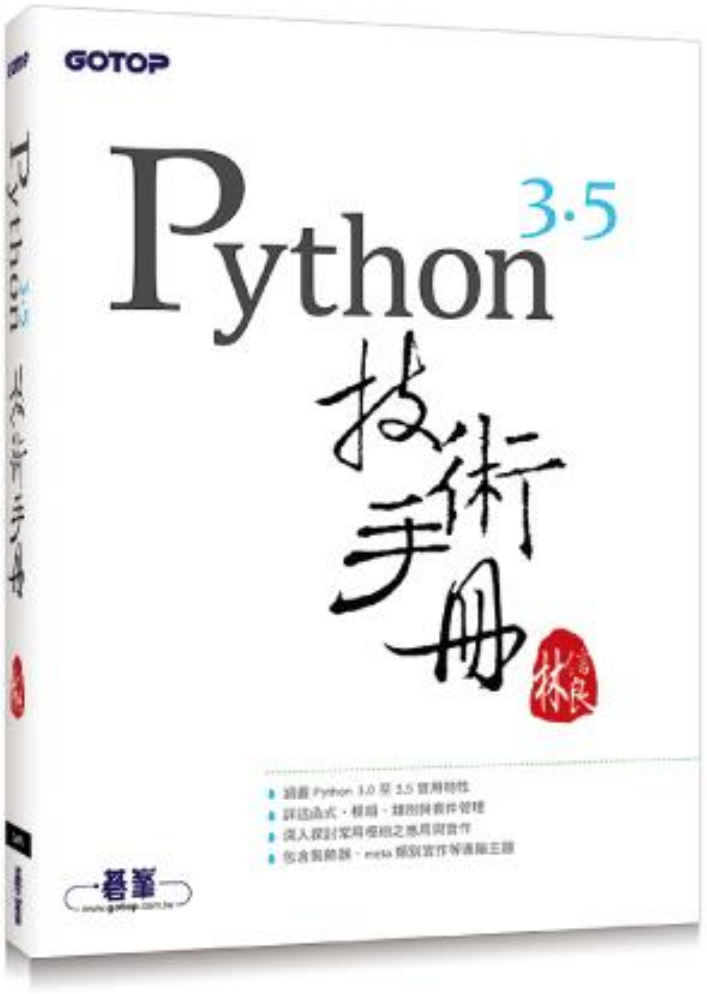


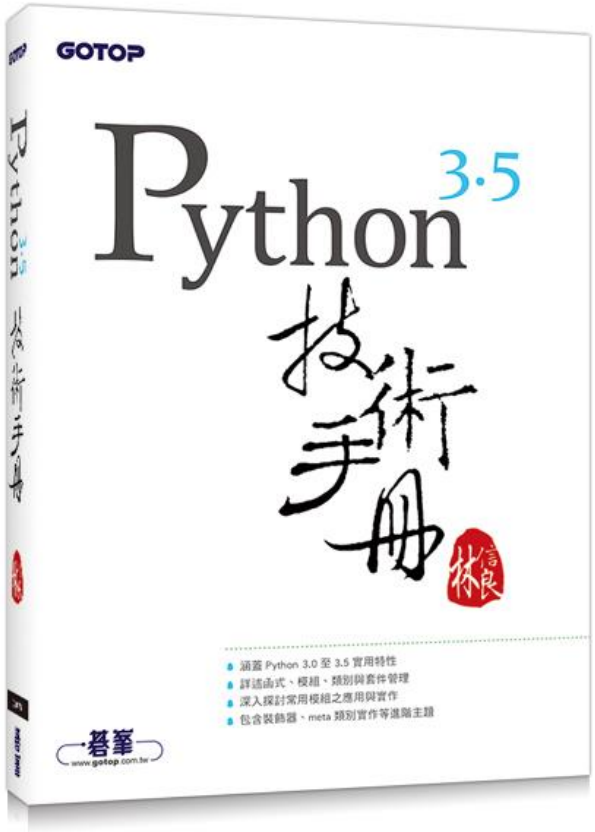
Python 3.5 技術手冊

林信良

CHAPTER

3.型態與運算子

- 學習目標
  - 認識內建型態
  - 學習字串格式化
  - 瞭解變數與運算子
  - 運用切片運算



內建型態

- Pascal 之父 Niklaus E. Writh 曾說過：

  Algorithms + Data Structures = Programs

- 在 Python 中所有的資料都是物件

數值型態

- 整數
  - 型態為 `int`，不再區分整數與長整數
  - 整數的長度不受限制（除了硬體上的限制之外）

```
>>> 10
10
>>> 0b1010
10
>>> 0o12
10
>>> 0xA
10
>>>
```

- 想知道某個資料的型態，可以使用 `type()`

```
>>> type(10)
<class 'int'>
>>> type(0b1010)
<class 'int'>
>>> type(0o12)
<class 'int'>
>>> type(0xA)
<class 'int'>
>>>
```

- int()、oct()、hex()

```
>>> int('10')
10
>>> int(3.14)
3
>>> int(True)
1
>>> int(False)
0
>>> oct(10)
'0o12'
>>> hex(10)
'0xa'
>>>
```

```
>>> int('10', 2)
2
>>> int('10', 8)
8
>>> int('10', 16)
16
>>>
```

- 浮點數
  - float 型態

```
>>> type(3.14)
<class 'float'>
>>> 3.14e-10
3.14e-10
>>> float('1.414')
1.414
>>>
```

```
>>> int('3.14')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.14'
>>> int(float('3.14'))
3
>>>
```

- 布林
  - bool 型態
  - 只有 True 與 False 兩個值
  - bool() 可將 0 轉換為 False，而非 0 值轉換為 True
  - 將 None、False、0、0.0、0j（複數）、''（空字串）、()（空 Tuple）、[]（空清單）、{}（空字典）等傳給 bool()，都會傳回 False，這些型態的其他值傳入bool() 則都會傳回 True

