Programming Principles in Python (CSCI 503/490)

Lazy Evaluation & Strings

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

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

- USDA Food Data
- Looking at branded data and nutrition information
- Start with the sample notebook (or copy its code) to download the data
- Data is a list of dictionaries
- Need to iterate through, update, and create new lists & dictionaries
- Part 6 is CSCI 503 students Only, but CSCI 490 students may complete for extra credit

Test 1

- Next Wednesday, Feb. 23
- In-class, 2:00-3:15pm in PM 153
- Format:
  - Multiple Choice
  - Free Response
- Information at the link above

Example

- Suppose I want to write Python code to print the numbers from 1 to 100. What errors do you see? How could you improve the code?

```
// print the numbers from 1 to 100
int counter = 1
while counter < 100 {
    print counter
    counter++
}
```

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)`
  ```

Strings

Strings

- Remember strings are sequences of characters
- Strings are collections so have `len`, `in`, and iteration
  - ```
    s = "Huskies"
    len(s); "usk" in s; [c for c in s if c == 's']
    ```
- Strings are sequences so have
  - indexing and slicing: `s[0], s[1:]`
  - concatenation and repetition: `s + " at NIU"; s * 2`
- Single or double quotes `'string1'`, `"string2"`
- Triple double-quotes: `"""A string over many lines"""`
- Escaped characters: `'\n'` (newline) `'\t'` (tab)
    ```

Unicode and ASCII

- Conceptual systems
- ASCII:
  - old 7-bit system (only 128 characters)
  - English-centric
- Unicode:
  - modern system
  - Can represent over 1 million characters from all languages + emoji 🎉
  - Characters have hexadecimal representation: é = U+00E9 and name (LATIN SMALL LETTER E WITH ACUTE)
  - Python allows you to type `"é"` or represent via code `"\u00e9"`

Unicode and ASCII

- Encoding: How things are actually stored
- ASCII "Extensions": how to represent characters for different languages
  - No universal extension for 256 characters (one byte), so…
  - ISO-8859-1, ISO-8859-2, CP-1252, etc.
- Unicode encoding:
  - UTF-8: used in Python and elsewhere (uses variable # of 1—4 bytes)
  - Also UTF-16 (2 or 4 bytes) and UTF-32 (4 bytes for everything)
  - Byte Order Mark (BOM) for files to indicate endianness (which byte first)

Strings are Objects with Methods

- We can call methods on strings like we can with lists
  - ```
    s = "Peter Piper picked a peck of pickled peppers"
    s.count('p')
    ```
- Doesn't matter if we have a variable or a literal
  - ```
    "Peter Piper picked a peck of pickled peppers".find("pick")
    ```

Finding & Counting Substrings

- `s.count(sub)`: Count the number of occurrences of `sub` in `s`
- `s.find(sub)`: Find the first position where `sub` occurs in `s`, else `-1`
- `s.rfind(sub)`: Like `find`, but returns the right-most position
- `s.index(sub)`: Like `find`, but raises a ValueError if not found
- `s.rindex(sub)`: Like `index`, but returns right-most position
- `sub in s`: Returns `True` if `s` contains `sub`
- `s.startswith(sub)`: Returns `True` if `s` starts with `sub`
- `s.endswith(sub)`: Returns `True` if `s` ends with `sub`

Removing Leading and Trailing Strings

- `s.strip()`: Copy of `s` with leading and trailing whitespace removed
- `s.lstrip()`: Copy of `s` with leading whitespace removed
- `s.rstrip()`: Copy of `s` with trailing whitespace removed
- `s.removeprefix(prefix)`: Copy of `s` with `prefix` removed (if it exists)
- `s.removesuffix(suffix)`: Copy of `s` with `suffix` removed (if it exists)

Transforming Text

- `s.replace(oldsub, newsub)`:
  Copy of `s` with occurrences of `oldsub` in `s` with `newsub`
- `s.upper()`: Copy of `s` with all uppercase characters
- `s.lower()`: Copy of `s` with all lowercase characters
- `s.capitalize()`: Copy of `s` with first character capitalized
- `s.title()`: Copy of `s` with first character of each word capitalized

Checking String Composition

String Method	Description
`isalnum()`	Returns True if the string contains only alphanumeric characters (i.e., digits & letters).
`isalpha()`	Returns True if the string contains only alphabetic characters (i.e., letters).
`isdecimal()`	Returns True if the string contains only decimal integer characters
`isdigit()`	Returns True if the string contains only digits (e.g., '0', '1', '2').
`isidentifier()`	Returns True if the string represents a valid identifier.
`islower()`	Returns True if all alphabetic characters in the string are lowercase characters
`isnumeric()`	Returns True if the characters in the string represent a numeric value w/o a + or - or .
`isspace()`	Returns True if the string contains only whitespace characters.
`istitle()`	Returns True if the first character of each word is the only uppercase character in it.
`isupper()`	Returns True if all alphabetic characters in the string are uppercase characters

[Deitel & Deitel]

Splitting

- `s = "Venkata, Ranjit, Pankaj, Ali, Karthika"`
- `names = s.split(',') # names is a list`
- `names = s.split(',', 3) # split by commas, split <= 3 times`
- separator may be multiple characters
- if no separator is supplied (`sep=None`), runs of consecutive whitespace delimit elements
- `rsplit` works in reverse, from the right of the string
- `partition` and `rpartition` for a single split with before, sep, and after
- `splitlines` splits at line boundaries, optional parameter to keep endings

Joining

- `join` is a method on the **separator** used to join a list of strings
- `','.join(names)`
  - `names` is a list of strings, `','` is the separator used to join them
- Example:
  - ```
    def orbit(n):
        # ...
        return orbit_as_list
    print(','.join(orbit_as_list))
    ```
    ```