# Examining the Identity of Sliced Python Sequences

David Prager Branner
Hacker School
PyGotham, New York
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]
```

```
>>> 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
>>> 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 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'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 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'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 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'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 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'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 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'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 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 full 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 full 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 full 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 full 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 full 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 full 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 full 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
```

| *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 full 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
```

| *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 full 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 full 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 full 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 full 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 full 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 full 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 full 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 full 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 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(buffer('123')[:]) == id(buffer('123')[:])                              # ditto, buffer
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(buffer('123')[:]) == id(buffer('123')[:])                              # ditto, buffer
True
>>> 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(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 know so far in a table:

We can summarize what we know 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 know 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 know 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 know 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 know 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 know 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 know 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 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* | *buffer* | *tuple* | *string* |
|---|---|---|---|---|
| id(object) == id(object[:]) | False | False | False but... | **True** |
| id(object[:]) == id(object[:]) | False | **True** | False but... | **True** |
| var = object<br>id(var[:]) == id(var[:]) | **True** | **True** | **True** | **True** |
| var2 = var[:]<br>id(var[:]) == id(var2[:]) | **True** | False | **True** | **True** |

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

| *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<br>id(var[:]) == id(var[:]) | **True** | **True** | **True** | **True** |
| var2 = var[:]<br>id(var[:]) == id(var2[:]) | **True** | False | **True** | **True** |

There are some other interesting inconsistencies of this sort 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 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, it would make sense for a deep copy of an object to be a different entity from that object

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

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

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

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

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

In particular, it would make sense for a deep copy of an object to be a different entity from that object and also from another deep copy of the same 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 a deep moral correctness in this diverse behavior?

In particular, it would make sense for a deep copy of an object to be a different entity from that object and also from another deep copy of the same 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** |

And again:

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

# Internals will explain these inconsistencies

Internals will explain these inconsistencies, but the high level patterns of distribution are more eloquent:

Internals will explain these inconsistencies, but the high level patterns of distribution are more eloquent: The implementations CPython, PyPy, and Jython (each v. 2.7) return different patterns of True and False with respect to these questions

Internals will explain these inconsistencies, but the high level patterns of distribution are more eloquent: The implementations CPython, PyPy, and Jython (each v. 2.7) return different patterns of True and False with respect to these questions (only Jython is fully consistent):

Internals will explain these inconsistencies, but the high level patterns of distribution are more eloquent: The implementations CPython, PyPy, and Jython (each v. 2.7) return different patterns of True and False with respect to 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<br>id(var[:]) == id(var[:]) | **C** Py/J | **C/Py** J | **C** Py/J | **C/Py** J |
| var2 = var[:]<br>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 will explain these inconsistencies, but the high level patterns of distribution are more eloquent: The implementations CPython, PyPy, and Jython (each v. 2.7) return different patterns of True and False with respect to 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<br>id(var[:]) == id(var[:]) | **C** Py/J | **C/Py** J | **C** Py/J | **C/Py** J |
| var2 = var[:]<br>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.

That is the main point of this presentation.

Internals will explain these inconsistencies, but the high level patterns of distribution are more eloquent: The implementations CPython, PyPy, and Jython (each v. 2.7) return different patterns of True and False with respect to 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<br>id(var[:]) == id(var[:]) | **C** Py/J | **C/Py** J | **C** Py/J | **C/Py** J |
| var2 = var[:]<br>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.

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

Internals will explain these inconsistencies, but the high level patterns of distribution are more eloquent: The implementations CPython, PyPy, and Jython (each v. 2.7) return different patterns of True and False with respect to 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<br>id(var[:]) == id(var[:]) | **C** Py/J | **C/Py** J | **C** Py/J | **C/Py** J |
| var2 = var[:]<br>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.

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.

Internals will explain these inconsistencies, but the high level patterns of distribution are more eloquent: The implementations CPython, PyPy, and Jython (each v. 2.7) return different patterns of True and False with respect to 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<br>id(var[:]) == id(var[:]) | **C** Py/J | **C/Py** J | **C** Py/J | **C/Py** J |
| var2 = var[:]<br>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.

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.

# 劇終

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 |
| id(var) == id(var[:]) | False | False | **C** Py/J | **C/Py** J |
| 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

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 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** |

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 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.

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 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:

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 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

(really, this time)