

Module 11

Objects

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 do we do that?

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

```
>>> import Point3 from introcs # 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 ints or even strings
 - **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 webpage
 - But use only that one! Else might not show all.

Metaphor: Objects are Folders

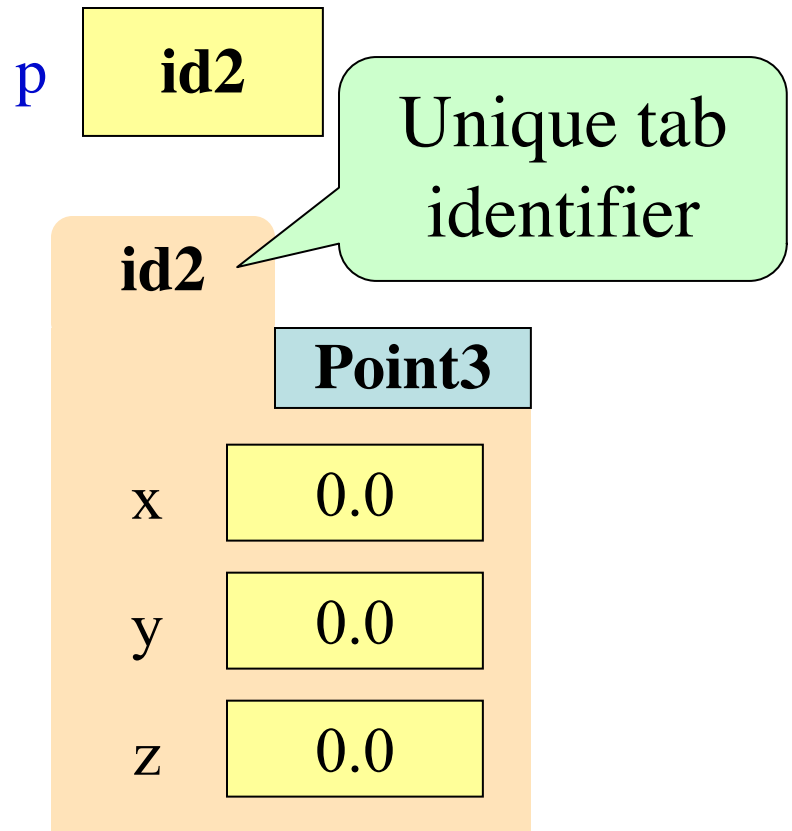
```
>>> import introcs
```

Need to import module
that has Point class.

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

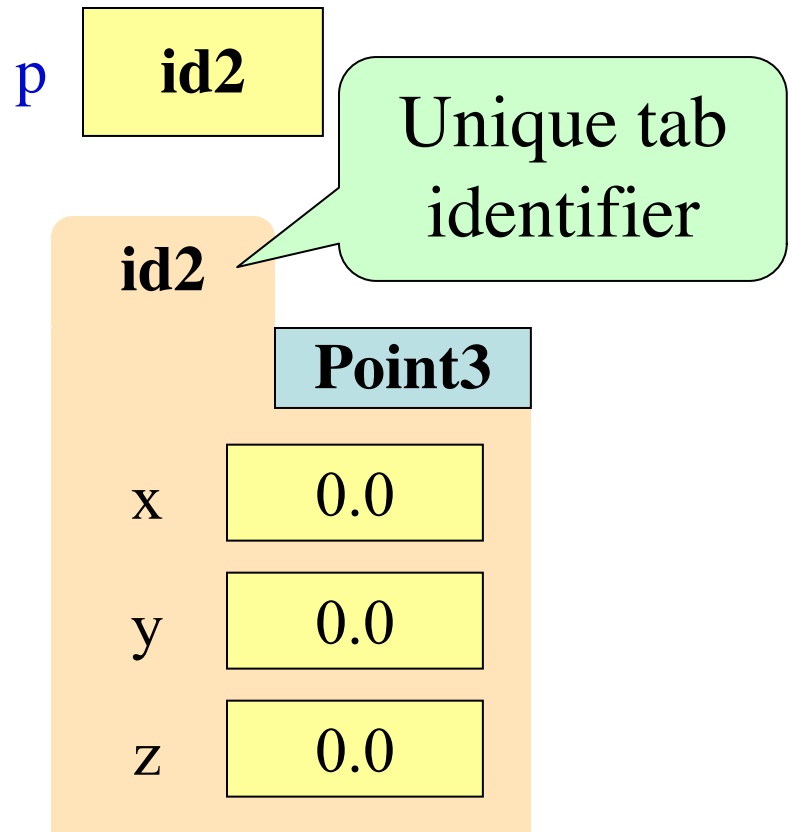
Constructor is function.
Prefix w/ module name.

Reminder: Turn off arrows!



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; picked by Python
 - Cannot ever change
 - Has no real meaning; only identifies



Metaphors Versus Reality