- 複數
  - 型態為 `complex`
  - 撰寫時使用`a + bj` 的形式

```
>>> a = 3 + 2j
>>> b = 5 + 3j
>>> a + b
(8+5j)
>>> type(a)
<class 'complex'>
>>>
```

字串型態

- 可以使用''或""包括文字
- Python 3 之後的版本都是產生 `str` 實例
- 多數 Python 開發者的習慣是使用單引號

```
>>> "Just'in"
"Just'in"
>>> 'Just"in'
'Just"in'
>>> 'c:\workspace'
'c:\\workspace'
>>> "c:\workspace"
'c:\\workspace'
>>>
```

符號	說明
\\	反斜線。
\'	單引號，當你使用''來表示字串，又要表示單引號時使用，例如'Justin\'s Website'。
\"	雙引號，當你使用""來表示字串，又要表示單引號時使用，例如"\"text\" is a string"。
\ooo	以 8 進位數字指定字元碼點(Code point), 最多三位數, 例如'\101'表示字串'A'。
\xhh	以 16 進位數字指定字元碼點，至少兩位數，例如'\0x41'表示字串'A'。
\uhhhh	以 16 位元 16 進位值指定字元，例如'\u54C8\u56C9'表示'哈囉'。
\Uhhhhhhhh	以 32 位元 16 進位值指定字元，例如'\U000054C8\U000056C9'表示'哈囉'。
\0	空字元，請別與空字串搞混，'\0'相當於'\x00'。
\n	換行。
\r	歸位。
\t	Tab。

```
>>> print('c:\todo')
c:  odo
>>> print('c:\\todo')
c:\todo
>>>
```

```
>>> print('\\t')
\t
>>> print(r'\t')
\t
>>> r'\t'
'\\t'
>>> print(r'c:\todo')
c:\todo
>>>
```

- 在三重引號間輸入的任何內容，在最後的字串會照單全收，像是包括換行、縮排等

```
>>> '''Justin is caterpillar!
...    caterpillar is Justin!'''
'Justin is caterpillar!\n   caterpillar is Justin!'
>>> print('''Justin is caterpillar!
...    caterpillar is Justin!''')
Justin is caterpillar!
   caterpillar is Justin!
>>>
```

