Lecture 9

Dictionaries & Objects
Type: Set of values and the operations on them

- **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:** +, slicing

- **Type list:**
  - **Values:** list of values
    - Indicate with []
  - **Ops:** +, slicing
Type: Set of values and the operations on them

- **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
  - **Ops:** +, slicing
    - Double quotes: "abc"
    - Single quotes: 'abc'

- **Type list:**
  - **Values:** list of values
  - **Ops:** +, slicing
    - Indicate with []

Are these the only types that exist?
# Dictionaries (Type `dict`)

## Description

- List of **key-value** pairs
  - Keys are unique
  - Values need not be
- Example: net-ids
  - net-ids are **unique** (a key)
  - names need not be (values)
  - `js1` is John Smith (class '13)
  - `js2` is John Smith (class '16)
- Many other applications

## Python Syntax

- Create with format: `{k1:v1, k2:v2, …}`
- Keys must be non-mutable
  - ints, floats, bools, strings
  - **Not** lists or custom objects
- Values can be anything
- Example:

```python
d = {'js1':'John Smith', 'js2':'John Smith', 'wmw2':'Walker White'}
```
Using Dictionaries (Type dict)

- Access elts. like a list
  - `d['js1']` evaluates to 'John'
  - But cannot slice ranges!
- Dictionaries are **mutable**
  - Can reassign values
    - `d['js1'] = 'Jane'`
  - Can add new keys
    - `d['aa1'] = 'Allen'`
  - Can delete keys
    - `del d['wmw2']`

\[
d = \{ 'js1': 'John', 'js2': 'John', 'wmw2': 'Walker' \}
\]
Using Dictionaries (Type `dict`)

- Access elts. like a list
  - `d['js1']` evaluates to 'John'
  - But cannot slice ranges!
- Dictionaries are **mutable**
  - Can reassign values
  - `d['js1']` = 'Jane'
  - Can add new keys
  - `d['aa1']` = 'Allen'
  - Can delete keys
  - `del d['wmw2']`

```
d = {'js1': 'John', 'js2': 'John', 'wmw2': 'Walker'}
```

Key-Value order in folder is not important
Using Dictionaries (Type `dict`)

- Access elts. like a list
  - `d['js1']` evaluates to 'John'
  - But cannot slice ranges!
- Dictionaries are **mutable**
  - Can reassign values
  - `d['js1'] = 'Jane'`
  - Can add new keys
  - `d['aa1'] = 'Allen'`
  - Can delete keys
  - `del d['wmw2']`

```python

d = {'js1': 'John', 'js2': 'John', 'wmw2': 'Walker'}
```

9/26/16 Dictionaries & Objects 7
Using Dictionaries (Type `dict`)

- Access elts. like a list
  - `d['js1']` evaluates to 'John'
  - But cannot slice ranges!
- Dictionaries are **mutable**
  - Can reassign values
    - `d['js1']` = 'Jane'
  - Can add new keys
    - `d['aa1']` = 'Allen'
  - Can delete keys
    - `del d['wmw2']`

\[
d = \{ 'js1' : 'John', 'js2' : 'John', 'wmw2' : 'Walker' \}
\]

Deleting key deletes both
Dictionaries and For-Loops

- Dictionaries != sequences
  - Cannot slice them

- **Different** inside for loop
  - Loop variable gets the key
  - Then use key to get value

- Has **methods** to convert dictionary to a sequence
  - Seq of keys: `d.keys()`
  - Seq of values: `d.values()`
  - key-value pairs: `d.items()`

```python
for k in d:
    # Loops over keys
    print k  # key
    print d[k]  # value

# To loop over values only
for v in d.values():
    print v  # value
```

See grades.py
Thinking About Assignment 2

• **A2**: three color models
  - **RGB**: 3 ints 0 to 255
  - **CMYK**: 4 floats 0.0 to 100.0
  - **HSV**: 3 floats, multi. bounds
  - We could represent as lists

• Can get really confusing
  - Easy to mix-up models
  - Easy to go out of bounds

• **We want custom types**
  - One for each color model
  - Motivation for *classes*
Classes are Customized Types

- Classes are how we add new types to Python
- Values look like *dicts*
  - Represent as a folder
  - Variables are named

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

**Classes**
- RGB
- CMYK
- HSV

**RGB**
- id2
- red: 255
- green: 128
- blue: 0

**class name**
Classes are how we add new types to Python

Class values are called objects

• Classes are Customized Types
• Values look like dicts
  ▪ Represent as a folder
  ▪ Variables are named

Types
• int
• float
• bool
• str
• list

Class values are called objects
• RGB
• CMYK
• HSV

id2
• RGB
  • red
  • green
  • blue

255
128
0

class name
Why Are They Better Than dicts?

