Collection Types

zyBooks
"Programming in Python 3"
chapter 3

Collections

- Python supports data types that have internal structure – known as *collections*
  - Simple types represent a single value each
- Some collections put internal members into a definite order, or sequence
  - Strings, lists, tuples
- Some collections have special properties or characteristics
  - Dictionaries, sets

Ordered collections

- Elements of ordered collections can be selected by position
  - *indexed*
  - *sliced*

Strings

- Literals
  - A sequence of characters – lowercase and uppercase letters, digits, punctuation, etc.
    - "abcdefg"
    - "12345"
    - "qwe_rty? Uiop!"
- Single-quote characters (') or double-quote characters (") surround the literal
  - Used in pairs – ' ' or " "
  - Single quotes stylistically preferred, but either are acceptable

Strings in Variables

- Assignment – literal stored in a variable
  - `first = "O'Hare"` `# Use double-quotes to include singles`
  - `middle = 'Intl.'` `# Punctuation is just a character`
  - `last = 'Airport'`
  - `space = ' '` `# whitespace is an important character!`
- *Concatenation* and assignment
  - `chicago = first + space + middle + space + last`
- Display
  - `print( chicago )` `# what does chicago contain?`

Things to Do With Strings

- Find a string's length: use the **`len()`** function
  - `len('AbCdEfGh')` `# gives 8, the number of chars`
  - `len(chicago)` `# How long is the string in chicago?`
- Select one character from a string: use indexing –
  - `'AbCdEfGh'[0]` → `'A'` `# Indexes start at 0`
  - `'AbCdEfGh'[7]` → `'h'` `# Last char is 1 less than length`
- Select a substring: *slice* the string using indexes –
  - `'AbCdEfGh'[2:5]` → `'CdE'` `# Start : (End - 1)`
  - `print(chicago[0:4])` `# Print first 4 chars of string in chicago`

Indexing From the Front

- Positions in a string start at zero
  - so chicago**[0]** is the first character in chicago
- Last position in a string is one less than the string's length
  - chicago**[len(chicago)-1]** is the last character in chicago
- Indexed characters from a string are strings themselves

Indexing From the Back

- Negative indexes count backwards from the string's end
  - chicago**[-1]** is the last character in chicago
- Count backwards: -2 is the second to last, -3 is the third to last, etc.

try:

- Create strings with your first, last names
- How long are each of your names?
  - Print the lengths
- Use indexing to create a third string with your initials, and print that
- Concatenate your names to form your full name, and save in a string
- Use indexing to print the *last* letter of your first name, last name, and full name

Slicing *Strings into Substrings*

- Looks similar to indexing; *copies* a substring
- Example:
  - chicago**[3 : 7]**
    - Slice (substring) starts with element 3, ends *just before* element 7 (*i.e.* last element is element 6)
  - chicago**[12 : -1]**
    - Up to, but *not including*, the last character
  - chicago**[0 : len(chicago)]**
    - This "slice" is actually a copy of the entire string

Slicing Strings into Substrings

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

Immutable Strings

- "Cannot be mutated" – strings are unchangeable once created
  - print(chicago**[3]**) # is okay
  - o3 = chicago**[3]** # is okay
  - chicago**[3]** = 'A' # will fail – can't alter contents
- Make a changed *copy* instead
  - Use slices to copy the unchanged parts:
  - p0 = chicago**[: 3]** # beginning up to pos'n 3
    p1 = chicago**[4 :]** # pos'n 4 to the end

    chicago = p0 + 'A' + p1 # chicago now contains a new string

Sequences – Lists and Tuples

Sequences

- String – a sequence of characters
  - has definite order
  - elements can't be changed (immutable)
- General sequence – elements are other things than characters
  - Numbers, strings (including 1-char-long strings), other things
  - Elements may or may not be changeable
- Indexed, sliced just like strings

Lists

- A list is a sequence of elements that CAN be modified
  - Replace an element with something else
  - add elements to the list
  - Delete elements from the list
  - Elements can be dissimilar
- Often used to collect related items and keep them together
- Individual elements accessed by index
  - Similar to string operations

Examples

- [3, 6, 9, 12, 15, 17, 19]
- [ ] – empty list, a common starting point
- ['Robert', 'Montante', "Ben Franklin 209"]
- ['Abraca', 2, 4, 'Dabra', ['a', 'b', 'c'], 0]
  - List elements can even be other lists
- Jupyter examples:

```
In [49]: primes = [2,3,5,7,11,13,17,19,23]

In [50]: print(primes)
[2, 3, 5, 7, 11, 13, 17, 19, 23]

In [51]: print(len(primes), primes[0], primes[-3])
9 2 17

In [52]: print(primes[3:8])
[7, 11, 13, 17, 19]
```

Modifying a list

- **Replacing** an element is easy:

```
myname = ['R.', 'Montante']
print( myname )
myname[0] = 'Robert'  # replace initial w/ name
print( myname )
```

- **Adding** an element:

```
myname.append(', Ph.D.')
  # .append() adds to the end
myname.insert(1, 'Abdon')
  # .insert(index, value) inserts before the index pos'n
print(myname)
```

