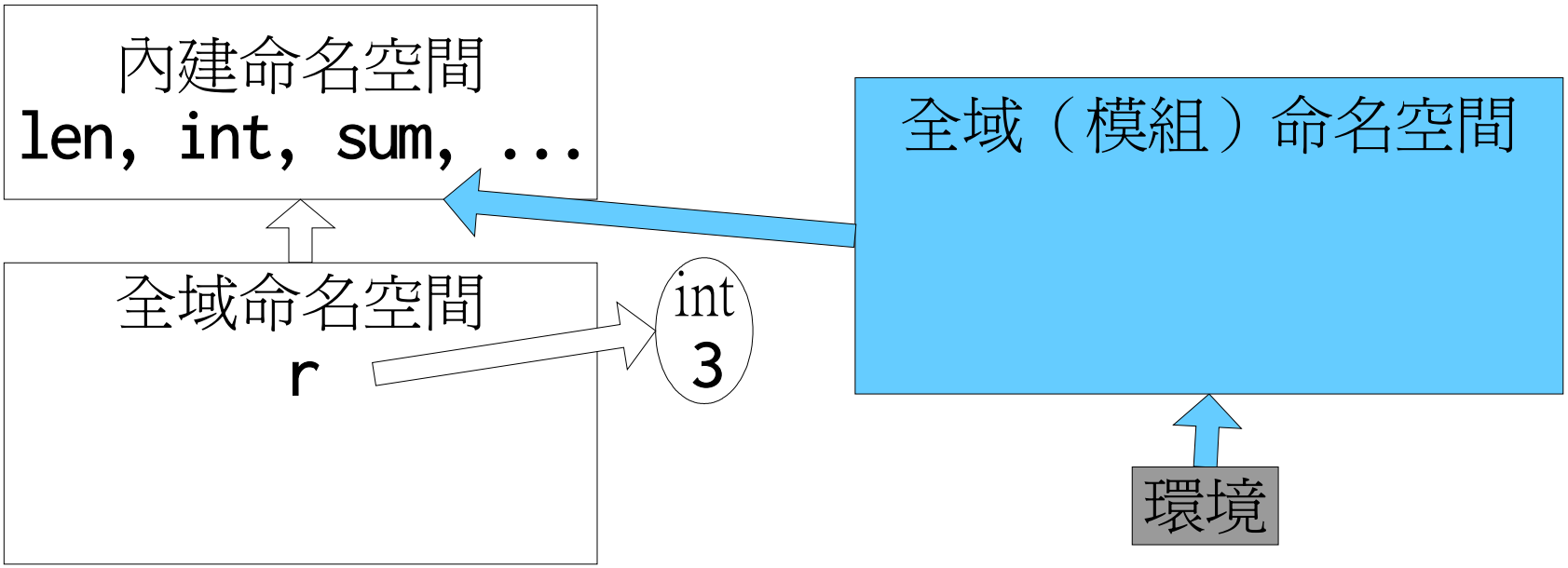
- 執行「`r = 3`」



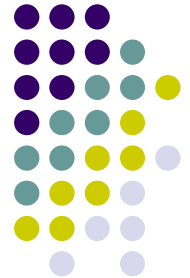
內建與全域（模組）範圍想像圖（4/7）



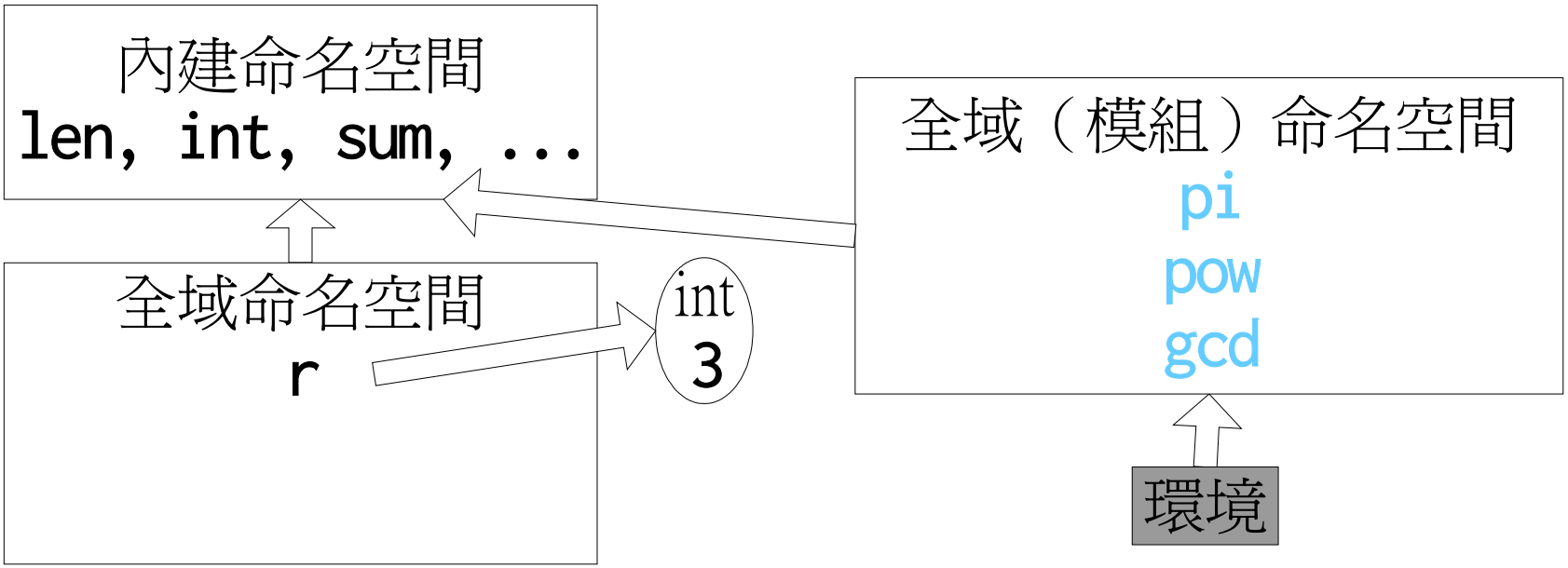
- 「import math」第一部分



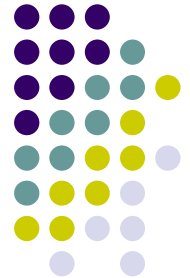
內建與全域（模組）範圍想像圖（5/7）



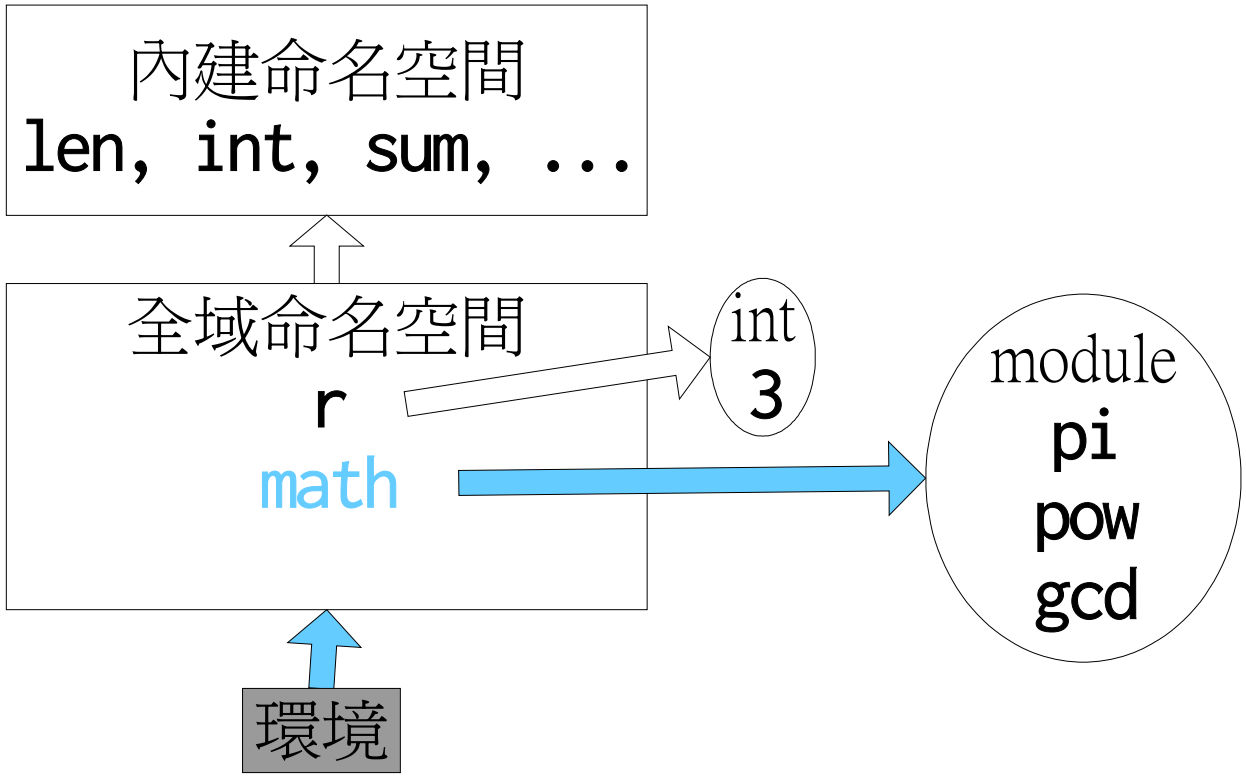
- 「import math」第一部分



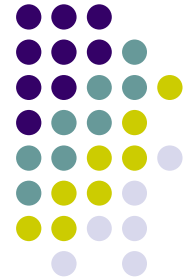
內建與全域（模組）範圍想像圖（6/7）



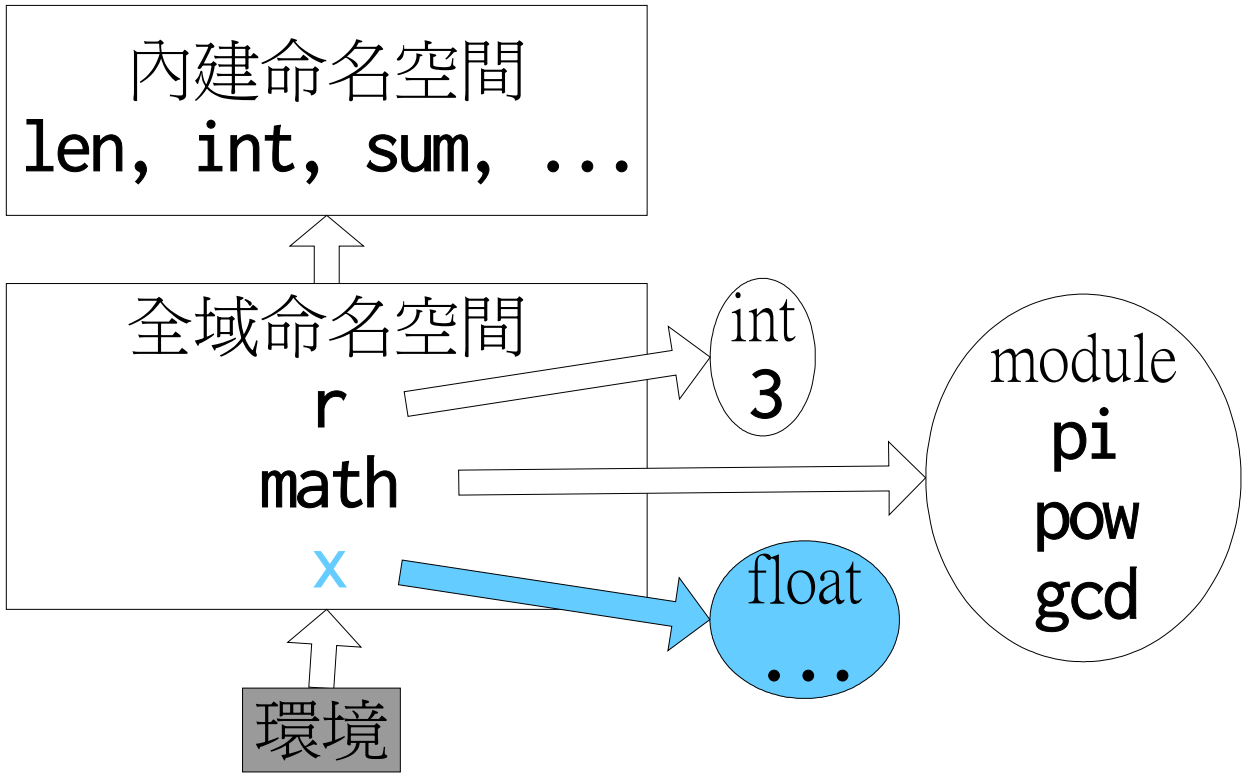
- 「import math」第二部分



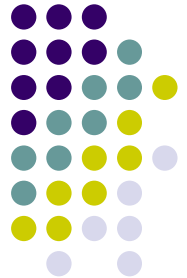
內建與全域（模組）範圍想像圖（7/7）



- 執行「`x = math.pi * r**2`」

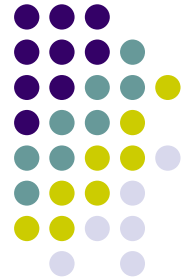


環境模型：函式定義與呼叫



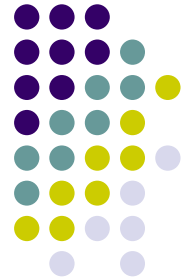
- 命名空間（namespace）：可儲存名稱與綁定關係（以及指向的物件）
- 環境（environment）：一串命名空間
  - 函式定義：建立函式物件，函式物件會記錄定義時所處環境；指派名稱，名稱放在當時環境裡
  - 函式呼叫：建立區域（函式）命名空間，串聯函式物件記錄的環境，成爲新環境，在新環境裡執行函式體

階乘（遞迴形式）

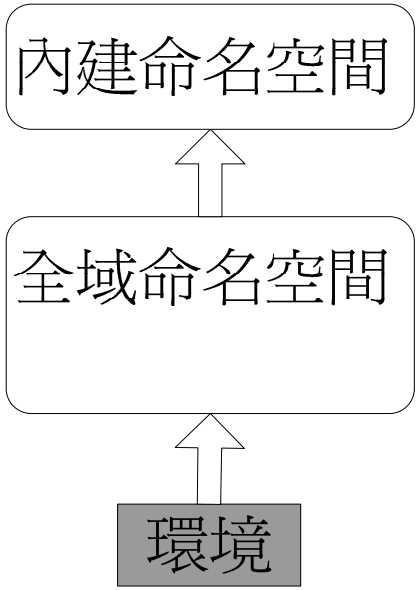


```
def fact_r(n):
    if n == 0 or n == 1:            # 終止條件
        return 1
    else:
        return n * fact_r(n-1)      # 縮減問題

fact_r(3)
```

