Outline

• What is a Data Structure?
• What is a Sequence?
• Sequences in Python
• All About Strings
What is a Data Structure?

- A Structure for Storing Data
- Formally defined parts
- Formally defined relations between parts
- Particular algorithms are designed to run with particular data structures
- We will focus on some data structures that are implemented in Python
  - Note that other programming languages may use the same names for different structures
What is a Sequence? What is a Sequence in Python?

• An ordered set of elements (math, e.g., permutations)
• In computer science, there are more than one way for elements to be arranged in a sequence. Python Examples:
  – Lists, Strings, Ranges, Tuples
    • different syntax
    • different functions designed for handling them
  – A string is a sequence of characters
  – Ranges are defined by start and end numbers
  – A list must contain a collection of elements
    • It is possible to alter a list, once created
  – Tuples:
    • Can consist of multiple types
    • Cannot be changed once created
Strings in Python

• A String is a sequence consisting of characters
  – Characters also have special properties

• Special syntax allows the identification of subsequences or “slices”

• Special Python functions operate on the data structure “string”
  – testing, searching, changing case, formatting, stripping, splitting, etc.
New Data Type: Character

• Character
  – The smallest part of a string
  – Typically represented by one byte

• Character ↔ Unicode Number:
  – chr(number) ## Number to unicode character
  – ord(character) ## Unicode character to number
  – Unicode Chart (base 10):
    • http://www.tamasoft.co.jp/en/general-info/unicode-decimal.html
Using Characters

• Convert Upper Case to Lower Case
  – Let's try to figure this out logically by trying out the type conversions on the previous slide
    • ord('a')
    • ord('A')
    • Use `chr` to convert numbers to characters
    • Use `for` loop to convert words
  – Do the reverse: convert Lower Case to Upper Case

• Convert Number Characters 1-9 to corresponding letters using a similar strategy

• Convert whole strings using a `for loop`
Printing, Characters and Strings

- Special Characters can be part of strings
  - \n = newline character
  - \t = tab character

- Try
  - print('Hello\nWorld')
  - print('Hello\tWorld')

- Unicode Characters
  - Python supports both ASCII and Unicode
  - \uxxxx = 4 digit (base 16) unicode character
    - Equivalent to the base 10 numbers used for ord and chr
  - print('\u0770') ## Arabic letter (sh sound)
  - [http://www.utf8-chartable.de/unicode-utf8-table.pl?number=1024&utf8=string-literal](http://www.utf8-chartable.de/unicode-utf8-table.pl?number=1024&utf8=string-literal)
Common Escape Characters

- `\` backslash
- `'` single quote
- `"` double quote
- `\n` newline
- `\r` (carriage) return
- `\t` tab
Let's number the positions around the characters, beginning with 0 and ending with the length of the string

• Given a string: 'chicken'
• Lets number the positions around each character starting with zero:
  \[\text{c h i c k e n} \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7\]
• This now allows us to refer to:
  – the characters beginning at 0 or 1 or 2 ….  
  – the characters preceding or following 3 
  – the characters between 2 and 5 
  – etc
Indices from Either Direction

- An Index allows access to items in a sequence beginning at any position from 0 to length – 1 (no character begins at length – that would be the end of the string)
  - 'Hello'[0] == 'H'
  - 'Hello'[1] == 'e'
  - ...
  - 'Hello'[4] == 'o'

- A Negative index allows access to items in a sequence counting in reverse – negative indices refer to distances from the end.
  - 'Hello'[-1] == 'o'
  - 'Hello'[-2] == 'l'
  - ...
  - 'Hello'[-5] == 'H'
Slices: Parts of Strings (and some other sequences)

