Python Programming

3.2 Data Structures

Recall List operators and functions

lists can be manipulated with operators and functions

Usage	Explanation
`x in lst`	`x` is an item of `lst`
`x not in lst`	`x` is not an item of `lst`
`lst + lstB`	Concatenation of `lst` and `lstB`
`lst*n, n*lst`	Concatenation of `n` copies of `lst`
`lst[i]`	Item at index `i` of `lst`
`len(lst)`	Number of items in `lst`
`min(lst)`	Minimum item in `lst`
`max(lst)`	Maximum item in `lst`
`sum(lst)`	Sum of items in `lst`

Built in Types

- Tuples
- Sets
- Dictionaries

Tuples

Tuples are ordered, immutable collections of elements.

The only difference between a tuple and a list is that once a tuple has been made, it can't be changed!

Tuples

Making a tuple:

a = (1, 2, 3)

Accessing a tuple:

someVar = a[0]

The syntax for access is exactly like a list. However, you can’t reassign things.

Tuples So Far

We've already used tuples without knowing it!

```
def myFunc():
    return 1, 2

def main():
    result = myFunc()
    print(result)
```

When you return multiple things and store it in a single variable, it comes back as a tuple!

Tuples So Far

Why would you want a tuple?

Sometimes it's important that the contents of something not be modified in the future.

Instead of trying to remember that you shouldn't modify something, just put it in a tuple! A lot of programming is learning to protect you from yourself.

Sets

A set is an unordered collection of elements where each element must be unique. Attempts to add duplicate elements are ignored.

Creating a set:

mySet = set([‘a’, ‘b’, ‘c’, ‘d’])

Or:

myList = [1, 2, 3, 1, 2, 3]

mySet2 = set(myList)

Note that in the second example, the set would consist of the elements {1, 2, 3}

Sets

Things we can do with a set:

```
mySet = set(['a'])

mySet.add('b')  # Adds an element
mySet.remove('b')  #Removes an element
mySet.pop()  # Removes and returns a random element
```

Sets

There is also support for combining sets.

mySet.union(someOtherSet) – this returns a new set with all the elements from both sets.

mySet.intersection(someOtherSet) – this returns a new set with all the elements that both sets had in common.

Tons more methods can be found here:

https://docs.python.org/3/tutorial/datastructures.html

User-defined indexes and dictionaries

Goal: a container of employee records indexed by employee SS#

```
>>> employee[987654321]
['Yu', 'Tsun']
>>> employee[864209753]
['Anna', 'Karenina']
>>> employee[100010010]
['Hans', 'Castorp']
```

Problems:

- the range of SS#s is huge
- SS#s are not really integers

Solution: the dictionary class `dict`

key	value
'864-20-9753'	['Anna', 'Karenina']
'987-65-4321'	['Yu', 'Tsun']
'100-01-0010'	['Hans', 'Castorp']

```
>>> employee = {
        '864-20-9753': ['Anna',
'Karenina'],
        '987-65-4321': ['Yu', 'Tsun'],
        '100-01-0010': ['Hans', 'Castorp']}
>>> employee['987-65-4321']
['Yu', 'Tsun']
>>> employee['864-20-9753']
['Anna', 'Karenina']
```

A dictionary contains
(key, value) pairs

A key can be used as an index to access the corresponding value

Properties of dictionaries

Dictionaries are not ordered

Dictionaries are mutable

- new (key,value) pairs can be added
- the value corresponding to a key can be modified

The empty dictionary is { }

Dictionary keys must be immutable

```
>>> employee = {
        '864-20-9753': ['Anna',
'Karenina'],
        '987-65-4321': ['Yu', 'Tsun'],
        '100-01-0010': ['Hans', 'Castorp']}
>>> employee
{'100-01-0010': ['Hans', 'Castorp'], '864-
20-9753': ['Anna', 'Karenina'], '987-65-
4321': ['Yu', 'Tsun']}
>>>
```

```
>>> employee = {[1,2]:1, [2,3]:3}
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    employee = {[1,2]:1, [2,3]:3}
TypeError: unhashable type: 'list'
```

Dictionary operators

Class `dict` supports some of the same operators as class `list`

```
>>> days = {'Mo':1, 'Tu':2, 'W':3}
>>> days['Mo']
1
>>> days['Th'] = 5
>>> days
{'Mo': 1, 'Tu': 2, 'Th': 5, 'W': 3}
>>> days['Th'] = 4
>>> days
{'Mo': 1, 'Tu': 2, 'Th': 4, 'W': 3}
>>> 'Fr' in days
False
>>> len(days)
4
```

Class dict does not support all the operators that class list supports

- `+` and `*` for example

Dictionary methods

Operation	Explanation
`d.items()`	Returns a view of the (key, value) pairs in `d`
`d.keys()`	Returns a view of the keys of `d`
`d.pop(key)`	Removes the (key, value) pair with key `key` from `d` and returns the value
`d.update(d2)`	Adds the (key, value) pairs of dictionary `d2` to `d`
`d.values()`	Returns a view of the values of `d`

The containers returned by `d.items()`, `d.keys()`, and `d.values()` (called views) can be iterated over

```
>>> days
{'Mo': 1, 'Tu': 2, 'Th': 4, 'W': 3}
>>> days.pop('Tu')
2
>>> days
{'Mo': 1, 'Th': 4, 'W': 3}
>>> days2 = {'Tu':2, 'Fr':5}
>>> days.update(days2)
>>> days
{'Fr': 5, 'W': 3, 'Th': 4, 'Mo': 1,
'Tu': 2}
>>> days.items()
dict_items([('Fr', 5), ('W', 3), ('Th',
4), ('Mo', 1), ('Tu', 2)])
>>> days.keys()
dict_keys(['Fr', 'W', 'Th', 'Mo', 'Tu'])
>>> >>> vals = days.values()
>>> vals
dict_values([5, 3, 4, 1, 2])
>>>
```

