CSc 120
Introduction to Computer Programing II

Adapted from slides by
Dr. Saumya Debray

01-c: Python review
python review:
lists ↔ strings
Strings → lists

>>> names = "John, Paul, Megan, Bill, Mary"

>>> names

'John, Paul, Megan, Bill, Mary'

>>> names.split()

['John,', 'Paul,', 'Megan,', 'Bill,', 'Mary']

>>> names.split('n')

['Joh', 'Paul, Megan', 'Bill, Mary']

>>> names.split(',')

['John', ' Paul', ' Megan', ' Bill', ' Mary']
Strings → lists

```python
>>> names = "John, Paul, Megan, Bill, Mary"
>>> names
'John, Paul, Megan, Bill, Mary'
```

```python
>>> names.split()
['John,', 'Paul,', 'Megan,', 'Bill,', 'Mary']
```

```python
>>> names.split('n')
['John', ',', 'Paul', ',', 'Megan', ',', 'Bill', ',', 'Mary']
```

```python
>>> names.split(',)
['John', ',', 'Paul', ',', 'Megan', ',', 'Bill', ',', 'Mary']
```
Lists → strings

```python
>>> x = ['one', 'two', 'three', 'four']
```

```python
>>> 
>>> "-".join(x)
'one-two-three-four'
```

```python
>>> 
>>> "!.!".join(x)
'one!.!two!.!three!.!four'
```

`delim.join(list)`: joins the strings in list using the string `delim` as the delimiter

returns a string
String trimming

```python
g >> x = '  abc  

g >> x.strip()
'abcd'

g >> y = "Hey!!!"

g >> y.strip('!')
'Hey'

g >> z = "*%^^stuff stuff stuff%^%^**"

g >> z.strip("*%^")
'stuff stuff stuff'
```
String trimming

```python
>>> x = '  abc
d  

'  

x.strip() : removes whitespace from either end of the string x  
returns a string

>>> y = "Hey!!!"

>>> y.strip('!')  

'Hey'

>>> z = "*%^stuf
tuff stuff stuff^%%^*"

>>> z.strip('"*^%"')  

'stuff stuff stuff'
```
String trimming

\[ x \text{.strip}() : \text{removes whitespace from either end of the string } x \]

\[ x \text{.strip}() : \text{given an optional argument } string, \text{ removes any character in } string \text{ from either end of } x \]

\[ \text{rstrip()}, \text{lstrip()} : \text{similar to strip() but trims from one end of the string} \]
EXERCISE

>>> text = "Bear Down, Arizona. Bear Down, Red and Blue."  
create a list of words with no punctuation

>>> words = text.split()  

['Bear', 'Down,', 'Arizona.', 'Bear', 'Down,', 'Red', 'and', 'Blue.']

>>> words_lst = []

>>> for w in words:
    words_lst.append(w.strip(".,"))

>>> words_lst
['Bear', 'Down', 'Arizona', 'Bear', 'Down', 'Red', 'and', 'Blue']

>>>
python review:
reading user input II:
file I/O
suppose we want to read (and process) a file "this_file.txt"
Reading user input II: file I/O

>>> infile = open("this_file.txt")

>>> for line in infile:
    print(line)

line 1 line 1 line 1
line 2 line 2
line 3 line 3

• open() the file
• read and process the file
Reading user input II: file I/O

```python
>>> infile = open("this_file.txt")
>>> for line in infile:
    print(line)

line 1 line 1 line 1
line 2 line 2
line 3 line 3
```
Reading user input II: file I/O

```python
>>> infile = open("this_file.txt")

>>> for line in infile:
    print(line)

line 1 line 1 line 1 line 1

line 2 line 2

line 3 line 3

>>>```

- `fileobj = open(filename)`
  - `filename`: a string
  - `fileobj`: a file object
- `for var in fileobj`:
  - reads the file a line at a time
  - assigns the line (a string) to `var`
Reading user input II: file I/O

```python
>>> infile = open("this_file.txt")
```
Reading user input II: file I/O

```python
>>> infile = open("this_file.txt")

>>> for line in infile:
    print(line)

line 1 line 1 line 1

line 2 line 2

line 3 line 3

>>>  
```

At this point we've reached the end of the file and there is nothing left to read
Reading user input II: file I/O

```python
>>> infile = open("this_file.txt")
>>> 
>>> for line in infile:
    print(line)

line 1 line 1 line 1
line 2 line 2
line 3 line 3

>>> 
>>> infile.close()
>>> 
>>> infile = open("this_file.txt")
```

at this point we've reached the end of the file so there's nothing left to read to re-read the file, we have to close it and then re-open it
Reading user input II: file I/O

```python
>>> infile = open("this_file.txt")

>>> for line in infile:
    print(line.strip())
```

line 1 line 1 line 1
line 2 line 2
line 3 line 3

NOTE: we can use strip() to get rid of the newline character at the end of each line
Writing output to a file