- 'dishes'[0:2] == 'di'
- 'dishes'[4:6] = 'es'
- 'dishes'[:2] == 'di'
- 'dishes'[-2:] == 'es'
- 'dishes'[:] == 'dishes'
- `SEQUENCE[start:end]`
  - `start` and `end` can be positive integers from 0 to the length of the sequence or negative integers up to -1 X the string length
  - If `start` is left out, the string starts from the beginning
  - If `end` is left out, the string goes all the way to the end
Example: Regular Plurals in English

• This is for “normal” words, not exceptions
  – Not *sheep*, *oxen*, *octopi*, *aircraft*, *men*, *women*, …
  – These could be handled by a separate dictionary
• If final letter is a vowel, add 's'
• Else if final letter is “y”
  – If second-to-last letter is vowel, add 's'
  – Else remove “y” and add “ies”
• Else if final letters are a member of (x, s, z, ch, sh)
  – Add “es”
• Else add 's'
Morphological Rules in Linguistics

• Morphological rules include
  – Rules that add suffixes and/or prefixes
    • noun + -s
  – Other regular sound changes that result in different forms of the same word
    • 'sit' + past → 'sat'

• Irregular morphology
  – Depends on the grammar, one assumes
    • 'sit' → 'sat' is either irregular or a regular instance of an irregular paradigm
  – Some cases would be irregular for all grammars
    • 'go' + past → 'went'
Implementing the Plural Rule in Python

• morphology.py

• Uses the member operator **in**
  – A boolean operator which tests whether an item is a member of a sequence

• Uses another kind of sequence: the list
  – Delimiters = square brackets
  – Members = python objects
  – Separators = commas

• Structure of program: Decision tree using logical operators
Several Slides Listing String Functions

- Go to example-string-functions.py
- A few listed on the next few slides
  - A small subset used on midterm2 and final
- These all take the form:
  string.functionname(arguments)
- Examples,
  - 'abc'.islower()
    - Evaluates as True
  - 'Hello World'.center(20,'*')
    - Evaluates as '****Hello World*****'
**string.functions(): Case/Format**

- **Case-Changing Functions**
  - Example: `s = "the tourist saw Mary"
  - `s.lower()`, `s.upper()`, `s.swapcase()`
  - `s.captitalize()` --- `s[0]` only
  - `s.title()` – similar except capital after space

- **Format Functions**
  - `s.center(LENGTH, ch)` – e.g., *** string ***
  - `s.ljust(length, ch)`, `s.rjust(length, ch)` – similar
  - `s.format(vars)`
    - '{whose} {thing} is nice'.format(whose = 'John\'s', thing = 'code')
    - Additional details for fine-tuning print out
      - [https://docs.python.org/3.1/library/string.html#formatspec](https://docs.python.org/3.1/library/string.html#formatspec)
    - Example 2 with fancier formatting (:.4 and :.2 indicate number of decimal places):
      - 'First Number: {num1:.4} Second number: {num2:.2}'.format(num1=5.344,num2=6.923455)
string.function(): Tests and Search

• Testing (Boolean)
  – endswith(suffix)
  – startswith(prefix)
  – isalnum(), isalpha(), isdigit(), isnumeric(), isidentifier(), islower(), isupper, istitle(), isprintable(), isspace()

• Search functions
  – find(substring), rfind(substring)
    • return index or -1
  – index(substring), rindex(substring)
    • return index or error
string.functions(): Stripping off Characters

- **Stripping Functions**
  - Remove unwanted characters from edges of string

- **s.strip(optional_arg)**
  - If left out all white space characters are stripped
    - (tab, space, newline, …)
  - Otherwise all characters in optional_arg string

- **s.lstrip and s.rstrip (left or right only)**
Split and Partition functions

• Partition
  – s.partition(arg), s.split(arg)
  – create a list of substrings, partitioned by arg

• Split **** Useful for Homework ****
  – Example: “five hundred thirty”.split(' ') → ['five','hundred','thirty']
  – Split does not include the separators, but partition does
    • Try “five hundred thirty”.partition(' ')

