



# COMP1730/COMP6730

## Programming for Scientists

### Sequence types, part 2



# Lecture outline

- \* Lists (recap)
- \* Mutable objects & references

# Sequence data types (recap)

- \* A *sequence* contains  $n \geq 0$  values (its *length*), each at an *index* from 0 to  $n - 1$ .
- \* python's built-in sequence types:
  - strings (`str`) contain only characters;
  - lists (`list`) can contain a mix of value types;
  - tuples (`tuple`) are like lists, but immutable.
- \* Sequence types provided by other modules:
  - e.g., NumPy arrays (`numpy.ndarray`)

# Lists

- \* python's `list` is a general sequence type:  
elements in a `list` can be values of any type.
- \* List literals are written in square brackets with  
comma-separated elements:

```
>>> a_list_of_ints = [2, -4, 2, -8 ]
>>> a_date = [12, "August", 2015]
>>> pairs = [ [ 0.4, True ],
               [ "C", False ] ]
>>> type(pairs)
<class 'list'>
```

# Creating lists

```
>>> monday = [18, "July"]
>>> friday = [22, "July"]
>>> [monday, friday]
[[18, "July"], [22, "July"]]
>>> list("abcd")
['a', 'b', 'c', 'd']
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [1/x for x in range(1,6)]
[1.0, 0.5, 0.33333333, 0.25, 0.2]
```

# Lists of lists

```
>>> A = [ [1, 2, 3], [4, 5, 6],  
          [7, 8, 9] ]
```

```
>>> A[0]
```

```
[1, 2, 3]
```

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

```
3
```

```
>>> A[0][2]
```

```
3
```

- \* Indexing and slicing are *operators*
- \* Indexing and slicing associate to the left.  
`a_list[i][j] == (a_list[i])[j]`.

# Lists of lists

```
>>> A[0]
```

```
[1, 2, 3]
```

```
>>> A[0:1]
```

```
[ [1, 2, 3] ]
```

```
>>> A[0:1][1:]
```

```
[ ]
```

```
>>> A[0:1][1]
```

```
IndexError: list index out of range
```

- \* Indexing a list returns an element, but slicing a list returns a list.



# Mutable objects and references



# Values are objects

- \* In python, every value is an *object*.
- \* Every object has a unique<sup>(\*)</sup> identifier.

```
>>> id(1)
136608064
```

(Essentially, its location in memory.)

- \* *Immutable* objects never change.
  - For example, numbers (`int` and `float`) and strings.
- \* *Mutable* objects can change.
  - For example, arrays and lists.

# Immutable objects

- \* Operations on immutable objects create new objects, leaving the original unchanged.

```
>>> a_string = "spam"
>>> id(a_string)
3023147264
>>> b_string = a_string.replace('p', 'l')
>>> b_string
'slam'
>>> id(b_string)
3022616448
>>> a_string
'spam'
```

*not the same!*

# Mutable objects

- \* A mutable object can be modified yet it's identity remains the same.
- \* Lists and arrays can be modified through:
  - element and slice assignment; and
  - modifying methods/functions.
- \* `ndarray` and `list` is the only mutable types we have seen so far but there are many other (sets, dictionaries, user-defined classes).

# Element & slice assignment

```
>>> a_list = [1, 2, 3]
>>> id(a_list)
3022622348 ←
>>> b_list = a_list
>>> a_list[2] = 0
>>> b_list
[1, 2, 0]
>>> b_list[0:2] = ['A', 'B']
>>> a_list
['A', 'B', 0]
>>> id(b_list)
3022622348 ←
```

*the same object!*

# Modifying list methods

- \* `a_list.append(new element)`
- \* `a_list.insert(index, new element)`
- \* `a_list.pop(index)`
  - `index` defaults to `-1` (last element).
- \* `a_list.insert(index, new element)`
- \* `a_list.extend(an iterable)`
- \* `a_list.sort()`
- \* `a_list.reverse()`
- \* Note: Most do not return a value.

