Python 3.1 快速導覽 - 迭代器型態 (iterator)

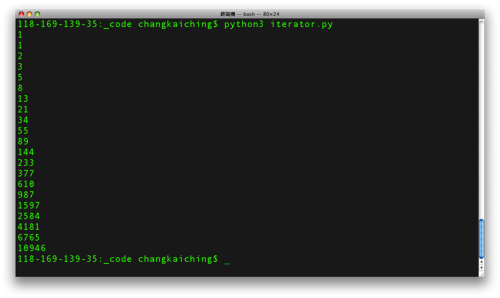
所謂的迭代器型態 (iterator types) 是一種複合資料型態 (compound data type) ，其內的元素 (element) 可依序由迭代規則計算出來，因此可用於 **for** 迴圈 (for loop) 。  
  
  
  
迭代器型態依據迭代器協定 (iterator protocol) 定義，也就是設計類別實需加入以下兩個方法 (method)

|  |  |
| --- | --- |
| **方法** | **描述** |
| \_\_iter\_\_() | 回傳迭代器物件 (object) 本身 |
| \_\_next\_\_() | 取得下一個元素值 |

舉例示範如下

[?](http://pydoing.blogspot.tw/2011/03/python-iteratortype.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class Fibs:      def \_\_init\_\_(self):          self.i = 0          self.j = 1        def \_\_str\_\_(self):          return str(self.i)        def \_\_iter\_\_(self):          return self        def \_\_next\_\_(self):          self.i, self.j = self.j, self.i + self.j          return self.i    fibs = Fibs()  for f in fibs:      print(f)      if f > 10000:          break    # 《程式語言教學誌》的範例程式  # <http://pydoing.blogspot.com/>  # 檔名：iterator.py  # 功能：示範 Python 程式  # 作者：張凱慶  # 時間：西元 2010 年 12 月 |

執行結果如下  
  
  
  
\_\_iter\_\_() 就是回傳 self 本身

|  |  |
| --- | --- |
| 9  10 | def \_\_iter\_\_(self):      return self |

\_\_next\_\_() 取得下一個元素，此例中就是前兩個數字相加

|  |  |
| --- | --- |
| 12  13  14 | def \_\_next\_\_(self):      self.i, self.j = self.j, self.i + self.j      return self.i |

Fibs 提供的是費伯納西數的迭代計算模式，因此可以很快的計算出費伯納西數列。

|  |  |
| --- | --- |
| **中英文術語對照** | |
| 迭代器型態 | iterator types |
| 複合資料型態 | compound data type |
| 元素 | element |
| **for** 迴圈 | for loop |
| 迭代器協定 | iterator protocol |
| 方法 | method |
| 物件 | object |

要把迭代的結果存放在set,list,tuple,dict結構中，是否必須先將迭代結果轉化成物件???

涉及到數值型態時，是否要透過剖析轉換型別或是使用內建的型別專用函式進行數值資料的攫取???

**4.5. Iterator Types**

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

container.\_\_iter\_\_()

Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the tp\_iter slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:

iterator.\_\_iter\_\_()

Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements. This method corresponds to the tp\_iter slot of the type structure for Python objects in the Python/C API.

iterator.\_\_next\_\_()

Return the next item from the container. If there are no further items, raise the StopIteration exception. This method corresponds to the tp\_iternext slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

Once an iterator’s \_\_next\_\_() method raises StopIteration, it must continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken.

4.5.1. Generator Types

Python’s generators provide a convenient way to implement the iterator protocol. If a container object’s \_\_iter\_\_() method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the \_\_iter\_\_() and \_\_next\_\_() methods. More information about generators can be found in the documentation for the yield expression.

**4.6.6. Ranges**

The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

class range(stop)class range(start, stop[, step])

The arguments to the range constructor must be integers (either built-in int or any object that implements the \_\_index\_\_ special method). If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. If step is zero, ValueError is raised.

For a positive step, the contents of a range r are determined by the formula r[i] = start + step\*i where i >= 0 and r[i] < stop.

For a negative step, the contents of the range are still determined by the formula r[i] = start + step\*i, but the constraints are i >= 0 and r[i] > stop.

A range object will be empty if r[0] does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than sys.maxsize are permitted but some features (such as len()) may raise OverflowError.

Range examples:

>>>>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(1, 11))

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> list(range(0, 30, 5))

[0, 5, 10, 15, 20, 25]

>>> list(range(0, 10, 3))

[0, 3, 6, 9]

>>> list(range(0, -10, -1))

[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

>>> list(range(0))

[]

>>> list(range(1, 0))

[]

Ranges implement all of the common sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).

Range objects implement the collections.abc.Sequence ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices (see Sequence Types — list, tuple, range):

>>>>>> r = range(0, 20, 2)

>>> r

range(0, 20, 2)

>>> 11 in r

False

>>> 10 in r

True

>>> r.index(10)

5

>>> r[5]

10

>>> r[:5]

range(0, 10, 2)

>>> r[-1]

18

Testing range objects for equality with == and != compares them as sequences. That is, two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different start, stop and step attributes, for example range(0) == range(2, 1, 3) or range(0, 3, 2) == range(0, 4, 2).)

Changed in version 3.2: Implement the Sequence ABC. Support slicing and negative indices. Test int objects for membership in constant time instead of iterating through all items.

Changed in version 3.3: Define ‘==’ and ‘!=’ to compare range objects based on the sequence of values they define (instead of comparing based on object identity).

New in version 3.3: The start, stop and step attributes.

