Python Applications for Digital Design and Signal Processing

Applications for Digital Design and Signal Processing
Session 3

Dan Boschen

Course Outline

Session	Topics
1	Course Intro: Python, Spyder and Jupyter
2	Core Python
3	**Core Python**
4	Core Python
5	Python Modules and Packages
6	NumPy
7	NumPy, SciPy
8	Python for Verification, Modelling and Analysis

Session 3 Contents

Goals for this Session: Review of the core Python language: Collections, Iterators, Flow Control

Core Python

(continued from Session 2)

Collections

Collections

Collections: A container of multiple heterogeneous objects

Python's Built-in Collections

Type	Category	Ordering	Mutability	Comment
List	Sequence	Ordered	Mutable	
Tuple	Sequence	Ordered	Immutable	
Set	Set	Unordered	Mutable	No duplicate items
Frozenset	Set	Unordered	Immutable	No duplicate items
Dictionary	Mapping	Unordered	Mutable	Indexed by unique keys

Collections

List : [a,…]	`['a', 1, my_func, 12.0, 'a']`
Tuple: (a,…)	`('a', 1, my_func, 12.0, 'a')`
Set : {a,…}	`{'a', 1, my_func, 12.0}`
Frozenset: (a,…)	`frozenset({'a', 1, my_func, 12.0})`
Dictionary: {a:b,…}	`{'key1': 54, 'key2': 'Fred', 'key3': 7.2, 'key4': my_func}`

Slicing Collections - Lists and Tuples

`my_list[`*index*`]`	Select one item
`my_list[`*start* `:` *stop*`]`	Select a range
`my_list[`*start* `:` *stop* `:` *step*`]`	Select a range and increment

Must use **[ ]** to select items; as **( )** would call a function

index, start, stop:

can be **positive** to start on left at **index 0**

or, can be **negative** to start on right at **index -1**

```
my_list[1]     my_list[-2]

my_list =[ _, _, _, _, _, _ ]
```

Slicing Collections - Lists and Tuples

`my_list[`*index*`]`	Select one item
`my_list[`*start* `:` *stop*`]`	Select a range
`my_list[`*start* `:` *stop* `:` *step*`]`	Select a range and increment

Must use **[ ]** to select items; as **( )** would call a function

Slicing Collections - Lists and Tuples

`my_list[`*index*`]` Select one item

index (required) is **a signed integer**

positive start on left at **0**
negative start on right at **-1**

Slicing Collections - Lists and Tuples

`my_list[`*start* `:` *stop*`]` Select a range

Selected range is from *start* up to but **not including** *stop*

Either *start* or *stop* can be **omitted**:

start if blank will default to be the first item in list:	`my_list[:stop]`
stop if blank will default to be the last item in list:	`my_list[start:]`
Omitting both will return **copy** of entire list:	`my_list[:]`

Slicing Collections - Lists and Tuples

`my_list[`*start* `:` *stop* `:` *step*`]` Optional step

Include *step* to increase **increment** of the selection

When *step* is **negative**, selection will traverse collection in the **reverse** direction.

Either *start* or *stop* can be **omitted**: default will be consistent with sign of step

Example:

```
y = [1,2,3]
y[::-1] = [3,2,1]
```

Slicing Collections - Lists and Tuples

Examples with 8 element list: `y= [a, b, c, d, e, f, g, h]`

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
-8	*-7*	*-6*	*-5*	*-4*	*-3*	*-2*	*-1*

slice	result	slice	result
`y[0]`	`a`	`y[-8]`	`a`
`y[7]`	`h`	`y[-1]`	`h`
`y[5:]`	`[f, g, h]`	`y[-3:7]`	`[f, g]`
`y[5:-1]`	`[f, g]`	`y[-3:-1]`	`[f, g]`
`y[-1:4:-1]`	`[h, g, f]`	`y[:-4:-1]`	`[h, g, f]`
`y[7:4: -1]`	`[h, g, f]`	`y[7:-4:-1]`	`[h, g, f]`
`y[:]`	`[a,b,c,d,e,f,g,h]`	`y[::-1]`	`[h,g,f,e,d,c,b,a]`
`y[::2]`	`[a, c, e, g]`	`y[-2::-3]`	`[g, d, a]`

Slicing Collections - Lists and Tuples

Examples with 8 element list: `y= [a, b, c, d, e, f, g, h]`

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
-8	*-7*	*-6*	*-5*	*-4*	*-3*	*-2*	*-1*

Lists cannot wrap; will return an empty collection if stop is sooner that start for direction traversed.

These will all return empty collections:

```
y[-1:4]
y[:4:-1]
y[2:5:-2]
```

Assignments - Lists and Tuples

IMPORTANT!

Case 1: Assign y to x (list or tuple)

x = y

x bound to existing List object
x any y are same object

Case 2: Assign a slice of y to name x (can be list or tuple)

x = y[*start* : *stop* : *step*]

x bound to new List object, is a "shallow copy"
(new list object with same items)

Case 3: Update (change) a slice of y with new values

Cannot do this with a tuple (immutable)
y must be already assigned
z must have same # of elements as slice

y[*start* : *stop* : *step*] = z

updated contents will be the same items

Slicing Collections - Dictionaries

When slicing a **dictionary** the **keys** are used

```
>>> my_dict = {
...   'key1': 5,
...   'key2': 6,
...   'key3': 8
... }
```

`>>> my_dict['key1']` will return 5

to select a range use a "dictionary comprehension"

- (covered in next Session)

Manipulating Collections

Mutable collections have methods to change in place

Use dir() on object to view possible methods

For example, with a list you can append(), reverse(), sort() …

```
>>> a = [1, 4, 2, 6]
>>> a.append(5)
```

result is [1, 4, 2, 6, 5]

Unpacking Collections

unpacking tuples, lists and sets:

```
a, b, c = (item1, item2, item3)
a, b, c = [item1, item2, item3]
a, b, c = {item1, item2, item3}
```

unpacking dictionaries:

```
a, b, c = my_dict                returns the keys
a, b, c = my_dict.values()       returns the values
a, b, c = my_dict.items()        returns (k,v)
```

Extended Iterable Unpacking

```
a, *b, c, d = [1, 2, 5, 2, 12, 1]
```

Result:

a = 1

b = [2, 5, 2]

c = 12

d = 1

Useful! Skipping columns in data, or to get last item in any iterable:

```
*_, last_item = my_iterable
```

Nested Collections

```
my_list = ['a', ['b', 'c']]
```

`my_list[1]` results in ['b', 'c']

`my_list[1][1]` results in 'c'

Boolean Value of Collections

All Python objects return a Boolean value.

`list1 == list2`	Evaluates to true if all items are equal (order matters)
`set1 == set2`	Evaluates to true if all items are equal (in any order)
`dic1 == dic2`	Evaluates to true if both dictionaries have equal "key, value" pairs (in any order)
`all(my_list)`	Evaluates to true if all items are True
`any(my_list)`	Evaluates to true if any items are True

Mutability and Collections

Assignments - Lists and Tuples

Case 1: Assign reference b to name a (list or tuple)

```
a = b
```

Case 2: Assign slice of b to name a (list or tuple)

```
a = b[start : stop : step]
```

Case 3: Update (change) slice of a with new values (mutable change!)

```
a[start : stop : step] = c
```

