Classes
Announcements for This Lecture

Assignments

- **A4 Thursday** at midnight
  - Hopefully you are on Task 4
  - Minor extension for reasons
- Will post **A5** on Wednesday
  - Written assignment like A2
  - Needs material from Tues
- Will post **A6** on **Friday**
  - Not due until November 14
  - Want to avoid exam crunch

Lab this Week

- More prelim exercises
  - This time for-loops
  - Plus a simple class

Exams

- Last week for regrades
  - Limit them to valid issues
- Getting closer to prelim 2
Recall: Objects as Data in Folders

- An object is like a **manila folder**
- It contains other variables
  - Variables are called **attributes**
  - Can change values of an attribute (with assignment statements)
- It has a “tab” that identifies it
  - Unique number assigned by Python
  - Fixed for lifetime of the object

<table>
<thead>
<tr>
<th>x</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>3.0</td>
</tr>
<tr>
<td>z</td>
<td>5.0</td>
</tr>
</tbody>
</table>
Recall: Classes are Types for Objects

- Values must have a type
  - An object is a **value**
  - Object type is a **class**

- Classes are how we add new types to Python

![Diagram](image.png)

**Types**
- **int**
- **float**
- **bool**
- **str**

**Classes**
- **Point3**
- **RGB**
- **Turtle**
- **Window**
Recall: Classes are Types for Objects

- Values must have a type
  - An object is a value
  - Object type is a class

- Classes are how we add new types to Python

But in Python3, `type` and `class` are now both synonyms
Classes Have Folders Too

Object Folders

- Separate for each *instance*

Class Folders

- Data common to all instances
Name Resolution for Objects

- \(\langle object\rangle.\langle name\rangle\) means
  - Go the folder for \textit{object}
  - Find attribute/method \textit{name}
  - If missing, check \textit{class folder}
  - If not in either, raise error

- What is in the class folder?
  - Data common to \textit{all} objects
  - First must understand the \textit{class definition}
The Class Definition

\textbf{class} \textit{<class-name>}(object):

"""Class specification"""

\textit{<function definitions>}

\textit{<assignment statements>}

\textit{<any other statements also allowed>}

\textbf{Example}

\textbf{class} Example(object):

"""The simplest possible class."""

\textbf{pass}
The Class Definition

- **keyword** `class`
  - Beginning of a class definition

**class** `<class-name>`(object):

```
"""Class specification""
```

- `<function definitions>`
- `<assignment statements>`
- `<any other statements also allowed>`

- Do not forget the colon!
- More on this later
- ...but not often used

- Goes inside a module, just like a function definition.

**Example**

```python
class Example(object):
    """The simplest possible class.""
    pass
```

Python creates after reading the class definition
Recall: Constructors

- Function to create new instances
  - Function name == class name
  - Created for you automatically
- Calling the constructor:
  - Makes a new object folder
  - Initializes attributes
  - Returns the id of the folder
- By default, takes no arguments
  - \texttt{e = Example()}

\texttt{e = Example()}
Instances and Attributes

• Assignments add object attributes
  - \(<\text{object}>.\text{<att>} = \text{<expression>}\>
  - **Example**: e.b = 42

• Assignments can add class attributes
  - \(<\text{class}>.\text{<att>} = \text{<expression>}\>
  - **Example**: Example.a = 29

• Objects can access class attributes
  - **Example**: print e.a
  - But assigning it creates object attribute
  - **Example**: e.a = 10

• **Rule**: check object first, then class
Instances and Attributes

- **Assignments add object attributes**
  - `<object>.<att> = <expression>`
  - **Example:** `e.b = 42`

- **Assignments can add class attributes**
  - `<class>.<att> = <expression>`
  - **Example:** `Example.a = 29`

- **Objects can access class attributes**
  - **Example:** `print e.a`
  - But assigning it creates object attribute
  - **Example:** `e.a = 10`

- **Rule:** check object first, then class

Not how usually done
Instances and Attributes

• Assignments add object attributes
  - `<object>.<att> = <expression>`
  - **Example**: `e.b = 42`

• Assignments can add class attributes
  - `<class>.<att> = <expression>`
  - **Example**: `Example.a = 29`

• Objects can access class attributes
  - **Example**: `print e.a`
  - But assigning it creates object attribute
  - **Example**: `e.a = 10`

• **Rule**: check object first, then class
Invariants

- Properties of an attribute that must be true
- Works like a precondition:
  - If invariant satisfied, object works properly
  - If not satisfied, object is “corrupted”
- Examples:
  - **Point3** class: all attributes must be floats
  - **RGB** class: all attributes must be ints in 0..255
- Purpose of the **class specification**
class Worker(object):

"""An instance is a worker in an organization.

