Lecture 5

Objects and Lists

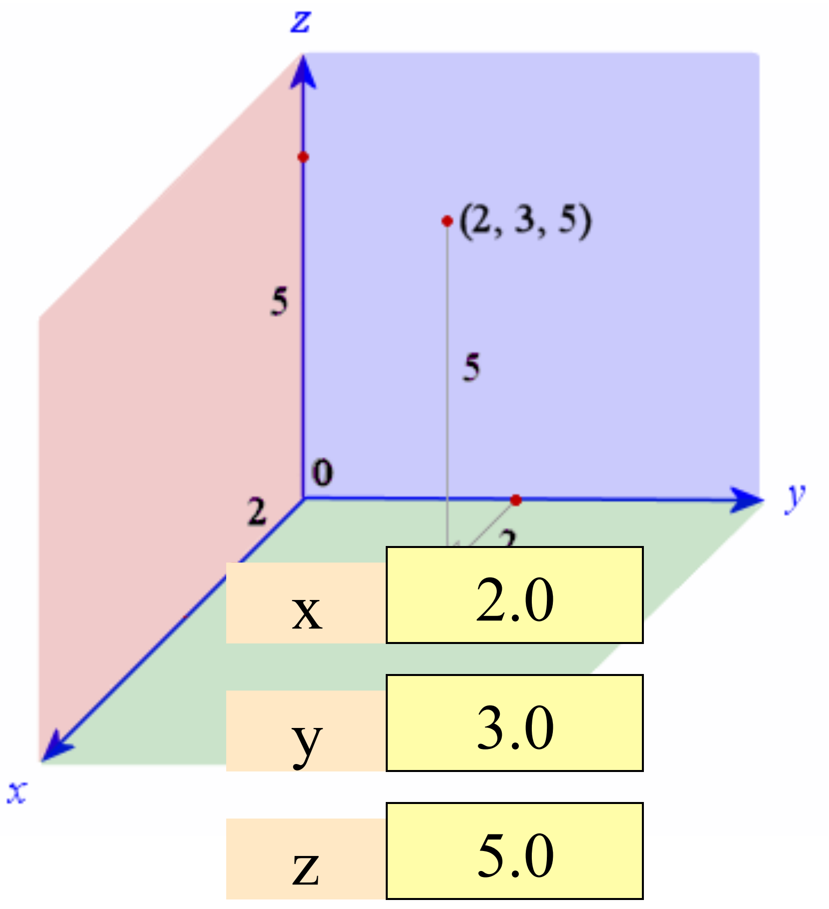
Type: Set of values and the operations on them

- Type **int**:
  - **Values**: integers
  - **Ops**: +, –, *, /, %, **
- Type **float**:
  - **Values**: real numbers
  - **Ops**: +, –, *, /, **
- Type **bool**:
  - **Values**: **True** and **False**
  - **Ops**: not, and, or
- Type **str**:
  - **Values**: string literals
    - Double quotes: `"abc"`
    - Single quotes: `'abc'`
  - **Ops**: + (concatenation)

Are the the only types that exist?

Type: Set of values and the operations on them

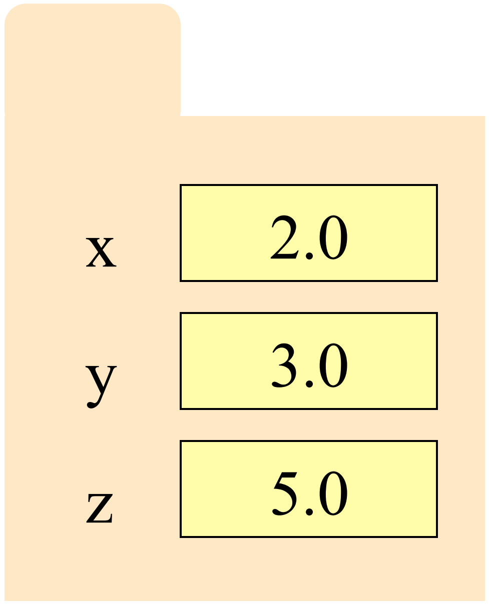
- Want a point in 3D space
  - We need three variables
  - *x*, *y*, *z* coordinates
- What if have a lot of points?
  - Vars x0, y0, z0 for first point
  - Vars x1, y1, z1 for next point
  - …
  - This can get really messy
- How about a single variable that represents a point?



Type: Set of values and the operations on them

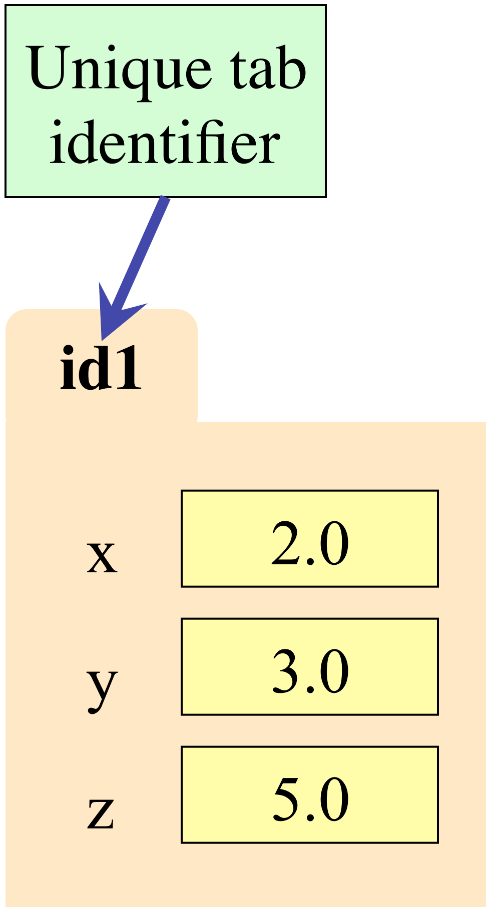
- Want a point in 3D space
  - We need three variables
  - *x*, *y*, *z* coordinates
- What if have a lot of points?
  - Vars x0, y0, z0 for first point
  - Vars x1, y1, z1 for next point
  - …
  - This can get really messy
- How about a single variable that represents a point?

- Can we stick them together in a "folder"?
- Motivation for **objects**



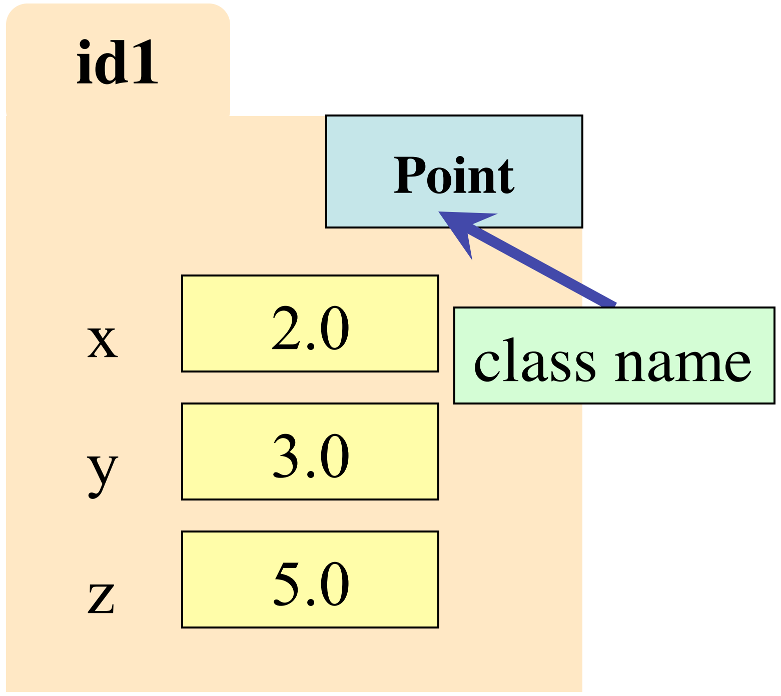
Objects: Organizing Data in Folders

- An object is like a **manila folder**
- It contains other variables
  - Variables are called **attributes**
  - These values can change
- It has an **ID** that identifies it
  - Unique number assigned by Python (just like a NetID for a Cornellian)
  - Cannot ever change
  - Has no meaning; only identifies



