Module 15

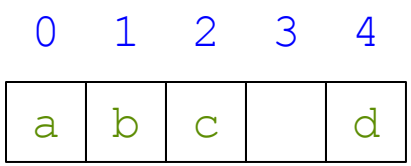
Sequences

Motivation for this Video Series

- Strings are a very, very useful type
- But they are also very *limited*
  - Break everything into individual letters
  - What if we want to work with numbers?
  - Or if want to work with words, not letters?
- This is going to require a new type
  - Let's look at what features strings have
  - See how to make them more general

Recall: String are Indexed

- s = 'abc d'

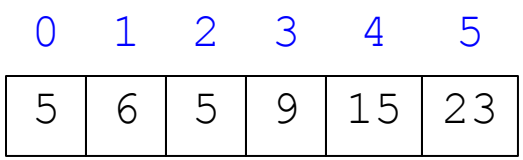


- Access chars with []
  - s[0] is 'a'
  - s[4] is 'd'
  - s[0:2] is 'ab' (no c)
  - s[2:] is 'c d'

- What are limitations?
- Slots: chars not words
  - **Ex:** 'Hello World'
  - Want word positions?
  - Needs many steps
- Cannot do numbers
  - **Ex:** '123, 456'
  - Only access digits

Tuple: Sequence of Value

- `x = (5, 6, 5, 9, 15, 23)`



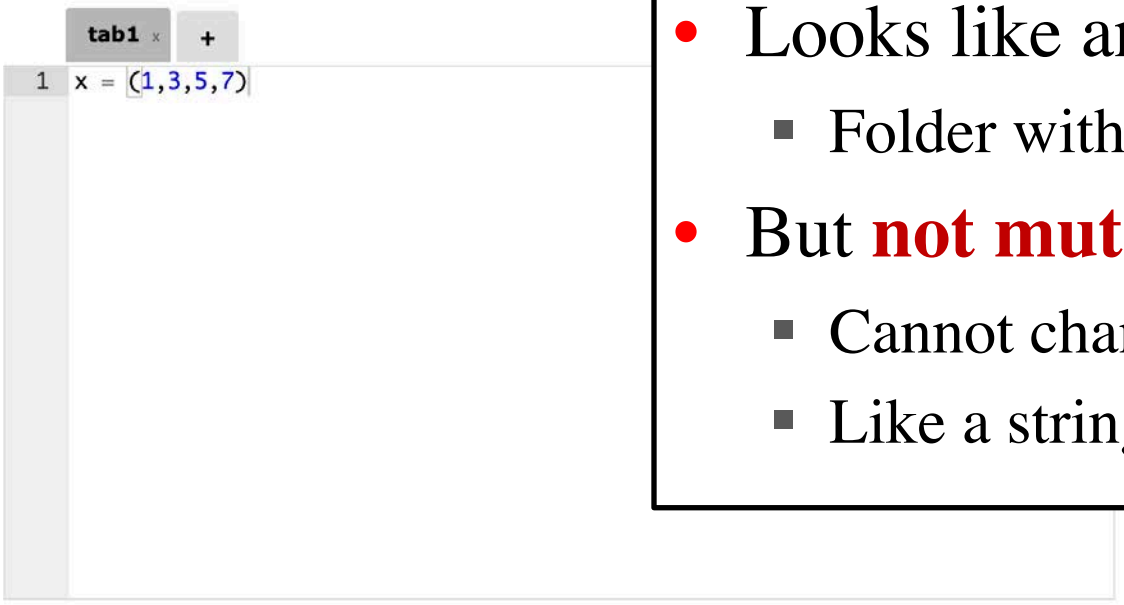
Inside parens, comma separated

- Access values with []
  - x[0] is 5
  - x[4] is 15
  - x[0:2] is (5,6)
  - x[3:] is (9,15,23)
- Can put anything in it
  - (True, False)
  - ('Hello', 'World')
- Can mix-and-match
  - (True, 1)
  - ('Hello', 3)