```
>>> 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.

p

id2

Actually a big number

id2

Point3

x

0.0

y

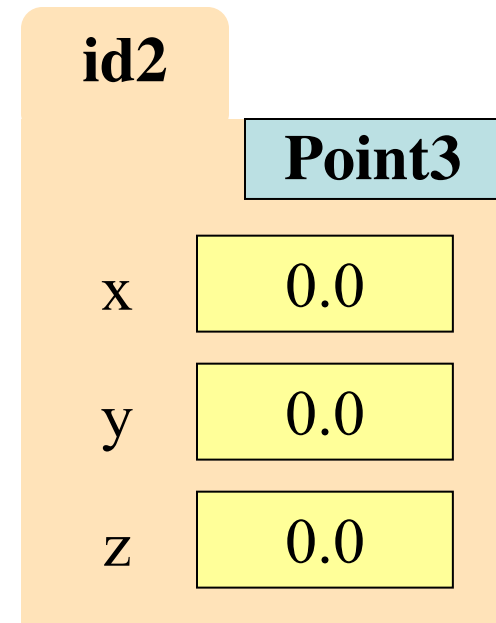
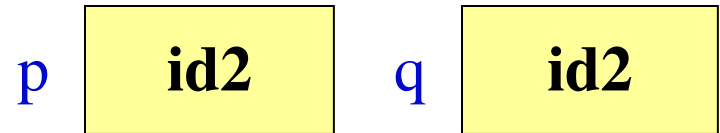
0.0

z

0.0

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



Learning with the Interactive Shell

- Interactive shell is a helpful learning tool
 - It gives you immediate feedback
 - Allows you to experiment on small programs
- Often best way to understand types in Python
 - Type an expression into interactive shell
 - Python displays back a value
 - Could use that value in your python
- **But approach only works with basic types!**

Example with a Basic Type

```
>>> 1 + 2      # Expression
```

```
3
```

```
>>> x = 3      # 3 is ALSO an expression
```

- Why does this work?
 - Basic types have **literal expressions**
 - Literals: expression and value are same
- But objects **do not have literals!**

Trying this With an Object

```
>>> Point3(0,0,0) # Expression  
<class 'Point3'>(0.0,0.0,0.0)  
>>> x = <class 'Point3'>(0.0,0.0,0.0) # ERROR!
```

- Why does this not work?
 - `<class 'Point3'>(0.0,0.0,0.0)` *not an expression*
 - Cannot type it back into Python
- This is an **object representation**

Object Representations

- Anything shown in `<>` is a **representation**
 - Quick summary of the object and its contents
 - Because interactive shell cannot draw folders
 - It is **not** a valid Python expression
- *Almost* the same as calling `repr` on the object

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

```
>>> p
```

```
<class 'Point3'>(0.0,0.0,0.0)
```

```
>>> repr(p)
```

```
"<class 'Point3'>(0.0,0.0,0.0)"
```