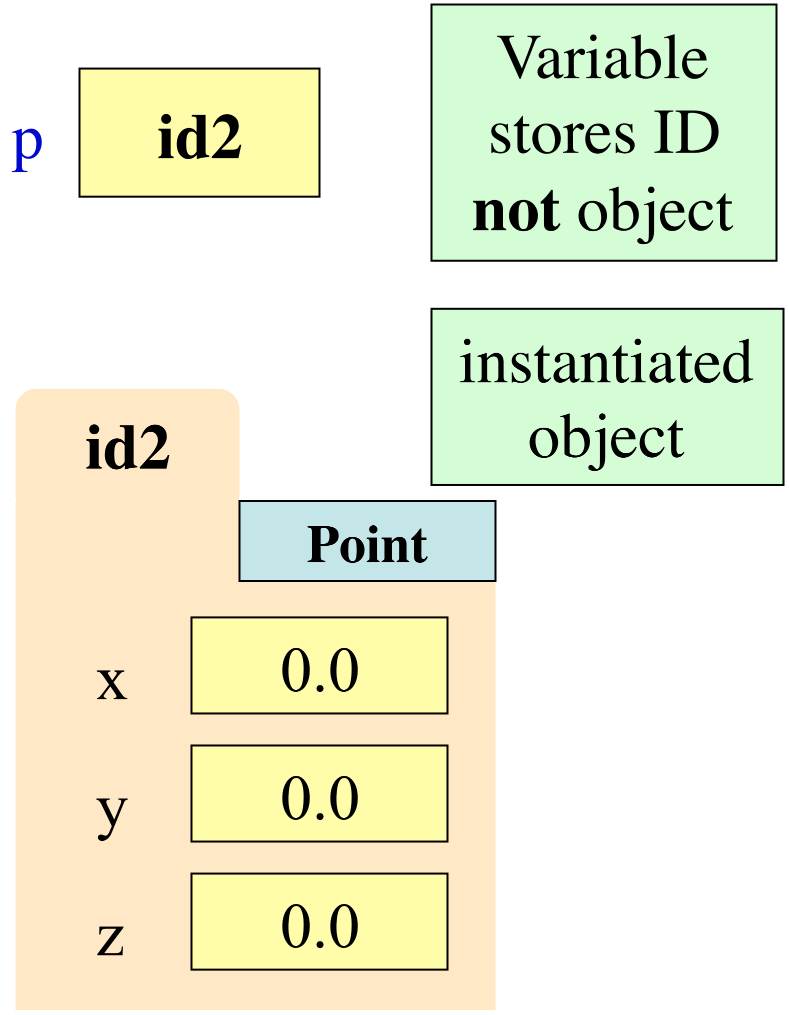
Classes: Types for Objects

- Values must have a type
  - An object is a **value**
  - Object type is a **class**
- **Modules** provide classes
  - Will show how later
- **Example**: `tuple3d`
  - Part of CornellExtensions
  - Just need to import it
  - Classes: `Point`, `Vector`



Constructor: Function to make Objects

- How do we create objects?
  - Other types have **literals**
  - **Example**: `1`, `"abc"`, **true**
  - No such thing for objects
- **Constructor Function**:
  - Same name as the class
  - **Example**: `Point(0,0,0)`
  - Makes an object (manila folder)
  - Returns folder ID as value
- **Example**: `p = Point(0, 0, 0)`
  - Creates a Point object
  - Stores object's ID in p



Constructors and Modules

```
>>> import tuple3d
```

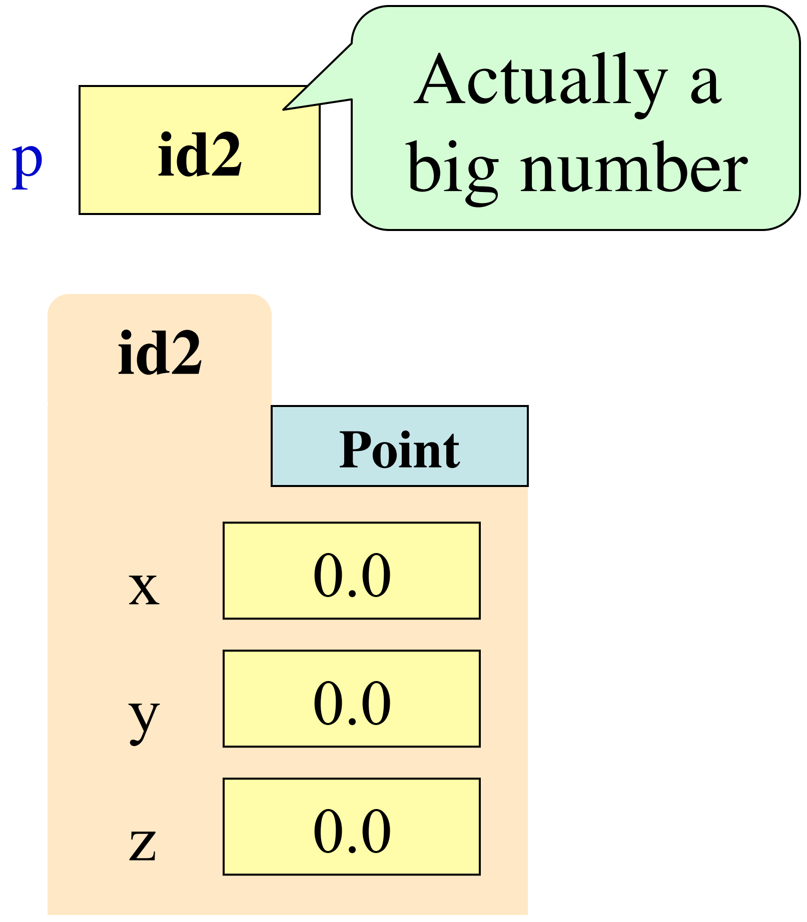
Need to import module that has Point class.

```
>>> p = tuple3d.Point(0,0,0)
```

Constructor is function. Prefix w/ module name.

