Week 5 - Monday

COMP 1800

Last time

- What did we talk about last time?
- Vigenère cipher
- Work time for Assignment 3

Questions?

Assignment 4

Data

Managing data

- **Data** is a collection of items
  - Often numbers
  - Collected by observation or measurement
- In the modern age, companies and businesses can collect millions of data points every day
  - Weather information
  - Customer purchases
  - Online behavior
  - Sales numbers
- Fortunately, computers are great at storing and processing huge amounts of data

Example: Finding the biggest of four things

- Assignment 3 included a function to find the biggest of three things
- What if we wanted the biggest of four things?

```
def biggest(a, b, c, d):
    largest = a
    if b > largest:
        largest = b
    if c > largest:
        largest = c
    if d > largest:
        largest = d
    return largest
```

- The code is getting more and more complicated
- We have to use a different function for different numbers of items

Collections

- To avoid the messiness of having a new variable for each piece of data, Python provides **data structures**
  - Strings
  - Lists
  - Dictionaries
  - Tuples
  - Ranges
- A single data structure can hold many pieces of data
- These data structures all allow iteration
  - Visiting each item of data inside them

Terminology

Contents	**Homogeneous** data structures hold items that are all the same kind	**Heterogeneous** data structures can hold different kinds of data
Ordering	**Ordered** (or sequential) data structures hold items in a particular order	**Unordered** (or non-sequential) data structures make no guarantees about the order of items
Mutability	**Mutable** data structures can have their contents changed	**Immutable** data structures cannot be changed

- Data structures have different ways to **index** into them, meaning getting the contents inside
- Data structures use different delimiters to mark their contents: `()`, `{}`, `[]`, `''`, `""`

Strings

- Let's start with strings, since they're a data structure we know
  - **Homogeneous**: all contents are characters
  - **Sequential**: characters are stored in a particular order
  - **Immutable**: a string can't be changed
  - Delimited by `' '` or `" "`
  - Indexed by integer position using `[ ]`
  - Negative indexing is allowed
  - Slicing with `[ : ]` is supported

Lists

Lists

- Python provides a way to make lists of general objects
- To make a list, you can put a collection of objects inside square brackets

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday']
```

- They can even be different types

```
stuff = ['Danger!', 3, True, 1.7]
```

Lists

- Using the terminology introduced before, lists are:
  - **Heterogeneous**: you can put different kinds of data into a list, but Python programmers usually try not to do this, since it's confusing
  - **Sequential**: items are stored in a particular order
  - **Mutable**: individual items can be changed, and the size of the list can be changed
  - Delimited by `[]`
  - Indexed by integer position using `[]`
  - Negative indexing is allowed
  - Slicing with `[:]` is supported

Accessing an element

- As with strings, use square brackets and a number to access an element in the list

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday']
bleh = days[0] # contains 'Monday'
```

- Like strings, elements are numbered from 0 to the length – 1
- You can use the **`len()`** function to get the length of a list

```
count = len(days)   # contains 7
```

Changing elements in a list

- You can also change the elements in a list using the square bracket notation

```
birds = ['Duck', 'Duck', 'Duck']
birds[2] = 'Goose'
print(birds) #prints ['Duck', 'Duck', 'Goose']
```

- This is one of the bigger differences between strings and general lists
- You **cannot** change the characters in a string
- You have to make a new string

Slices on lists

- Just like strings, you can use the slice notation to get a copy of part of a list

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday']
weekend = days[5:7]
print(weekend) #prints ['Saturday', 'Sunday']
```

- The same shortcuts for string slices still work:
  - Python assumes 0 if you leave off the first number
  - It assumes the length if you leave off the last number

```
weekdays = days[:5] #Monday through Friday
```

Multiplying a list

- Like with strings, you can multiply a list by an integer to get a list made up of multiple copies of the list repeated

```
greeting = ['Hello']
manyGreetings = greeting * 5
# manyGreetings contains:
# ['Hello', 'Hello', 'Hello', 'Hello', 'Hello']
```

Empty list

- Just like an empty string, you can have an empty list

