Programming Principles in Python (CSCI 503)

Comprehensions, Iterators, and Generators

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

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
  ```

```
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
  ```

```
t = (a, b-a)
return t
```

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

```
t = f(4, 3)
(c, d) = t
```

- 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
    ```

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

Iteration

- Even though dictionaries are not sequences, we can still iterate through them
- Principle: Don't depend on order
- ```
  for k in d:              # iterate through keys
      print(k, end=" ")
  ```
- ```
  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)
  ```

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`
  ```

Assignment 3

- Pokémon Data
- Lots of iteration and dictionary access
- Also create new lists and dictionaries

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, replace, 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 | 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)`
- 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**

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
    - ```
      colors = ['smith', 'Smith', 'John', 'mary', 'jan']
      colors2 = [item.upper() for item in colors]
      ```
      ```

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(c)`
  - calls `next(it)` each time through the loop
  - 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
    ```

Short-Circuit Evaluation

- Back to our example
- ```
  if s > t and compute_slow_function(s, t) > 50:
      c = compute_slow_function(s, t)
  else:
      c = compute_fast_function(s, t)
  ```
- `s, t = 10, 12 # compute_slow_function is never run`
- `s, t = 5, 4   # compute_slow_function is run once`
- `s, t = 12, 10 # compute_slow_function is run twice`
  ```

Short-Circuit Evaluation

- Walrus operator saves us one computation
- ```
  if s > t and (c := compute_slow_function(s, t) > 50):
      pass
  else:
      c = s ** 2 + t ** 2
  ```
- `s, t = 10, 12 # compute_slow_function is never run`
- `s, t = 5, 4   # compute_slow_function is run once`
- `s, t = 12, 10 # compute_slow_function is run once`
  ```

What about multiple executions?

- ```
  for s, t in [(12, 10), (4, 5), (5, 4), (12, 10)]:
      if s > t and (c := compute_slow_function(s, t) > 50):
          pass
      else:
          c = compute_fast_function(s, t)
  ```
- What's the problem here?
  ```

What about multiple executions?

- ```
  for s, t in [(12, 10), (4, 5), (5, 4), (12, 10)]:
      if s > t and (c := compute_slow_function(s, t) > 50):
          pass
      else:
          c = compute_fast_function(s, t)
  ```
- What's the problem here?
- Executing the function for the same inputs twice!
  ```

Memoization

- ```
  memo_dict = {}
  def memoized_slow_function(s, t):
      if (s, t) not in memo_dict:
         memo_dict[(s, t)] = compute_slow_function(s, t)
      return memo_dict[(s, t)]
  ```
- ```
  for s, t in [(12, 10), (4, 5), (5, 4), (12, 10)]:
      if s > t and (c := memoized_slow_function(s, t) > 50):
          pass
      else:
          c = compute_fast_function(s, t)
  ```
- Second time executing for `s=12`, `t=10`, we don't need to compute!
- Tradeoff memory for compute time

Memoization

- Heavily used in functional languages because there is no assignment
- Cache (store) the results of a function call so that if called again, returns the result without having to compute
- If arguments of a function are **hashable**, fairly straightforward to do this for any Python function by caching in a dictionary
- In what contexts, might this be a bad idea?

Memoization

- Heavily used in functional languages because there is no assignment
- **Cache** (store) the results of a function call so that if called again, returns the result without having to compute
- If arguments of a function are **hashable**, fairly straightforward to do this for any Python function by caching in a dictionary
- In what contexts, might this be a bad idea?
  - ```
    def memoize_random_int(a, b):
        if (a,b) not in random_cache:
            random_cache[(a,b)] = random.randint(a,b)
        return random_cache[(a,b)]
    ```
  - When we want to rerun, e.g. random number generators
    ```