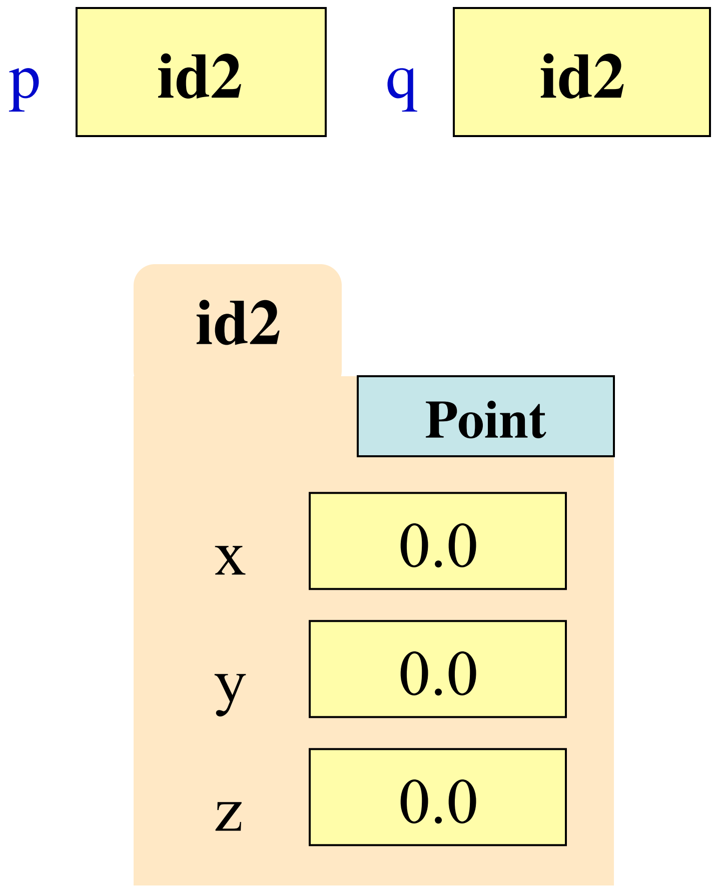
```
>>> id(p)
```

Shows the ID of p.



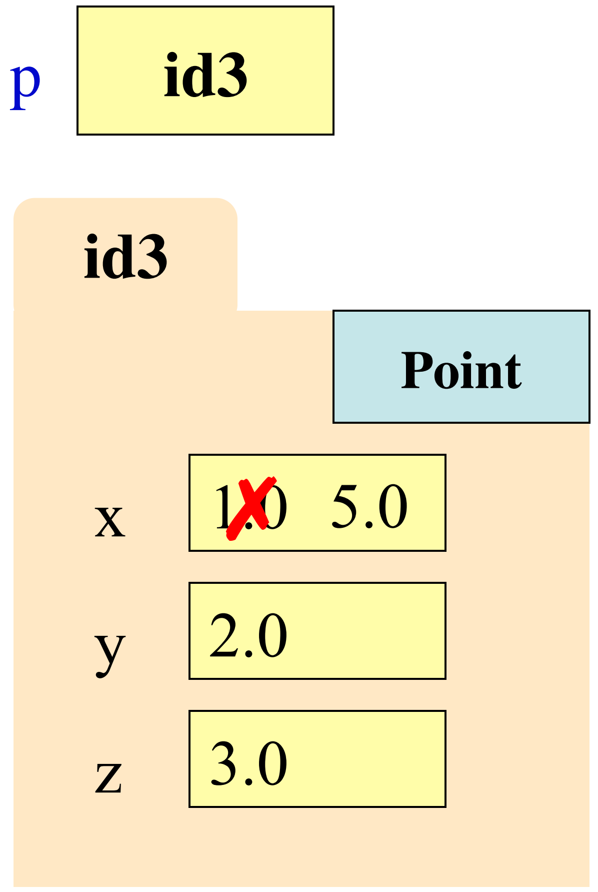
Object Variables

- Variable stores object name
  - **Reference** to the object
  - Reason for folder analogy
- Assignment uses object name
  - **Example**: q = p
  - Takes name from p
  - Puts the name in q
  - Does not make new folder!
- This is the cause of many mistakes in this course



Objects and Attributes

- Attributes are variables that live inside of objects
  - Can **use** in expressions
  - Can **assign** values to them
- **Access**: `<variable>.<attr>`
  - **Example**: `p.x`
  - Look like module variables
- Putting it all together
  - `p = tuple3d.Point(1,2,3)`
  - `p.x = p.y + p.z`



Exercise: Attribute Assignment

- Recall, `q` gets name in `p`

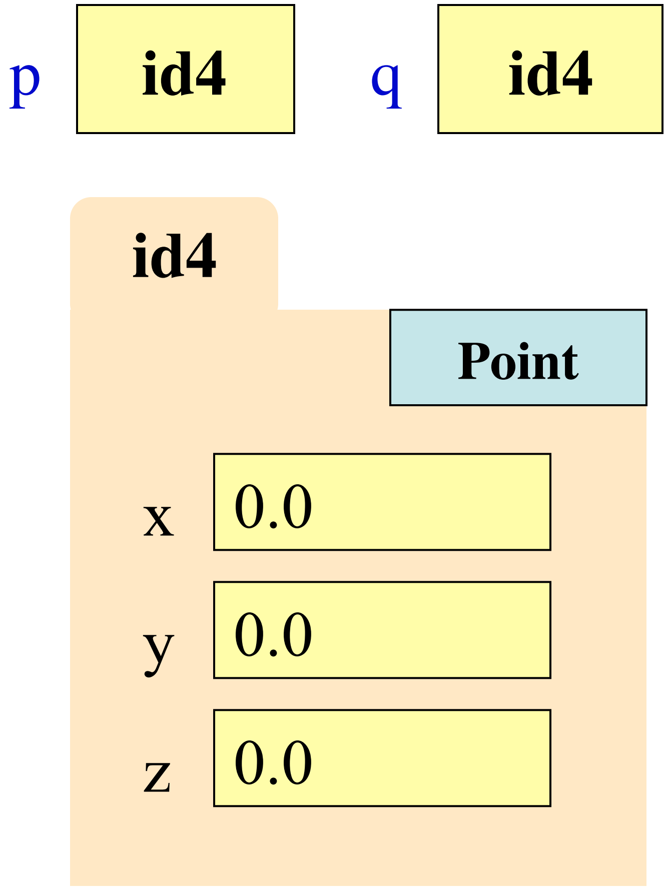
```
>>> p = tuple3d.Point(0,0,0)
>>> q = p
```

- Execute the assignments:

```
>>> p.x = 5.6
>>> q.x = 7.4
```

- What is value of `p.x`?

A: 5.6
B: 7.4
C: **id4**
D: I don't know



Exercise: Attribute Assignment

- Recall, `q` gets name in `p`

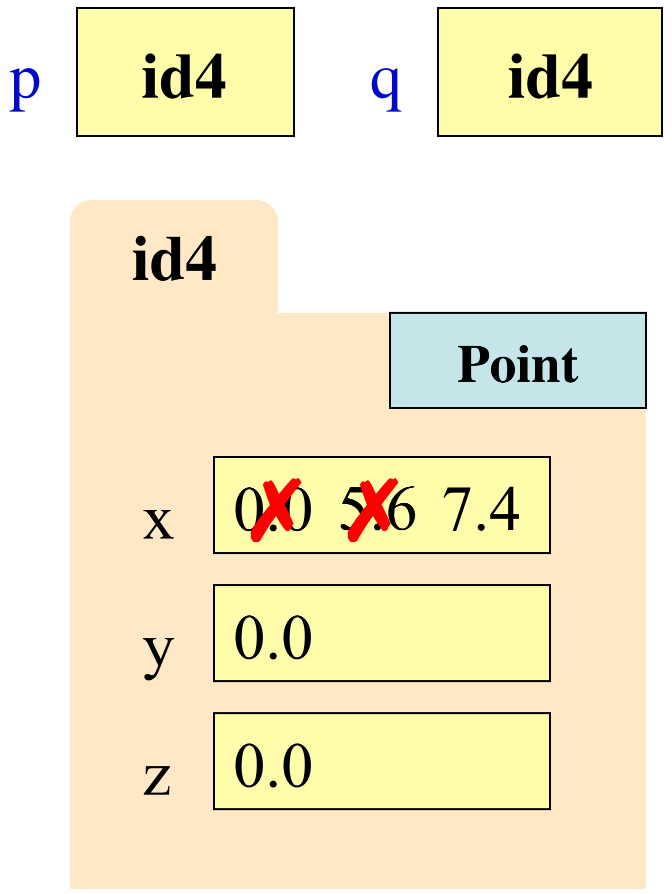
```
>>> p = tuple3d.Point(0,0,0)
>>> q = p
```

