Programming Principles in Python (CSCI 503)

Dictionaries & Sets

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Functions

- def <function-name>(<parameter-names>):
  # do stuff
  return res

- Use return to return a value

- Can return more than one value using commas
  - def <function-name>(<parameter-names>):
    # do stuff
    return res1, res2

- Use simultaneous assignment when calling:
  - a, b = do_something(1, 2, 5)

- If there is no return value, the function returns None (a special value)
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**
Local Scope

• def f(): # no arguments
  x = 2
  print("x in function:", x)

  x = 1
  f()
  print("x in main:", x)

• Output:
  - x in function: 2
  x in main: 1

• Here, the x in f is in the local scope
Global Keyword for Global Scope

• def f(): # no arguments
global x
x = 2
print("x in function:", x)

x = 1
f()
print("x in main:", x)

• Output:
  - x in function: 2
  x in main: 2

• Here, the x in f is in the global scope because of the global declaration
Python as Pass-by-Value?

• def change_list(inner_list):
  inner_list = [10, 9, 8, 7, 6]

  outer_list = [0, 1, 2, 3, 4]
  change_list(outer_list)
  outer_list # [0, 1, 2, 3, 4]

• Looks like pass by value!
Python as Pass-by-Reference?

• def change_list(inner_list):
  inner_list.append(5)

  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4,5]

• Looks like pass by reference!
Python is 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
Default Parameter Values

- Can add \texttt{=\text{<value>}} to parameters
- \texttt{def rectangle\_area(width=30, height=20): return width * height}

- All of these work:
  - \texttt{rectangle\_area()} \# 600
  - \texttt{rectangle\_area(10)} \# 200
  - \texttt{rectangle\_area(10,50)} \# 500

- If the user does not pass an argument for that parameter, the parameter is set to the default value

- Cannot add non-default parameters after a defaulted parameter
  - \texttt{def rectangle\_area(width=30, height)}
Keyword Arguments

• Keyword arguments allow someone calling a function to specify exactly which values they wish to specify without specifying all the values
• This helps with long parameter lists where the caller wants to only change a few arguments from the defaults
• def f(alpha=3, beta=4, gamma=1, delta=7, epsilon=8, zeta=2, eta=0.3, theta=0.5, iota=0.24, kappa=0.134):
   # ...
• f(beta=12, iota=0.7)
Tuple Packing and Unpacking

- `def f(a, b):
  if a > 3:
    return a, b-a # tuple packing
  return a+b, b # tuple packing
- `c, d = f(4, 3) # tuple unpacking

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack
- Sometimes, check return value before unpacking:
  - `retval = f(42)
    if retval is not None:
      c, d = retval

D. Koop, CSCI 503/490, Fall 2021
Tuple Packing and Unpacking

- `def f(a, b):
  if a > 3:
    return a, b-a # tuple packing
  return a+b, b # tuple packing
  t = (a, b-a)
  return t`

- `c, d = f(4, 3) # tuple unpacking`

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:
  - `retval = f(42)
    if retval is not None:
      c, d = retval`
Tuple Packing and Unpacking

- `def f(a, b):`
  - `if a > 3:`
    - `return a, b-a` # tuple packing
  - `return a+b, b` # tuple packing
- `c, d = f(4, 3)` # tuple unpacking

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a` # ValueError: too many values to unpack
- Sometimes, check return value before unpacking:
  - `retval = f(42)`
    - `if retval is not None:`
      - `c, d = retval`
enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`
- `for i, d in enumerate(my_list):
  print("index:", i, "element:", d)`
- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!
- `for t in enumerate(my_list):
  i = t[0]
  d = t[1]
  print("index:", i, "element:", d)"
enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`

```python
for i, d in enumerate(my_list):
    print("index:", i, "element:", d)
```

- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!

```python
for t in enumerate(my_list):
    i = t[0]
    d = t[1]
    print("index:", i, "element:", d)
```
Other sequence methods

• my_list = [7, 2, 1, 12]

• Math methods:
  - max(my_list) # 12
  - min(my_list) # 1
  - sum(my_list) # 22

• zip: combine two sequences into a single sequence of tuples
  - zip_list = list(zip(my_list, "abcd"))
    zip_list # [(1, 'a'), (2, 'b'), (7, 'c'), (12, 'd')]
  - Use this instead of using indices to count through both
Assignment 3

