Copying a Python List

David Branner
Hacker School, New York

20 February 2014

A well-attested feature of Python lists: they are normally copied by reference:

A well-attested feature of Python lists: they are normally copied by reference:

```
>>> x = [1, 2, 3]
>>>
```

A well-attested feature of Python lists: they are normally copied by reference:

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

A well-attested feature of Python lists: they are normally copied by reference:

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(4)
>>>
```

A well-attested feature of Python lists: they are normally copied by reference:

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(4)
>>> x
```

A well-attested feature of Python lists: they are normally copied by reference:

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>>
```

A well-attested feature of Python lists: they are normally copied by reference:

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
```

A well-attested feature of Python lists: they are normally copied by reference:

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3, 4]
>>>
```

A well-attested feature of Python lists: they are normally copied by reference:

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3, 4]
>>> # y changes because it isn't independent of x
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>>
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>> x = [1, 2, 3]
>>>
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>> x = [1, 2, 3]
>>> y = copy.deepcopy(x)
>>>
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>> x = [1, 2, 3]
>>> y = copy.deepcopy(x)
>>> x.append(4)
>>>
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>> x = [1, 2, 3]
>>> y = copy.deepcopy(x)
>>> x.append(4)
>>> x
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>> x = [1, 2, 3]
>>> y = copy.deepcopy(x)
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>>
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>> x = [1, 2, 3]
>>> y = copy.deepcopy(x)
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>> x = [1, 2, 3]
>>> y = copy.deepcopy(x)
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3]
>>>
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>> x = [1, 2, 3]
>>> y = copy.deepcopy(x)
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3]
>>>
```

As a way of avoiding copying by reference, people usually recommend using the `copy.deepcopy` method:

```
>>> import copy
>>> x = [1, 2, 3]
>>> y = copy.deepcopy(x)
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3]
>>> # As a deep copy, y is independent of x.
```

An alternate (and much faster) way to make a shallow copy of a list is to make a complete slice of it:

An alternate (and much faster) way to make a shallow copy of a list is to make a complete slice of it:

```
>>> x = [1, 2, 3]
>>>
```

An alternate (and much faster) way to make a shallow copy of a list is to make a complete slice of it:

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>>
```

An alternate (and much faster) way to make a shallow copy of a list is to make a complete slice of it:

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> y
[1, 2, 3]
>>>
```

An alternate (and much faster) way to make a shallow copy of a list is to make a complete slice of it:

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> y
[1, 2, 3]
>>> x.append(4)
>>>
```

An alternate (and much faster) way to make a shallow copy of a list is to make a complete slice of it:

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> y
[1, 2, 3]
>>> x.append(4)
>>> x
```

An alternate (and much faster) way to make a shallow copy of a list is to make a complete slice of it:

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> y
[1, 2, 3]
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>>
```

An alternate (and much faster) way to make a shallow copy of a list is to make a complete slice of it:

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> y
[1, 2, 3]
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
```

An alternate (and much faster) way to make a shallow copy of a list is to make a complete slice of it:

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> y
[1, 2, 3]
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3]
>>>
```

Slicing a list creates an independent list.

Slicing a list creates an independent list.

```
>>> x = [1, 2, 3]
>>>
```

Slicing a list creates an independent list.

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>>
```

Slicing a list creates an independent list.

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> x == y
```

Slicing a list creates an independent list.

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> x == y
True
>>>
```

Slicing a list creates an independent list.

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> x == y
True
>>> id(x) == id(y) # id() returns memory location
```

Slicing a list creates an independent list.

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> x == y
True
>>> id(x) == id(y) # id() returns memory location
False
>>>
```

Slicing a list creates an independent list.