Dictionary vs. multi-way `if` statement

Uses of a dictionary:

- container with custom indexes
- alternative to the multi-way `if` statement

```
def complete(abbreviation):
    'returns day of the week corresponding to abbreviation'

    if abbreviation == 'Mo':
        return 'Monday'
    elif abbreviation == 'Tu':
        return 'Tuesday'
    elif
        ......
    else: # abbreviation must be Su
        return 'Sunday'
```

```
def complete(abbreviation):
    'returns day of the week corresponding to abbreviation'

    days = {'Mo': 'Monday', 'Tu':'Tuesday', 'We': 'Wednesday',
            'Th': 'Thursday', 'Fr': 'Friday', 'Sa': 'Saturday',
            'Su':'Sunday'}

    return days[abbreviation]
```

Dictionary as a container of counters

Uses of a dictionary:

- container with custom indexes
- alternative to the multi-way `if` statement
- container of counters

Problem: computing the number of occurrences of items in a list

```
>>> grades = [95, 96, 100, 85, 95, 90, 95, 100, 100]
>>> frequency(grades)
{96: 1, 90: 1, 100: 3, 85: 1, 95: 3}
>>>
```

Solution: Iterate through the list and, for each grade, increment the counter corresponding to the grade.

Problems:

- impossible to create counters before seeing what's in the list
- how to store grade counters so a counter is accessible using the corresponding grade

Solution: a dictionary mapping a grade (the key) to its counter (the value)

Dictionary as a container of counters

Problem: computing the number of occurrences of items in a list

```
>>> grades = [95, 96, 100, 85, 95, 90, 95, 100, 100]
              ^   ^    ^    ^   ^   ^   ^
```

counters

95	96	100	85	90
3	1	1	1	1

```
def frequency(itemList):
    'returns frequency of items in itemList'

    counters = {}
    for item in itemList:
        if item in counters: # increment item counter
            counters[item] += 1
        else: # create item counter
            counters[item] = 1
    return counters
```

Exercise

Implement function `wordcount()` that takes as input a text—as a string— and prints the frequency of each word in the text; assume there is no punctuation in the text.

```
def wordCount(text):
    'prints frequency of each word in text'
```

```
>>> text = 'all animals are equal but some animals are more equal than other'
>>> wordCount(text)
all      appears 1 time.
animals  appears 2 times.
some     appears 1 time.
equal    appears 2 times.
but      appears 1 time.
other    appears 1 time.
are      appears 2 times.
than     appears 1 time.
more     appears 1 time.
>>>
```

space

every word will be 8 char long

```
            print('{:8} appears {} time.'.format(word, counters[word]))
        else:
            print('{:8} appears {} times.'.format(word, counters[word]))
```

Exercise

Implement function `lookup()` that implements a phone book lookup application. Your function takes, as input, a dictionary representing a phone book, mappingtuples (containing the first and last name) to strings (containing phone numbers)

```
>>> phonebook = {
        ('Anna','Karenina'):'(123)456-78-90',
        ('Yu', 'Tsun'):'(901)234-56-78',
        ('Hans', 'Castorp'):'(321)908-76-54'}
>>> lookup(phonebook)
Enter the first name: Anna
Enter the last name: Karenina
(123)456-78-90
Enter the first name:
```

```
def lookup(phonebook):
    '''implements interactive
       phonebook dictionary''
    while True:
        first = input('Enter
        last = input('Enter t

        person = (first, last)    # construct the key

        if person in phonebook:   # if key is in dictionary
            print(phonebook[person])  # print value
        else:                     # if key not in dictionary
            print('The name you entered is not known.')
```

Stacks and Queues

Sometimes, when we use a data-structure in a very specific way, we have a special name for it. This is to make it clear how the list is to be used.

A lot of languages even provide special kinds of variables for these special cases.

Stacks and Queues

What we're going to talk about in the following slides are not always going to be python things. We are going to talk about the **idea** of what a stack and a queue is. The way we actually implement them in python will be included separately.

Queues

A queue is a special kind of list where we can only perform the following operations:

enqueue(someItem) – puts some item at the end of a queue.

dequeue() – removes and returns the item at the start of the queue.

These two operations are what makes something a queue!

Queues

What this means:

The result of only having these two operations is that the whatever you remove is always the thing that's been in the queue longest!

Imagine a queue like a line at the grocery store—whoever has been in line the longest gets to go first. enqueue() is like someone new getting line. dequeue() is the person at the front of the line checking out.

Queues are referred to as "First in first out," or "FIFO".

Queues In Python

How to use a python list as a queue:

In python, we don't have functions named enqueue and dequeue. Instead, we have insert(0, myItem), which adds something to the beginning of the list, and a function called pop(), which removes and returns the endof the list. If these are the only two things we are using to modify the list, it's a queue!

Stacks

A stack supports two operations:

push(someItem) – puts something at the end of the stack.

pop() – returns the item at the end of the stack.

Stacks

Having only these two functions gives us a behavior where whenever we remove something, it's the thing we put on the stack **most recently.**

Imagine a stack of plates. push() is analogous to putting a plate on the top of the stack, pop() is like taking that plate back off.

Stacks are referred to as "Last in first out", or "LIFO".

Stacks in Python

If we want to simulate a stack in python, we can use append() to add something to the stack, and pop() to remove something. pop() will always return exactly what append just added.

Other Data Structures (Abbreviated)

- Lists
  - Array List
  - Linked List
- Binary Trees
  - Binary Search Tree
  - Red-Black Tree
  - B-tree
- Heaps
- Hashtables
- Graphs