Programming Principles in Python (CSCI 503/490)

Lazy Evaluation

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Quiz

Question 1

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

Question 2

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

Question 3

- Which of the following is a valid **comprehension**?
  - (a) `(d * 2 for d in range(10) if d % 2 == 0)`
  - (b) `[d * 2 if d % 2 == 0 for d in range(10)]`
  - (c) `(d * 2 if d % 2 == 0 for d in range(10))`
  - (d) `{d * 2 for d in range(10) if d % 2 == 0}`

Question 4

- Given the function signature `def f(a, b=3, c=7)`, which of the following expressions runs without an error?
  (a) `f(b=5,c=1)`
  (b) `f()`
  (c) `f(a=5)`
  (d) `f(3,4,9,2)`

Question 5

- If `d = {'a': 12,'b': 13,'b': 27 ,'a': 34}` what is `len(d)`?
  (a) 4
  (b) 2
  (c) 8
  (d) 3

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

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 for other collections

- Dictionaries
  - `{k: v for (k, v) in other_dict.items() if k.startswith('a')}`
  - Example: 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}`
- Tuples? Not exactly
  - `(s[0] for s in names)`
  - Not a tuple, a **generator expression**

Assignment 3

- Use dictionaries, lists, and sets to analyze US Senate Stock Trades
- Due Monday

Test 1

- Wednesday, Feb. 21, 12:30-1:45pm
- In-Class, paper/pen & pencil
- Covers material through this week
- Info is posted on the course webpage

Next Monday

- No in-person lecture, no in-person office hours
- Will publish video lecture on strings
- Email questions about test

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
    ```

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
    ```

Functional Programming

- Programming without imperative statements like assignment
- In addition to comprehensions & iterators, have functions:
  - map: iterable of n values to an iterable of n transformed values
  - filter: iterable of n values to an iterable of m (m <= n) values
- Eliminates need for concrete looping constructs

Map

- Generator function (lazy evaluation)
- First argument is a **function**, second argument is the **iterable**
- ```
  def upper(s):
      return s.upper()
  ```
- `map(upper, ['sentence', 'fragment']) # generator`
- Similar comprehension:
  - `[upper(s) for s in ['sentence', 'fragment']] # comprehension`

- This only calls `upper` **once**
- ```
  for word in map(upper, ['sentence', 'fragment']):
      if word == "SENTENCE":
          break
  ```

Filter

- Also a generator
- ```
  def is_even(x):
      return (x % 2) == 0
  ```
- `filter(is_even, range(10)) # generator`
- Similar comprehension:
  - `[d for d in range(10) if is_even(d)] # comprehension`
  ```

Lambda Functions

- ```
  def is_even(x):
      return (x % 2) == 0
  ```
- `filter(is_even, range(10) # generator`
- Lots of code to write a simple check
- Lambda functions allow inline function definition
- Usually used for "one-liners": a simple data transform/expression
- `filter(lambda x: x % 2 == 0, range(10))`
- Parameters follow `lambda`, **no parentheses**
- **No** `return` keyword as this is implicit in the syntax
- JavaScript has similar functionality (arrow functions): `(d => d % 2 == 0)`
  ```