Lecture 7: Objects

(Chapter 15)

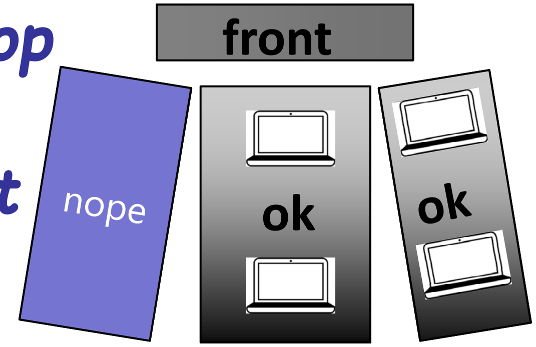
CS 1110

Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Fan, D. Gries, L. Lee,
S. Marschner, C. Van Loan, W. White]

Announcements

No-laptop zone on your left



- **Optional 1-on-1** with a staff member to help *just you* with course material. Sign up for a slot on CMS under “SPECIAL: one-on-ones“.
- A1: Two updates on course website—see **orange text** on cover page of A1 on website. We encourage you to use Piazza
- A **new** AEW section has been added: M 7:30-9:30pm (search for “ENGRG 1010” on Student Center for details)
- Before next lecture, read § 5.1-5.7

Programming Practice in A1

Testing
Debugging

Unit Test

Still need to import modules name, testcase

```
def test_last_name_first():
    """Calls all the tests for name.last_name_first"""
    print('Testing function name.last_name_first')
    # Test 1
    result = name.last_name_first('Katherine Johnson')
    testcase.assert_equals('Johnson, Katherine', result)
    # Test 2
    result = name.last_name_first('Katherine      Johnson')
    testcase.assert_equals('Johnson, Katherine', result)
```

Put all tests inside one function

```
# Execution of the testing code
test_last_name_first()
print('All tests of the function last_name_first passed')
```

No tests happen if you forget to call the function

How to debug

Do **not** ask:

"Why doesn't my code do what I want it to do?"

Instead, ask:

"What is my code doing?"

Two ways to inspect your code:

1. Step through your code, drawing pictures (or *use python tutor!*)
2. Use print statements

Take a look in the python tutor!

```
def last_name_first(full_name):
    <snip out comments for ppt slide>
    #get index of space
    space_index = full_name.find(' ')
    #get first name
    first = full_name[:space_index]
    #get last name
    last  = full_name[space_index+1:]
    #return "<last-name>, <first-name>"
    return last+', '+first

last_name_first("Katherine     Johnson")
```

Pay attention to:

- Code you weren't 100% sure of as you wrote it
- Code relevant to the failed test case

Using print statement to debug

```
def last_name_first(full_name):
    #get index of space
    space_index = full_name.find(' ')
    print('space_index = '+ str(space_index))
    #get first name
    first = full_name[:space_index]
    print('first = '+ first)
    #get last name
    last  = full_name[space_index+1:]
    print('last = '+ last)
    #return "<last-name>, <first-name>"
    return last+', '+first
```

Sometimes this is your only option, but it does make a mess of your code, and introduces cut-n-paste errors.

How do I print this?

Be sure to start A1 now

- **Start A1 now**
  - Lots of time to think through any difficulty parts
  - Consulting/office hours not too busy—get help fast
  - There's time to schedule a 1-on-1 appt

- Start A1 the night before due date
  - No time to deal with "sudden" difficulties
  - Consulting/office hours very crowded—loooonnnng wait time

Type: set of values & operations on them

Type **float:**

- Values: real numbers
- Ops: +, -, *, /, //, **

Type **int:**

- Values: integers
- Ops: +, -, *, //, %, **

Type **bool:**

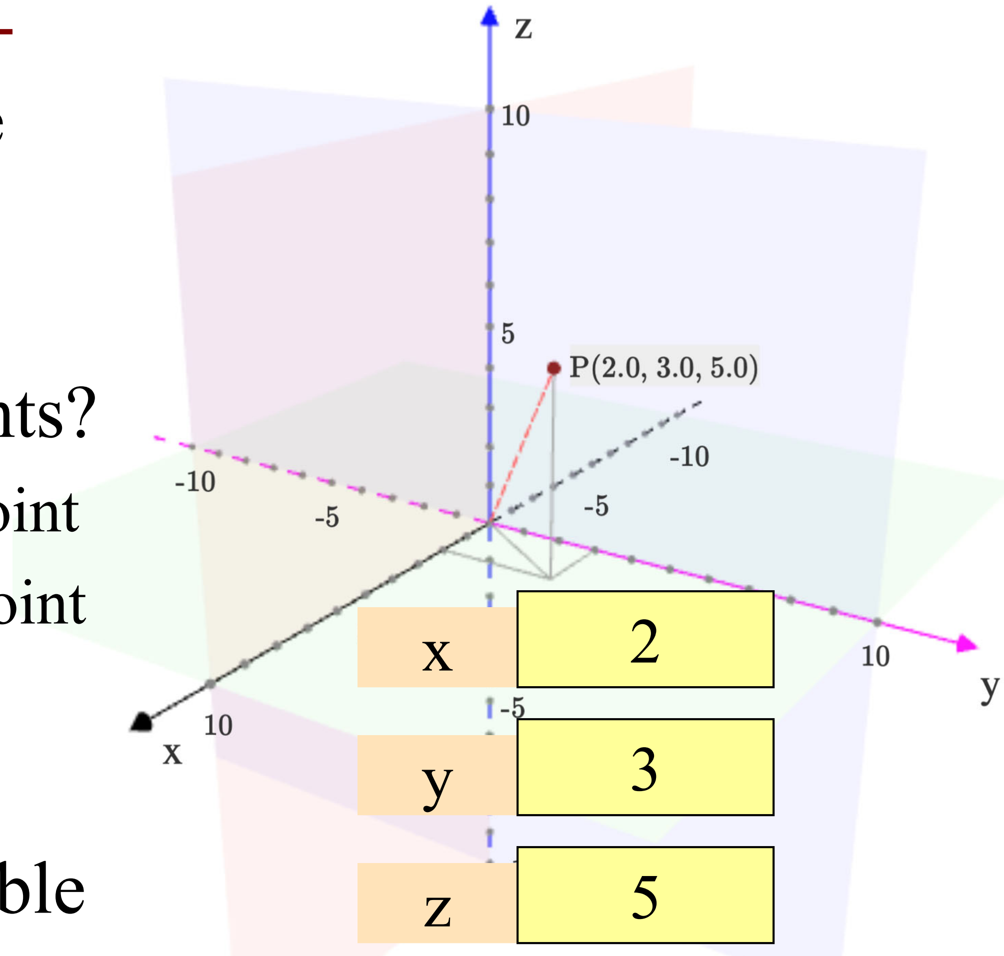
- Values: integers
- Ops: not, and, or

Type **str:**

- Values: string literals
  - Double quotes: “abc”
  - Single quotes: ‘abc’
- Ops: + (concatenation)

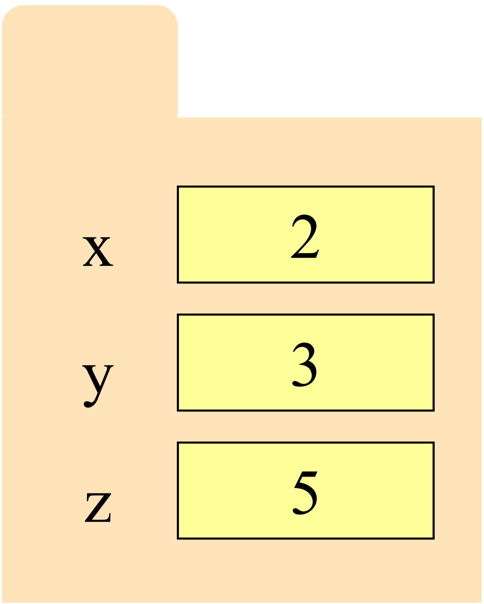
Built-in Types are not “Enough”

- 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?



