ECE 364
Software Engineering Tools Laboratory

Lecture 5
Python: Collections II

Lecture Summary

- Iterators
- Collection-Related Functions.
- List Comprehension

- Covered in lecture (but not in these slides)
  - range(…)
  - generators
  - generator functions
  - generator expressions
  - list comprehensions
  - set comprehensions
  - dict comprehensions

Covered in lecture (1 of 2)

- range(…)
  - Returns an iterable, not a list
- generators
  - Generators are an alternative to collection types
  - Avoid loading all data into memory at once
  - A *generator* are a kind of *iterable*
- generator functions
  - Use a `yield` statement, and no `return` statement
  - Control flow goes back and forth between caller and generator function

Covered in lecture (2 of 2)

- generator expressions
  - `(▒ for ▒ in ▒)`
  - `(▒ for ▒ in ▒ if ▒)`
- list comprehensions
  - `[▒ for ▒ in ▒]`
  - `[▒ for ▒ in ▒ if ▒]`
- set comprehensions
  - `{▒ for ▒ in ▒}`
  - `{▒ for ▒ in ▒ if ▒}`
- dict comprehensions
  - `{▒:▒ for ▒ in ▒}`
  - `{▒:▒ for ▒ in ▒ if ▒}`

Iterators

- Iteration is the process of visiting, or working with, one element at a time in a sequence of elements.
- Iteration is a key process in all programming language, and Python is no different.
- An Iterable, is a data structure that contains a sequence, or a collection of elements. You already have seen many iterables: lists, tuples, sets, dicts … etc.
- So, how do we iterate over the elements?

Iterators (2)

- An iterator over a sequence of elements:
  - Determines the first available element to visit; based on its defined criteria.
  - Returns the next available element when asked for it.
  - Stops the process after finishing all the elements.
- In Python, an iterator is an object that implements the Iteration Protocol.
- Iterators use lazy evaluation, i.e. they do not start the visiting process until it is needed.
  - To force enumeration of an iterator, convert it to a list: `list(iterator).`
  - This is NOT recommended in practice, but OK for debugging.
- For our purposes, iterators are just iterables that have not been enumerated. Ex: `range, reversed`

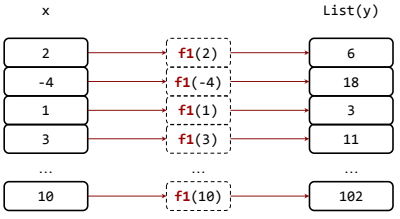
`map` Function

- An alternative, but even more powerful, mechanism for iterating over collections.
- `map` means: Collection(s) in => Collection Out.
- `map` takes in a “function” of `n` variables to apply it to n collections. (Collections need not be of the same type.)
- The function takes in the ‘elements’ of the collections NOT the collections.
- If the collections do not match in size, `map` will use the shortest collection as the output size.

`map` Function (2)

```
def f1(i):
    return i ** 2 + 2

x = [2, -4, 1, 3, …, 10]
y = map(f1, x) # "map" object
```



Note: Use a generator expression or list comprehension instead of `map(…)`. The `map(…)` function was not covered in lecture and is very rarely needed.

`map` Function (3)

Example 1: Scale a vector:

```
def scale_by_5(v):
    return 5 * v
vec = range(0, 11)
sVec = map(scale_by_5, vec)
# list(sVec) = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

Example 2: Element-wise product of two lists.

```
def getProduct(x, y):
    return x * y
lst_x = [2, 3, 6]; lst_y = [2, 4, 1]
pProd = map(getProduct, lst_x, lst_y)
# list(pProd) = [4, 12, 6]
```

Example 3:

```
lst_x = [0, 1, 2]; lst_y = [3, 4, 5]; lst_z = [6, 7, 8, 9, 10]
maxOfAll = map(max, lst_x, lst_y, lst_z)
# list(maxOfAll) = [6, 7, 8]
```

Note: Use a generator expression or list comprehension instead of `map(…)`. The `map(…)` function was not covered in lecture and is very rarely needed.

`map` Function (3)

Example 1: Scale a vector:

```
def scaleBy5(v):
    return 5 * v
vec = range(0, 11)
sVec = map(scaleBy5, vec)
# list(sVec) = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

Example 2: Element-wise product of two lists.

```
def getProduct(x, y):
    return x * y
lst_x = [2, 3, 6]; lst_y = [2, 4, 1]
pProd = map(getProduct, lst_x, lst_y)
# list(pProd) = [4, 12, 6]
```

Example 3:

```
lst_x = [0, 1, 2]; lst_y = [3, 4, 5]; lst_z = [6, 7, 8, 9, 10]
maxOfAll = map(max, lst_x, lst_y, lst_z)
# list(maxOfAll) = [6, 7, 8]
```

Note: Use a generator expression or list comprehension instead of `map(…)`. The `map(…)` function was not covered in lecture and is very rarely needed.