Two Tricky Things about Tuples

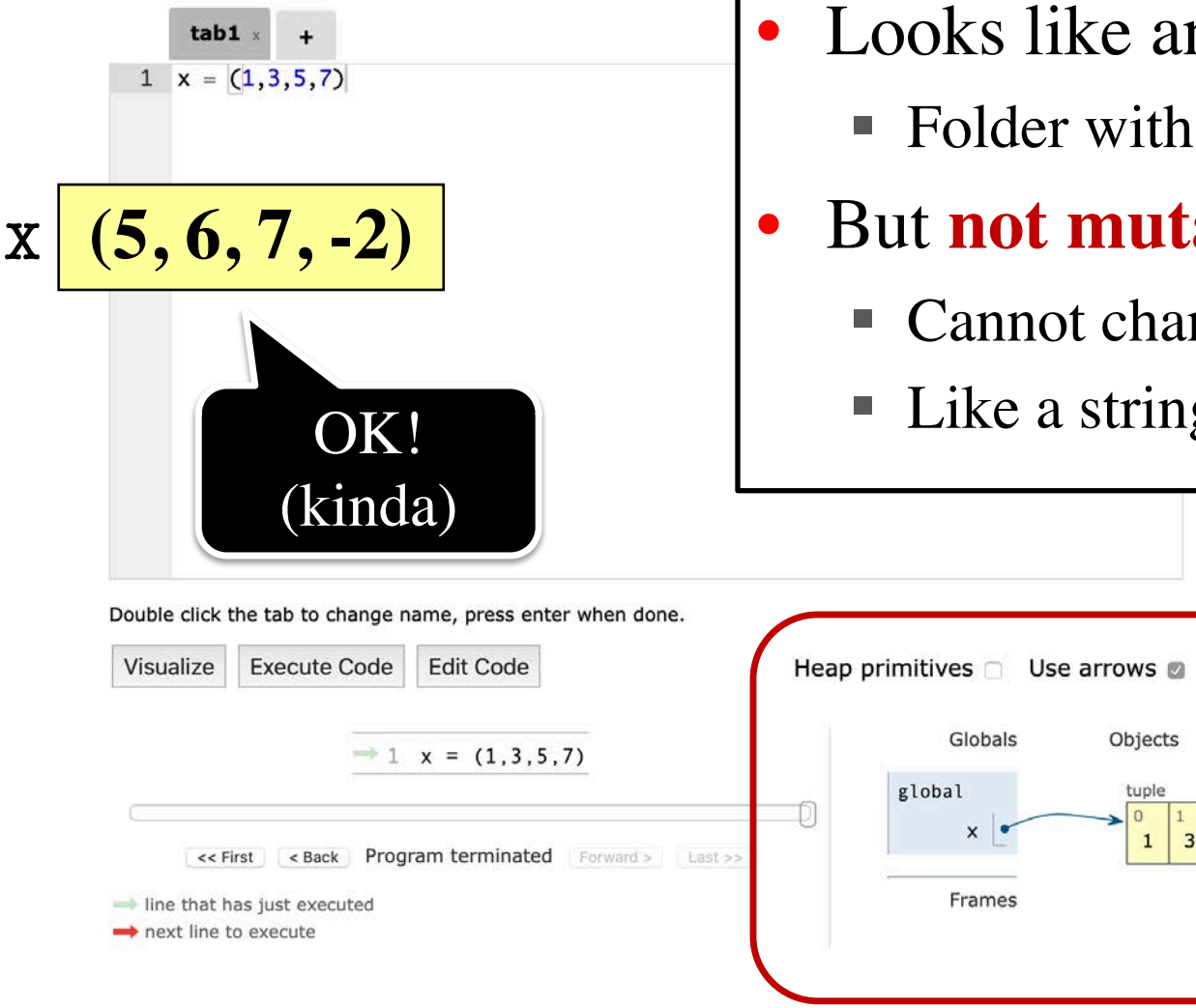
- What about an empty tuple?
  - Empty String: ''
  - Empty Tuple: ()
- What about a one element tuple?
  - Incorrect: (4) <= This is 4
  - Correct: (4,)
- But otherwise similar to strings