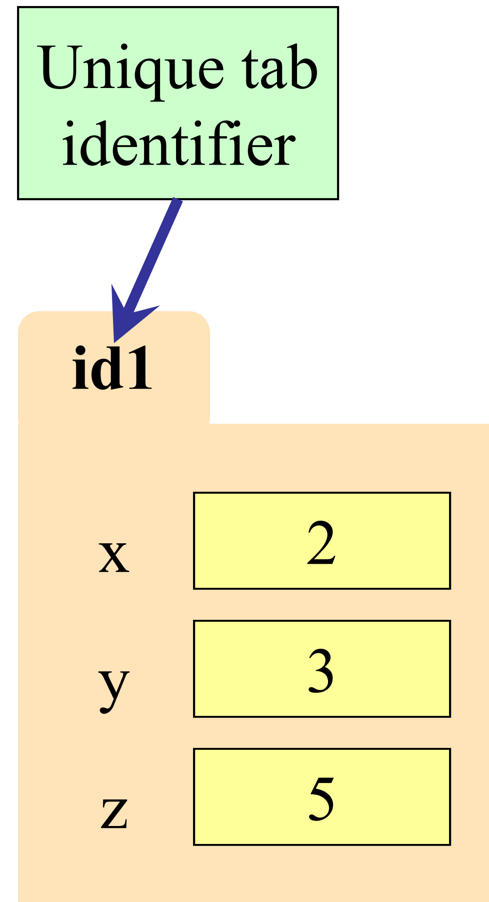
Built-in Types are not "Enough"

- 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**



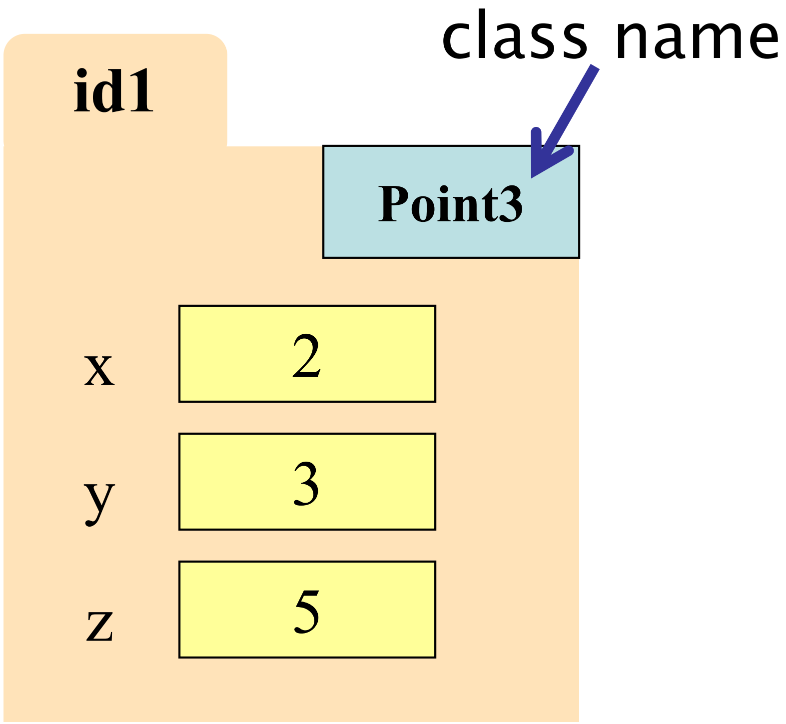
Objects: Organizing Data in Folders

- 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



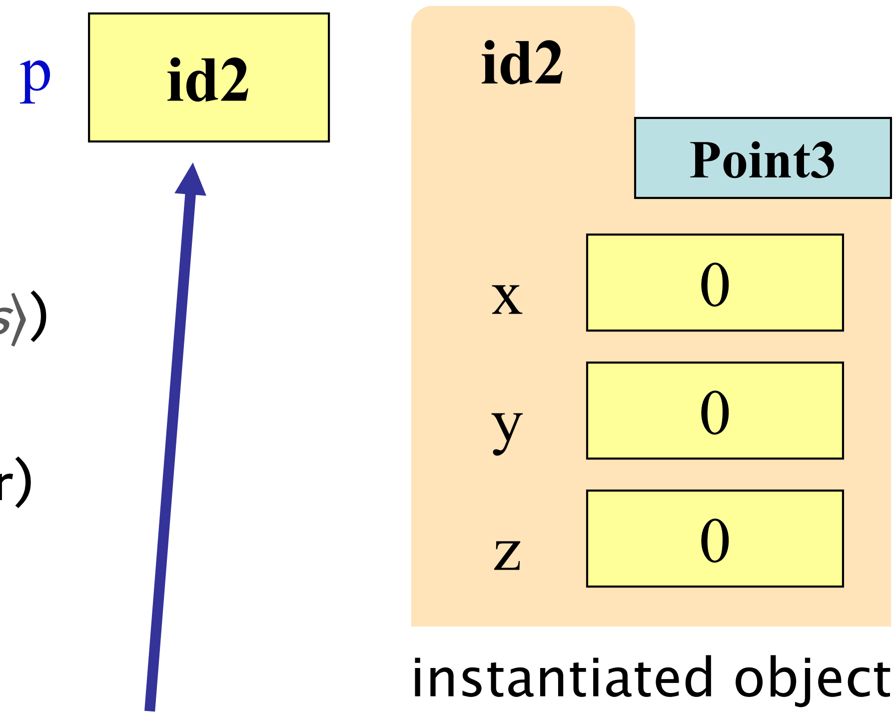
Classes: user-defined types for Objects

- Values must have a type
  - An object is a **value**
  - Object type is a **class**
- **Modules** provide classes
- **Example**: shapes.py
  - Defines: Point3, Rectangle classes