`map` Function (3)

Example 1: Get the Fibonacci numbers from 0 to 50:

```
def fibonacci(n):
    if n in (0, 1):
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

ints_0_to_10     = range(0, 10 + 1)
print("ints_0_to_10 ==", list(ints_0_to_10))
ints_0_to_10 == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

fib_0_to_10 = map(fibonacci, ints_0_to_10)
print(" fib_0_to_10 ==", list(fib_0_to_50_by_5))
fib_0_to_10 == [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
# list(to_50_by_5) = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

`filter` Function

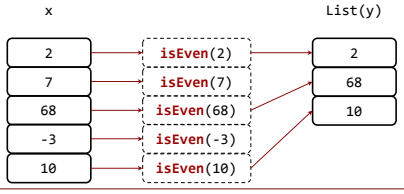
- Another function to work over collections.
- `filter` means: Collection in => Collection Out.
- `filter` takes in a “predicate”:
  - a function that returns `True/False.`
- The function operates on an ‘element’ of the collections NOT the collections.

Note: Use a generator expression or list comprehension instead of `filter(…)`. The `filter(…)` function was not covered in lecture and is very rarely needed.

filter Function (2)

```
def isEven(i):
    return i % 2 == 0

x = [2, 7, 68, -3, 10]
y = filter(isEven, x) # "filter" object
```



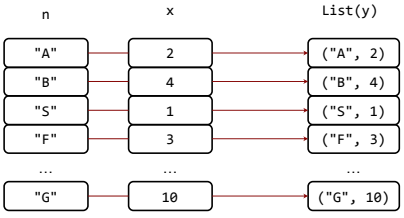
Note: Use a generator expression or list comprehension instead of `filter(…)`. The `filter(…)` function was not covered in lecture and is very rarely needed.

zip Function

- Creates an iterator of list of tuples from multiple lists.
- Useful for passing in multiple variables as one tuple.
- Keeps the shortest number of elements if lists have mismatched sizes.
- To use the longest list, use `itertools.zip_longest()`
- The inverse of `zip(…)` is `zip(*…)`: It unpacks a zipped iterable into different tuples.

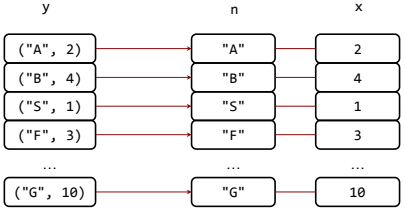
zip Function (2)

```
n = ["A", "B", "S", "F", "G"]
x = [2, 4, 1, 3, 10]
y = zip(n, x) # "zip" object
```



zip Function (3)

```
y = [('A', 2), ('B', 4), ('S', 1), ('F', 3), ('G', 10)]
n, x = zip(*y)
# n, x are tuples, not lists.
```



zip Function (4)

Example 1: Euclidean distance between two vectors

```
from math import sqrt
def squareDiff(i, j):
    return (i - j) ** 2

x = [2, 3, 6]
y = [0, 4, 7]

s = map(squareDiff, x, y)
# s = [(i - j) ** 2 for i, j in zip(x, y)]
dist = sqrt(sum(s)) # dist = 2.449489742783178
```

Example 2: Unpacking a list of tuples.

```
rep_list = [('Juan', 68), ('Rodney', 36),
            ('Edward', 57), ('Christine', 98)]
names, grades = zip(*rep_list)
# names = ('Juan', 'Rodney', 'Edward', 'Christine')
# grades = (68, 36, 57, 98)
```

enumerate Function

- The `enumerate` function facilitate accessing elements and their indices.
- Returns an iterator of list of tuples, which can be unpacked.
- Works in loops and in comprehension.

Example:

```
values = [28, 12, 71]
result = enumerate(values) # "enumerate" object.
# list(result) = [(0, 28), (1, 12), (2, 71)]
```

List Comprehension

- One of the most powerful features of Python.
- A syntactic construct for creating a list based on existing lists.
- Follows the form of the mathematical set-builder notation (set comprehension) as distinct from the use of map and filter functions.
- Provide a more concise and readable way to write loops.
- Remember: **List In -> List Out**

List Comprehension (2)

- General Form:

```
                |---- [] indicate a list result ---|
                |  one or more for clauses         |
                |           |                      |
                |     (<----|---->)                 |
l_out = [f(x) for x in l_in if predicate(x)]
  |       |       |      |  (<-----|----->)
  |       |       |      |         |
  |       |       |      |         |-----> (Optional) Filter
  |       |       |      |                 Function.
  |       |       |      |---------------> Iterator/Iterable (List,
  |       |       |                        Dict, Tuple ... etc.)
  |       |       |--------------------->  Element ID (For
  |       |                                every element in
  |       |                                l_in)
  |       |------------------------------> Function acting
  |                                        on element.
  |--------------------------------------> Resulting List.
```

List Comprehension (3)

Example 1:

```
x = [4, 3, -2, 11, 0, 9]
y = [3 * i for i in x]
# y = [12, 9, -6, 33, 0, 27]
```

Example 2:

```
names = ['jack', 'alex', 'mike', 'john', 'nancy', 'joan']
n = [name.capitalize() for name in names if name.startswith('j')]
# n = ['Jack', 'John', 'Joan']
```

Example 3:

```
people = [('Steve', 'Martin', 55),
          ('Thomas', 'Will', 37),
          ('Michelle', 'Angelo', 26)]
tags = ['"{0}, {1}" is {2} years old.'.format(last, first, age)
        for first, last, age in people]
# ['"Martin, Steve" is 55 years old.',
#  '"Will, Thomas" is 37 years old.',
#  '"Angelo, Michelle" is 26 years old.']
```

List Comprehension (4)

Example 4:

```
grades = [('Juan', 68), ('Rodney', 36),
          ('Edward', 57), ('Christine', 98)]
highest_grade = max([grade for _, grade in grades])
# highest_grade = 98
```

Example 5:

```
grades = [28, 12, 71, 64, 26, 97, 1, 7, 100, 68,
          57, 92, 29, 53, 8, 13, 84, 58, 69, 90]
above_90_count = len([ _ for g in grades if g >= 90])
# above_90_count = 4
```

Example 6:

```
numbers = [91, 4, 27, 74, 63]
nums = [(x, x % 3 == 0) for x in numbers]
# nums = [(91, False), (4, False), (27, True),
          (74, False), (63, True)]
```

List Comprehension (5)

Example 8:

```
x = [2, 3, 6]
y = [0, 4, 7]
cart_prod = [i * j for i in x for j in y]
# cart_prod = [0, 8, 14, 0, 12, 21, 0, 24, 42]

cart_prod2 = [i * j for i in x for j in y if i * j > 0]
# cart_prod2 = [8, 14, 12, 21, 24, 42]

cart_prod3 = [i * j for j in y for i in x]
# cart_prod3 = [0, 0, 0, 8, 12, 24, 14, 21, 42]
```

Dict Comprehension

- Dict comprehension is also very useful.
- General form is similar:

```
              |------ {} indicate a dict result -------|
              |        one or more for clauses         |
              |                 |                      |
              |           (<----|---->)                 |
dict_out = {key: value for x in l_in if predicate(x)}
```

- Set comprehensions are also possible.

```
set_out = {value for x in l_in if predicate(x)}
```

- For tuple comprehensions, use list comprehension, then cast to `tuple`.

Dict Comprehension (2)

Example 1: Dictionary Construction

```
ids = [121, 295, 330]
names = ["Mary Krantz", "Chris Haste", "Elizabeth Trudy"]
myMap = {i: n for i, n in zip(ids, names)}
# {121: 'Mary Krantz',
   330: 'Elizabeth Trudy',
   295: 'Chris Haste'}
```

Example 2: Dictionary Reversal

```
phoneLookup = {'Chris Haste': '(765) 394-8855',
               'Elizabeth Trudy': '(765) 471-0000',
               'Mary Krantz': '(765) 555-1234'}
nameLookup = {phone: name for name, phone in phoneLookup.items()}
# nameLookup = {'(765) 394-8855': 'Chris Haste',
                '(765) 555-1234': 'Mary Krantz',
                '(765) 471-0000': 'Elizabeth Trudy'}
```

NOTE: For this to work with no side effects, check value uniqueness.