Tuples and the Python Tutor



- Looks like an *object*
  - Folder with id
- But **not mutable**
  - Cannot change contents
  - Like a string

Tuples and the Python Tutor



- Looks like an *object*
  - Folder with id
- But **not mutable**
  - Cannot change contents
  - Like a string

Tuples Support String-like Operations

- **Operation** +: x_1 + x_2
  - Glues if x_2 to end of x_1
  - Called *concatenation*
  - Evaluates to a tuple
- **Examples**:
  - (1,2) + (3,4) is (1,2,3,4)
  - (1,2) + (3,) is (1,2,3)
  - (1,2) + () is (1,2)

- **Operation** in: x_1 in x_2
  - Tests if x_1 "a value in" x_2
  - **Not** a subsequence
  - Evaluates to a boolean
- **Examples**:
  - 5 in (5,6,9) is True
  - 2 in (5,6,9) is False
  - (5,6) in (5,6,9) is False

Built-In Tuple Functions

- The `len` function
  - Returns length (# of elements) of tuple
  - **Example:** `len((1,2,3))` is `3`
- The `tuple` function
  - Converts a value to a tuple
  - Can only be applied to *iterable* types
  - Right now: strings and tuples
  - **Example:** `tuple('abc')` is `('a', 'b', 'c')`

Tuples Have Methods (Like Strings)

- **Example:** count
  - x.count(3) == 2
  - x.count(9) == 1
  - x.count(1) == 0
  - x.count(5) == 3
- **Example:** index
  - x.index(3) == 0
  - x.index(9) == 5
  - x.index(1) CRASHES
  - x.index(5) == 1

```
x = (3,5,3,5,5,9)
```

Just like string methods with the same name

Tuples and Expressions

- Tuple parens () can contain expressions
- Called a tuple **expression**
  - Python must evaluate it
  - Evaluates each expression
  - Puts the value in tuple
- Example:

```
>>> a = (1+2,3+4,5+6)
>>> a
(3, 7, 11)
```

- Execute the following:

```
>>> a = 5
>>> b = 7
>>> x = (a, b, a+b)
```

- What is x[2]?

A: 'a+b'
B: 12
C: 57
D: **ERROR**
E: I don't know

Tuples and Expressions

- Tuple parens () can contain expressions
- Called a tuple **expression**
  - Python must evaluate it
  - Evaluates each expression
  - Puts the value in tuple
- Example:

```
>>> a = (1+2,3+4,5+6)
>>> a
(3, 7, 11)
```

- Execute the following:

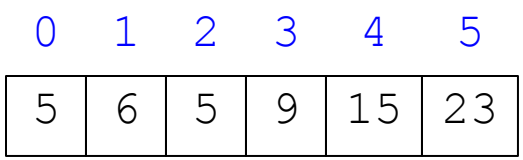
```
>>> a = 5
>>> b = 7
>>> x = (a, b, a+b)
```

- What is x[2]?

12

Lists are Almost the Same as Tuples

- `x = [5, 6, 5, 9, 15, 23]`



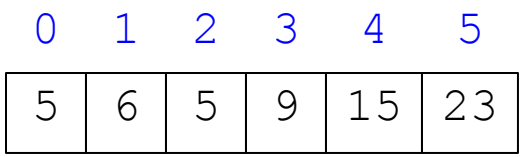
Inside **brackets**, comma separated

- Access values with []
  - x[0] is 5
  - x[4] is 15
  - x[0:2] is (5,6)
  - x[3:] is (9,15,23)
- Can put anything in it
  - [True, False]
  - ['Hello', 3]
- Expressions eval first

```
>>> [1+2, 4*2]
[3, 8]
```

Lists are Almost the Same as Tuples

- `x = [5, 6, 5, 9, 15, 23]`



Inside **brackets**, comma separated

- Access values with []
  - x[0] is 5
  - x[4] is 15
  - x[0:2] is (5,6)
  - x[3:]
- Can put anything in it
  - [True, False]
  - ['Hello', 3]
- Expressions eval first