Constructor: Function to make Objects

- How do we create objects?
  - Other types have literals
  - No such thing for objects
- **Call a Constructor Function:**
  - **Format:** ⟨*class name*⟩(⟨*arguments*⟩)
  - **Example**: Point3(0,0,0)
  - Makes a new object (manila folder) with a ***new id***
  - Called an *instantiated* object
  - Returns folder ***id*** as value
- **Example:** p = Point3(0, 0, 0)
  - Creates a Point object
  - Stores object's ***id*** in p

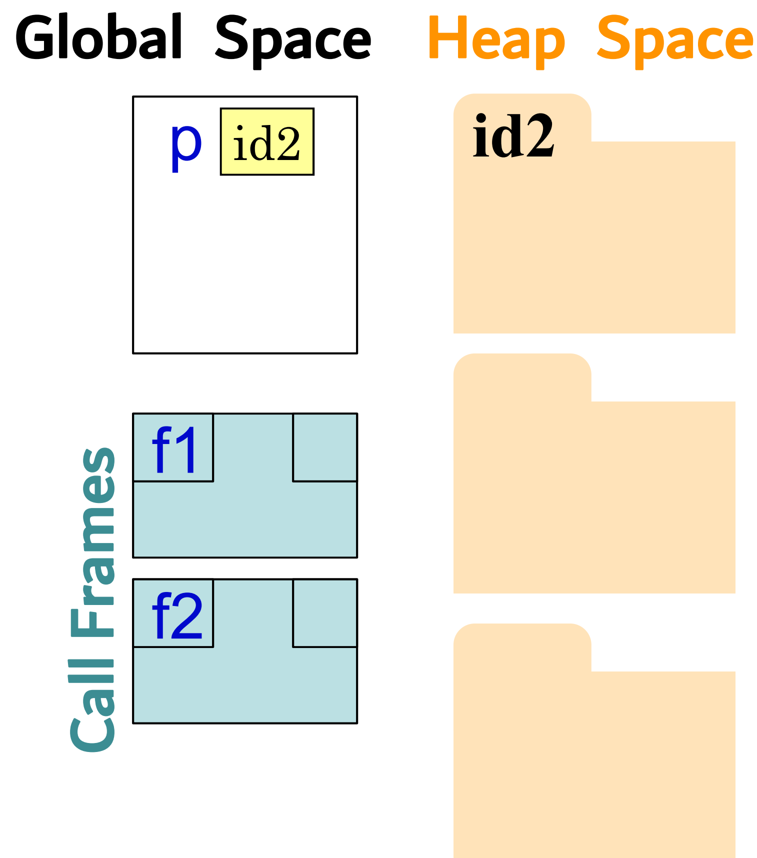


variable stores id not object

instantiated object

Storage in Python

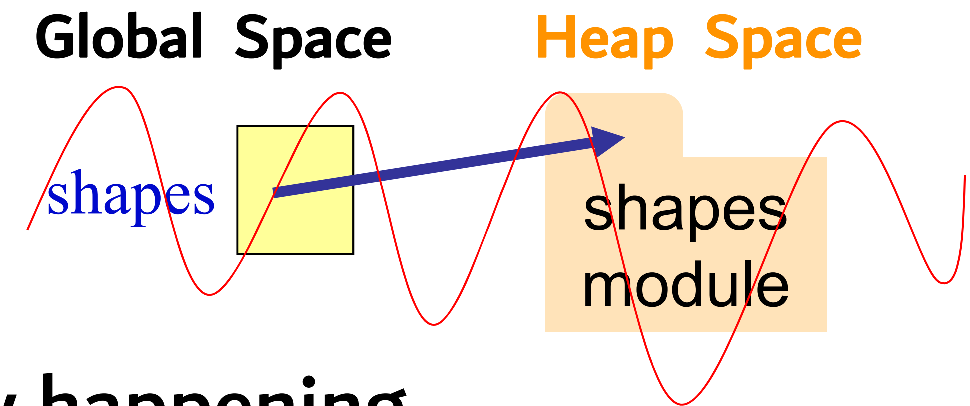
- **Global Space**
  - What you "start with"
  - Stores global variables
  - Lasts until you quit Python
- **Heap Space**
  - Where "folders" are stored
  - Have to access indirectly
- **Call Frames**
  - Parameters
  - Other variables local to function
  - Lasts until function returns



Constructors and Modules

>>> import shapes

Need to import module that has Point3 class.



- This is what's actually happening
- Python Tutor draws this.
- Knowing this will help you debug.

CS 1110 doesn't draw module variables & folders (also skips all the built-in functions)

→ makes your diagrams cleaner

Constructors and Modules

```
>>> import shapes
```

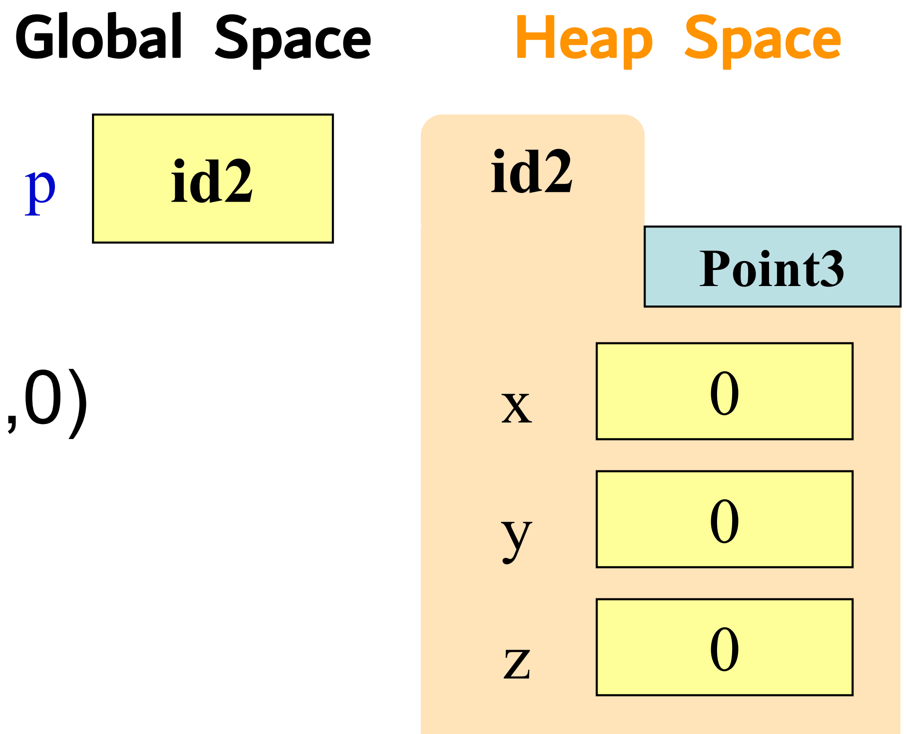
Need to import module that has Point3 class.

```
>>> p = shapes.Point3(0,0,0)
```

Constructor is function. Prefix w/ module name.

