

Lecture 9

Objects

Announcements for Today

Assignment 1

- Grades posted at **5 pm**
- Resubmit until correct
 - Read feedback in CMS
 - Reupload/request regrade
- If you were very **wrong**...
 - Will receive an e-mail
 - More 1-on-1s this week
- Finish **Survey 1**

Assignment 2

- Posted last **Friday**
 - Written assignment
 - Do while revising A1
 - Relatively short (2-3 hrs)
- Due **Sunday**
 - Submit as a PDF
 - Scan or phone picture
 - **US Letter format!**

The Basic Python Types

- 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?

Example: Points in 3D Space

```
def distance(x0,y0,z0,x1,y1,z1):
```

```
    """Returns distance between points (x0,y0,y1) and (x1,y1,z1)
```

```
    Param x0: x-coord of 1st point
```

```
    Precond: x0 is a float
```

```
    Param y0: y-coord of 1st point
```

```
    Precond: y0 is a float
```

```
    Param z0: z-coord of 1st point
```

```
    Precond: z0 is a float
```

```
    ....
```

```
    """
```

- This is very unwieldy
 - Specification is too long
 - Calls needs many params
 - Typo bugs are very likely
- Want to reduce params
 - Package points together
 - How can we do this?

Points as Their Own Type

```
def distance(p0,p1):
```

```
    """Returns distance between points p0 and p1
```

```
    Param p0: The second point
```

```
    Precond: p0 is a Point3
```

```
    Param p1: The second point
```

```
    Precond: p1 is a Point3"""
```

```
    ...
```

This lecture will help you
make sense of this spec.

Classes: Custom Types

- **Class**: Custom type **not built into** Python
 - Just like with functions: built-in & defined
 - Types not built-in are **provided by modules**
- Might seem weird: `type(1) ==> <class 'int'>`
 - In Python 3 type and class are **synonyms**
 - We will use the historical term for clarity

introc provides several classes

Objects: Values for a Class

- **Object**: A specific **value** for a class type
 - Remember, a type is a set of values
 - Class could have infinitely many objects
- **Example**: Class is Point3
 - One object is **origin**; another **x-axis** (1,0,0)
 - These objects go in params distance function
- Sometimes refer to objects as **instances**
 - Because a value is an instance of a class
 - Creating an object is called *instantiation*

How to Instantiate an Object?

- Other types have **literals**
 - **Example:** 1, 'abc', True
 - No such thing for objects
- Classes are provided by modules
 - Modules typically provide new functions
 - In this case, gives a function to make objects
- **Constructor function** has same name as class
 - Similar to types and type conversion
 - **Example:** **str** is a type, str(1) is a function call

Demonstrating Object Instantiation

```
>>> from introcs import Point3 # Module with class
>>> p = Point3(0,0,0)           # Create point at origin
>>> p                           # Look at this new point
<class 'introcs.geom.point.Point3'>(0.0,0.0,0.0)
>>> type(p) == Point3          # Check the type
True
>>> q = Point3(1,2,3)          # Make new point
>>> q                           # Look at this new point
<class 'introcs.geom.point.Point3'>(1.0,2.0,3.0)
```

What Does an Object Look Like?

- Objects can be a bit strange to understand
 - Don't look as simple as strings or numbers
 - **Example:** <class 'intros.Point3'>(0.0,0.0,0.0)
- To understand objects, need to *visualize* them
 - Use of metaphors to help us think like Python
 - Call frames (assume seen) are an example
- To visualize we rely on the **Python Tutor**
 - Website linked to from the course page
 - But use only that one! Other tutors are different.

Metaphor: Objects are Folders

```
>>> import introcs
```

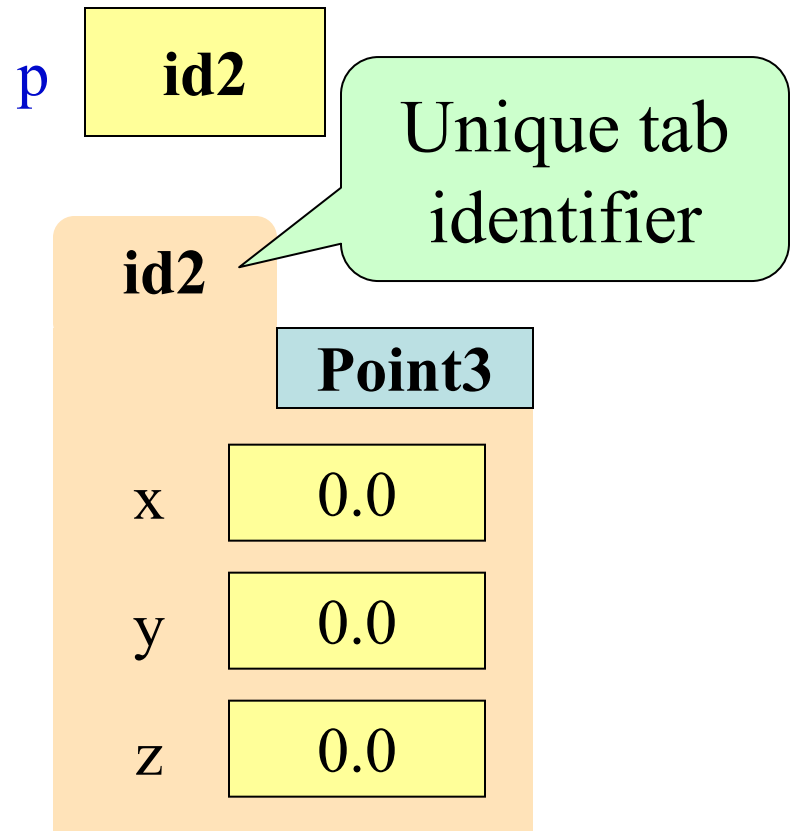
Need to import module
that has Point class.