```
data = []
```

- Such a list contains no items and has a length of zero

```
length = len(data)
print(length) #prints 0
```

- Why would you want an empty list?

Adding elements to a list

- You can add elements to a list, empty or otherwise
- One way is by using the **`append()`** method, which adds elements to the end of a list

```
data = []
data.append(3)
data.append(7)
data.append(8)
print(data) # prints [3, 7, 8]
```

- There are other ways to add (and remove) items from a list

Useful list methods

Method	Example	Description
`list`	`list(range(100))`	Make a list from the given sequence
`append`	`items.append('goat')`	Add an item to the end of the list
`insert`	`items.insert(4, 'thing')`	Insert an item at a location, moving everything else down
`pop`	`items.pop()`	Remove the last item in the list and return it
`pop`	`items.pop(5)`	Remove item at a given location ad return it
`sort`	`items.sort()`	Sort the list
`reverse`	`items.reverse()`	Reverse the list
`index`	`items.index('walnut')`	Return the first location where an item can be found
`count`	`items.count('apple')`	Count the occurrences of an item
`remove`	`items.remove('goat')`	Remove the first occurrence of an item
`clear`	`items.clear()`	Remove everything from a list

split() method

- Sometimes you get a string that contains a lot of words
- You'd like to split the string up into a list of those individual words so that you can deal with each
- Calling the **`split(' ')`** method breaks up a string based on the space character, giving such a list

```
sentence = 'I seem to be having tremendous difficulty with
my lifestyle.'
words = sentence.split(' ')
print(words[5])      # prints tremendous
print( len(words) ) # prints 10
```

Looping over the contents of lists

- Just as with strings, we can use a **`for`** loop to iterate over everything in a list
- Directly:

```
for item in list:
    print(item)
```

- Or by using an index:

```
for i in range(len(list)):
    print(list[i])
```

- The first version is simpler, but sometimes we need to know the index

Sieve of Eratosthenes

- The Sieve of Eratosthenes is an ancient approach for finding prime numbers
- Quick reminder: Prime numbers are integers greater than 1 that are divisible only by themselves and 1
- Algorithm:
  - Make a list of all the numbers up to some point
  - For each prime number, eliminate all the numbers that are multiples of it
  - Working through the list of numbers, each time you find a number that hasn't been eliminated yet, it must be prime

Sieve of Eratosthenes

- Let's use the Sieve of Eratosthenes to print prime numbers up to some number **`n`**
- Algorithm:
  - Make a list called **`isPrime`** of length **`n + 1`** by multiplying a list containing the value **`True`** by **`n + 1`**
  - Make an empty list called **`primes`**
  - Loop over every index in **`isPrime`**, starting at 2
    - If its value is **`True`**
      - Append the index to the list **`primes`**
      - Loop from twice the index up to the end of **`isPrime`**, taking steps as big as the index
        - Mark each element **`False`**, since it's a multiple of index
  - Return **`primes`**

```
def eratosthenes(n):
```

Visualization of Sieve of Eratosthenes

- Let's consider only the numbers from 1 to 10 to keep it simple
- We make a list that includes 0 and 1, just to make the indexes easy to deal with

True	True	True	True	True	True	True	True	True	True	True
0	1	2	3	4	5	6	7	8	9	10

- Starting at index 2, we mark every multiple of 2 (starting at 4) **`False`**

True	True	True	True	False	True	False	True	False	True	False
0	1	2	3	4	5	6	7	8	9	10

- Then, on the next **`True`** element (3), we mark all the multiples of 3 (starting at 6) **`False`**

True	True	True	True	False	True	False	True	False	False	False
0	1	2	3	4	5	6	7	8	9	10

Upcoming

Next time…

- Statistics
  - Mean
  - Median
  - Mode
- Dictionaries

Reminders

- Focus on section 4.5 of the textbook
- Start Assignment 4