Lecture 03: Lists and iteration

Eric Franzosa, Ph.D.

franzosa@hsph.harvard.edu

Outline

- Homework overview
- L02 review
- Intro to collections (the `list`)
- Intro to iteration (the `for` loop)

Review

- Numbers (`ints` and `floats`) and `strings` are immutable
  - They may be used as input to a transformation, but are unchanged by it
  - Such transformations return NEW data

- Data are transformed with operators, functions, and methods
  - Typically taking one or more pieces of input data (arguments)
  - Typically returning one or more pieces of output data
  - Returned data can be directly acted upon

In [ ]:

```
# a chain of transformations
print( 2 * "banana".replace( "a", "o" ) )
```

Review

- Variables are "buckets" for storing data
    - Hard-coded data, data read from files, or outputs from transformations
    - Variables are **defined** or **updated** with the = operator

- Acting on a variable **is** acting on its stored data

- Number and string data in a variable aren't changed under transformation
    - Must explicitly update/overwrite the variable

Canvas poll

- What will the following two code blocks print to the screen?

```
# block 1
var1 = 5
var1 * 2
print( var1 + 2 )
```

```
# block 2
var2 = 5
var2 = var2 * 2
print( var2 + 2 )
```

The `list`

In [ ]:
```
# example of a Python list
[23, "Bob", -5, 3.1416, "popsicle"]
```

- The `list` is our first example of a collection data type
  - Can store multiple pieces of data (items)
  - Enclosed by `[ ]`s
  - Data can be of heterogeneous types
  - Items have a natural order by position (1st, 2nd, 3rd, etc.)

- Critical for organizing multiple pieces of "atomic" data
  - E.g. a vector of `float` measurements over time
  - E.g. lines of text (`str`s) from a document

- We can store list data in a variable (just like any other data)

In [ ]:
```
my_list = [5, 3, 1, 2, 4]
```

- And we can act on it (through the variable) with built-in functions

In [ ]:
```
# len( ) returns the number of items in the list
len( my_list )
```

In [ ]:
```
# sorted( ) returns a sorted copy of the list
sorted( my_list )
```

In [ ]:
```
# sum( ) will return the sum of the list, if all entries are numbers
sum( my_list )
```

- Like other data types, `lists` have associated methods
- **Key concept**: But unlike numbers and strings, lists are **mutable**
    - Their methods can *change the data in the list*

In [ ]:
```
# an empty list
my_list = []
```

In [ ]:
```
# .append( ) adds an item to the end of a list
my_list.append( "Apple" )
```

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

In [ ]:
```
my_list.append( "Banana" )
my_list.append( "Cantaloupe" )
print( my_list )
```

- List methods that change the list **in place** may not return anything
    - Hence we use `print( )` to inspect the transformed list

In [ ]:
```
my_list = ["Apple", "Banana", "Cantaloupe"]
```

In [ ]:
```
# .reverse( ) reverses the elements of the list in place
my_list.reverse( )
print( my_list )
```

In [ ]:
```
# .sort( ) sorts the list in-place
my_list.sort( )
print( my_list )
```

- `list.pop( )`
    - Removes the last item from the `list`
    - It *also* returns that item for us to work with

In [ ]:
```
my_list.pop( )
```

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

The index operator, [ ]

- The index operator allows us to access or set specific entries in a list
- We access the entries based on their position in the list

Key Concept: Python starts counting positions from 0 not 1!

- This is more the rule than the exception in computer programming
    - R and MATLAB are exceptions
- Counting from 1 (and being "off-by-one") is among the most common programming mistakes

In [ ]:
```
names = ["Alex", "Brian", "Chris", "David", "Eric"]
```

In [ ]:
```
# the first element, index 0
names[0]
```

In [ ]:
```
# the second element, index 1
names[1]
```

In [ ]:
```
# we can index from the end using negative numbers
names[-1]
```

- Indexing *returns* an element of a list
- We can immediately act on the returned value

In [ ]:
```
names[-1] * 5
```

In [ ]:
```
"Am" + names[-1].lower( ) + "a"
```

- Indexing can be used with assignment (=) to change existing list values

In [ ]:
```
names[1] = "BETHANY"
names[3] = "DENISE"
print( names )
```

- Trying to access or change a position that doesn't exist will raise an error

In [ ]:
```
# an "off-by-one" error: we forgot to start counting from 0
names[5].upper( )
```

The slice operator, `[ : ]`

- Returns a range of values from a list (a "slice") as a new list
- The syntax is `DATA[start:end]`
- `DATA[end]` is NOT included in the slice
    - It's where we stop, not the last item we add

