P y t h o n - 4

Table Of Contents

1 Copying An Object

1.1 Introduction

Module copy in the standard Python library offers two functions to create copies.

```
import copy
new_list = copy.copy(existing_list)

new_list_of_dicts = copy.deepcopy(existing_list_of_dicts)
```

- the module copy from the Python Standard Library offers two functions to make copies. Normally, you use copy.copy, which makes a shallow copy—it copies an object, but for each attribute or item of the object, it continues to share references, which is faster and saves memory.
- The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):
- If you are about to alter an object, but you want to keep the original object unaltered, you must make a copy.

1.2 Reference Vs Copies

it is crucial that you learn to draw the distinction between *altering an object* and *assigning to a name*,These two kinds of operations have nothing to do with each other.

the issue of references versus copies is meaningful only when you alter some object.

A statement such as a=[ ] rebinds name a but performs no alteration at all on the object that was previously bound to name a.

```
In [46]: list2=[1,[2,3],'hello']
In [49]: test=copy.copy(list2)
In [50]: test[0]=7
In [51]: print test,list2
[7, [2, 3], 'hello'] [1, [2, 3], 'hello']

In [87]: test[1].append(9)    # alter item.
In [89]: print test,list2
[7, [2, 3, 9], 'hello'] [1, [2, 3, 9], 'hello']

In [76]: test[1]=4            # assign to a item.
In [78]: print test, list2
[7, 4, 'bingo'] [1, [2, 3], 'hello']

In [69]: list3=list2[:]       # another way to copy a list.
In [70]: list3
Out[70]: [1, [2, 3], 'hello']
```

```
In [63]: d={'a':4,'b':7}
In [64]: d2=dict(d)                # shallow copy

In [65]: d2
Out[65]: {'a': 4, 'b': 7}

In [66]: d2['a']=8                 # rebind 'a'

In [67]: print d,d2
{'a': 4, 'b': 7} {'a': 8, 'b': 7}

In [71]: d2.pop('a')
Out[71]: 8

In [79]: print d,d2
{'a': 4, 'b': 7} {'b': 7}
```

dict	dict(dict-instance),copy.copy(dict-instance)	immutable
list	list-instance[:],copy.copy(list-instance)	immutable
own-class	supply special method _copy__ or _deepcopy__	

1.3 Noted

Note that you do not need to copy immutable objects (strings, numbers, tuples, etc.) because you don’t have to worry about altering them. (is checks for identity; for checking mere equality, you use the == opera- tor).

```
In [90]: s = 'cat'

In [91]: t = copy.copy(s)

In [92]: t is s
Out[92]: True
```

is checks for identity; for checking mere equality, you use the == operator

2 Construct Lists With List Comprehension

set all items greater than 100 to 100

```
L[:] = [min(x,100) for x in L]
```

In Python 2.4, you should consider using a generator expression, rather than a list comprehension, when the sequence may be long and you only need one item at a time,The syntax of generator expressions is just the same as for list comprehensions, except that generator expressions are surrounded by parentheses, (and), not brackets, [and].

3 Return An Element Of A List If It Exists

The function in this recipe just checks whether i is a valid index by applying Python’s indexing rule: valid indices are negative ones down to -len(L) inclusive, and non-negative ones up to len(L) exclusive.the negative is for the situation we got the list item from back.

Comparing below 2 approach, the first one is recommended, it is more fast, clarity and readable.

```
def list_get(L, i, v=None):
    if -len(L) <= i < len(L): return L[i]
    else: return v

def list_get_egfp(L, i, v=None):
    try: return L[i]
    except IndexError: return v
```

4 Looping Over Items And Their Indices In A Sequence

Python offers the built-in function enumerate, which takes any iter- able argument and returns an iterator yielding all the pairs (two-item tuples) of the form (index, item), one pair at a time.

```
for index, item in enumerate(sequence):
    if item > 23:
        sequence[index] = transform(item)

for item in sequence:   # more directly way to get the item. but index is not available.
    process(item)
```

4.1 Dict VS List

d=dict(enumerate(L)). This gives a dictionary d that’s equivalent to list L, d[i]=L[i]

```
In [93]: list2
Out[93]: [1, [2, 3, 9], 'hello']

In [94]: enumerate(list2)
Out[94]: <enumerate at 0x2e361e0>

In [95]: d =dict(enumerate(list2))

In [96]: d
Out[96]: {0: 1, 1: [2, 3, 9], 2: 'hello'}

In [98]: print d[1],list2[1]
[2, 3, 9] [2, 3, 9]
```

5 Creating Lists Of Lists Without Sharing Reference

You want to create a multidimensional list but want to avoid implicit reference sharing.it means one item updated will not cause other change of this list..

