Lecture #14: Dictionaries and Classes

Dictionaries

- *Dictionaries* (type dict) are mutable mappings from one set of values (called *keys*) to another.
- Constructors:

```
>>> {}   # A new, empty dictionary
>>> { 'brian' : 29, 'erik': 27, 'zack': 18, 'dana': 25 }
{'brian': 29, 'erik': 27, 'dana': 25, 'zack': 18}
>>> L = ('aardvark', 'axolotl', 'gnu', 'hartebeest', 'wombat')
>>> successors = { L[i-1] : L[i] for i in range(1, len(L)) }
>>> successors
{'aardvark': 'axolotl', 'hartebeest': 'wombat',
 'axolotl': 'gnu', 'gnu': 'hartebeest'}
```

- Queries:

```
>>> len(successors)
4
>>> 'gnu' in successors
True
>>> 'wombat' in successors
False
```


Dictionary Selection and Mutation

- Selection and Mutation

```
>>> ages = { 'brian' : 29, 'erik': 27, 'zack': 18, 'dana': 25 }
>>> ages['erik']
27
>>> ages['paul']
...
KeyError: 'paul'
>>> ages.get('paul', "?")
'?'
```

- Mutation:

```
>>> ages['erik'] += 1; ages['john'] = 56
ages
{'brian': 29, 'john': 56, 'erik': 28, 'dana': 25, 'zack': 18}
```


Dictionary Keys

- Unlike sequences, ordering is not defined.
- Keys must typically have immutable types that contain only immutable data [can you guess why?] that have a `__hash__` method. Take CS61B to find out what's going on here.
- When converted into a sequence, get the sequence of keys:

```
>>> ages = { 'brian' : 29, 'erik': 27, 'zack': 18, 'dana': 25 }
>>> list(ages)
['brian', 'erik', 'dana', 'zack']
>>> for name in ages: print(ages[name], end=",")
29, 27, 25, 18,
```


A Dictionary Problem

```
def frequencies(L):
    """A dictionary giving, for each w in L, the number of times w
    appears in L.
    >>> frequencies(['the', 'name', 'of', 'the', 'name', 'of', 'the',
    ...              'song'])
    {'of': 2, 'the': 3, 'name': 2, 'song': 1}
    """
```


Using Only Keys

- Suppose that all we need are the keys (values are irrelevant):

```
def is_duplicate(L):
    """True iff L contains a duplicated item."""
    items = {}
    for x in L:
        if x in items: return True
        items[x] = True   # Or any value
    return False
def common_keys(D0, D1):
    """Return dictionary containing the keys in both D0 and D1."""
    result = {}
    for x in D0:
         if x in D1: result[x] = True
    return result
```

- These dictionaries function as *sets* of values.

Sets

- Python supplies a specialized set data type for slightly better syntax (and perhaps speed) than dictionaries for set-like operations.
- Operations

Set operation	Python Syntax	Modification
$\{\}$	`set([])`	
$\{1,2,3\}$	`{ 1, 2, 3 }, set([1,2,3])`	
$\{x \in L \mid P(x)\}$	`{ x for x in L if P(x) }`	
$A \cup B$	`A | B`	`A |= B`
$A \cap B$	`A & B`	`A &= B`
$A \setminus B$	`A - B`	`A -= B`
$A \cup \{x\}$	`A | {x}`	`A.add(x)`
$A \setminus \{x\}$	`A - {x}`	`A.discard(x)`
$x \in A$	`x in A`	
$A \subseteq B$	`A <= B`	

Reworked Examples with Sets

```
def is_duplicate(L):
    """True iff L contains a duplicated item."""
    items = set([]){}
    for x in L:
        if x in items: return True
        items.add(x)
    return False
def common_keys(D0, D1):
    """Return set containing the keys in both D0 and D1."""
    return set(D0) & set(D1)
```

- As shown in the last example, anything that can iterated over can be used to create a set.

Extending the Mutable Objects: Classes

- We've seen a variety of builtin mutable types (sets, dicts, lists).
- . . . And a general way of constructing new ones (functions referencing nonlocal variables).
- But in actual practice, we use a different way to construct new types—syntax that leads to clearer programs that are more convenient to read and maintain.
- The Python **class** statement defines new classes or types, creating new, vaguely dictionary-like varieties of object.

Simple Classes: Bank Account

```
      type
      name
class Account:
    constructor method
    def __init__(self, initial_balance):
        self.__balance = initial_balance

    def balance(self): instance method
        return self.__balance instance variable

    def deposit(self, amount):
        if amount < 0:
            raise ValueError("negative deposit")
        self.__balance += amount

    def withdraw(self, amount):
        if 0 <= amount <= self.__balance:
            self.__balance -= amount
        else: raise ValueError("bad withdrawal")
```

```
>>> mine = Account(1000)
>>> mine.deposit(100)
>>> mine.balance()
1100
>>> mine.withdraw(200)
>>> mine.balance()
900
```


Class Concepts

- Classes beget objects called *instances*, created by "calling" the class: Account(1000).
- Each such Account object contains *attributes*, accessed using object.attribute notation.
- The defs inside classes define attributes called *methods* (full names: Account.balance, etc.) Each object has a copy.
- A method call mine.deposit(100) is essentially the same as Account.deposit(mine, 100).
- By convention, we therefore call the first argument of a method something like "self" to indicate that it is the object from which we got the method.
- When an object is created, the special __init__ method is called first.
- Each Account object has other attributes (__balance), which we create by assignment, again using dot notation.