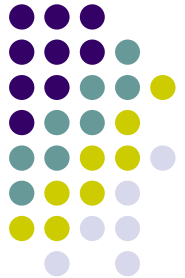
階乗（遡迴）：環境模型（1/6）



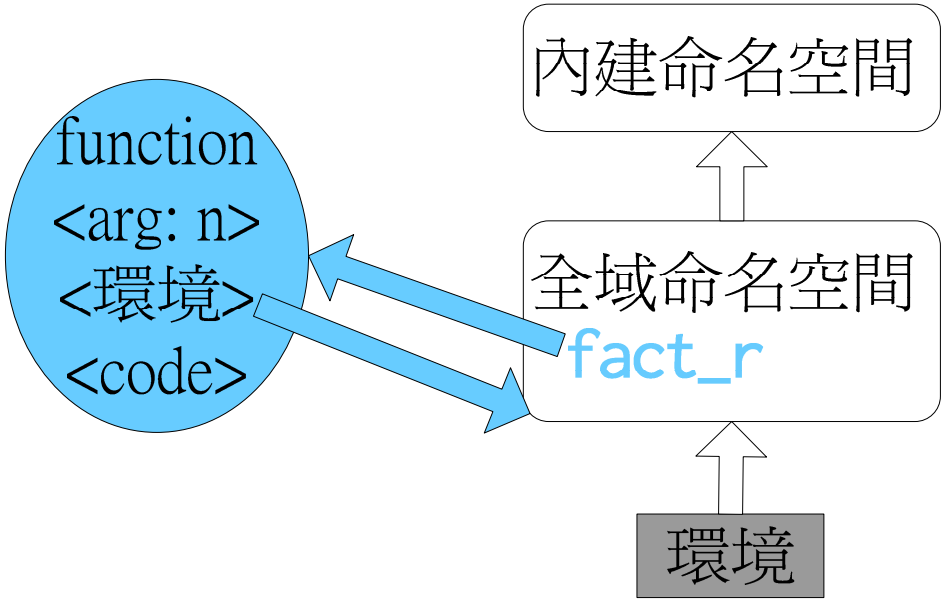
- fact_r定義之前



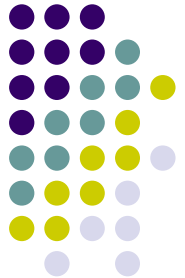
階乘（遞迴）：環境模型（2/6）



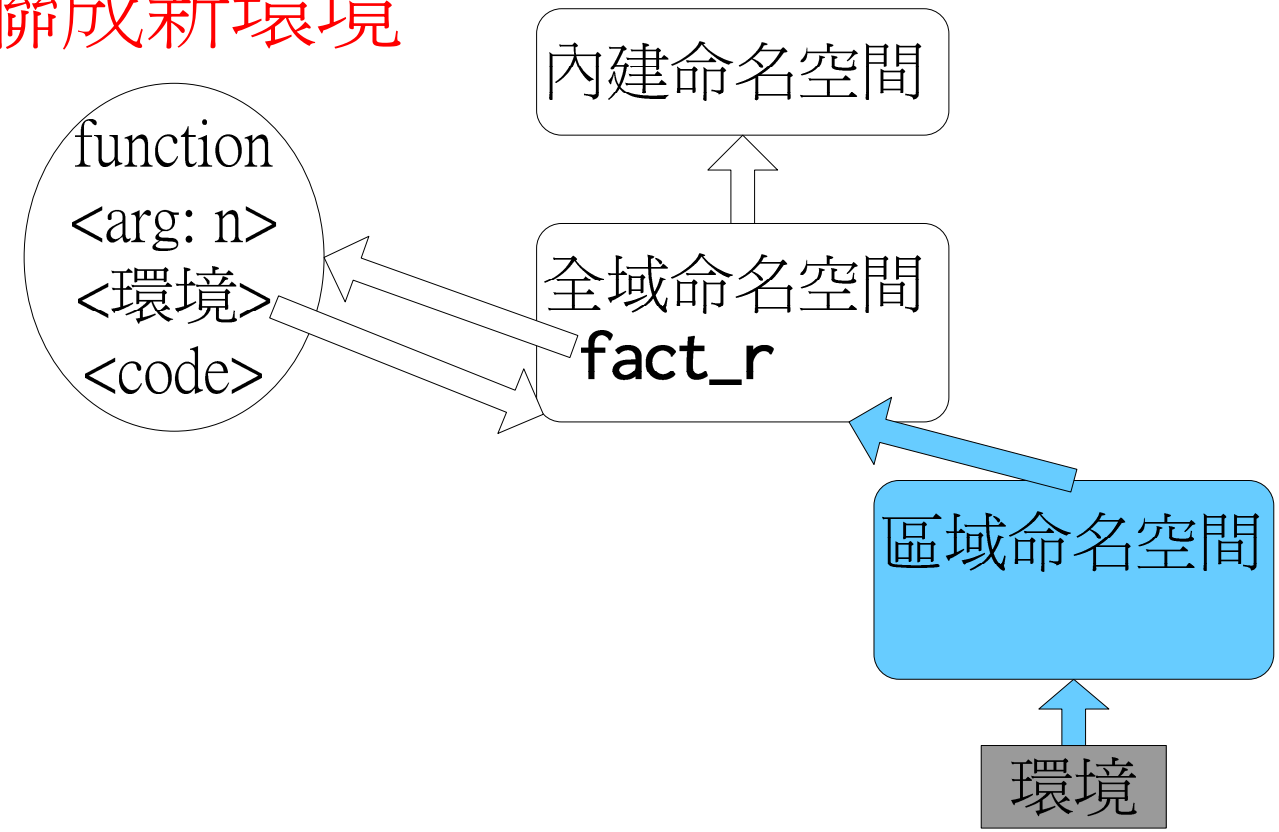
- fact_r函式定義，會記錄當時環境



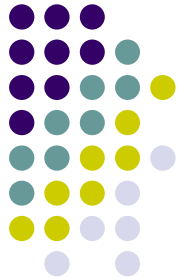
階乘（遞迴）：環境模型（3/6）



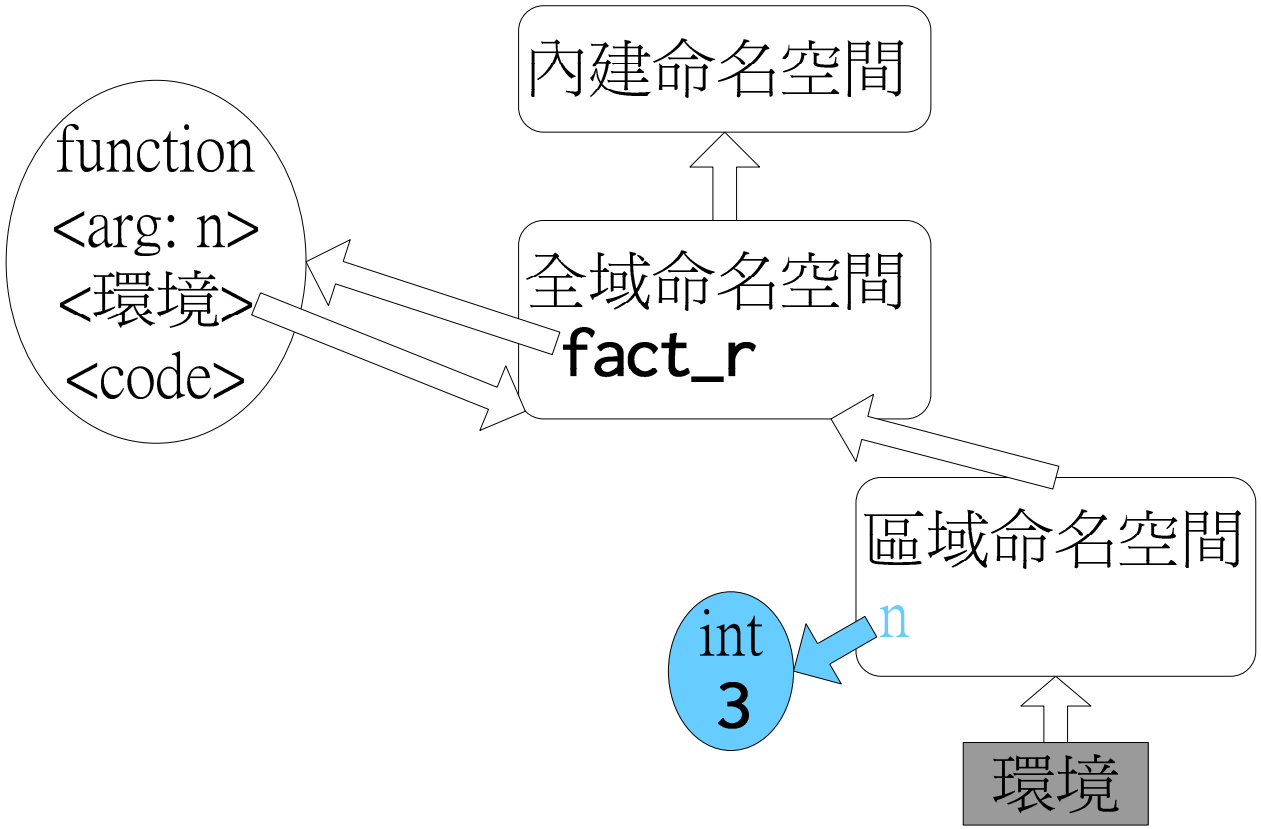
- 呼叫fact_r(3)：建立區域命名空間、串聯成新環境



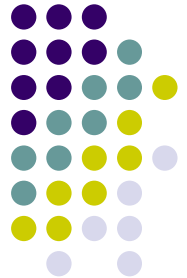
階乘（遞迴）：環境模型（4/6）



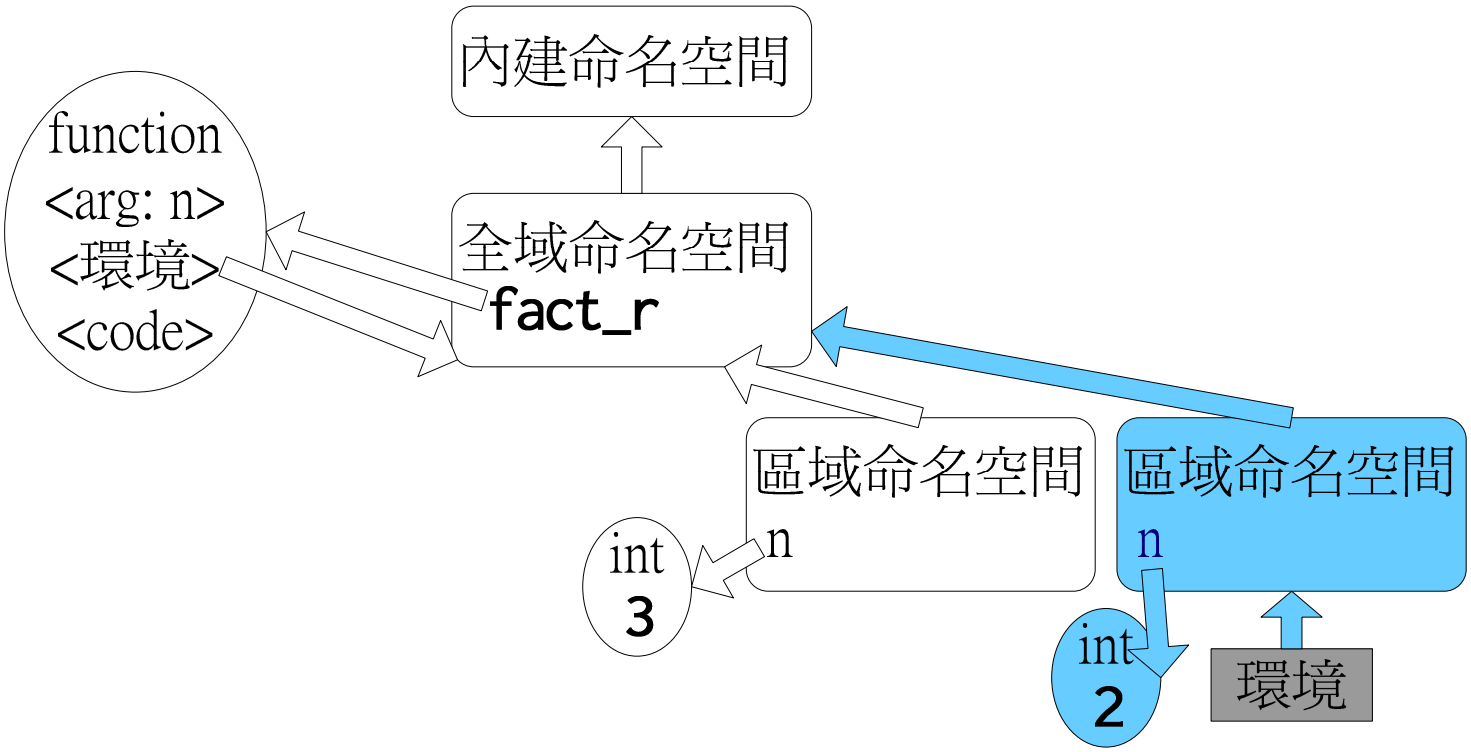
- 呼叫fact_r(3)：在新環境裡執行



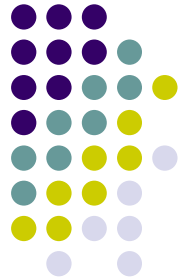
階乘（遞迴）：環境模型（5/6）



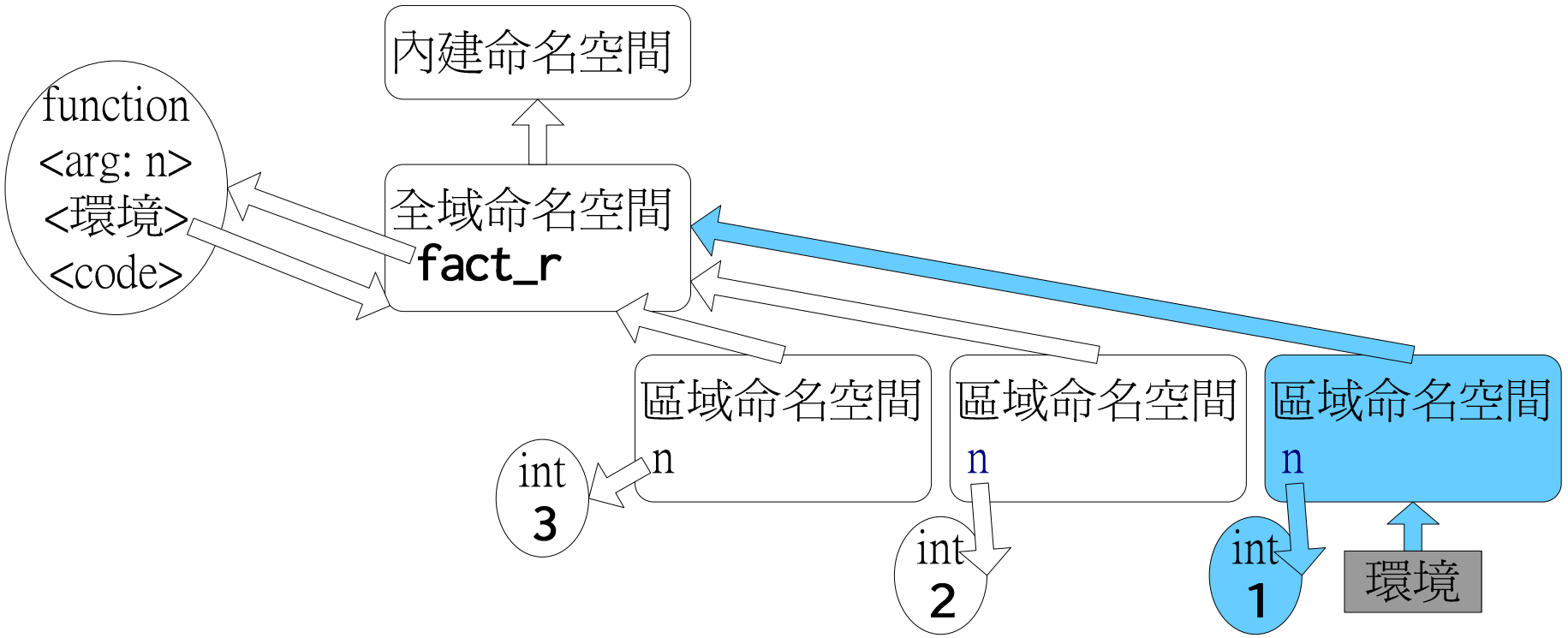
- 呼叫fact_r(2) ：串聯函式記錄的環境



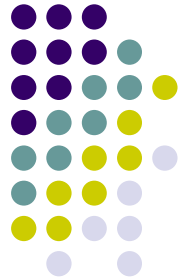
階乘（遞迴）：環境模型（6/6）



- 呼叫fact_r(1)