- 可以使用 `str()` 類別將數值轉換為字串
- 若想知道某個字元的碼點，可以使用 `ord()`
- 使用 `chr()` 則可以將指定碼點轉換為字元

```
>>> str(3.14)
'3.14'
>>> ord('哈')
21704
>>> chr(21704)
'哈'
>>>
```

格式化字串

- print() 函式的顯示預設是會換行

```
name = 'Justin'
print('Hello ', name)
```

- print() 有個 end 參數，在指定的字串顯示之後，end 參數指定的字串就會輸出

```
name = 'Justin'
print('Hello ', end = '')
print(name)
```

- 預設的分隔符號是一個空白字元，如果想要指定其他字元的話，可以指定 sep 參數

```
name = 'Justin'
print('Hello', name, sep = ', ') # 顯示 Hello, Justin
```

- 目前的Python 3 支援兩種格式化方式
  - 舊式（從Python 2 就存在）
  - 新式（從Python 2.6、2.7 開始支援）

- 舊的字串格式化是使用 `string % data` 或 `string % (data1, data2, …)`

```
>>> '哈囉！%s！' % '世界'
'哈囉！世界！'
>>> '你目前的存款只剩 %f 元' % 1000
'你目前的存款只剩 1000.000000 元'
>>> '%d 除以 %d 是 %f' % (10, 3, 10 / 3)
'10 除以 3 是 3.333333'
>>> '%d 除以 %d 是 %.2f' % (10, 3, 10 / 3)
'10 除以 3 是 3.33'
>>> '%5d 除以 %5d 是 %.2f' % (10, 3, 10 / 3)
'   10 除以     3 是 3.33'
>>> '%-5d 除以 %-5d 是 %.2f' % (10, 3, 10 / 3)
'10    除以 3     是 3.33'
>>> '%-5d 除以 %-5d 是 %10.2f' % (10, 3, 10 / 3)
'10    除以 3     是       3.33'
>>>
```

符號	說明
%%	因為%符號已經被用來作為控制符號前置，所以規定使用%%才能在字串中表示%。
%d	10 進位整數。
%f	10 進位浮點數。
%g	10 進位整數或浮點數。
%e, %E	以科學記號浮點數格式化，%e 表示輸出小寫表示，如 2.13 e+12，%E 表示大寫表示。
%o	8 進位整數。
%x, %X	以 16 進位整數格式化，%x 表示字母輸出以小寫表示，%X 則以大寫表示。
%s	字串格式符號。
%r	以 repr()函式取得的結果輸出字串，本章稍後會談到 repr()。

- 使用 Python 3 之後的版本（或者Python 2.6、2.7），建議使用新的格式化

```
>>> '{} 除以 {} 是 {}'.format(10, 3, 10 / 3)
'10 除以 3 是 3.3333333333333335'
>>> '{2} 除以 {1} 是 {0}'.format(10 / 3, 3, 10)
'10 除以 3 是 3.3333333333333335'
>>> '{n1} 除以 {n2} 是 {result}'.format(result = 10 / 3, n1 = 10, n2 = 3)
'10 除以 3 是 3.3333333333333335'
>>>
```

```
>>> '{0:d} 除以 {1:d} 是 {2:f}'.format(10, 3, 10 / 3)
'10 除以 3 是 3.333333'
>>> '{0:5d} 除以 {1:5d} 是 {2:10.2f}'.format(10, 3, 10 / 3)
'   10 除以     3 是       3.33'
>>> '{n1:5d} 除以 {n2:5d} 是 {r:.2f}'.format(n1 = 10, n2 = 3, r = 10 / 3)
'   10 除以     3 是 3.33'
>>> '{n1:<5d} 除以 {n2:<5d} 是 {r:.2f}'.format(n1 = 10, n2 = 3, r = 10 / 3)
'10    除以 3     是 3.33'
>>> '{n1:>5d} 除以 {n2:>5d} 是 {r:.2f}'.format(n1 = 10, n2 = 3, r = 10 / 3)
'   10 除以     3 是 3.33'
>>> '{n1:*^5d} 除以 {n2:!^5d} 是 {r:.2f}'.format(n1 = 10, n2 = 3, r = 10 / 3)
'*10** 除以 !!3!! 是 3.33'
>>>
```

```
>>> names = ['Justin', 'Monica', 'Irene']
>>> 'All Names: {n[0]}, {n[1]}, {n[2]}'.format(n = names)
'All Names: Justin, Monica, Irene'
>>> passwords = {'Justin': 123456, 'Monica': 654321}
>>> 'The password of Justin is {passwds[Justin]}'.format(passwds = passwords)
'The password of Justin is 123456'
>>> import sys
>>> 'My platform is {pc.platform}'.format(pc = sys)
'My platform is win32'
>>>
```

```
>>> format(3.14159, '.2f')
'3.14'
>>>
```

