

Programming Principles in Python (CSCI 503/490)

Dictionaries, Sets, & Comprehensions

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Function Scope

- The **scope** of a variable refers to where in a program it can be referenced
- Python has three scopes:
 - **global**: defined outside a function
 - **local**: in a function, only valid in the function
 - **nonlocal**: can be used with nested functions
- Python allows variables in different scopes to have the **same name**

Global keyword

- ```
def f(): # no arguments
 x = 2
 print("x inside:", x)
```

```
x = 1
f()
print("x outside:", x)
```

- Output:
  - x inside: 2
  - x outside: 1

- ```
def f(): # no arguments
    global x
    x = 2
    print("x inside:", x)
```

```
x = 1
f()
print("x outside:", x)
```

- Output:
 - x inside: 2
 - x outside: 2

Is Python pass-by-value or pass-by-reference?

Pass by Value or Pass by Reference?

- ```
def change(inner_list):
 inner_list = [9,8,7]
```

```
outer_list = [0,1,2]
change_list(outer_list)
outer_list # [0,1,2]
```

- Looks like pass by value!

- ```
def change(inner_list):  
    inner_list.append(3)
```

```
outer_list = [0,1,2]  
change_list(outer_list)  
outer_list # [0,1,2,3]
```

- Looks like pass by reference!

Pass by object reference

- AKA passing object references by value
- Python doesn't allocate space for a variable, it just links identifier to a value
- **Mutability** of the object determines whether other references see the change
- Any immutable object will act like pass by value
- Any mutable object acts like pass by reference unless it is reassigned to a new value

Don't use mutable values as defaults!

- ```
def append_to(element, to=[]):
 to.append(element)
 return to
```
- ```
my_list = append_to(12)  
my_list # [12]
```
- ```
my_other_list = append_to(42)
my_other_list # [12, 42]
```

[K. Reitz and T. Schlusser]

# Use None as a default instead

---

- ```
def append_to(element, to=None):  
    if to is None:  
        to = []  
    to.append(element)  
    return to
```
- ```
my_list = append_to(12)
my_list # [12]
```
- ```
my_other_list = append_to(42)  
my_other_list # [42]
```
- If you're not mutating, this isn't an issue

Dictionary

- AKA associative array or map
- Collection of key-value pairs
 - Keys are unique (repeats clobber existing)
 - Values need not be unique
- Syntax:
 - Curly brackets { } delineate start and end
 - Colons separate keys from values, commas separate pairs
 - `d = { 'DeKalb' : 783, 'Kane' : 134, 'Cook' : 1274, 'Will' : 546 }`
- No type constraints
 - `d = { 'abc' : 25, 12 : 'abc', ('Kane', 'IL') : 123.54 }`

Collections

- A dictionary is **not** a sequence
- Sequences are **ordered**
- Conceptually, dictionaries need no order
- A dictionary is a **collection**
- Sequences are also collections
- All collections have length (`len`), membership (`in`), and iteration (loop over values)
- Length for dictionaries counts number of key-value **pairs**
 - Pass dictionary to the `len` function
 - `d = { 'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54 }`
`len(d) # 3`

Assignment 3

- Pokémon Data
- Use dictionaries, lists, sets, and iteration
- Due Friday

Test 1

- Wednesday, Oct. 1, 9:30-10:45am
- In-Class, paper/pen & pencil
- Covers material through this week
- Format:
 - Multiple Choice
 - Free Response
 - One extra 2-sided page for CSCI 503 Students
- Info on the course webpage

Quiz Wednesday

Mutability

- Dictionaries are **mutable**, key-value pairs can be added, removed, updated
- (Each key must be immutable)
- Accessing elements parallels lists but with different "indices" possible
- Index → Key
- `d = { 'DeKalb': 783, 'Kane': 134, 'Cook': 1274, 'Will': 546 }`
- `d['Winnebago'] = 1023` # add a new key-value pair
- `d['Kane'] = 342` # update an existing key-value pair
- `d.pop('Will')` # remove an existing key-value pair
- `del d['Winnebago']` # remove an existing key-value pair

Key Restrictions

- Many types can be keys... including tuples
 - `d = { 'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54 }`
- ...but the type must be immutable—lists cannot be keys
 - ~~`d = { ['Kane', 'IL']: 2348.35, [1, 2, 3]: "apple" }`~~
- Why?

