Examining the Identity of a Sliced Python Object

David Prager Branner
Hacker School and PyGotham, New York
20140814 and 20140816

Slicing is a quick way to make a deep copy

Slicing is a quick way to make a deep copy – a copy of the actual values

Slicing is a quick way to make a deep copy – a copy of the actual values – of a sequence such as a list

Slicing is a quick way to make a deep copy – a copy of the actual values – of a sequence such as a list whose elements would otherwise be copied by reference:

Slicing is a quick way to make a deep copy – a copy of the actual values – of a sequence such as a list whose elements would otherwise be copied by reference:

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

Slicing is a quick way to make a deep copy – a copy of the actual values – of a sequence such as a list whose elements would otherwise be copied by reference:

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

Slicing is a quick way to make a deep copy – a copy of the actual values – of a sequence such as a list whose elements would otherwise be copied by reference:

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

Slicing is a quick way to make a deep copy – a copy of the actual values – of a sequence such as a list whose elements would otherwise be copied by reference:

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

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
```

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
>>> b
```

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
```

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
```

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
```

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

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
```

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

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
```

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

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
```

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

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
```

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

Slicing is a quick way to make a deep copy – a copy of the actual values – of a 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'безумный'
>>> b
[1, 2, 3] # unaffected by change to a
```

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

Branner, The Identity of Sliced Python Objects. p. 19

"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

Branner, The Identity of Sliced Python Objects. p. 21

“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

Branner, The Identity of Sliced Python Objects. p. 22

"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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
>>> id(a) == id(b)
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'безумный'
>>> b
[1, 2, 3] # unaffected by change to 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
>>> id(a) == id(b)
True
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'безумный'
>>> b
[1, 2, 3] # unaffected by change to 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
>>> id(a) == id(b)
True
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'безумный'
>>> b
[1, 2, 3] # unaffected by change to 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
>>> id(a) == id(b)
True
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'безумный'
>>> b
[1, 2, 3] # unaffected by change to 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'безумный'
>>> b
[u'\u0431\u0435\u0437\u0443\u043c
\u043d\u044b\u0439', 2, 3]
>>> id(a) == id(b)
True
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
>>> a[0] = u'безумный'
>>> b
[1, 2, 3] # unaffected by change to 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:

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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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
```

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:

```
>>> 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
```

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:

```
>>> 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:

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

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

question	*list*	*array*	*bytearray*	*tuple*	*buffer*
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:

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

question	*list*	*array*	*bytearray*	*tuple*	*buffer*
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. 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. 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. 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 there are a number of identity questions we can ask about slices of objects that have different answers depending on the kind of object and the way we ask the question:

And it turns out that there are a number of identity questions we can ask about slices of objects that have different answers depending on the kind of object 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 there are a number of identity questions we can ask about slices of objects that have different answers depending on the kind of object 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 there are a number of identity questions we can ask about slices of objects that have different answers depending on the kind of object 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 there are a number of identity questions we can ask about slices of objects that have different answers depending on the kind of object 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((1, 2, 3)[:]) == id((1, 2, 3)[:]) # ditto, tuple
False
```

And it turns out that there are a number of identity questions we can ask about slices of objects that have different answers depending on the kind of object 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((1, 2, 3)[:]) == id((1, 2, 3)[:]) # ditto, tuple
False
>>> id('123'[:]) == id('123'[:]) # ditto, string
True
```

And it turns out that there are a number of identity questions we can ask about slices of objects that have different answers depending on the kind of object 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((1, 2, 3)[:]) == id((1, 2, 3)[:]) # ditto, tuple
False
>>> id('123'[:]) == id('123'[:]) # ditto, string
True
>>> id(buffer('123')[:]) == id(buffer('123')[:]) # ditto, buffer
True
```

We can summarize what we know so far in a table:

We can summarize what we know so far in a table:

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

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

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

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

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

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

Apparently not.

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

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

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

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

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

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

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

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

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

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

```
>>> id(x[:]) == id(x[:]) # list
True
>>> id(x[:]) == id(y[:]) # list
True
```

Variables and the objects from which they were originally assigned also behave differently:

Branner, The Identity of Sliced Python Objects. p. 61

Variables and the objects from which they were originally assigned also behave differently:

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

Variables and the objects from which they were originally assigned also behave differently:

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

>>> id(x[:]) == id([1, 2, 3]) # list => True >>> id(x) == id((1, 2, 3)) # tuple => False

Branner, The Identity of Sliced Python Objects. p. 63

It seems random.

It seems random. Is it?

It seems random. Is it? Or is there a deep moral correctness in this diverse behavior?

It seems random. Is it? Or is there a deep moral correctness in this diverse behavior?

In particular, is there some reason why an object being compared with itself should sometimes be considered to be the same object in both cases and sometimes two different objects?

An answer by distribution, not internals: The three implementations CPython, PyPy, and Jython (each v. 2.7) all return different patterns of True and False with respect to these questions:

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

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

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

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

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

That is the main point of this presentation.

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

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

That is the main point of this presentation. (Note that Jython tests False everywhere that CPython tests **True** in these examples; PyPy is mixed.)

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

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

That is the main point of this presentation. (Note that Jython tests False everywhere that CPython tests **True** in these examples; PyPy is mixed.) In sum: the behavior of the `id()` function is uniform neither with respect to the various sequences nor among the three main implementations.

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

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

That is the main point of this presentation. (Note that Jython tests False everywhere that CPython tests **True** in these examples; PyPy is mixed.) In sum: the behavior of the `id()` function is uniform neither with respect to the various sequences nor among the three main implementations. It seems doubtful that this is a matter of performance following prescription.

For reference here is how "identity" is defined in the three versions illustrated here:

- 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

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 appearing 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**

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 appearing 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.

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 appearing 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;

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 appearing 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:

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 appearing 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*	*tuple*	*string*
id(var[:]), 4x in fn same	**C True** Py/J	**C True** Py/J	**C/Py True** J
id(object[:]), 4x in fn same	C False*; Py/J	False	**C/Py True** J

* IDs appear in alternation. qqq buffer? bytearray?

劇終