- Can add new variables
- Does not check bounds of the content variables
- Variables fixed (sort-of)
- Possibly checks bounds of the content variables
Why Are They Better Than dicts?

- Can add new variables
- Does not check bounds of the content variables
- Variables fixed (sort-of)
- Possibly checks bounds of the content variables

Designed for the purpose of safety (and simplicity)
Using Classes in Python

• **Modules** provide classes
  – Import to use the class
  – Will show contents later

• **Example**: colormodel
  – Color classes for A3
  – RGB, CMYK, HSV

• **Example**: geom
  – Geometry classes
  – Point2, Point3
Constructor: Function to make Objects

- How do we create objects?
  - Other types have **literals**
  - **Example**: 1, 'abc', true
  - No such thing for objects
- **Constructor Function**:
  - Same name as the class
  - **Example**: Point3(0,0,0)
  - Makes an object (manila folder)
  - Returns folder ID as value
- **Example**: p = Point3(0, 0, 0)
  - Creates a Point object
  - Stores object’s ID in p
Constructors and Modules

>>> import geom

Need to import module that has Point class.

>>> p = geom.Point3(0,0,0)

Constructor is function. Prefix w/ module name.

>>> id(p)

Shows the ID of p.

Actually a big number
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!
- Like we saw with lists
  - Reason for using folders
Objects and Attributes

- Attributes are variables that live inside of objects
  - Can **use** in expressions
  - Can **assign** values to them

- **Access**: `<variable>_<attr>`
  - **Example**: `p.x`
  - Look like module variables

- Putting it all together
  - `p = geom.Point3(1,2,3)`
  - `p.x = p.y + p.z`
Objects and Attributes

- Attributes are variables that live inside of objects
  - Can *use* in expressions
  - Can *assign* values to them

- **Access**: `<variable>..<attr>`
  - **Example**: `p.x`
  - Look like module variables

- Putting it all together
  - `p = geom.Point3(1,2,3)`
  - `p.x = p.y + p.z`
Exercise: Attribute Assignment

- Recall, q gets name in p
  ```python
  >>> p = geom.Point3(0,0,0)
  >>> q = p
  ```
- Execute the assignments:
  ```python
  >>> 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 = geom.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 = geom.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
Methods: Functions Tied to Objects

• **Method**: function tied to object
  - Method call looks like a function call preceded by a variable name:
    \[ \text{variable} \cdot \text{method}(\text{arguments}) \]
  - **Example**: `p.distanceTo(q)`
  - **Example**: `p.abs()` # makes x,y,z ≥ 0

• Just like we saw for strings
  - `s = 'abracadabra'`
  - `s.index('a')`

• **Are strings objects?**
Surprise: All Values are in Objects!

• Including basic values
  ▪ int, float, bool, str

• Example:
  >>> x = 2.5
  >>> id(x)

• But they are *immutable*
  ▪ Contents cannot change
  ▪ Distinction between *value* and *identity* is immaterial
  ▪ So we can ignore the folder
Surprise: All Values are in Objects!

• Including basic values
  ▪ int, float, bool, str

• Example:
  >>> x = 'foo'
  >>> id(x)

• But they are immutable
  ▪ No string method can alter the contents of a string
  ▪ x.replace('o','y') evaluates to 'fyy' but x is still 'foo'
  ▪ So we can ignore the folder
# Base Types vs. Classes

<table>
<thead>
<tr>
<th>Base Types</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Built-into Python</td>
<td>• Provided by modules</td>
</tr>
<tr>
<td>• Refer to instances as <em>values</em></td>
<td>• Refer to instances as <em>objects</em></td>
</tr>
<tr>
<td>• Instantiate with <em>_literals</em></td>
<td>• Instantiate w/ <em>constructors</em></td>
</tr>
<tr>
<td>• Are all immutable</td>
<td>• Can alter attributes</td>
</tr>
<tr>
<td>• Can ignore the folders</td>
<td>• Must represent with folders</td>
</tr>
</tbody>
</table>