Examining the Identity of a "Whole-Sliced" Python Sequence

David Prager Branner
Hacker School

presented at PyGotham, New York – 20140816

Abstract

This presentation examines the behavior of IDs (memory locations) of “whole slices” of Python sequences, and particularly whether those IDs are the same or different for different copies of those slices.

In the standard implementation (CPython), whether or not the IDs are the same depends on which sequence is involved. The pattern for the standard implementation is itself different from the patterns for two other major implementations, PyPy and Jython. ID behavior of whole-sliced sequences is most consistent and predictable in Jython, where whole-slicing a sequence produces a new object in each and every case examined.

“Whole-slicing” is a quick hack for making a deep copy

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

"Whole-slicing" is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

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

"Whole-slicing" is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
```

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
```

"Whole-slicing" is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
```

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
```

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
```

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
```

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
```

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

"Whole-slicing" is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
```

"Whole-slicing" is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
```

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
```

"Whole-slicing" is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'泥古不化'
```

"Whole-slicing" is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'泥古不化'
>>> b
```

“Whole-slicing” is a quick hack for making a deep copy – a copy of the actual values – of an entire sequence such as a list whose elements would otherwise be copied by reference:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'泥古不化'
>>> b
[1, 2, 3] # unaffected by the change in “a”
```

“Copying” by reference does not actually produce a new object

“Copying” by reference does not actually produce a new object, whereas slicing does.

“Copying” by reference does not actually produce a new object, whereas slicing does. We can determine whether two objects are the same or not

“Copying” by reference does not actually produce a new object, whereas slicing does. We can determine whether two objects are the same or not, at the moment they are being compared

“Copying” by reference does not actually produce a new object, whereas slicing does. We can determine whether two objects are the same or not, at the moment they are being compared, by testing the congruence of their “identities,” returned by the built-in `id()` function.

"Copying" by reference does not actually produce a new object, whereas slicing does. We can determine whether two objects are the same or not, at the moment they are being compared, by testing the congruence of their "identities," returned by the built-in `id()` function.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
>>> id(a) == id(b)
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'泥古不化'
>>> b
[1, 2, 3] # unaffected by the change in "a"
>>> id(a) == id(b)
```

"Copying" by reference does not actually produce a new object, whereas slicing does. We can determine whether two objects are the same or not, at the moment they are being compared, by testing the congruence of their "identities," returned by the built-in `id()` function.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
>>> id(a) == id(b)
True
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'泥古不化'
>>> b
[1, 2, 3] # unaffected by the change in "a"
>>> id(a) == id(b)
False
```

"Copying" by reference does not actually produce a new object, whereas slicing does. We can determine whether two objects are the same or not, at the moment they are being compared, by testing the congruence of their "identities," returned by the built-in `id()` function.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
>>> id(a) == id(b)
True
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'泥古不化'
>>> b
[1, 2, 3] # unaffected by the change in "a"
>>> id(a) == id(b)
False
```

"Copying" by reference does not actually produce a new object, whereas slicing does. We can determine whether two objects are the same or not, at the moment they are being compared, by testing the congruence of their "identities," returned by the built-in `id()` function.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
>>> id(a) == id(b)
True
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'泥古不化'
>>> b
[1, 2, 3] # unaffected by the change in "a"
>>> id(a) == id(b)
False
```

The `id()` function returns an integer that (in CPython) is the memory address of the argument.

"Copying" by reference does not actually produce a new object, whereas slicing does. We can determine whether two objects are the same or not, at the moment they are being compared, by testing the congruence of their "identities," returned by the built-in `id()` function.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = u'應變'
>>> b
[u'\u61c9\u8b8a', 2, 3]
>>> id(a) == id(b)
True
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'泥古不化'
>>> b
[1, 2, 3] # unaffected by the change in "a"
>>> id(a) == id(b)
False
```

The `id()` function returns an integer that (in CPython) is the memory address of the argument. Different objects that coexist at some moment have different memory addresses.