```python
>>> out_file = open("names.txt", "w")
>>> name = input("Enter a name: ")
Enter a name: Tom
>>> out_file.write(name + '
')
4
>>> name = input("Enter a name: ")
Enter a name: Megan
>>> out_file.write(name + '
')
6
>>> out_file.close()
>>> ```
Writing output to a file

```python
>>> out_file = open("names.txt", "w")

>>> name = input("Enter a name: ")
Enter a name: Tom

>>> out_file.write(name + '\n')
4

>>> name = input("Enter a name: ")
Enter a name: Megan

>>> out_file.write(name + '\n')
6

>>> out_file.close()
```
Writing output to a file

```python
>>> in_file = open("names.txt", "r")
>>> for line in in_file:
    print(line)

Tom
Megan
```
python review: tuples
Tuples

>>> x = (111, 222, 333, 444, 555)
>>> x
(111, 222, 333, 444, 555)
>>> x[0]
111
>>> x[2]
333
>>> x[-1]
555
>>> x[-2]
444
>>> a tuple is a sequence of values (like lists)
Tuples

a tuple is a sequence of values (like lists)

```python
>>> x = (111, 222, 333, 444, 555)
>>> x
(111, 222, 333, 444, 555)
>>> x[0]
111
>>> x[2]
333
>>> x[-1]
555
>>> x[-2]
444
>>> 
```

tuples use parens ()
- by contrast, lists use square brackets []
  - parens can be omitted if no confusion is possible
- special cases for tuples:
  - empty tuple: ()
  - single-element tuple: must have comma after the element:
    
    `(111,)`
Tuples

a tuple is a sequence of values (like lists)
tuples use parens ()
• by contrast, lists use square brackets []
  • parens can be omitted if no confusion is possible
• special cases for tuples:
  • empty tuple: ()
  • single-element tuple: must have comma after the element:
    (111,)

indexing in tuples works similarly to strings and lists
Tuples

```python
>>> x = (111, 222, 333, 444, 555)

>>> len(x)
5

>>> x[2:]
(333, 444, 555)

>>> x[:4]
(111, 222, 333, 444)

>>> x[1:4]
(222, 333, 444)
```

computing a length of a tuple: similar to strings and lists
Tuples

```python
>>> x = (111, 222, 333, 444, 555)

>>> len(x)
5

>>> x[2:]
(333, 444, 555)

>>> x[:4]
(111, 222, 333, 444)

>>> x[1:4]
(222, 333, 444)
```

computing a length of a tuple: similar to strings and lists

computing slices of a tuple: similar to strings and lists
Tuples

>>> x = (111, 222, 333, 444, 555)
>>> x
(111, 222, 333, 444, 555)

>>> y = (666, 777, 888)

>>> x + y
(111, 222, 333, 444, 555, 666, 777, 888)

>>> y * 3
(666, 777, 888, 666, 777, 888, 666, 777, 888)

+ and * work similarly on tuples as for lists and strings
Tuples

```python
>>> x = (111, 222, 333, 444, 555)
>>> for item in x:
    print(item)
111
222
333
444
555
>>> 222 in x
True
>>> 999 in x
False
```
Tuples

```python
tuples

>>> x = (111, 222, 333, 444, 555)
>>> for item in x:
    print(item)
111
222
333
444
555
>>> 222 in x
True
>>> 999 in x
False
```
Tuples

```python
>>> x = (111, 222, 333, 444, 555)
>>> x
(111, 222, 333, 444, 555)
>>> x[2]
333
>>> x[2] = 999
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in <module>
    x[2] = 999
TypeError: 'tuple' object does not support item assignment
```
Sequence types: mutability

tuples are immutable

```python
>>> x = ( ['aa', 'bb'], ['cc', 'dd'], ['ee'] )
>>> x[0] = 'ff'
Traceback (most recent call last):
  File "<pyshell#108>" , line 1, in <module>
    x[0] = 'ff'
TypeError: 'tuple' object does not support item assignment
```
Sequence types: mutability

```python
>>> x = ( ['aa', 'bb'], ['cc', 'dd'], ['ee'] )
``` tuples are immutable

```python
>>> x[0] = 'ff'

Traceback (most recent call last):
  File "<pyshell#108>", line 1, in <module>
    x[0] = 'ff'
TypeError: 'tuple' object does not support item assignment
```

```python
>>> x[0][0] = 'ff'
```

lists are mutable

```python
>>> x
(['ff', 'bb'], ['cc', 'dd'], ['ee'])
```
Sequence types: mutability

```python
>>> x = ( ['aa', 'bb'], ['cc', 'dd'], ['ee'] )
>>> x[0] = 'ff'
```
tuples are immutable

```
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in <module>
    x[0] = 'ff'
TypeError: 'tuple' object does not support item assignment
```

```python
>>> x[0][0] = 'ff'
```
lists are mutable

```
>>> x = ( ['ff', 'bb'], ['cc', 'dd'], ['ee'] )
```

```python
>>> x[0][0][0] = 'a'
```
strings are immutable

```
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in <module>
    x[0][0][0] = 'a'
TypeError: 'str' object does not support item assignment
```
Sequence types: mutability

tuple (immutable)

list (mutable)

string (immutable)
Sequence types: mutability

```python
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> x = ( ['aaa', 'bbb'], ['ccc', 'ddd'], ['eee'] )
>>> x[0] = 'fff'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x[0] = 'fff'
TypeError: 'tuple' object does not support item assignment
>>> x[0][0] = 'fff'
>>> x
(['fff', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>> x[0][0] = 'a'
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    x[0][0] = 'a'
TypeError: 'str' object does not support item assignment
>>> |
```

- **tuple** (immutable)
- **list** (mutable)
- **string** (immutable)
EXERCISE

```python
>>> x = [ (1, 2, 3), (4, 5, 6), (7, 8, 9) ]
>>> x[0][0] = (2, 3, 4)
```

What do you think will be printed out?

```python
>>> x[0] = [ 2, 3, 4 ]
```

What do you think will be printed out?
Why use tuples?