- Execute the assignments:

```
>>> p.x = 5.6
>>> q.x = 7.4
```

- What is value of `p.x`?

A: 5.6
B: 7.4 **CORRECT**
C: **id4**
D: I don't know



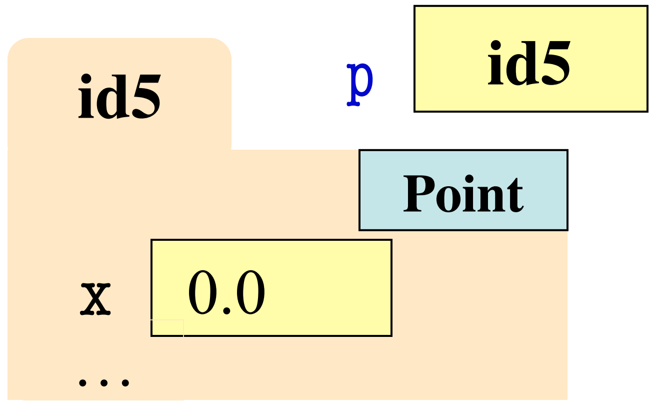
Call Frames and Objects

- Mutable objects can be altered in a function call
  - Object vars hold names!
  - Folder accessed by both global var & parameter
- **Example**:

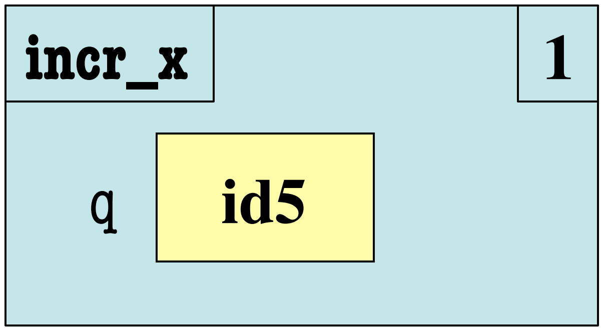
```
  def incr_x(q):
1     q.x = q.x + 1
```

```
>>> p = Point(0,0,0)
>>> incr_x(p)
```

Global **STUFF**



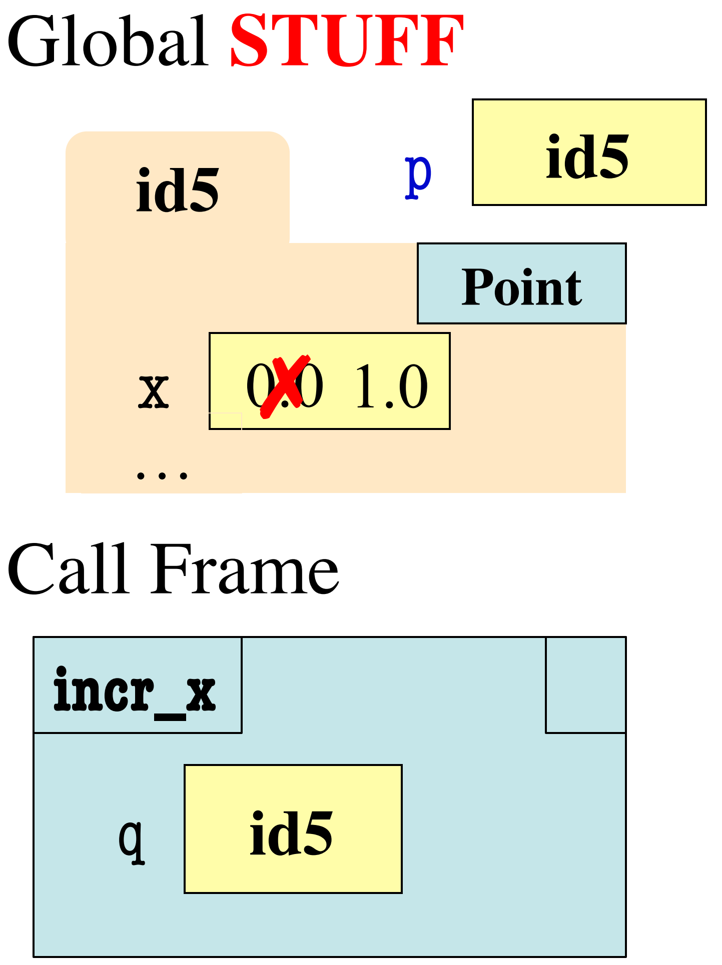
Call Frame



Call Frames and Objects

- Mutable objects can be altered in a function call
  - Object vars hold names!
  - Folder accessed by both global var & parameter
- **Example**:

```
def incr_x(q):
1       q.x = q.x + 1
>>> p = Point()
>>> incr_x(p)
```



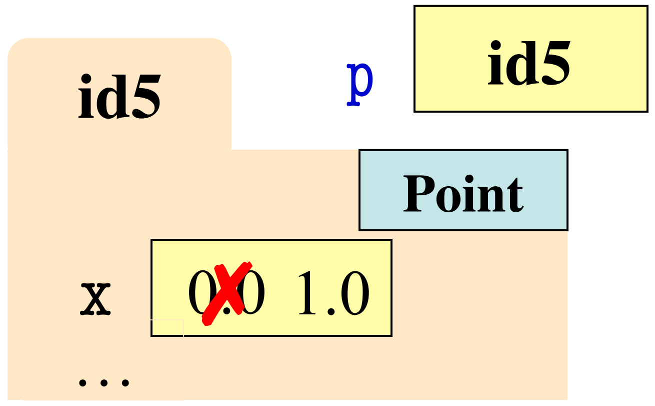
Call Frames and Objects

- Mutable objects can be altered in a function call
  - Object vars hold names!
  - Folder accessed by both global var & parameter
- **Example**:

```
def incr_x(q):
1       q.x = q.x + 1
```

```
>>> p = Point()
>>> incr_x(p)
```