But singletons are easier: [3]

Lists Operations are the Same

- **Operation** +: x_1 + x_2
  - [1,2] + [3,4] is [1,2,3,4]
  - [1,2] + [3] is [1,2,3]
  - [1,2] + [] is [1,2]
- **Functions** same(ish)
  - len([1,2,3]) is 3
  - list('abc') is ['a', 'b', 'c']
- **Operation** in: x_1 in x_2
  - 5 in [5,6,9] is True
  - 2 in [5,6,9] is False
  - [5,6] in [5,6,9] is False
- **Methods** are same
  - [1,2,1].count(1) is 2
  - [1,2,1].index(2) is 1

List [] Can Contain Expressions

- Called a list **expression** (just as with a tuple)
  - Python must evaluate it
  - Evaluates each expression
  - Puts the value in tuple
- Example:

```
>>> a = [1+2,3+4,5+6]
>>> a
[3, 7, 11]
```

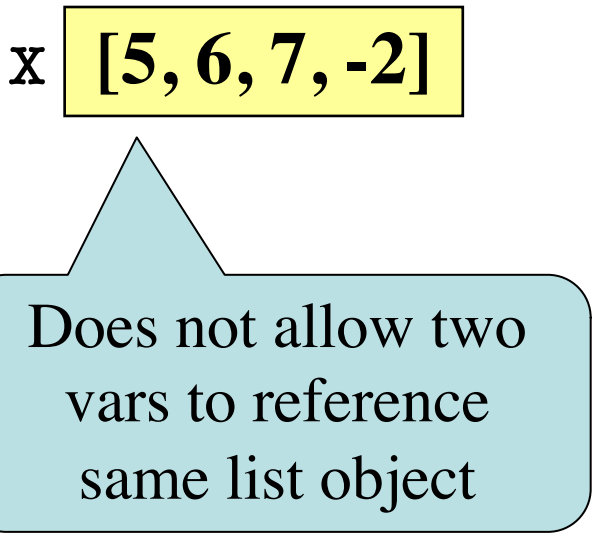
Aren't these redundant?

List, Tuples, Strings are Similar

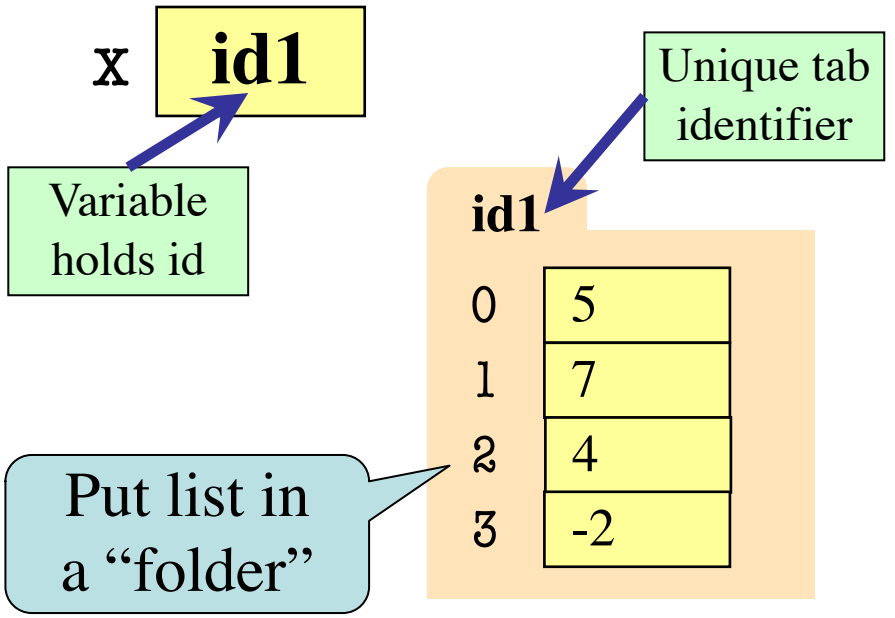
- Strings, tuples, lists are all **sequences**
  - A classification of a group of types
  - Means a type that can be sliced
- They are also all **iterables**
  - Means there is an order to the elements
  - Can access elements one at a time in order
- But only lists are **mutable**
  - You can reach into the folder and change

