Lecture 04: `dict`s and more iteration

Eric Franzosa, Ph.D.

franzosa@hsph.harvard.edu

Outline

- HW1 reminders
- HW2 notes
- `list` and `for` loop review
- `dict`ionaries
- Practice

Review

- The `list`
  - A **collection** `type` that stores multiple pieces of (potentially heterogeneous) data
  - Data (items) are enclosed by `[ ]`s and separated by `,`s
  - The data are indexed by 1st, 2nd, 3rd, etc. position (a sequence)

In [ ]:
```
my_list = [2, 3.1416, "crusts"]
```

- `lists` are mutable
  - Their contents can be changed under transformation
  - `list` methods [e.g. `.append( )`] can act **in place**

In [ ]:
```
my_list.append( "cost" )
my_list.append( 1.99 )
```

In [ ]:
```
print( my_list )
```

- We retrieve or set list positions using the index operator `[ ]`
  - Counting starts from 0, not 1, or from the list end using negative indices
- We can slice sections of the list with the slice operator `[start:end]`
  - `end` is not included in the slice

In [ ]:
```
print( my_list[0], my_list[1], my_list[-2] )
```

In [ ]:
```
my_list[-1] = 9.99
print( my_list )
```

In [ ]:
```
print( my_list[1:-2] )
```

Read the docs!

- You can learn more about Python's `list` type here:

  https://docs.python.org/3/tutorial/datastructures.html#more-on-lists (https://docs.python.org/3/tutorial/datastructures.html#more-on-lists)

- This page is also a good general introduction to the Python docs.
- *See if you can find a way to remove/pop the item from an arbitrary index in a list!*

the `for` loop

- Syntax for repeating actions without repeating code
  - E.g. applying the same transformation to each item in a collection

```
for X in Y:
    # transformations of X
    X = len( X )
    X = 2 * X
    print( X )
```

- Y is an *iterable* piece of data (e.g. a `list`)
- The "block" of indented lines define the *loop body*
- We will execute the code in the *loop body* for each item in Y
- X is a variable that holds the item we are *currently* working with

In [ ]:
```
my_list = [2, 3.1416, "crusts", "cost", 1.99]
```

In [ ]:
```
# iterating over a list in a variable
for x in my_list:
    x2 = 2 * x
    print( "in loop:", x, "->", x2 )
# this <print> is OUTSIDE the loop
print( "outside:", my_list )
```

In [ ]:
```
# impractical example
my_list = [2, 3.1416, "crusts", "cost", 1.99]
indices = [0, 1,      2,        3,      4   ]
```

In [ ]:
```
# iterating over a list by index to update it
for i in indices:
    old_value = my_list[i]
    new_value = 2 * my_list[i]
    my_list[i] = new_value # <- list is updated!
    print( old_value, "->", new_value )
```

In [ ]:
```
# this process CHANGED the list
print( my_list )
```

- `range( X )` returns X numbers starting with 0
- `len( list )` returns the number of items in the list
- Hence, `range( len( my_list ) )` returns the indices of `my_list`

In [ ]:
```
# a better approach to indexing
my_list = [2, 3.1416, "crusts", "cost", 1.99]
for i in range( len( my_list ) ):
    print( my_list[i] )
```

- `range( )` is a special type of function called a **generator**
  - It's safe to *think* of it as returning a `list` of numbers
  - In reality, it returns one number at a time, specifically for use with `for` loops

The `dict`

In [ ]:
```
# example of a Python <dict>
prices = {"Apple":0.49, "Banana":0.49, "Cantaloupe":2.99}
```

- The `dict` (short for "dictionary") is our next **collection** data type
- Dictionaries store a mapping from a set of keys to corresponding values
- (*key*, *value*) pairs are joined by `:`s, separated by `,`s, and enclosed by `{}`s

- Keys must be unique, immutable data (typically `str`ings)
- Values can be any kind of data (`str`ings, numbers, `list`s, other `dict`s)
- *Dictionaries are my person favorite data type: they are super versatile!*

In [ ]:
```
# another way of defining a <dict> that's easier to read
prices = {
    "Apple":      0.49,
    "Banana":     0.49,
    "Cantaloupe": 1.99, # <- "extra" comma avoids error later if we add 4th item
}
print( prices )
```

