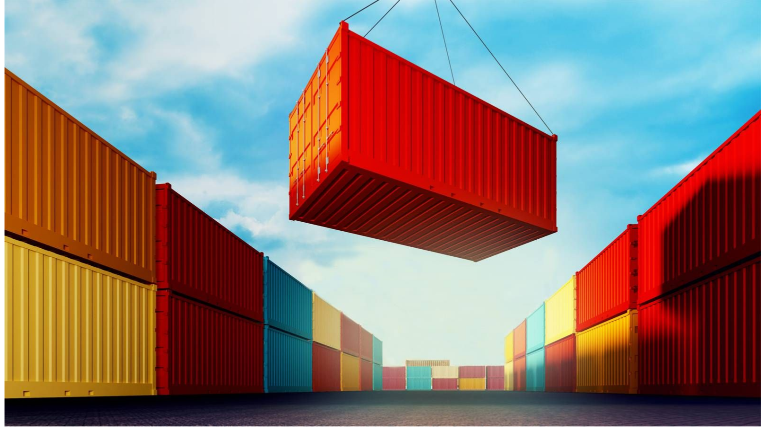
Lecture 5



Containers

Containers

- Containers are any object that holds an arbitrary number of other objects.
- Generally, containers provide a way to access the contained objects and to *iterate* over them.

Iterable (naive)

An ITERABLE Object is:

- anything that can be *looped* over (i.e. you can loop over a string or file) or
- anything that can appear on the right-side of a for-loop:

```
    for x in iterable:
        ....
```

Lists

Lists

```
>>> lst =  [ ]
>>> lst
[ ]
```

```
>>> lst = list()
>>> lst
[ ]
```

```
>>> lst =  [1, 2, ‘Marina’]  # initialization
>>> lst
[1, 2, ‘Marina’]
```

```
>>> lst2 = list (lst)  # list takes iterable objects
>>> lst2
[1, 2, ‘Marina’]
```

List manipulations

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> test[-2]
???
```

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> test[-2]
```

```
>>> ????
15
```

```
>>> test[3][1]     # or test[-1][1],  or test[-1][-2]
15
```

List manipulations

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> baby_list = test[1:3]  # list slicing
>>> baby_list
???
```

```
>>> baby_list
[‘data’, 1.4]
```

```
>>> magic = test[3] + baby_list
>>> magic
???
```

```
>>> magic
[13, 15, 1, 'fall', 1.4]
```

A few important things

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> test[1:3] = [17]      # shrinking lists. They are mutable
>>> test
[1, 17, [13, 15, 1]]
```

Definition: object is *mutable* if it can be changed after it is created

A few important things

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> test[1:3] = [17]      #shrinking lists. They are mutable
>>> test
[1, 17, [13, 15, 1]]
```

Definition: object is *mutable* if it can be changed after it is created

```
>>> lst = [1, 2, 3]
>>> new_list = lst.append(4)
>>> new_list
[1, 2, 3, 4]
```

A few important things

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> test[1:3] = [17]       #shrinking lists. They are mutable
>>> test
[1, 17, [13, 15, 1]]

>>> test[3] = [“Marina”]  #add a string to the end.
>>> test
???
```

A few important things

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> test[1:3] = [17]      #shrinking lists. They are mutable
>>> test
[1, 17, [13, 15, 1]]
```

```
>>> test[3] = [“Marina”]  #add a string to the end.
>>> test
IndexError: list assignment index out of range
```

A few important things

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> test[1:3] = [17]      #shrinking lists. They are mutable
>>> test
[1, 17, [13, 15, 1]]
>>> test[3] = [“Marina”]  #add a string to the end.
>>> test
IndexError: list assignment index out of range
>>> result = test + [“Marina”]
>>> result
[1, 17, [4], 'Marina']
```

A few important things

```
>>> test = [1, "spring", 1.4, [13, 15, 1]]
>>> test[1:3] = [17]      #shrinking lists. They are mutable
>>> test
[1, 17, [13, 15, 1]]
>>> test[3] = [“Marina”]  #add a string to the end.
>>> test
IndexError: list assignment index out of range
>>> result = test + [“Marina”] #plus returns a new list
>>> result
[1, 17, [13, 15, 1], 'Marina']
>>> ?       # in one line
['Marina', 1, 17, [13, 15, 1], 'Marina']
```

A few important things

```
>>> test = [1, "spring", 1.4, [13, 15, 1]]
>>> test[1:3] = [17]      #shrinking lists. They are mutable
>>> test
[1, 17, [13, 15, 1]]
>>> test[3] = [“Marina”]  #add a string to the end.
>>> test
IndexError: list assignment index out of range
>>> result = test + [“Marina”] #plus returns a new list
>>> result
[1, 17, [4], 'Marina']

>>> [“Marina”] + result       # in one line
['Marina', 1, 17, [4], 'Marina']
```

Difference between plus and append.