Representing Lists

Wrong



Correct



x = [5, 7, 4,-2]

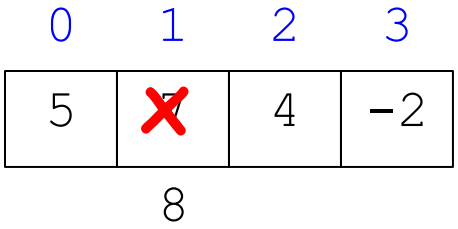
List Assignment

- **Basic Syntax**:

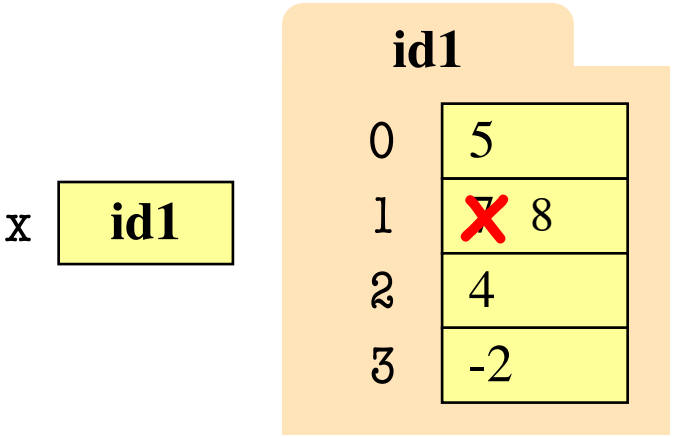
  <var>[<index>] = <value>

  - Reassign at index
  - Affects folder contents
  - Variable is unchanged
- Tuples cannot do this
  - x = (5, 7, 4, -2)
  - x[1] = 8 **ERROR**
  - Tuples are **immutable**

- x = [5, 7,4,-2]



- x[1] = 8



When Do We Need to Draw a Folder?

- When the value **contains** other values
  - This is essentially want we mean by 'object'
- When the value is **mutable**

Type	Container?	Mutable?
int	No	No
float	No	No
str	Yes*	No
Point3	Yes	Yes
RGB	Yes	Yes
list	Yes	Yes

When Do We Need to Draw a Folder?

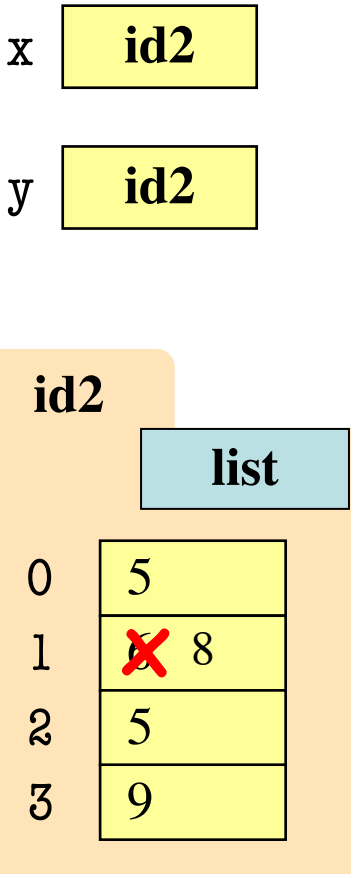
- When the value **contains** other values
  - This is essentially want we mean by 'object'
- When the value is **mutable**

Type	Container?	Mutable?
str	Yes*	No
Point3	Yes	Yes
RGB	Yes	Yes
list	Yes	Yes

tuples are a "grey area"