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> import array
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> import array
>>> id(array.array('i', [1, 2, 3])) == id(array.array('i', [1, 2, 3])[:])       # array
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> import array
>>> id(array.array('i', [1, 2, 3])) == id(array.array('i', [1, 2, 3])[:])       # array
False
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> import array
>>> id(array.array('i', [1, 2, 3])) == id(array.array('i', [1, 2, 3])[:])       # array
False
>>> id(bytearray('123')) == id(bytearray('123')[:])                             # bytearray
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> import array
>>> id(array.array('i', [1, 2, 3])) == id(array.array('i', [1, 2, 3])[:])        # array
False
>>> id(bytearray('123')) == id(bytearray('123')[:])                              # bytearray
False
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> import array
>>> id(array.array('i', [1, 2, 3])) == id(array.array('i', [1, 2, 3])[:])       # array
False
>>> id(bytearray('123')) == id(bytearray('123')[:])                             # bytearray
False
```

Let's summarize what we know in a table:

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> import array
>>> id(array.array('i', [1, 2, 3])) == id(array.array('i', [1, 2, 3])[:])      # array
False
>>> id(bytearray('123')) == id(bytearray('123')[:])                            # bytearray
False
```

Let's summarize what we know in a table:

question	*list*	*array*	*bytearray*
id(object) == id(object[:])	False	False	False

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> import array
>>> id(array.array('i', [1, 2, 3])) == id(array.array('i', [1, 2, 3])[:])      # array
False
>>> id(bytearray('123')) == id(bytearray('123')[:])                            # bytearray
False
```

Let's summarize what we know in a table:

question	*list*	*array*	*bytearray*
id(object) == id(object[:])	False	False	False

(Here I use "object" to represent a literal object rather than a variable representing it.)

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> id(buffer('123')) == id(buffer('123')[:])       # buffer
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> id(buffer('123')) == id(buffer('123')[:])       # buffer
False
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> id(buffer('123')) == id(buffer('123')[:])       # buffer
False
>>> id((1, 2, 3)) == id((1, 2, 3)[:])               # tuple
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> id(buffer('123')) == id(buffer('123')[:])       # buffer
False
>>> id((1, 2, 3)) == id((1, 2, 3)[:])               # tuple
False
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> id(buffer('123')) == id(buffer('123')[:])      # buffer
False
>>> id((1, 2, 3)) == id((1, 2, 3)[:])              # tuple
False
```

question	*list*	*array*	*bytearray*	*buffer*	*tuple*
id(object) == id(object[:])	False	False	False	False	False

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities:

```
>>> id(buffer('123')) == id(buffer('123')[:])      # buffer
False
>>> id((1, 2, 3)) == id((1, 2, 3)[:])              # tuple
False
```

question	list	array	bytearray	buffer	tuple
id(object) == id(object[:])	False	False	False	False	False

But it may be surprising that not all do...

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities. But it may be surprising that not all do:

```
>>> id('123') == id('123'[:])   # string
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities. But it may be surprising that not all do:

```
>>> id('123') == id('123'[:])   # string
True
```

It isn't surprising that some other Python sequence types display the same behavior as a list when comparing the `id()` of the original object with a beginning-to-end slice – the object and its "whole slice" are different entities. But it may be surprising that not all do:

```
>>> id('123') == id('123'[:])   # string
True
```

question	*list*	*array*	*bytearray*	*tuple*	*buffer*	*string*
id(object) == id(object[:])	False	False	False	False	False	**True**

And it turns out that a number of identity questions about "whole slices" of sequences have different answers depending on the sequence type and the way we ask the question:

And it turns out that a number of identity questions about "whole slices" of sequences have different answers depending on the sequence type and the way we ask the question:

```
>>> id([1, 2, 3][:]) == id([1, 2, 3][:]) # Are concurrent whole slices of a list one object?
False
```

And it turns out that a number of identity questions about "whole slices" of sequences have different answers depending on the sequence type and the way we ask the question:

```
>>> id([1, 2, 3][:]) == id([1, 2, 3][:]) # Are concurrent whole slices of a list one object?
False
>>> id(array.array('i', [1, 2, 3])[:]) == id(array.array('i', [1, 2, 3])[:])  # ditto, array
False
```

And it turns out that a number of identity questions about "whole slices" of sequences have different answers depending on the sequence type and the way we ask the question:

```
>>> id([1, 2, 3][:]) == id([1, 2, 3][:]) # Are concurrent whole slices of a list one object?
False
>>> id(array.array('i', [1, 2, 3])[:]) == id(array.array('i', [1, 2, 3])[:])  # ditto, array
False
>>> id(bytearray('123')[:]) == id(bytearray('123')[:])                        # ditto, bytearray
False
```

And it turns out that a number of identity questions about "whole slices" of sequences have different answers depending on the sequence type and the way we ask the question:

```
>>> id([1, 2, 3][:]) == id([1, 2, 3][:]) # Are concurrent whole slices of a list one object?
False
>>> id(array.array('i', [1, 2, 3])[:]) == id(array.array('i', [1, 2, 3])[:])  # ditto, array
False
>>> id(bytearray('123')[:]) == id(bytearray('123')[:])                        # ditto, bytearray
False
>>> id(buffer('123')[:]) == id(buffer('123')[:])                              # ditto, buffer
True
```

And it turns out that a number of identity questions about "whole slices" of sequences have different answers depending on the sequence type and the way we ask the question:

```
>>> id([1, 2, 3][:]) == id([1, 2, 3][:]) # Are concurrent whole slices of a list one object?
False
>>> id(array.array('i', [1, 2, 3])[:]) == id(array.array('i', [1, 2, 3])[:])  # ditto, array
False
>>> id(bytearray('123')[:]) == id(bytearray('123')[:])                        # ditto, bytearray
False
>>> id(buffer('123')[:]) == id(buffer('123')[:])                              # ditto, buffer
True
>>> id((1, 2, 3)[:]) == id((1, 2, 3)[:])                                     # ditto, tuple
False
```

And it turns out that a number of identity questions about "whole slices" of sequences have different answers depending on the sequence type and the way we ask the question:

```
>>> id([1, 2, 3][:]) == id([1, 2, 3][:]) # Are concurrent whole slices of a list one object?
False
>>> id(array.array('i', [1, 2, 3])[:]) == id(array.array('i', [1, 2, 3])[:])  # ditto, array
False
>>> id(bytearray('123')[:]) == id(bytearray('123')[:])                        # ditto, bytearray
False
>>> id(buffer('123')[:]) == id(buffer('123')[:])                              # ditto, buffer
True
>>> id((1, 2, 3)[:]) == id((1, 2, 3)[:])                                     # ditto, tuple
False
>>> id('123'[:]) == id('123'[:])                                             # ditto, string
True
```

We can summarize what we have shown so far in a table:

We can summarize what we have shown so far in a table:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False	**True**
id(object[:]) == id(object[:])	False	**True**	False	**True**

We can summarize what we have shown so far in a table:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False	**True**
id(object[:]) == id(object[:])	False	**True**	False	**True**

We should ask right now whether mutability is sufficient to explain this pattern of behaviors.

We can summarize what we have shown so far in a table:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False	**True**
id(object[:]) == id(object[:])	False	**True**	False	**True**
mutable?	**yes**	**can be**	no	no

Mutability alone is apparently not sufficient to explain this pattern of behaviors.

We can summarize what we have shown so far in a table:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False	**True**
id(object[:]) == id(object[:])	False	**True**	False	**True**
mutable?	**yes**	**can be**	no	no

Mutability alone is apparently not sufficient to explain this pattern of behaviors. In addition to mutability, there is the question of "changeability": we can get different results for a tuple if it contains a mutable object:

We can summarize what we have shown so far in a table:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False	**True**
id(object[:]) == id(object[:])	False	**True**	False	**True**
mutable?	**yes**	**can be**	no	no

Mutability alone is apparently not sufficient to explain this pattern of behaviors. In addition to mutability, there is the question of "changeability": we can get different results for a tuple if it contains a mutable object:

```
>>> x = (1, 2, 3) # integers are immutable
>>> id((1, 2, 3)) == id((1, 2, 3)[:])
False
>>> id((1, 2, 3)[:]) == id((1, 2, 3)[:])
False
```

We can summarize what we have shown so far in a table:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False	**True**
id(object[:]) == id(object[:])	False	**True**	False	**True**
mutable?	**yes**	**can be**	no	no

Mutability alone is apparently not sufficient to explain this pattern of behaviors. In addition to mutability, there is the question of "changeability": we can get different results for a tuple if it contains a mutable object:

```
>>> x = (1, 2, 3) # integers are immutable
>>> id((1, 2, 3)) == id((1, 2, 3)[:])
False
>>> id((1, 2, 3)[:]) == id((1, 2, 3)[:])
False
```

```
>>> a = [6, 7, 8] # lists are mutable
>>> x = (a, 2, 3) # so "x" is "changeable"
>>> id((a, 2, 3)) == id((a, 2, 3)[:])
True
>>> id((a, 2, 3)[:]) == id((a, 2, 3)[:])
True
```

We can summarize what we have shown so far in a table:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**True**
id(object[:]) == id(object[:])	False	**True**	False but...	**True**
mutable?	**yes**	**can be**	no	no

Mutability alone is apparently not sufficient to explain this pattern of behaviors. In addition to mutability, there is the question of "changeability": we can get different results for a tuple if it contains a mutable object:

```
>>> x = (1, 2, 3) # integers are immutable
>>> id((1, 2, 3)) == id((1, 2, 3)[:])
False
>>> id((1, 2, 3)[:]) == id((1, 2, 3)[:])
False
```