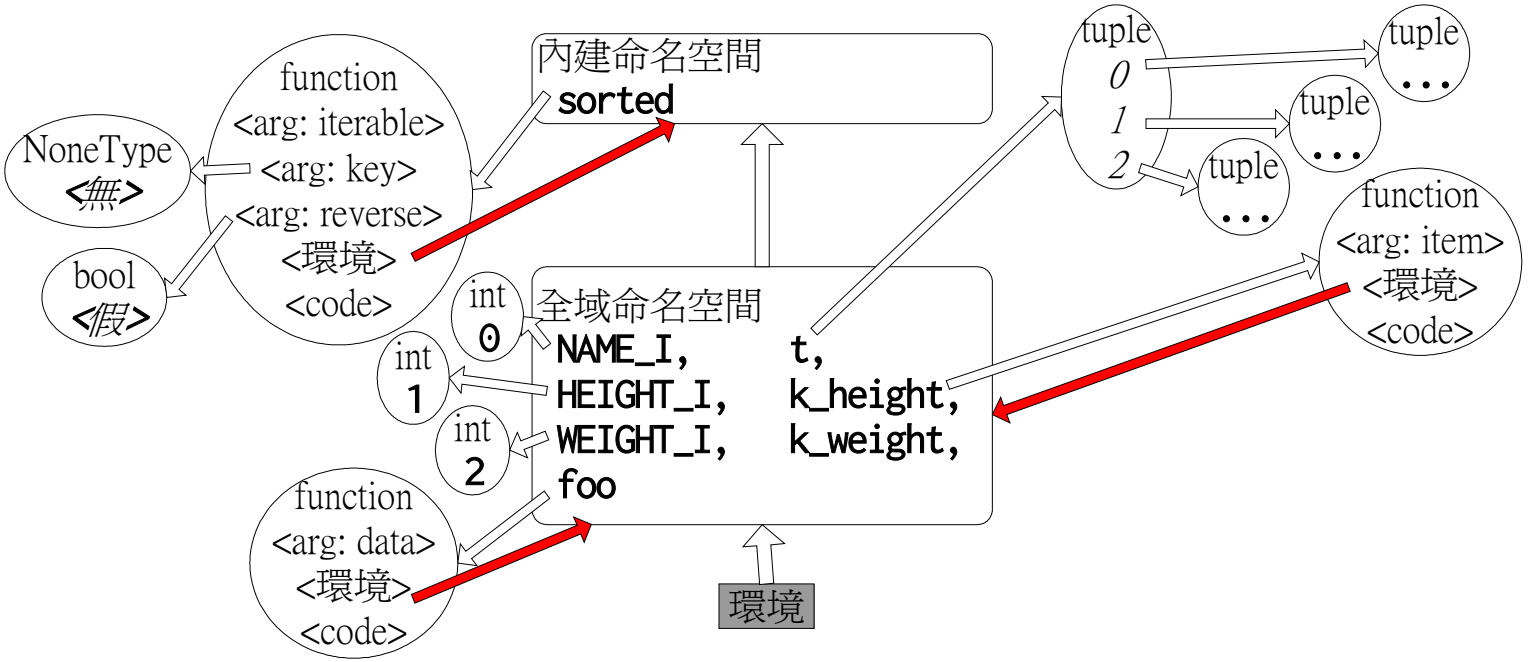


高階函式

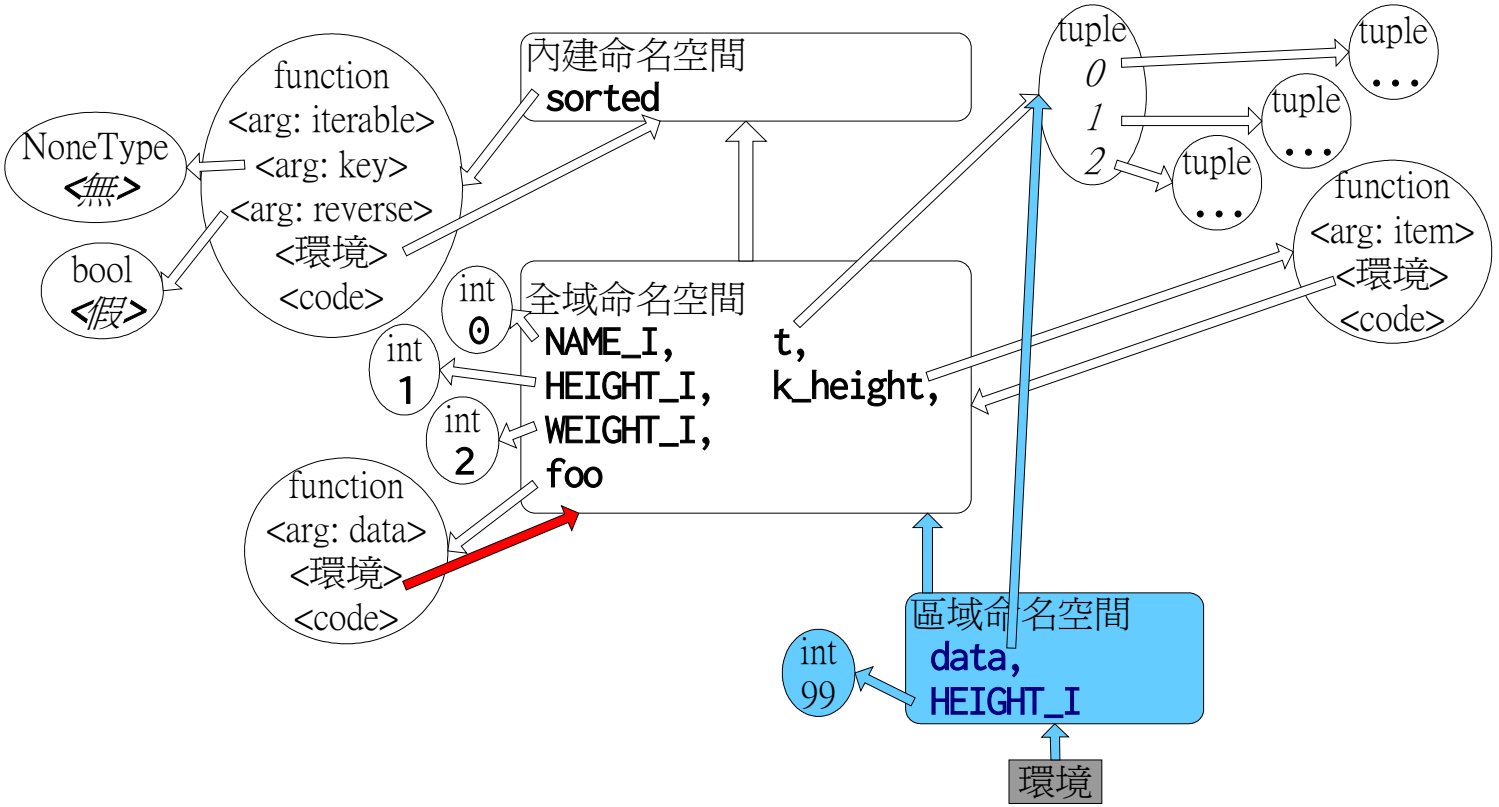


- 函式也是物件，可指派名稱
- 函式作爲參數傳入另一個函式
  範例：scope_sorted.py
- 函式作爲回傳值（函式回傳函式）
  範例：scope_counter.py

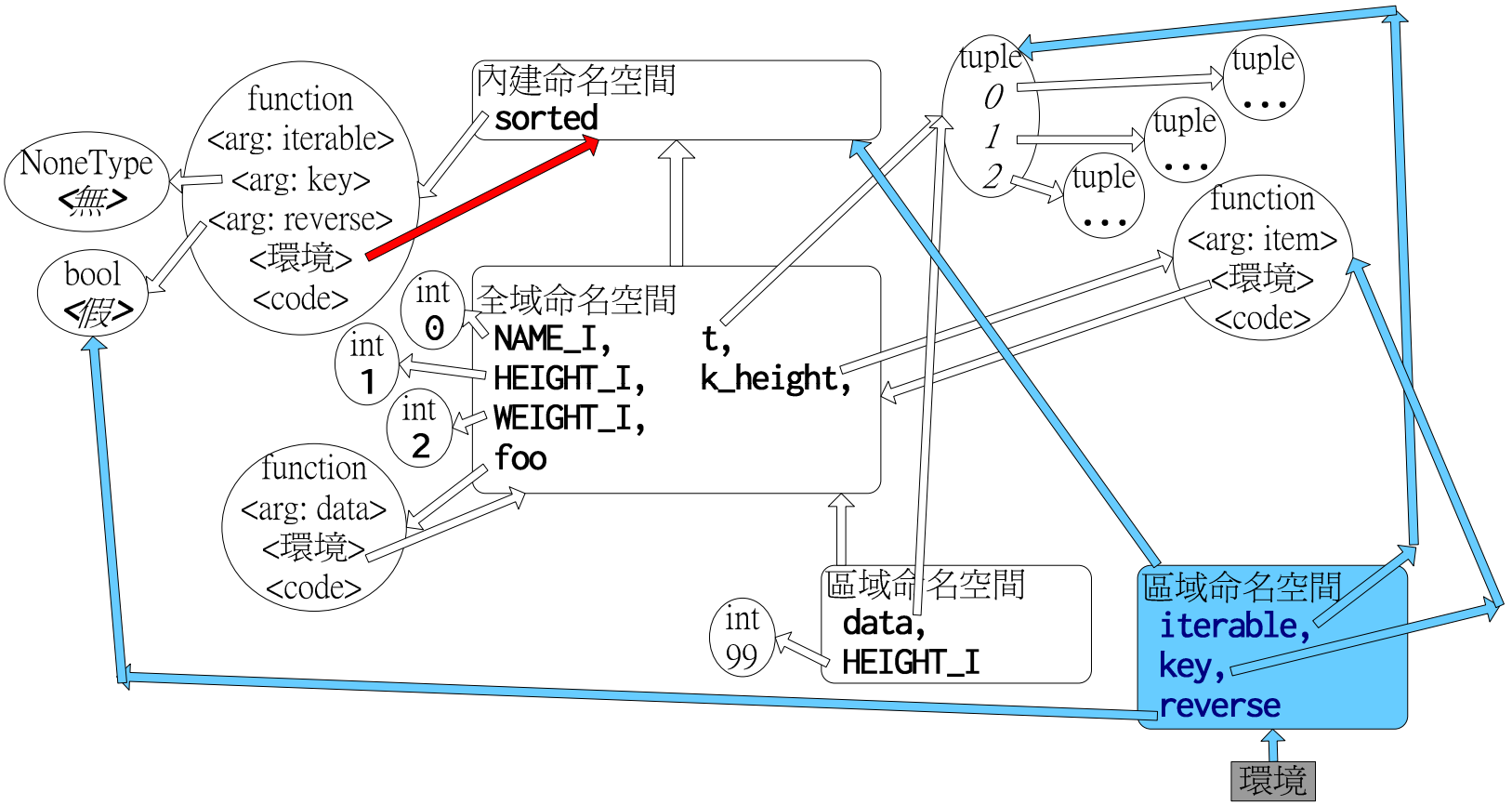
範例：scope_sorted.py（1/4）呼叫foo之前



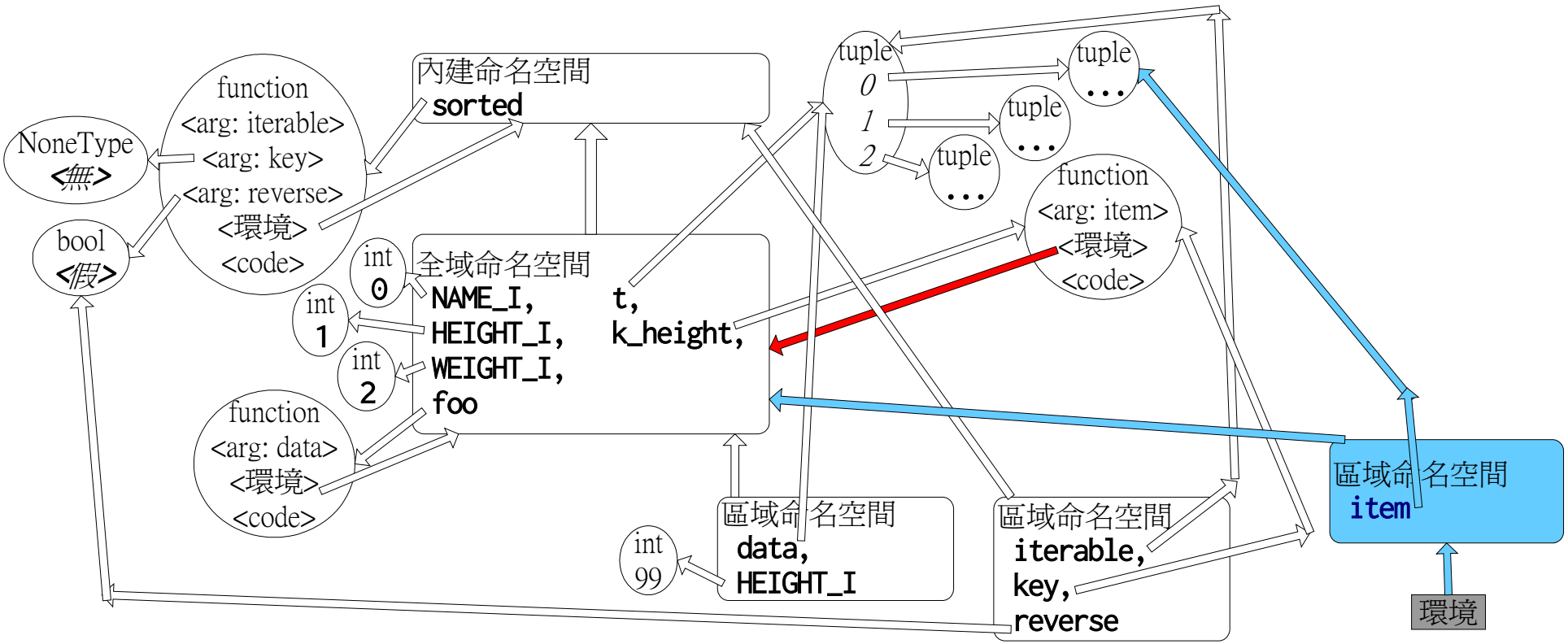
範例：scope_sorted.py（2/4）呼叫foo



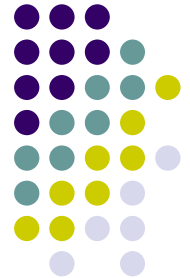
範例：scope_sorted.py（3/4）
foo呼叫sorted



範例：scope_sorted.py（4/4）sorted呼叫key函式



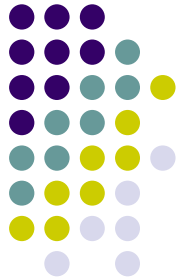
注意：參數預設值若是可變物件



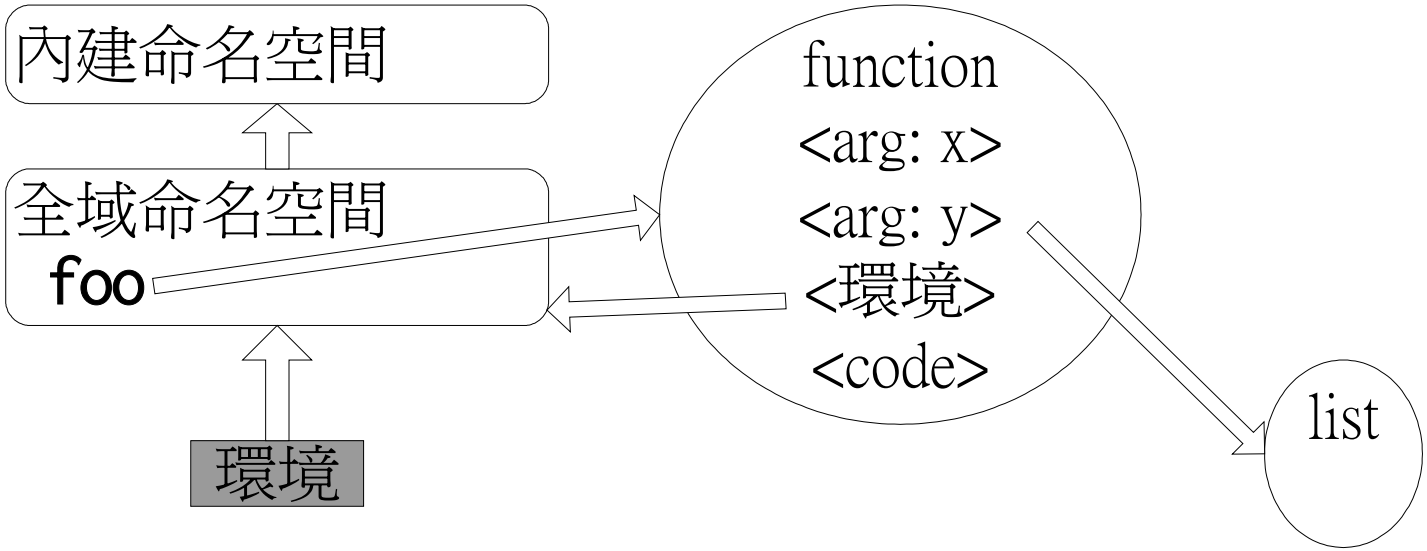
- 參數預設值的運算式只會在定義（def述句）時執行一次，函式物件會記住它

```
def foo(x, y=[]):    # 注意
    y.append(x)
    print(y)

foo(1)   # 印出什麼？
foo(2)   # 印出什麼？
```

