Programming Principles in Python (CSCI 503/490)

Comprehensions, Generators, and Lazy Evaluation

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Quiz

Question 1

- How does Python pass arguments to functions?
  (a) pass by key
  (b) pass by reference
  (c) pass by object reference
  (d) pass by value

Question 2

- If `d = {'a': 12,'b': 12,'c': 12,'a': 12}` what is `len(d)`?

  (a) 3

  (b) 1

  (c) 4

  (d) 2

Question 3

- Which of the following is **not** a valid operation on a dictionary?
  (a) slicing
  (b) iteration
  (c) membership
  (d) None of the above

Question 4

- Which expression evaluates to `"abcabc"`?
  (a) `"abc" + sorted("cba")`
  (b) `"abc" - "abc"`
  (c) `"abc" * 2`
  (d) `"abc + "abc"[::-1]`

Question 5

- Given the function signature `def f(a, b, c=7)`, which of the following expressions runs without an error?
  (a) `f(3,4,d=9)`
  (b) `f(b=5,a=1)`
  (c) `f()`
  (d) `f(a=5)`

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 = {'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
    ```
    ```

Mutability

- Dictionaries are **mutable**, key-value pairs can be added, removed, updated
- `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`
- `d.update({'Winnebago': 1023, 'Kane': 324})`
- `d.update([('Winnebago', 1023), ('Kane', 324)])`
- `d.update(Winnebago=1023, Kane=324)`

Dictionary Methods

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

Dictionary Methods

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

Dictionary Iteration

- `for k in d.keys():  # iterate through keys`
  `    print('key:', k)`
- `for k in d:  # iterates 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

Sets & Operations

- `s = {'DeKalb', 'Kane', 'Cook', 'Will'}`
  `t = {'DeKalb', 'Winnebago', 'Will'}`
- Union: `s | t # {'DeKalb', 'Kane', 'Cook', 'Will', 'Winnebago'}`
- Intersection: `s & t # {'DeKalb', 'Will'}`
- Difference: `s - t # {'Kane', 'Cook'}`
- Symmetric Difference: `s ^ t # {'Kane', 'Cook', 'Winnebago'}`
- Object method variants: `s.union(t)`, `s.intersection(t)`, `s.difference(t)`, `s.symmetric_difference(t)`
- `*_update` and augmented operator variants

Set 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**

Assignment 3

- Use dictionaries, lists, and iteration to analyze foods and their ingredients
- Due next Monday

Test 1

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

Comprehensions

Comprehension

- Shortcut for loops that **transform** or **filter** collections
- Functional programming features this way of thinking:
  Pass functions to functions!
- Imperative: a loop with the actual functionality buried inside
- Functional: specify both functionality and data as inputs

List Comprehension

- ```
  output = []
  for d in range(5):
      output.append(d ** 2 - 1)
  ```

- Rewrite as a map:
  - `output = [d ** 2 - 1 for d in range(5)]`
- Can also filter:
  - `output = [d for d in range(5) if d % 2 == 1]`
- Combine map & filter:
  - `output = [d ** 2 - 1 for d in range(5) if d % 2 == 1]`
  ```

Comprehensions using other collections

- Comprehensions can use existing collections, too (not just ranges)
- Anything that is **iterable** can be used in the for construct (like for loop)
- `names = ['smith', 'Smith', 'John', 'mary', 'jan']`
- `names2 = [item.upper() for item in names]`

Any expression works as output items

- Tuples inside of comprehension
  - `[(s, s+2) for s in slist]`
- Dictionaries, too
  - `[{'i': i, 'j': j} for (i, j) in tuple_list]`
- Function calls
  - ```
    names = ['smith', 'Smith', 'John', 'mary', 'jan']
    names2 = [item.upper() for item in names]
    ```
    ```

Multi-Level and Nested Comprehensions

- **Flattening** a list of lists
  - `my_list = [[1,2,3],[4,5],[6,7,8,9,10]]`
    `[v for vlist in my_list for v in vlist]`
  - `[1,2,3,4,5,6,7,8,9,10]`
- Note that the for loops are in order
- Difference between **nested** comprehensions
  - `[[v**2 for v in vlist] for vlist in my_list]`
  - `[[1,4,9],[16,25],[36,49,64,81,100]]`

Comprehensions for other collections

- Dictionaries
  - `{k: v for (k, v) in other_dict.items() if k.startswith('a')}`
  - Sometimes used for one-to-one map inverses
    - How?