Indexing dictionaries

- Like `lists`, we can use the index operator `[]` to look up or set dictionary values
- Instead of indexing by position, we index using a key as an argument
- This returns the corresponding dictionary value for that key

In [ ]:
```
prices["Apple"]
```

In [ ]:
```
# looking up a non-existant key raises an error
prices["Tomato"]
```

- We can use the same indexing logic to update or define `dictionary` values

In [ ]:
```
# raise the price of an apple
prices["Apple"] = 99.99
```

In [ ]:
```
# add a price for another fruit
prices["Durian"] = 3.99
```

In [ ]:
```
# see the updated dictionary
print( prices )
```

- Like all data types, `dictionaries` have associated helper methods
- Like with `lists`, these methods can alter the dictionary in place and/or return values

In [ ]:
```
# .pop( KEY ) removes a key:value pair and returns the value
prices.pop( "Banana" )
```

In [ ]:
```
# note that we popped by key from the "middle" of the dictionary
print( prices )
```

In [ ]:
```
# .get( X, Y ) returns the value associated with key <X> like indexing
# ...but will return <Y> if key <X> is not defined
prices.get( "Elderberries", "price not found!" )
```

Looping over `dicts`

- By default, a `for` loop iterates over the keys of the `dict`
- I.e. the key is stored in the loop's temporary variable with each pass

In [ ]:
```
from time import sleep
prices = {"Apple": 0.49, "Banana":0.49, "Cantaloupe": 1.99, "Durian": 3.99}
```

In [ ]:
```
# note the use of an informative temp variable name, <fruit>
for fruit in prices:
    sleep( 1 )
    print( fruit )
```

- When it matters, the dictionary's order is the order in which keys were inserted
- We don't use this order for indexing, however (unlike with `lists`)

In [ ]:
```
# let's also see the price of the fruit
for fruit in prices:
    sleep( 1 )
    print( fruit, "->", prices[fruit] )
```

In [ ]:
```
# let's double the price of all fruits
for fruit in prices:
    # set new cost as 2 times (looked-up) old cost
    prices[fruit] = 2 * prices[fruit]
```

In [ ]:
```
# the price data were updated in place
print( prices )
```

Read the docs!

- You can learn more about Python's `dict` type here:

  https://docs.python.org/3/library/stdtypes.html#dict (https://docs.python.org/3/library/stdtypes.html#dict)

Application: Going to the store

In [ ]:
```
# Here's my shopping list, coded as a <dict>
to_buy = {
    "Apple":      2,
    "Banana":     6,
    "Cantaloupe": 1,
}
# we'll track my spending here, starting at 0
money_spent = 0
```

In [ ]:
```
# for each fruit on my shopping list...
for fruit in to_buy:
    how_many = to_buy[fruit] # <- look up how many I want
    subtotal = how_many * prices[fruit] # <- current fruit subtotal
    money_spent = money_spent + subtotal # <- update money spent
```

In [ ]:
```
print( money_spent )
```

- A more realistic example that will break
- Let's try to fix it together

In [ ]:
```
to_buy = {
    "Apple":      2,
    "Banana":     6,
    "Grapefruit": 1, # <- item not in prices!
}
money_spent = 0
```

In [ ]:
```
for fruit in to_buy:
    how_many = to_buy[fruit]
    subtotal = how_many * prices[fruit]
    money_spent = money_spent + subtotal
```

- `dict.items( )` is a handy method for iterating over (*key*, *value*) **pairs**
- Requires using two `,`-separated loop variables

In [ ]:
```
# using .items( ) saves us a line of code
money_spent = 0
for fruit, how_many in to_buy.items( ):
    subtotal = how_many * prices.get( fruit, 0 )
    money_spent = money_spent + subtotal
print( money_spent )
```

In [ ]:
```
# an even shorter version
money_spent = 0
for fruit, how_many in to_buy.items( ):
    money_spent = money_spent + how_many * prices.get( fruit, 0 )
print( money_spent )
```

- It's important to see that these versions are equivalent
- However, using fewer lines of code *is not necessarily better*
- **Favor clarity over brevity**