Object Representations

- Anything shown in `<>` is a **representation**
 - Quick summary of the object and its contents
 - Because interactive shell cannot draw folders
 - It is **not** a valid Python expression
- *Almost* the same as calling `repr` on the object

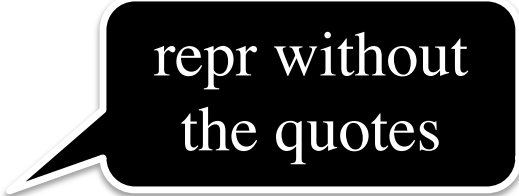
```
>>> p = Point3(0,0,0)
```

```
>>> p
```

```
<class 'Point3'>(0.0,0.0,0.0)
```

```
>>> repr(p)
```

```
"<class 'Point3'>(0.0,0.0,0.0)"
```



repr without
the quotes

Have Seen this Before with Types

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

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

```
>>> x = <class 'int'>           # ERROR!!
```

```
...
```

```
>>> x = int                     # Correct
```

```
>>> x                           # Display representation
```

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

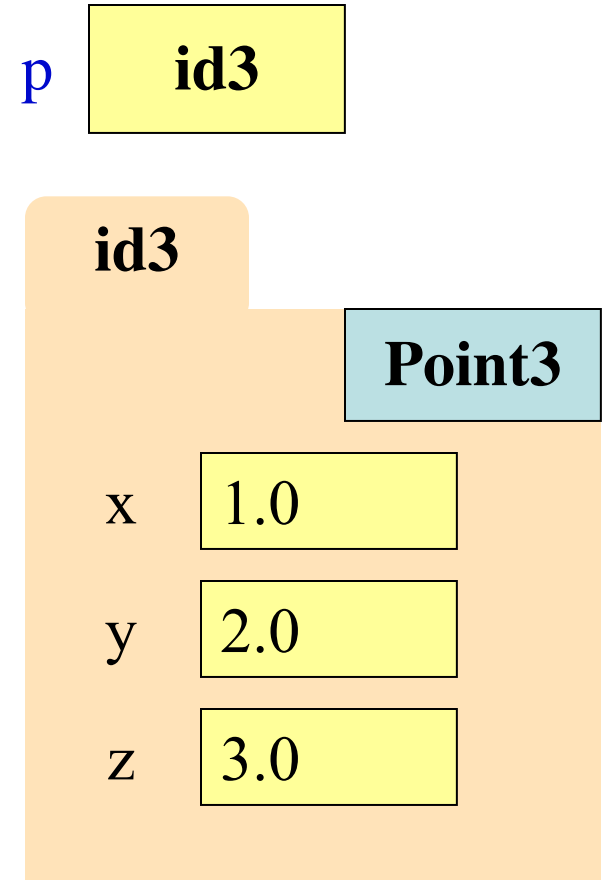
```
>>> repr(x)                     # Display the string
```