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

Constructor is function.
Prefix w/ module name.

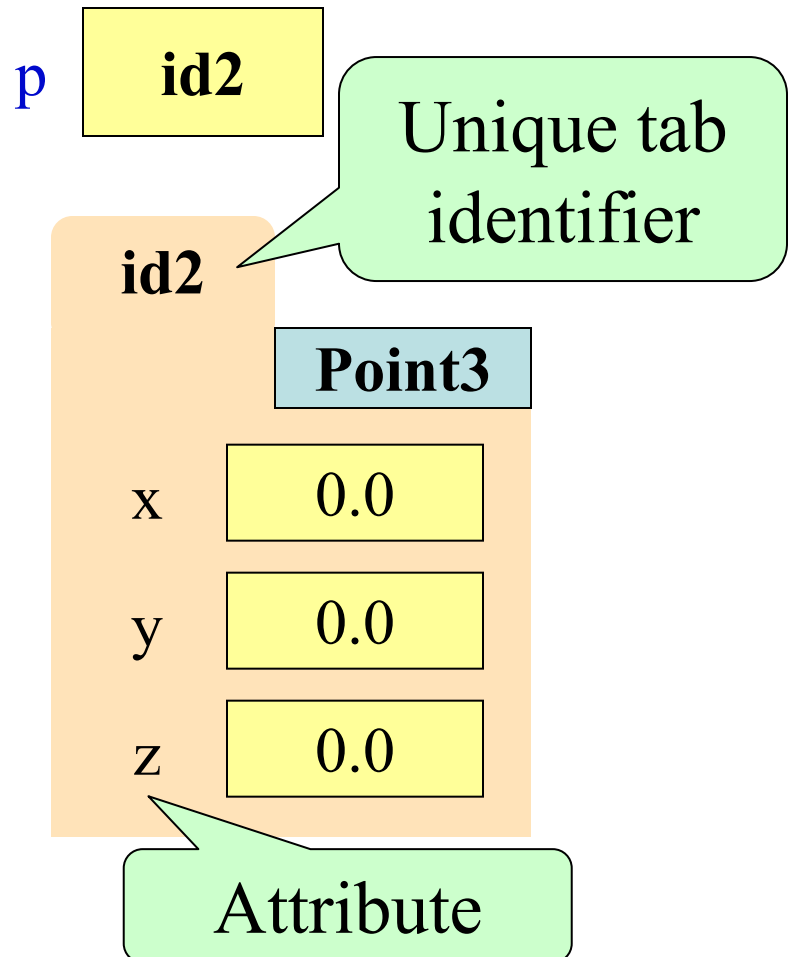
```
>>> id(p)
```

Shows the ID of p.



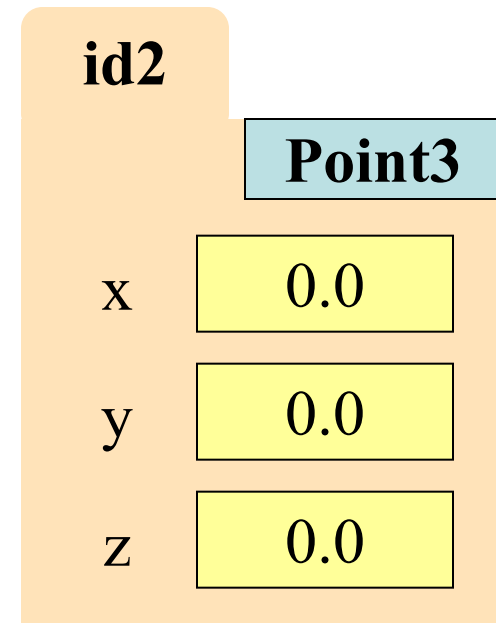
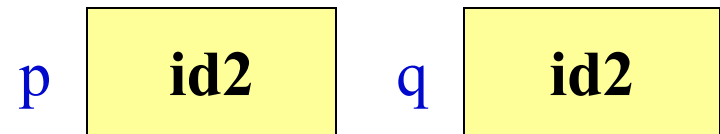
Metaphor: Objects are Folders