• Rightward Versions
  – rpartition and rsplit variants: search for separators from right
    • only relevant if an optional max argument is used
Lists in Python

• left square bracket, elements separated by commas, right square bracket
  – Example: [1,2,3,4]

• Same system for slices and identifying elements as used for strings
  – list_of_4 = [1,2,3,4]
  – list_of_4[0] → 1
  – list_of_4[1:3] → [2,3]

• Additional feature, you can change a list using indices
  – list_of_4 = [1,2,3,4]
  – list_of_4 → [1,2,3,'jello']

• Convert strings to list of strings
  – 'This is a list'.split(' ') → ['This','is','a','list']
Lists with *in*, *len* and *loops*

- The *in* operator and function *len* behave as expected
  - 4 in [1,2,3,4] → True
  - 99 in [1,2,3,4] → False
  - len([1,2,3,4]) → 4

- *for* loops behave as expected
  - for item in [1,2,3,4]:
    print(item)

- *while* loops with accumulators
  
  ```python
  big_string = "
  index = 0
  words = ['the', 'big','green','monster']
  while index < len(words)
      big_string = big_string+words[index]+''
      index = index + 1
  big_string → 'the big green monster ' ## note extra space at the end
  ```
List Methods that Change Lists

- list.append(X) – adds an item to the end of a list, by changing the list
  - Abc = ['a','b','c']
  - Abc.append('d')
  - Abc → ['a','b','c','d']

- list.pop() – removes the last item in the list and returns it
  - Abc.pop()
    - returns 'd'
    - Abc → ['a','b','c']

- list.pop(indexX) – removes the item beginning at indexX (similar to keyword del)
  - Abc.pop(0)  ## like del Abc[0] (except del does not return anything)
    - Returns 'a'
    - Abc → ['b','c']
List Methods that Change Lists 2

- List.extend(list2) – adds items in list2 to list
  - Abc.extend(['d','e'])
  - Abc → ['a','b','c','d','e']

- List.reverse() – changes the order of a list, turning it backwards
  - Abc.reverse()
  - Abc → ['d','e','c','b','a']
Lists are Mutable

- Lists can be changed in a different way than other data types we have discussed up until now.
- Functions/Methods on strings create new strings
  - Abc = 'abcd'
  - Abc.upper() ## produces a new string
  - Abc = ['a','b','c']
- Functions/Methods on lists change existing list
  - Abc.reverse()
  - The variable Abc points to a list
    - The list exists independently of the variable
    - Using list methods on the variable will change the list it points to
    - Even if Abc is global, a function can change the list it points to
Other Operator/Functions for lists/strings

• + – like List.extend, but does not change the list
  – Abc = ['a','b','c']
  – Efg = ['e','f','g']
  – Abc + Efg → ['a','b','c','d','e','f','g']  ## returns combo
  – Abc → ['a','b','c']  ## does not change input list

• >, < – sort order of strings (by unicode number)
  – 'abc' < 'efg'
  – 'EFG' < 'abc'

• max, min – finds first/last item in list (per unicode order)
  – max(['abc','efg','EFG']) → 'efg'
  – min(['abc','efg','EFG']) → 'EFG'

• List.sort() – sorts the items in a list, comparing elements with max
  – my_list = ['abc','efg','EFG']
  – my_lists.sort()
  – my_list → ['EFG', 'abc', 'efg']
Converting Spelled Out Numbers (HW)

- What is “two hundred sixty two”?
- 'two hundred sixty two'.split() → ['two', 'hundred', 'sixty', 'two']
- Convert
  - two → 2, hundred → 100, sixty → 60, two → 2
- Combining numbers in a sequence
  - Lower Higher: multiplication
    - two hundred → 200
  - Higher Lower: addition
    - two hundred sixty → 260
      - works if 2 X 100 was applied first.
  - Equal Equal: Error
    - two two ???
    - Doesn't happen in normal text
- This method would not work for numbers over 1000
Extending to Cover Numbers 1000 and higher

