Iterators, Generators, References and Assignment

Iteration

- Iteration defined: Looping over items

```
a = [2, 4, 10, 37, 62]
# Iterate over a
for x in a:
    ...
```

- A very common pattern, fundamental to computer science
- Loops, list comprehensions, tuple unpacking, equality comparisons with sequences, etc
- Most programs do a huge amount of iteration

Iteration: Protocol

- Iteration

```
for x in obj:
    # statements
    ...
```

- Underneath the covers

```
_iter = obj.__iter__()              # Get iterator object
while True:
    try:
        x = _iter.__next__()        # Get next item
    except StopIteration:           # No more items
        break
    # statements
    ...
```

- Objects that work with the for-loop all implement this low level protocol

Iteration: Protocol

- Example: Manual iteration over a list

```
>>> x = [1, 2, 3]
>>> it = iter(x)      # equivalent of x.__iter__()
>>> next(it)          # equivalent of it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Delegating Iteration

- Iteration is part of the container protocol
- The easiest way to make a custom container iterable is to delegate to a builtin iterator

```
class Portfolio:
    def __init__(self):
        self._holdings = []
    def __iter__(self):
        return iter(self._holdings)
```

- An example of object composition with delegation

Generators

- Generators provide "stateful iteration"
- Generators simplify custom iteration, using the yield keyword

```
def countdown(n):
    print('Counting down from', n)
    while n > 0:
        yield n
        n -= 1

>>> for i in countdown(5):
...     print('T-minus', i)
...
Counting down from 5
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
```

Generator Functions

- Generators are full coroutines, this is a subset of their functionality
- Behaviour is very different from normal functions
- Calling a generator function creates a generator object (an iterator), it does not start running the function

```
def countdown(n):
    print('Counting down from', n)
    while n > 0:
        yield n
        n -= 1

>>> x = countdown(10)
>>> x
<generator object at 0x58490>
```

Notice that no output was produced!

Generator Functions

- Execution only begins on next (when iteration starts)

```
>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>> next(x)                      # invokes x.__next__()
Counting down from 10
10
```

Function execution starts here

- yield produces a value and pauses execution
- Control flow is yielded along with the value
- When next is called again execution resumes, until we hit yield again

```
>>> next(x)
9
>>> next(x)
8
```

Generator Functions

- When the generator returns, iteration stops

```
>>> next(x)
1
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- Observation: A generator function implements the same low level iteration protocol that the for-loop uses on all iterables; lists, tuples, dicts, files, etc
- Any function that contains the "yield" keyword is a generator function

Reusing Generators

- Generators are "one shot", single use

```
>>> c = countdown(5)
>>> for x in c:
...     print('T-minus', x)
...
Counting down from 5
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
>>> for x in c:
...     print('T-minus', x)
...
>>>
```

- To reuse, recreate the generator

```
>>> c = countdown(5)
```

Reusable Generators

- Subtle trick: Making iterable objects with __iter__

```
class Countdown:
    def __init__(self, n):
        self.n = n
    def __iter__(self):
        n = self.n
        while n > 0:
            yield n
            n -= 1
```

- The object can be iterated over many times, because every call to __iter__ produces a new generator

Understanding Assignment References and Mutable Objects

References and Assignment

- Names (variables) are one way to take a reference to an object
- Python uses reference counting for garbage collection
- There are many ways to take a reference to (store) an object

```
a = value                       # Assignment to a variable
s.append(value)                 # Appending to a list
thing.attribute = value         # Setting as an object attribute
d['foo'] = value                # Putting in a dictionary
```