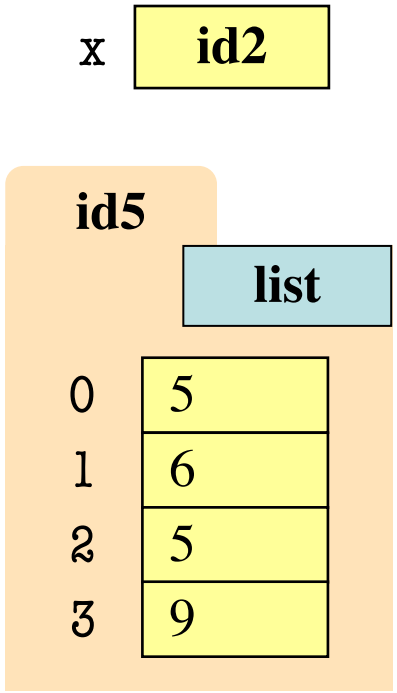
List Variables are Object Variables

```
>>> x = [5,6,5,9]
>>> y = x
>>> id(x)
4422305480
>>> id(y)
4422305480
>>> y[1] = 8
>>> x
[5,8,5,9]
```

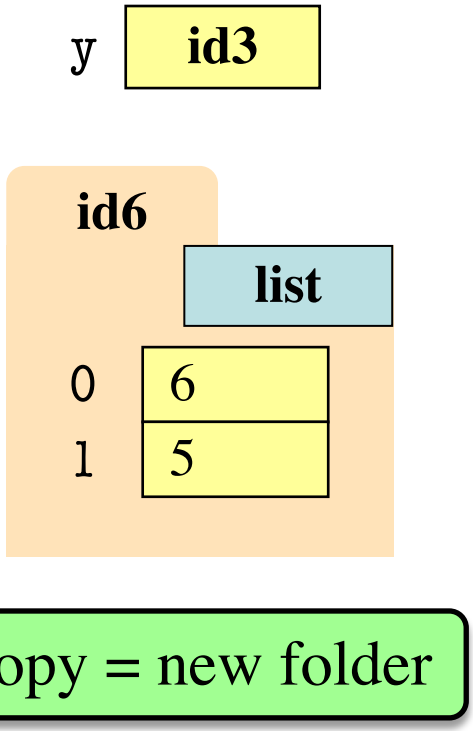


However, List Slices Make Copies

```
x = [5, 6, 5, 9]
```




```
y = x[1:3]
```



This is Why Lists are Advanced!

- You must pay close attention to the folder
  - Sometimes have a copy, sometimes do not
  - Do not always want to modify the original
  - Reason degenerate slicing is useful: `x[:]`
- If in doubt use the **Python Tutor**
  - Lists are a major reason it is so useful
- But need to learn to work without

Lists Share Methods with Tuple

```
x = [5, 6, 5, 9, 15, 23]
```

- index(value)
  - Return position of the value
  - **ERROR** if value is not there
  - x.index(9) evaluates to 3
- count(value)
  - Returns number of times value appears in list
  - x.count(5) evaluates to 2

These are **immutable** methods

List Methods Can Alter the List

```
x = [5, 6, 5, 9]
```

- append(value)
  - A **procedure method**, not a fruitful method
  - Adds a new value to the end of list
  - `x.append(-1)` *changes* the list to `[5, 6, 5, 9, -1]`
- insert(index, value)
  - Put the value into list at index; shift rest of list right
  - `x.insert(2,-1)` changes the list to `[5, 6, -1, 5, 9,]`
- sort()

What do you think this does?

Where To Learn About List Methods?

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*iterable*)
Extend the list by appending all the items from the

In the documentation!

`list.`**`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)
Remove the first item from the list whose value is equal to *x*. It raises a `ValueError` if there is no such item.