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> x == y
True
>>> id(x) == id(y) # id() returns memory location
False
>>> # Not the same memory location;
>>> # x and y are different objects
```

`Deepcopy` is much, much slower than slicing.

`Deepcopy` is much, much slower than slicing.

Timings (`timeit`):

	time (μsec)	
	CPython 2.7.5	
n	list[:]	deepcopy
10	0.121	15.9
100	0.375	72
1000	2.01	631
10000	23.3	5650
100000	247	57500
1000000	2780	569000

`Deepcopy` is much, much slower than slicing.

Timings (`timeit`):

	time (μsec)	
	CPython 2.7.5	
n	list[:]	deepcopy
10	0.121	15.9
100	0.375	72
1000	2.01	631
10000	23.3	5650
100000	247	57500
1000000	2780	569000

Both appear to be running in linear time.

`Deepcopy` is much, much slower than slicing.

Timings (`timeit`):

	time (μsec)	
	CPython 2.7.5	
n	list[:]	deepcopy
10	0.121	15.9
100	0.375	72
1000	2.01	631
10000	23.3	5650
100000	247	57500
1000000	2780	569000

Both appear to be running in linear time. (A tenfold increase in list-length brings about a ten-fold increase in running time.)

Not only the standard (“reference”) implementation, CPython, does things this way, but so do competitors PyPy and Jython.

Not only the standard (“reference”) implementation, CPython, does things this way, but so do competitors PyPy and Jython.

Timings (`timeit`):

	time (μsec)					
	CPython 2.7.5		PyPy ~ Python 2.7.3		Jython ~ Python 2.5.2	
n	list[:]	deepcopy				
10	0.121	15.9				
100	0.375	72				
1000	2.01	631				
10000	23.3	5650				
100000	247	57500				
1000000	2780	569000				

Not only the standard (“reference”) implementation, CPython, does things this way, but so do competitors PyPy and Jython.

Timings (`timeit`):

	time (μsec)					
	CPython 2.7.5		PyPy ~ Python 2.7.3		Jython ~ Python 2.5.2	
n	list[:]	deepcopy	list[:]	deepcopy		
10	0.121	15.9	0.0373	2.36		
100	0.375	72	0.0847	18.2		
1000	2.01	631	0.621	177		
10000	23.3	5650	8.14	3820		
100000	247	57500	387	63400		
1000000	2780	569000	5450	1410000		

Not only the standard (“reference”) implementation, CPython, does things this way, but so do competitors PyPy and Jython.

Timings (`timeit`):

	time (μsec)					
	CPython 2.7.5		PyPy ~ Python 2.7.3		Jython ~ Python 2.5.2	
n	list[:]	deepcopy	list[:]	deepcopy	list[:]	deepcopy
10	0.121	15.9	0.0373	2.36	0.0756	37
100	0.375	72	0.0847	18.2	0.245	300
1000	2.01	631	0.621	177	1.91	1300
10000	23.3	5650	8.14	3820	18.2	13000
100000	247	57500	387	63400	207	163000
1000000	2780	569000	5450	1410000	2450	2290000

Not only the standard (“reference”) implementation, CPython, does things this way, but so do competitors PyPy and Jython.

Timings (`timeit`):

	time (µsec)					
	CPython 2.7.5		PyPy ~ Python 2.7.3		Jython ~ Python 2.5.2	
n	list[:]	deepcopy	list[:]	deepcopy	list[:]	deepcopy
10	0.121	15.9	0.0373	2.36	0.0756	37
100	0.375	72	0.0847	18.2	0.245	300
1000	2.01	631	0.621	177	1.91	1300
10000	23.3	5650	8.14	3820	18.2	13000
100000	247	57500	387	63400	207	163000
1000000	2780	569000	5450	1410000	2450	2290000

(PyPy is fast at small values of *n*, but then slows down and does not seem to be running consistently in linear time at some higher values of *n*.)

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>>
```

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>> x.append(x)
>>>
```

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>> x.append(x)
>>> x
[[...]]
>>>
```

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>> x.append(x)
>>> x
[[...]]
>>> y = copy.deepcopy(x)
>>>
```

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>> x.append(x)
>>> x
[[...]]
>>> y = copy.deepcopy(x)
>>> y
```

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>> x.append(x)
>>> x
[[...]]
>>> y = copy.deepcopy(x)
>>> y
[[...]]
>>>
```

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>> x.append(x)
>>> x
[[...]]
>>> y = copy.deepcopy(x)
>>> y
[[...]]
>>> z = x[:]
>>>
```

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>> x.append(x)
>>> x
[[...]]
>>> y = copy.deepcopy(x)
>>> y
[[...]]
>>> z = x[:]
>>> z
```

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>> x.append(x)
>>> x
[[...]]
>>> y = copy.deepcopy(x)
>>> y
[[...]]
>>> z = x[:]
>>> z
[[[...]]]
>>>
```

Slicing is adequate for most occasions when a “shallow” copy of a list is needed.

It returns a different result from `deepcopy` on some occasions, e.g., when recursion is involved:

```
>>> x = []
>>> x.append(x)
>>> x
[[...]]
>>> y = copy.deepcopy(x)
>>> y
[[...]]
>>> z = x[:]
>>> z
[[[...]]]
>>>
```

End