- Plus: `List1 + List2` returns a copy
- Append: `List1.append(List2)` mutates List1

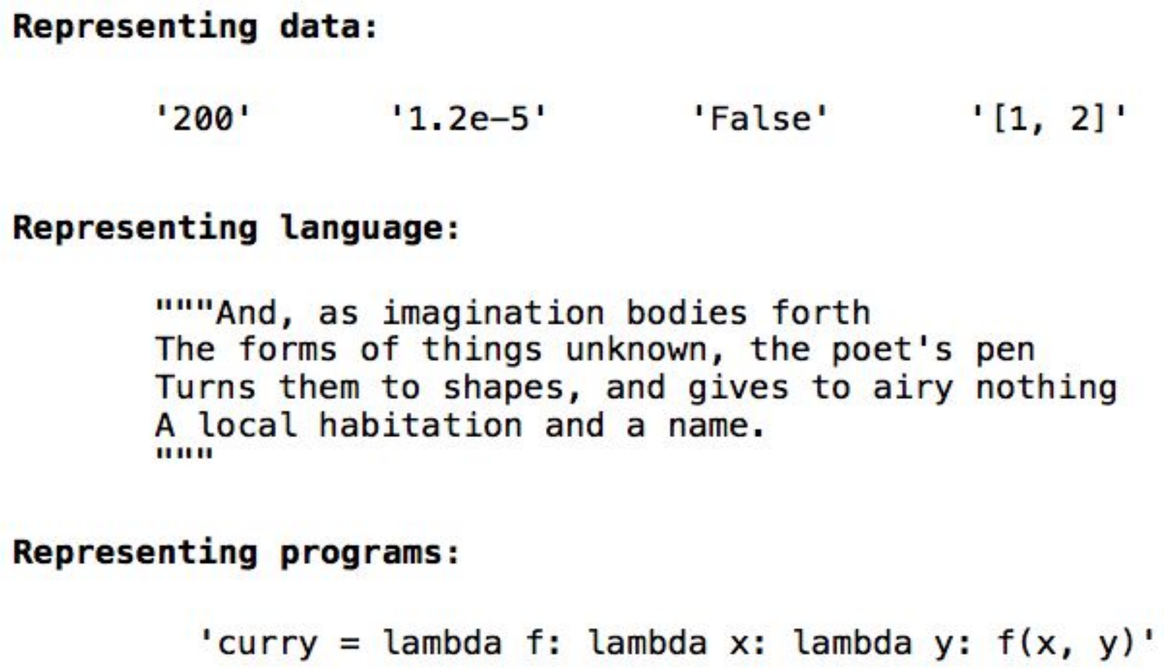
A few important things

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> test[1:3] = [17]      #shrinking lists. They are mutable
>>> test
[1, 17, [13, 15, 1]]
>>> del test[1:3]
>>> test
???
```

A few important things

```
>>> test = [1, "data", 1.4, [13, 15, 1]]
>>> test[1:3] = [17]      #shrinking lists. They are mutable
>>> test
[1, 17, [13, 15, 1]]
>>> del test[1:3]
>>> test    #del mutates the list
[1]
```

Strings



Representing data:

```
'200'      '1.2e-5'       'False'       '[1, 2]'
```

Representing language:

```
"""And, as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.
"""
```

Representing programs:

```
'curry = lambda f: lambda x: lambda y: f(x, y)'
```

Python source files are just strings.

“Escape” characters

```
>>> 'Marina Langlois'
'Marina Langlois'
```

```
>>> "doesn't"
"doesn't"
>>> 'doesn\'t'
"doesn't"
```

```
>>> '"Yes," he said.'
'"Yes," he said.'
```

```
>>> with escape?
'"Yes," he said.'
```

Notes:

Use `print` function to see the escape symbols in action:

```
print(“\a\a\a\a Marina \a\a\a\a”)
```

List of escape characters:

https://linuxconfig.org/list-of-python-escape-sequence-characters-with-examples

String Slicing:

```
>>> str = "Soon, " * 3 + "we will have a midterm"+"!" * 3
>>> str
Soon, Soon, Soon, we will have a midterm!!!
```

String Slicing:

```
>>> str = "Soon, " * 3 + "we will have a midterm"+"!" * 3
>>> str
Soon, Soon, Soon, we will have a midterm!!!

>>> str[4]
','
```

String Slicing:

```
>>> str = "Soon, " * 3 + "we will have a midterm"+"!" * 3
>>> str
Soon, Soon, Soon, we will have a midterm!!!

>>> str[4]
','

>>> str[0:4]   # last index is not inclusive
'Soon'

>>> str[33:40]
'midterm'
```

String Slicing:

```
Soon, Soon, Soon, we will have a midterm!!!
>>> str[0:4]   #last number is not inclusive
'Soon'
>>> str[33:40]
'midterm'
>>> str[33:]   #everything from 33 till the end. Works for lists too
'midterm!!!'
```

String and List Slicing:

```
Soon, Soon, Soon, we will have a midterm!!!
>>> str[0:4]   #last number is not inclusive
'Soon'
>>> str[33:40]
'midterm'

>>> str[33:]   # everything from 33 till the end
'midterm!!!'