Global **STUFF**

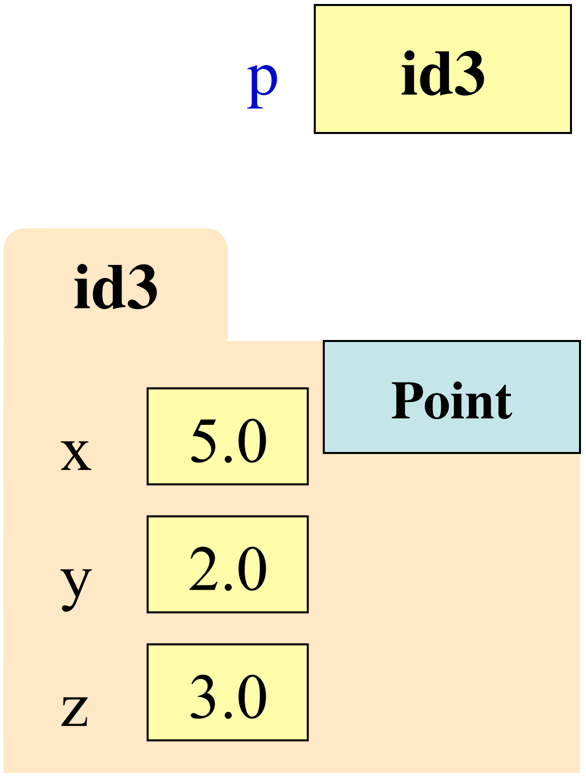


Call Frame

Methods: Functions Tied to Objects

- **Method**: function tied to object
  - Method call looks like a function call preceded by a variable name:

    ⟨*variable*⟩.⟨*method*⟩(⟨*arguments*⟩)
  - **Example**: `p.distanceTo(q)`
  - **Example**: `p.abs() # makes x,y,z ≥ 0`
- Just like we saw for strings
  - `s = 'abracadabra'`
  - `s.index('a')`
- Are strings objects?

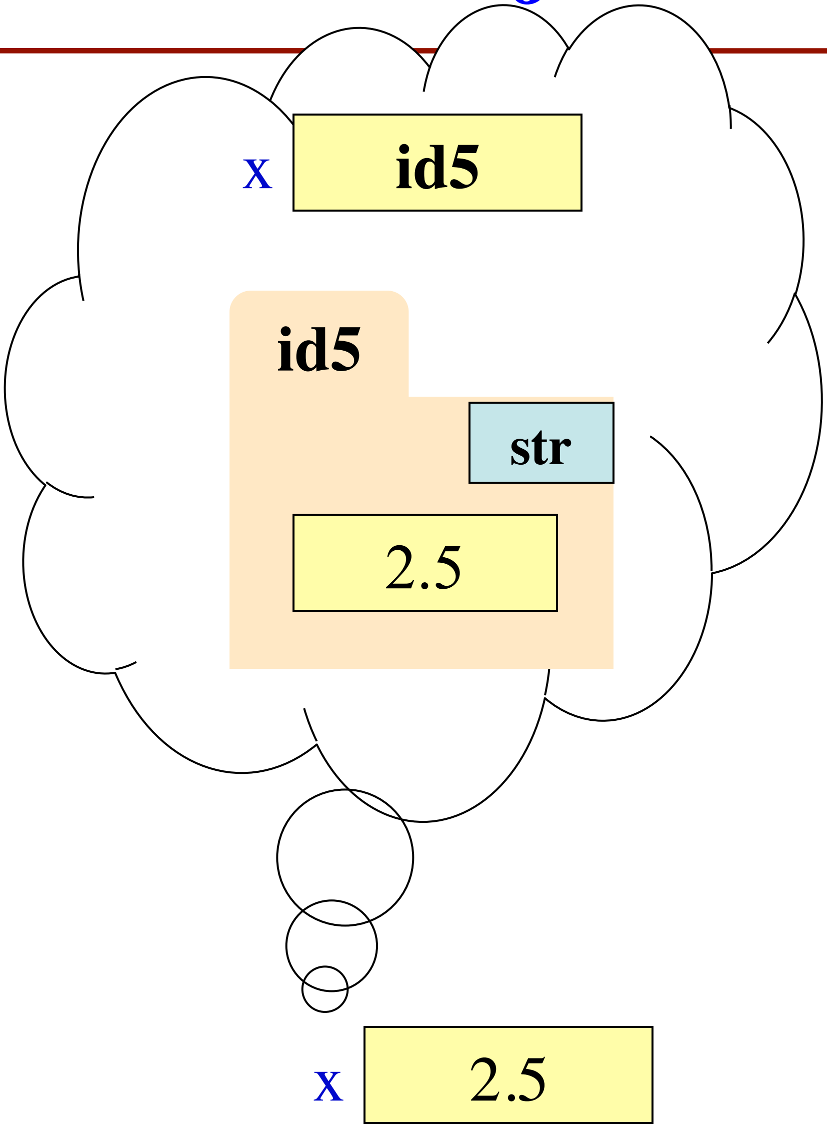


Surprise: All Values are in Objects!

- Including basic values
  - int, float, bool, str
- **Example**:

```
>>> x = 2.5
>>> id(x)
```

- But they are *immutable*
  - Contents cannot change
  - Distinction between *value* and *identity* is immaterial
  - So we can ignore the folder



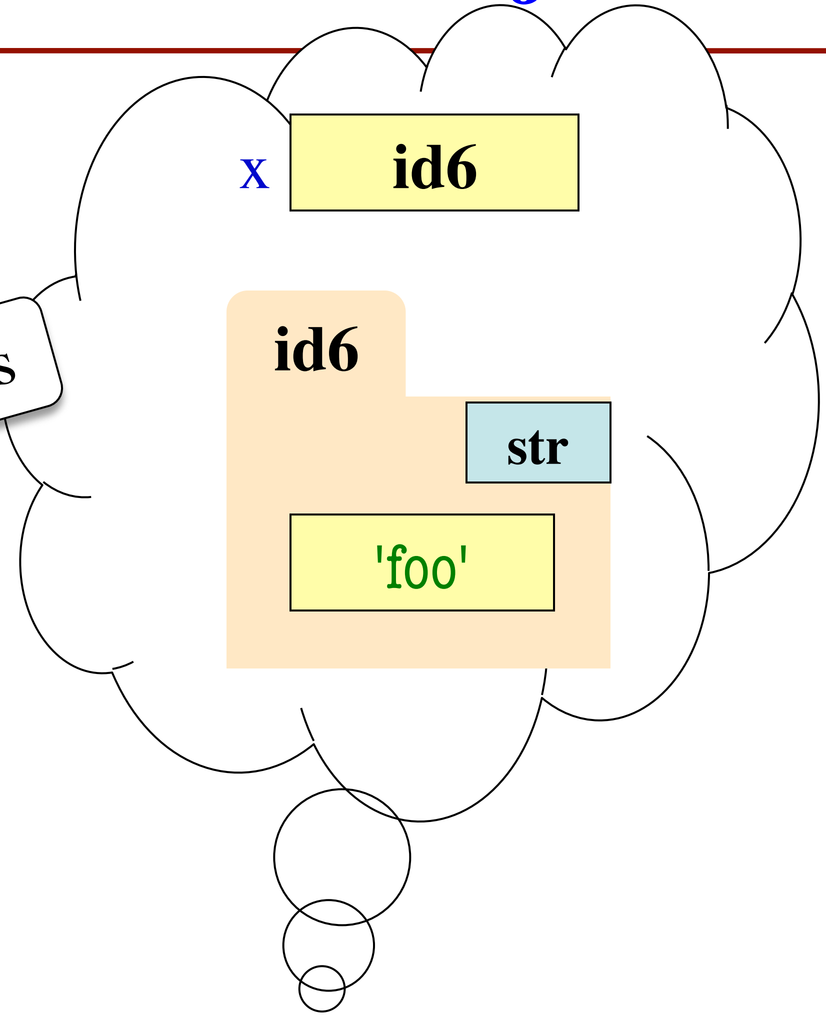
Surprise: All Values are in Objects!

