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

introcs 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*

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)
```

Metaphor: Objects are Folders

```
>>> import introcs
```

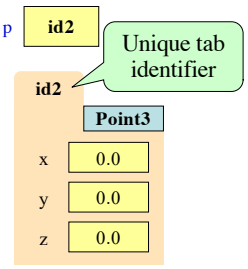
Need to import module that has Point class.

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

Constructor is function. Prefix w/ module name.

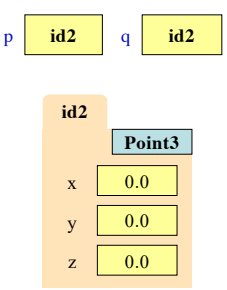
```
>>> id(p)
```

Shows the ID of p.



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

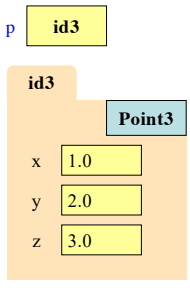


Objects and Attributes

- 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
```

a 3.0



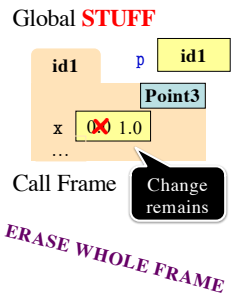
Objects Allow for Mutable Functions

- **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
```



Methods: Functions Tied to Objects

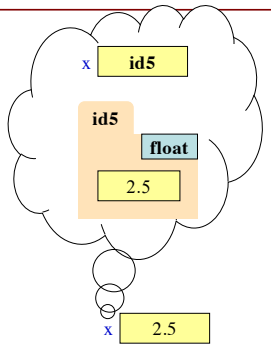
- Have seen object folders contain variables
  - **Syntax**: ⟨obj⟩.⟨attribute⟩ (e.g. p.x)
  - These are called *attributes*
- They can also contain functions
  - **Syntax**: ⟨obj⟩.⟨method⟩(⟨*arguments*⟩)
  - **Example**: p.clamp(-1,1)
  - These are called *methods*
- Visualizer will not show these inside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

Surprise: All Values are Objects!

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

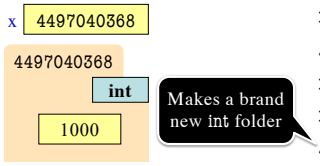
- **Example**:

```
>>> x = 1000
>>> id(x)
```



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 = 1000
>>> y = 1000
>>> id(x)
4497040368
>>> id(y)
4497040400
>>> y = y+1
>>> id(y)
4497040432
```

Basic Types vs. Classes

Basic 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

In doubt? Use the Python Tutor