Data Structures

DS 8015

Outline

Lists

Basic Lists

- Finite, ordered, mutable sequence of elements

```
easy_as = [1,2,3]
# square brackets delimit lists
# commas separate elements

# Create a new list
empty = []
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5]

# Lists can contain elements of different types
mixed = [4, 5, "seconds"]

# Append elements to the end of a list
numbers.append(7) # numbers == [2, 3, 5, 7]
numbers.append(11) # numbers == [2, 3, 5, 7, 11]
```

Inspecting List Elements

```
# Access elements at a particular index
numbers[0] # => 2
numbers[-1] # => 11

# You can also slice lists - the usual rules apply
letters[:3] # => ['a', 'b', 'c']
numbers[1:-1] # => [3, 5, 7]
```

Nested Lists

```
# Lists really can contain anything, even other lists!
x = [letters, numbers]
x # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
x[0] # => ['a', 'b', 'c', 'd']
x[0][1] # => 'b'
x[1][2:] # => [5, 7, 11]
```

List Method Reference

```
# Extend list by appending elements from the iterable
my_list.extend(iterable)
# Insert object before index
my_list.insert(index, object)
# Remove first occurrence of value, or raise ValueError
my_list.remove(value)
# Remove all items
my_list.clear()
# Return number of occurrences of value
my_list.count(value)
# Return first index of value, or raise ValueError
my_list.index(value, [start, [stop]])
# Remove, return item at index (def. last) or IndexError
my_list.pop([index])
# Stable sort *in place*
my_list.sort(key=None, reverse=False)
# Reverse *in place*.
my_list.reverse()
```

General Queries on Iterables

```
# Length (len)
len([]) # => 0
len("python") # => 6
len([4,5,"seconds"]) # => 3

# Membership (in)
0 in [] # => False
'y' in 'python' # => True
'minutes' in [4, 5, 'seconds'] # => False
```

Dictionaries

Create a Dictionary

- Mutable map from hashable values to arbitrary objects
- Keys can be a variety of types, as long as they are hashable
- Values can be a variety of types too

```
empty = {}
type(empty) # => dict
empty == dict() # => True

a = dict(one=1, two=2, three=3)
b = {"one": 1, "two": 2, "three": 3}
a == b # => True
```

Access and Mutate

```
d = {"one": 1, "two": 2, "three": 3}

# Get
d['one'] # => 1
d['five'] # raises KeyError

# Set
d['two'] = 22 # Modify an existing key
d['four'] = 4 # Add a new key
```

GET WITH DEFAULT

```
d = {"CS":[106, 107, 110], "MATH": [51, 113]}
d["COMPSCI"] # raises KeyError
# Use get() method to avoid the KeyError

d.get("CS") # => [106, 107, 110]
d.get("PHIL") # => None (not a KeyError!)

english_classes = d.get("ENGLISH", [])
num_english = len(english_classes)
# Works even if there were no English classes
#  in our dictionary!
```

DELETE

```
d = {"one": 1, "two": 2, "three": 3}

del d["one"]
# Raises KeyError if invalid key

d.pop("three", default) # => 3
# Remove and return d['three'] or
#  default value if not in the map


d.popitem() # => ("two", 2)
# Remove and return an arbitrary (key, value) pair.
#  Useful for destructive iteration
```

DICTIONARY VIEWS

```
d = {"one": 1, "two": 2, "three": 3}
d.keys()
d.values()
d.items()
len(d.keys()) # => 3

('one', 1) in d.items()
for value in d.values():
    print(value)
keys_list = list(d.keys())

# These dictionary views are dynamic,
#  reflecting changes in the underlying dictionary!
```

COMMON DICT OPERATIONS

```
len(d)
key in d # equiv. to `key in d.keys()`
value in d.values()
d.copy()
d.clear()

for key in d: # equiv. to `for key in d.keys():`
    print(key)
```

Tuples

MOTIVATIONS FOR TUPLES

- Immutable Sequences

```
t = (1, "cat")
# Tuples are delimited by parentheses
# Elements are separated by commas
```

- Store collections of heterogeneous data

  $\Rightarrow$ Think struct- or SQL-like objects

- "Freeze" sequence to ensure hashability

  $\Rightarrow$ Tuples can be dictionary keys, but lists cannot

- Enforce immutability for fixed-size collections

TUPLE BASICS

```
fish = (1, 2, "red", "blue")
fish[0] # => 1
fish[0] = 7 # Raises a TypeError
# You can't change any elements in a tuple!

len(fish) # => 4
fish[:2] # => (1, 2)
"red" in fish # => True
# Although the usual sequence methods still work
```

Argument Packing and Unpacking

```
t = 12345, 54321, 'hello!'
print(t) # (12345, 54321, 'hello!')
type(t) # => tuple
# Comma-separated Rvalues are converted to a tuple

x, y, z = t
x # => 12345
y # => 54321
z # => 'hello!'
# Comma-separated Lvalues are unpacked automatically
```

SWAPPING VALUES

```
# Have x=5, y=6, but want to have y=5, x=6

# temporary variable
temp = x
x = y
y = temp
print(x, y) # => 6 5

# tuple packing
x, y = y, x
print(x, y) # => 6 5
# First, y, x is packed into the tuple (6, 5)
# Then, (6, 5) is unpacked into the variables x & y respectively
```

FIBONACCI SEQUENCE

```
def fib(n):
 """Prints the first n Fibonacci numbers."""
 a, b = 0, 1
 for i in range(n):
   print(i, a)
   a, b = b, a + b
```

ENUMERATE

```
for index, color in enumerate(['red','green','blue']):
    print(index, color)
# =>
# 0 red
# 1 green
# 2 blue

# This also means you should almost never use
#  for i in range(len(sequence)):
```