- **Idea:** Data too “big” for p
 - Split into many variables
 - Put the variables in folder
 - They are called **attributes**
- Folder has an identifier
 - Unique (like a netid)
 - Cannot ever change
 - Has no real meaning; only identifies folder



Object Variables

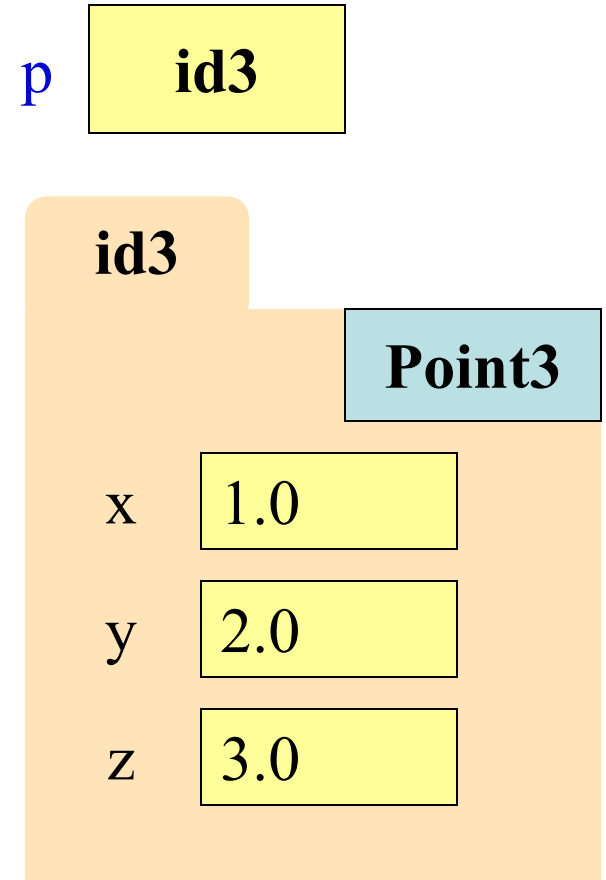
- Variable stores object name
 - **Reference** to the object
 - Reason for folder analogy
- Assignment uses object name
 - **Example:** $q = p$
 - Takes name from p
 - Puts the name in q
 - Does not make new folder!
- This is the cause of many mistakes for beginners



Objects and Attributes

- Attributes live inside objects
 - Can access these attributes
 - Can use them in expressions
- **Access:** `<variable>.<attr>`
 - Look like module variables
 - **Recall:** `math.pi`
- **Example**

```
>>> p = introcs.Point3(1,2,3)  
>>> a = p.x + p.y
```



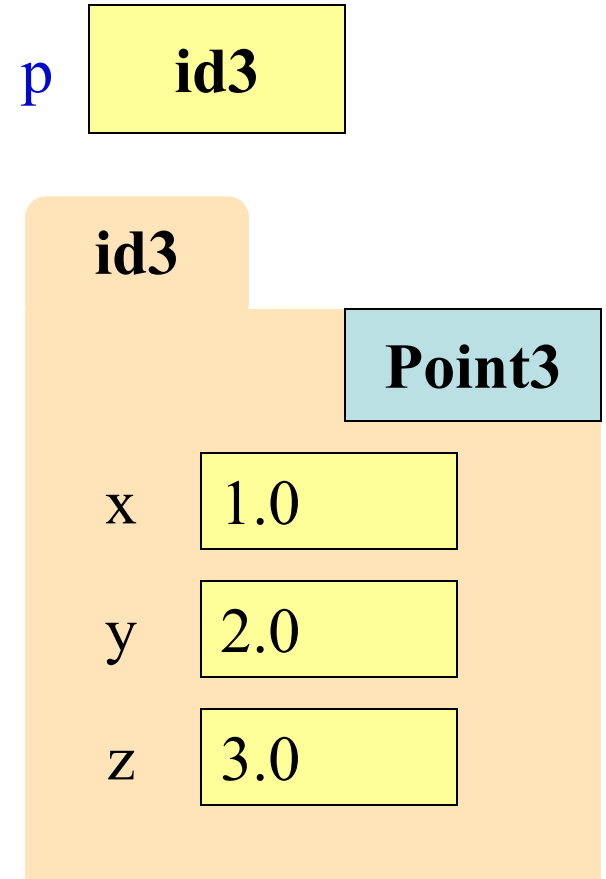
Objects and Attributes

- Attributes live inside objects
 - Can access these attributes
 - Can use them in expressions
- **Access:** `<variable>.<attr>`
 - Look like module variables
 - **Recall:** `math.pi`
- **Example**