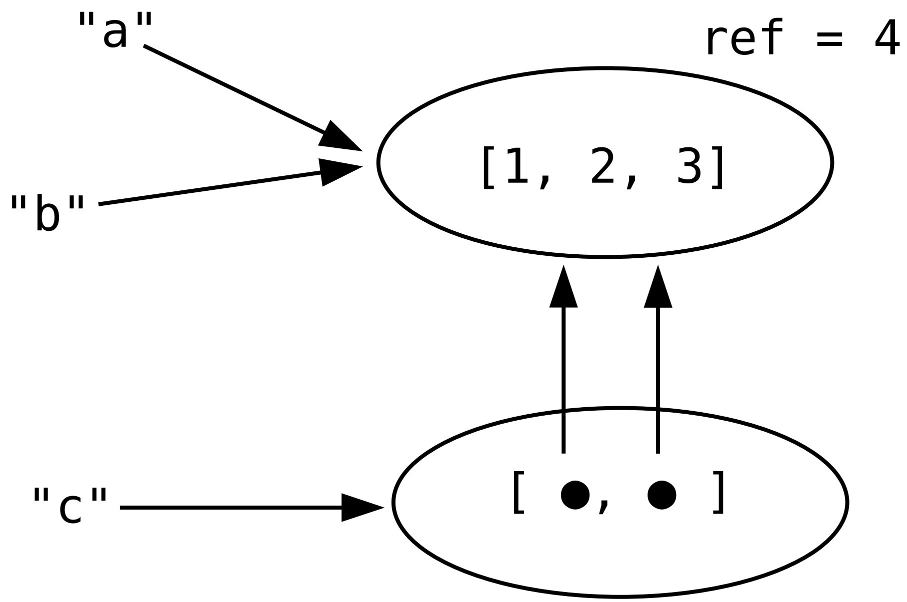
Assignment never copies, it's a reference copy (or pointer copy).

Reference Example

- Here's some interesting code, how many *different* list objects are there here?

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

- Here are the references:



Mutable Objects and References

- Modifying a mutable object by any reference shows up everywhere you have a reference

```
>>> a = [1, 2, 3]
>>> b = a
>>> c = [a, b]
>>> a.append(999)
>>> c
[[1, 2, 3, 999], [1, 2, 3, 999]]
```

This is because no copies were made, all the references point to the same object. This is by design and is not limited to Python.

Call by Object

- Functions receive a reference to objects, not a copy

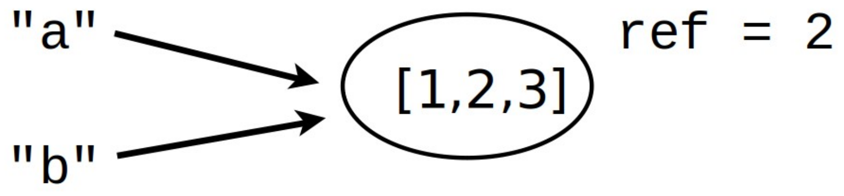
```
>>> def function(thing):
...     thing['new data'] = 33
...
>>> data = {'data': 99}
>>> function(data)
>>> data
{'data': 99, 'new data': 33}
```

- This is useful, not a bug!
- The primitive types (int, float, bool, str) are immutable
- Containers and class instances are *usually* mutable (not tuple or frozenset)

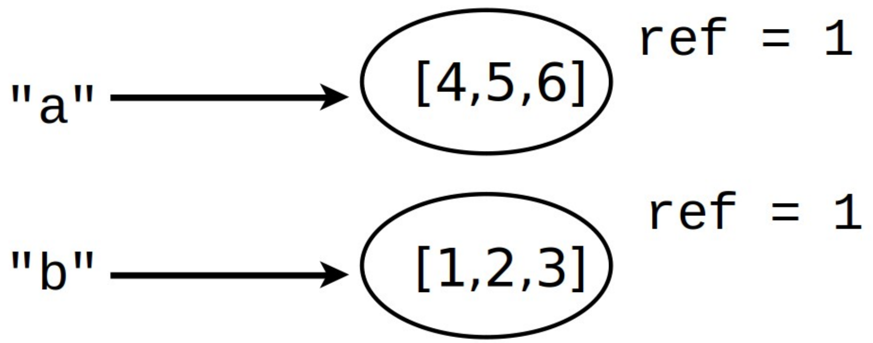
Reassignment

- Reassigning a name (a "rebind" operation) creates a new value rather than modifying the original.