**4.6.4. Lists**

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

class list([iterable])

Lists may be constructed in several ways:

•Using a pair of square brackets to denote the empty list: []

•Using square brackets, separating items with commas: [a], [a, b, c]

•Using a list comprehension: [x for x in iterable]

•Using the type constructor: list() or list(iterable)

The constructor builds a list whose items are the same and in the same order as iterable‘s items. iterable may be either a sequence, a container that supports iteration, or an iterator object. If iterable is already a list, a copy is made and returned, similar to iterable[:]. For example, list('abc') returns ['a', 'b', 'c'] and list( (1, 2, 3) ) returns [1, 2, 3]. If no argument is given, the constructor creates a new empty list, [].

Many other operations also produce lists, including the sorted() built-in.

Lists implement all of the common and mutable sequence operations. Lists also provide the following additional method:

sort(\*, key=None, reverse=None)

This method sorts the list in place, using only < comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

sort() accepts two arguments that can only be passed by keyword (keyword-only arguments):

key specifies a function of one argument that is used to extract a comparison key from each list element (for example, key=str.lower). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of None means that list items are sorted directly without calculating a separate key value.

The functools.cmp\_to\_key() utility is available to convert a 2.x style cmp function to a key function.

reverse is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

This method modifies the sequence in place for economy of space when sorting a large sequence. To remind users that it operates by side effect, it does not return the sorted sequence (use sorted() to explicitly request a new sorted list instance).

The sort() method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

CPython implementation detail: While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python makes the list appear empty for the duration, and raises ValueError if it can detect that the list has been mutated during a sort.

**4.6.5. Tuples**

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the enumerate() built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a set or dict instance).

class tuple([iterable])

Tuples may be constructed in a number of ways:

•Using a pair of parentheses to denote the empty tuple: ()

•Using a trailing comma for a singleton tuple: a, or (a,)

•Separating items with commas: a, b, c or (a, b, c)

•Using the tuple() built-in: tuple() or tuple(iterable)

The constructor builds a tuple whose items are the same and in the same order as iterable‘s items. iterable may be either a sequence, a container that supports iteration, or an iterator object. If iterable is already a tuple, it is returned unchanged. For example, tuple('abc') returns ('a', 'b', 'c') and tuple( [1, 2, 3] ) returns (1, 2, 3). If no argument is given, the constructor creates a new empty tuple, ().

Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity. For example, f(a, b, c) is a function call with three arguments, while f((a, b, c)) is a function call with a 3-tuple as the sole argument.

Tuples implement all of the common sequence operations.

For heterogeneous collections of data where access by name is clearer than access by index, collections.namedtuple() may be a more appropriate choice than a simple tuple object.

**len(s)**

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

**list([iterable])**

Rather than being a function, list is actually a mutable sequence type, as documented in Lists and Sequence Types — list, tuple, range.

**locals()**

Update and return a dictionary representing the current local symbol table. Free variables are returned by locals() when it is called in function blocks, but not in class blocks.

Note:

The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

**map(function, iterable, ...)**

Return an iterator that applies function to every item of iterable, yielding the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see itertools.starmap().

**max(iterable, \*[, key, default])max(arg1, arg2, \*args[, key])**

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The key argument specifies a one-argument ordering function like that used for list.sort(). The default argument specifies an object to return if the provided iterable is empty. If the iterable is empty and default is not provided, a ValueError is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as sorted(iterable, key=keyfunc, reverse=True)[0] and heapq.nlargest(1, iterable, key=keyfunc).

New in version 3.4: The default keyword-only argument.

**memoryview(obj)**

Return a “memory view” object created from the given argument. See Memory Views for more information.

min(iterable, \*[, key, default])min(arg1, arg2, \*args[, key])

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

There are two optional keyword-only arguments. The key argument specifies a one-argument ordering function like that used for list.sort(). The default argument specifies an object to return if the provided iterable is empty. If the iterable is empty and default is not provided, a ValueError is raised.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as sorted(iterable, key=keyfunc)[0] and heapq.nsmallest(1, iterable, key=keyfunc).

New in version 3.4: The default keyword-only argument.

**next(iterator[, default])**

Retrieve the next item from the iterator by calling its \_\_next\_\_() method. If default is given, it is returned if the iterator is exhausted, otherwise StopIteration is raised.

**object()**

Return a new featureless object. object is a base for all classes. It has the methods that are common to all instances of Python classes. This function does not accept any arguments.

Note:

object does not have a \_\_dict\_\_, so you can’t assign arbitrary attributes to an instance of the object class.

**iter(object[, sentinel])**

Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, object must be a collection object which supports the iteration protocol (the \_\_iter\_\_() method), or it must support the sequence protocol (the \_\_getitem\_\_() method with integer arguments starting at 0). If it does not support either of those protocols, TypeError is raised. If the second argument, sentinel, is given, then object must be a callable object. The iterator created in this case will call object with no arguments for each call to its \_\_next\_\_() method; if the value returned is equal to sentinel, StopIteration will be raised, otherwise the value will be returned.

See also Iterator Types.

One useful application of the second form of iter() is to read lines of a file until a certain line is reached. The following example reads a file until the readline() method returns an empty string:

with open('mydata.txt') as fp:

for line in iter(fp.readline, ''):

process\_line(line)

range函式，會回傳一能產生整數的迭代器

可參考: 精通Python3程式設計 (第二版) pp. 115-116, pp. 118-119,

range()和len()的使用

可參考: 精通Python3程式設計 (第二版) pp. 140-141

運用迭代相關的函式取得和執行迭代器

可參考: Python錦囊妙計 (第三版) p. 126

List comprehension

可參考: 精通Python3程式設計 (第二版) pp. 119-120