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

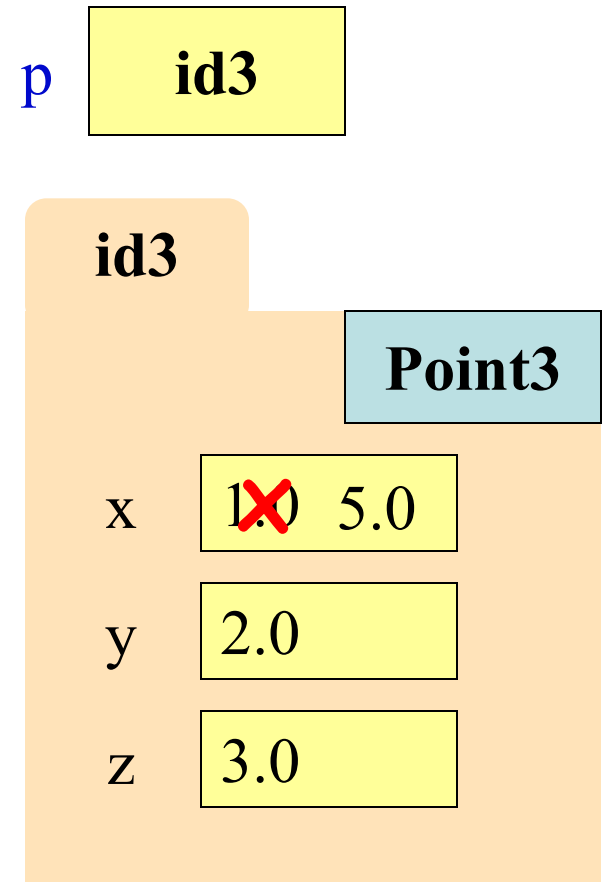
```
>>> a = p.x + p.y
```

`a` 3.0



Objects and Attributes

- Can also **assign** attributes
 - Reach into folder & change
 - Do without changing p
- $\langle \text{var} \rangle . \langle \text{attr} \rangle = \langle \text{exp} \rangle$
 - **Example:** $p.x = p.y + p.z$
 - See this in visualizer
- This is very powerful
 - Another reason for objects
 - Why need visualization



