

Programming Principles in Python (CSCI 503/490)

Comprehensions, Generators, and Lazy Evaluation

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Quiz

Quiz

1. If $d = \{ 'a': 1, 'b': 2, 'a': 3 \}$ and $e = \{ 'b': 4, 'c': 5 \}$, what is $d \mid e$?

- (a) $\{ 'a': 1, 'b': 2, 'c': 5 \}$
- (b) $\{ 'a': 3, 'b': 4, 'c': 5 \}$
- (c) $\{ 'a': 1, 'b': 4, 'c': 5 \}$
- (d) $\{ 'a': 1, 'b': 2, 'a': 3, 'b': 4, 'c': 5 \}$

Quiz

2. Given a list `a = ['a', 'b', 'c', 'd']`, which expression evaluates to `['b', 'c', 'd']`?

- (a) `a[2:4]`
- (b) `a[-2:]`
- (c) `a[1:3]`
- (d) `a[-3:99]`

Quiz

3. Given the function signature `def f(a, b, c=7)`, which of the following expressions does not produce an error?

- (a) `f(1, 3, 5)`
- (b) `f(a=3, c=1)`
- (c) `f(c=6)`
- (d) `f(b=3, 4)`

Quiz

4. Given `mylist = [3, 1, 2, 5, 4]`, what does `mylist.sort(reverse=True)` evaluate to?
- (a) `[1, 2, 3, 4, 5]`
 - (b) `[4, 5, 2, 1, 3]`
 - (c) `[5, 4, 3, 2, 1]`
 - (d) `None`

Quiz

5. How does Python pass arguments to functions?

- (a) By key
- (b) By reference
- (c) By value
- (d) By object reference

Dictionary Methods

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

Dictionary Methods

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

CSAN Panel: Real Jobs in the Real World



NIU

**COMPUTER SCIENCE
ALUMNI NETWORK**

**REAL
JOBS IN
THE REAL
WORLD**

A Panel
Discussion

TUESDAY, OCT. 7, 2025
Barsema Alumni & Visitors Center (Ballroom)
5:30–7:30 p.m.

- Tuesday, Oct. 7, 5:30–7:30pm
- Provides an insight into jobs from NIU alumni
- Food is Provided
- Sponsored by the Computer Science Alumni Network and the NIU Alumni Association

Assignment 3

- Pokémon Data
- Use dictionaries, lists, sets, and iteration
- Due Friday

Test 1

- Wednesday, Oct. 1, 9:30-10:45am
- In-Class, paper/pen & pencil
- Covers material through this week
- Format:
 - Multiple Choice
 - Free Response
 - One extra 2-sided 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]`

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 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]`

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`

# 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