5.1 Recommended

We have below 2 method to create a 2X3 array. you will see if we change the item in this array, it will not cause any change for other items..

```
In [100]: multilist = [[0 for col in range(3)] for row in range(2)]
In [101]: multilist
Out[101]: [[0, 0, 0], [0, 0, 0]]
In [102]: multilist[0][0]=4
In [103]: multilist
Out[103]: [[4, 0, 0], [0, 0, 0]]

In [113]: multilist = [[0]*5 for row in range(3)]
In [114]: multilist[0][0]=4
In [115]: multilist
Out[115]: [[4, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

5.2 The Reason Why We Talking About "Implicit Reference"

Because As a beginner we always be confused, we could get an array of 5 zeros through below method.

```
In [104]: alist = [0] * 5   # get an array of 5 zeros.

In [105]: alist
Out[105]: [0, 0, 0, 0, 0]
```

So when we want to got a multi-dimension. it will be a nature progression like below.To understand the issue, it helps to decompose the creation of the multidimensional list into two steps:

```
In [142]: row = [0]*5       # a list with 5 reference to 0

In [143]: multi =[row]*3    # a list with 3 reference to the row object.

In [144]: multi             # at first sight, it is exactly the format of 2 dimension array. but it have below
Out[144]: [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

# because multi is compose of 3 reference to a same row object. so if you assign a value to multi[0][0], indeed
# is updated, so multi[1] & multi[2] is also update accordingly, because they share same object.
In [106]: multi = [[0] * 5] * 3

In [107]: multi             # we get a second dimension. but the three item refer to same object.
Out[107]: [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

In [108]: multi[0][0]='op'

In [109]: multi
```

```
Out[109]: [['op', 0, 0, 0, 0], ['op', 0, 0, 0, 0], ['op', 0, 0, 0, 0]]
```

Question:

1. why it is '[row]*3' but not 'row*3'? refer to **python-common-question.html**
2. Frankly, it is meaningless to immutable object.

6 Flattening A Nested Sequence

loop over a “flattened” sequence, “expanding” each sub-sequence into a single; Flattening a nested sequence, or, equivalently, “walking” sequentially over all the leaves of a “tree”.

6.1 Resolution

- By default, we can arbitrarily say that every list or tuple is to be “expanded”, and ***nothing else.*** Then, a recursive generator offers the simplest solution:
- The key is : take an argument that’s a predicate to tell us what items we are to expand.
- Strings are iterable, but almost invariably applications want to treat them as scalars, not as subsequences.

```
def list_or_tuple(x):
    return isinstance(x, (list, tuple))
def flatten(sequence, to_expand=list_or_tuple):
    for item in sequence:
        if to_expand(item):
            for subitem in flatten(item, to_expand):
                yield subitem
        else:
            yield item

if __name__ == "__main__":
    for x in flatten([1, 2, [3, [ ], 4, [5, 6], 7, [8,], ], 9]):
        print x

#emit: 1,2,3,4,5,6,7,8,9
```

6.2 Summary

The only problem with this common task is that, in the general case, determining what is to be “expanded”, and what is to be yielded as a scalar.

identify whether object is iterable	iter(obj) will raise typeError
identify whether object is string like	isinstance(obj, basestring) is True

below is an extension if you want to return a non-string argument.

```
def nonstring_iterable(obj):
    try: iter(obj)
    except TypeError: return False
    else: return not isinstance(obj, basestring)
```

7 Removing Or Reordering Columns In A List Of Row.

You have a list of lists (rows) and need to get another list of the same rows but with some columns removed and/or reordered.

in other words: reorder or omit some of the original columns,we should keep row unchanged just update columns.