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

Objects and Attributes

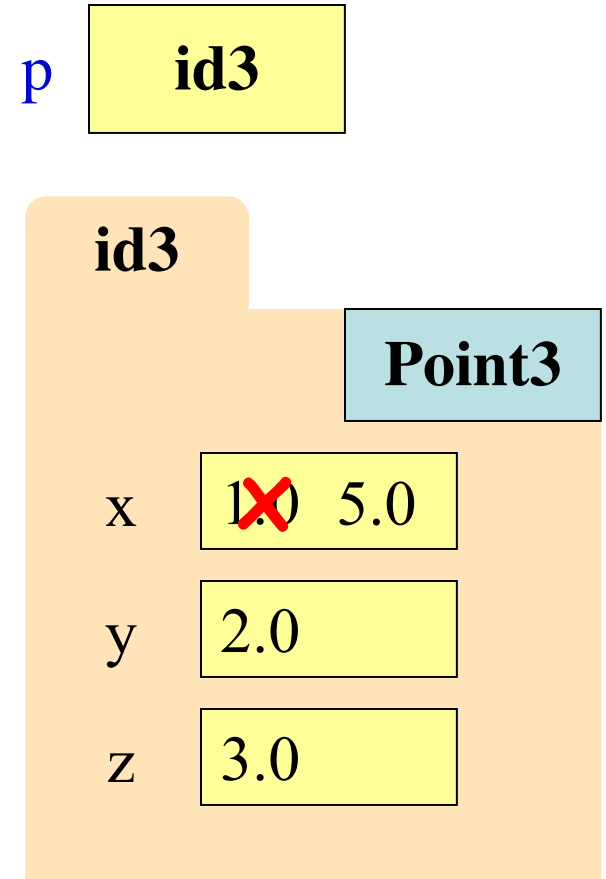
- Attributes live inside objects
 - Can access these attributes
 - Can use them in expressions
- **Access:** `<variable>.<attr>`
 - Look like module variables
 - **Example:** `math.pi`
- **Example**

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



Objects and Attributes

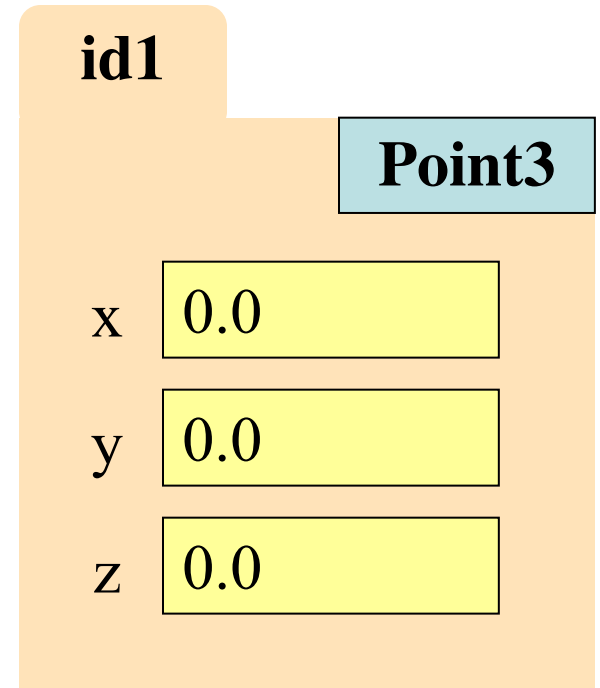
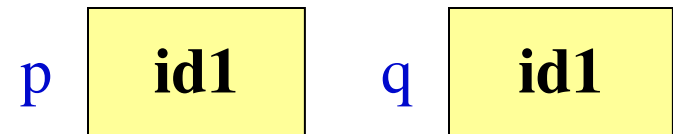
- Can also **assign** attributes
 - Reach into folder & change
 - Do without changing p
- $\langle \text{var} \rangle . \langle \text{attr} \rangle = \langle \text{exp} \rangle$
 - **Example:** $p.x = 5.0$
 - See this in visualizer
- This is very powerful
 - Another reason for objects
 - Why need visualization



Exercise: Attribute Assignment

- Recall, q gets name in p
 - >>> p = introscs.Point3(0,0,0)
 - >>> q = p
- Execute the assignments:
 - >>> p.x = 5.6
 - >>> q.x = 7.4
- What is value of p.x?