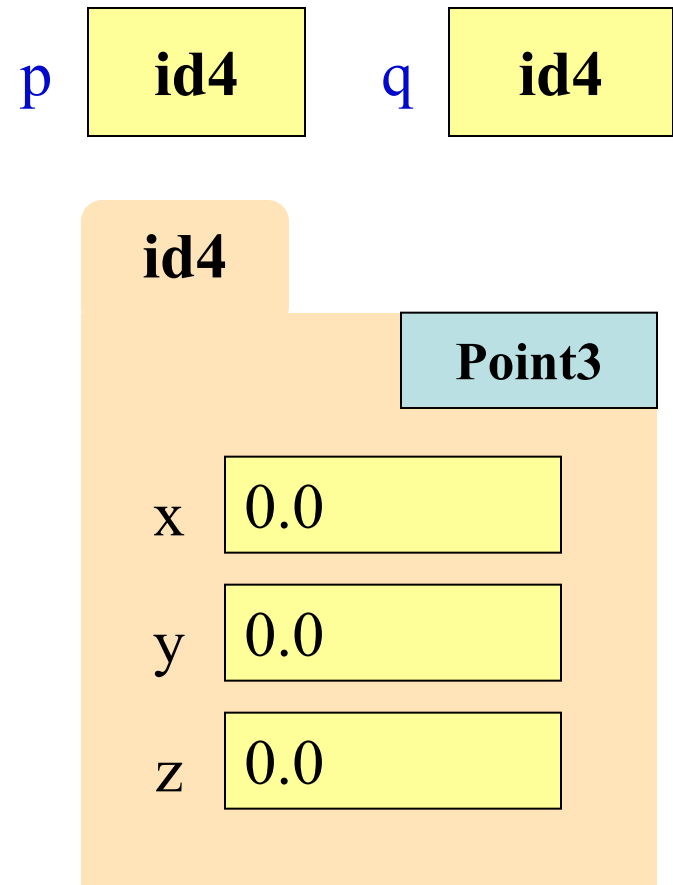
Exercise: Attribute Assignment

- Recall, q gets name in p

```
>>> p = introcs.Point3(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



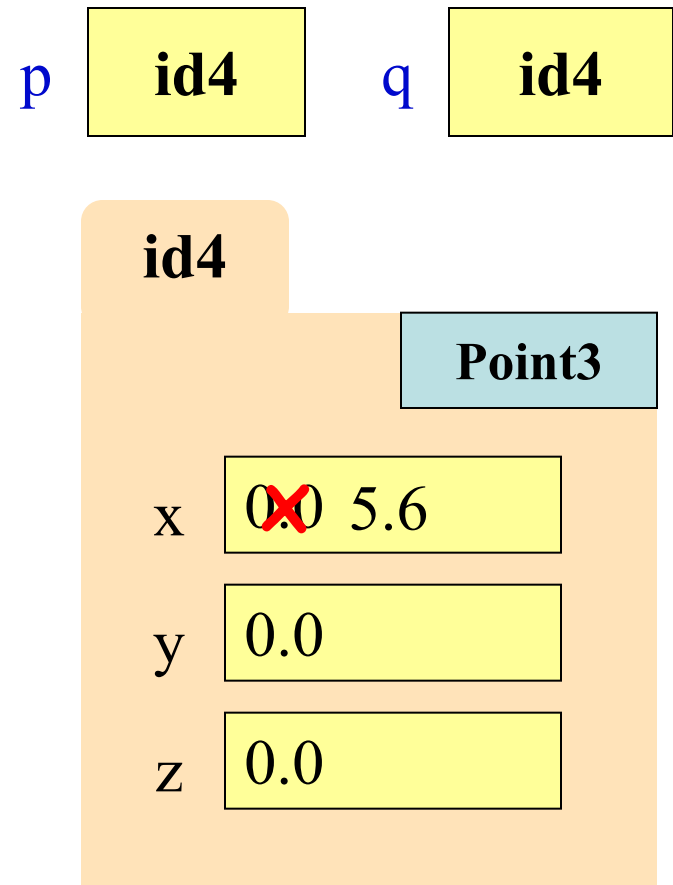
Exercise: Attribute Assignment

- Recall, q gets name in p

```
>>> p = introcs.Point3(0,0,0)
>>> q = p
```
- Execute the assignments:

```
>>> p.x = 5.6
>>> q.x = 7.4
```
- What is value of p.x?

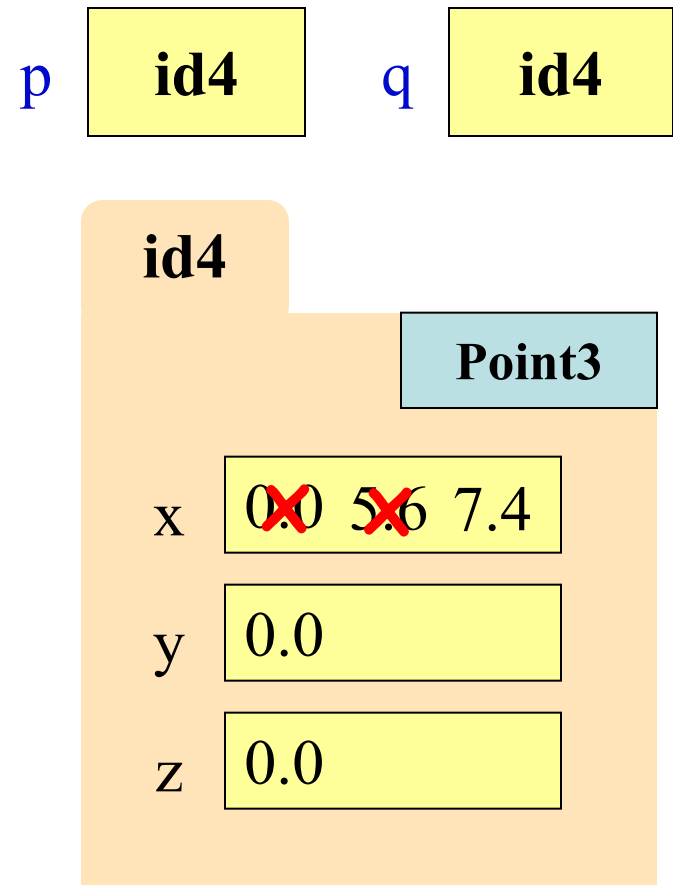
A: 5.6
B: 7.4 **CORRECT**
C: id4
D: I don't know



Exercise: Attribute Assignment

- Recall, q gets name in p
 - >>> p = introcs.Point3(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



Objects Allow for **Mutable** Functions

- **Mutable function**: *alters* the parameters
 - Often a procedure; no return value
- Until now, this was impossible
 - Function calls **COPY** values into new variables
 - New variables erased with call frame
 - Original (global?) variable was unaffected
- But object variables are *folder names*
 - Call frame refers to same folder as original
 - Function may modify the contents of this folder

Example: Mutable Function Call

- **Example:**

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

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

