

CSc 120

Introduction to Computer Programming 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, Mega', ' ', 'Bill, Mary']
```

```
>>>
```

```
>>> names.split(',')
```

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

```
>>>
```

split() : splits a string on whitespace
returns a list of 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, Mega', ', Bill, Mary']
```

```
>>>
```

```
>>> names.split(',')
```

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

```
>>>
```

`split()` : splits a string on whitespace
returns a list of strings

`split(delim)` :
delim, splits the string
on *delim*

Lists → strings

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

```
>>>
```

```
>>> "-".join(x)
```

```
'one-two-three-four'
```

```
>>>
```

```
>>> "!.".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

```
>>> x = '  abcd  '
```

```
>>>
```

```
>>> x.strip()
```

```
'abcd'
```

```
>>>
```

```
>>> y = "Hey!!!"
```

```
>>>
```

```
>>> y.strip("!")
```

```
'Hey'
```

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

```
>>>
```

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

```
'stuff stuff stuff'
```

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

returns a string

String trimming

```
>>> x = '  abcd  '
```

```
>>>
```

```
>>> x.strip()
```

```
'abcd'
```

```
>>>
```

```
>>> y = "Hey!!!"
```

```
>>>
```

```
>>> y.strip("!")
```

```
'Hey'
```

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

```
>>>
```

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

```
'stuff stuff stuff'
```

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

returns a string

`x.strip(string)` : given an optional argument *string*, removes any character in *string* from either end of `x`

String trimming

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

`x.strip(string)` : given an optional
argument *string*, removes
any character in *string* from
either end of `x`

`rstrip()`, `lstrip()` : similar to `strip()` but
trims from one end of
the string

EXERCISE

```
>>> text = "Bear Down, Arizona. Bear Down, Red and Blue."
>>> words = text.split()
>>> words
['Bear', 'Down,', 'Arizona.', 'Bear', 'Down,', 'Red', 'and', 'Blue.']
>>> words_lst = []
>>> for w in words:
    words_lst.append(w.strip(",."))

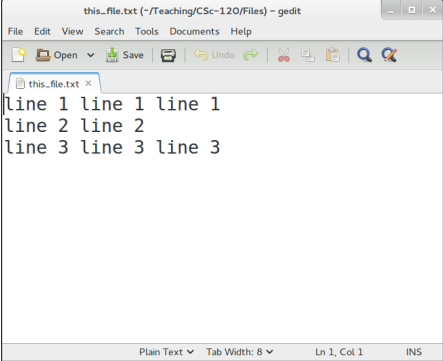
>>> words_lst
['Bear', 'Down', 'Arizona', 'Bear', 'Down', 'Red', 'and', 'Blue']
>>>
```

*create a list of words with
no punctuation*

python review:
reading user input II:
file I/O

Reading user input II: file I/O

suppose we want to read
(and process) a file
"this_file.txt"



```
this_file.txt (~Teaching/CSc-120/Files) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
this_file.txt x
line 1 line 1 line 1
line 2 line 2
line 3 line 3 line 3
Plain Text Tab Width: 8 Ln 1, Col 1 INS
```

Reading user input II: file I/O

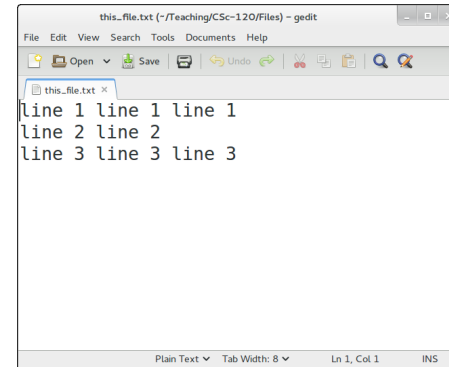
```
>>> infile = open("this_file.txt")
>>>
>>> for line in infile:
    print(line)
```

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

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

```
>>>
```

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

- *fileobj = open(filename)*
 - *filename*: a string
 - *fileobj*: a file object

```
line 1 line 1 line 1
```

```
line 2 line 2
```

```
line 3 line 3
```

```
>>>
```

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
```

```
>>>
```

- *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

```
>>> 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
```

```
>>>
```

- ***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*

Note that each line read ends in a newline ('\n') character

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

```
>>>
```

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

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

```
>>>
```

```
>>> for line in infile:
```

```
    print(line)
```

```
line 1 line 1 line 1
```

at this point we've reached the end of the file so there's nothing left to read

```
line 2 line 2
```

to re-read the file, we have to close it and then re-open it

```
line 3 line 3
```

```
>>>
```

```
>>> infile.close()
```