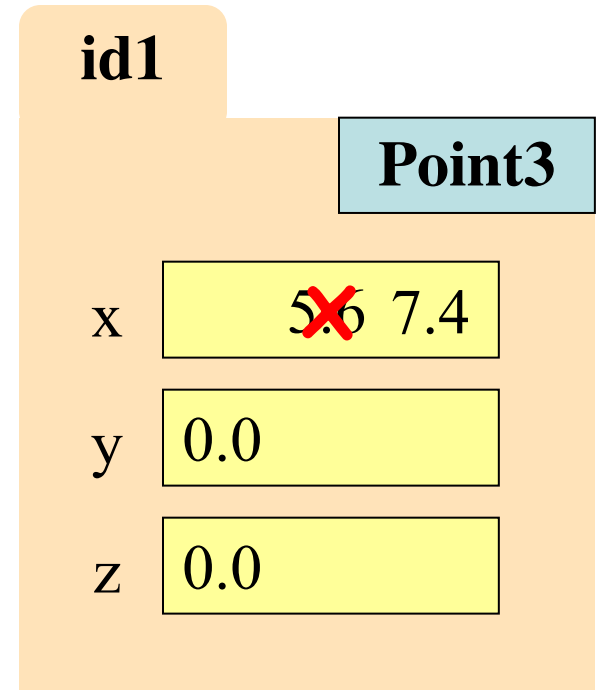
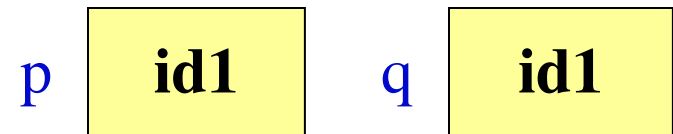
A: 5.6
B: 7.4
C: **id1**
D: I don't know



Exercise: Attribute Assignment

- Recall, q gets name in p
 - >>> p = introscs.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: id1
D: I don't know



Attribute Invariants

Invariants: Attribute Restrictions

- Some attributes have invariants
 - Restrictions on types may take
 - Similar to function preconditions

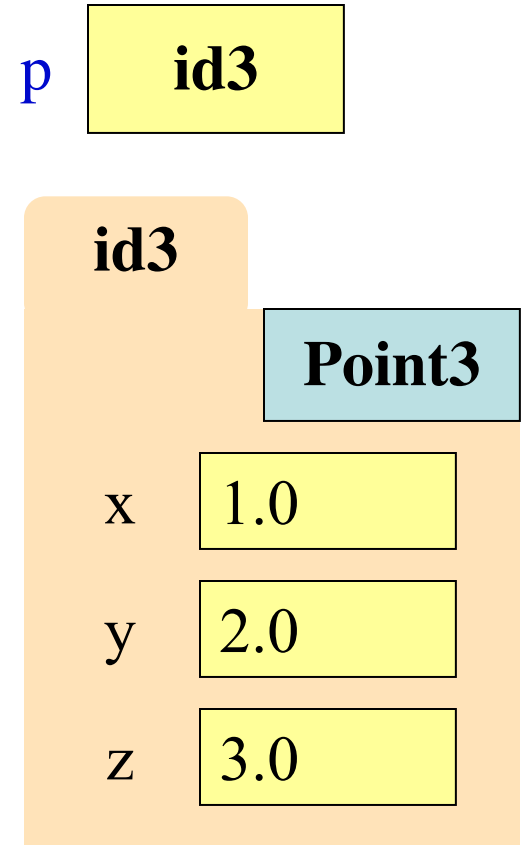
- **Example:** Point3

- Attributes must be floats
- If try an int will convert
- Else get AssertionError

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

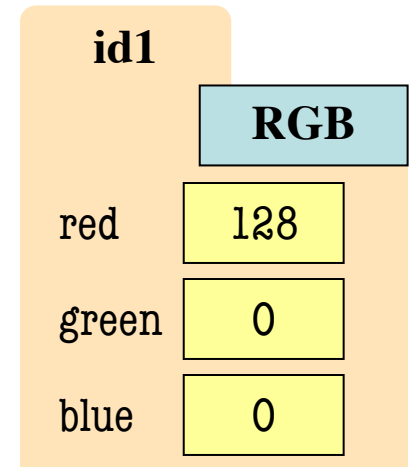
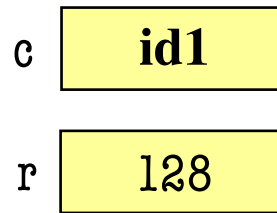
```
>>> p.x = 'a'
```

```
AssertionError
```



