LECTURE 4: OBJECT ORIENTED IDIOMS IN PYTHON

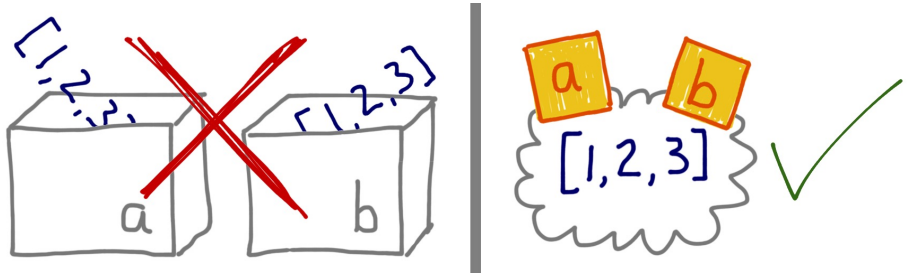
ANDREY.BOCHARNIKOV@GMAIL.COM

TELEGRAM: @RICKO_X

- Recap:
  - Variables
  - Tuples
  - Immutability
  - Shallow copy
  - Deep copy
  - Function parameters
  - Garbage collection

VARIABLES ARE NOT BOXES

- Python variables are like reference variables in Java, so it's better to think of them as **labels attached to objects**.




```
>>>a=[1,2,3]
>>>b=a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

PythonTutor link

OBJECT CREATED BEFORE THE ASSIGNMENT

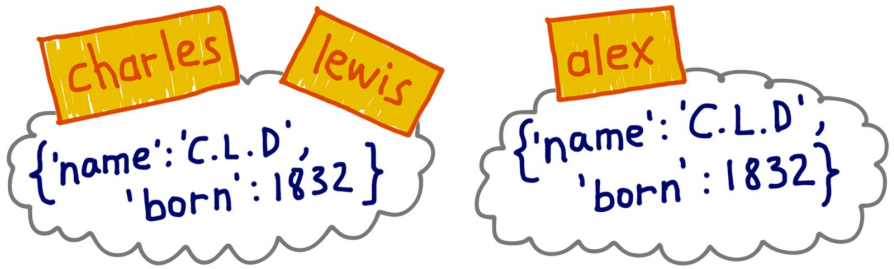
- Always read the right-hand side:
  - first that's where the object is created or retrieved
  - After that, the variable on the left is bound to the object, like a label stuck to it

```
>>> x = Gizmo()
Gizmo id: 4301489152
>>>y=Gizmo()*10
Gizmo id: 4301489432
Traceback (most recent call last):
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'
```

```
class Gizmo:
... def __init__(self):
...     print('Gizmo id: %d' % id(self))
```

IDENTITY, EQUALITY AND ALIASES

- Several object can have several labels assigned to it (variables)




```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles
>>> lewis is charles
True
>>> id(charles), id(lewis)
(4300473992, 4300473992)
>>> lewis['balance'] = 950
>>> charles
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}
>>> alex == charles
True
>>> alex is not charles
True
```

PythonTutor Link

== VS 'IS'

- **==** compares the values of objects
  - Calls __eq__method
  - may involve a lot of processing
- ***is*** compares identities of objects
  - Faster, cannot be overloaded
- Affects
  - Tuples
  - Deep copies
  - Shallow copies

```
x is None
x is not None
```

IMMUTABLE TUPLES

- Tuples hold references to objects
- Tuple may not be changed
  - Immutability refers only to the references it holds
  - But referenced items may change

```
>>> t1 = (1, 2, [30, 40])
>>> t2 = (1, 2, [30, 40])
>>>t1==t2
True
>>> id(t1[-1])
4302515784
>>> t1[-1].append(99)
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1])
4302515784
>>>t1==t2
False
```

PythonTutor Link

COPIES ARE SHALLOW BY DEFAULT

- The outermost container is duplicated
- But the copy is filled with references to the same items held by the original container
- May or may not be what you want.

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1)
>>> l2
[3, [55, 44], (7, 8, 9)]
>>>l2==l1
True
>>>l2 is l1
False

>>>l1 = [1,2,3]
>>>l2 = l1[:]
```

PythonTutor Link

DEEP COPY

- Deep copy - duplicates that do not share references of embedded objects
- Use module **copy** (copy and deepcopy functions)
- Cyclic references

FUNCTION PARAMETERS, CALL BY SHARING

- Parameters inside the function become aliases of the actual arguments.
- Mutable types as parameter defaults: bad idea

```
>>> def f(a, b):
...     a+=b
...     return a ...
>>>x=1
>>>y=2
>>> f(x, y)
3
>>>x,y
(1, 2)
>>>a=[1,2]
>>>b=[3,4]
>>> f(a, b)
[1, 2, 3, 4]
>>>a,b
([1, 2, 3, 4], [3, 4])
>>>t=(10,20)
>>>u=(30,40)
>>> f(t, u)
(10, 20, 30, 40)
>>>t,u
((10, 20), (30, 40))
```

GARBAGE COLLECTION

- Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected.
- reference counting

```
>>> import weakref
>>>s1={1,2,3}
>>>s2=s1

>>> def bye():
... print('Gone with the wind...') ...

>>> ender = weakref.finalize(s1, bye)
>>> ender.alive
True

>>> del s1
>>> ender.alive
True
>>> s2 = 'spam'
Gone with the wind...
>>> ender.alive
False
```

SUMMARY

- Every Python object has an identity, a type and a value. Only the value of an object changes over time
- Simple assignment does not create copies
- The identities of the objects within an immutable collection never change
- Function parameters are passed as aliases
- Using mutable objects as default values for function parameters is dangerous

PROTOCOLS AND DUCK TYPING

- The base sequence protocol in Python entails just the __len__ and __getitem__ methods

```
import collections
Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]
    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

Don't check whether it is-a duck: check whether it quacks-like-a duck, walks-like-a duck, etc, etc, depending on exactly what subset of duck-like behavior you need to play your language-games with

Alex Martelli

RECOMMENDED BOOK

- Fluent Python: Clear, Concise, and Effective Programming by Luciano Ramalho

- Assignment #1
  - Saturday 2:30 PM (Novisibirsk Time, UTC+7)
- Good news