>>> str[:10]   # from the beginning till 10 (not inclusive)
'Soon, Soon'   # Also works for the lists
```

String and List Slicing:

```
Soon, Soon, Soon, we will have a midterm!!!
>>> str[-1]                       #the last character
'!'

>>> str[-2]                       #The last-but-one character
'!'
>>> str[-2:]                      #The last two characters
'!!'
>>> str[:-2]                      #Everything except the last two characters
'Soon, Soon, Soon, we will have a midterm!'


>>> str[::-1]                     #reverse a string
'!!!mretdim a evah lliw ew ,nooS ,nooS ,nooS'
```

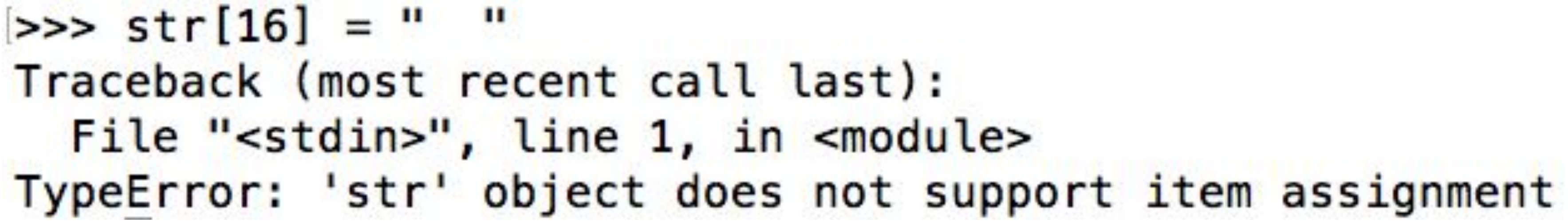
String Modification:

```
Soon, Soon, Soon, we will have a midterm!!!

>>> str[16] = “ “  #you do not like that extra comma
```

String Modification

```
Soon, Soon, Soon, we will have a midterm!!!
```

```
>>> str[16] = " "  #you do not like that extra comma
```



```
>>> str[16] = "  "
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

String are immutable: Once you assign a **string** object, that object can not be changed in memory.

Question. Output?

```
a = "Dog"
b = "eats"
c = "treats"

print (a + " " + b + " " + c)

a = a + " " + b + " " + c
print(a)
```

A:
Dog eats treats
Dog eats treats

B:
Dog eats treats
Error

C:
Dog eats treats
Dog

D: Error

Question. Output?

```
a = "Dog"
b = "eats"
c = "treats"

a = a + " " + b + " " + c
print(a)

a[0:3] = "Cat"
print(a)
```

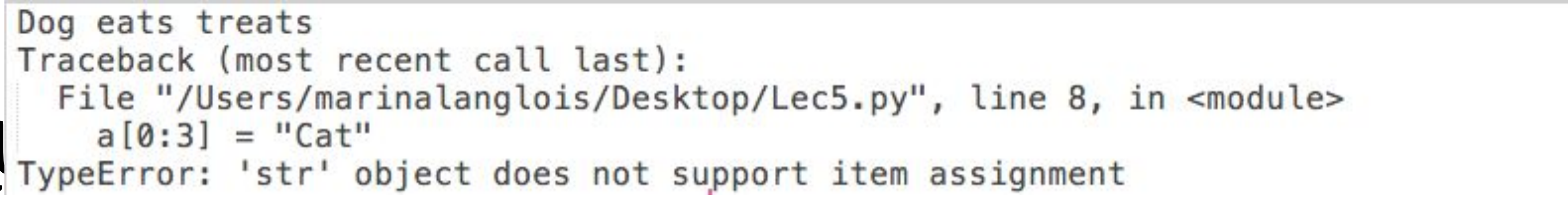
A:
Dog eats treats
Cat eats treats

B:
Dog eats treats
Error

C:
Dog eats treats
Cat

D: Error

Qu



```
Dog eats treats
Traceback (most recent call last):
  File "/Users/marinalanglois/Desktop/Lec5.py", line 8, in <module>
    a[0:3] = "Cat"
TypeError: 'str' object does not support item assignment
```

```
a = "Dog"
b = "eats"
c = "treats"

a = a + " " + b + " " + c
print(a)

a[0:3] = “Cat”
print(a)
```

A:
Dog eats treats
Cat eats treats

B:
Dog eats treats
Error

C:
Dog eats treats
Cat

D: Error

Substitution. replace

```
>>> newStr = "Marina Langlois is good."

>>> newStr.replace("good", "great")
```

Output?