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




```
a = [4,5,6]
```



- The name "a" points to a new object, "b" is unchanged

Identity versus Equality

- Two objects are equal if they have the same value, but they can be different objects
- We can use the "is" operator to check if two references point to the *same object*

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> c = a
>>> a == b
True
>>> a is c
True
>>> a is not b
True
>>> b.append(999)
>>> a == b
False
```

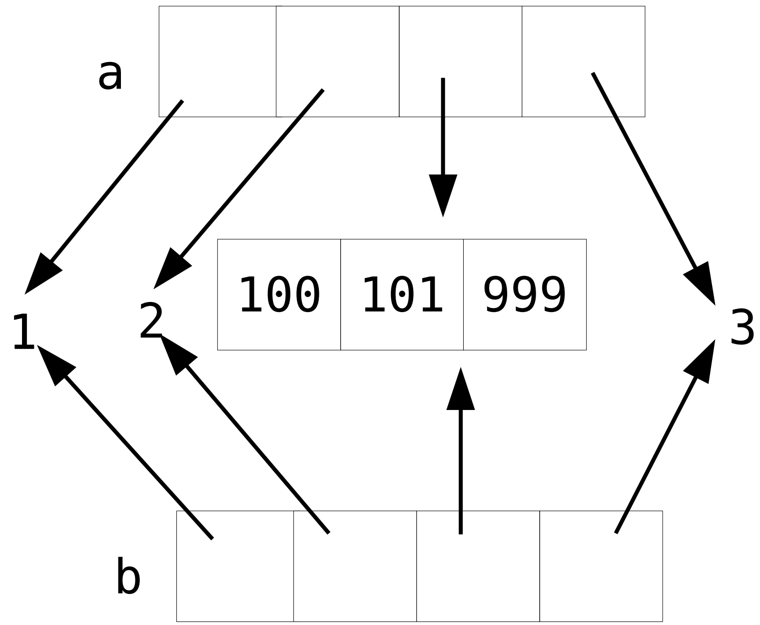
```
>>> id(a)
140522824988032
>>> id(b)
140522825033216
>>> id(c)
140522824988032
```

Note: id is an integer unique for the lifetime of the object.

Shallow Copies

- To avoid problems with mutable objects we can copy them

```
>>> a = [1, 2, [100, 101], 3]
>>> b = list(a)
>>> a is b
False
```



- But notice this:

```
>>> b[2].append(999)
>>> a
[1, 2, [100, 101, 999], 3]
```

- We took a shallow copy, copying references

Deep Copying

- For nested data structures, or objects with shared references, you need to take a "deep copy"
- For this we use the "copy" module

```
>>> import copy
>>> a = [1, 2, [100, 101], 3]
>>> b = copy.deepcopy(a)
>>> b[2].append(999)
>>> a
[1, 2, [100, 101], 3]
>>> a[2] is b[2]
False
```

- There is also copy.copy for shallow copies, but making shallow copies of objects is usually easy

Everything is an Object

- Everything is an object
- Every object has a type
- No special objects, everything is an object:
  - Numbers and strings
  - Containers
  - Exceptions
  - None and the bools
  - Even classes are objects (so what is the type of a class?)

In Python we call all objects "first class" objects.

Example: Functions as Objects

- We can use the fact that functions are objects to simplify the code on the left

```python
from operator import add, sub, mul, truediv as div

if op == '+':
    r = add(x, y)
elif op == '-':
    r = sub(x, y)
elif op == '*':
    r = mul(x, y)
elif op == '/':
    r = div(x, y)
```

```python
ops = {
    "+": add,
    "-": sub,
    "*": mul,
    "/": div
}

r = ops[op](x, y)
```

First Class Objects

- A simple example:

```
>>> import math
>>> items = [abs, math, ValueError]
>>> items
[<built-in function abs>, <module 'math' (built-in)>, <class
'ValueError'>]
>>> items[0](-45)
45
>>> items[1].sqrt(2)
1.4142135623730951
>>> try:
...     x = int('not a number')
... except items[2]:
...     print('Failed!')
...
Failed!
```

A list containing a function, a module, and an exception.

You use items in the list in place of the original names.