- a cannot be a tuple (tuples are immutable!)
- a must be already assigned

Case 1 Example

```
x = [-6,[2.1]]
```

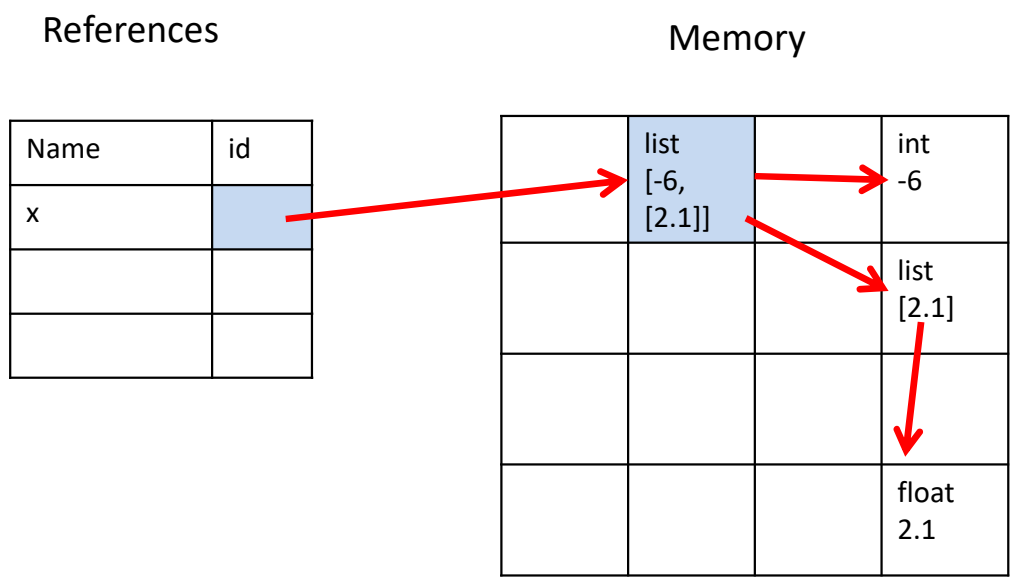
References

Name	id

Memory

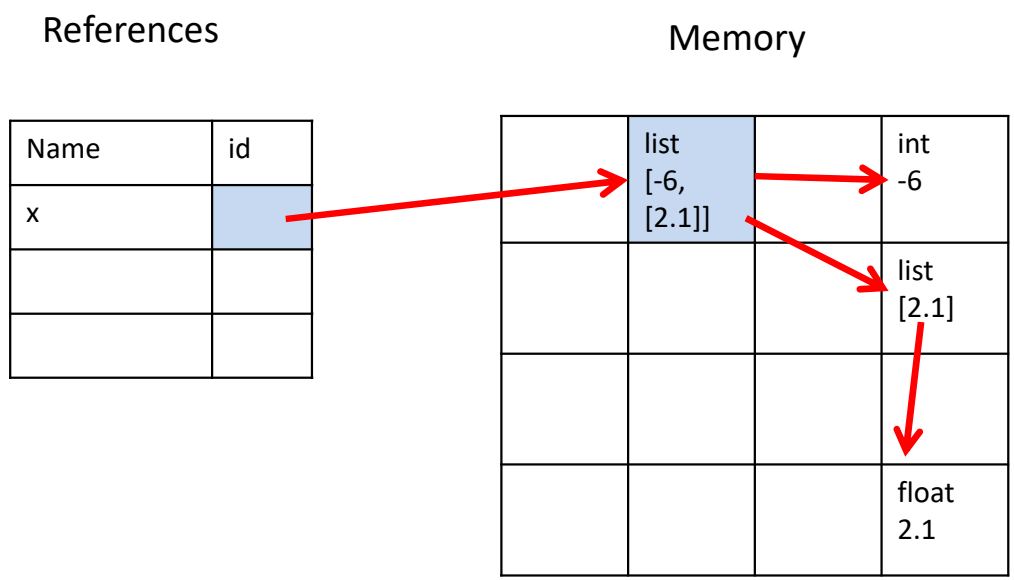
Case 1 Example

```
x = [-6,[2.1]]
```



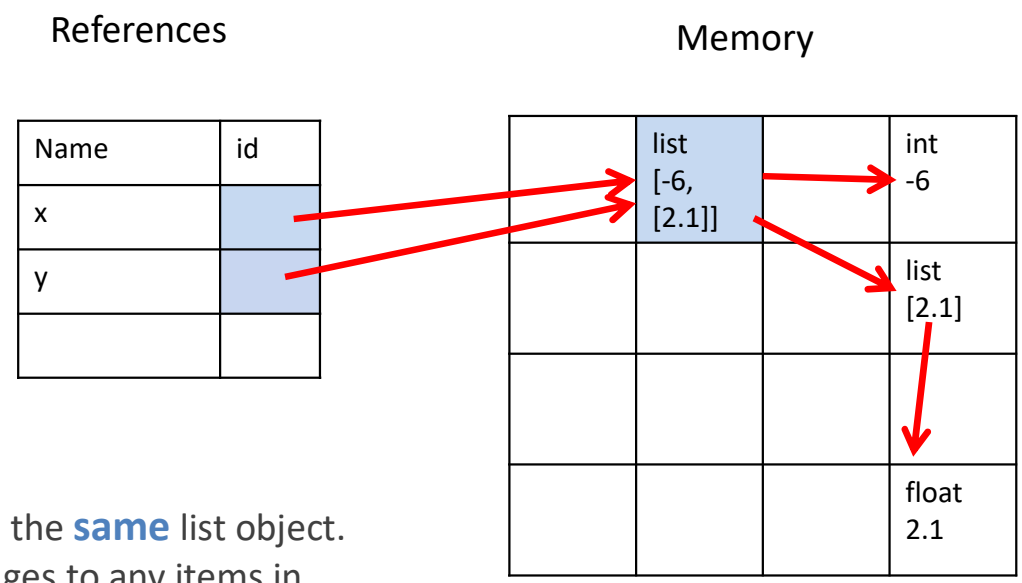
Case 1 Example

```
x = [-6,[2.1]]
y = x
```



Case 1 Example

```
x = [-6,[2.1]]
y = x
```



x and y are bound to the **same** list object. Assignments or changes to any items in either will be visible in the other.