Practice: `for` loops (repeated from last time)

In [ ]:
```
# (1) replace <pass> with Python code that will print the first seven POWERS of
 2 (i.e. 1, 2, 4, etc.)
for n in [0, 1, 2, 3, 4, 5, 6]:
    pass
```

In [ ]:
```
# (2) replace <pass> with Python code that SUCCESSIVELY ADDS (i.e. sums) add the
 first seven primes as <my_sum>
primes = [2, 3, 5, 7, 11, 13, 17]
my_sum = 0
for p in primes:
    pass
print( my_sum )
```

In [ ]:
```
# (2') [NEW QUESTION!] Replace [] with range( ) to sum the numbers from 0 to N
# How big must N be for the calculation to be non-instantaneous?
my_sum = 0
for n in []:
    my_sum = my_sum + n
print( my_sum )
```

In [ ]:
```
# (3) replace [] with a range( len( ) ) motif to replace each number in <numbers
> with its absolute value
numbers = [-5, -3, -1, 0, 1, 3, 5]
for i in []:
    numbers[i] = abs( numbers[i] )
print( numbers )
```

Practice: `dicts` and loops

- The following dictionary stores grades for students at another school

In [ ]:
```
# (0) don't forget to execute this cell to define <grades>
grades = {
    "Alex": "A",
    "Beth": "A",
    "Carl": "B",
}
print( grades )
```

In [ ]:
```
# (1) Alter the SYNTAX of the code in (0) above to assign "Dina" a grade of "C",
 then re-evaluate (0)
```

In [ ]:
```
# (2) Use the index operator to assign "Fred" a grade of "B"
print( grades )
```

In [ ]:
```
# (3) Use the index operator to change Carl's grade to an "A"
print( grades )
```

In [ ]:
```
# (4) Complete/fix the <for> loop to print each student's name and grade on its
 own line
for name in grades:
    grade = ""
    print( name )
```

In [ ]:
```
# (5) Write a <for> loop above the print( ) statement to assign each student a g
rade of "A"
print( grades )
```

In [ ]:
```
# (6) Same task as above, but assign each student the first letter of their name
 as a grade
print( grades )
```

- **Thought question**: Many database systems behave like collections of dictionaries mapping one attribute onto another.
- (7) Why would using first (and/or last) names *not* be a good choice of key in a university database storing student grades?
- (8) What would make a better key?
- (9) Based on your answer to (8), what *other* mapping likely exists in a university's grade database?

Bonus: Counting with `dicts`

- One of the useful features of a dictionary is counting repeated elements in a list (or other iterable data)
- Note that in the first example here, we use numbers (`int`s) as keys to a dict
  - While less common than `str`ing-based keys, `int`s are also immutable, so this works just fine

In [ ]:
```
numbers = [1, 2, 2, 7, 4, 9, 7, 4, 8, 2, 4, 4, 6, 6, 7, 2, 4, 1, 2, 9, 8, 1, 8,
 7, 9]
# an empty dictionary
counts = {}
for n in numbers:
    # what does this line of code do?
    counts[n] = counts.get( n, 0 ) + 1
print( counts )
```

In [ ]:
```
# len( dict ) returns the number of keys in the dict, i.e. the number of unique
 numbers in <numbers>
len( counts )
```

In [ ]:
```
# (1) Challenge: Write python code to count the UNIQUE characters in the followi
ng phrase
phrase = "Challenge: Write python code to count the UNIQUE characters in the fol
lowing phrase"
```

Bonus: Nesting collections and loops

- Collections (like lists and dicts) can store other collections as items/values

In [ ]:
```
# a dict containing small lists as values
nested = {
    "A": [1, 2, 3],
    "B": [4, 5, 6],
    "C": [7, 8, 9],
}
```

In [ ]:
```
# explore nested data with nested <for> loops
for key, numbers in nested.items( ):
    for n in numbers:
        print( key, n )
```

In [ ]:
```
# (1) Challenge: Write a nested <for> loop to decrease all list values in <neste
d> by 1
# for example, after your code runs, "A" should map to [0, 1, 2]
print( nested )
```