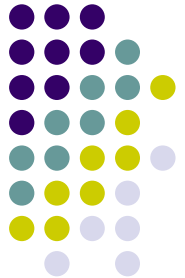
參數預設值若是可變物件（1/4）



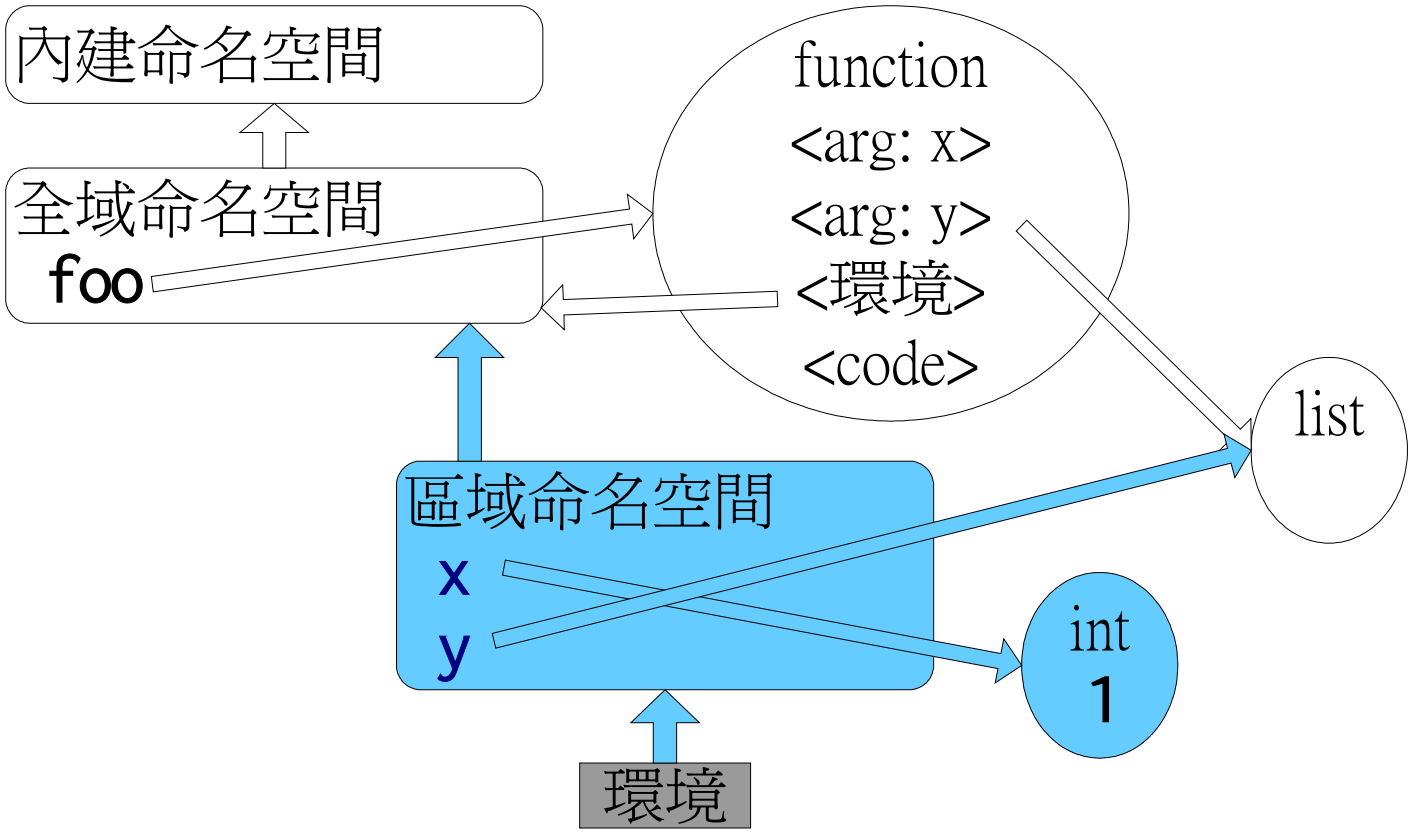
- def述句：foo函式定義完成之後



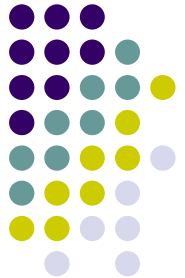
參數預設值若是可變物件（2/4）



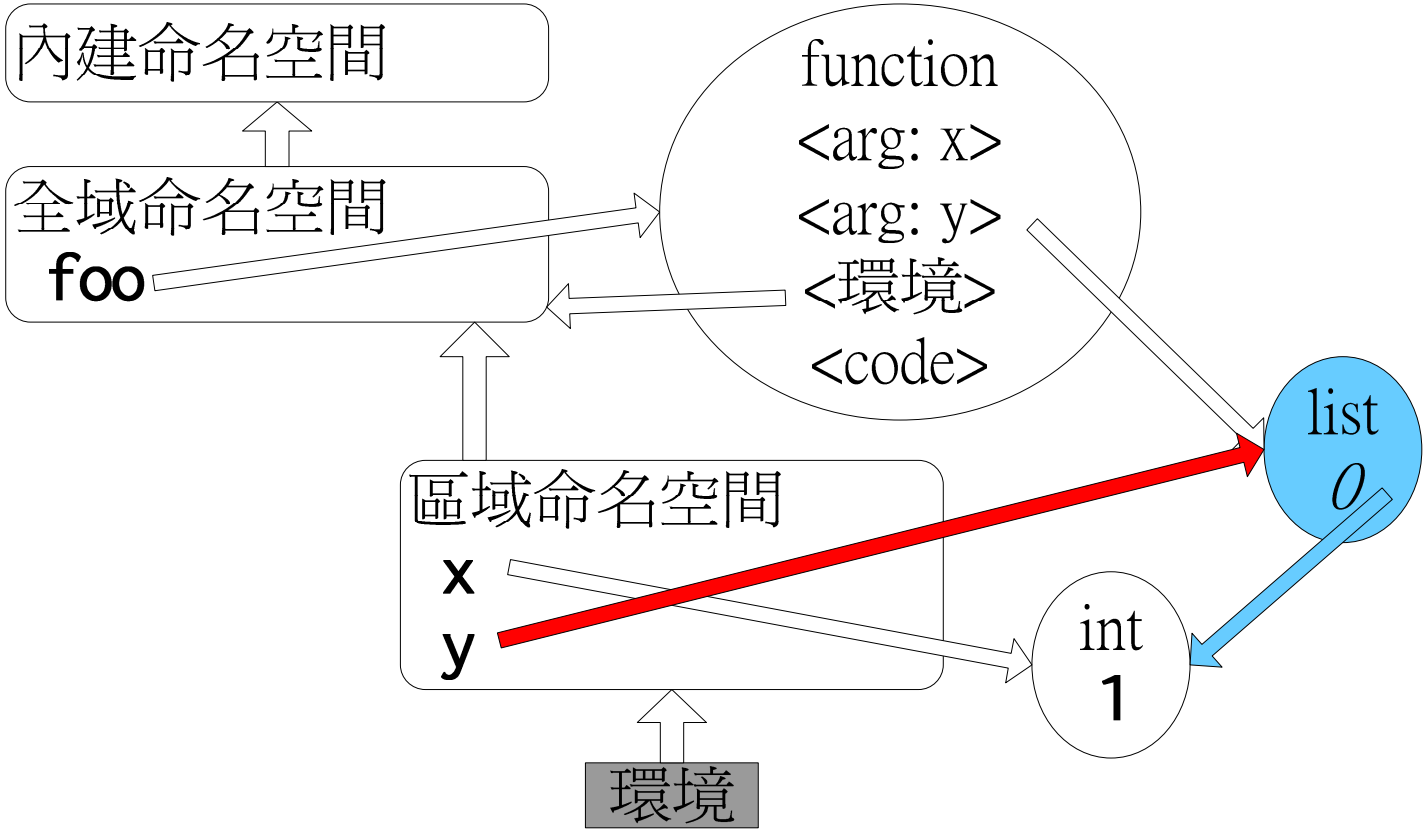
- 呼叫foo(1)



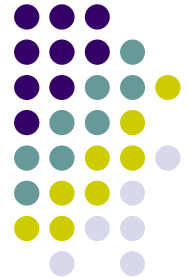
參數預設值若是可變物件（3/4）



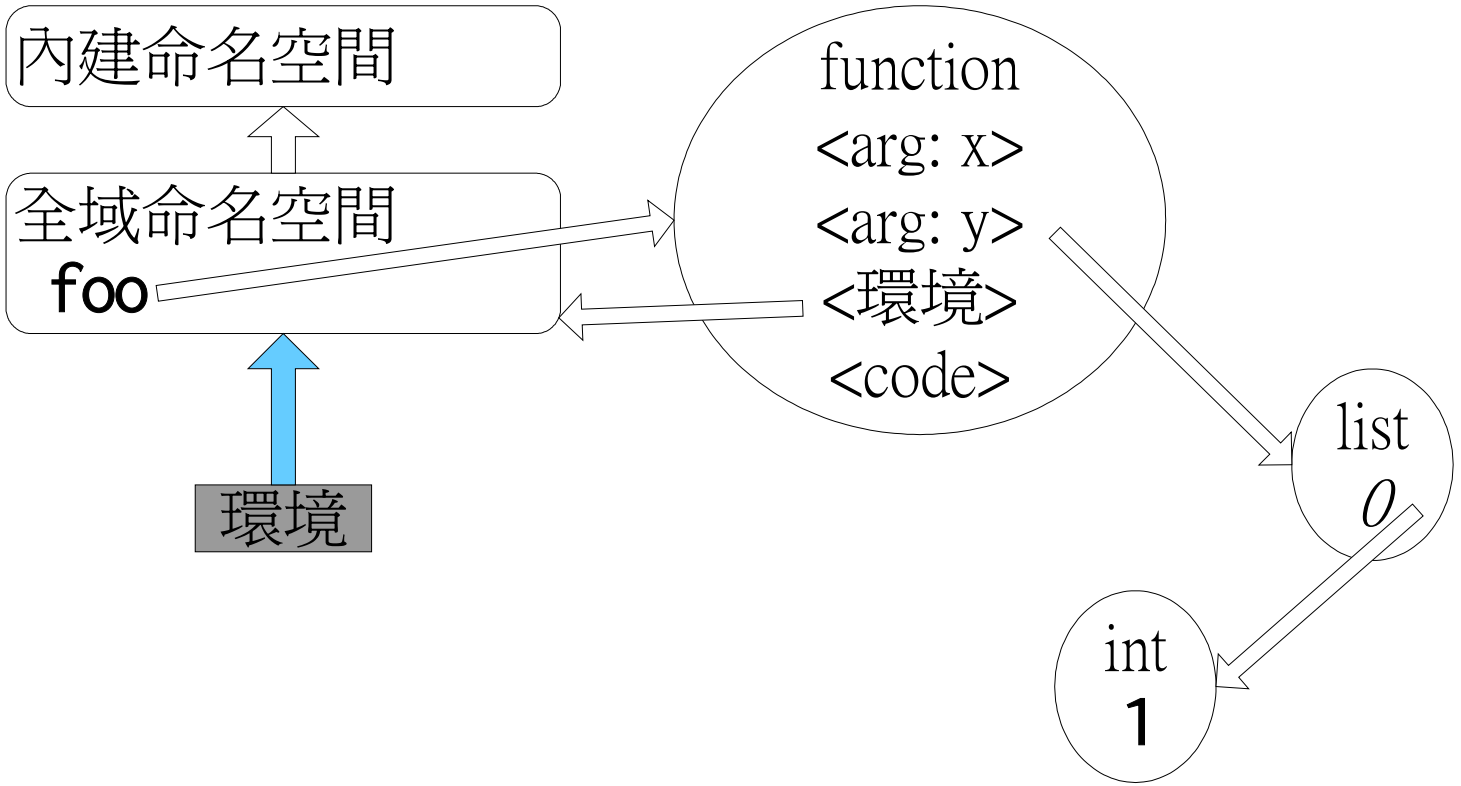
- 執行y.append(x)



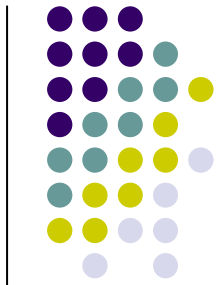
參數預設值若是可變物件（4/4）



- foo(1)執行完畢，呼叫foo(2)之前



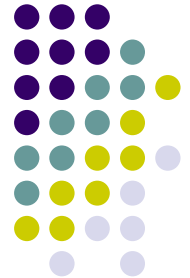
計數器counter（scope_counter.py）



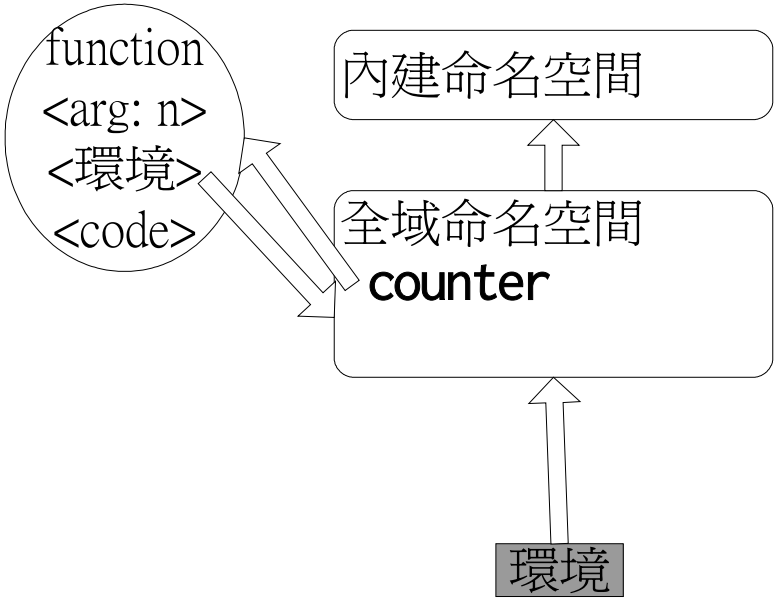
- 高階函式：函式回傳函式
- 外圍範圍（enclosing scope）

```
def counter(n):
    li = [n]
    def bar(x):
        li[0] += x              # 在bar裡，原地修改counter的li
        return li[0]
    return bar                  # counter回傳「函式」

c0 = counter(0)                 # 名稱c0與c100都指向函式物件
c100 = counter(100)

print(c0(1))            # 印出1
print(c100(10))         # 印出110
print(c0(1))            # 印出2
print(c0(3))            # 印出5
print(c100(20))         # 印出130
```