Case 2 Example

```
x = [-6,[2.1]]
```

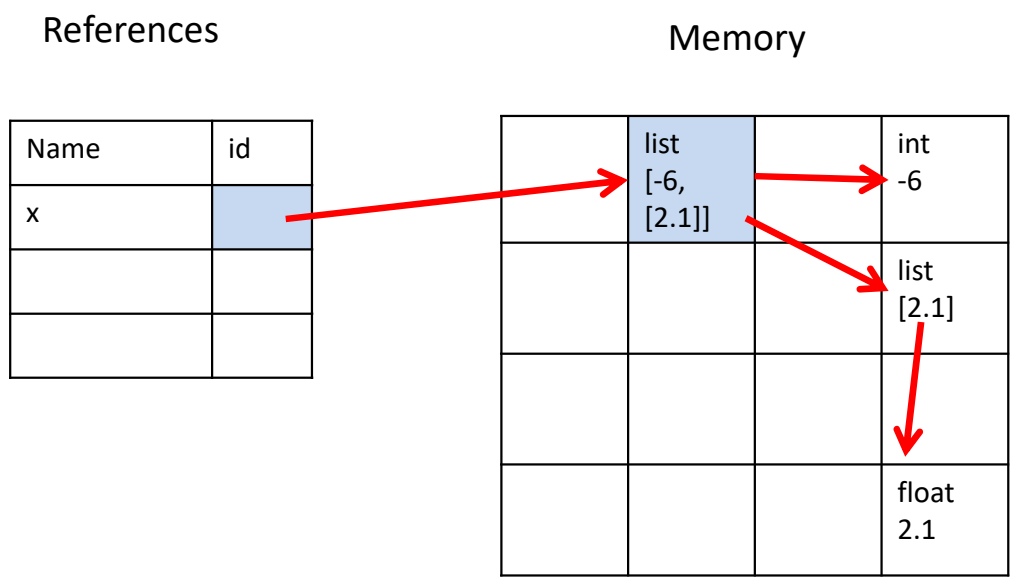
References

Name	id

Memory

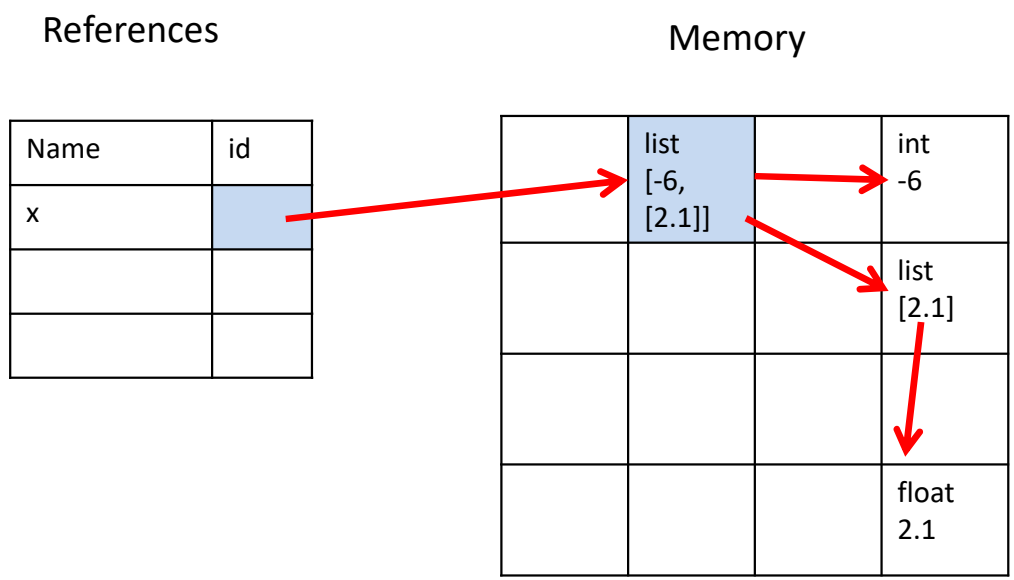
Case 2 Example

```
x = [-6,[2.1]]
```



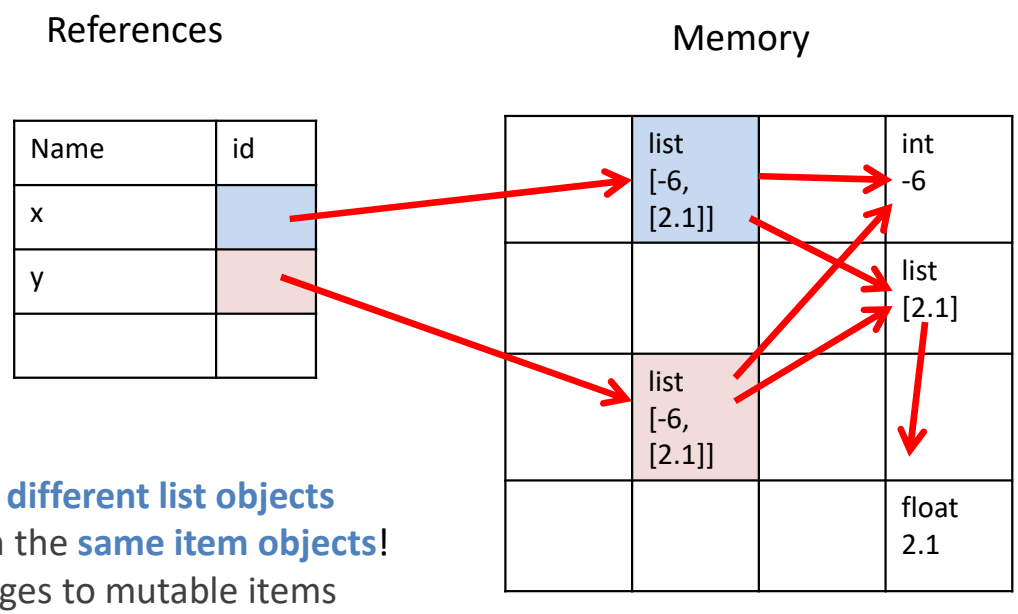
Case 2 Example

```
x = [-6,[2.1]]
y = x[:]
```



Case 2 Example

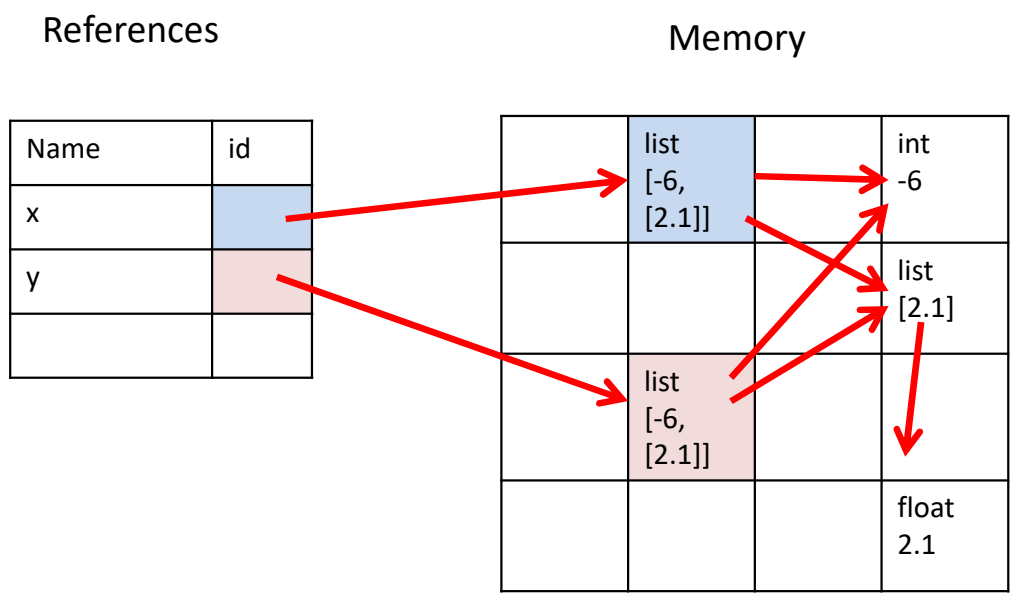
```
x = [-6,[2.1]]
y = x[:]
```



x and y are bound to **different list objects** but both lists contain the **same item objects**! Assignments or changes to mutable items (only) will be visible in the other list.

Case 2 Example

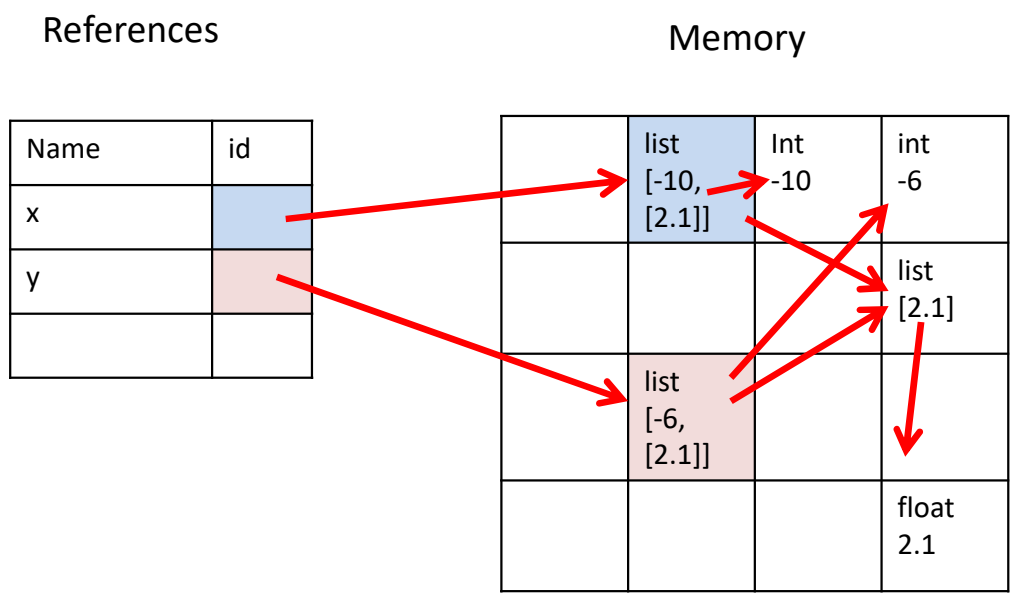
```
x = [-6,[2.1]]
y = x[:]
x[0] = -10
```