str 與 bytes

- 從Python 3 之後，每個字串都包含了 Unicode 字元
- 每個字串都是 `str` 型態
- 可以使用 `encode()` 方法指定編碼，取得一個 `bytes` 實例
- 如果有個 `bytes` 實例，也可以使用 `decode()` 方法，指定該位元組代表的編碼，將 `bytes` 解碼為 `str` 實例

```
>>> text = '哈'
>>> len(text)
2
>>> text.encode('UTF-8')
b'\xe5\x93\x88'
>>> text.encode('Big5')
b'\xab\xa2'
>>> big5_impl = text.encode('Big5')
>>> type(big5_impl)
<class 'bytes'>
>>> big5_impl.decode('Big5')
'哈'
>>>
```

- 可以在字串前加上個 `b` 來建立一個 `bytes`，這是從 Python 3.3 之後開始支援的語法
- 可以在字串前加上一個 `u`，結果會是個 `str`
  - 為了增加與 Python 2 的相容性

- 在Python 2 中，如果有個 u'哈囉' 字串，
- 會建立一個 unicode，而len(u'哈囉') 的結果會是 2
- 如果單純撰寫'哈囉'字串，會建立一個 str，然而 len('哈囉') 的結果，視原始碼檔案文字編碼而定
  - 如果是 UTF-8 編碼的話，結果會是 6
  - 如果是 Big5 編碼的話，結果會是 4

- 從Python 3 之後，想要取得字串中某個位置字元時，可以使用索引，索引從0 開始
- 想測試某字元是否在字串中，可以使用 `in`
- 字串都是不可變動（Immutable）

```
>>> text = '哈囉'
>>> text[0]
'哈'
>>> text[1]
'囉'
>>>
>>> '哈' in text
True
>>>
```

清單（list）

- 型態是 `list`
- 特性為有序、具備索引，內容與長度可以變動
- 要建立串列，可以使用 `[]` 實字，串列中每個元素，使用逗號「`,`」區隔

```
>>> numbers = [1, 2, 3]
>>> numbers
[1, 2, 3]
>>> numbers.append(4)
>>> numbers
[1, 2, 3, 4]
>>> numbers[0]
1
>>> numbers[1]
2
>>> numbers[3] = 0
>>> numbers
[1, 2, 3, 0]
>>> numbers.remove(0)
>>> numbers
[1, 2, 3]
>>> del numbers[0]
>>> numbers
[2, 3]
>>> 2 in numbers
True
>>>
```

- 從可迭代（Iterable）的物件中建立 `list`

```
>>> list('哈囉！世界！')
['哈', '囉', '！', '世', '界', '！']
>>> list({'哈', '囉', '哈', '囉'})
['哈', '囉']
>>> list((1, 2, 3))
[1, 2, 3]
>>>
```

集合（set）

- 無序、元素不重複
- 可以使用{}包括元素，元素間使用「,」區隔，這會建立 set 實例
- 想建立空集合，必須使用 set()

```
>>> users = set()
>>> users.add('caterpillar')
>>> users.add('Justin')
>>> users
{'caterpillar', 'Justin'}
>>> users.remove('caterpillar')
>>> 'caterpillar' in users
False
>>>
```