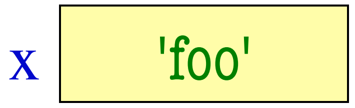
- Including basic values
  - int, float, bool, str
- **Example**:

```
>>> x = 'foo'
>>> id(x)
```

includes strings

- But they are *immutable*
  - No string method can alter the contents of a string
  - x.replace('o','y') evaluates to 'fyy' but x is still 'foo'
  - So we can ignore the folder

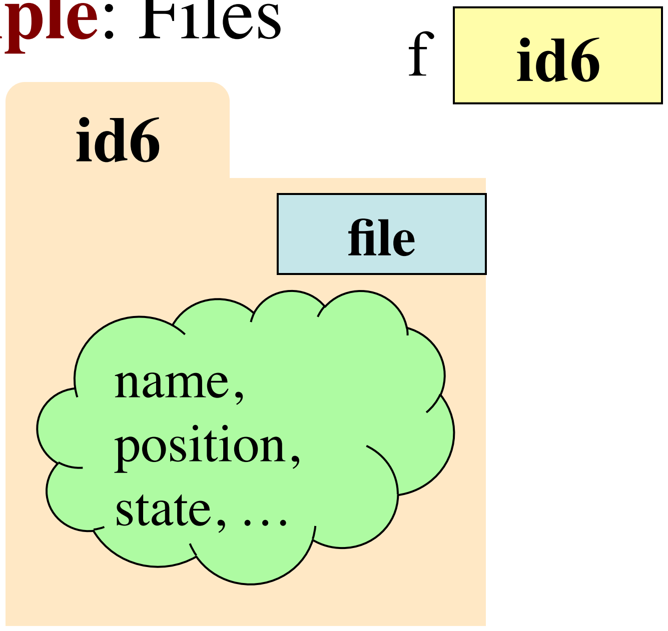




Class Objects

- Use name **class object** to distinguish from other values
  - Not int, float, bool, str
- Class objects are **mutable**
  - You can change them
  - Methods can have effects besides their return value
- **Example**:
  - p = Point(3,-3,0)
  - p.clamp(-1,1)

Example: Files



```
f = open('jabber.txt')
s = f.read()
f.close()
```

Opens a file on your disk; returns a **file object** you can read

Base Types vs. Classes

Base Types	Classes
Built-into Python	Provided by modules
Refer to instances as *values*	Refer to instances as *objects*
Instantiate with *literals*	Instantiate w/ *constructors*
Are all immutable	Can alter attributes
Can ignore the folders	Must represent with folders

Sequences: Lists of Values

String

- s = 'abc d'

0	1	2	3	4
a	b	c		d

- Put characters in quotes
  - Use \' for quote character
- Access characters with []
  - s[0] is 'a'
  - s[5] causes an error
  - s[0:2] is 'ab' (excludes c)
  - s[2:] is 'c d'

List

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

0	1	2	3	4	5
5	6	5	9	15	23

- Put values inside [ ]
  - Separate by commas
- Access **values** with []
  - x[0] is 5
  - x[6] causes an error
  - x[0:2] is [5, 6] (excludes 2nd 5)
  - x[3:] is [9, 15, 23]

Sequences: Lists of Values

String

- s = 'abc d'

0	1	2	3	4
a	b	c		d

- Put characters in quotes
  - Use \' for quote character
- Access ch
  - s[0] is 'a
  - s[5] causes
  - s[0:2] is 'ab' (excludes c)
  - s[2:] is 'c d'

List

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

0	1	2	3	4	5
5	6	5	9	15	23

- Put values inside [ ]
  - mmas
- with []
  - [0] is 5
  - x[6] causes an error
  - x[0:2] is [5, 6] (excludes 2nd 5)
  - x[3:] is [9, 15, 23]

Sequence is a name we give to both

Lists Have Methods Similar to String

```
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

But you get length of a list with a regular function, not method:

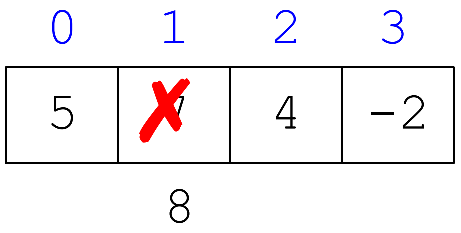
`len(x)`

Lists are Mutable

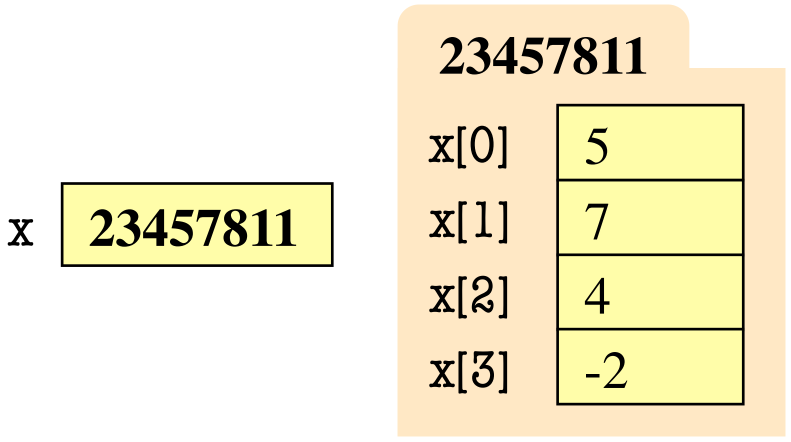
- Can alter their contents
  - Use an assignment:

    <*var*>[<*index*>] = <*value*>
  - Index is position, not slice
- Does not work for strings
  - s = 'Hello World!'
  - s[0] = 'J' **ERROR**
- Represent list as a folder
  - Variable holds tab name
  - Contents are attributes

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



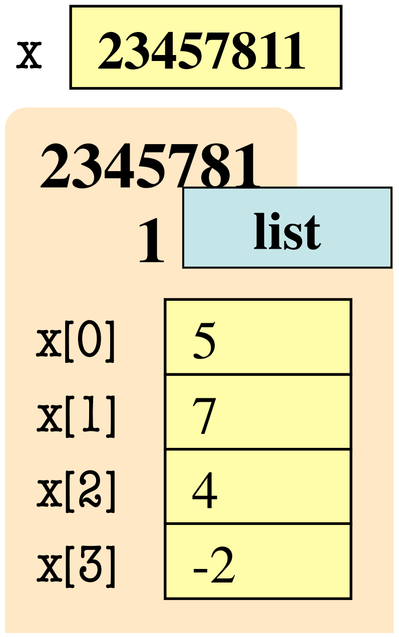
- `x[1] = 8`



Lists vs. Custom Objects

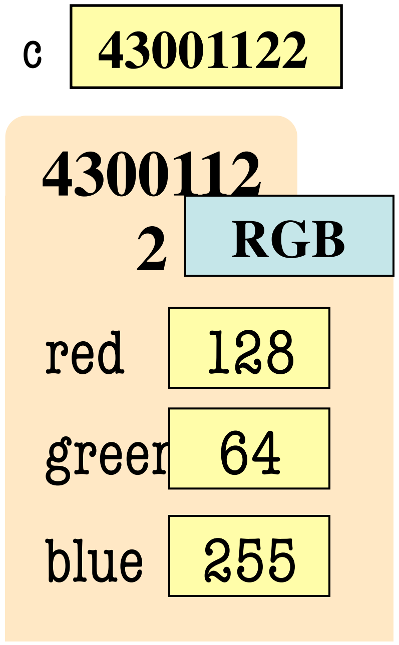
List

- Attributes are indexed
  - Example: x[2]



RGB

- Attributes are named
  - Example: c.red



List Methods Can Alter the List

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

See Python API for more

- 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()

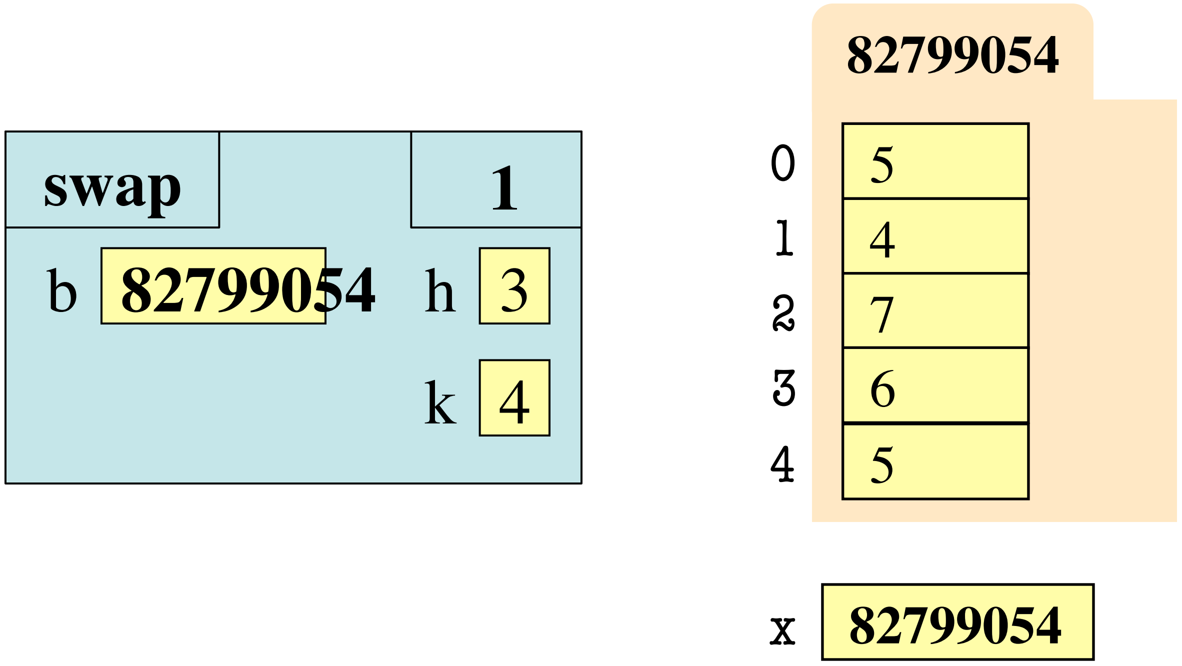
What do you think this does?

Lists and Functions: Swap