Case 2 Example

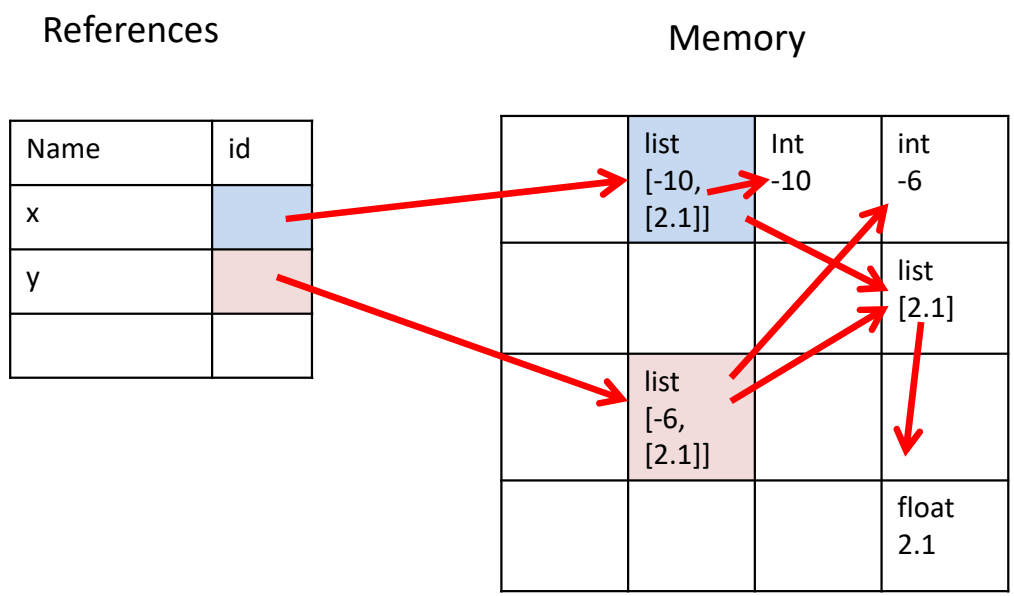
```
x = [-6,[2.1]]
y = x[:]
x[0] = -10
```

```
>>> print(x)
[-10, [2.1]]
>>> print(y)
[-6, [2.1]]
```



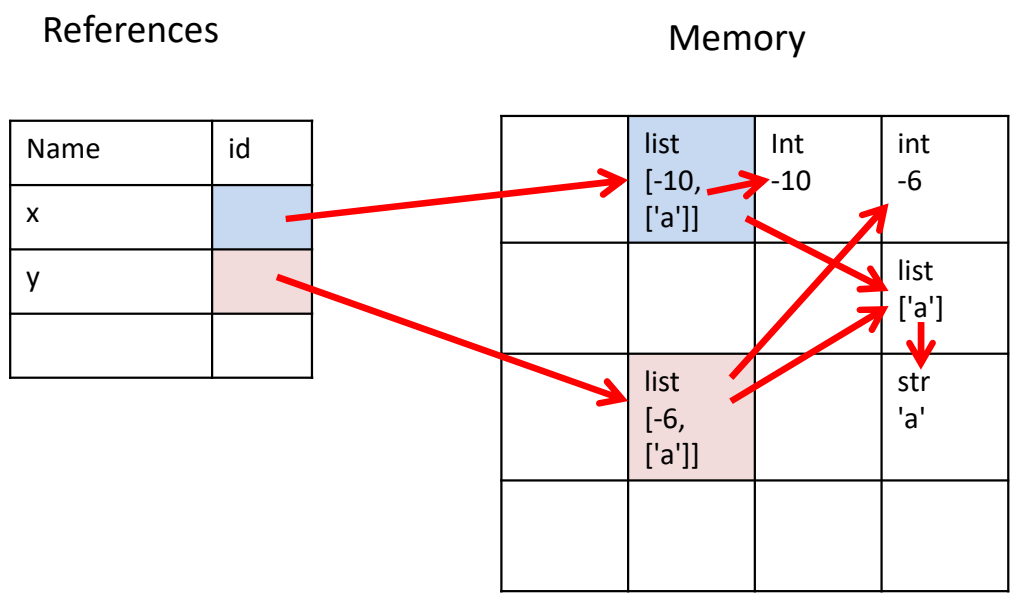
Case 2 Example

```
x = [-6,[2.1]]
y = x[:]
x[0] = -10
x[1][0]='a'
```



Case 2 Example

```
x = [-6,[2.1]]
y = x[:]
x[0] = -10
x[1][0]='a'
```




```
>>> print(x)
[-10, ['a']]
>>> print(y)
[-6, ['a']]
```