QUIRKS

```
empty = ()
singleton = ("value",)
plain_string = "value" # Note plain_string != singleton
len(empty) # => 0
len(singleton) # => 1

# Tuples contain (immutable) references
#  to underlying objects!
v = ([1, 2, 3], ['a', 'b', 'c'])
v[0].append(4)
v # => ([1, 2, 3, 4], ['a', 'b', 'c'])
```

Sets

MOTIVATIONS FOR SETS

- Unordered collection of distinct hashable elements
- Fast membership testing

  $\Rightarrow$ O(1) vs. O(n)
- Eliminate duplicate entries
- Easy set operations (intersection, union, etc.)

```
s = {1, 3, 4}
# Sets are delimited by curly braces
# Elements are separated by commas
# Unordered collection of distinct hashable items
```

COMMON SET OPERATIONS - 1

```
empty_set = set()
set_from_list = set([1, 2, 1, 4, 3]) # => {1, 3, 4, 2}

basket = {"apple", "orange", "apple", "pear", "banana"}
len(basket) # => 4
"orange" in basket # => True
"crabgrass" in basket # => False
# O(1) membership testing

for fruit in basket:
    print(fruit, end='/')
# => pear/banana/apple/orange/
```

COMMON SET OPERATIONS - 2

```
a = set("mississippi") # {'i', 'm', 'p', 's'}

a.add('r')
a.remove('m') # raises KeyError if 'm' is not present
a.discard('x') # same as remove, except no error

a.pop() # => 's' (or 'i' or 'p')
a.clear()
len(a) # => 0
```

COMMON SET OPERATIONS - 3

```
a = set("abracadabra") # {'a', 'r', 'b', 'c', 'd'}
b = set("alacazam") # {'a', 'm', 'c', 'l', 'z'}

# Set difference
a - b # => {'r', 'd', 'b'}

# Union
a | b # => {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

# Intersection
a & b # => {'a', 'c'}

# Symmetric Difference
a ^ b # => {'r', 'd', 'b', 'm', 'z', 'l'}
```

REWRITING IS_EFFICIENT

```
EFFICIENT_LETTERS = "BCDGIJLMNOPSUVWZ"
def is_efficient(word):
    for letter in word:
        if letter not in EFFICIENT_LETTERS:
            return False
    return True

EFFICIENT_LETTERS = set("BCDGIJLMNOPSUVWZ")
def is_efficient(word):
    return set(word) <= EFFICIENT_LETTERS

# <= checks if subset

# Is the set of letters in this word
#  a subset of the efficient letters?
```

Looping Techniques

ITEMS IN DICTIONARY

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
    print(k, v)

# =>
# gallahad the pure
# robin the brave
```

ZIP

```
#The zip() function generates pairs of entries from
# its arguments.
questions = ['name', 'quest', 'favorite color']
answers = ['Lancelot', 'To seek the holy grail', 'Blue']
for q, a in zip(questions, answers):
    print('What is your {0}? {1}.'.format(q, a))

# =>
# What is your name? Lancelot.
# What is your quest? To seek the holy grail.
# What is your favorite color? Blue.
```

REVERSE ITERATION

```
#specify the sequence in a forward direction
# and then call the reversed() function.

for i in reversed(range(1, 10, 2)):
    print(i, end=', ')

# =>
# 9, 7, 5, 3, 1,
```

SORTED ITERATION

```
#To loop over a sequence in sorted order,
# use the sorted() function which returns
# a new sorted list while leaving the source unaltered.

basket = ['pear', 'banana', 'orange', 'pear', 'apple']
for fruit in sorted(basket):
    print(fruit)
# =>
# apple
# banana
# orange
# pear
# pear
```

Comprehensions

THE USUAL WAY

```
squares = []
for x in range(100):
    squares.append(x**2)

print(squares[:5] + squares[-5:])
# [0, 1, 4, 9, 16, 9025, 9216, 9409, 9604, 9801]
```

THE ALTERNATIVE

```
#Square brackets indicate that we're building a list

[f(xs) for xs in iter]
#Loop over the specified iterable
#Apply some operation to the loop variable(s)
# to generate new list elements

[f(xs) for xs in iter if pred(xs)]
#Only keep elements that satisfy a predicate condition
```

EXAMPLES OF LIST COMPREHENSIONS

```
[word.lower() for word in sentence]
[word for word in sentence if len(word) > 8]

[(x, x ** 2, x ** 3) for x in range(10)]
[(i,j) for i in range(5) for j in range(i)]

#Be careful - "simple is better than complex"
```

YOUR TURN!

```
# how to obtain second item from the first

[0, 1, 2, 3] -> [1, 3, 5, 7]
[3, 5, 9, 8] -> [True, False, True, False]
range(10) -> [0, 1, 4, 9, ..., 81]

["apple", "orange", "pear"] -> ["A", "O", "P"]
["apple", "orange", "pear"] -> ["apple", "pear"]
["apple", "orange", "pear"] ->
[("apple", 5), ("orange", 6), ("pear", 4)]
```

OTHER COMPREHENSIONS

```
# Dictionary Comprehensions
{key_func(vars):val_func(vars) for vars in iterable}
{v:k for k, v in d.items()}

# Set Comprehensions
{func(vars) for vars in iterable}
{word for word in hamlet if is_palindrome(word.lower())}
```

Comprehensions as Higher-Level Transformations

- **Usual Focus** Modify individual elements.
- **Comprehensions** Abstract transformations.
  - Don't say how to build a collection.
  - Just describe what output you want.
- **Functional Programming** (Upcoming) Go to the extreme!