```
def swap(b, h, k):
    """Procedure swaps b[h] and b[k] in b
       Precondition: b is a mutable list, h
       and k are valid positions in the list"""
1   temp= b[h]
2   b[h]= b[k]
3   b[k]= temp
```

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

```
swap(x, 3, 4)
```

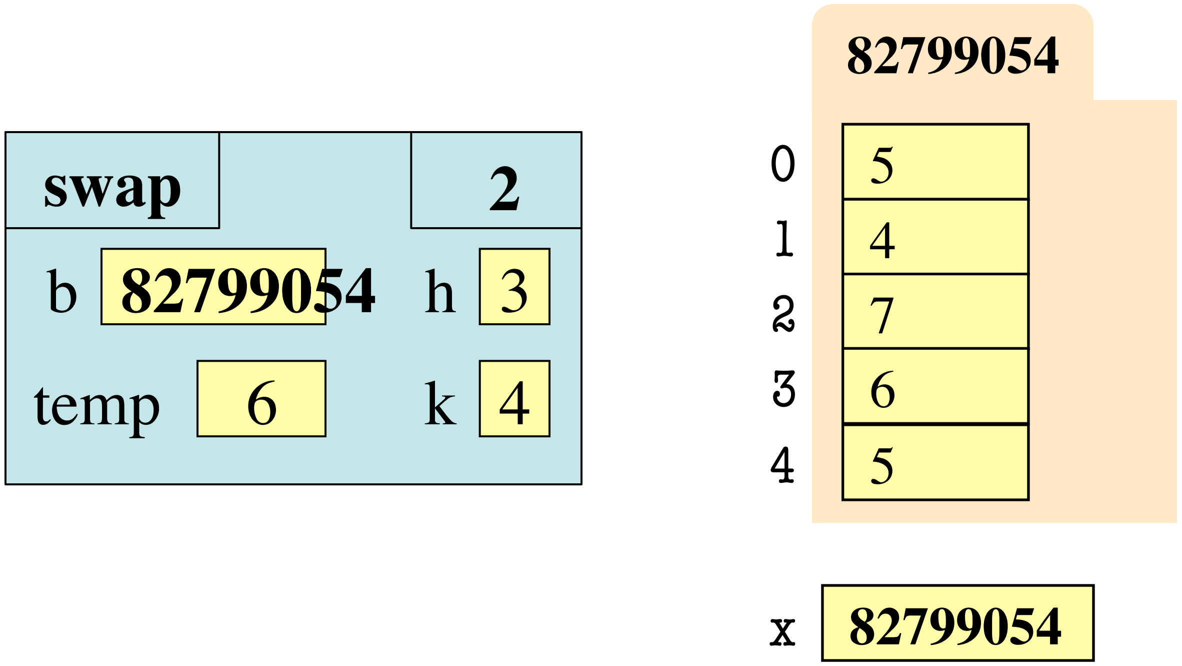


Lists and Functions: Swap