Case 3 Example

```
x = [-6,[2.1]]
```

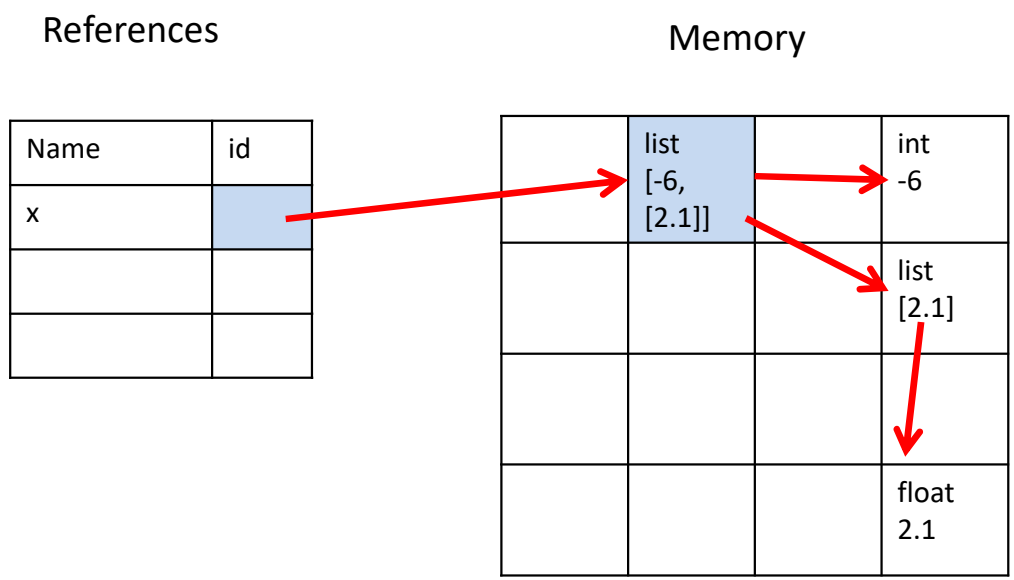
References

Name	id

Memory

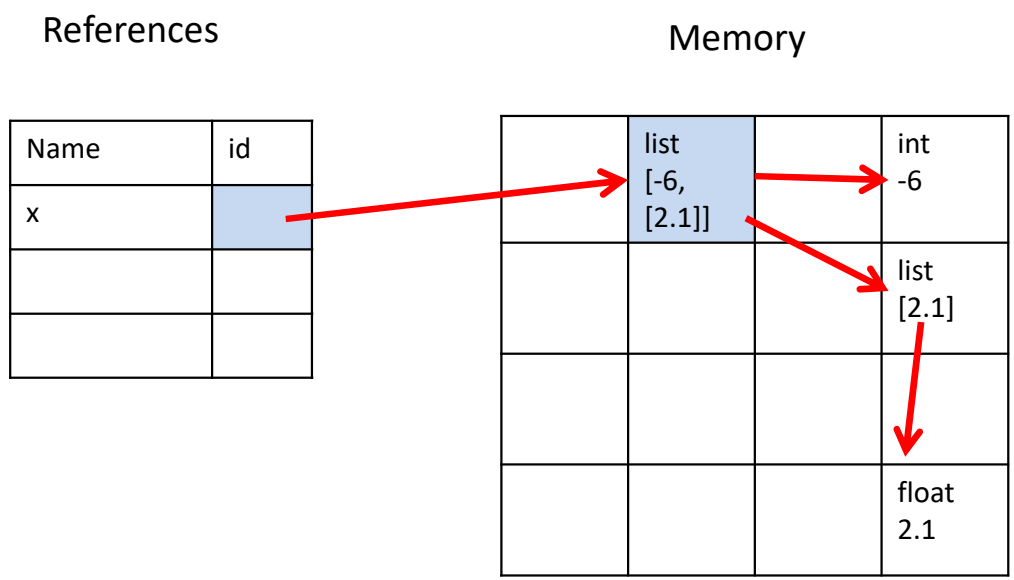
Case 3 Example

```
x = [-6,[2.1]]
```



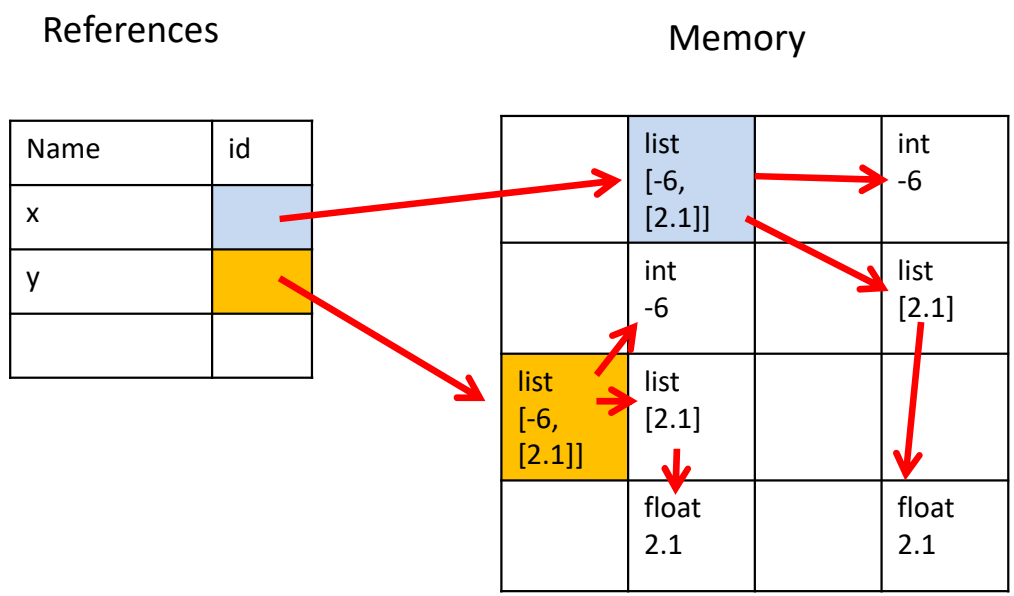
Case 3 Example

```
x = [-6,[2.1]]
y = [-6,[2.1]]
```



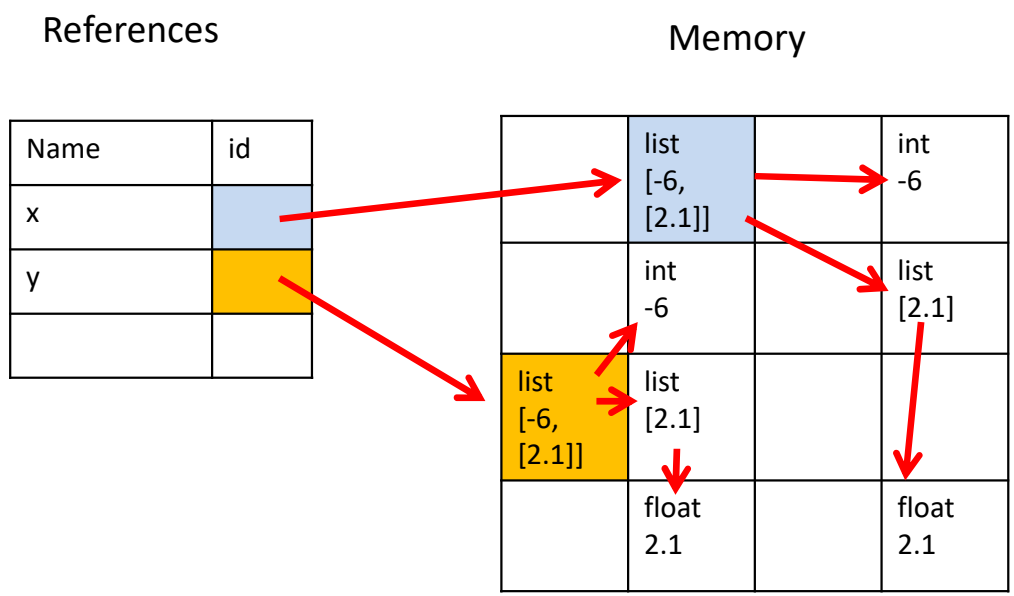
Case 3 Example

```
x = [-6,[2.1]]
y = [-6,[2.1]]
```



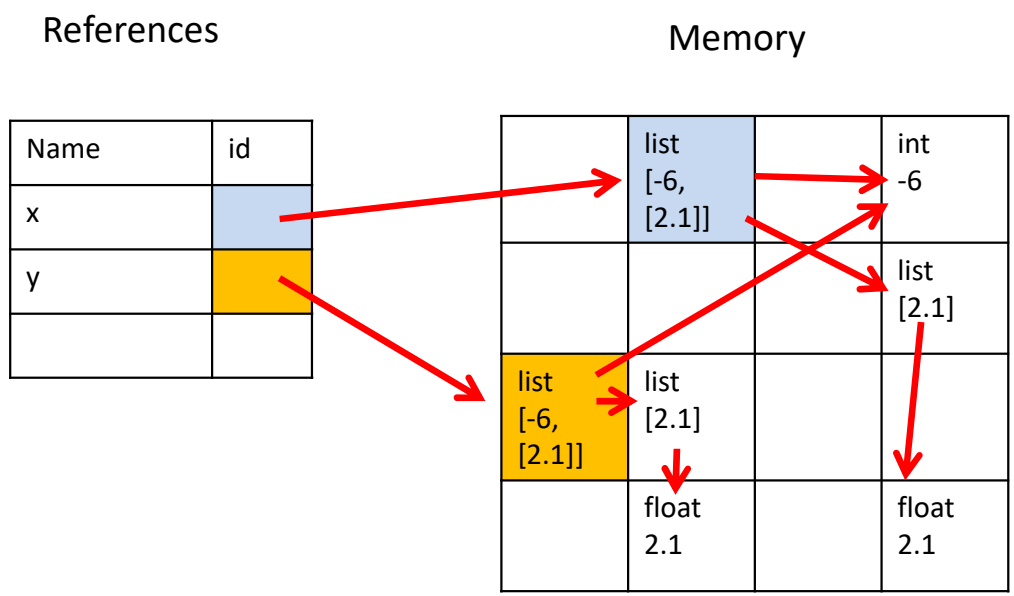
Case 3 Example

```
x = [-6,[2.1]]
y = [-6,[2.1]]
y[0] = x[0]
```