Substitution. replace

```
>>> newStr = "Marina Langlois is good."

>>> newStr.replace("good", "great")

'Marina Langlois is great.'
```

Substitution. replace

```
>>> newStr = "Marina Langlois is good."

>>> newStr.replace("good", "great")

'Marina Langlois is great.'

>>> newStr

'Marina Langlois is good.'
```

;(demoted

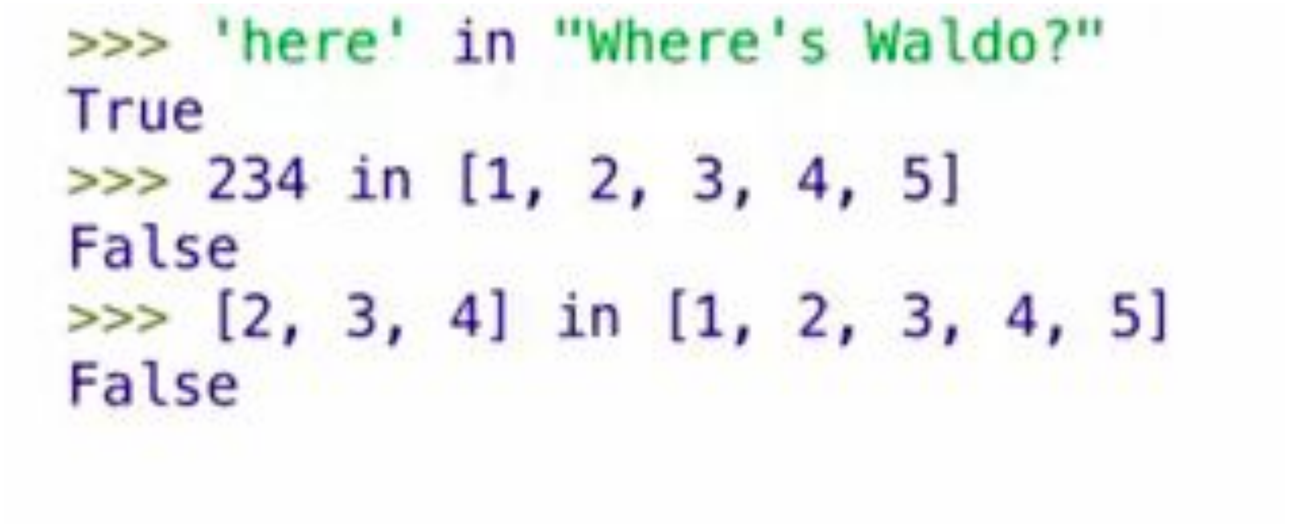
Replace does not modify newStr.

Creates a new one.

Need to save the result.

```
mlg = newStr.replace("good", "great")
```

in and not in



```
>>> 'here' in "Where's Waldo?"
True
>>> 234 in [1, 2, 3, 4, 5]
False
>>> [2, 3, 4] in [1, 2, 3, 4, 5]
False
```

Tuples

Tuples are Immutable Sequences

- Immutable values are protected from mutation

```
>>> new_y = (‘rooster’, ‘dog’)

# can use all list manipulation methods except
>>> new_y[0] = ‘pig’

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

File Input/Output

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

Basic file usage

file object = open(**file_name** [, access_mode])

- file_name – The *file_name* argument is a string value

- access_mode – The *access_mode* determines the mode in which the file has to be opened, i.e., read, write, append, etc

Basic file usage

file object = open(file_name [, access_mode])

- file_name – The file_name argument is a string value
- access_mode – The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc

```
f = open('myfile.txt','w')
f.write('this is line 1')
f.close()
```

Basic file usage

file object = open(file_name [, access_mode])

- file_name – The file_name argument is a string value
- access_mode – The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc

```
f = open('myfile.txt','w')
f.write('this is line 1')
f.close()
```

```
f=open('myfile.txt','a')
f.write('this is line 2')
f.close()
```

Basic file usage

file object = open(file_name [, access_mode])

- file_name – The file_name argument is a string value
- access_mode – The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc

```
f = open('myfile.txt','w')
f.write('this is line 1')
f.close()
```

```
f=open('myfile.txt','a')
f.write('this is line 1')
f.close()
```

```
f=open('myfile.txt','r')
f.write('this is line 1')
f.close()
```

Basic file usage

file object = open(file_name [, access_mode])

- file_name – The file_name argument is a string value
- access_mode – The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc

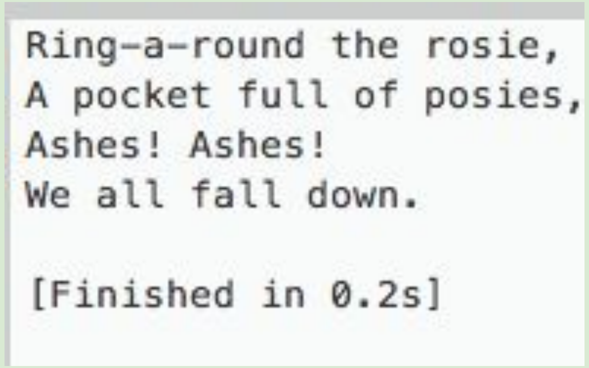
```
f = open('myfile.txt','w')
f.write('this is line 1')
f.close()
```