```
In [154]: listOfRows = [ [1,2,3,4], [5,6,7,8], [9,10,11,12] ]

In [155]: newList = [ [row[0], row[3], row[2]] for row in listOfRows ]
In [159]: newList = [ [row[ci] for ci in (0, 3, 2)] for row in listOfRows ] # nest a inner list comprehension l
In [160]: newList
Out[160]: [[1, 4, 3], [5, 8, 7], [9, 12, 11]]
```

Comparing the first and second approach, if you adopt the second one,you gain some potential generality, because you can choose to give a name to the auxiliary sequence of indices,pass it as an argument to a function.

```
def pick_and_reorder_columns(listofRows, column_indexes):
    return [ [row[ci] for ci in column_indexes] for row in listofRows ]
```

8 Transposing A List

8 Transposing A List

This recipe shows a concise yet clear way to turn rows into columns,

```
In [5]: arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

In [7]: print [[r[col] for r in arr] for col in range(len(arr[0]))]
[[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]

In [15]: print map(list, zip(*arr))              # built-in function.
[[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
```

8.1 Slight Improvement In Performance

we don’t use zip’s result as a list directly,(because izip does not materialize its result as a list in memory, but rather yields it one item at a time):

```
In [160]: newList
Out[160]: [[1, 4, 3], [5, 8, 7], [9, 12, 11]]
In [161]: import itertools
In [162]: test=itertools.izip(*newList)
In [163]: test
Out[163]: <itertools.izip at 0x2e4d368>
In [164]: next(test)
Out[164]: (1, 5, 9)

import itertools
print map(list, itertools.izip(*arr))
```

9 Getting A Value From Dictionary

You need to obtain a value from a dictionary, without having to handle an exception if the key you seek is not in the dictionary.

```
print d.get('key', 'not found')    # just reads d and never changes it
print d.pop('key', 'not found')    # does a get-and-remove
```

10 Adding An Entry To An Dictionary

Working with a dictionary d, you need to use the entry d[k] when it’s already present, or add a new value as d[k] when k isn’t yet a key in d. if k is not a key in the dictionary, the setdefault method assigns d[k]=v as a side effect,

```
In [17]: d={'a':1,'b':2}

In [18]: d
Out[18]: {'a': 1, 'b': 2}

In [19]: d.setdefault('c',[]).append(3)

In [20]: d
Out[20]: {'a': 1, 'b': 2, 'c': [3]}
```

This code is equivalent to more verbose approaches such as:

```
def addword(theIndex, word, pagenumber):
    try:
        theIndex[word].append(pagenumber)
    except KeyError:
        theIndex[word] = [pagenumber]
```

10.1 Discussion

format	feature	comment
d.get(k,v)	if k not a key, return v	
d.setdefault(k,v)	if k not a key, set d[k]=v	get-like function.
d.setdefault(k,v).append(v2)	more useful when value is list	meaningless when value is immutable

The third format, when the value is immutable,you must rebind the dictionary entry at theIndex[word] anyway. in case it have a original value, we'd better use get() method instead of setdefault()

```
def addword(theIndex, word, pagenumber):
    theIndex[word] = theIndex.get(word, [ ]) + [pagenumber]
```

11 Building A Dictionary Without Excessive Quoting

11 Building A Dictionary Without Excessive Quoting

you can avoid quoting them by calling dict with named-argument syntax:

```
data = dict(red=1, green=2, blue=3)       # it is neater than below format. with named-argument syntax
data = {'red': 1, 'green': 2, 'blue': 3}  # dictionary-display syntax, standard format with colons and braces.
```

by calling dict, you can avoid having to quote keys, when the keys are literal strings that happen to be **syntactically valid** for use as Python identifiers.

You cannot use this approach for keys such as the literal strings '12ba' or 'for':

- '12ba' starts with a digit
- 'for' happens to be a Python keyword, not an identifier

11.1 Discussion

1. dict(d) give another possibility. it return a copy of dict, just like d.copy()
2. mix up with zip() function,where the_keys is a sequence of keys and the_values a “parallel” sequence of corre- sponding values.

