UNIVERSITY of WISCONSIN
LA CROSSE
COMPUTER SCIENCE

CS 224 Introduction to Python

Dictionaries

1

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

2

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
```

3

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
```

4

We can fix that

```
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
```

Both must be defined

5

Et voila

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

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

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

Output: 'checking'

6

There's a big BUT

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

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

Output: 'checking'

a1.deposit(100)
print(d[a1])

KeyError: <account.Account instance at 0x104eee5f0>
```

A change to the instance used as key

7

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.

8

Methods: the basics

- `len(d)` : returns number of items
- `d[k]` : returns value associated with k
- `d[k] = v` : sets d[k] to v (defines new item if k not already in d)
- `del d[k]` : removes item with key k from d
- `k in d` : returns True if k is a key in d

9

Making a copy

```
d.copy()
     makes shallow copies of the objects in d
     and puts them in a new dictionary

d1 = {'L1': [1, 2, 3], 'L2': [2, 4, 6]}
d2 = d1.copy()

d1['L1'].append(4)
print(d1['L1'])  ------------------>  [1, 2, 3, 4]
print(d2['L1'])  ------------------>  [1, 2, 3, 4]
```

Why?

10

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?

11

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'}
```

Key Collision: imported item replaces original

```
d2.update(d1)
   {3: 'CS225', 4: 'CS353', 1: 'CS224', 2: 'CS120'}

d3.update(d1)
   {5: 'CS340', 1: 'CS224', 2: 'CS120'}
```

12

Methods that return a sequence

```
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
```

13

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?

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

d.clear()
     removes all items from d
```

14

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`

15

Oddities

```
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
```

16

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
```

17