```
f=open('myfile.txt','a')
f.write('this is line 1')
f.close()
```

```
f=open('myfile.txt','r')
f.write('this is line 1')
f.close()
```

Printing the content: read()

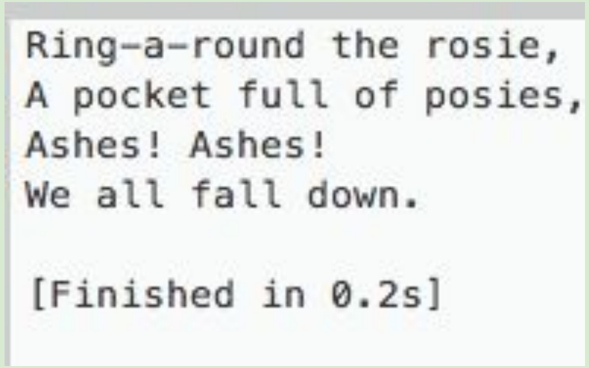
```
f=open('myfile.txt','r')
print(f.read())
f.close()
```



```
Ring-a-round the rosie,
A pocket full of posies,
Ashes! Ashes!
We all fall down.

 [Finished in 0.2s]
```

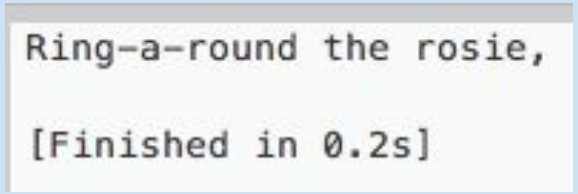
Printing the content

```
f=open('myfile.txt','r')
print(f.read())
f.close()
```



```
Ring-a-round the rosie,
A pocket full of posies,
Ashes! Ashes!
We all fall down.

[Finished in 0.2s]
```

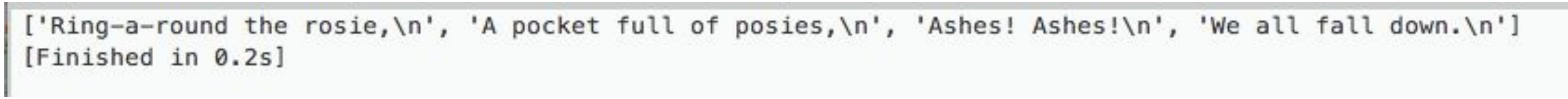
```
f=open('myfile.txt','r')
print(f.readline())
f.close()
```



```
Ring-a-round the rosie,

[Finished in 0.2s]
```

Working with the file

```
f = open('myfile.txt','r')
lines = f.readlines()
f.close()
print(lines)
```



```
['Ring-a-round the rosie,\n', 'A pocket full of posies,\n', 'Ashes! Ashes!\n', 'We all fall down.\n']
[Finished in 0.2s]
```

Working with the file

```
f = open('myfile.txt','r')
for line in f:
    print(line)
```

Proper (modern) way to open files

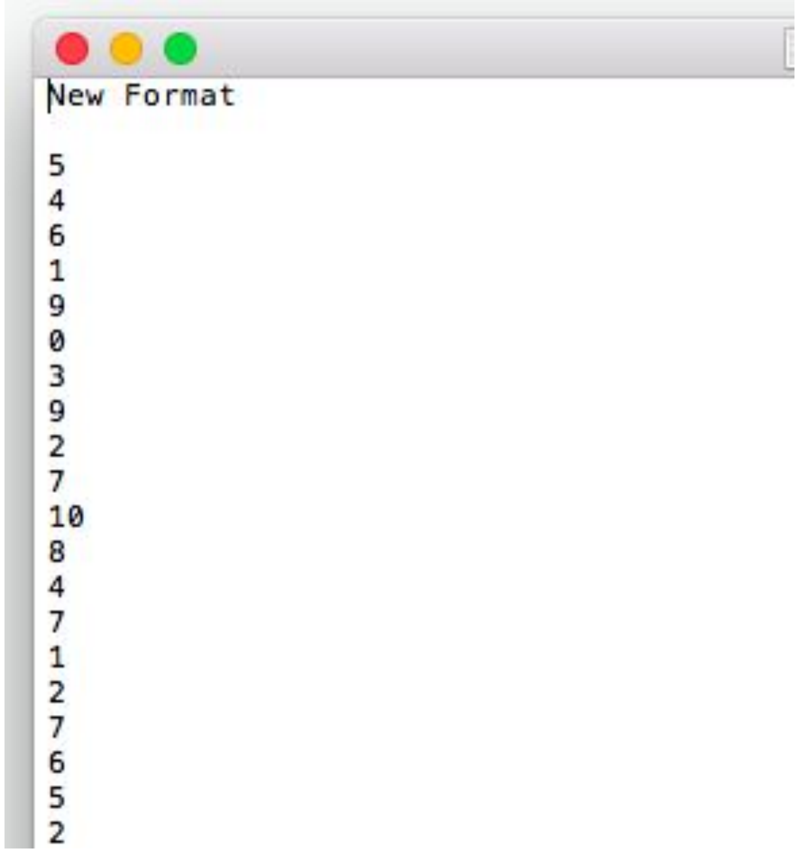
It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