`list.`**`pop`**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

Recall: Mutable Functions

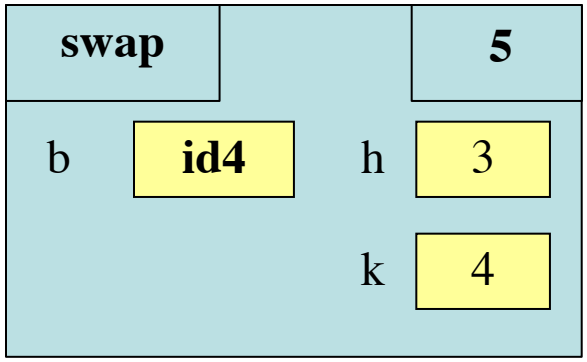
- A **mutable function** alters an object parameter
  - Often a procedure; no return value
  - Possible because folders persist outside frame
- Lists are mutable objects too!
  - So we can make functions to alter them
  - One of main reasons to use lists over tuples
- Often for matters of efficiency
  - Changing a tuple requires a complete copy
  - Expensive if the tuple is large

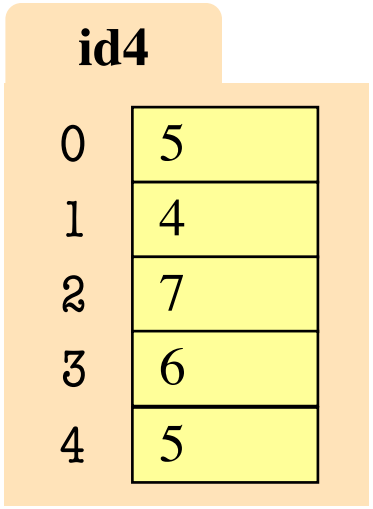
Lists and Functions: Swap

```
1.  def swap(b, h, k):
2.      """ Swaps b[h] and b[k] in b
3.      Precond: b is a mutable list,
4.      h, k are valid positions"""
5.      temp= b[h]
6.      b[h]= b[k]
7.      b[k]= temp
```

Swaps b[h] and b[k], because parameter b contains name of list.

swap(x, 3, 4)





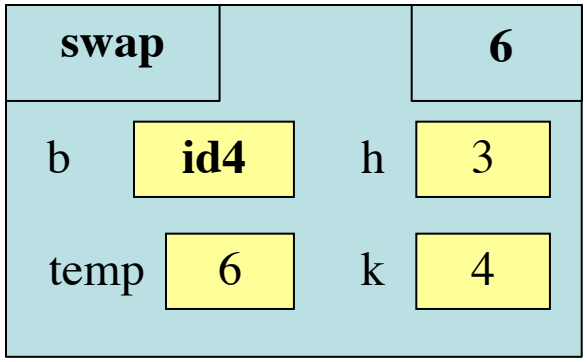
x | id4

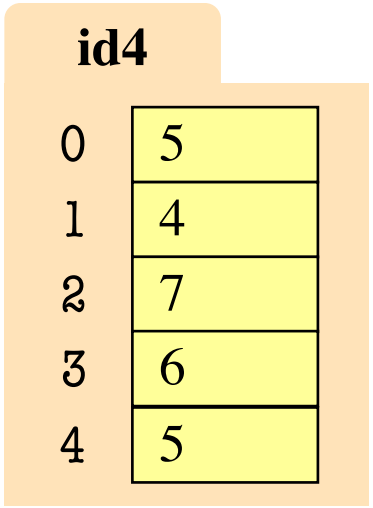
Lists and Functions: Swap

```
1.  def swap(b, h, k):
2.      """ Swaps b[h] and b[k] in b
3.      Precond: b is a mutable list,
4.      h, k are valid positions"""
5.      temp= b[h]
6.      b[h]= b[k]
7.      b[k]= temp
```

Swaps b[h] and b[k], because parameter b contains name of list.

swap(x, 3, 4)