Case 3 Example

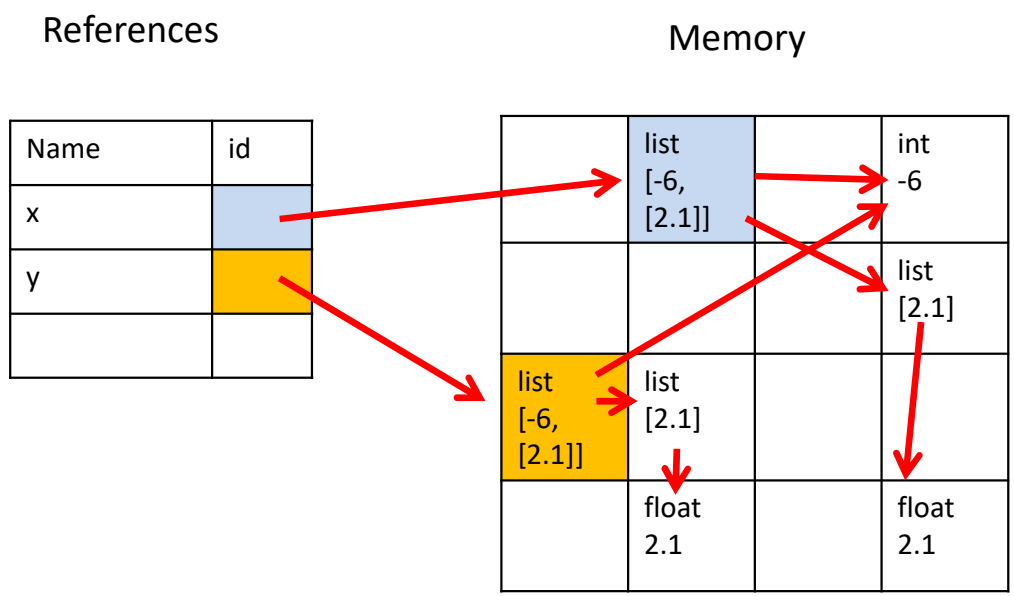
```
x = [-6,[2.1]]
y = [-6,[2.1]]
y[0] = x[0]
```




```
>>> print(x)
[-6, [2.1]]
>>> print(y)
[-6, [2.1]]
```

Case 3 Example

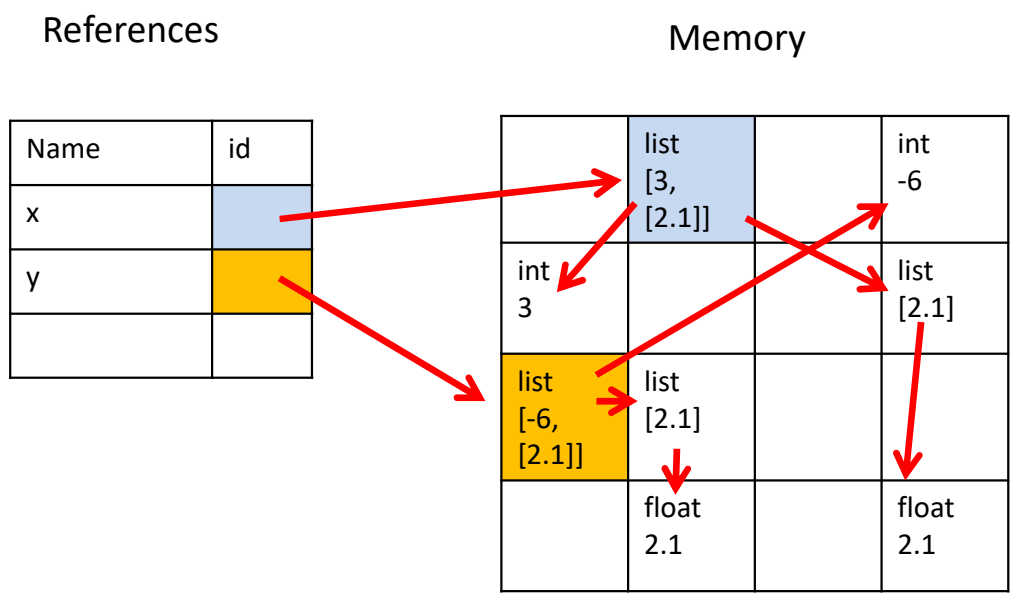
```
x = [-6,[2.1]]
y = [-6,[2.1]]
y[0] = x[0]
x[0] = 3
```



Case 3 Example

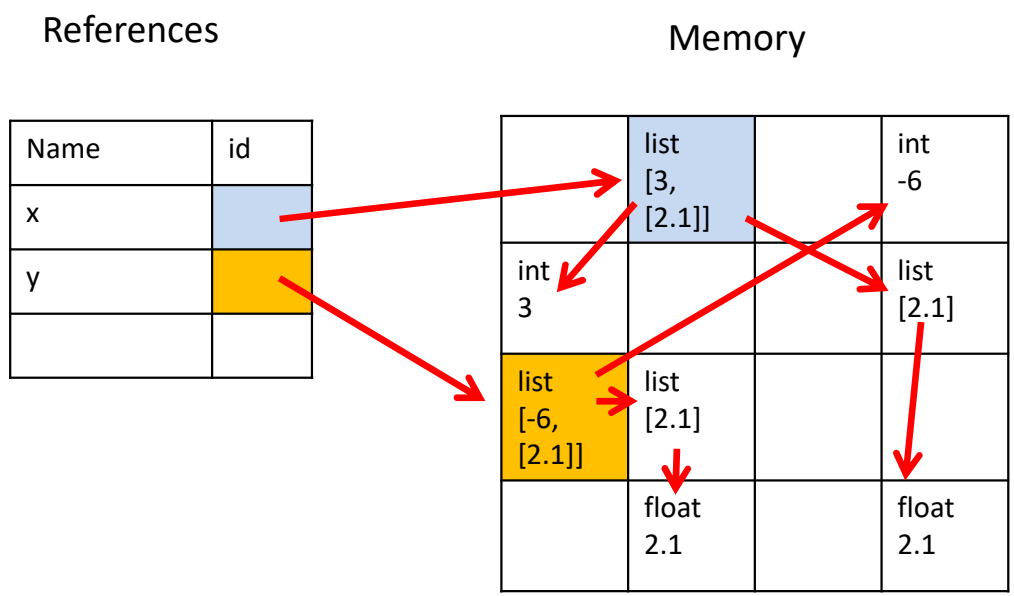
```
x = [-6,[2.1]]
y = [-6,[2.1]]
y[0] = x[0]
x[0] = 3
```

```
>>> print(x)
[3, [2.1]]
>>> print(y)
[-6, [2.1]]
```



Case 3 Example

```
x = [-6,[2.1]]
y = [-6,[2.1]]
y[0] = x[0]
x[0] = 3
y[1] = x[1]
```



Case 3 Example

```
x = [-6,[2.1]]
y = [-6,[2.1]]
y[0] = x[0]
x[0] = 3
y[1] = x[1]
```