- 並非任何元素，都能放到集合

```
>>> {[1, 2, 3]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> {{1, 2, 3}}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>>
```

- 想從其他可迭代的物件中建立 set，像是字串、list 或 Tuple 等，可以使用 set()

```
>>> set('哈囉！世界！')
{'哈', '囉', '！', '世', '界'}
>>> set([1, 2, 3])
{1, 2, 3}
>>> set((1, 2, 3))
{1, 2, 3}
>>>
```

字典（dict）

- 儲存兩兩對應的鍵與值，為 dict 型態
- dict 中的鍵不重複，必須是 hashable

```
>>> passwords = {'Justin' : 123456, 'caterpillar' : 933933}
>>> passwords['Justin']
123456
>>> passwords['caterpillar']
933933
>>> passwords['Irene'] = 970221
>>> passwords
{'caterpillar': 933933, 'Irene': 970221, 'Justin': 123456}
>>> passwords['Irene']
970221
>>> del passwords['caterpillar']
>>> passwords
{'Irene': 970221, 'Justin': 123456}
>>>
```

- 直接使用[]指定鍵要取得值時，若 dict 中並沒有該鍵的存在，會發生 KeyError

```
>>> passwords = {'Justin' : 123456, 'caterpillar' : 933933}
>>> 'Justin' in passwords
True
>>> passwords['Monica']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Monica'
>>> passwords.get('Monica')
>>> passwords.get('Monica') == None
True
>>> passwords.get('Monica', 9999)
9999
>>>
```

- items()、keys()、values()

```
>>> passwords = {'Justin' : 123456, 'caterpillar' : 933933}
>>> list(passwords.items())
[('caterpillar', 933933), ('Justin', 123456)]
>>> list(passwords.keys())
['caterpillar', 'Justin']
>>> list(passwords.values())
[933933, 123456]
>>>
```

- 也可以使用 dict()來建立字典

```
>>> passwords = dict(justin = 123456, momor = 670723, hamimi = 970221)
>>> passwords
{'hamimi': 970221, 'justin': 123456, 'momor': 670723}
>>> passwords = dict([('justin', 123456), ('momor', 670723), ('hamimi',
970221)])
>>> passwords
{'hamimi': 970221, 'justin': 123456, 'momor': 670723}
>>> dict.fromkeys(['Justin', 'momor'], 'NEED_TO_CHANGE')
{'Justin': 'NEED_TO_CHANGE', 'momor': 'NEED_TO_CHANGE'}
>>>
```

Tuple (tuple)

- 許多地方都跟 `list` 很像
- 不過 Tuple 建立之後，就不能變動了

```
>>> 10,
(10,)
>>> 10, 20, 30,
(10, 20, 30)
>>> acct = 1, 'Justin', True
>>> acct
(1, 'Justin', True)
>>> type(acct)
<class 'tuple'>
>>>
```

- 可以將 Tuple 中的元素拆解（Unpack）

```
>>> data = (1, 'Justin', True)
>>> id, name, verified = data
>>> id
1
>>> name
'Justin'
>>> verified
True
>>>
```

- Python 中最常被拿來津津樂道的特色：

```
>>> x = 10
>>> y = 20
>>> x, y = y, x
>>> x
20
>>> y
10
>>>
```

- Python 2 或Python 3，拆解元素指定給變數的特性，在 `list`、`set` 等物件上，也可以使用

```
>>> a, *b = (1, 2, 3, 4, 5)
>>> a
1
>>> b
[2, 3, 4, 5]
>>> a, *b, c = [1, 2, 3, 4, 5]
>>> a
1
>>> b
[2, 3, 4]
>>> c
5
>>>
```

變數

- 這麼寫可不行：

```
print('圓半徑：', 10)
print('圓周長：', 2 * 3.14 * 10)
print('圓面積：', 3.14 * 10 * 10)
```

- 使用變數：

```
radius = 10
PI = 3.14
print('圓半徑：', radius)
print('圓周長：', 2 * PI * radius)
print('圓面積：', PI * radius * radius)
```