Removing from a list

- Using an item's index – **`.pop(i)`**
  - The "opposite" of the `.insert()` method
  - Defaults to last element of list
  - Returns the popped element

```
item = myname.pop(1)  # remove, return element 1
```

- Using an item's value – **`.remove(v)`**
  - No default argument, no return

- General approach for any object – **`del`**
  - Deletes anything, including list elements
  - Not the preferred way

try:

- Create a list of even numbers:
  - [2, 4, 6, 8, 10, 12]
- Add another number to the end of the list
  - Display the modified list
- Remove, display the *first* element
  - Display the modified list again
- Remove, display the first element again
  - Display the modified list again…
- Remove the *last* element
  - Display the modified list again…

Some Things to Do to a List

Operation	method or operator	Remark
list's length	len(lst)	number of elements in lst
concatenation	lstA + lstB	join lstA, lstB into a new list
location	lst.index(v)	find pos'n of value v
count	lst.count(v)	number of times value v occurs
minimum	min(lst)	numeric-only or string-only lists
maximum	max(lst)	
sum	sum(lst)	numeric-only lists

try this example:

- Add:
  - print a slice
  - *Sort* the list
  - remove an element

```
l = []  # empty list

count = 0
x = float(input('number '+str(count)+'? '))
l.append(x)

count += 1
x = float(input('number '+str(count)+'? '))
l.append(x)

count += 1
x = float(input('number '+str(count)+'? '))
l.append(x)

count += 1
x = float(input('number '+str(count)+'? '))
l.append(x)

count += 1
x = float(input('number '+str(count)+'? '))
l.append(x)

print(len(l), 'elements')
print(l)
```

Tuples

- Similar to lists, but immutable
  - as is a string
- Syntax: uses **( )** instead of **[ ]**
- Some operations,actions require immutable operands
  - tuples instead of lists
- Tuples can be changed into lists; lists can be changed into tuples

```
mytuple = tuple( [0, 1, 2, 3, 4, 5] )
mylist = list( (6, 7, 8, 9, 10) )
```

Dictionaries

User-Defined Indexes

- Dictionaries use immutable objects in place of indexes, called *keys*
- Sometimes called "Look-Up Tables"
- Syntax: create *key:value* pairs with **{ }** and **:**
- Order of *key:value* pairs is *not* guaranteed (but sometimes works)
- Example:

```
airports = { 'ORD' : "O'Hare",
             'JFK' : 'Kennedy',
             'EWR' : 'Newark',
             'PHL' : 'Philadelphia' }
```

Working with Dictionaries

- Accessing:

```
print( airports['chicago'] )
  # value keyed by the string 'chicago', similar to a list index

k = airports.keys()  # k is a list of all keys
print( k[0], airports[ k[0] ] )
  # probably the same element, but not guaranteed

contents = airports.values()  # list of all values
print( contents )
  # which is the first element?
```

- Changing:

```
airports['PHL'] = 'Philly' # just like for a list
```

Dictionary Features

- Adding elements: just use a new key, and assign the new value

```
airports['ATL'] = 'Hartsfield-Jackson'
airports['IAD'] = 'Dulles'
```

- Removing elements: use **`.pop()`** with optional default, or use **`del`**

```
chicago = airports.pop('ORD')
print(chicago)

wbs = airports.pop('AVP', 'no entry')
print(wbs)

del airports['IAD']  # no return value to print
```

Dictionary Example

```
family_tree = {}
mom = input( 'Mother's name? ' )
family_tree['mother'] = mom
dad = input('Father's name? ')
family_tree['father'] = dad
child = input( 'Child's name? ')
family_tree['children'] = [child]
     # list of children, currently one element long
print(family_tree)

child = input('Another child? ')
family_tree['children'].append(child)
     # add another child to the 'children' list
print(family_tree)
```

Converting Between Types

Compatible Types

- Conversion between compatible types is easy
  - lists ↔ tuples
  - ints → floats
  - floats → ints (with loss of fractional parts)
  - dicts → lists
    - gives keys only; better to use **`.keys()`** and **`.values()`**
- Conversion from strings to numbers is easy *if* the string looks like a number

Conversion Examples

- float(99) → 99.0
- int(3.14159) → 3
  - fractional part is lost
- int('-176') → -176
- float('-176') → -176.0
- tuple([1,2,3]) → (1,2,3,)
- list(('abc',)) → ['abc']
  - single-element tuple to single-element list

Converting To Strings; Formatted Output

Converting Objects to Strings

- Basic conversion: use **`str()`** or **`repr()`**
  - This produces what you see when you **`print()`** the object
  - Example:
    str(3.14159)
  - repr(airports)
- *Formatted* conversions – embed representations of object values within other strings
  - Includes control over values' appearances
  - Good for formatted output

String Formatting the New Way

- Similar to the old way, but uses **`.format()`** method instead of "%" operator
- Conversion specifiers are "{:}" sequences instead of "%" specifiers
  - More extensive "{:}" sequences support more different object types
- Simple example:

```
popmsg = 'Community: {:s} ; population: {:d}'
print( popmsg.format('Bloomsburg', 14586) )
```

New-Style Format Specifiers

- Specifiers are "{...}" sequences within string
  - Unrelated to dictionaries...
  - Each "{}" specifier matches one object-to-be-converted-to-string
- Optional characters within "{}" braces control object appearance
  - Field width, alignment, data type
  - Numeric representation
    - exponential, hexadecimal, binary, octal, etc.