```
In [30]: d = dict(zip(('a','b','c'),(1,2,3)))

In [31]: d
Out[31]: {'a': 1, 'b': 2, 'c': 3}

In [32]: d = dict(zip(['a','b','c'],[1,2,3]))

In [33]: d
Out[33]: {'a': 1, 'b': 2, 'c': 3}

d = dict(itertools.izip(the_keys, the_values)) # is recommended,itertools.izip yields only one pair at a time.
```

11.2 Conclusion

So far we have 3 ways to create a dict. most of all pay attention to the third one, mixed up with function zip. we should decompose the procedure into two steps.

```
In [9]: test1=[1,2]

In [10]: test2=[3,4]

In [11]: d=dict(zip(test1,test2))

In [12]: d
Out[12]: {1: 3, 2: 4}

In [13]: zip(test1,test2)
Out[13]: [(1, 3), (2, 4)]

In [14]: dict((1,2))    # according to the result of zip(),I think this format will be ok, but it return error.
TypeError: cannot convert dictionary update sequence element #0 to a sequence

In [16]: dict([(1,2)])  # it should be like this.a list of tuple..
Out[16]: {1: 2}
# then compare the success and fault, you will find the success will used more general, because we always want
# update a list of something, not just only this tuple.you should realize (1,2) is a tuple.then you could found
# the next() of izip() funtion, it is a tuple. so as my predict, for the "discussion part" every time after nex
# the value should be changed to list first.
In [17]: dict(((1,2)))

In [32]: dict([(1,2),(3,4),(5,6)]) # covert from a list of tuple is more useful. so you give a tuple is not cor
Out[32]: {1: 2, 3: 4, 5: 6}
TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

12 Building A Dict From A List Of Alternating Keys And Values

First you should know what is "alternating keys and values", it is the task of getting a sequence of pairs from a flat sequence

```
In [19]: def dictFromList(keysAndValues):
   ....:     return dict(zip(keysAndValues[::2], keysAndValues[1::2]))
   ....:

In [21]: x=[1,2,3,4]

In [22]: dictFromList(x)
Out[22]: {1: 2, 3: 4}
```

if keysAndValues is a long sequence, all of this list construction activity can cost some performance.

```
def pairwise(iterable):
    itnext = iter(iterable).next
    while True:
        yield itnext(), itnext()
def dictFromSequence(seq):
    #  print pairwise(seq).next()
    return dict(pairwise(seq))
```

12.1 Dict(Zip(Seq,Seq))

Making a conclusion from the last 2 part, dict also could covert a **tuple generator** or **list of tuple** to a dictionary.

The generator part of this part, you would find the value is a tuple, just like the description in the last part's conclusion,so so as my predict, for the "discussion part" every time after next(), the value should be changed to list first.

13 Extracting A Subset From A Dictionary

Giving you a set of keys then got the corresponding pairs.

```
def sub_dict(somedict, somekeys, default=None):            # leaving the original dict intact
    return dict([ (k, somedict.get(k, default)) for k in somekeys ])

def sub_dict_remove(somedict, somekeys, default=None):     # remove matched item the original dict
    return dict([ (k, somedict.pop(k, default)) for k in somekeys ])

In [41]: d = {'a': 5, 'b': 6, 'c': 7}

In [42]: print sub_dict(d, 'ab'), d
{'a': 5, 'b': 6} {'a': 5, 'c': 7, 'b': 6}

In [43]: print sub_dict_remove(d, 'ab'), d
{'a': 5, 'b': 6} {'c': 7}
```

you can use generator expressions, instead of list comprehensions, Just change the syntax of the calls to dict, from dict([. . .]) to dict(. . .)