- Python 屬於動態定型語言，變數本身並沒有型態資訊
- 建立變數都沒有宣告型態，只要命名變數並使用指定運算「=」指定一個值
- 建立變數前就嘗試存取某變數， 會發生 NameError

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

- 變數始終是個參考至實際物件的名稱，指定運算只是改變了變數的參考對象

```
>>> x = 1.0
>>> y = x
>>> print(id(x), id(y))
25711904 25711904
>>> y = 2.0
>>> print(id(x), id(y))
25711904 25711872
>>>
```

```
>>> x = 1
>>> id(x)
1795962640
>>> x = x + 1
>>> id(x)
1795962656
>>>
```

- 變數在 Python 中只是個參考至物件的名稱，對於可變動物件，才會有以下的操作結果：

```
>>> x = [1, 2, 3]
>>> y = x
>>> x[0] = 10
>>> y
[10, 2, 3]
>>>
```

```
>>> list1 = [1, 2, 3]
>>> list2 = [1, 2, 3]
>>> list1 == list2
True
>>> list1 is list2
False
>>>
```

- 變數本身沒有型態，同一個變數可以前後指定不同的資料型態

```
>>> x = [1, 2, 3]
>>> x.index(2)
1
>>> x = (10, 20, 30)
>>> x.index(20)
1
>>>
```

- 鴨子定型（Duck typing）：「如果它走路像個鴨子，游泳像個鴨子，叫聲像個鴨子，那它就是鴨子。」

- 如果想要知道一個物件有幾個名稱參考至它，可以使用 `sys.getrefcount()`

```
>>> import sys
>>> x = [1, 2, 3]
>>> y = x
>>> z = x
>>> sys.getrefcount(x)
4
>>>
```

- 可以使用`del` 來刪除變數

```
>>> del x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

加減乘除運算

- 1 + 1、1 - 0.1 對你來說都不成問題
- 那麼 0.1 + 0.1 + 0.1、1.0 - 0.8 會是多少

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 1.0 - 0.8
0.19999999999999996
>>>
```

- 開發人員基本上都要知道 IEEE 754 浮點數算術標準
- 不使用小數點，而是使用分數及指數來表示小數
  - 0.5 會以 1/2 來表示
  - 0.75 會以 1/2+1/4 來表示
  - 0.875 會以 1/2+1/4+1/8
  - 0.1 會是1/16+1/32+1/256+1/512 +1/4096+1/8192+...沒有止境

- 如果對小數點的精度要求很高的話，就要小心這個問題
- `0.1 + 0.1 + 0.1 == 0.3`，結果會 `False`
- 如果需要處理小數，而且需要精確的結果，那麼可以使用 `decimal.Decimal`

operator decimal_demo.py

```
import sys
import decimal

n1 = float(sys.argv[1])
n2 = float(sys.argv[2])
d1 = decimal.Decimal(sys.argv[1])
d2 = decimal.Decimal(sys.argv[2])

print('# 不使用 decimal')
print('{0} + {1} = {2}'.format(n1, n2, n1 + n2))
print('{0} - {1} = {2}'.format(n1, n2, n1 - n2))
print('{0} * {1} = {2}'.format(n1, n2, n1 * n2))
print('{0} / {1} = {2}'.format(n1, n2, n1 / n2))

print('\n# 使用 decimal')
print('{0} + {1} = {2}'.format(d1, d2, d1 + d2))
print('{0} - {1} = {2}'.format(d1, d2, d1 - d2))
print('{0} * {1} = {2}'.format(d1, d2, d1 * d2))
print('{0} / {1} = {2}'.format(d1, d2, d1 / d2))
```

```
>python decimal_demo.py 1.0 0.8
# 不使用 decimal
1.0 + 0.8 = 1.8
1.0 - 0.8 = 0.19999999999999996
1.0 * 0.8 = 0.8
1.0 / 0.8 = 1.25

# 使用 decimal
1.0 + 0.8 = 1.8
1.0 - 0.8 = 0.2
1.0 * 0.8 = 0.80
1.0 / 0.8 = 1.25
```