```
>>> with open(“myfile.txt”, “r”) as f:
...     read_data = f.read()
>>> f.closed
True
```

Try yourself

- open( )
- close( )
- write( )
- read ( )
- “\n”

File named out.txt ---->



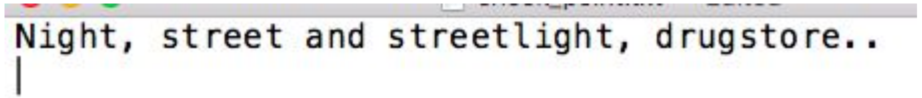
```
New Format

5
4
6
1
9
0
3
9
2
7
10
8
4
7
1
2
7
6
5
2
```

practice content: 5,4,6,1,9,0,3,9,2,7,10,8,4,7,1,2,7,6,5,2,8,2,0,1,1,1,2,10,6,2

A few possible solutions.

```
f_in = open('practice.txt','r')
f_out = open('out.txt', 'w')
f_out.write("New Format\n\n")
for line in f_in:
   line = line.replace(',','\n')
   f_out.write(line)
f_in.close
f_out.close()
```

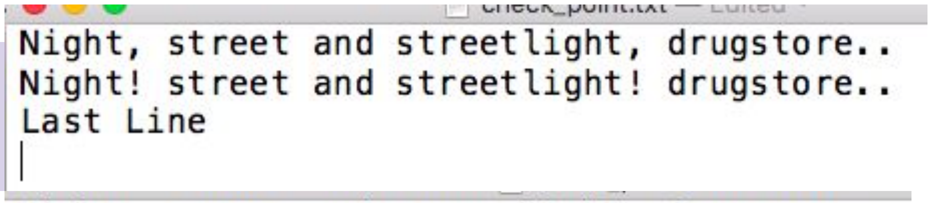
Check point



```
Night, street and streetlight, drugstore..
```

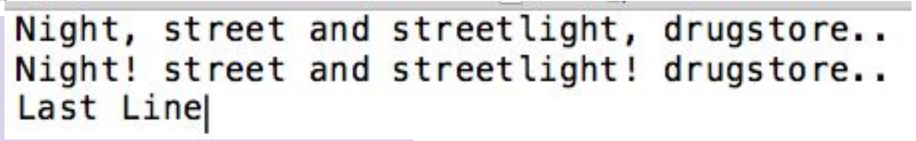
```
with open("check_point.txt", "r") as f:
    read_data = f.read()
    read_data = read_data.replace(",", "!")
    with open("check_point.txt", "a") as f:
        f.write(read_data)
        f.write("Last Line")
```

A:



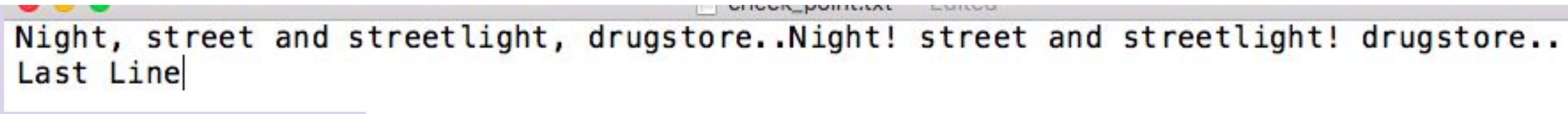
```
Night, street and streetlight, drugstore..
Night! street and streetlight! drugstore..
Last Line
```

B:



```
Night, street and streetlight, drugstore..
Night! street and streetlight! drugstore..
Last Line
```

C:

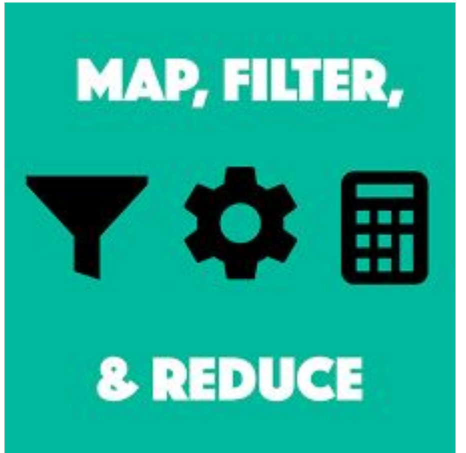