```
>>> p.x
```

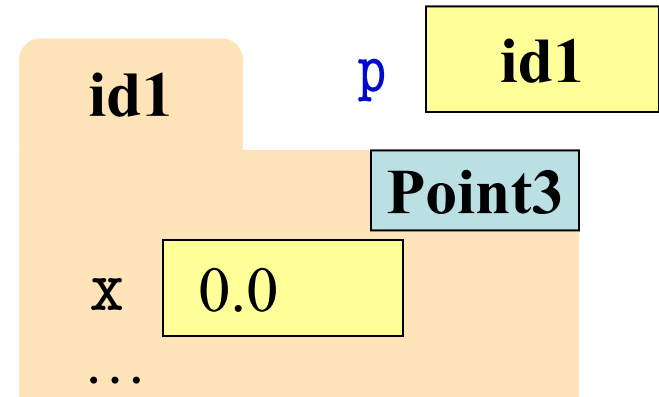
```
0.0
```

```
>>> incr_x(p)
```

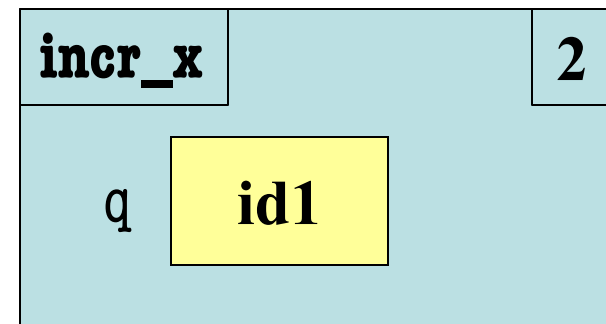
```
>>> p.x
```

```
1.0
```

Global **STUFF**



Call Frame



Example: Mutable Function Call

- **Example:**

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

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

```
>>> p.x
```

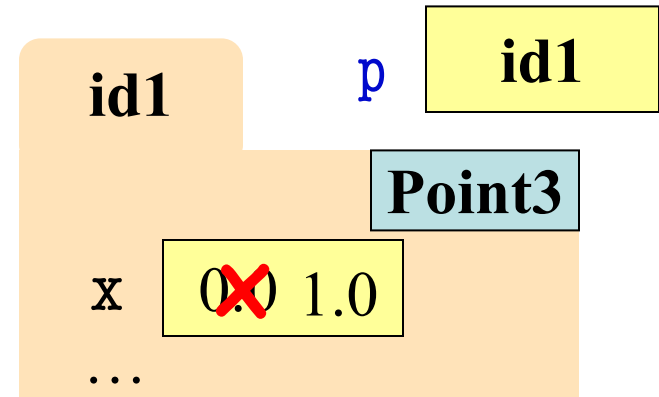
```
0.0
```

```
>>> incr_x(p)
```

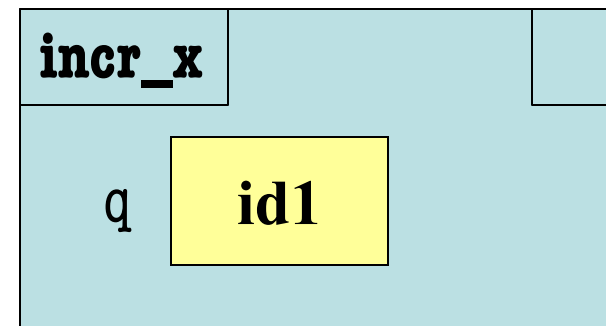
```
>>> p.x
```

```
1.0
```

Global **STUFF**



Call Frame



Example: Mutable Function Call

- **Example:**

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

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