- 在乘法運算上，除了可以使用 * 進行兩個數字的相乘，還可以使用 ** 進行指數運算

```
>>> 2 ** 3
8
>>> 2 ** 5
32
>>> 2 ** 10
1024
>>> 9 ** 0.5
3.0
>>>
```

- 在除法運算上，有 / 與 // 兩個運算子

```
>>> 10 / 3
3.3333333333333335
>>> 10 // 3
3
>>> 10 / 3.0
3.3333333333333335
>>> 10 // 3.0
3.0
>>>
```

- a % b 會進行除法運算並取餘數作為結果
- 布林值需要進行+、-、*、/ 等運算時，True 會被當成是 1，False 會被當成是 0，接著再進行運算

- 使用+運算子可以串接字串，使用*可以重複字串：

```
>>> text1 = 'Just'
>>> text2 = 'in'
>>> text1 + text2
'Justin'
>>> text1 * 10
'JustJustJustJustJustJustJustJustJustJust'
>>>
```

- Python 偏向強型別，也就是型態間在運算時，比較不會自行發生轉換

```
>>> '10' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> '10' + str(1)
'101'
>>> int('10') + 1
11
>>>
```

- `list` 有許多方面與字串類似

```
>>> nums1 = ['one', 'two']
>>> nums2 = ['three', 'four']
>>> nums1 + nums2
['one', 'two', 'three', 'four']
>>> nums1 * 2
['one', 'two', 'one', 'two']
>>>
```

- +串接兩 list，實際上會產生新的 list，然後將原有的兩個 list 中之元素參考，複製至新產生的 list 上
- 同樣的道理也應用在使用 * 重複 list 時

```
>>> nums1 = ['one', 'two']
>>> nums2 = ['three', 'four']
>>> nums_lt = [nums1, nums2]
>>> nums_lt
[['one', 'two'], ['three', 'four']]
>>> nums1[0], nums1[1] = '1', '2'
>>> nums_lt
[['1', '2'], ['three', 'four']]
>>>
```

- tuple 與 list 有許多類似之處

```
>>> nums1 = ['one', 'two']
>>> nums2 = ['three', 'four']
>>> nums_tp = (nums1, nums2)
>>> nums_tp
(['one', 'two'], ['three', 'four'])
>>> nums1[0], nums1[1] = '1', '2'
>>> nums_tp
(['1', '2'], ['three', 'four'])
>>>
```

比較與指定運算

- 大於、小於、等於...

  `>`、`>=`、`<`、`<=`、`==`、`!=`、`<>`
- `<>`效果與`!=`相同，不過建議不要再用
- `x < y <= z`
- `w == x == y == z`
- 可以實作`__gt__()`、`__ge__()`、`__lt__()`、`__le__()`、`__eq__()`或`__comp__()`等方法

- 字串與 `list` 也可以進行

operator_compare.py

```
import sys

str1 = sys.argv[1]
str2 = sys.argv[2]

print('"{0}" > "{1}"? {2}'.format(str1, str2, str1 > str2))
print('"{0}" == "{1}"? {2}'.format(str1, str2, str1 == str2))
print('"{0}" < "{1}"? {2}'.format(str1, str2, str1 < str2))
```

```
>python compare.py Justin Monica
"Justin" > "Monica"? False
"Justin" == "Monica"? False
"Justin" < "Monica"? True
```

指定運算子	範例	結果
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b

邏輯運算

operator uppers.py

```
import sys

str1 = sys.argv[1]
str2 = sys.argv[2]

print('兩個都大寫？', str1.isupper() and str2.isupper())
print('有一個大寫？', str1.isupper() or str2.isupper())
print('都不是大寫？', not (str1.isupper() or str2.isupper()))
```

```
>python uppers.py Justin MONICA
兩個都大寫？ False
有一個大寫？ True
都不是大寫？ False
```

- and、or 有捷徑運算的特性

```
>> [] and 'Justin'
[]
>>> [1, 2] and 'Justin'
'Justin'
>>> [] or 'Justin'
'Justin'
>>> [1, 2] or 'Justin'
[1, 2]
>>>
```