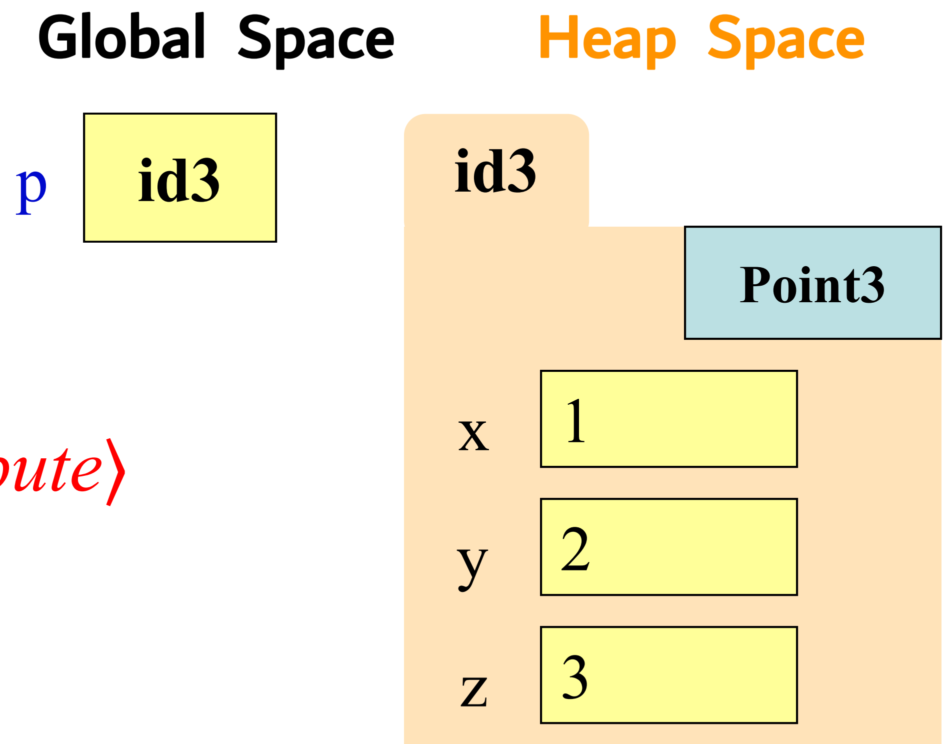
```
>>> id(p)
```

Shows the ***id*** of p



Accessing Attributes

- Attributes are variables that live inside of objects
  - Can **use** in expressions
  - Can **assign** values to them
- **Format**: ⟨*variable*⟩.⟨*attribute*⟩
  - **Example**: p.x
  - Look like module variables
- To evaluate p.x, Python:
  1. finds folder with ***id*** stored in p
  2. returns the value of x in that folder

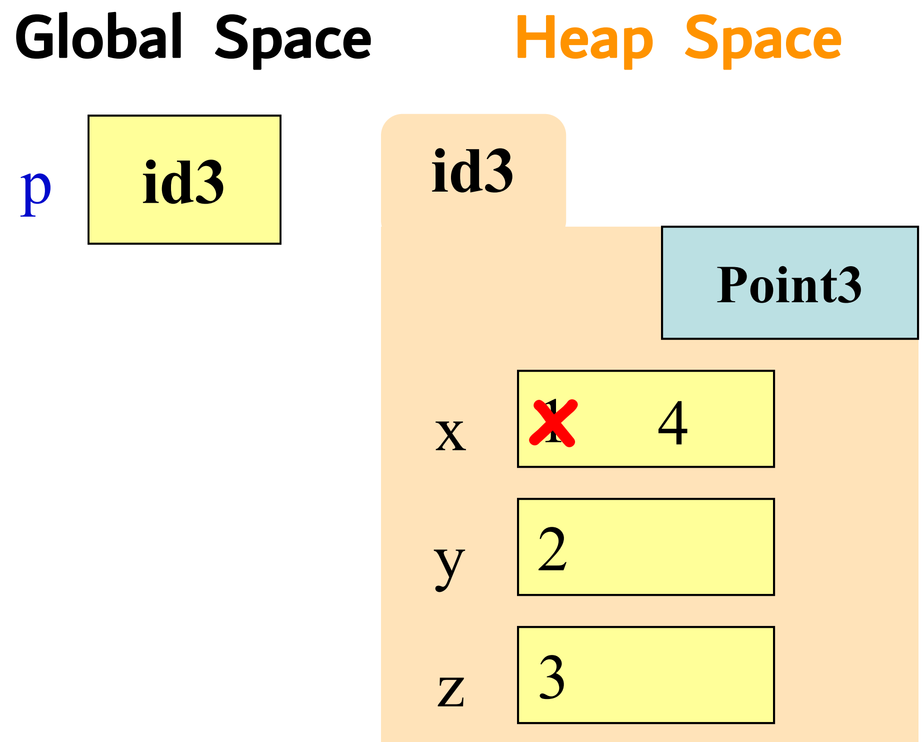


Accessing Attributes Example

- **Example**:

  p = shapes.Point3(1, 2, 3)

  p.x = p.x + 3

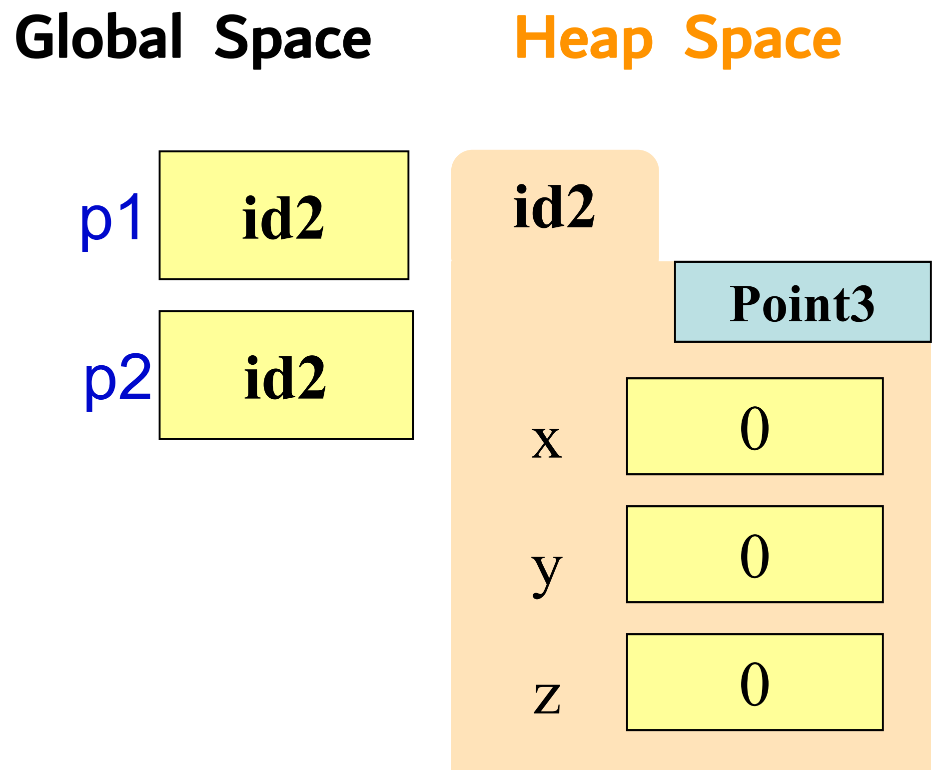


Object Variables

- Variable stores object ***id***
  - **Reference** to the object
  - Reason for folder analogy
- Assignment uses object ***id***
  - **Example**:

  p1 = shapes.Point3(0, 0, 0)

  p2 = p1
  - Takes contents from p1
  - Puts contents in p2
  - Does not make new folder!