Instance has basic worker info, but no salary information.

ATTRIBUTES:

lname: Worker’s last name.  [str]
ssn: Social security no.  [int in 0..999999999]
boss: Worker's boss.    [Worker, or None if no boss]
class Worker(object):
    """An instance is a worker in an organization.
    Instance has basic worker info, but no salary information.
    
    ATTRIBUTES:
    lname:  Worker's last name.  [str]
    ssn:    Social security no.  [int in 0..999999999]
    boss:   Worker's boss.       [Worker, or None if no boss]"""
Recall: Objects can have Methods

- **Method**: function tied to object
  - Function call: `<function-name>(<arguments>)`
  - Method call: `<object-variable>..<function-call>`

- **Example**: `p.distance(q)`
  - Both `p` and `q` act as arguments
  - Very much like `distanceTo(p, q)`

- For most Python objects
  - **Attributes** are in `object` folder
  - **Methods** are in `class` folder
Method Definitions

- Looks like a function `def`
  - But indented *inside* class
  - The first parameter is always called `self`
- In a method call:
  - Parentheses have one less argument than parameters
  - The object in front is passed to parameter `self`
- **Example**: `a.distance(b)`

```python
class Point3(object):
    """Instances are points in 3d space
    x: x coord [float]
y: y coord [float]
z: z coord [float]  ""

def distance(self, q):
    """Returns: dist from self to q
    Precondition: q a Point3""
    assert type(q) == Point3
    sqrdst = ((self.x-q.x)**2 +
               (self.y-q.y)**2 +
               (self.z-q.z)**2)
    return math.sqrt(sqrdst)
```

10/23/18
Methods Calls

- **Example:** `a.distance(b)`

```python
class Point3(object):
    """Instances are points in 3d space
    x: x coord [float]
y: y coord [float]
z: z coord [float]   """

def distance(self, q):
    """Returns: dist from self to q
    Precondition: q a Point3"""
    assert type(q) == Point3
    sqrdst = ((self.x - q.x)**2 +
              (self.y - q.y)**2 +
              (self.z - q.z)**2)
    return math.sqrt(squaredst)
```

```
a  id2  
  x  1.0
  y  2.0
  z  3.0

b  id3  
  x  0.0
  y  3.0
  z -1.0
```
**Methods Calls**

- **Example**: `a.distance(b)`

```python
class Point3(object):
    
    """Instances are points in 3d space
    x: x coord [float]
y: y coord [float]
z: z coord [float]  ""
    
    def distance(self, q):
        """Returns: dist from self to q
        Precondition: q a Point3""
        assert type(q) == Point3
        sqrdst = ((self.x-q.x)**2 +
                    (self.y-q.y)**2 +
                    (self.z-q.z)**2)
        return math.sqrt(sqrdst)
```

<table>
<thead>
<tr>
<th>a</th>
<th>id2</th>
<th>b</th>
<th>id3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1.0</td>
<td>x</td>
<td>0.0</td>
</tr>
<tr>
<td>y</td>
<td>2.0</td>
<td>y</td>
<td>3.0</td>
</tr>
<tr>
<td>z</td>
<td>3.0</td>
<td>z</td>
<td>-1.0</td>
</tr>
</tbody>
</table>
Initializing the Attributes of an Object (Folder)

• Creating a new Worker is a multi-step process:
  - \( w = \text{Worker}() \)
  - \( w.lname = 'White' \)
  - ...