14 Inverting A Dictionary

mapping each value to its key.

```
def invert_dict(d):
    return dict([ (v, k) for k, v in d.iteritems( ) ])

# for large dictionary.it's faster to use generator izip
from itertools import izip
def invert_dict_fast(d):
    return dict(izip(d.itervalues( ), d.iterkeys( )))

# emit:
In [2]: d=dict(red=2,yellow=3,blue=4)

In [3]: invert_dict(d)
Out[3]: {2: 'red', 3: 'yellow', 4: 'blue'}

In [4]: d=dict(red=2,yellow=3,blue=3)

In [5]: invert_dict(d)      # the duplicate will be omitted.
Out[5]: {2: 'red', 3: 'yellow'}
```

If the value in dict d is not unique.the functions shown in this recipe still construct, even in such cases, a “pseudo-inverse”. so x is the true inverse of d if and only if len(x)==len(d).

15 Associating Multiple Value With Each Key In A Dict

1. one-to-one mapping to one-to-multiple mapping.
2. Each approach relies on the **setdefault** method of a dictionary,
3. **d1[key].remove(value)** to remove the value.

```
# value is a list.
In [11]: d1 = { }
In [12]: d1 .setdefault('a', [ ]).append(2)
In [13]: d1
Out[13]: {'a': [2]}
In [17]: d1 .setdefault('a', [ ]).append(3)
In [18]: d1
Out[18]: {'a': [2, 3]}
In [20]: d1['a'].remove(2)
In [21]: d1
Out[21]: {'a': [3]}
# value is a dict. no remove method .. because dict don't have remove method.
In [9]: d2.setdefault('d', { })[1] = 1
In [10]: d2
Out[10]: {'d': {1: 1}}
In [26]: d2.setdefault('d', { })[2] = 2
In [27]: d2
Out[27]: {'d': {1: 1, 2: 2}}
# value is a set
In [14]: d3 = { }
In [15]: d3.setdefault('c', set( )).add(4)
In [16]: d3
Out[16]: {'c': set([4])}
In [22]: d3.setdefault('c', set( )).add(5)
In [23]: d3
Out[23]: {'c': set([4, 5])}
In [24]: d3['c'].remove(4)
In [25]: d3
Out[25]: {'c': set([5])}
```

16 Using Dict To Dispatching Method Or Function

In Python, dictionaries, and the fact that functions are first-class objects (in particular, functions can be values in a dictionary)

```
tokenDict = {"cat": deal_with_a_cat} # set a function value to a key.
cat()   # invoke this function.
```

17 Finding Unions And Intersections Of Dict

Given two dictionaries,

- Intersection: you need to find the set of keys that are in both dictionaries
- Union: the set of keys that are in either dictionary

You can use the type of dict or set. However, it’s best to use type set instead of representing sets by dictionaries.because we just care about the keys,not the corresponding values. Method 1: using dict.

```
# build the dict by calls to dict.fromkeys()
In [23]: d=dict(a=3,b=4)
In [24]: d
Out[24]: {'a': 3, 'b': 4}
In [25]: a=dict.fromkeys(d)    # update the value to none. we don't care value.
In [26]: a
Out[26]: {'a': None, 'b': None}
In [27]: e=dict(a=8,c=9)
In [28]: b=dict.fromkeys(e)
In [29]: dict(a,**b)          # get the union through dict.
Out[29]: {'a': None, 'b': None, 'c': None}

if len(a) < len(b):           # get inter through dict. in for is the short list,for speed.
    inter = dict.fromkeys([x for x in a if x in b])
else:
    inter = dict.fromkeys([x for x in b if x in a])
```

Method 2: using set. first extract the key to a set,because we just care key.

```
In [14]: d
Out[14]: {'a': 3, 'b': 4}

In [15]: set(d)
Out[15]: set(['a', 'b'])

union = a | b
inter = a & b
union = a.union(b)  # don’t need to turn both dicts into sets: just one of them(a)
inter = a.intersection(b)
a.symmetric_difference(b)  # in either a or b but not both a^b
```

17.1 Discussion

- Dict phones : maps names to phone number
- Dict address : maps names to address

print all names for which you know both address and phone number

```
for name in set(phones) & set(addresses):
    print name, phones[name], addresses[name]
```

```
for name in set(phones).intersection(addresses):   # address no need to change to set.
    print name, phones[name], addresses[name]
```

Date: 2013-01-11T00:32+0900
Author: ryu
Org version 7.9.2 with Emacs version 24
Validate XHTML 1.0