This is the cause of many mistakes when starting to use objects

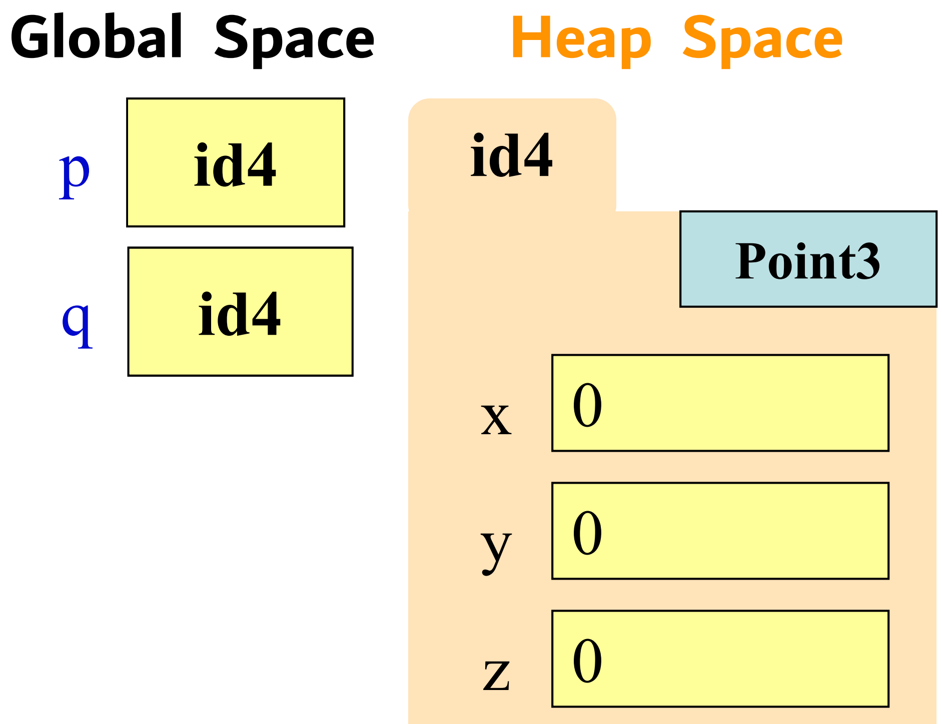
Attribute Assignment (Question)

```
>>> p = shapes.Point3(0,0,0)
>>> q = p
```

- Execute the assignments:

  ```
  >>> p.x = 5
  >>> q.x = 7
  ```

- What is value of p.x?

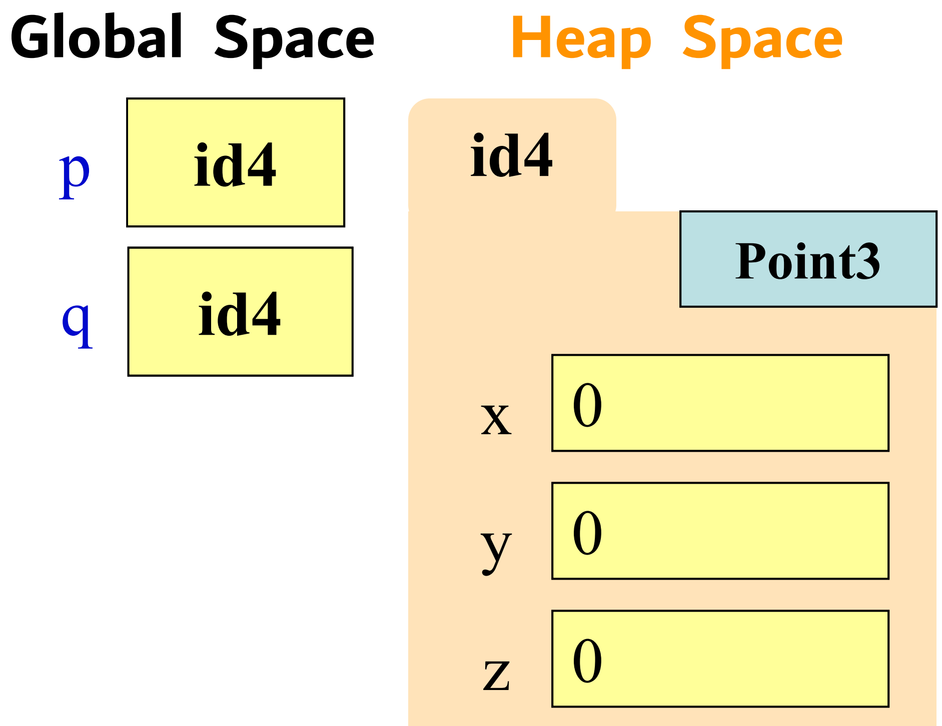


A: 5
B: 7
C: **id4**
D: I don't know

Attribute Assignment (Solution)

```
>>> p = shapes.Point3(0,0,0)
>>> q = p
```

- Execute the assignments:
  ```
  >>> p.x = 5
  >>> q.x = 7
  ```
- What is value of p.x?