Key Restrictions

- Many types can be keys... including tuples
 - `d = { 'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54 }`
- ...but the type must be **immutable*** —lists cannot be keys
 - ~~`d = { ['Kane', 'IL']: 2348.35, [1, 2, 3]: "apple" }`~~
- *technically, the type must be hashable, but having a mutable and still hashable type almost always causes problems
- Why?
 - Dictionaries are fast in Python because are implemented as hash tables
 - No matter how long the key, python hashes it stores values by hash
 - Given a key to lookup, Python hashes it and finds the value quickly ($O(1)$)
 - If the key can mutate, the hash will not match the key!

Principle

- Be careful using floats for keys
- Why?

Principle

- Be careful using floats for keys
- `a = 0.123456`
`b = 0.567890`

```
values = [a, b, (a / b) * b, (b / a) * a]
found = {}
for d in values:
    found[d] = True
len(found) # 3 !!!
found.keys() # [0.123456, 0.56789, 0.12345599999999999998]
```

Accessing Values by Key

- To get a value, we start with a key
- Things work as expected
 - `d['Kane'] + d['Cook']`
- If a value does not exist, get `KeyError`
 - `d['Boone'] > 12 # KeyError`

Membership

- The membership operator (`in`) applies to **keys**
 - `'Boone' in d # False`
 - `'Cook' in d # True`
- To check the negation (if a key doesn't exist), use `not in`
 - `'Boone' not in d # True`
 - `not 'Boone' in d # True` (equivalent but less readable)
- Membership testing is much **faster** than for a list
- Checking and accessing at once
 - `d.get('Boone')` # no error, evaluates to `None`
 - `d.get('Boone', 0)` # no error, evaluates to `0` (default)

Updating multiple key-value pairs

- Update adds or replaces key-value pairs
- Update from another dictionary:
 - `d.update({'Winnebago': 1023, 'Kane': 324})`
- Update from a list of key-value tuples
 - `d.update([('Winnebago', 1023), ('Kane', 324)])`
- Update from keyword arguments
 - `d.update(Winnebago=1023, Kane=324)`
 - Only works for strings!
- Syntax for update also works for constructing a **new** dictionary
 - `d = dict([('Winnebago', 1023), ('Kane', 324)])`
 - `d = dict(Winnebago=1023, Kane=324)`

Dictionary Methods

Method	Meaning
<code><dict>.clear()</code>	Remove all key-value pairs
<code><dict>.update(other)</code>	Updates the dictionary with values from <code>other</code>
<code><dict>.pop(k, d=None)</code>	Removes the pair with key <code>k</code> and returns value or default <code>d</code> if no key
<code><dict>.get(k, d=None)</code>	Returns the value for the key <code>k</code> or default <code>d</code> if no key
<code><dict>.items()</code>	Returns iterable view over all pairs as (key, value) tuples
<code><dict>.keys()</code>	Returns iterable view over all keys
<code><dict>.values()</code>	Returns iterable view over all values

Dictionary Methods

Method	Meaning	Mutate
<code><dict>.clear()</code>	Remove all key-value pairs	
<code><dict>.update(other)</code>	Updates the dictionary with values from <code>other</code>	
<code><dict>.pop(k, d=None)</code>	Removes the pair with key <code>k</code> and returns value or default <code>d</code> if no key	
<code><dict>.get(k, d=None)</code>	Returns the value for the key <code>k</code> or default <code>d</code> if no key	
<code><dict>.items()</code>	Returns iterable view over all pairs as (key, value) tuples	
<code><dict>.keys()</code>	Returns iterable view over all keys	
<code><dict>.values()</code>	Returns iterable view over all values	