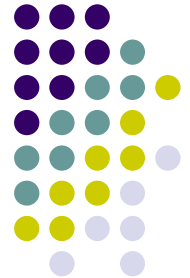
計數器counter示意圖（1/9）



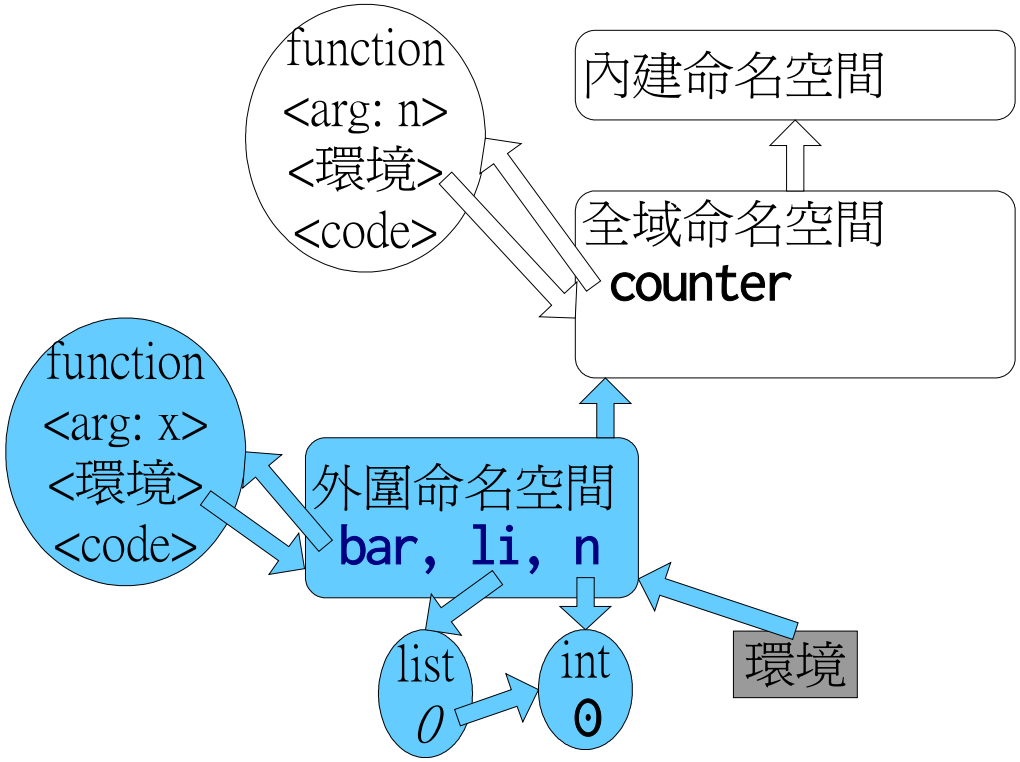
- counter函式定義



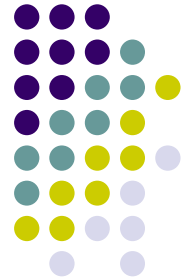
計數器counter示意圖（2/9）



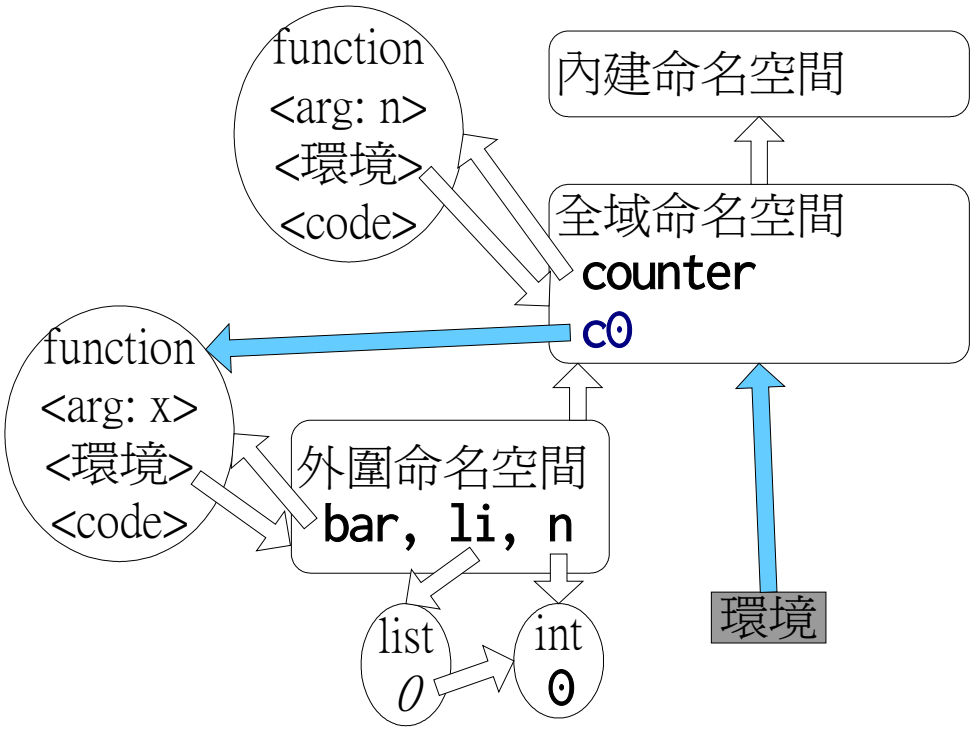
- 呼叫counter(0)



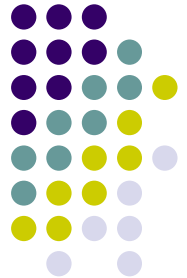
計數器counter示意圖（3/9）



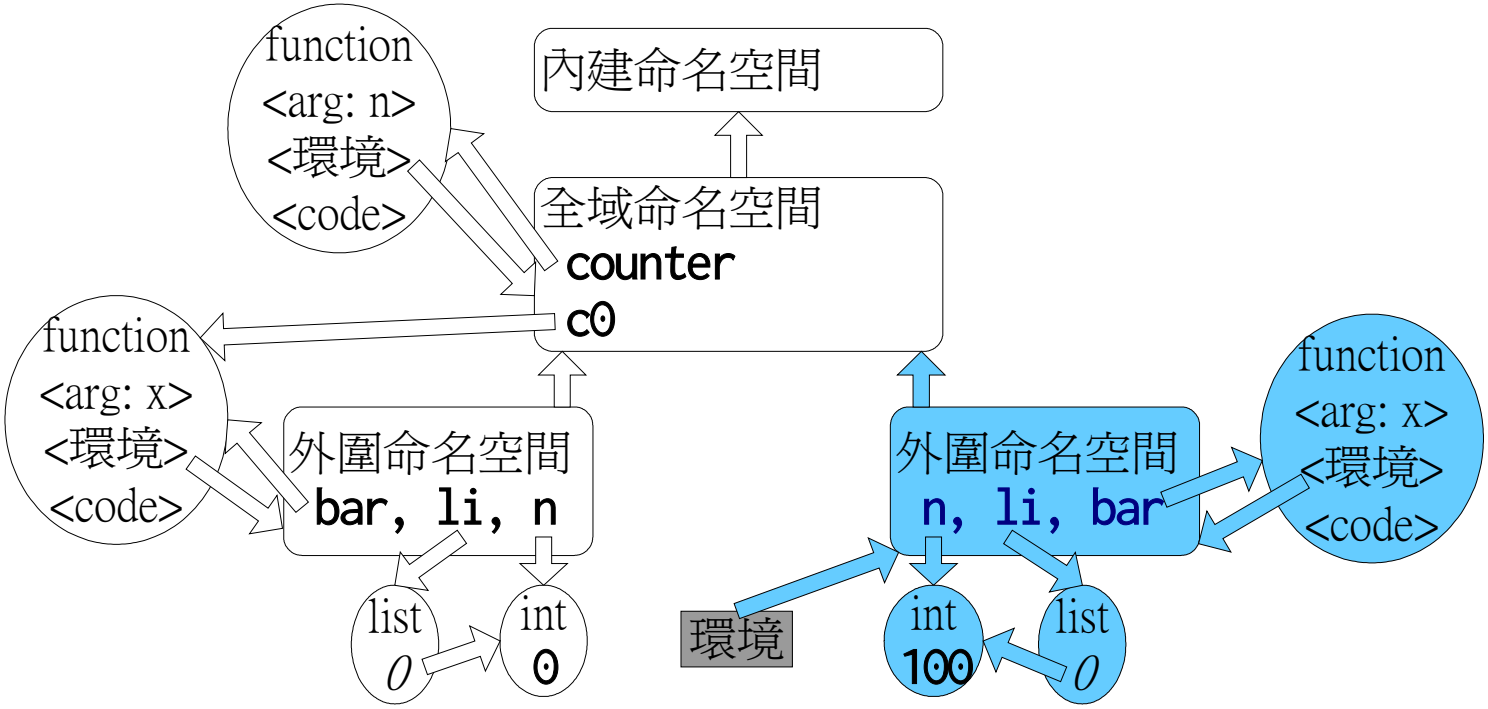
- c0 = counter(0)執行後



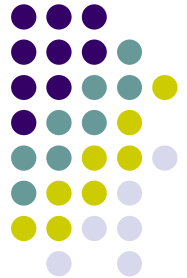
計數器counter示意圖（4/9）



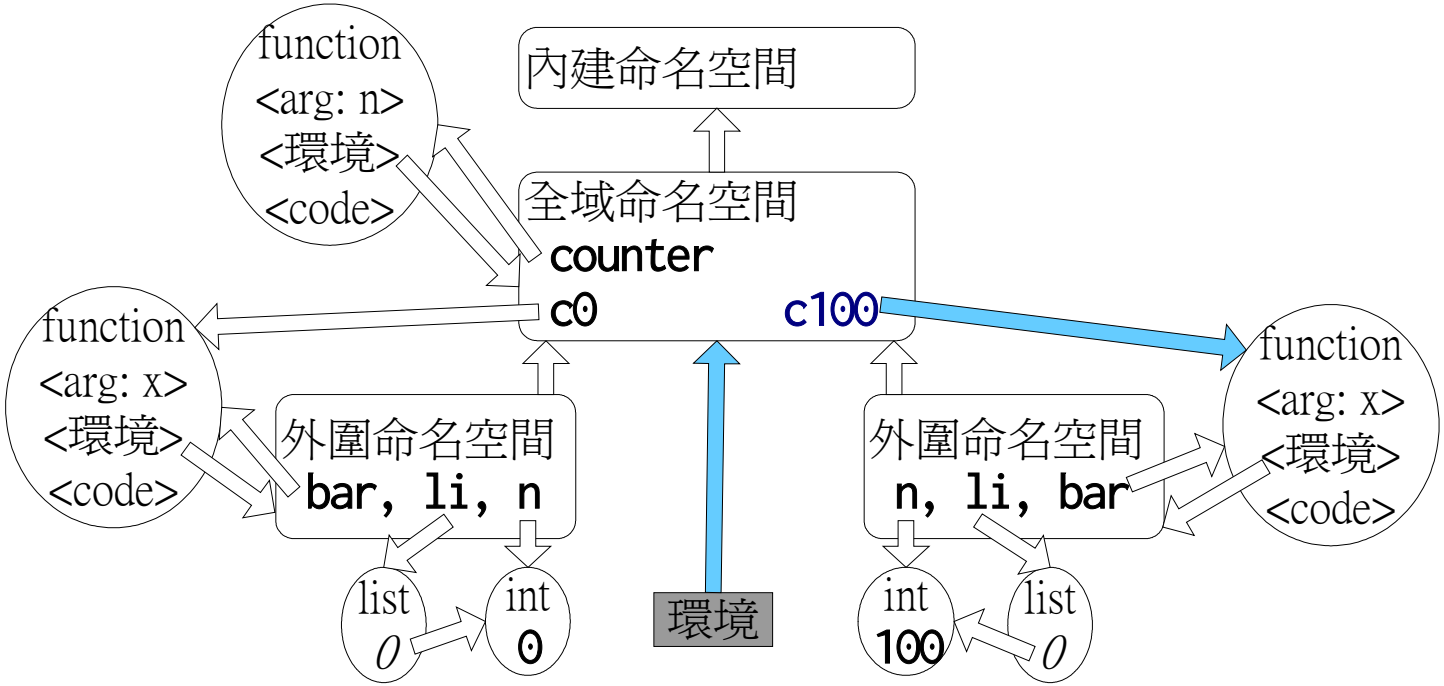
- 呼叫counter(100)



