Python Dictionaries

Python dictionaries are the only built-in mapping type:
- unordered (modeled by mathematical set)
- they store key: value pairs
- key must be immutable (why?)
  - no lists or dictionaries as keys
- keys must be unique
- values need not be unique
- value can be any valid object
- Heterogeneous
  - both keys and values can be mixed types
Creating a dictionary

d1 = {1: 'a', 2: 'b', 3: 'c'}
d2 = {5: [5, 10, 15], 10: [10, 20 , 30], 'a': 0}

d3 = {}
d3[10] = 'ten'
d3[100] = 'hundred'

d4 = dict()
d4['Pinto'] = 1.7
d4['Bluto'] = 0.0
d4['D-Day'] = None

Custom class objects as keys

To be used as a key an object must be hashable

a1 = Account('Fred', 1000)
a2 = Account('Ethel', 5000)

d = {a1: 'checking'}  # a1 can't be hashed so...

TypeError: unhashable instance
We can fix that

```python
class Account(object):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def __hash__(self):
        return hash((self.name, self.balance))

    def __eq__(self, other):
        return (self.name, self.balance) == (other.name, other.balance)

    def deposit(self, amt):
        self.balance += amt
```

Et voila

To be used as a key an object must be hashable, and now it is!

```python
a1 = Account('Fred', 1000)
a2 = Account('Ethel', 5000)

d = {a1: 'checking', a2: 'savings'}
print(d[a1])

Output: 'checking'
```
There’s a big BUT

```python
a1 = Account('Fred', 1000)
a2 = Account('Ethel', 5000)
d = {a1: 'checking', a2: 'savings'}
print(d[a1])
Output: 'checking'
a1.deposit(100)  # A change to the instance used as key
print(d[a1])
KeyError: <account.Account instance at 0x104eee5f0>
```

The moral of the story

You can “trick” Python into letting you use instances of a mutable object as a key (by defining __hash__ and __eq__) but if you change the values of the instances, you cause Pypocalypse.
Methods: the basics

• \texttt{len(d)} : returns number of items

• \texttt{d[k]} : returns value associated with \texttt{k}

• \texttt{d[k] = v} : sets \texttt{d[k]} to \texttt{v} (defines new item if \texttt{k} not already in \texttt{d})

• \texttt{del d[k]} : removes item with key \texttt{k} from \texttt{d}

• \texttt{k in d} : returns True if \texttt{k} is a key in \texttt{d}

Making a copy

\begin{verbatim}
d.copy()

makes shallow copies of the objects in \texttt{d}
and puts them in a new dictionary

\texttt{d1 = \{’L1’: [1, 2, 3], ’L2’: [2, 4, 6]\}}
\texttt{d2 = d1.copy()}
\texttt{d1[L1].append(4)}
\texttt{print(d1[L1])} \rightarrow \texttt{[1, 2, 3, 4]}
\texttt{print(d2[L1])} \rightarrow \texttt{[1, 2, 3, 4]}
\end{verbatim}

Why?
Another copy example

```python
Another copy example
d1 = {'L1': [1, 2, 3], 'L2': [2, 4, 6]}
d2 = d1.copy()
d1[L1] = [3, 6, 9]
print(d1[L1])  # [3, 6, 9]
print(d2[L1])  # [1, 2, 3]
```

Why?

Extending a dictionary

```python
Extending a dictionary
d1.update(d2)
    adds all items from d2 to d1
d1 = {1: 'CS224', 2: 'CS120'}
d2 = {3: 'CS225', 4: 'CS353'}
d3 = {1: 'CS220', 5: 'CS340'}
d2.update(d1)
    (3: 'CS225', 4: 'CS353', 1: 'CS224', 2: 'CS120')
d3.update(d1)
    (5: 'CS340', 1: 'CS224', 2: 'CS120')
```
Methods that return a sequence

```python
d.keys()
returns a sequence of the keys in d

d.values()
returns a sequence of the values in d

d.items()
returns a sequence of all (key, value) pairs in d
```

Methods that remove items

```python
d.pop(k [, default])
returns d[k], if found, and removes it from d;
otherwise returns default, if supplied or raises
KeyError if not — why would you want this?

d.popitem()
removes a random (key, value) pair from d and
returns it as a tuple

d.clear()
removes all items from d
Methods that remove items

\[ d.pop(k [, default]) \]

- returns \( d[k] \), if found, and removes it from \( d \); otherwise returns default, if supplied or raises KeyError if not — why would you want this?

Supplying default (None, for example) allows you to attempt to pop items without concern for whether or not they exist — there is no danger of a KeyError

Oddities

\[ \text{dict.fromkeys}(s [, , value]) \]

- returns a new dictionary with keys taken from sequence \( s \) and all values set to value if supplied, None otherwise. Note that it’s a class method.

\[ d.get(k [, , value]) \]

- returns \( d[k] \) if found; otherwise returns value if supplied or None if not. Because of this, no chance of KeyError (unlike using \( d[k] \))

\[ d.setdefault(k [, , value]) \]

- returns \( d[k] \) if found; otherwise returns value and sets \( d[k] = value \), or None if value not supplied
More oddities

list(d)
    returns a list of the keys from d. This is just casting the dictionary to a list but only the keys are put in the list

for e in d:
    print e
    -- prints the keys from d

for e in d:
    print e, d[e]
    -- prints the keys and values from d