Programming Principles in Python (CSCI 503/490)

Dictionaries

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Updating collections

• There are three ways to deal with operations that update collections:
  - Returns an **updated copy** of the collection
  - Updates the collection **in place**
  - Updates the collection in place **and returns it**

• `list.sort` and `list.reverse` work **in place** and **don't return** it

• `sorted` and `reversed` return an **updated copy**
  - `reversed` actually returns an iterator
  - these also work for immutable sequences like strings and tuples
**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, Spring 2023
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`
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, Spring 2023
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 = [9, 8, 7]
  outer_list = [0, 1, 2]
  change_list(outer_list)
  outer_list # [0, 1, 2]

• 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
Assignment 2

- Due Thursday
- Python control flow and functions
- Do not use containers like lists!
- Compute orbit and number of steps for mathematical sequences
- Make sure to follow instructions
  - Name the submitted file a2.ipynb
  - Put your name and z-id in the first cell
  - Label each part of the assignment using markdown
  - Make sure to produce output according to specifications
Remember: global allows assignment in functions

- def change_list():
  global a_list
  a_list = [10,9,8,7,6]

  a_list = [0,1,2,3,4]
  change_list()
  a_list # [10,9,8,7,6]
Default Parameter Values

- Can add =<value> to parameters
- `def rectangle_area(width=30, height=20):
  return width * height`

- All of these work:
  - `rectangle_area()` # 600
  - `rectangle_area(10)` # 200
  - `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
  - `def rectangle_area(width=30, height)`
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]
Use None as a default instead

- `def append_to(element, to=None):`
  
  ```python
  if to is None:
      to = []
  to.append(element)
  return to
  ```

- `my_list = append_to(12)`
  ```python
  my_list # [12]
  ```

- `my_other_list = append_to(42)`
  ```python
  my_other_list # [42]
  ```

- If you're not mutating, this isn't an issue
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)
```
Positional & Keyword Arguments

- Generally, any argument can be passed as a keyword argument
- ```
def f(alpha, beta, gamma=1, delta=7, epsilon=8, zeta=2, eta=0.3, theta=0.5, iota=0.24, kappa=0.134):
    # ...
```
- `f(5, 6)`
- `f(alpha=7, beta=12, iota=0.7)`
Position-Only Arguments

• **PEP 570** introduced position-only arguments
• Sometimes it makes sense that certain arguments must be position-only
• Certain functions (those implemented in C) only allow position-only: `pow`
• Add a slash (`/`) to delineate where keyword arguments start

```python
def f(alpha, beta, /, gamma=1, delta=7, epsilon=8, zeta=2, eta=0.3, theta=0.5, iota=0.24, kappa=0.134):
    # ...
    - f(alpha=7, beta=12, iota=0.7) # ERROR
    - f(7, 12, iota=0.7) # WORKS
```
Arbitrary Argument Containers

• `def f(*args, **kwargs):`
  # ...

• `args`: a list of arguments
• `kwargs`: a key-value dictionary of arguments
• Stars in function signature, not in use
• Can have named arguments before these arbitrary containers
• Any values set by position will not be in `kwargs`:

• `def f(a, *args, **kwargs):
   print(args)
   print(kwargs)
   f(a=3, b=5) # args is empty, kwargs has only b`
Programming Principles: Defining Functions

• List arguments in an order that makes sense
  - May be convention $\Rightarrow \text{pow}(x, y)$ means $x^y$
  - May be in order of expected frequency used

• Use default parameters when meaningful defaults are known

• Use position-only arguments when there is no meaningful name or the syntax might change in the future
Calling module functions

- Some functions exist in modules (we will discuss these more later)
- Import module
- Call functions by prepending the module name plus a dot

```python
import math
math.log10(100)
math.sqrt(196)
```
Calling object methods

• Some functions are defined for objects like strings
• These are **instance methods**
• Call these using a similar dot-notation
• Can take arguments

• \( s = 'Mary' \)
  \( s.upper() \) # 'MARY'

• \( t = ' extra spaces ' \)
  \( t.strip() \) # 'extra spaces'

• \( u = '1+2+3+4' \)
  \( u.split(sep='+') \) # ['1','2','3','4']
Dictionaries
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}`
# 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
  - `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

• \[d = \{\text{'DeKalb': 783, 'Kane': 134, 'Cook': 1274, 'Will': 546}\}\]
• \[d[\text{'Winnebago'}] = 1023\] # add a new key-value pair
• \[d[\text{'Kane'}] = 342\] # update an existing key-value pair
• \[d.pop(\text{'Will'})\] # remove an existing key-value pair
• \[\text{del } d[\text{'Winnebago'}]\] # 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

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

```python
calculated = [a, b, (a / b) * b, (b / a) * a]
found = {}
for d in calculated:
    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.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

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