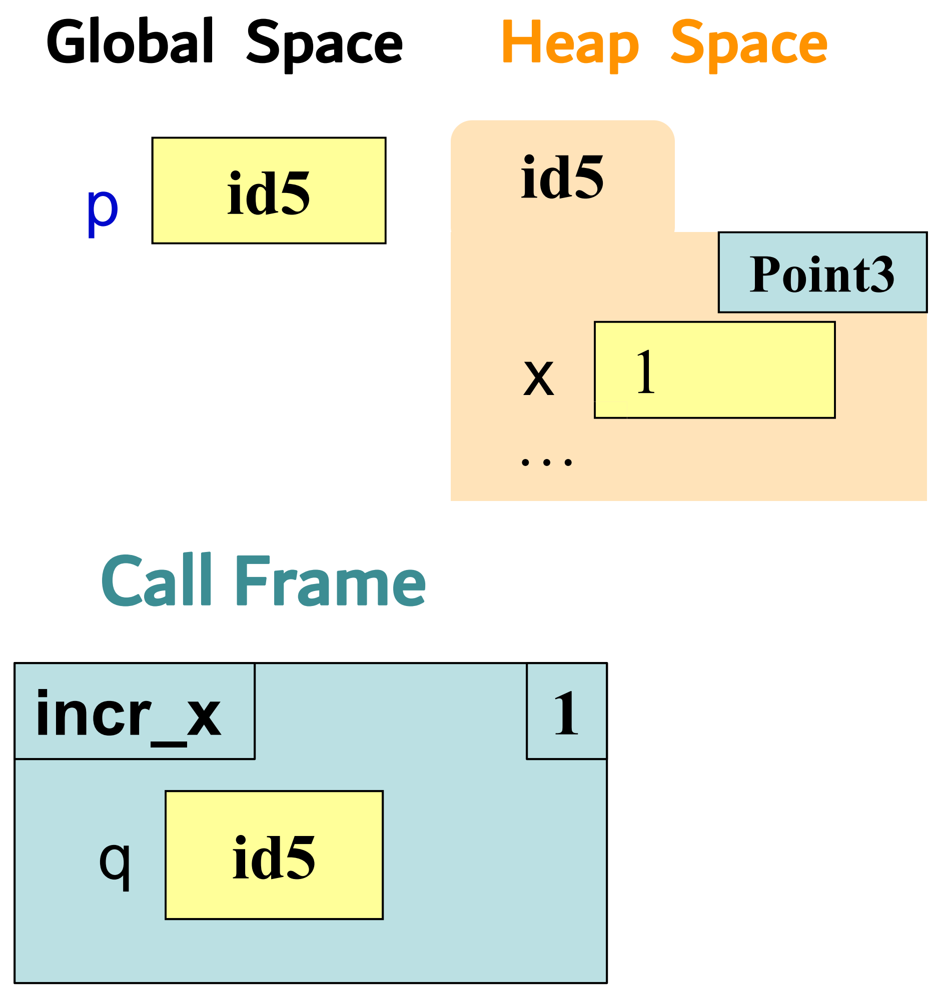
A: 5
B: 7 **CORRECT**
C: **id4**
D: I don't know

Call Frames and Objects (1)

- Objects can be altered in a function call
  - Object variables hold ***id***s!
  - Folder can be accessed from global variable or parameter
- **Example**:

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

```
>>> p = shapes.Point3(1, 2, 3)
>>> incr_x(p)
```

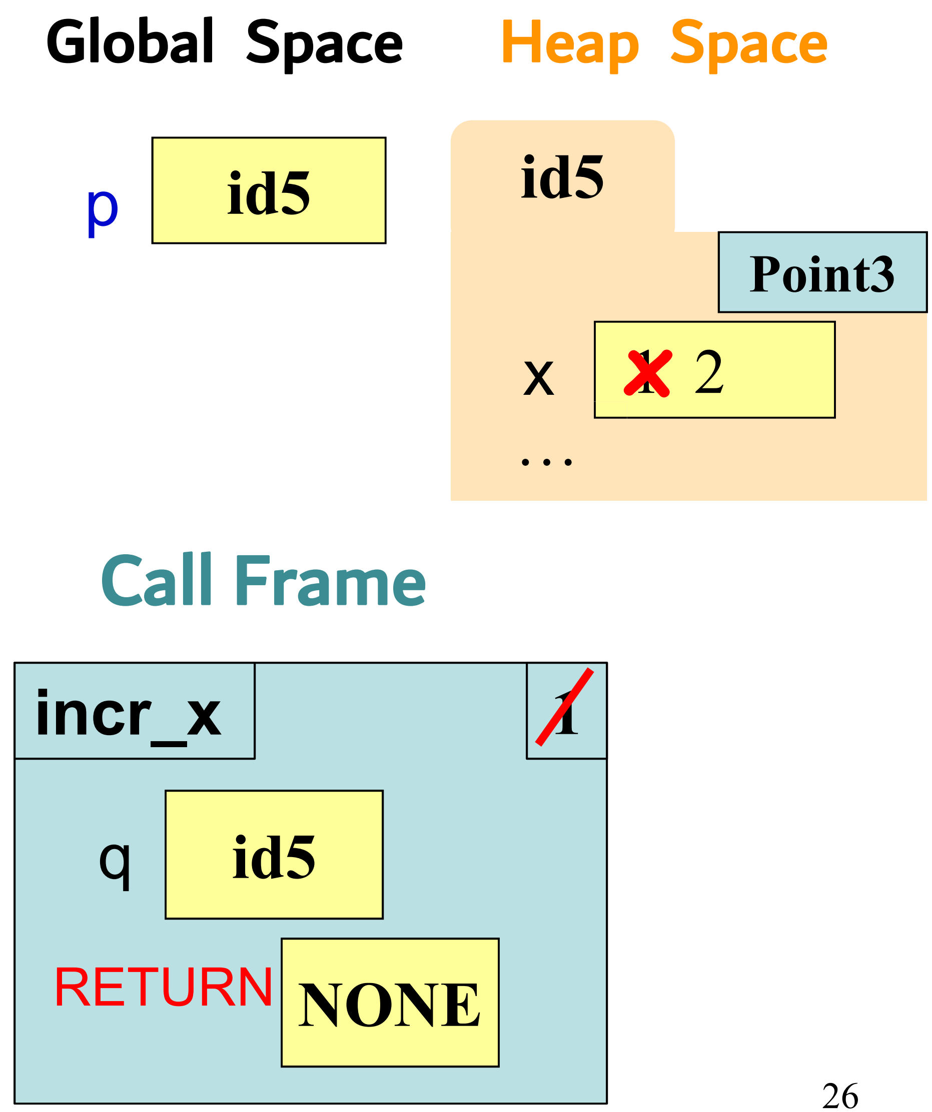


Call Frames and Objects (2)

- Objects can be altered in a function call
  - Object variables hold ***id***s!
  - Folder can be accessed from global variable or parameter
- **Example**:

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

```
>>> p = shapes.Point3(1, 2, 3)
>>> incr_x(p)
```