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

Reading user input II: file I/O

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

```
>>>
```

```
>>> for line in infile:
```

```
    print(line.strip())
```

NOTE: we can use `strip()` to get rid of the newline character at the end of each line

```
line 1 line 1 line 1
```

```
line 2 line 2
```

```
line 3 line 3
```

```
>>>
```

Writing output to a file

```
>>> 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()
```

```
>>>
```

open(filename, "w") : opens filename in write mode, i.e., for output

Writing output to a file

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

```
>>>
```

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

`open(filename, "w")` : opens *filename* in write mode, i.e., for output

```
Enter a name: Tom
```

```
>>>
```

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

`fileobj.write(string)` : writes string to fileobj

```
4
```

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

```
Enter a name: Megan
```

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

```
6
```

```
>>> out_file.close()
```

```
>>>
```

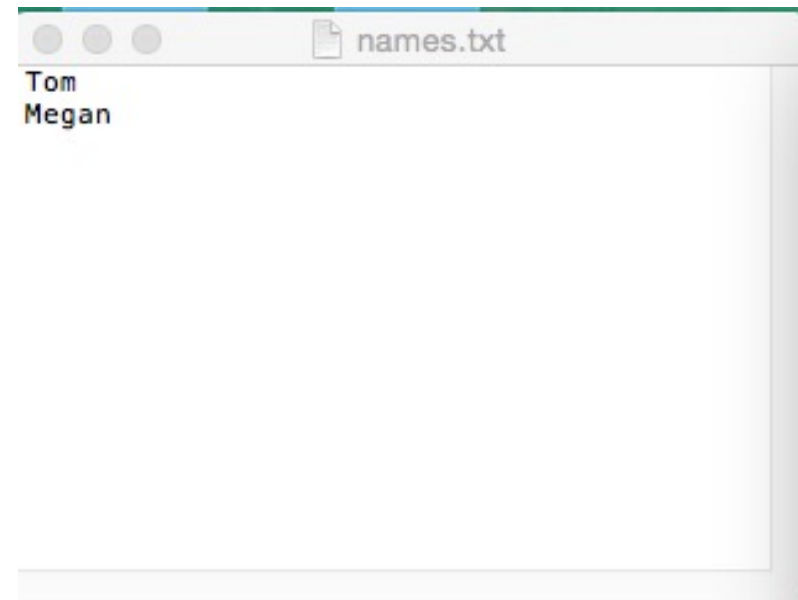
Writing output to a file

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

open the file in read mode ("r")
to see what was written

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

```
>>>
>>> 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)

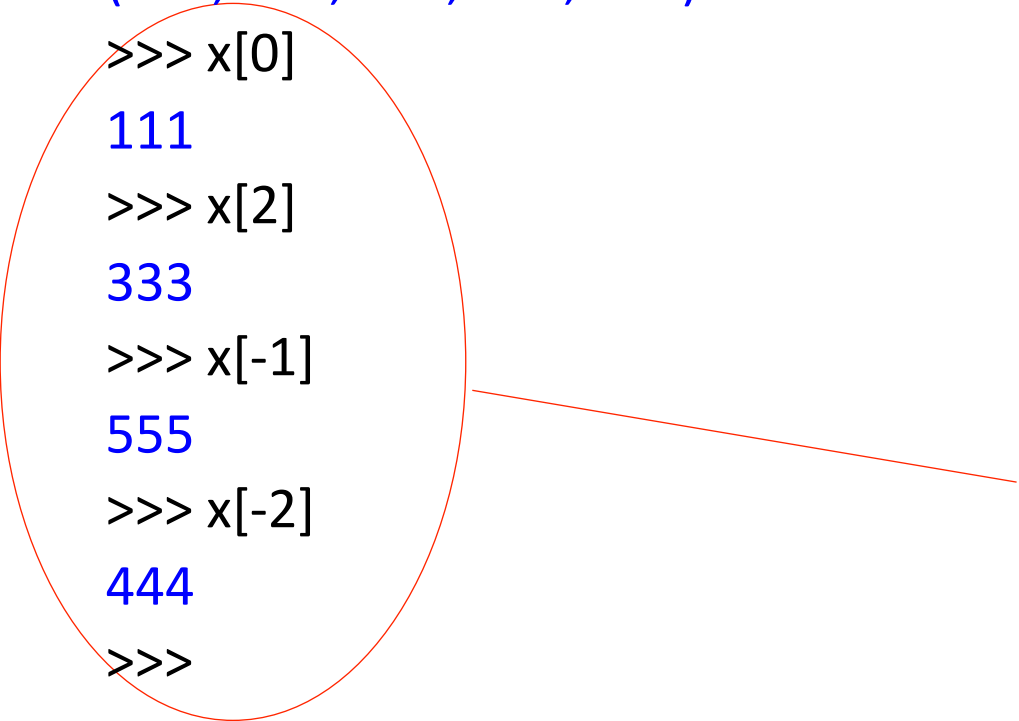
tuples use parens ()

- 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