```
def swap(b, h, k):
    """Procedure swaps b[h] and b[k] in b
      Precondition: b is a mutable list, h
      and k are valid positions in the list"""
1   temp= b[h]
2   b[h]= b[k]
3   b[k]= temp
```

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

swap(x, 3, 4)

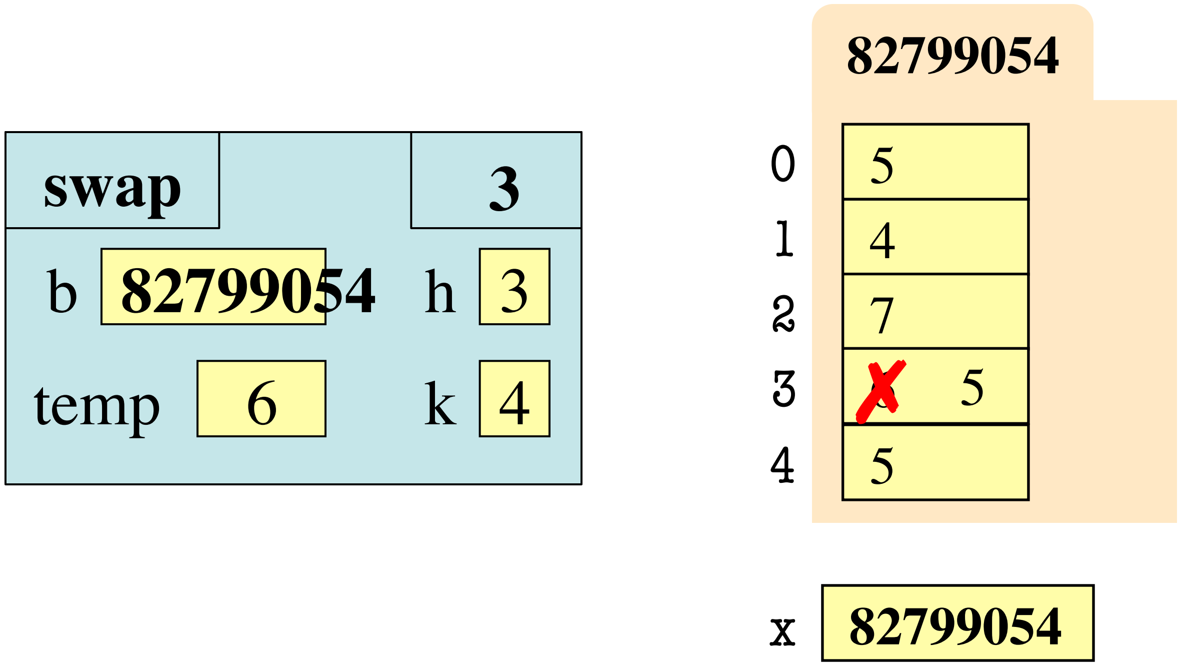


Lists and Functions: Swap

```
def swap(b, h, k):
    """Procedure swaps b[h] and b[k] in b
        Precondition: b is a mutable list, h
        and k are valid positions in the list""”
1   temp= b[h]
2   b[h]= b[k]
3   b[k]= temp
```

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

swap(x, 3, 4)

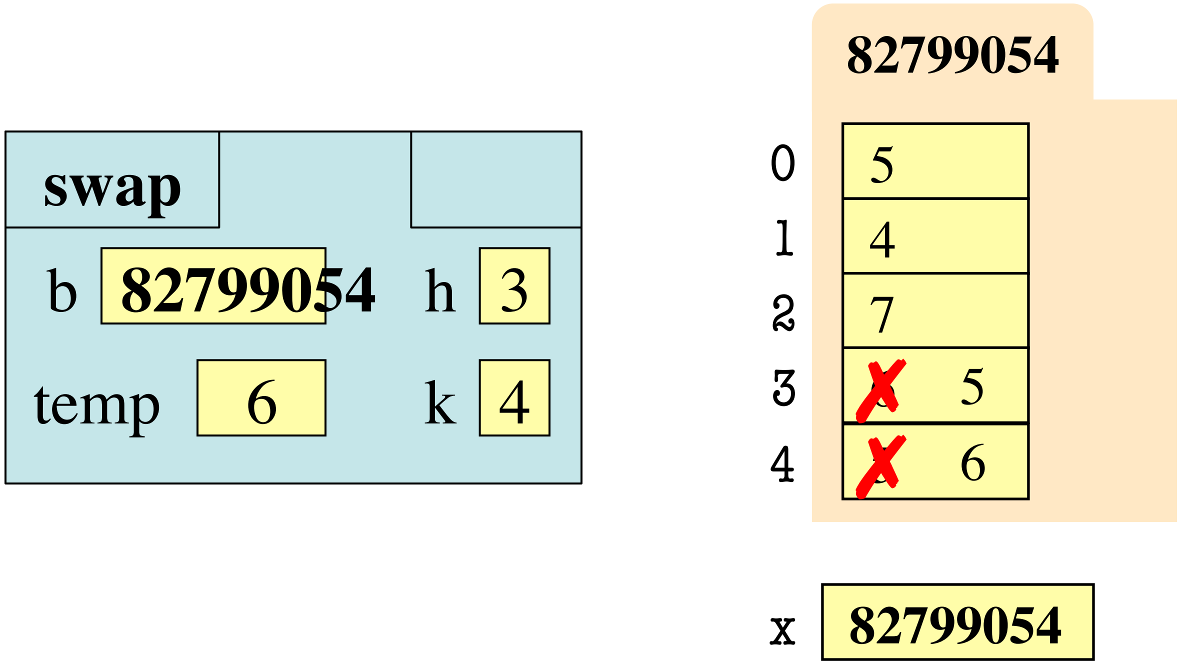


Lists and Functions: Swap

```
def swap(b, h, k):
    """Procedure swaps b[h] and b[k] in b
       Precondition: b is a mutable list, h
       and k are valid positions in the list"""
1   temp= b[h]
2   b[h]= b[k]
3   b[k]= temp
```

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

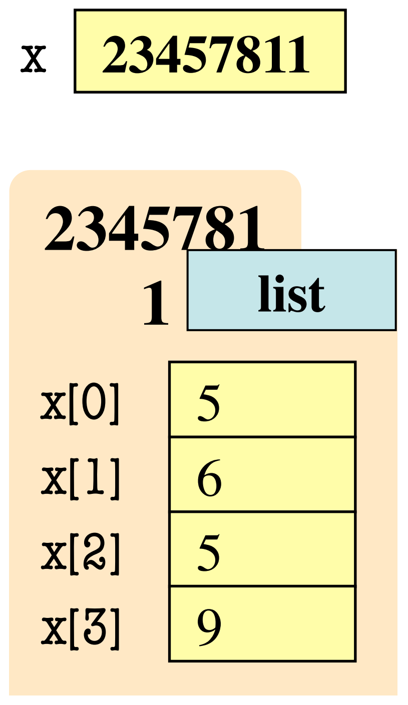
swap(x, 3, 4)

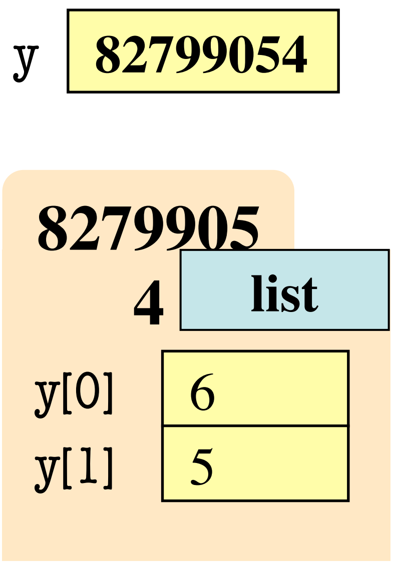


List Slices Make Copies

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

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





copy = new folder

Exercise Time

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
>>> x[3] = -1
>>> x.insert(1,2)
```

- What is x[4]?

A: 10
B: 9
C: -1
D: **ERROR**
E: I don't know

Exercise Time

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
>>> x[3] = -1
>>> x.insert(1,2)
```

- What is x[4]?

-1

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
>>> y = x[1:]
>>> y[0] = 7
```

- What is x[1]?

A: 7
B: 5
C: 6
D: **ERROR**
E: I don't know

Exercise Time

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
>>> x[3] = -1
>>> x.insert(1,2)
```

- What is x[4]?

-1

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
>>> y = [1:]
>>> y[0] = 7
```

- What is x[1]?

6