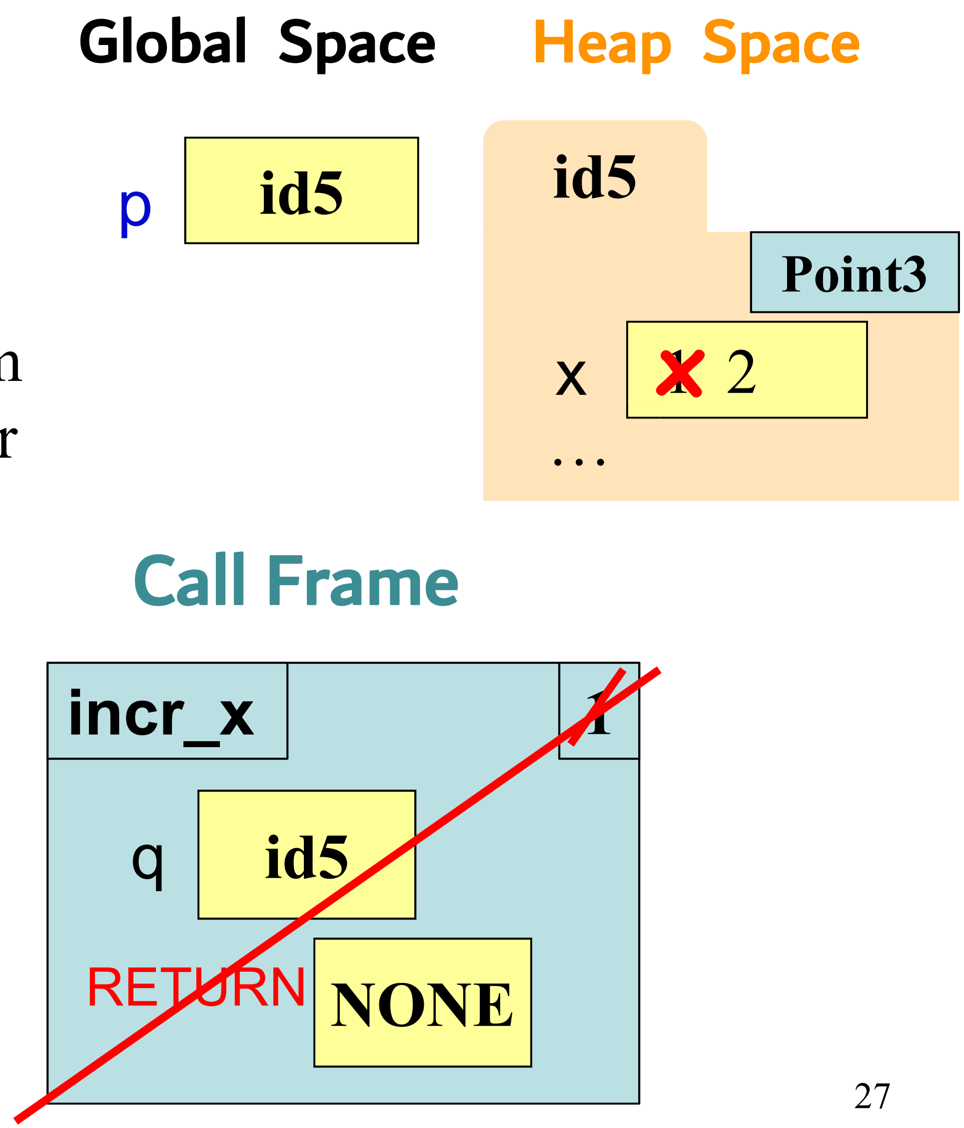


Call Frames and Objects (3)

- Objects can be altered in a function call
  - Object variables hold ***id***s!
  - Folder can be accessed from global variable or parameter
- **Example**:

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

```
>>> p = shapes.Point3(1, 2, 3)
>>> incr_x(p)
```



How Many Folders (Question)

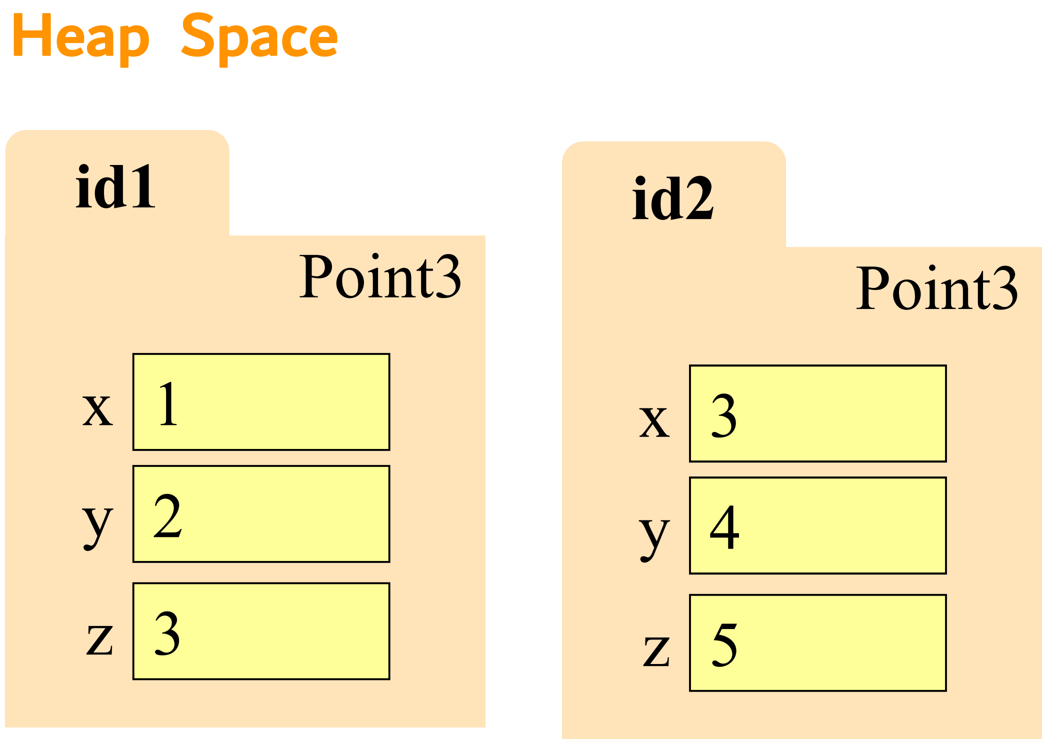
```
import shapes
p = shapes.Point3(1,2,3)
q = shapes.Point3(3,4,5)
```

Draw everything that gets created.
How many folders get drawn?

How Many Folders (Solution)

```
import shapes
p = shapes.Point3(1,2,3)
q = shapes.Point3(3,4,5)
```

Draw everything that gets created.
How many folders get drawn?

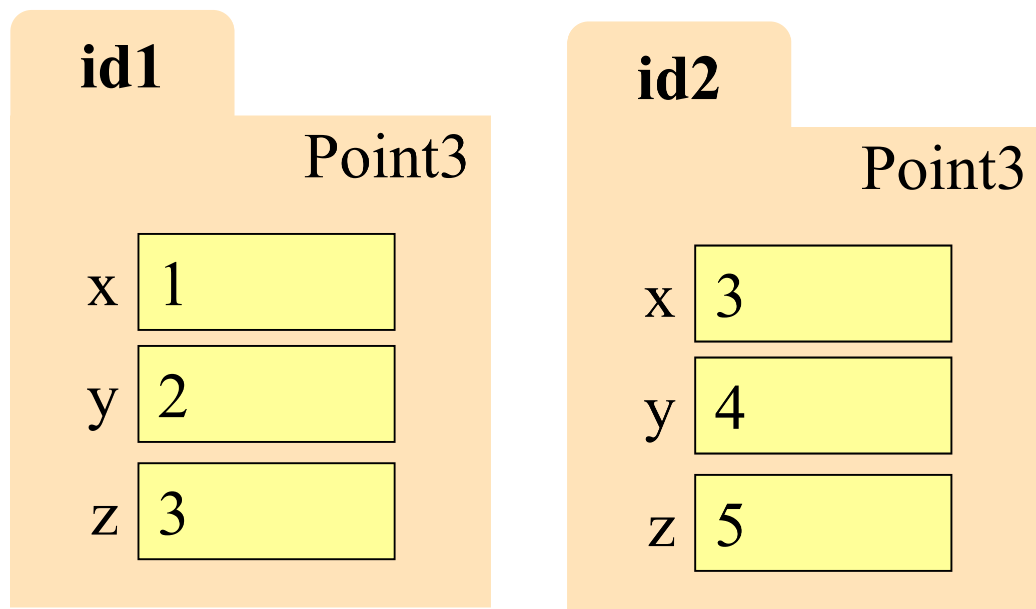


What Else? (Question)

```
import shapes
p = shapes.Point3(1,2,3)
q = shapes.Point3(3,4,5)
```

Draw everything that gets created.
How many folders get drawn?
What else gets drawn?