```
>>>
>>> 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 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

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

```
>>>
```

```
len(x)
```

```
5
```

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

```
>>> x[2:]
```

```
(333, 444, 555)
```

```
>>>
```

```
>>> x[:4]
```

```
(111, 222, 333, 444)
```

```
>>> x[1:4]
```

```
(222, 333, 444)
```

```
>>>
```

```
>>>
```

Tuples

```
>>> 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

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

```
>>> for item in x:  
    print(item)
```

iterating through the elements of a tuple: similar to lists and strings

```
111
```

```
222
```

```
333
```

```
444
```

```
555
```

```
>>>
```

```
>>> 222 in x
```

```
True
```

```
>>> 999 in x
```

```
False
```

```
>>>
```

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
```

```
>>>
```

iterating through the elements of a tuple: similar to lists and strings

checking membership in a tuple: similar to lists and strings

Tuples

```
>>> 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

```
>>>
```

tuples are not mutable



Sequence types: mutability

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

tuples are immutable

```
>>> 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

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

tuples are immutable

```
>>> 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

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

lists are mutable

```
>>> x
```

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

Sequence types: mutability

```
>>> 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

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

lists are mutable

```
>>> x
```

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

```
>>> x[0][0][0] = 'a'
```

Traceback (most recent call last):

strings are immutable

File "<pyshell#112>", line 1, in <module>

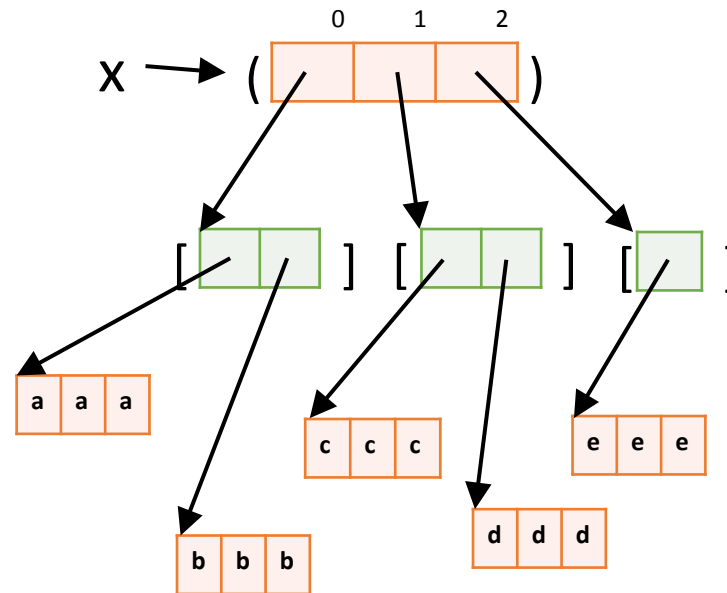
x[0][0][0] = 'a'