- Applying method on previous slide to larger numbers requires refinement:
  - Ex: One hundred twenty seven thousand three hundred one
    - \(((1 \times 100) + 20 + 7) \times 1000) + (3 \times 100) + 1 \rightarrow 127,101\)

- English numbers divide into units of 0 → 999
  - Go through the number list more than once, creating smaller lists on each pass
    - First only combine numbers less than 1000 (as per previous slide)
      - handle cases like “one hundred fifty three” wherever they occur in the string (even if they modify thousand, million, etc.)
    - Next multiply instances of numbers more than 1000, with preceding numbers less than 1000
    - On a final pass, add the remaining numbers together

- For example, 'five hundred thirty five thousand seven hundred one'
  - ['five', 'hundred', 'thirty', 'five', 'thousand', 'seven', 'hundred', 'one'] # split
  - [5, 100, 30, 5, 1000, 7, 100, 1] ## convert to numbers
  - [535, '1000,701] # on 1st pass, covert sequences of less than 1000
  - [535000, '701] # on second pass, multiply 1000 and up, with preceding numbers less than 1000
  - 535701 ## finally add all numbers together
Walk Through for number over 1000

- Your loop must keep track of more than one item by looking ahead or behind or storing intermediate solutions to problems:
  - 2 variables: **output** (accumulates output); **hold** stores number you are “working on”
  - Part 1: ['four', 'thousand' 'two', 'hundred', 'sixty', 'two'] → [4, 1000, 2,100,60,2]
  - for number in [4, 1000, 2, 100, 60, 2]
    - Iteration 1: store 4 in hold
    - Iteration 2: 1000 is over 999, store both 4 and 1000 in output (empty hold)
    - Iteration 3: store 2 in hold
    - Iteration 4: multiply 2 X 100 and store 200 in hold (replacing 2)
    - Iteration 5: add stored 200 and 60 – store 260 in hold (replacing 200)
    - Iteration 6: add 260 and 2 – store 262
  - But the remaining item in **hold** into **output**.
  - **Output** now equals: [4, 1000, 262]
  - Program stores partial results in **hold** and puts those results into **output** when ready

- The remaining steps:
  - Multiply: [4, 1000, 262] → [4000, 262]
  - Add: [4000, 262] → 4262
Larger Example:
One million five hundred three thousand four hundred seventy three

- Make number list: One million five hundred three thousand four hundred seventy three → [1, 1000000, 5, 100, 3, 1000, 4, 100, 70, 3]
- Run on parts of sequence less than 1000:
  - [1, 1000000, 5, 100, 3, 1000, 4, 100, 70, 3] → [1, 1000000, 503, 100000, 473]
  - (requires repeatedly storing temporary results less than 1000)
- Do Multiplication
  - [1, 1000000, 543, 1000, 473] → [1000000, 543000, 473]
- Do Addition
  - [1000000, 543000, 473] → 1,543,473
Summary I

• Sequences are Data Structures in which items are combined together in a predescribed order
• Sequences share certain properties in Python, but many also have special functions and operators specific to them.
• Strings are sequences of Characters
• Strings are important for the print function, as well as other processing involving text
Summary II

• String manipulation involves
  – slicing and concatenating strings
  – converting characters to other characters
  – looping through sequences and making regular changes

• String manipulation is important for several applications
  – Applications involving linguistics: morphology, spell-checking, information extraction, machine translation, search, etc.
Summary III

- Lists are sequences of any type of element
- Lists are mutable
  - Rather than creating new lists, some functions actually change the lists that they operate on
  - If a local variable points to a list, functions operating on that variable can change the list
- Strings can be split apart to create lists
- Lists are useful for applying functions to particular items in a sequence.
Homework (Due 17th Class)

• http://cs.nyu.edu/courses/spring16/CSCI-UA.0002-004/hw6.html