Another Example: RGB

- RGB is another class
 - Represents pixel colors
 - **Attribs**: red, blue, green
 - Also alpha (transparency)
 - Useful for image manip.
- All have same invariants
 - Values must be **ints**
 - Must be in range 0..255
 - Why? Color theory



```
>>> import introcs
>>> c = introcs.RGB(128,0,0)
>>> r = c.red
>>> c.red = 500 # out of range
AssertionError: 500 outside [0,255]
```

Why Invariants?

- As we said, like a function precondition
 - Function behavior not guaranteed if violated
 - Invariants ensure the object “works properly”
- If invariants are violated, say object **corrupted**
 - Same idea as when a file is corrupted
 - Corrupted objects can do weird things
- Not all objects *enforce* their invariants
 - Again same as function preconditions
 - But the classes in `intros` module do

Where Do We Find Invariants?

Attributes

red

The red channel.

Invariant: Value must be an int between 0 and 255, inclusive.

green

The green channel.

Invariant: Value must be an int between 0 and 255, inclusive.

blue ¶

The blue channel.

Invariant: Value must be an int between 0 and 255, inclusive.

alpha

The alpha channel.

This value is used for transparency effects (but not always supported).

Invariant: Value must be an int between 0 and 255, inclusive.

In the documentation!

Prefer webpage.
Help menu for
classes/objects
is very *arcane*.

Objects Can Be Used in Fruitful Functions

```
def copy2d(p):
```

```
    """Returns a 2d copy of the point p
```

```
    This function makes a new point with same x, y
    value as p, but whose z value is 0.
```

```
    Parameter p: The point to copy
```

```
    Precondition: p is a Point3 object"""
```

```
    q = introcs.Point3(p.x,p.y,0)
```

```
    return q
```



Needed to make
a new Point3

Can Also Be Used in **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
```

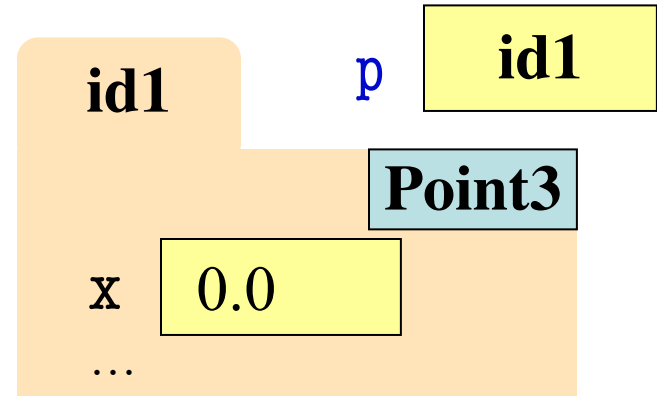
```
0.0
```

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

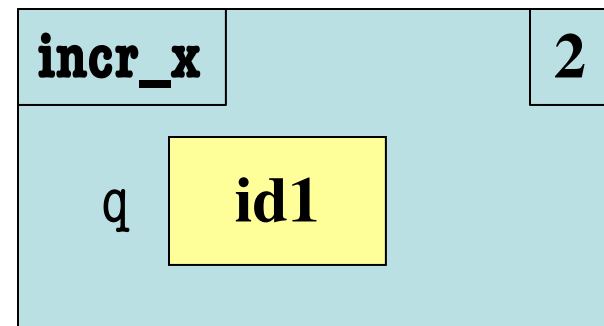
```
>>> p.x
```