```
>>> p.x
```

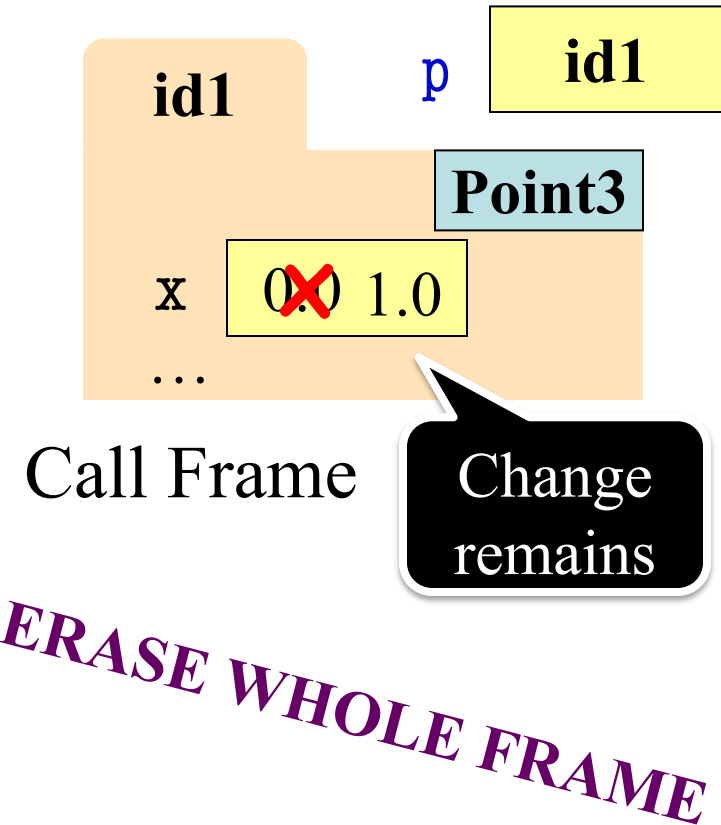
```
0.0
```

```
>>> incr_x(p)
```

```
>>> p.x
```

```
1.0
```

Global **STUFF**



Methods: Functions Tied to Objects

- Have seen object folders contain variables
 - **Syntax:** `<obj>.<attribute>` (e.g. `p.x`)
 - These are called *attributes*
- They can also contain functions
 - **Syntax:** `<obj>.<method>(<arguments>)`
 - **Example:** `p.clamp(-1,1)`
 - These are called *methods*
- Visualizer will not show these inside folders
 - Will see why in **October** (when cover Classes)

Understanding Method Calls

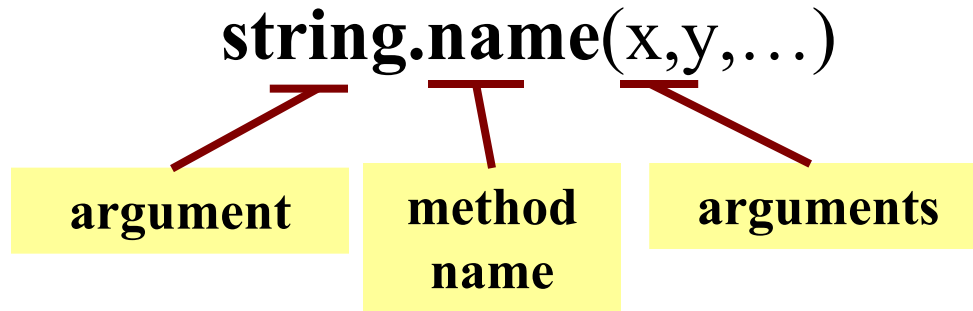
- Object before the name is an *implicit* argument

- **Example:** distance

```
>>> p = Point3(0,0,0)    # First point
>>> q = Point3(1,0,0)    # Second point
>>> r = Point3(0,0,1)    # Third point
>>> p.distance(r)        # Distance between p, r
1.0
>>> q.distance(r)        # Distance between q, r
1.4142135623730951
```

Recall: String Method Calls

- **Method calls** have the form

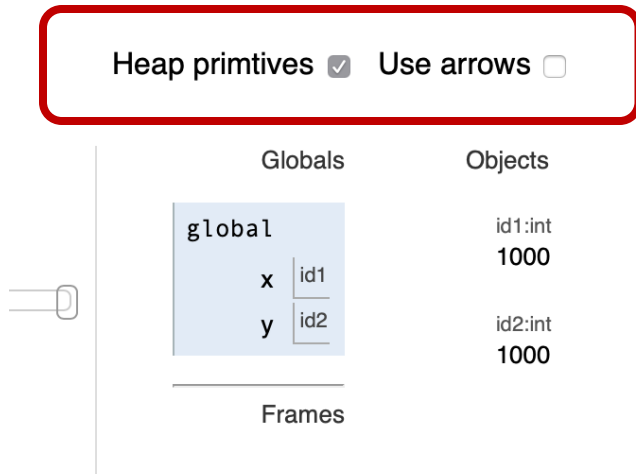


- The string in front is an **additional** argument
 - Just one that is not inside of the parentheses
 - **Why?** Will answer this later in course.

Are strings objects?

Surprise: All Values are Objects!

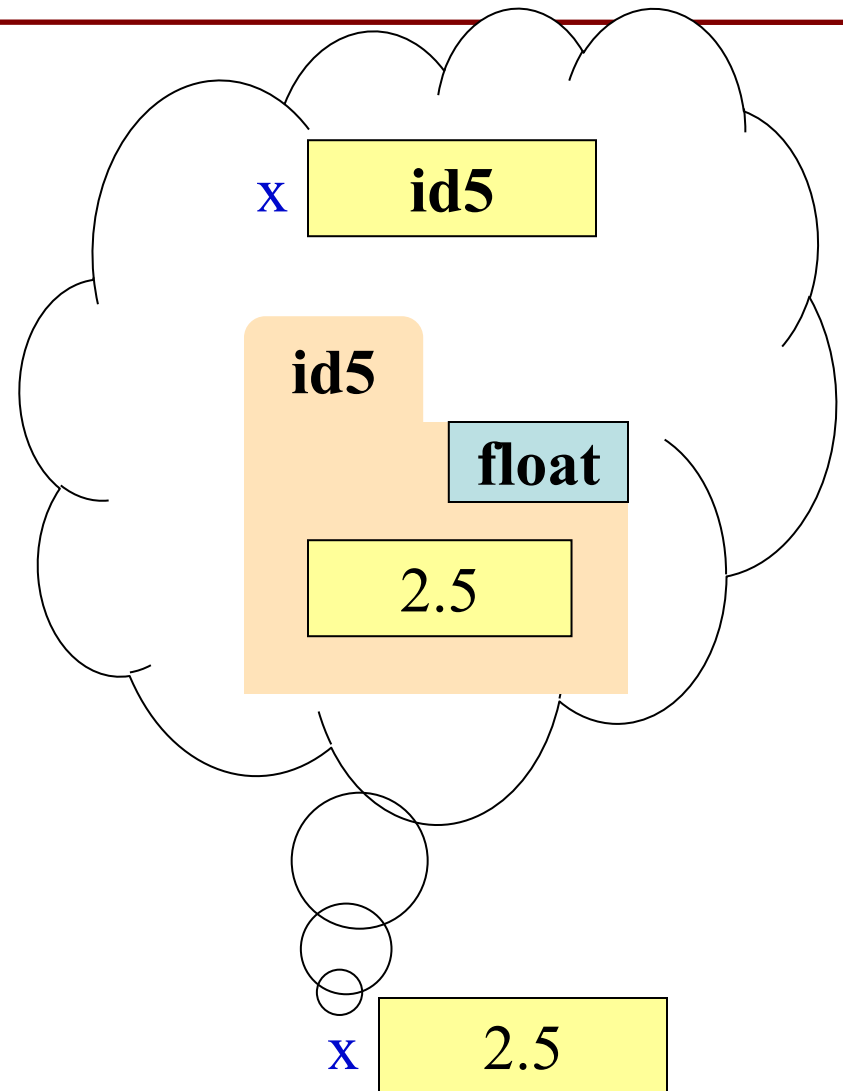
- Including basic values
 - int, float, bool, str



- **Example:**