# Lists contain references

- \* Assignment associates a (variable) name with a *reference* to a value (object).
  - The variable still references the same object (unless reassigned) even if the object is modified.
- \* *A list contains references to its elements.*
- \* Slicing a list creates a new list, but containing references to the same objects (“shallow copy”).
- \* Slice assignment *does not copy*.

```
>>> a_list = [1,2,3]
>>> b_list = a_list
>>> a_list.append(4)
>>> print(b_list)
```

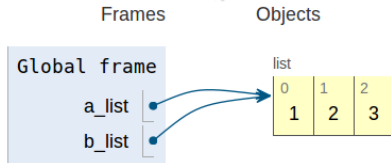


Image from [pythontutor.com](http://pythontutor.com)

```
>>> a_list = [1,2,3]
>>> b_list = a_list[:]
>>> a_list.append(4)
>>> print(b_list)
```

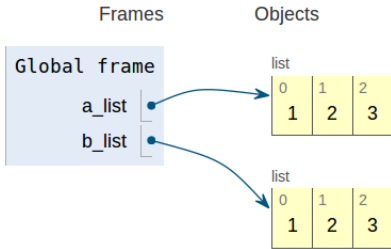
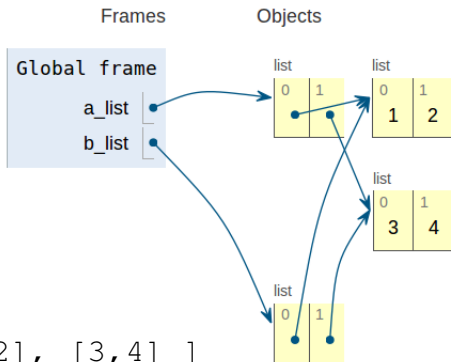


Image from [pythontutor.com](http://pythontutor.com)



```
>>> a_list = [ [1,2], [3,4] ]  
>>> b_list = a_list[:]  
>>> a_list[0].reverse()  
>>> b_list.reverse()  
>>> print(b_list)
```

Image from [pythontutor.com](http://pythontutor.com)



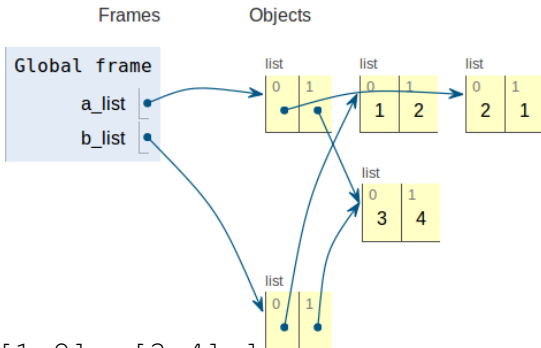


Image from [pythontutor.com](http://pythontutor.com)

```

>>> a_list = [ [1,2], [3,4] ]
>>> b_list = a_list[:]
>>> a_list[0] = a_list[0][::-1]
>>> b_list.reverse()
>>> print(b_list)

```

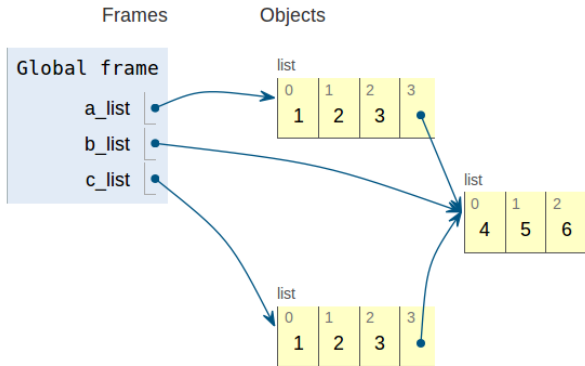


Image from [pythontutor.com](http://pythontutor.com)

```
>>> a_list = [1,2,3]
>>> b_list = [4,5,6]
>>> a_list.append(b_list)
>>> c_list = a_list[:]
>>> b_list[0] = 'A'
```

# Common mistakes

```
>>> a_list = [3, 1, 2]
>>> a_list = a_list.sort()
```

```
>>> a_list = [1, 2, 3]
>>> b_list = a_list
>>> a_list.append(b_list)
```

```
>>> a_list = [[]] * 3
>>> a_list[0].append(1)
```

# Shallow vs. deep copy

```
>>> import copy
>>> a_list = [[1,2], [3,4]]
>>> id(a_list)
3054870700
>>> id(a_list[0]), id(a_list[1])
(3054874028, 3073291596)
>>> b_list = a_list[:]
>>> id(b_list)
3072077420
>>> id(b_list[0]), id(b_list[1])
(3054874028, 3073291596)
>>> c_list = copy.deepcopy(a_list)
>>> id(c_list[0]), id(c_list[1])
(3057394764, 3057585932)
```

equal!

not equal!



# Never use deepcopy!

- \* Creating 10,000 copies of a list of 1,000 lists of 10 integers.

|              | Time  | Memory  |
|--------------|-------|---------|
| Shallow copy | 0.4s  | 39.3 MB |
| Deep copy    | 305 s | 1071 MB |