```
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
```

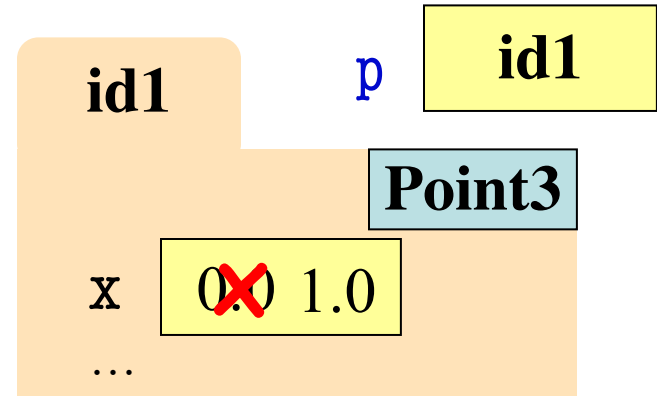
```
0.0
```

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

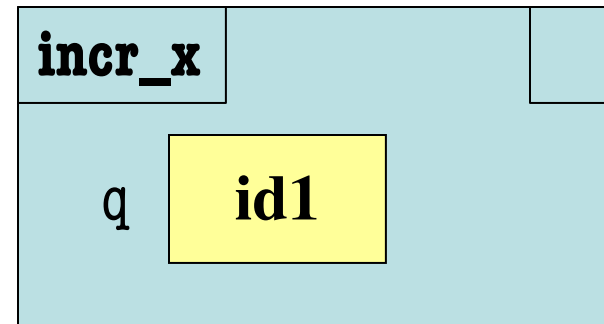
```
>>> p.x
```

```
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
```

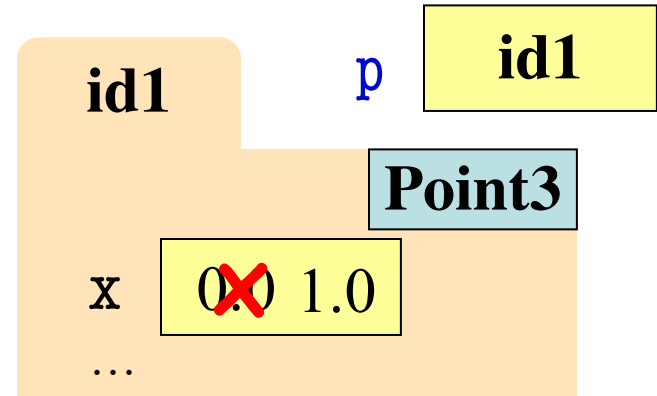
```
0.0
```

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

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

```
1.0
```

Global **STUFF**



Call Frame

ERASE WHOLE FRAME

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.abs()`
 - These are called *methods*
- Visualizer will not show these inside folders
 - Technical reasons beyond scope of course

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.abs()`
 - These are called *methods*
- Visualizer will not show these inside folders

Not a problem, since do not change like attributes.

Understanding Method Calls

- Object before the name is an *implicit* argument

- **Example:** distance

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

```
>>> q = Point3(1,0,0)
```

```
>>> r = Point3(0,0,1)
```

```
>>> p.distance(r)
```

```
1.0
```

```
>>> q.distance(r)
```

```
1.4142135623730951
```

- Clear if had call frame
 - Object gets special var
 - Called the *self* variable
- But need def for frame
 - Why visualizer no frames
 - Will cover defs later
 - So learn from practice

Where Do We Find Methods?

Mutable Methods

Mutable methods modify the underlying object.

In the documentation!

`self.interpolate(other, alpha)`

Interpolates this object with another in place

This method will modify the attributes of this object. The new attributes will be equivalent to:

```
alpha*self+(1-alpha)*other
```

according to the rules of addition and scalar multiplication.

This method returns this object for chaining.

Parameters:

- **other** (`Vector3`) – object to interpolate with
- **alpha** (`int` or `float`) – scalar to interpolate by

Returns: This object, newly modified

`self.clamp(low, high)`

Clamps this point to the range [`low`, `high`].