Heap Space

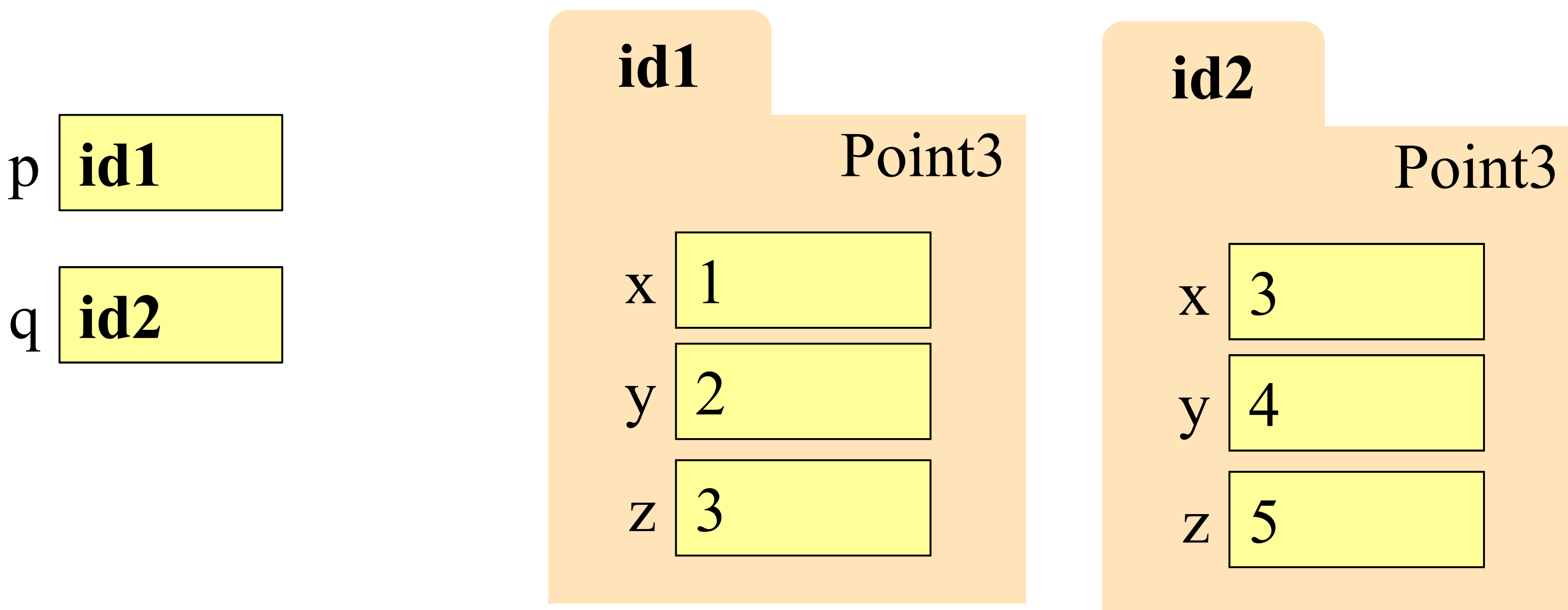


What Else? (Solution)

```
import shapes
p = shapes.Point3(1,2,3)
q = shapes.Point3(3,4,5)
```

Draw everything that gets created.
How many folders get drawn?
What else gets drawn?

Global Space **Heap Space**



Swap (Question)

```
import shapes
p = shapes.Point3(1,2,3)
q = shapes.Point3(3,4,5)

def swap_x(p, q):
    t = p.x
    p.x = q.x
    q.x = t

swap_x(p, q)
```

What is in p.x at the end of this code?

A: 1
B: 2
C: 3
D: I don't know

Global Space

p	**id1**
q	**id2**

Heap Space

id1 Point3

x	1
y	2
z	3

id2 Point3

x	3
y	4
z	5

Global p (Question)

```
import shapes
p = shapes.Point3(1,2,3)
q = shapes.Point3(3,4,5)

def swap(p, q):
    t = p
    p = q
    q = t

swap(p, q)
```

What is in global p after calling swap?

A: **id1**
B: **id2**
C: I don't know

Global Space

p	**id1**
q	**id2**

Heap Space

id1 Point3

x	1
y	2
z	3

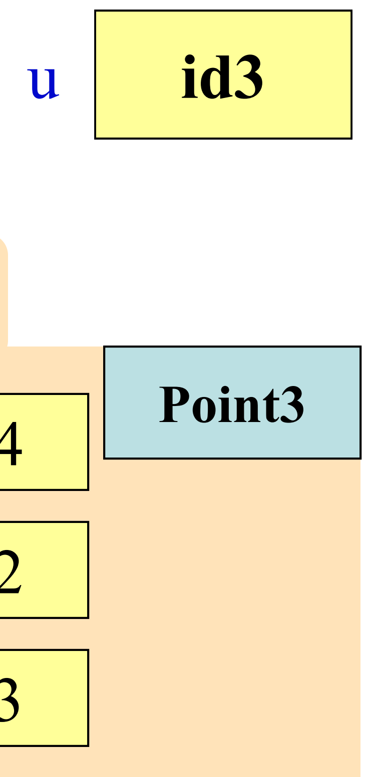
id2 Point3

x	3
y	4
z	5

Methods: Functions Tied to Classes

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

    ⟨*variable*⟩.⟨*method*⟩(⟨*arguments*⟩)



Example:

```
import shapes
u = shapes.Point3(4,2,3)
u.greet()
```

"Hi! I am a 3-dimensional point located at (4,2,3)"

Where else have you seen this??

Example: String Methods

- s_1.upper()
  - Returns returns an upper case version of s_1
- s.strip()
  - Returns a copy of s with white-space removed at ends
- s_1.index(s_2)
  - Returns position of the first instance of s_2 in s_1
  - error if s_2 is not in s_1
- s_1.count(s_2)
  - Returns number of times s_2 appears inside of s_1

Built-in Types vs. Classes

Built-in types	Classes
Built-into Python	Provided by modules
Refer to instances as *values*	Refer to instances as *objects*
Instantiate with *literals*	Instantiate w/ *constructors*
Can ignore the folders	Must represent with folders

Where To From Here?

- First, Understand **objects**
  - All Python programs use objects
  - Most small programs use objects of classes that are part of the Python Library
- Eventually, create your own **classes:**
  - the heart of OO Programming
  - the primary tool for organizing Python programs

- But we need to learn more basics first!