Strings again

zyBooks
"Programming in Python 3"
chapter 7

Strings and Slices

- Review from chapter 3 slides:
  - A *slice* of a string is a substring
  - Uses index-like notation, but with starting and stopping index values
  - The substring is a *copy* of what was in the string
    - if the string gets changed, the substring remains the same

Slicing Strings into Substrings

- Example:
  - ord = "O'Hare International Airport'
  - ord**[3 : 7]** ('are ')
    - Slice (substring) starts with element 3, ends *just before* element 7 (*i.e.* last element is element 6)
  - ord**[12 : -1]** ('national Airpor')
    - Up to, but *not including*, the last character
  - ord**[0 : len(ord)]**
    - This "slice" is actually a copy of the entire string
    - Can also be done as ord**[:]**

Slicing Strings into Substrings

- Starting-index of 0 can be omitted; finishing-index that is equal to string's length can be omitted
  - ord**[: 5]** copies characters from beginning, up to (but not including) position 5
  - ord**[3 :]** copies characters from position 3 through end of string
  - ord**[:]** copies characters from beginning through to the end
    - *i.e.* makes a copy of the string "ord"

Old-Style String Formatting: Conversion Modifiers

- Conversion specifications describe the type of conversion to do
  - **%i** *or* **%d** – integer (decimal) value
  - **%f** – floating-point (real) value
  - **%s** – string value
  - **%c** – integer, or single character
  - **%x** – integer again, as a hexadecimal value
- Modifiers affect how the conversion is done
  - field width, leading/padding characters
  - number of decimal places
  - left- *or* right-justification

Example: Multiplication Table Revisited

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Mon Mar 29 11:09:40 2021
4
5  @author: rmontant
6  """
7
8  maxrows = int(input('Number of rows? '))
9  maxcols = int(input('Number of columns? '))
10
11 for r in range(1, (maxrows + 1)):
12
13     for c in range(1, (maxcols +1)):
14         p = r * c
15
16         print('%4d' % (p), end='')
17
18     print()
19 print('Done.')
20
```

Example: Word Display

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Mon Mar 29 11:23:46 2021
4
5  @author: rmontant
6  """
7  filename = input('File name? ')
8
9  with open(filename, 'r') as handle:
10     contents = handle.read()
11
12 print(contents)
13
14 # Split string into a list of substrings, separated by whitespace:
15 words = contents.split()
16 print('%s contains %d words.' % (filename, len(words)))
17
18 swords = sorted(words)
19
20 lengths = []
21 for w in words:
22     lengths.append(len(w))
23
24 # Column width given by '*', and determined by the maximum word-length
25 for i in range(len(swords)):
26     print('%3d  %*s' % (i, max(lengths), swords[i]))
27
```

Mapping Keys and Dictionaries

- Useful if converting many items, or converting some items multiple times
- Conversion specifiers can be labeled with *mapping keys*
- List of items-to-convert becomes a dictionary of items-to-convert
  - Items are values, matched to mapping keys

Example: Time

```
import time
ttuple = time.localtime()
timedict = {
    'yr':ttuple.tm_year,
    'mon':ttuple.tm_mon,
    'day':ttuple.tm_mday,
}
timestring = \
    '%(mon)02d/%(day)02d/%(yr)04d' \
    % timedict
print(timestring)
```

String Methods

- *Methods* are functions that "belong" to another object
  - String methods are specialized functions that operate on whatever string they're attached to
- We've seen:
  - **`.split(`** *`separator`* **`)`** – splits a string into pieces, separated by the '*`separator`*' character(s)

Some Useful String Methods

- **.find(sub)**
  - locate a substring sub within a string
  - returns -1 if sub isn't found
- **.replace(old, new, count)**
  - replaces occurrences of old substring with new
  - optional count argument limits number of substitutions
- **.rfind(sub)**
  - like .find( ), but works from the end forwards
- **.count(sub)**
  - counts number of occurrences of sub

Example – finding URLs in a logfile

- Download, open, read file https://montcs.bloomu.edu/Datasets/Logfiles/access.log.1
- Split into lines
  - or just use .readlines()
- Find lines containing 'GET' and 'HTTP'
- Extract substrings *between* 'GET' and 'HTTP'
  - Add to a list
  - Add to a dictionary, with a count
  - Add to a dictionary, with a list of lines
- Report