In [ ]:

```
primes = [2, 3, 5, 11, 13, 17, 23]
primes[1:4]
```

In [ ]:

```
primes[-3:-1]
```

- start and end are optional

In [ ]:
```
# if not supplied, <start> defaults to 0
primes[:5]
```

In [ ]:
```
# while <end> defaults to len( list )
primes[2:]
```

In [ ]:
```
# combining them slices the full list (makes a copy)
primes[:]
```

- Indexing and slicing works on any "ordered" data
    - E.g. `strings` (but not numbers)

In [ ]:
```
"Eric Franzosa"[0]
```

In [ ]:
```
"Eric Franzosa"[0:4]
```

In [ ]:
```
"Eric Franzosa"[5:]
```

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 = f( X )
    X = g( 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 [ ]:
```
# ignore this for one moment
from time import sleep
```

In [ ]:
```
for x in [0, 1, 2, 3, 4, 5]:
    #sleep( 1 )
    x2 = x ** 2
    print( x, "squared is", x2 )
```

In [ ]:

```
# same idea, but the list is in a variable
numbers = [0, 1, 2, 3, 4, 5]
for x in numbers:
    x2 = x ** 2
    print( x, "squared is", x2 )
```

- `range( X )` returns X numbers starting with 0
    - Hence the last number will be X - 1

In [ ]:

```
# same idea using range( )
for x in range( 6 ):
    x2 = x ** 2
    print( x, "squared is", x2 )
```

- Let's use a for loop to capitalize some words

In [ ]:
```
words = ["purple", "monkey", "dishwasher"]
# note: using <w> for temp variable now
for w in words:
    w = w.upper( )
    print( w )
```

In [ ]:
```
# the data in <words> didn't change
print( words )
```

- To change the elements of the list, we use the `range( len( ) )` motif

```
In [ ]: words = ["purple", "monkey", "dishwasher"]
        # using <i> as the temp variable, short for "index"
        for i in range( len( words ) ):
            words[i] = words[i].upper( )
            print( "the word at index", i, "is now", words[i] )
```

```
In [ ]: # <words> now has updated data
        print( words )
```

- Here's another approach to the same problem

In [ ]:

```
words = ["purple", "monkey", "dishwasher"]
words2 = []
for w in words:
    words2.append( w.upper( ) )
words = words2
```

In [ ]:

```
# we overwrote <words> with the data we want
print( words )
```

Practice: A `list` of planets

In [ ]:

```
# (0) evaluate this cell to store the planets in <planets> (sorry, Pluto)
planets = ["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Neptune", "Uranus"
]
```

In [ ]:

```
# (1) use the index operator to return "Earth"
planets
```

In [ ]:

```
# (2) use the index operator with a NEGATIVE argument to return "Nepture"
planets
```

In [ ]:

```
# (3) use the slice operator to pull out the rocky planets as a new list (MVEM)
planets
```

In [ ]:

```
# (4) use .append( ) to restore Pluto's status as a planet
planets
print( planets )
```

In [ ]:

```
# (5) write Python code to identify the first planet in ALPHABETICAL order
abc_first = ""
print( abc_first )
```

Practice: for loops

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

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

Practice: More `lists` and loops

In [ ]:
```
# (0) evaluate this cell to store the days of the week in <days>
days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
```

In [ ]:
```
# (1) write a <for> loop to extract and store the FIRST LETTER of each day's name in <da
ys2>
days2 = []
# your loop here
print( days2 )
```

- `Y.join( X )` will join a list of strings (X) with the string Y as a spacer
  - The concatenated result is returned as output
  - X can be the empty string, `""`

In [ ]:
```
# (2) use join to concatenate the abbreviations in <days2> as a single string
days2
```

- `X.split( Y )` will split a string X at each instance of the string Y
  - A `list` of substrings of X is returned
  - `X.split( Y )` reverses `Y.join( X )`

In [ ]:
```
# (3) use .split( ) to redefine <months> as list of the individual months
months = "January February March April May June July August September October November D
ecember"
months = []
```

In [ ]:

```
# (4) write a <for> loop to replace each month's full name with its 3-letter abbreviatio
n
print( months )
```

Bonus: `lists` and operators

- lists interact with the + and * operators similarly to strings

In [ ]:
```
# evaluate this cell to define the two lists
A = [1, 2, 3]
B = [4, 5, 6]
```

In [ ]:
```
# (1) "add" the two lists together and inspect the results
```

In [ ]:
```
# (2) multiply <A> by a small integer and inspect the results
```

In [ ]:
```
# (3) how does the following transformation differ from what you did in (1)?
A.append( B )
```