```
Night, street and streetlight, drugstore..Night! street and streetlight! drugstore..
Last Line
```

D: Error

E: Other

Map( ), Filter( ), Reduce(?)



Map( )

- `map(function_to_apply, list_of_inputs)`

- `Function_to_apply: can be a lambda function`
- `List_of_inputs: iterable`

Map()

- `map(function_to_apply, list_of_inputs)`

- `Function_to_apply: can be a lambda function`
- `List_of_inputs: iterable`

Question: double the value of each element in the list

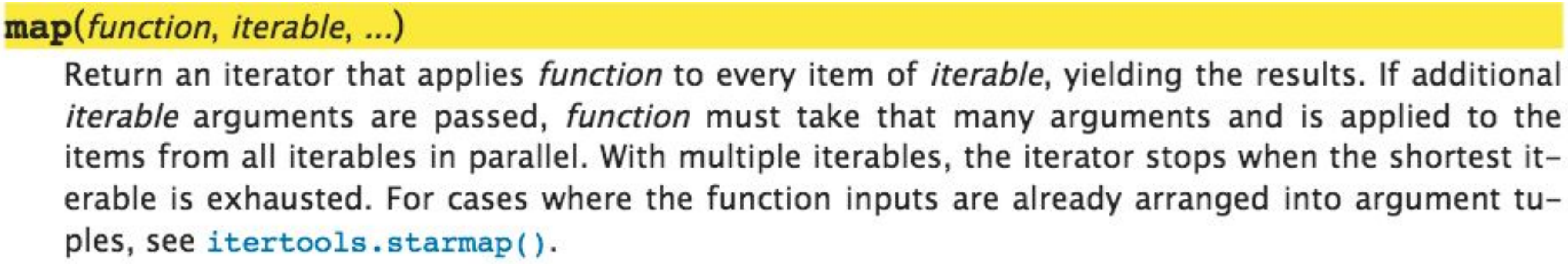
Double the value with map ( )

```
def double(x):

    return 2 * x




# map (function, list)

result = map(double, [1, 2, 3, 4])

>>> result
```

Double the value with map ( )

```
def double(x):

    return 2 * x

# map (function, list)

result = map(double, [1, 2, 3, 4])
>>> result


<map object at 0x10871c8d0>
```

Map ( )

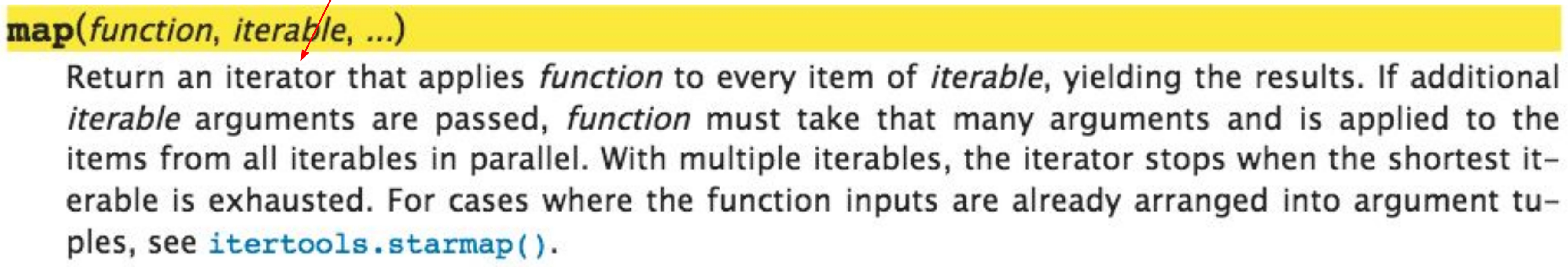


map(*function, iterable, ...*)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

https://docs.python.org/3/library/functions.html

Map ( )



map(*function*, *iterable*, ...)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

Iterator object

- Think of it as a *cursor* that goes along the list (or something iterable)

```
>>> s = "La Jolla"
>>> it = iter(s)
>>> next(it)
L
>>> next(it)
a
...
>>> next(it)   # all the way at the end
```

Iterator object

- Think of it as a cursor that goes along the list (or something iterable)

```
>>> s = "La Jolla"
>>> it = iter(s)
>>> next(it)
L
>>> next(it)
a
...
>>> next(it)
    File "/Users/marinalanglois/Desktop/Lec5.py", line 8, in <module>
        next(it)
    StopIteration
```

Iterator object

- Think of it as a cursor that goes along the list (or something iterable)

```
>>> s = "La Jolla"
>>> it = iter(s)
>>> list(it)  #gets the whole list

La Jolla
```

lazy evaluation

is useful when you have a very large data set to compute. It allows you to start using the data immediately, while the whole data set is being computed.

Double the value with map ( )

```
def double(x):

    return 2 * x

# map (function, list)

result = map(double, [1, 2, 3, 4])
>>> result


<map object at 0x10871c8d0>
```

Double the value with map ( )

```
def double(x):

    return 2 * x

# map (function, list)

result = map(double, [1, 2, 3, 4])
>>> list(result)
[2, 4, 6, 8]
```

Double the value with map ( )

```
def double(x):
    return 2 * x

result = map(double, [1, 2, 3, 4])
>>> list(result)
[2, 4, 6, 8]


result = list(map(double, [1, 2, 3, 4]))
>>> result   # converted to a list
[2, 4, 6, 8]
```

Lambda, reminder

```
>>> double = lambda x: x * 2
>>> double(4)
8
```

```
>>> (lambda x:x*2)(4)
8
```

```
>>> simple = lambda: print(“example”)
>>> simple()
example
```

Rewrite using lambda function

