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

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

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["aal"] = 'Allen'
  - Can delete keys
    - del d["wmw2"]

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

```python
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']`

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

```python
id8
d
    id8
dict
    'js1'  'Jane'
    'js2'  'John'
    'wmw2' 'Walker'
    'aa1'  'Allen'
```
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['aal'] = 'Allen'`
  - Can delete keys
    - `del d['wmw2']`

```python
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()`

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, mult. 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

### Example

```python
id2 = RGB(
    red=255,
    green=128,
    blue=0
)
```

- `id2` is an instance of the `RGB` class.
Classes are how we add new types to Python

- Classes are Customized

Values look like **dicts**
- Represent as a folder
- Variables are named

**Class values are called objects**

- RGB
- CMYK
- HSV

**Examples**

```
id2
```

<table>
<thead>
<tr>
<th>class name</th>
<th>red</th>
<th>green</th>
<th>blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>255</td>
<td>128</td>
<td>0</td>
</tr>
</tbody>
</table>
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
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

```python
>>> 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

\[\begin{align*}
  p & \quad \text{id2} \\
  q & \quad \text{id2}
\end{align*}\]

\[\begin{align*}
  \text{id2} & \quad \text{Point3} \\
  x & \quad 0.0 \\
  y & \quad 0.0 \\
  z & \quad 0.0
\end{align*}\]
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`

Dictionaries & Objects
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
  >>> 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  
  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:
    \[
    \langle \text{variable} \rangle.\langle \text{method} \rangle(\langle \text{arguments} \rangle)
    \]
  - **Example**: `p.distanceTo(q)`
  - **Example**: `p.abs()` # makes \(x, y, z \geq 0\)

- 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:**
  ```python
  >>> x = 2.5
  >>> id(x)
  >>> id(x)
  ```

- But they are *immutable*
  - Contents cannot change
  - Distinction between *value* and *identity* is immaterial
  - So we can ignore the folder

9/26/16 Dictionaries & Objects 25
Surprise: All Values are in Objects!

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

- **Example:**
  ```python
  >>> 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

9/26/16

Dictionaries & Objects
# Base Types vs. Classes

## Base 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 with *constructors*
- Can alter attributes
- Must represent with folders