```
>>> a = [6, 7, 8] # lists are mutable
>>> x = (a, 2, 3) # so "x" is "changeable"
>>> id((a, 2, 3)) == id((a, 2, 3)[:])
True
>>> id((a, 2, 3)[:]) == id((a, 2, 3)[:])
True
```

A variable to which an object is assigned also has a different pattern of slice-identity among the various types of sequences:

A variable to which an object is assigned also has a different pattern of slice-identity among the various types of sequences:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**True**
id(object[:]) == id(object[:])	False	**True**	False but...	**True**
var = object id(var[:]) == id(var[:])	**True**	**True**	**True**	**True**
var2 = var[:] id(var[:]) == id(var2[:])	**True**	False	**True**	**True**

A variable to which an object is assigned also has a different pattern of slice-identity among the various types of sequences:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**True**
id(object[:]) == id(object[:])	False	**True**	False but...	**True**
var = object id(var[:]) == id(var[:])	**True**	**True**	**True**	**True**
var2 = var[:] id(var[:]) == id(var2[:])	**True**	False	**True**	**True**

There are some other interesting inconsistencies of this sort documented in Appendix 2, but I omit them here for the sake of brevity.

It seems random.

It seems random. Is it?

It seems random. Is it? Or is there underlying order, carefully prescribed, in this diverse behavior?

It seems random. Is it? Or is there underlying order, carefully prescribed, in this diverse behavior?

In particular, it would make sense for a deep copy of an object to be an entity different from the original object

It seems random. Is it? Or is there underlying order, carefully prescribed, in this diverse behavior?

In particular, it would make sense for a deep copy of an object to be an entity different from the original object and different also from any other deep copy of that object.

It seems random. Is it? Or is there underlying order, carefully prescribed, in this diverse behavior?

In particular, it would make sense for a deep copy of an object to be an entity different from the original object and different also from any other deep copy of that object. But Python objects behave inconsistently in these respects:

It seems random. Is it? Or is there underlying order, carefully prescribed, in this diverse behavior?

In particular, it would make sense for a deep copy of an object to be an entity different from the original object and different also from any other deep copy of that object. But Python objects behave inconsistently in these respects:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**True**
id(object[:]) == id(object[:])	False	**True**	False but...	**True**

It seems random. Is it? Or is there underlying order, carefully prescribed, in this diverse behavior?

In particular, it would make sense for a deep copy of an object to be an entity different from the original object and different also from any other deep copy of that object. But Python objects behave inconsistently in these respects:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**True**
id(object[:]) == id(object[:])	False	**True**	False but...	**True**

With whole-slice variables, acting on an indeterminate datatype could lead to expected results:

It seems random. Is it? Or is there underlying order, carefully prescribed, in this diverse behavior?

In particular, it would make sense for a deep copy of an object to be an entity different from the original object and different also from any other deep copy of that object. But Python objects behave inconsistently in these respects:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**True**
id(object[:]) == id(object[:])	False	**True**	False but...	**True**

With whole-slice variables, acting on an indeterminate datatype could lead to expected results:

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
var = object id(var[:]) == id(var[:])	**True**	**True**	**True**	**True**
var2 = var[:] id(var[:]) == id(var2[:])	**True**	False	**True**	**True**

Internals can explain these inconsistencies

Internals can explain these inconsistencies, but high level distribution patterns are more eloquent:

Internals can explain these inconsistencies, but high level distribution patterns are more eloquent: The implementations CPython, PyPy, and Jython (see Appendix 1 for versions) return different patterns of True and False vis-à-vis these questions

Internals can explain these inconsistencies, but high level distribution patterns are more eloquent: The implementations CPython, PyPy, and Jython (see Appendix 1 for versions) return different patterns of True and False vis-à-vis these questions (only Jython is fully consistent):