Tests and Comparisons

- Boolean tests for strings and substrings
  - **==** tests for equality
  - **is** tests for identity
    - Not usually useful
- Methods to check for categories of strings
  - Return True or False depending on string's contents
  - *(see next slide)*

String-Testing Methods

Method	Behavior
.isdigit()	returns True if all characters are the numbers 0-9
.isalpha()	returns True if all characters are alphabetic a-z or A-Z
.isalnum()	returns True if all characters in the string are lowercase or uppercase letters, or the numbers 0-9
.islower()	returns True if all characters are lowercase letters
.isupper()	return True if all cased characters are uppercase letters
.isspace()	return True if all characters are whitespace
.startswith(x)	return True if the string starts with x
.endswith(x)	return True if the string ends with x

String Modifications

- Methods to return altered versions of strings:

Method	Behavior
`.strip()`	returns string with leading and trailing whitespace removed
`.lstrip()`, `.rstrip()`	returns string with left-side / right-side whitespace removed
`.lower()`	returns string with all alphabetic characters converted to lowercase
`.upper()`	returns string with all alphabetic characters converted to uppercase
`.capitalize()`	returns string with first character uppercase, remainder lowercase
`.title()`	returns string with each "word" capitaiized

.split(), .join()

- **`.split(`** separator **`)`**
  - splits a string into a *list* of substrings that were originally separated by separator
  - separator can be any character
  - default separator is "any whitespace"
- separator**`.join(`** list **`)`**
  - joins elements of a *list* of substrings, using separator
  - separator can be `' '` to join list elements as distinct words
    - inverse of **`.split()`**
  - separator can be `''` to simply concatenate all list elements into one long string

The .format() Method

- Alternative to the '%' formatting operator
- Provides more capability than '%'
- `'replacement-fields'.format(items)`
  - items elements substituted into replacement-fields
- Replacement fields:
  - contained in curly braces **{ }**
  - includes positional index
    - number or keyword argument
  - optional format modifiers follow a colon **:**
    - type, width, precision, etc.

.format( ) Examples

- Numbers:
  - `'integer: {:i}'.format( 17 )`
  - `'real number: {:f}'.format( 3.14159 )`
  - `'string: {:s}'.format( 'hello world' )`
- field-width, precision:
  - `'integer: {:5i}'.format( 17 )`
  - `'real number: {:6.3f}'.format( 3.14159 )`
  - `'string: {:10s}'.format( 'hello world' )`

.format() and Alignment

- numbers default to right-aligned within field
- strings default to left-aligned within field

- "**>**" modifier forces right-alignment in field
- "**<**" modifier forces left-alignment in field
- "**^**" modifier forces centering in field
- Examples:
  - `print( 'left: {:<10d}'.format(5) )`
  - `print( 'right: {:>10d} (default)'.format(5) )`
  - `print( 'centered: {:^10d}'.format(5) )`

.format() and Number Bases

- "**d**", "**i**" convert integers to base-10
- "**x**" converts integers to base-16 (*hexadecimal*)
- "**o**" converts integers to base-8 (*octal*)
- "**b**" converts integers to base-2 (*binary*)
- Examples:
  - `print( 'decimal: {:#10d}'.format(54321) )`
  - `print( 'hexadecimal: {:#10x}'.format(54321) )`
  - `print( 'octal: {:#10o}'.format(54321) )`
  - `print( 'binary: {:#10b}'.format(54321) )`
- "**#**" modifier tags conversions with 0x/0o/0b

Rework the last examples:

- `print('{:>12s}: {:#10d}'.format('decimal', 54321))`
- `print('{:>12s}: {:#10x}'.format('hexadecimal', 54321))`
- `print('{:>12s}: {:#10o}'.format('octal', 54321))`
- `print('{:>12s}: {:#10b}'.format('binary', 54321))`
- And one more variation:
  - "," comma separates conversions into groups of 3
- `print('{:>12s}: {:#10,}'.format('comma-grouped', 54321))`