TypeError: 'str' object does not support item assignment

```
>>>
```

Sequence types: mutability

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
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][0] = 'a'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x[0][0][0] = 'a'
TypeError: 'str' object does not support item assignment
>>> |
```



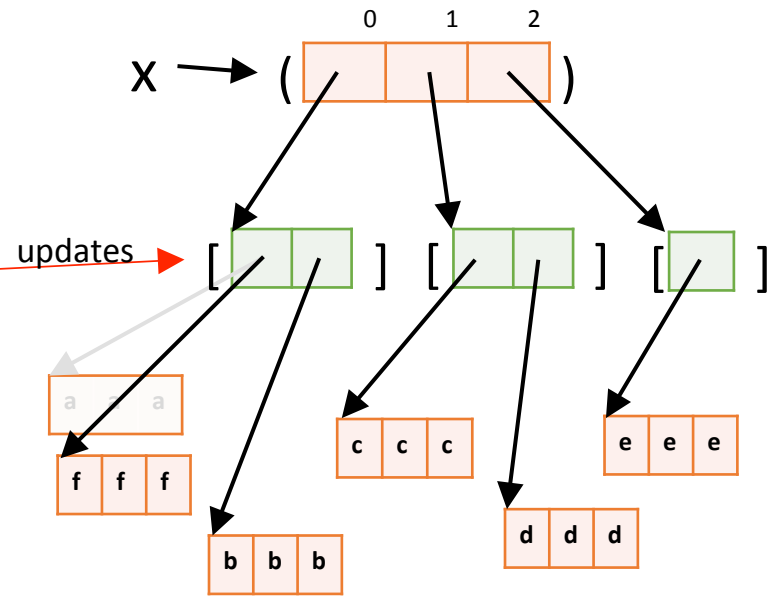
tuple
(immutable)

list
(mutable)

string
(immutable)

Sequence types: mutability

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
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][0] = 'a'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x[0][0][0] = 'a'
TypeError: 'str' object does not support item assignment
>>> |
```



tuple
(immutable)

list
(mutable)

string
(immutable)

EXERCISE

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

```
>>> x[0][0] = (2, 3, 4)
```

what do you think will be printed out?

```
>>> 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

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

The elements are: `i`, `i+k`, `i+2k`, ...

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

```
>>> crs_units = {} ————— empty dictionary
>>> 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

```
>>> crs_units = {}
```

empty dictionary

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

```
>>> crs_units['csc 120'] = 4
```

```
>>> crs_units['csc 352'] = 3
```

populating the dictionary

- in this example, one item at a time

```
>>> course = 'csc 110'
```

```
>>>
```

```
>>> crs_units[course]
```

```
4
```

```
>>> crs_units
```

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

```
>>>
```

Dictionaries

```
>>> crs_units = {}
```

empty dictionary

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

```
>>> crs_units['csc 120'] = 4
```

```
>>> crs_units['csc 352'] = 3
```

```
>>> course = 'csc 110'
```

```
>>>
```

```
>>> crs_units[course]
```

```
4
```

populating the dictionary

- in this example, one item at a time

looking using keys
(indexing)

```
>>> crs_units
```

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

```
>>>
```

Dictionaries

```
>>> 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

Dictionaries

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more informat
ion.
>>> 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}
>>>
>>> crs_units['mis 115']
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    crs_units['mis 115']
KeyError: 'mis 115'
>>>
```

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

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more informat
ion.
>>> crs_units = {'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
>>>
>>> crs_units['csc 110']
4
>>>
>>> list(crs_units.keys())
['csc 120', 'csc 352', 'csc 110']
>>> |
```

initializing the dictionary

- in this example, several items at once

Dictionaries

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more informat
ion.
>>> crs_units = {'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
>>>
>>> crs_units['csc 110']
4
>>>
>>> list(crs_units.keys())
['csc 120', 'csc 352', 'csc 110']
>>> |
```

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

Dictionaries

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
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.
>>> crs_units = {'csc 110':4, 'csc 120': 4, 'csc 352':3}
>>>
>>> for crs in crs_units:
>>>     print( "{0}: {1} units".format(crs, crs_units[crs]))
csc 120: 4 units
csc 352: 3 units
csc 110: 4 units
>>>
```

We can use a **for** loop to iterate through a dictionary

Dictionaries

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more informat
ion.
>>> crs_units = {'csc 110':4, 'csc 120': 4, 'csc 352':3}
>>>
>>> for crs in crs_units:
    print( "{0}: {1} units".format(crs, crs_units[crs]))

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

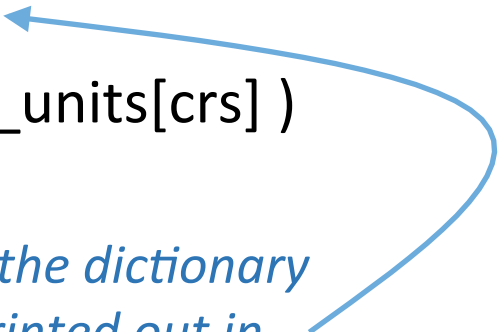
We can use a **for** loop to iterate through a dictionary

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

EXERCISE

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

```
>>> for crs in   
    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?)*