Internals can explain these inconsistencies, but high level distribution patterns are more eloquent: The implementations CPython, PyPy, and Jython (see Appendix 1 for versions) return different patterns of True and False vis-à-vis these questions (only Jython is fully consistent):

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**C** Py/J
id(object[:]) == id(object[:])	False	**C/Py** J	False but...	**C** Py/J
var = object id(var[:]) == id(var[:])	**C** Py/J	**C/Py** J	**C** Py/J	**C/Py** J
var2 = var[:] id(var[:]) == id(var2[:])	**C** Py/J	**Py** C/J	**C** Py/J	**C/Py** J

Key: **blue & bold = True**; red & non-bold = False

C: CPython; Py: PyPy; J: Jython.

Internals can explain these inconsistencies, but high level distribution patterns are more eloquent: The implementations CPython, PyPy, and Jython (see Appendix 1 for versions) return different patterns of True and False vis-à-vis these questions (only Jython is fully consistent):

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**C** Py/J
id(object[:]) == id(object[:])	False	**C/Py** J	False but...	**C** Py/J
var = object id(var[:]) == id(var[:])	**C** Py/J	**C/Py** J	**C** Py/J	**C/Py** J
var2 = var[:] id(var[:]) == id(var2[:])	**C** Py/J	**Py** C/J	**C** Py/J	**C/Py** J

Key: **blue & bold = True**; red & non-bold = False

C: CPython; Py: PyPy; J: Jython.

The point: The behavior of the `id()` function with sliced sequences is

Internals can explain these inconsistencies, but high level distribution patterns are more eloquent: The implementations CPython, PyPy, and Jython (see Appendix 1 for versions) return different patterns of True and False vis-à-vis these questions (only Jython is fully consistent):

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**C** Py/J
id(object[:]) == id(object[:])	False	**C/Py** J	False but...	**C** Py/J
var = object id(var[:]) == id(var[:])	**C** Py/J	**C/Py** J	**C** Py/J	**C/Py** J
var2 = var[:] id(var[:]) == id(var2[:])	**C** Py/J	**Py** C/J	**C** Py/J	**C/Py** J

Key: **blue & bold = True**; red & non-bold = False

C: CPython; Py: PyPy; J: Jython.

The point: The behavior of the `id()` function with sliced sequences is

1) not uniform with respect to the various sequences in the standard implementation

Internals can explain these inconsistencies, but high level distribution patterns are more eloquent: The implementations CPython, PyPy, and Jython (see Appendix 1 for versions) return different patterns of True and False vis-à-vis these questions (only Jython is fully consistent):

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**C** Py/J
id(object[:]) == id(object[:])	False	**C/Py** J	False but...	**C** Py/J
var = object id(var[:]) == id(var[:])	**C** Py/J	**C/Py** J	**C** Py/J	**C/Py** J
var2 = var[:] id(var[:]) == id(var2[:])	**C** Py/J	**Py** C/J	**C** Py/J	**C/Py** J

Key: **blue & bold = True**; red & non-bold = False

C: CPython; Py: PyPy; J: Jython.

The point: The behavior of the `id()` function with sliced sequences is

1) not uniform with respect to the various sequences in the standard implementation;

2) not uniform among the three main implementations

Internals can explain these inconsistencies, but high level distribution patterns are more eloquent: The implementations CPython, PyPy, and Jython (see Appendix 1 for versions) return different patterns of True and False vis-à-vis these questions (only Jython is fully consistent):

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**C** Py/J
id(object[:]) == id(object[:])	False	**C/Py** J	False but...	**C** Py/J
var = object id(var[:]) == id(var[:])	**C** Py/J	**C/Py** J	**C** Py/J	**C/Py** J
var2 = var[:] id(var[:]) == id(var2[:])	**C** Py/J	**Py** C/J	**C** Py/J	**C/Py** J

Key: **blue & bold = True**; red & non-bold = False

C: CPython; Py: PyPy; J: Jython.

The point: The behavior of the `id()` function with sliced sequences is

1) not uniform with respect to the various sequences in the standard implementation;

2) not uniform among the three main implementations;

3) most uniform in Jython, which assigns a distinct ID to each full-sliced sequence.

劇終

This study was done at Hacker School, New York.
Thanks to Amber Wilcox-Hearn of Hacker School for a clarifying question on an earlier version of this talk.

Appendix 1: For reference here is how "identity" is defined in the three implementations:

- CPython: "Return the 'identity' of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value. CPython implementation detail: **This is the address of the object in memory.**" https://docs.python.org/2.7/library/functions.html?#id. Python 2.7.8 (default, Jul 2 2014, 10:14:46) [GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin

- PyPy: "**Using the default GC (called minimark), the built-in function `id()` [of PyPy] works like it does in CPython.** With other GCs it returns numbers that are not real addresses (because an object can move around several times) and calling it a lot can lead to performance problem." http://pypy.readthedocs.org/en/latest/cpython_differences.html Python 2.7.6 (32f35069a16d, Jun 06 2014, 20:12:47) [PyPy 2.3.1 with GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.79)] on darwin

- Jython: "Return the 'identity' of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value. (Implementation note: **this is the address of the object.**)" http://www.jython.org/docs/library/functions.html Jython 2.7b2 (default:a5bc0032cf79+, Apr 22 2014, 21:20:17) [Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on java1.7.0_51

Appendix 2: A fuller list of examples.

question	*list, array, bytearray*	*buffer*	*tuple*	*string*
id(object) == id(object[:])	False	False	False but...	**C** Py/J
id(object[:]) == id(object[:])	False	**C/Py** J	False but...	**C** Py/J
var = object id(var) == id(var[:])	False	False	**C** Py/J	**C/Py** J
var2 = var[:] id(var) == id(var2)	False	False	**C** Py/J	**C/Py** J
id(var[:]) == id(var[:])	**C** Py/J	**C/Py** J	**C** Py/J	**C/Py** J
id(var[:]) == id(var2[:])	**C** Py/J	**Py** C/J	**C** Py/J	**C/Py** J
id(var) == id(object)	False	False	False	**C** Py/J
id(var2) == id(object)	False	False	False	**C** Py/J
id(var[:]) == id(object)	**C** Py/J	False	False	**C** Py/J

Key: **blue & bold = True**; red & non-bold = False

C: CPython; Py: PyPy; J: Jython.

Appendix 3: Another interesting feature is that CPython alternates the IDs of a sliced object and a sliced variable differently if they are simply printed rather than in the same comparison:

>>> def test_list():	>>> def test_list():
... x = [1, 2, 3][:]	... print id([1, 2, 3][:])
... print id(x[:])	... print id([1, 2, 3][:])
... print id(x[:])	... print id([1, 2, 3][:])
... print id(x[:])	... print id([1, 2, 3][:])
... print id(x[:])	
>>> test_list()	>>> test_list()
4451744728	**4451745160**
4451744728	**4451676816**
4451744728	**4451745160**
4451744728	**4451676816**

Appendix 3: Another interesting feature is that CPython alternates the IDs of a sliced object and a sliced variable differently if they are simply printed rather than in the same comparison:

>>> def test_list():	>>> def test_list():
... x = [1, 2, 3][:]	... print id([1, 2, 3][:])
... print id(x[:])	... print id([1, 2, 3][:])
... print id(x[:])	... print id([1, 2, 3][:])
... print id(x[:])	... print id([1, 2, 3][:])
... print id(x[:])	
>>> test_list()	>>> test_list()
4451744728	**4451745160**
4451744728	**4451676816**
4451744728	**4451745160**
4451744728	**4451676816**

For a list, the literal object uses two alternating memory addresses in this example, while a variable uses the same memory address.

Appendix 3: Another interesting feature is that CPython alternates the IDs of a sliced object and a sliced variable differently if they are simply printed rather than in the same comparison:

>>> def test_list():	>>> def test_list():
... x = [1, 2, 3][:]	... print id([1, 2, 3][:]) # do this four times
... print id(x[:]) # do this four times	
>>> test_list()	>>> test_list()
4451744728	**4451745160**
4451744728 ...	**4451676816 ...**

The three implementations behave differently in this respect;

Appendix 3: Another interesting feature is that CPython alternates the IDs of a sliced object and a sliced variable differently if they are simply printed rather than in the same comparison:

>>> def test_list():	>>> def test_list():
... x = [1, 2, 3][:]	... print id([1, 2, 3][:]) # do this four times
... print id(x[:]) # do this four times	
>>> test_list()	>>> test_list()
4451744728	**4451745160**
4451744728 ...	**4451676816 ...**

The three implementations behave differently in this respect; Jython again is always False:

Appendix 3: Another interesting feature is that CPython alternates the IDs of a sliced object and a sliced variable differently if they are simply printed rather than in the same comparison:

>>> def test_list():	>>> def test_list():
... x = [1, 2, 3][:]	... print id([1, 2, 3][:]) # do this four times
... print id(x[:]) # do this four times	
>>> test_list()	>>> test_list()
4451744728	**4451745160**
4451744728 ...	**4451676816 ...**

The three implementations behave differently in this respect; Jython again is always False:

question	*list, array, bytearray, tuple, buffer*	*string*
id(var[:]), 4x in fn same	**C True** Py/J	**C/Py True** J

question	*list, array*	*bytearray*	*tuple*	*buffer, string*
id(object[:]), 4x in fn same	C False*; Py/J	**C True** Py/J	False	**C/Py True** J

* IDs appear in alternation.

劇
終

(really, this time)