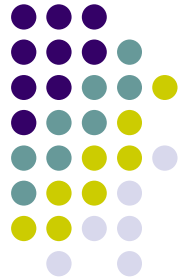
計數器counter示意圖（5/9）



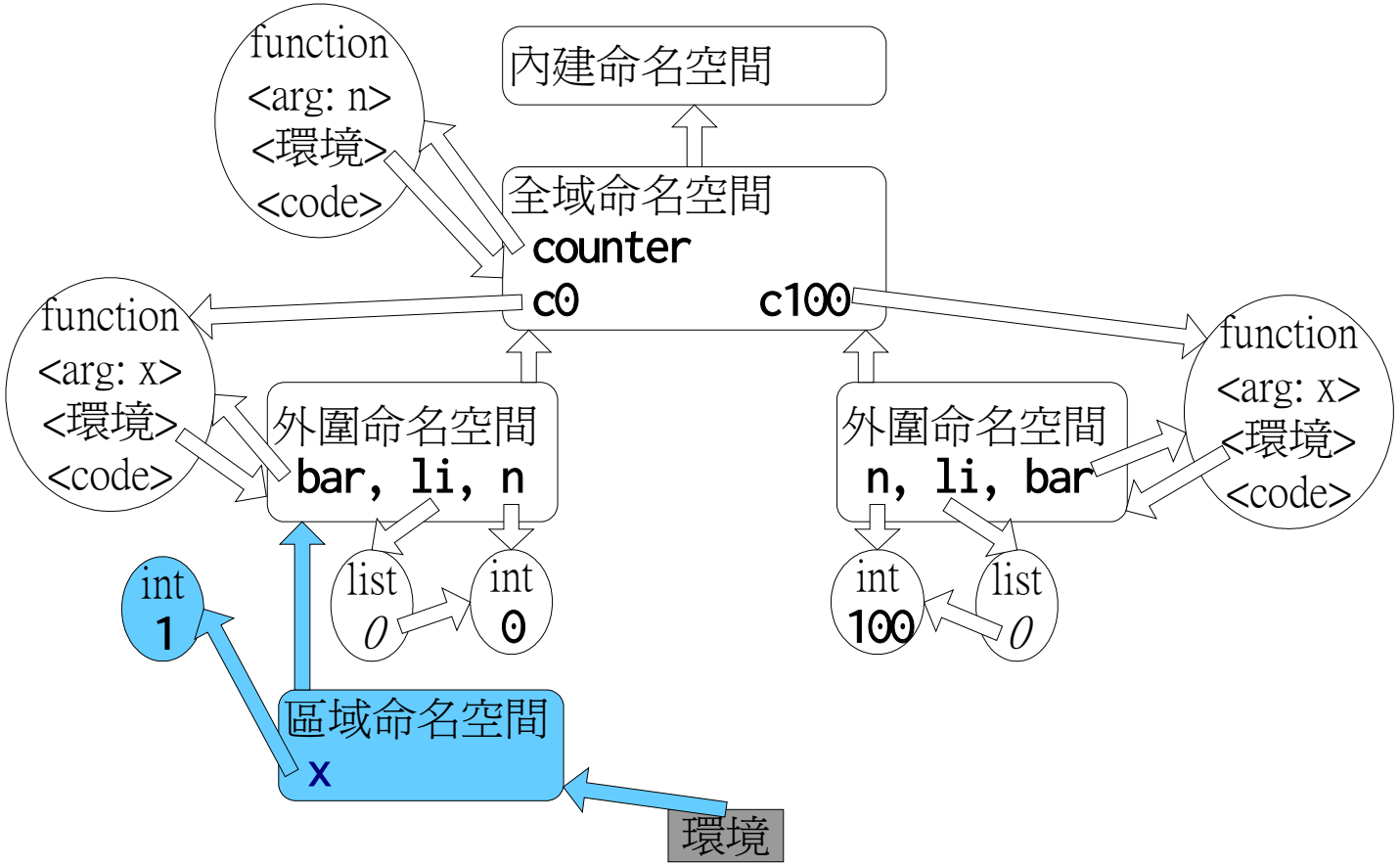
- c100 = counter(100)執行後



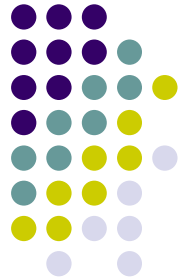
計數器counter示意圖（6/9）



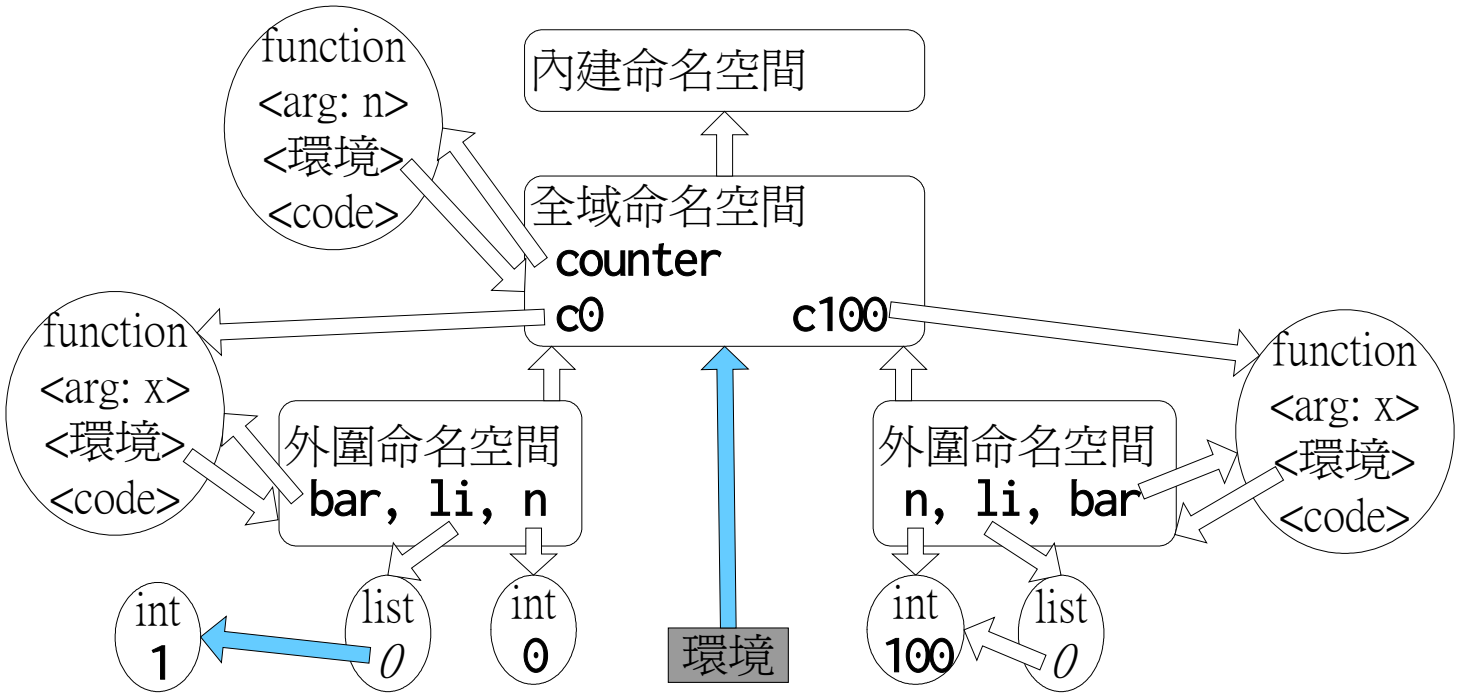
- 呼叫c0(1)



計數器counter示意圖（7/9）

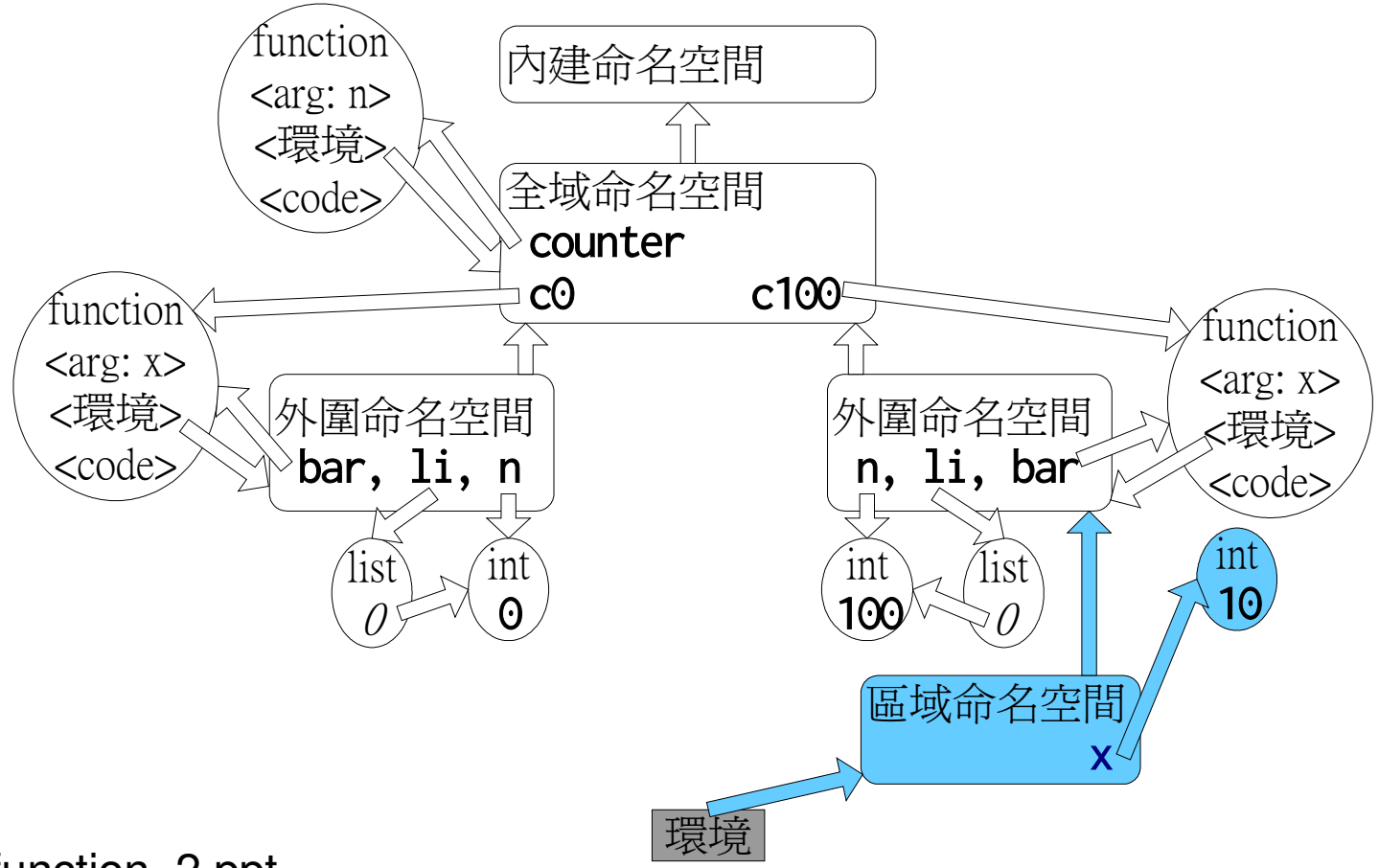


- c0(1)執行後

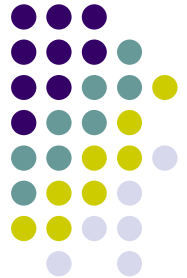


計數器counter示意圖（8/9）

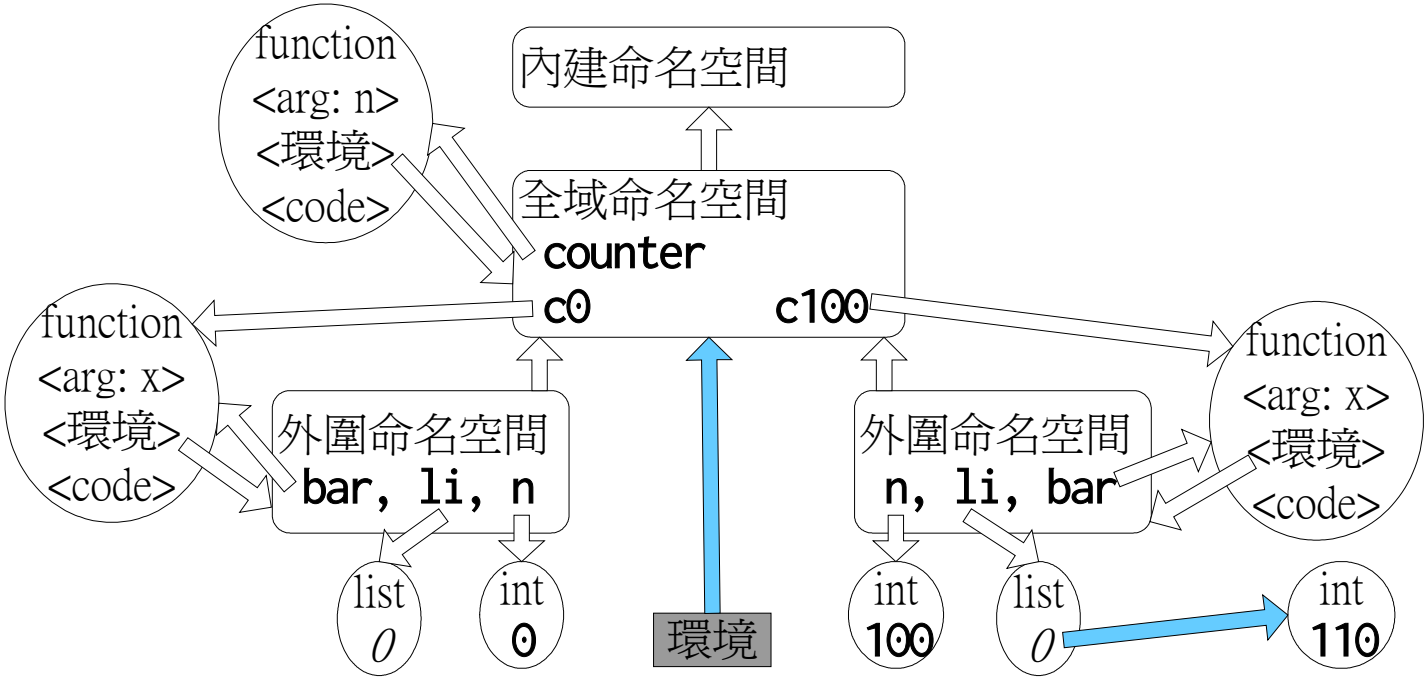
- 呼叫c100(10)



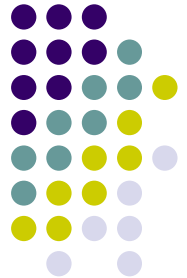
計數器counter示意圖（9/9）



- c100(10)執行後

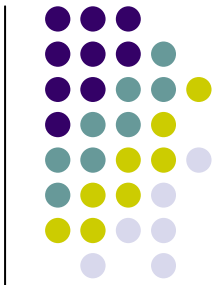


高階函式



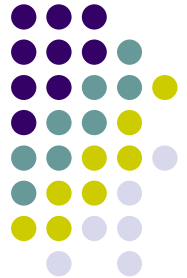
- 函式也是物件，可指派名稱
- 函式作爲參數，傳入另一個函式
- 函式作爲回傳值（函式回傳函式）
- 包含外圍命名空間的函式，叫做「閉包」（closure）

範圍（scope）



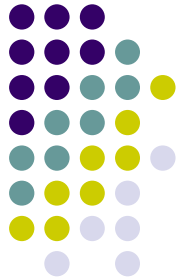
- LEGB：Local區域（Function函式）、Enclosing（外圍）、Global全域（Module模組）、Builtin（內建）
- 生成式（串列、字典、集合）
  除了2.x版的listcomp
- 產生器：運算式、函式
- class述句（類別）：有點兒不同

生成式有其範圍



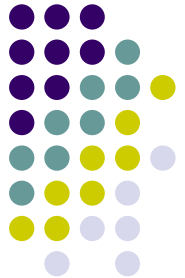
```
>>> i = 999
>>> li = [i**3 for i in range(5)]

>>> li
[0, 1, 8, 27, 64]
>>> i
999
```

fib_memo

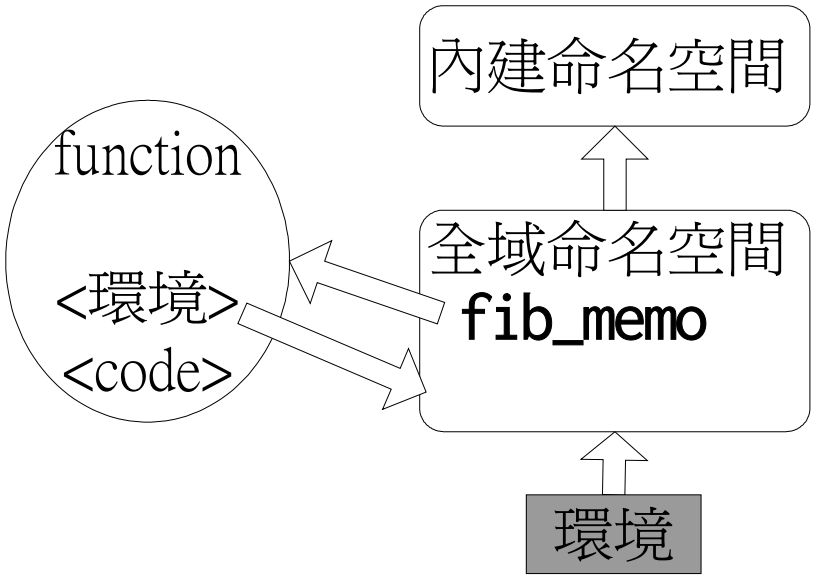


```
def fib_memo():
    memo = {0: 0, 1: 1}
    def sub(n):
        if n not in memo:
            memo[n] = sub(n-1) + sub(n-2)
        return memo[n]
    return sub

fib_m = fib_memo()
x = fib_m(2)
```

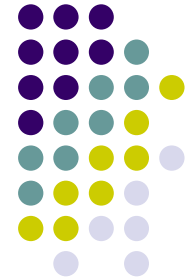
fib_memo示意圖(1/9)



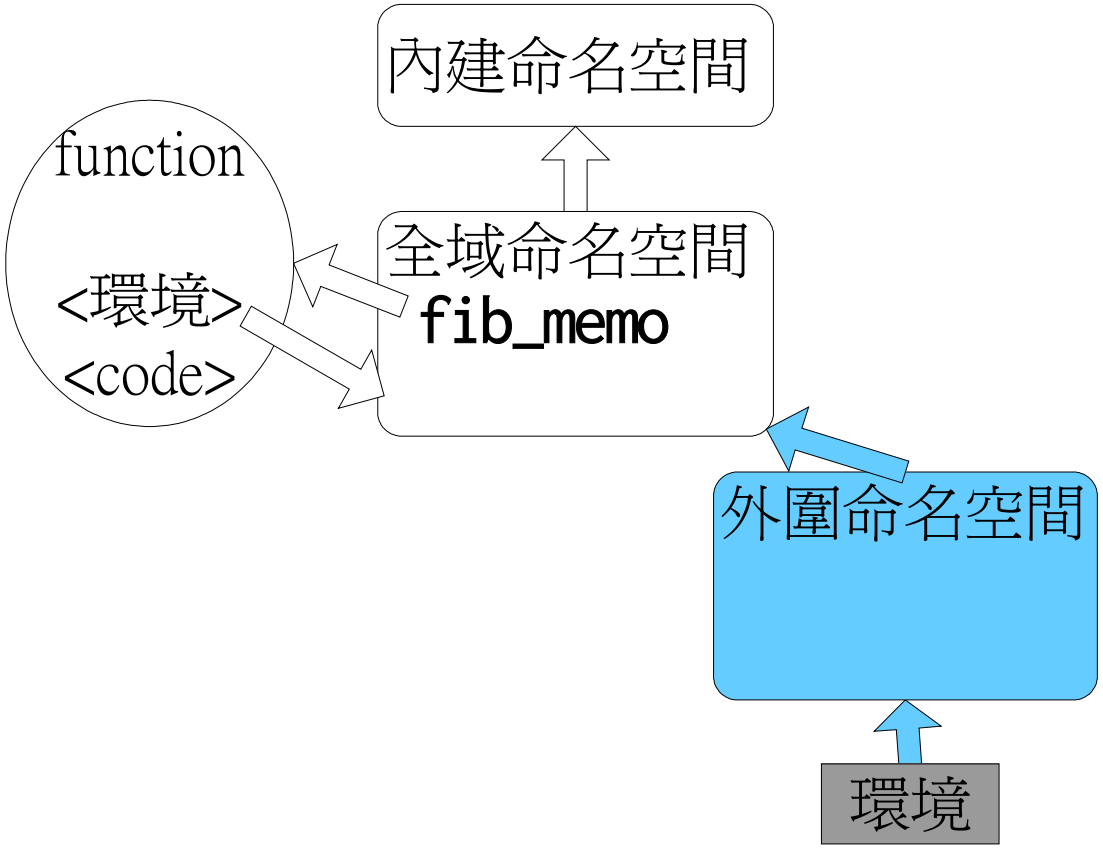
- fib_memo函式定義



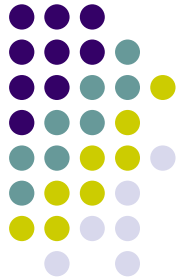
fib_memo示意圖(2/9)