Iteration

- Even though dictionaries are not sequences, we can still iterate through them
- Principle: Don't depend on order
- ```
for k in d:
 print(k, end=" ")
```
- This only iterates through the **keys!**
- We could get the values:
- ```
for k in d:  
    print('key:', k, 'value:', d[k], end=" ")
```
- ...but this is kind of like counting through a sequence (not pythonic)

Dictionary Views

- `for k in d.keys():` `# iterate through keys`
 `print('key:', k)`
- `for v in d.values():` `# iterate through values`
 `print('value:', v)`
- `for k, v in d.items():` `# iterate through key-value pairs`
 `print('key:', k, 'value:', v)`
- `keys()` is superfluous but is a bit clearer
- `items()` is the enumerate-like method

Exercise: Count Letters

- Write code to take a string and return the count of each letter that occurs in a dictionary
- `count_letters('illinois')`
returns {'i': 3, 'l': 2, 'n': 1, 'o': 1, 's': 1}

Sorting

- Order doesn't really mean anything in a dictionary
- There is **not** a `.sort` or `.reverse` method
- We can iterate through items in sorted order using `sorted`
- ```
d = count_letters('illinois')
for k, v in sorted(d.items()):
 print(k, ': ', v)
```
- `reversed` also works on dictionary views
- `sorted` and `reversed` work on any iterable (thus all collections)

# Sets

# Sets

---

- Sets are dictionaries but without the values
- Same curly braces, no pairs
- `s = {'DeKalb', 'Kane', 'Cook', 'Will'}`
- Only one instance of a value is in a set—sets **eliminate duplicates**
- Adding multiple instances of the same value to a set doesn't do anything
- `s = {'DeKalb', 'DeKalb', 'DeKalb', 'Kane', 'Cook', 'Will'}`  
    `# {'Cook', 'DeKalb', 'Kane', 'Will'}`
- Watch out for the empty set
  - `s = {}` # not a set!
  - `s = set()` # an empty set

# Sets are Mutable Collections

---

- Sets are **mutable** like dictionaries: we can add, and delete
- Again, no type constraints
  - `s = {12, 'DeKalb', 22.34}`
- Like a dictionary, a set is a **collection** but not a sequence
- Q: What three things can we do for any collection?

# Collection Operations on Sets

---

- `s = { 'DeKalb', 'Kane', 'Cook', 'Will' }`
- Length
  - `len(s) # 4`
- Membership: fast just like dictionaries
  - `'Kane' in s # True`
  - `'Winnebago' not in s # True`
- Iteration
  - `for county in s:  
 print(county)`

# Mathematical Set Operations

---

- $s = \{ 'DeKalb', 'Kane', 'Cook', 'Will' \}$   
 $t = \{ 'DeKalb', 'Winnebago', 'Will' \}$
- Union:  $s \mid t \# \{ 'DeKalb', 'Kane', 'Cook', 'Will', 'Winnebago' \}$ 
  - Unlike dictionaries, is commutative for sets ( $s \mid t == t \mid s$ )
- Intersection:  $s \& t \# \{ 'DeKalb', 'Will' \}$
- Difference:  $s - t \# \{ 'Kane', 'Cook' \}$
- Symmetric Difference:  $s \wedge t \# \{ 'Kane', 'Cook', 'Winnebago' \}$
- Object method variants:  $s.union(t)$ ,  $s.intersection(t)$ ,  
 $s.difference(t)$ ,  $s.symmetric\_difference(t)$
- Disjoint:  $s.isdisjoint(t) \# False$

# Mutation Operations

---

- add: `s.add('Winnebago')`
- discard: `s.discard('Will')`
- remove: `s.remove('Will')` # generates `KeyError` if not exist
- clear: `s.clear()` # removes all elements
- Variants of the mathematical set operations (have augmented assignments)
  - update (union): `|=`
  - intersection\_update: `&=`
  - difference\_update: `-=`
  - symmetric\_difference\_update: `^=`
- Methods take any **iterable**, operators require **sets**