```
def double(x):
    return 2 * x

result = list(map(double, [1, 2, 3, 4]))  # cast to a list



result = list(map(_____________________,[1, 2, 3, 4]))
```

Rewrite using lambda function

```
def double(x):
    return 2 * x

result = list(map(double, [1, 2, 3, 4]))  # cast to a list



result = list(map(lambda x: 2 * x,  [1, 2, 3, 4]))
```

Filter( )

- `filter(function_to_apply, list_of_inputs)`

- Function_to_apply: can be a lambda function
    - Returns true/false

- List_of_inputs: iterable

Filter( )

- filter(function_to_apply, list_of_inputs)

- Function_to_apply: can be a lambda function
    - Returns true/false

- List_of_inputs: iterable

Problem:

Filter the strings with even length out.

Filter out even length strings

```
def is_even(s):

    return len(s) % 2 == 0
```

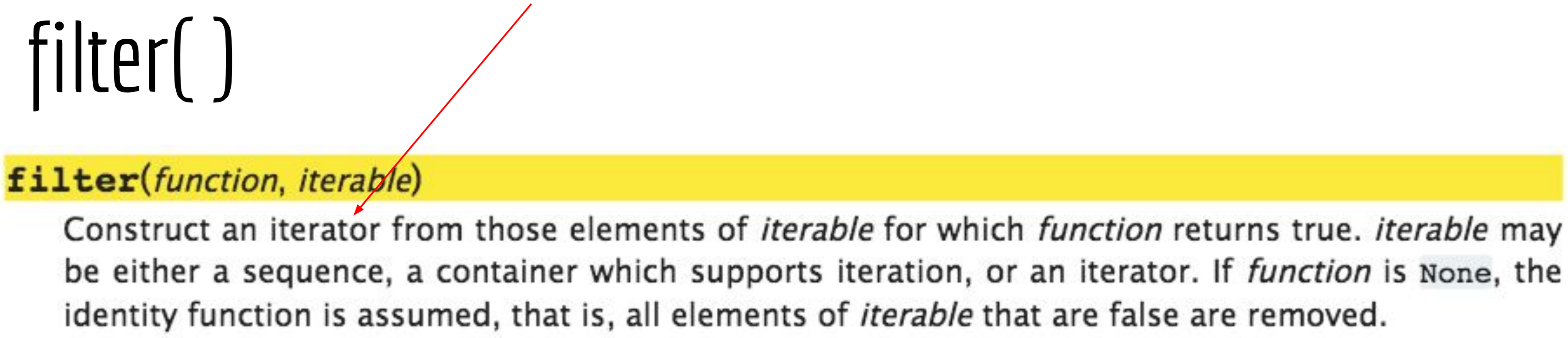
Filter out even length strings

```
def is_even(s):

    return len(s) % 2 == 0

# filter (function, list)

result = filter(is_even, [“I”, “am”, “so”, “cool”])

>>> result
```

Filter out even length strings

```
def is_even(s):

    return len(s) % 2 == 0

# filter (function, list)

result = filter(is_even, [“I”, “am”, “so”, “cool”])

>>> result

<map object at 0x10871c8d0>
```

filter( )



`filter`(*function*, *iterable*)

Construct an iterator from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Filter out even length strings

```
def is_even(s):

    return len(s) % 2 == 0

# filter (function, list)

result = list(filter(is_even, [“I”, “am”, “so”, “cool”]))

>>> result

['am', 'so', 'cool']
```

Rewrite using lambda function

```
def is_even(s):

    return len(s) % 2 == 0

result = list(filter(is_even, [“I”, “am”, “so”, “cool”]))


result = list(filter(___________________, [“I”, “am”, “so”, “cool”]))
```

Rewrite using lambda function

```
def is_even(s):

    return len(s) % 2 == 0

result = list(filter(is_even, [“I”, “am”, “so”, “cool”]))
```

```
result = list(filter(lambda s: len(s)%2 == 0,[“I”,“am”,“so”,“cool”]))
```

Question

- Is there a way in Python to call filter on a list where the filtering function has a *number of formal parameters* **bound** during the call ?

```
def func (a, b, c):
    return a + b < c
```

```
lst = [10, 20, 30, 40]
filter(func(a=10, c=35), lst) #Want to happen
```

```
def make_filter(a, c):
   def my_filter(b):
       return a + b < c
   return my_filter

filt = make_filter(10, 35)
lst = [10, 20, 30 , 40]

list(filter(filt, lst))
```

```
def func (a, b, c):
    return a + b < c

lst = [10, 20, 30, 40]

list(filter(lambda x: func(10, x, 35), lst))
```

Check point

```
lst = [1, -2, -3, 4, 5]
def func1(x):
    return x<2

it = filter(func1, lst)
print(list(it))
```

What is the output of the code shown?

A: [1, 4, 5]

B: Error

C: [-2, -3]

D: [1, -2, -3, None, None]

E: None of the above