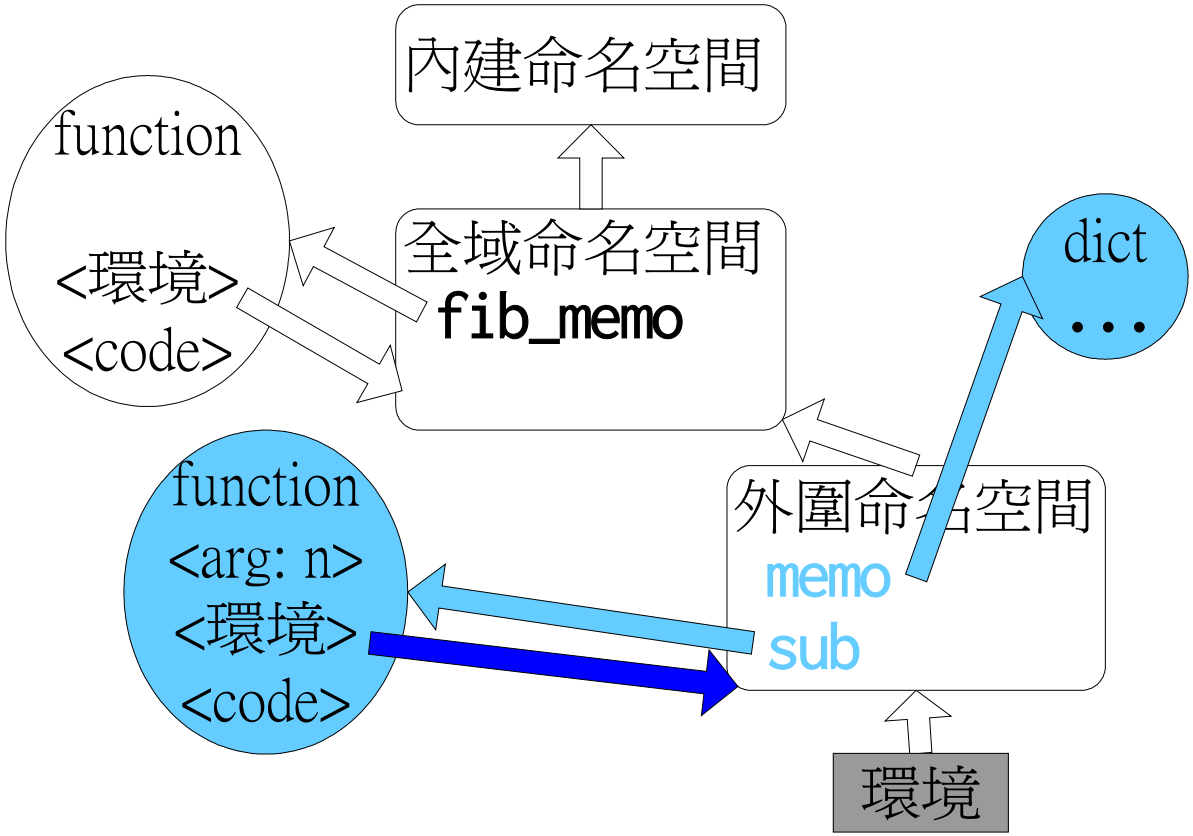
- 呼叫fib_memo，建立外圍命名空間，串聯成新環境



fib_memo示意圖(3/9)

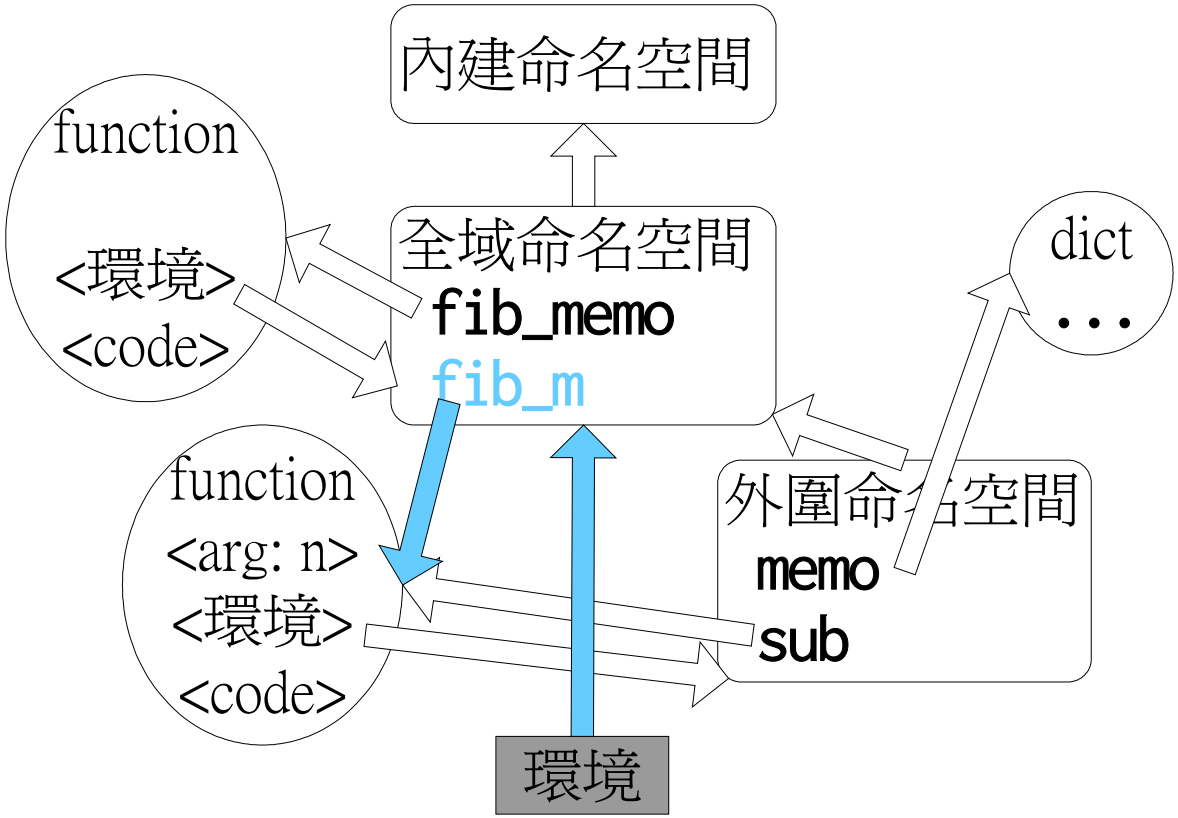


- 呼叫fib_memo()，在新環境裡執行函式體

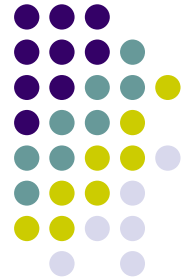


fib_memo示意圖(4/9)

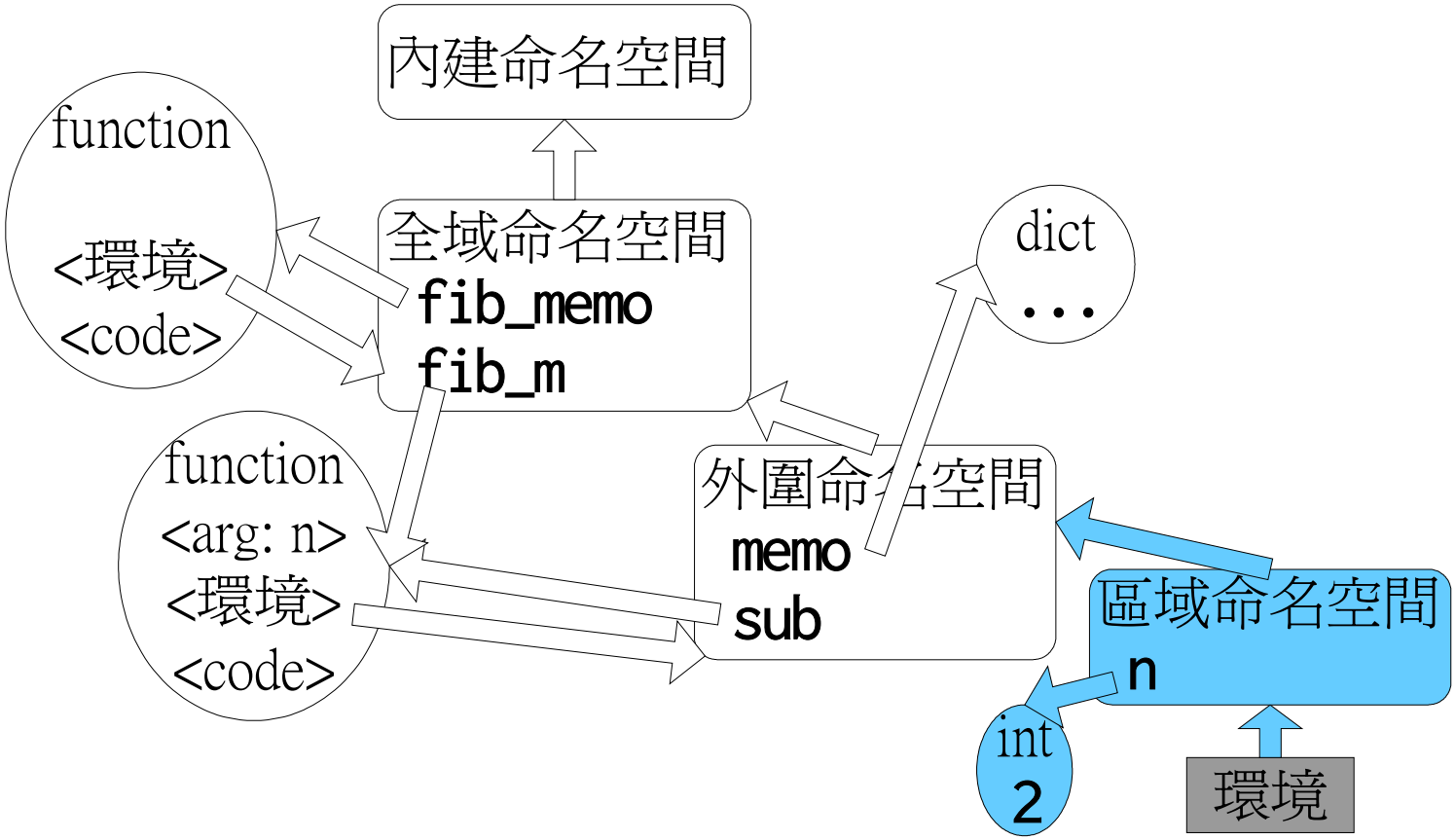
- 執行「fib_m = fib_memo()」



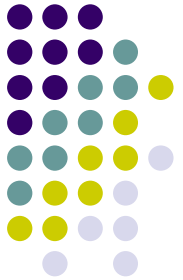
fib_memo示意圖(5/9)



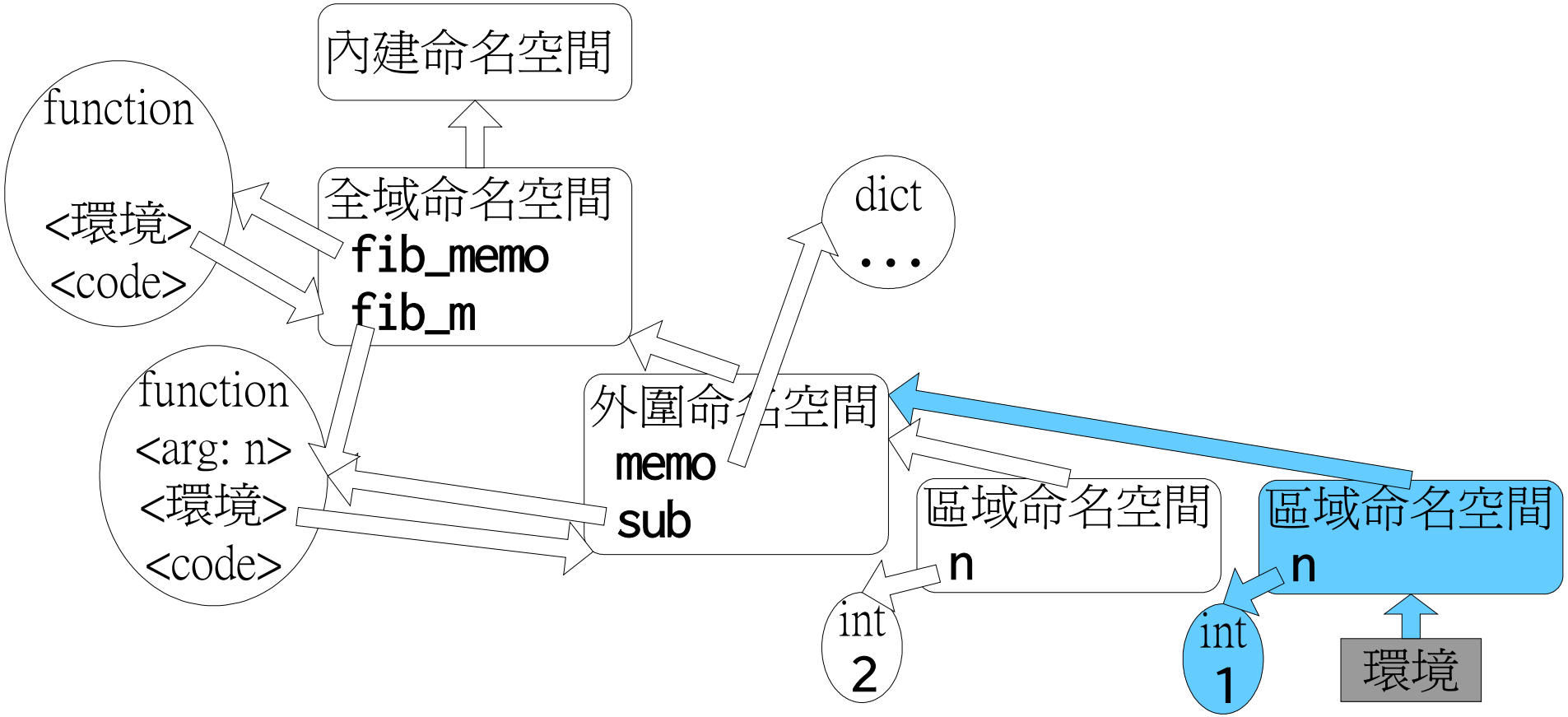
- 呼叫fib_m(2)



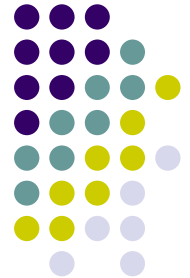
fib_memo示意圖(6/9)