References

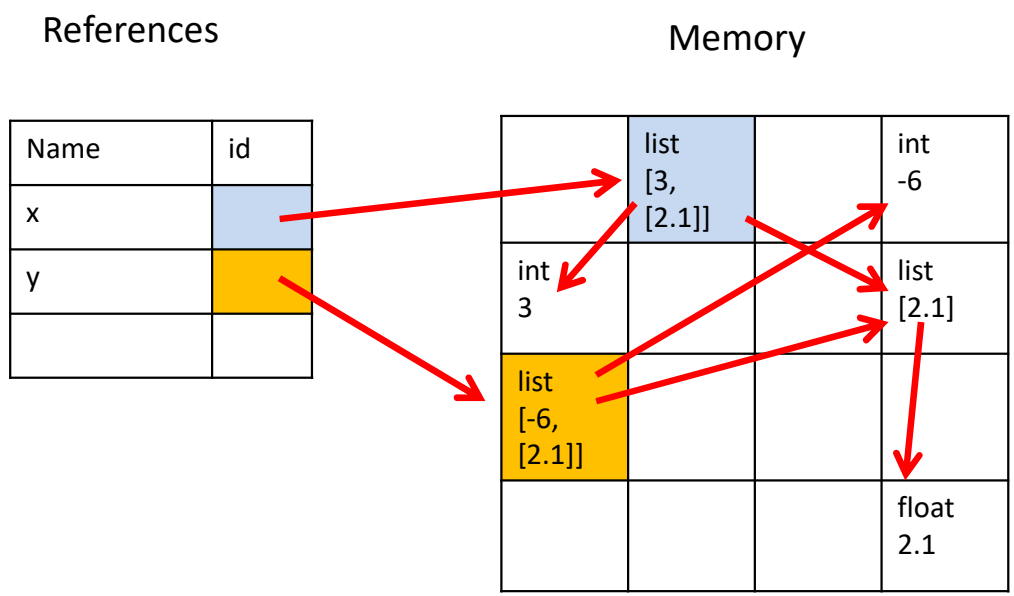
Name	id
x	
y	

Memory

	list [3, [2.1]]		int -6
int 3			list [2.1]
list [-6, [2.1]]			
			float 2.1

Case 3 Example

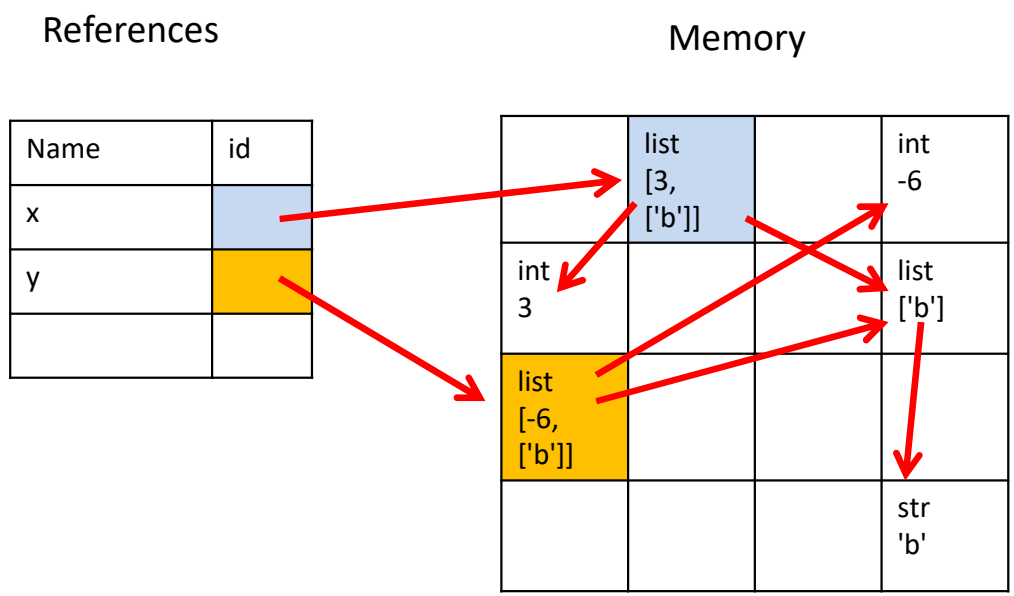
```
x = [-6,[2.1]]
y = [-6,[2.1]]
y[0] = x[0]
x[0] = 3
y[1] = x[1]
x[1][0] = 'b'
```



Case 3 Example

```
x = [-6,[2.1]]
y = [-6,[2.1]]
y[0] = x[0]
x[0] = 3
y[1] = x[1]
x[1][0] = 'b'

>>> print(x)
[3, ['b']]
>>> print(y)
[-6, ['b']]
```



Assignments - Lists and Tuples

IMPORTANT!

Case 1: Assign reference b to name a (list or tuple)

```
a = b
```

Case 2: Assign slice of b to name a (list or tuple)

```
a = b[start : stop : step]
```

Case 3: Update (change) slice of a with new values (mutable change!)

```
a[start : stop : step] = c
```

- a cannot be a tuple (tuples are immutable!)
- a must be already assigned

What happened?

We had conditions where **shared objects** existed.

It really only mattered if the shared object was **mutable**.

Use **tuples** for ordered collections where the updating efficiency of a mutable object (list) is not a concern.

Iterables and Iterators

Iterable and Iterator

Iterable: Anything that can be looped over

strings, files, lists, …

Using iter() on an iterable returns an iterator

Iterator:

Has state to keep track of location in iteration

Has a method to return next value using next()

Signals when it is done with StopIteration exception

Iterable and Iterator

Iterable protocol:

Has a **__iter__() method** (returns an iterator)

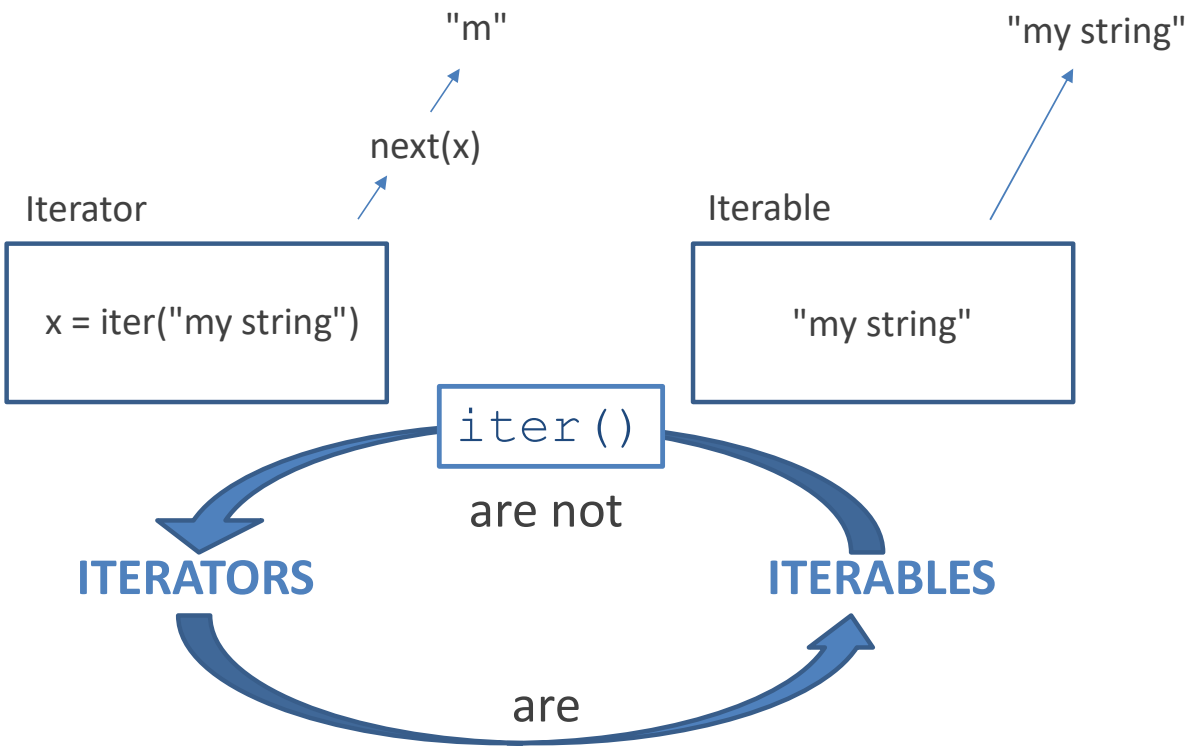
Iterator protocol:

Has a **__next__() method** (return next value)

Raises a **StopIteration Exception** when done

Has a **__iter__() method** (an iterator therefore is an iterable)

Iterable and Iterator



Iterables

Lists from iterables:	`list(`*iterable*`)`
Tuples from iterables:	`tuple(`*iterable*`)`
Sets from iterables:	`set(`*iterable*`)`
Frozenset from iterables:	`frozenset(`*iterable*`)`
Dictionary from iterables:	`dict(`*iterable*`)`
Unpacking from iterables:	`x, y, *z =` *iterable*

Flow Control and Loops

Flow Control

if, elif, else

```
>>> if test1:
...     # do stuff
>>> elif test2:
...     # do other stuff
>>> else:
...     # do something else
```

For Loops

for … in , continue, break, else

```
>>> for x in my_iterable:
...     # do stuff
...      if test2:
...         continue
...     # more stuff
...      if test3:
...         break
...      else:
...         print("Loop finished")
```

Skip everything after continue and do next loop iteration

Exit loop, do not print "Loop finished"

Print "Loop finished" if loop reaches end of iteration

Enumerate

```
>>> for count, item in enumerate(items):
...     # do something with count and item
```

`count` will increment at each loop cycle: 0, 1, 2, 3, …

enumerate() has optional parameter to start at non-zero

Dictionary Looping

Looping over dictionaries in Python 3:

```
>>> for key in my_dict:
...     # do this to each key

>>> for value in my_dict.values():
...     # do this to each value

>>> for key, value in my_dict.items():
...     # do this to each key and value
```

While Loops

while, continue, break, else

```
>>> while test1:
...     # do stuff
...         if test2:
...             continue
...     # more stuff
...         if test3:
...             break
...     else:
...         print("Loop finished")
```

execution next loop cycle skipping everything after continue

Exit Loop, do not print "Loop Finished"

Code to execute at completion (when test1 == False)

Additional Resources

Loop Like a Native by Ned Batchelder, PyCon 2013
https://nedbatchelder.com/text/iter.html
(Great talk on iterators and generators)

BACK-UP SLIDES

Built-in Types — Python 3.7.3rc1

Python Software Foundation [US] | https://docs.python.org/3.7/library/stdtypes.html#sequence-types-list-tuple-range

Sequence Types — `list`, `tuple`, `range`

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of binary data and text strings are described in dedicated sections.

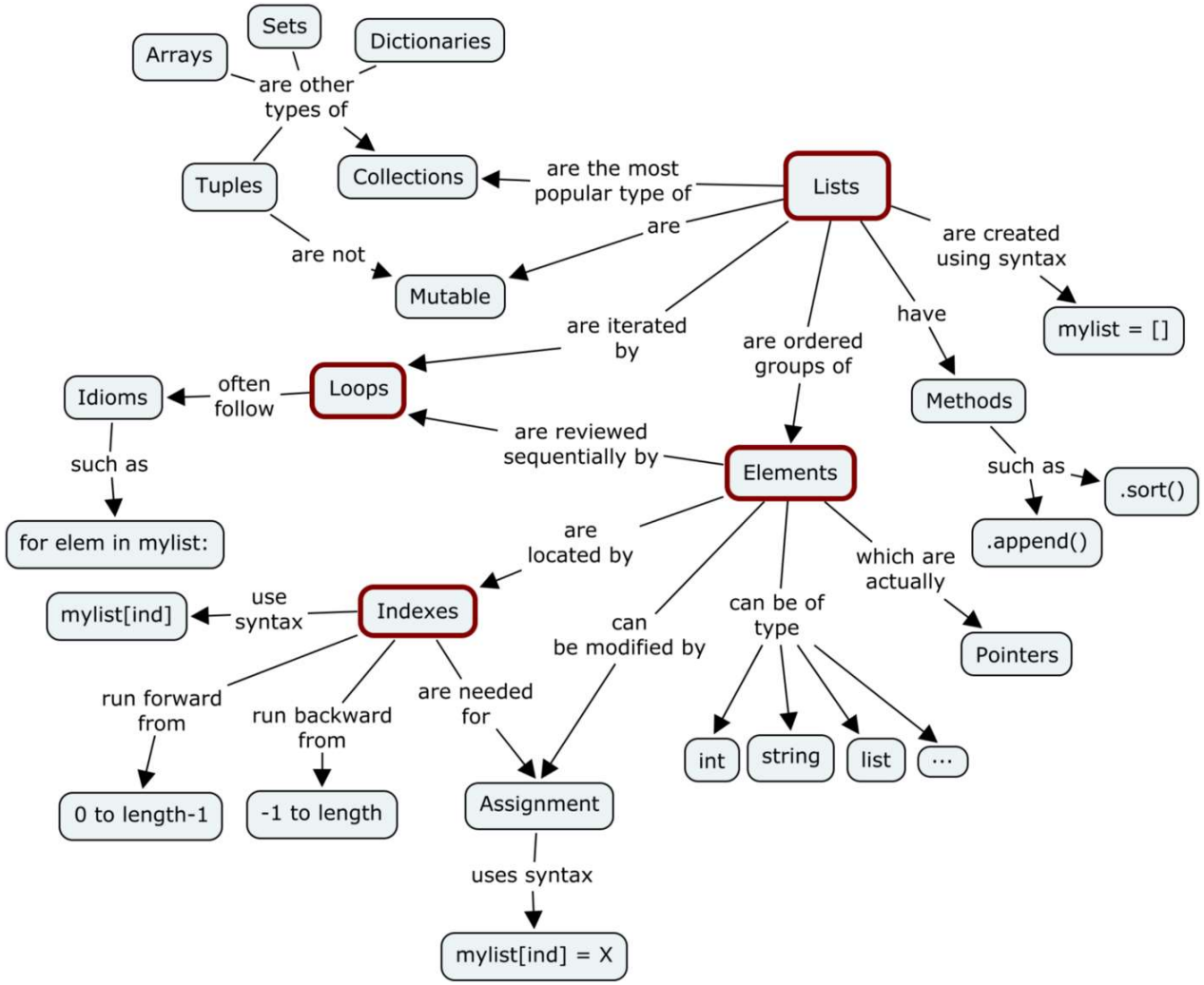
Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The `collections.abc.Sequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type, *n*, *i*, *j* and *k* are integers and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+` (concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations. [3]

Operation	Result	Notes
`x in s`	`True` if an item of *s* is equal to *x*, else `False`	(1)
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`	(1)
`s + t`	the concatenation of *s* and *t*	(6)(7)
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times	(2)(7)
`s[i]`	*i*th item of *s*, origin 0	(3)
`s[i:j]`	slice of *s* from *i* to *j*	(3)(4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3)(5)
`len(s)`	length of *s*	
`min(s)`	smallest item of *s*	
`max(s)`	largest item of *s*	
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)	(8)
`s.count(x)`	total number of occurrences of *x* in *s*	



https://medium.com/@meghamohan/mutable-and-immutable-side-of-python-c2145cf72747

Custom Iterator Class

```
class Count:
    def __init__(self, start, stop = 10):
        self.x = start-1
        self.y = stop

    def __iter__(self):
        return self
    def __next__(self):
        while (self.x < self.y):
            self.x += 1
            return self.x
        raise(StopIteration)
```

Presentation Formatting

Code format:

```
>>> def myfunc(a, b):
...     return a + b
...
>>> funcs = [myfunc]
>>> funcs[0]
<function myfunc at 0x107012230>
>>> funcs[0](2, 3)
>>> x = [-7,5.0]
>>> y = x
>>> x.append(257)
```