• Coming soon…
Dictionaries
Dictionary

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

<table>
<thead>
<tr>
<th>Keys</th>
<th>Key type</th>
<th>Values</th>
<th>Value type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country names</td>
<td>str</td>
<td>Internet country</td>
<td>str</td>
</tr>
<tr>
<td>Decimal numbers</td>
<td>int</td>
<td>Roman numerals</td>
<td>str</td>
</tr>
<tr>
<td>States</td>
<td>str</td>
<td>Agricultural</td>
<td>list of str</td>
</tr>
<tr>
<td>Hospital patients</td>
<td>str</td>
<td>Vital signs</td>
<td>tuple of floats</td>
</tr>
<tr>
<td>Baseball players</td>
<td>str</td>
<td>Batting averages</td>
<td>float</td>
</tr>
<tr>
<td>Metric</td>
<td>str</td>
<td>Abbreviations</td>
<td>str</td>
</tr>
<tr>
<td>Inventory codes</td>
<td>str</td>
<td>Quantity in stock</td>
<td>int</td>
</tr>
</tbody>
</table>
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
  ```python
  d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}
  len(d) # 3
  ```
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

```python
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 new hash will not match the original key’s hash!
Principle

- Be careful using floats for keys
- Why?
Principle

• Be careful using floats for keys

• \( a = 0.123456 \)
  \( b = 0.567890 \)

```python
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, 0.12345599999999998]
```
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:
  - \( d \cdot \text{update} \left( \{ \text{'Winnebago': 1023, 'Kane': 324} \} \right) \)

• Update from a list of key-value tuples
  - \( d \cdot \text{update} \left( \left[ (\text{'Winnebago', 1023}), (\text{'Kane', 324}) \right] \right) \)

• Update from keyword arguments
  - \( d \cdot \text{update} \left( \text{Winnebago=1023, Kane=324} \right) \)
    - Only works for strings!

• Syntax for update also works for constructing a **new** dictionary
  - \( d = \text{dict} \left( \left[ (\text{'Winnebago', 1023}), (\text{'Kane', 324}) \right] \right) \)
  - \( d = \text{dict} \left( \text{Winnebago=1023, Kane=324} \right) \)
What about merging?

- \( d = \{ 'DeKalb': 783, 'Kane': 134, 'Cook': 1274, 'Will': 546 \} \)
  \( e = \{ 'Winnebago': 1023, 'Kane': 324 \} \)

- Can update \( d \) in place, but more complicated if want a **new** dictionary
  - \( f = d.copy() \)
    
    \( f.update(e) \)
  - \( f = \{ **d, **e \} # \) a bit esoteric

- Python 3.9: union operator (**PEP 584**)
  - \( f = d | e \)
  - \( e \)'s values will overwrite \( d \)'s values (like update)
  - so not commutative (\( d | e \) != \( e | d \))
### Dictionary Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dict&gt;.clear()</code></td>
<td>Remove all key-value pairs</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.update(other)</code></td>
<td>Updates the dictionary with values from <code>other</code></td>
</tr>
<tr>
<td><code>&lt;dict&gt;.pop(k, d=None)</code></td>
<td>Removes the pair with key <code>k</code> and returns value or default <code>d</code> if no key</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.get(k, d=None)</code></td>
<td>Returns the value for the key <code>k</code> or default <code>d</code> if no key</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.items()</code></td>
<td>Returns iterable view over all pairs as (key, value) tuples</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.keys()</code></td>
<td>Returns iterable view over all keys</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.values()</code></td>
<td>Returns iterable view over all values</td>
</tr>
</tbody>
</table>
## Dictionary Methods

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

- Even though dictionaries are not sequences, we can still iterate through them
- Principle: Don't depend on order

```python
for k in d:
    print(k, end=" ")
```

- This only iterates through the keys!
- We could get the values:

```python
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 the number of each letter that occurs in a dictionary
• count_letters('illinois') # returns {'i': 3, 'l': 2, 'n': 1, 'o': 1, 's': 1}
Exercise: Count Letters

- `def count_letters(s):
  d = {}
  for c in s:
    if c not in d:
      d[c] = 1
    else:
      d[c] += 1
  return d
count_letters('illinois')`
Exercise: Count Letters

```python
• def count_letters(s):
    d = {}
    for c in s:
        d[c] = d.get(c, 0) + 1
    return d

count_letters('illinois')
```
Exercise: Count Letters (using collections)
Exercise: Count Letters (using collections)

- from collections import defaultdict
def count_letters(s):
    d = defaultdict(int)
    for c in s:
        d[c] += 1
    return d
count_letters('illinois')
Exercise: Count Letters (using collections)

• from collections import defaultdict
def count_letters(s):
    d = defaultdict(int)
    for c in s:
        d[c] += 1
    return d
count_letters('illinois')

• from collections import Counter
def count_letters(s):
    return Counter(s)
count_letters('illinois')
Sorting

- Order doesn't really mean anything in a dictionary
- There is no .sort or .reverse method
- We can iterate through items in sorted order using sorted
- `d = count_letters('illinois')`
  ```python
  for k, v in sorted(d.items()):
    print(k, ':
  ```
- reversed also works on dictionary views
- sorted and reversed work on any iterable (thus all collections)