At the implementation level, tuples are much simpler than lists:

• lists are mutable; tuples are immutable
  • this means that the implementation can process tuples without having to worry about the possibility of updates
• lists have methods (e.g., append); tuples do not have methods

⇒ Tuples can be implemented more efficiently than lists
Summary: sequence types

Sequence types include: strings, lists, and tuples

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x in s</td>
<td>True if an item of s is equal to x, else False</td>
</tr>
<tr>
<td>x not in s</td>
<td>False if an item of s is equal to x, else True</td>
</tr>
<tr>
<td>s + t</td>
<td>the concatenation of s and t</td>
</tr>
<tr>
<td>s * n or n * s</td>
<td>equivalent to adding s to itself n times</td>
</tr>
<tr>
<td>s[i]</td>
<td>ith item of s, origin 0</td>
</tr>
<tr>
<td>s[i:j]</td>
<td>slice of s from i to j</td>
</tr>
<tr>
<td>s[i:j:k]</td>
<td>slice of s from i to j with step k</td>
</tr>
<tr>
<td>len(s)</td>
<td>length of s</td>
</tr>
<tr>
<td>min(s)</td>
<td>smallest item of s</td>
</tr>
<tr>
<td>max(s)</td>
<td>largest item of s</td>
</tr>
<tr>
<td>s.index([x[, i[, j]]])</td>
<td>index of the first occurrence of x in s (at or after index i and before index j)</td>
</tr>
<tr>
<td>s.count(x)</td>
<td>total number of occurrences of x in s</td>
</tr>
</tbody>
</table>

Source: https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
python review: dictionaries
Dictionaries

• A dictionary is like an array, but it can be indexed using strings (or numbers, or tuples, or any immutable type)
  • the values used as indexes for a particular dictionary are called its keys
  • think of a dictionary as an unordered collection of key : value pairs
  • empty dictionary: {}

• It is an error to index into a dictionary using a non-existent key
Dictionaries

```python
>>> crs_units = {}
>>> crs_units['csc 110'] = 4
>>> crs_units['csc 120'] = 4
>>> crs_units['csc 352'] = 3
>>> course = 'csc 110'
>>> crs_units[course]
4
>>> crs_units
{'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
```
Dictionaries

```python
>>> crs_units = {}
>>> crs_units['csc 110'] = 4
>>> crs_units['csc 120'] = 4
>>> crs_units['csc 352'] = 3

>>> course = 'csc 110'
>>> crs_units[course]
4

>>> crs_units
{'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
```
Dictionaries

```python
>>> crs_units = {}
>>> crs_units['csc 110'] = 4
>>> crs_units['csc 120'] = 4
>>> crs_units['csc 352'] = 3
>>> course = 'csc 110'

>>> crs_units[course]
4
```

empty dictionary

populating the dictionary

- in this example, one item at a time

looking using keys (indexing)
Dictionaries

```python
>>> crs_units = {}
>>> crs_units['csc 110'] = 4
>>> crs_units['csc 120'] = 4
>>> crs_units['csc 352'] = 3
>>> course = 'csc 110'

>>> crs_units[course]
4
>>> crs_units
{'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
```

empty dictionary

populating the dictionary
- in this example, one item at a time

looking using keys (indexing)
- we can populate it using this syntax
empty dictionary
populating the dictionary
• in this example, one item at a time
looking up the dictionary (indexing)
looking at the dictionary
• we can use this syntax to populate the dictionary too

indexing with a key not in the dictionary is an error (KeyError)
Dictionaries

- initializing the dictionary
- in this example, several items at once
Dictionaries

initializing the dictionary
• in this example, several items at once

getting a list of keys in the dictionary
• useful since it’s an error to index into a dictionary with a key that is not in it
We can use a **for** loop to iterate through a dictionary.
Dictionaries

We can use a `for` loop to iterate through a dictionary.

Notice that this iteration may not list the items in the dictionary in the same order as when they were inserted.
EXERCISE

```python
>>> crs_units = { 'csc 352' : 3, 'csc 120': 4, 'csc 110': 4 }
>>> for crs in crs_units:
    print( "{0} : {1} units".format( crs, crs_units[crs] ) )
```

csc 110 : 4 units
csc 120 : 4 units
csc 352 : 3 units

How can we get the dictionary contents to be printed out in sorted order of the keys? (I.e., what goes in the box?)