Comprehensions for other collections

- Dictionaries
  - `{k: v for (k, v) in other_dict.items() if k.startswith('a')}`
  - Sometimes used for one-to-one map inverses
    - `{v: k for (k, v) in other_dict.items()}`
    - Be careful that the dictionary is actually one-to-one!
- Sets:
  - `{s[0] for s in names}`

Tuple Comprehension?

- ```
  thing = (x ** 2 for x in numbers if x % 2 != 0)
  thing # not a tuple! <generator object <genexpr> …>
  ```
- Actually a **generator**!
- This **delays** execution until we actually need each result
  ```

Iterators

- Key concept: iterators only need to have a way to get the next element
- To be **iterable**, an object must be able to **produce** an iterator
  - Technically, must implement the `__iter__` method
- An iterator must have two things:
  - a method to get the **next item**
  - a way to signal **no more** elements
- In Python, an **iterator** is an object that must
  - have a defined `__next__` method
  - raise `StopException` if no more elements available

Iteration Methods

- You can call iteration methods directly, but rarely done

```
- my_list = [2,3,5,7,11]
  it = iter(my_list)
  first = next(it)
  print("First element of list:", first)
```

- `iter` asks for the iterator from the object
- `next` asks for the next element
- Usually just handled by loops, comprehensions, or generators

For Loop and Iteration

- ```
  my_list = [2,3,5,7,11]
  for i in my_list:
      print(i * i)
  ```
- Behind the scenes, the for construct
  - asks for an iterator `it = iter(my_list)`
  - calls `next(it)` each time through the loop and assigns result to i
  - handles the `StopIteration` exception by ending the loop
- Loop won't work if we don't have an iterable!
  - ```
    for i in 7892:
        print(i * i)
    ```

Generators

- Special functions that return **lazy** iterables
- Use less memory
- Change is that functions `yield` instead of `return`
- ```
  def square(it):
      for i in it:
          yield i*i
  ```
- If we are iterating through a generator, we hit the first yield and immediately return that first computation
- Generator expressions just shorthand (remember no tuple comprehensions)
  - `(i * i for i in [1,2,3,4,5])`
  ```

Generators

- If memory is not an issue, a comprehension is probably faster
- …unless we don't use all the items
- ```
  def square(it):
      for i in it:
          yield i*i
  ```
- ```
  for j in square([1,2,3,4,5]):
      if j >= 9:
          break
      print(j)
  ```
- The square function only runs the computation for 1, 2, and 3
- What if this computation is **slow**?

Lazy Evaluation

- ```
  u = compute_fast_function(s, t)
  v = compute_slow_function(s, t)
  if s > t and s**2 + t**2 > 100:
      return u / 100
  else:
      return v / 100
  ```
- We don't write code like this! Why?
  ```

Lazy Evaluation

- ```
  u = compute_fast_function(s, t)
  v = compute_slow_function(s, t)
  if s > t and s**2 + t**2 > 100:
      return u / 100
  else:
      return v / 100
  ```
- We don't write code like this! Why?
- Don't compute values until you need to!
  ```

Lazy Evaluation

- Rewriting
- ```
  if s > t and s**2 + t**2 > 100:
      u = compute_fast_function(s, t)
      res = u / 100
  else:
      v = compute_slow_function(s, t)
      res = v / 100
  ```
- slow function will not be executed unless the condition is true
  ```

Lazy Evaluation

- What if this were rewritten as:
- ```
  def my_function(s, t, u, v):
      if s > t and s**2 + t**2 > 100:
          res = u
      else:
          res = v
      return res
  my_function(s, t, compute_fast_function(s, t),
                    compute_slow_function(s, t))
  ```
- In some languages (often pure functional languages), computation of `u` and `v` may be **deferred** until we need them
- Python doesn't work that way in this case
  ```

Short-Circuit Evaluation

- But Python, and many other languages, do work this way for **boolean** operations
- ```
  if b != 0 and a/b > c:
      return ratio - c
  ```
- Never get a divide by zero error!
- Compare with:
- ```
  def check_ratio(val, ratio, cutoff):
      if val != 0 and ratio > cutoff:
          return ratio - cutoff
  check_ratio(b, a/b, c)
  ```
- Here. `a/b` is computed before `check_ratio` is called (but **not used**!)

Short-Circuit Evaluation

- Works from left to right according to order of operations (and before or)
- Works for `and` and `or`
- and:
  - if **any** value is `False`, stop and return `False`
  -
```
a, b = 2, 3
a > 3 and b < 5
```
- or:
  - if **any** value is `True`, stop and return `True`
  -
```
a, b, c = 2, 3, 7
a > 3 or b < 5 or c > 8
```