```
>>> x = 1000
```

```
>>> id(x)
```



This Explains A Lot of Things

- Basic types act like classes
 - **Conversion function** is really a **constructor**
 - Remember **constructor**, **type** have same name

- Example:

```
>>> type(1)
```

```
<class 'int'>
```

```
>>> int('1')
```

```
1
```

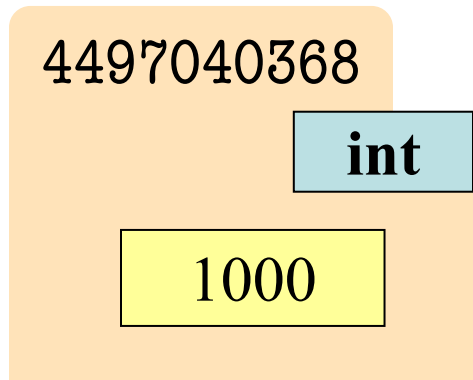
- Design goals of Python 3
 - Wanted everything an object
 - Makes processing cleaner
- But makes learning harder
 - Objects are complex topic
 - Want to delay if possible

But Not Helpful to Think This Way

- Number folders are **immutable**

- “Variables” have no names
- No way to reach in folder
- No way to change contents

x 4497040368



Makes a brand new int folder

```
>>> x = 1000
```

```
>>> y = 1000
```

```
>>> id(x)
```

```
4497040368
```

```
>>> id(y)
```

```
4497040400
```

```
>>> y = y+1
```

```
>>> id(y)
```

```
4497040432
```

But Not Helpful to Think This Way

- Number folders are **immutable**
 - “Variables” have no names
 - No way to reach in folder
 - No way to change contents
- Remember **purpose of folder**
 - Show how objects can be altered
 - Show how variables “share” data
 - This **cannot happen** in basic types
- So just **ignore the folders**
 - (The are just metaphors anyway)

```
>>> x = 1000
```

```
>>> y = 1000
```

```
>>> id(x)
```

```
4497040368
```

```
>>> id(y)
```

```
4497040400
```

```
>>> y = y+1
```

```
>>> id(y)
```

```
4497040432
```

Basic Types vs. Classes

Basic Types

- Built-into Python
- Refer to instances as *values*
- Instantiate with *literals*
- Are all immutable
- Can **ignore the folders**

Classes

- Provided by modules
- Refer to instances as *objects*
- Instantiate w/ *constructors*
- Can alter attributes
- Must **represent with folders**

In doubt? Use the Python Tutor

Where To From Here?

- Right now, just try to understand **objects**
 - **All** Python programs use objects
 - The object classes are provided by Python
- OO Programming is about **creating classes**
 - But we will not get to this until after Prelim 1
- Similar to the **separation of functions**
 - First learned to **call functions** (**create objects**)
 - Then how to **define functions** (**define classes**)