• Want to use something like
  - \( w = \text{Worker}('White', \text{1234}, \text{None}) \)
  - Create a new Worker and assign attributes
    - \( lname \) to 'White', \( ssn \) to 1234, and \( boss \) to None

• Need a custom constructor
Special Method: __init__

```python
def __init__(self, n, s, b):
    """Initializer: creates a Worker

    Has last name n, SSN s, and boss b

    Precondition: n a string, s an int in
    range 0..999999999, and b either
    a Worker or None.
    self.lname = n
    self.ssn = s
    self.boss = b"
```

Called by the constructor

```python
w = Worker('White', 1234, None)
```

```
Worker

lname    'White'
ssn      1234
boss     None
```
Special Method: `__init__`

```python
def __init__(self, n, s, b):
    """Initializer: creates a Worker
    Has last name n, SSN s, and boss b

    Precondition: n a string, s an int in range 0..999999999, and b either
    a Worker or None.
    self.lname = n
    self.ssn = s
    self.boss = b"
```

Called by the constructor

```
id8
Worker
lname   'White'
ssn     1234
boss    None
```

use `self` to assign attributes
Evaluating a Constructor Expression

Worker('White', 1234, None)

1. Creates a new object (folder) of the class Worker
   - Instance is initially empty
2. Puts the folder into heap space
3. Executes the method `__init__`
   - Passes folder name to self
   - Passes other arguments in order
   - Executes the (assignment) commands in initializer body
4. Returns the object (folder) name
Aside: The Value None

- The boss field is a problem.
  - boss refers to a Worker object
  - Some workers have no boss
  - Or maybe not assigned yet (the buck stops there)
- **Solution**: use value None
  - None: Lack of (folder) name
  - Will reassign the field later!
- Be careful with None values
  - var3.x gives error!
  - There is no name in var3
  - Which Point to use?
Making Arguments Optional

• We can assign default values to `__init__` arguments
  ▪ Write as assignments to parameters in definition
  ▪ Parameters with default values are optional

• Examples:
  ▪ `p = Point3()`  # (0,0,0)
  ▪ `p = Point3(1,2,3)`  # (1,2,3)
  ▪ `p = Point3(1,2)`  # (1,2,0)
  ▪ `p = Point3(y=3)`  # (0,3,0)
  ▪ `p = Point3(1,z=2)`  # (1,0,2)

```python
class Point3(object):
    
    """Instances are points in 3d space"
    x: x coord [float]
    y: y coord [float]
    z: z coord [float]  """

    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes a new Point"
        Precondition: x,y,z are numbers""
        self.x = x
        self.y = y
        self.z = z
        ...
```

10/23/18  Classes
Making Arguments Optional

- We can assign default values to `__init__` arguments
  - Write as assignments to parameters in definition
  - Parameters with default values are optional

- **Examples:**
  - `p = Point3()` # (0,0,0)
  - `p = Point3(1)` # (1,0,0)
  - `p = Point3(1,2)`
  - `p = Point3(y=3)` # (0,3,0)
  - `p = Point3(1,z=2)`

```python
class Point3(object):
    """Instances are points in 3d space
    x: x coord [float]
    y: y coord [float]
    z: z coord [float] ""

def __init__(self, x=0, y=0, z=0):
    """Initializer: makes a new Point
    Precondition: x,y,z are numbers""
    self.x = x
    self.y = y
    self.z = z
```

Assigns in order

Use parameter name when out of order

Can mix two approaches
Making Arguments Optional

- We can assign default values to `__init__` arguments
  - Write as assignments to parameters in definition
  - Parameters with default values are optional

**Examples:**
- `p = Point3()`  # (0, 0, 0)
- `p = Point3(x=1, y=2)`
- `p = Point3(z=3)`  # (0, 0, 3)
- `p = Point3(1, y=2)`  # (1, 2, 0)
- `p = Point3(y=3)`  # (0, 3, 0)
- `p = Point3(1, z=2)`  # (1, 0, 2)

```python
class Point3(object):
    """Instances are points in 3d space"""
    x: x coord [float]
    y: y coord [float]
    z: z coord [float]  """

    def __init__(self, x=0, y=0, z=0):
        """Initializer: makes a new Point"""
        self.x = x
        self.y = y
        self.z = z
```

Assigns in order

Use parameter name when out of order

Can mix two approaches

Not limited to methods. Can do with any function.