位元運算

operator bitwise_demo.py

```
print('AND 運算：')
print('0 AND 0 {:5d}'.format(0 & 0))
print('0 AND 1 {:5d}'.format(0 & 1))
print('1 AND 0 {:5d}'.format(1 & 0))
print('1 AND 1 {:5d}'.format(1 & 1))

print('\nOR 運算：')
print('0 OR 0 {:6d}'.format(0 | 0))
print('0 OR 1 {:6d}'.format(0 | 1))
print('1 OR 0 {:6d}'.format(1 | 0))
print('1 OR 1 {:6d}'.format(1 | 1))

print('\nXOR 運算：')
print('0 XOR 0 {:5d}'.format(0 ^ 0))
print('0 XOR 1 {:5d}'.format(0 ^ 1))
print('1 XOR 0 {:5d}'.format(1 ^ 0))
print('1 XOR 1 {:5d}'.format(1 ^ 1))
```

- 逐位元運算

```
>>> 0b10010001 & 0b01000001
1
>>> number1 = 0b0011
>>> number1
3
>>> ~number1
-4
>>> number2 = 0b1111
>>> number2
15
>>> ~number2
-16
>>>
```

- 左移 (<<) 與右移 (>>)

operator shift_demo.py

```
number = 1
print('2 的 0 次方: ', number);
print('2 的 1 次方: ', number << 1)
print('2 的 2 次方: ', number << 2)
print('2 的 3 次方: ', number << 3)
```

```
2 的 0 次方:  1
2 的 1 次方:  2
2 的 2 次方:  4
2 的 3 次方:  8
```

```
00000001 → 1
00000010 → 2
00000100 → 4
00001000 → 8
```

- 應用在 set 型態

operator_groups.py

```
import sys

admins = {'Justin', 'caterpillar'}
users = set(sys.argv[1:])
print('站長：{}'.format(admins & users))
print('非站長：{}'.format(users - admins))
print('全部使用者：{}'.format(admins | users))
print('身份不重複使用者：{}'.format(admins ^ users))
print('站長群包括使用者群？{}'.format(admins > users))
print('使用者群包括站長群？{}'.format(admins < users))
```

索引切片運算

```
>>> name = 'Justin'
>>> name[0:3]
'Jus'
>>> name[3:]
'tin'
>>> name[:4]
'Just'
>>> name[:]
'Justin'
>>> name[:-1]
'Justi'
>>> name[-5:-1]
'usti'
>>>
```

- [start:end:step]

```
>>> name = 'Justin'
>>> name[0:4:2]
'Js'
>>> name[2::2]
'si'
>>> name[:5:2]
'Jsi'
>>> name[::2]
'Jsi'
>>> name[::-1]
'nitsuJ'
>>>
```

- 以上的操作，對於 tuple 也是適用的

```
>>> nums[0:3]
(10, 20, 30)
>>> nums[1:]
(20, 30, 40, 50)
>>> nums[:4]
(10, 20, 30, 40)
>>> nums[-5:-1]
(10, 20, 30, 40)
>>> nums[::-1]
(50, 40, 30, 20, 10)
>>>
```

- [:]只是作淺層複製（Shallow copy）

```
>>> nums1 = [10, 20, 30, 40, 50]
>>> nums2 = [60, 70, 80, 90, 100]
>>> tlp1 = (nums1, nums2)
>>> tlp2 = tlp1[:]
>>> tlp2[0][0] = 1
>>> tlp1
([1, 20, 30, 40, 50], [60, 70, 80, 90, 100])
>>>
```

- 可以進行元素取代

```
>>> lt = ['one', 'two', 'three', 'four']
>>> lt[1:3] = [2, 3]
>>> lt
['one', 2, 3, 'four']
>>> lt[1:3] = ['ohoh']
>>> lt
['one', 'ohoh', 'four']
>>> lt[:] = []
>>> lt
[]
```

- 使用 `del` 結合切片運算

```
>>> lt = ['one', 'two', 'three', 'four']
>>> del lt[1:3]
>>> lt
['one', 'four']
>>>
```