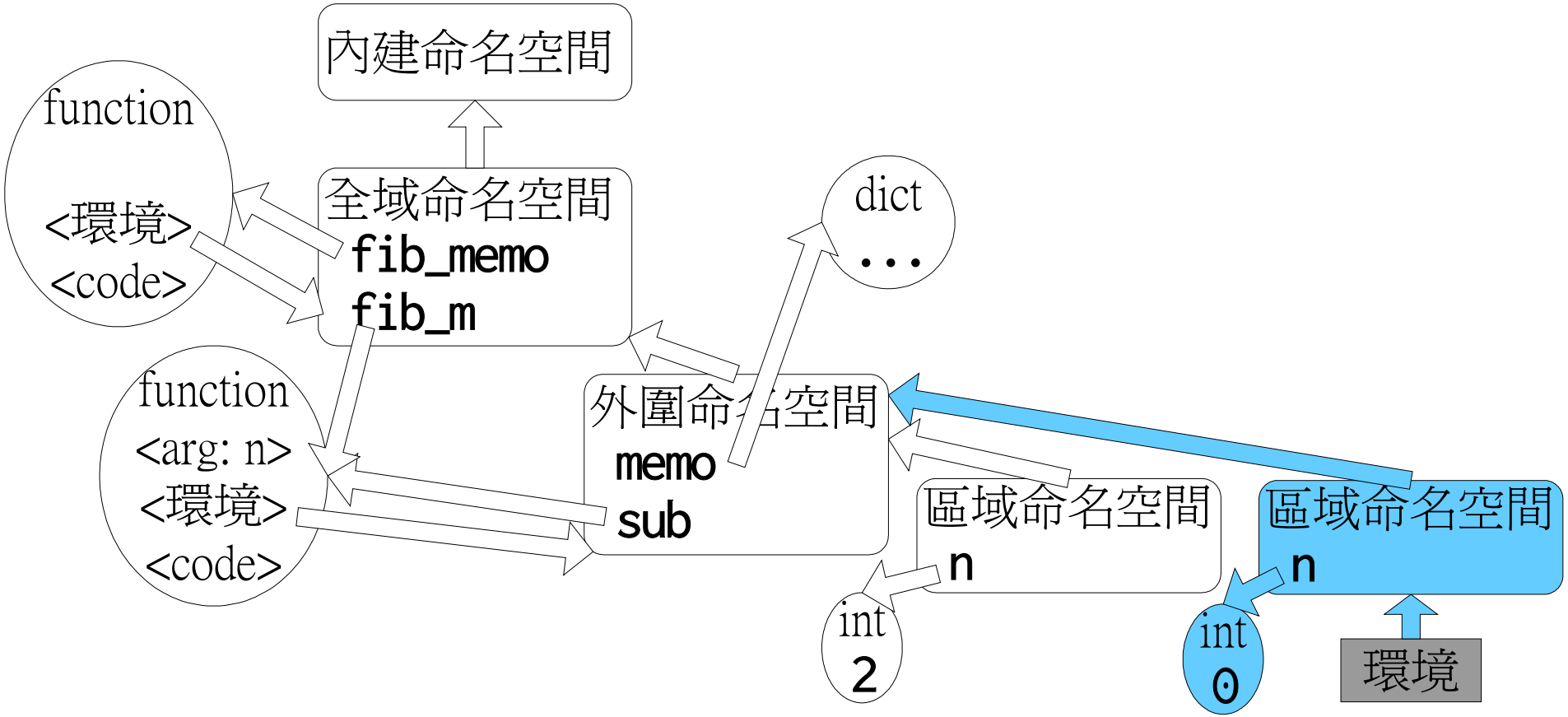
- 呼叫sub(1)



fib_memo示意圖(7/9)

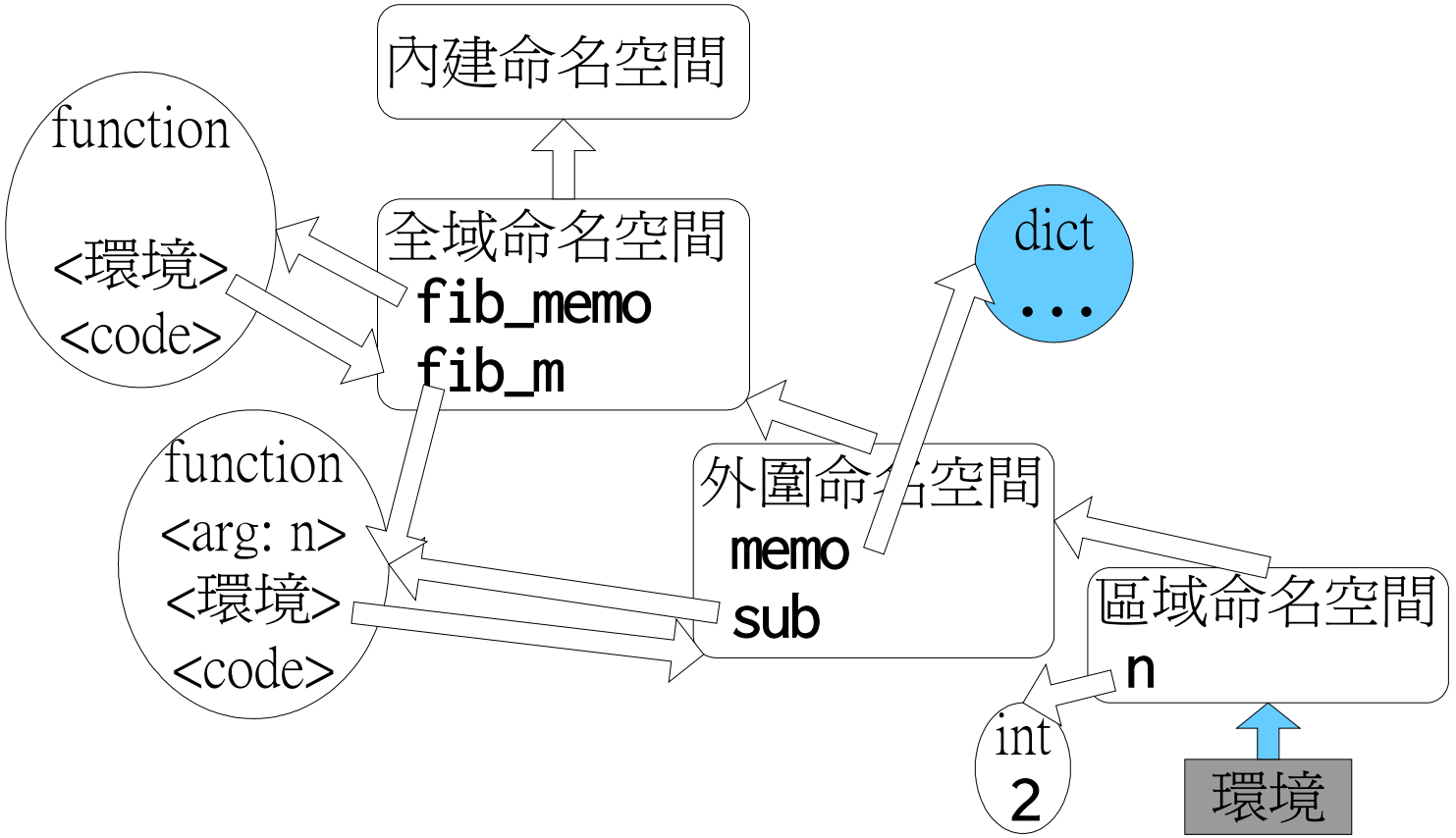


- 呼叫sub(0)

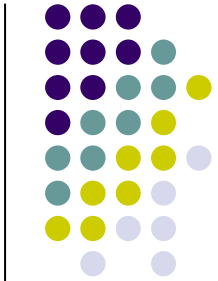


fib_memo示意圖(8/9)

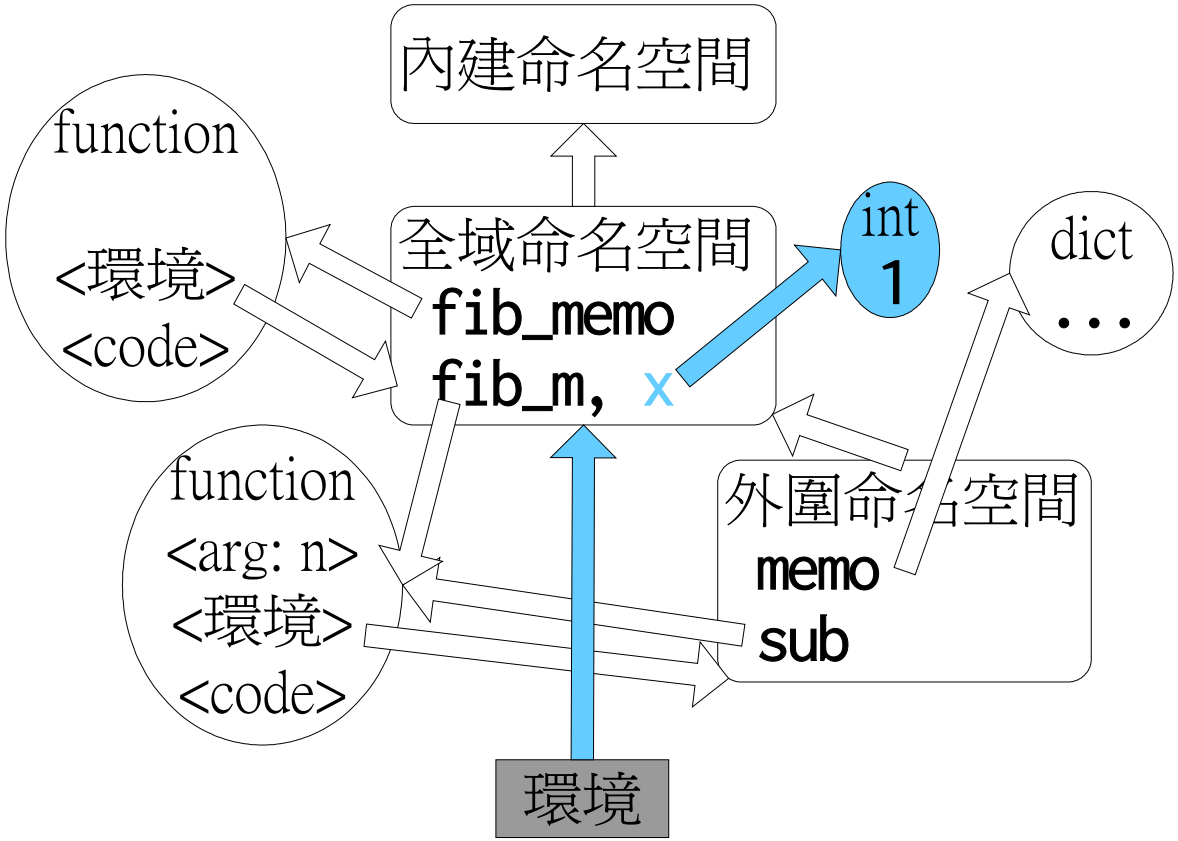
- 執行「memo[2] = sub(1) + sub(0)」



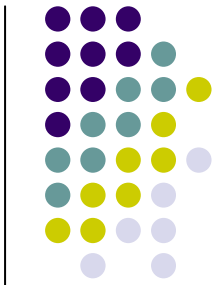
fib_memo示意圖(9/9)



- 執行「x = fib_m(2)」



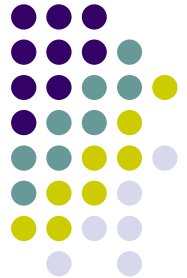
延遲綁定



- 函式定義，形式參數與函式體內的名稱，都尚未綁定到某物件
- 函式呼叫，形參才會綁定實參（物件），到環境裡去找名稱綁定的物件

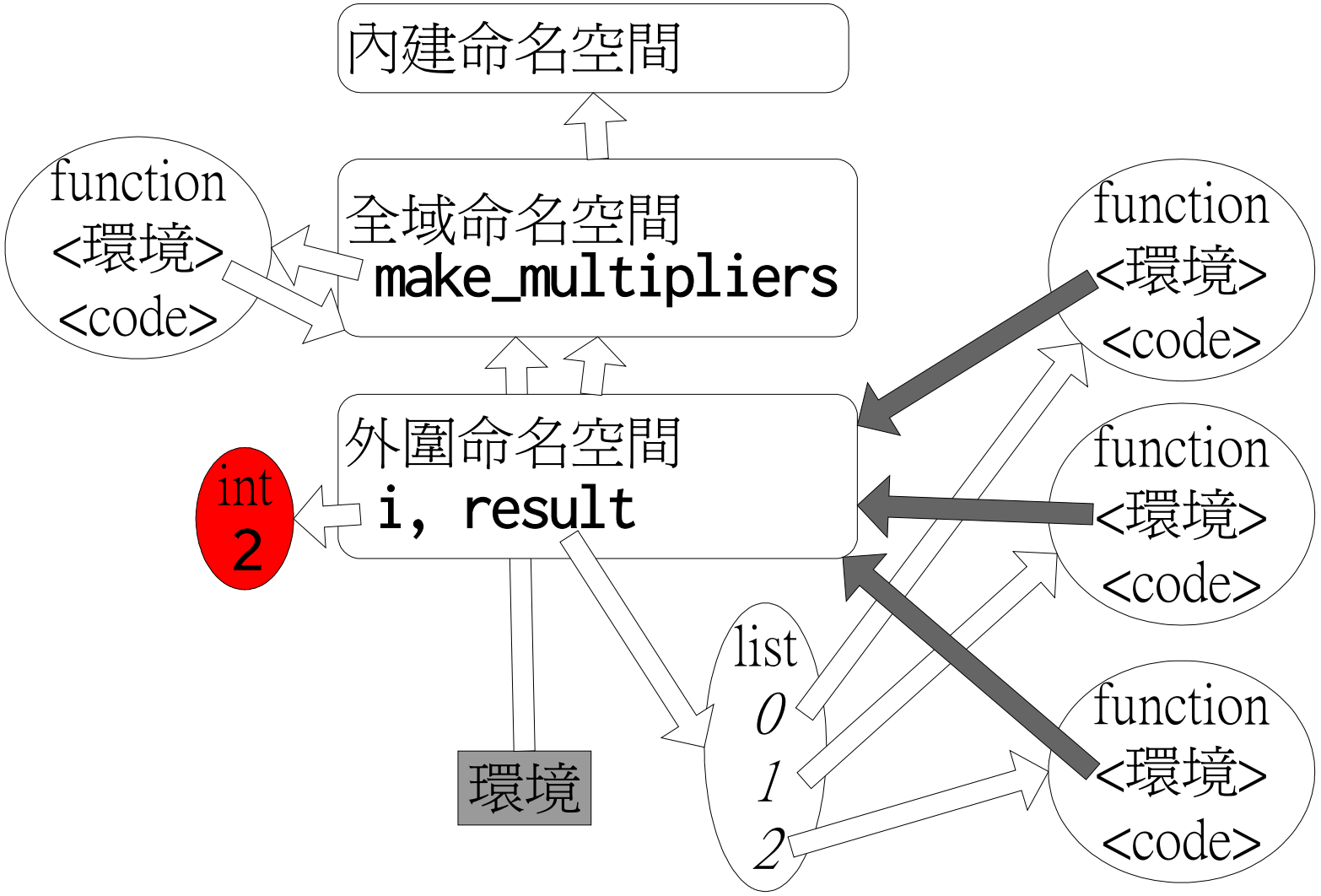
```
a = 3
def f(x):                # 函式定義
    return x + a         # x與a都尚未「綁定」

a = 5
print(f(10))             # 函式呼叫
```

請問會印出哪三個數字？

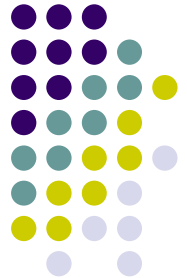


```
def make_multipliers():
    result = []
    for i in range(3):
        def m(x):
            return x * i
        result.append(m)
    return result

for m in make_multipliers():
    print(m(5))
```

make_multipliers()
呼叫完畢後

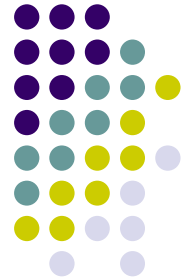


解決辦法



```
def make_multipliers():
    result = []

    for i in range(3):
        def m(x, y=i):      # 參數預設值
            return x * y
        result.append(m)

    return result
```

動態範圍（dynamic scoping）



```
b = 5
def foo():
    a = b + 3
    return a
def bar():              # 想像函式bar在遠處，
    b = 10              # foo根本不知道誰會呼叫它
    return foo()

print(foo())  # 會印出多少？
print(bar())  # 會印出多少？
```

Q&A