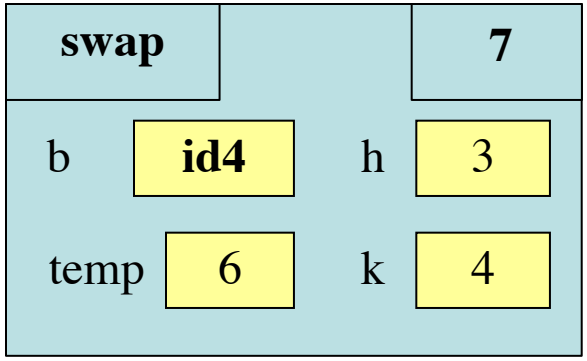
x | id4

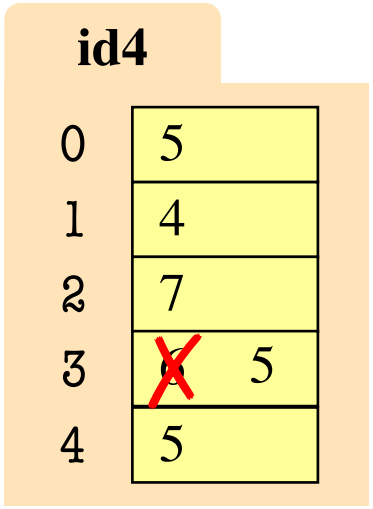
Lists and Functions: Swap

```
1.  def swap(b, h, k):
2.      """ Swaps b[h] and b[k] in b
3.      Precond: b is a mutable list,
4.      h, k are valid positions"""
5.      temp= b[h]
6.      b[h]= b[k]
7.      b[k]= temp
```

Swaps b[h] and b[k], because parameter b contains name of list.

swap(x, 3, 4)





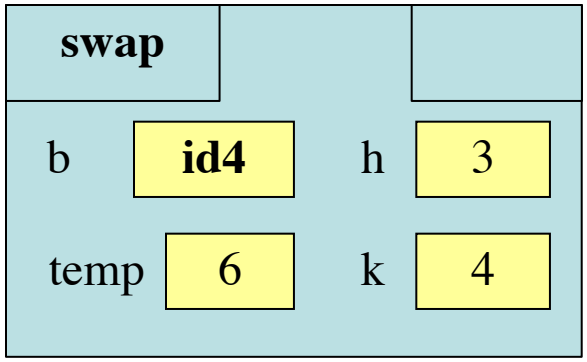
x | id4

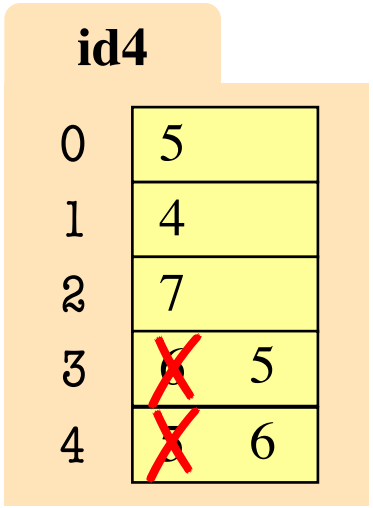
Lists and Functions: Swap

```
1.  def swap(b, h, k):
2.      """ Swaps b[h] and b[k] in b
3.      Precond: b is a mutable list,
4.      h, k are valid positions"""
5.      temp= b[h]
6.      b[h]= b[k]
7.      b[k]= temp
```

Swaps b[h] and b[k], because parameter b contains name of list.

swap(x, 3, 4)





x | id4

Slice Assignment

- List assignment not limited to one element
  - Slicing accesses several elements at once
  - Can use slicing to assign several at once
- **This is a very advanced topic**
  - Will never need this in this course
  - Just showing it for completeness
  - Something that is **very** unique to Python

Slice Assignment

- Can *embed* a new list inside of a list
  - **Syntax:** <var>[<start>:<end>] = <list>
  - Replaces that range with content of list
- **Example:**

```
>>> a = [1,2,3]
>>> b = [4,5]
>>> a[:2] = b
>>> a
[4, 5, 3]
```

Replaces [1,2] with [4,5]

Some Advanced Techniques

- Range and list size need not match

```
>>> a = [1,2,3]
>>> b = [4,5]
>>> a[:1] = b
>>> a
[4, 5, 2, 3]
```

Stretches list to fit

- Assigned value can be any iterable

```
>>> a = [1,2,3]
>>> a[:2] = 'hi'
>>> a
['h', 'i', 3]
```

Converts to list