Basic Elements of Format Specifiers

presentation type	usage	remark
blank	{}	Display as appropriate
s	{:s}	Display as string
d	{:d}	Display number as base-10 integer
x, X	{:x}	Display as hexadecimal integer
o	{:o}	Display as octal integer
b	{:b}	Display as binary integer
c	{:c}	Display as Unicode/ASCII character
n	{:n}	Display as base-10 integer, with separators

Basic Elements of Format Specifiers

- Floating-point / fixed-point numbers

presentation type	usage	remark
f	{:f}	Display number as fixed-point
e, E	{:e}	Display number as exponential
g, G	{:g}	Display as fixed-point or exponential as appropriate
%	{:%}	Display number as percentage

modifier	usage example	remark
.precision	{:.5d}	Display number with precision fractional digits

Format Specifier Modifiers

- Field width, grouping digits in numbers

Width and Grouping	usage example	remark
number	{:16d}	Display object in number-wide field
_ *or* ,	{:16,d} *or* {:16_b}	Display object in groups of 3 or 4 digits, separated by ',' or '_'

- Alignment within field:

Alignment	usage example	remark
<	{:<16s}	Left-justify object in field
>	{:16>s}	Right-justify object in field
^	{:^16s}	Center object in field

New-Style Formatting Examples

```
v = 12503
print('{:12d} {:<12d} {:^,d}'.format(v, v, v))
print('{:x}  {:>20_b}'.format(v, v))

v = 98765.43210123456
print('{:12.6f} {:<.16f}'.format(v, v))
print('{:.4e} {:12.6n} {:20g}'.format(v, v, v))
print('{:<12.3f}\n{:>12.2f}'.format(v, v))

v = 'abc'
print('{:<9s}\n{:^9s} \n{:>9s}'.format(v,v,v))
```

String Formatting the Old Way

- Use "%" formatting expression with *conversion specifiers*
- Formatting string contains arbitrary desired text, and "%" conversion-specifiers that match up with with object values
  - Works best for numeric and string objects
- "%" operator combines the formatting string with a list of objects being formatted

"%" Formatting Examples

```
"Date: %s %d, %d" % ("June", 30, 2018)

'Pi: %f  --- e: %f' % (3.14159, 2.71828)

k = 'EWR'
name = airports[k]
message = '%s airport code: %s' %(name, k)
print(message)

popmsg = 'Community: %s ; population: %d'
print( popmsg % ('Bloomsburg', 14586) )
```

Conversion Specifiers

Specifier	Matches object of type / appearance of result
%s	string
%f	floating-point (or integer) number
%e *or* %E	floating-point / in exponential notation
%d	integer
%x *or* %X	integer / in hexadecimal notation
%r	any object / uses **`repr()`** output

Conversion Specifiers – Appearance Controls

- Optional additional items in specifiers control width of field, placement within field, number of decimal places in floating-point numbers

```
print( ".%d.  .%5d.  .%8d." % (99, 99, 99) )

print(".%-12d.  .%12d." %(-99, -99))

print(".%-12s.  .%12s." %('Bob', 'Bob'))

print("%3.1f  %10.8f" % (3.14159, 3.14159))
```

Binary Numbers

Number bases

- Base 10 ("decimal"): each digit in a number is a power of 10
  - $12{,}345_{10}$ ➔ $1\times10^4 + 2\times10^3 + 3\times10^2 + 4\times10^1 + 5\times10^0$
  - 10 possible digits: 0 1 2 3 4 5 6 7 8 9
- Base 2 ("binary"): each digit is a power of 2 instead
  - Only 2 possible digits: 0 1 – known as ***bits***
  - 11011_2 ➔ $1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 1\times16 + 1\times8 + 0\times4 + 1\times2 + 1\times1$
    $= 27_{10}$

Binary Example

```
#!/usr/bin/env python3
import pprint

#Set up a dictionary of information:
person = {}
person['name'] = input( 'Your name? ' )
person['age'] = int(input( 'Your age? '))

print( person )
pprint.pprint( person )

#print a report:
fmt = '{:s} is {:d} years old, or {:b} in binary'
print( fmt.format(person['name'], person['age'], person['age']) )
```

binary format - looks like what the computer uses

Binary Numbers and Computers

- Computer circuits take on two values
  - off / on, False / True, 0 / 1
- Computer circuits actually use bits, *i.e.* binary
- Many computing quantities are measured in powers of 2
  - *e.g.* 256 colors = 2^8 colors
  - Screen resolution: $1024 \times 768 = 2^{10} \times (2^9 + 2^8)$
- *Base 16 (hexadecimal) and base 8 (octal) are compact forms of binary*
  - easier for people to read and use
  - 4 bits / hexadecimal digit
  - 3 bits / octal digit