

# Python Advanced Programming

## Unit 1: Tokenization

CHAPTER 1: BASIC TEXT PROCESSING

DR. ERIC CHOU

IEEE SENIOR MEMBER



## Python 3 Beginner's Reference Cheat Sheet

### Main data types

**boolean** = *True / False*  
**integer** = 10  
**float** = 10.01  
**string** = "123abc"  
**list** = [ value1, value2, ... ]  
**dictionary** = { key1:value1, key2:value2, ... }

### Numeric operators

**+** addition  
**-** subtraction  
**\*** multiplication  
**/** division  
**\*\*** exponent  
**%** modulus  
**//** floor division

### Comparison operators

**==** equal  
**!=** different  
**>** higher  
**<** lower  
**>=** higher or equal  
**<=** lower or equal

### Boolean operators

**and** logical AND  
**or** logical OR  
**not** logical NOT

### Special characters

**#** coment  
**\n** new line  
**\<char>** scape char

### String operations

**string[i]** retrieves character at position i  
**string[-1]** retrieves last character  
**string[i:j]** retrieves characters in range i to j

### List operations

**list = []** defines an empty list  
**list[i] = x** stores x with index i  
**list[i]** retrieves the item with index i  
**list[-1]** retrieves last item  
**list[i:j]** retrieves items in the range i to j  
**del list[i]** removes the item with index i

### Dictionary operations

**dict = {}** defines an empty dictionary  
**dict[k] = x** stores x associated to key k  
**dict[k]** retrieves the item with key k  
**del dict[k]** removes the item with key k

### String methods

**string.upper()** converts to uppercase  
**string.lower()** converts to lowercase  
**string.count(x)** counts how many times x appears  
**string.find(x)** position of the x first occurrence  
**string.replace(x,y)** replaces x for y  
**string.strip(x)** returns a list of values delimited by x  
**string.join(L)** returns a string with L values joined by string  
**string.format(x)** returns a string that includes formatted x

### List methods

**list.append(x)** adds x to the end of the list  
**list.extend(L)** appends L to the end of the list  
**list.insert(i,x)** inserts x at i position  
**list.remove(x)** removes the first list item whose value is x  
**list.pop(i)** removes the item at position i and returns its value  
**list.clear()** removes all items from the list  
**list.index(x)** returns a list of values delimited by x  
**list.count(x)** returns a string with list values joined by S  
**list.sort()** sorts list items  
**list.reverse()** reverses list elements  
**list.copy()** returns a copy of the list

### Dictionary methods

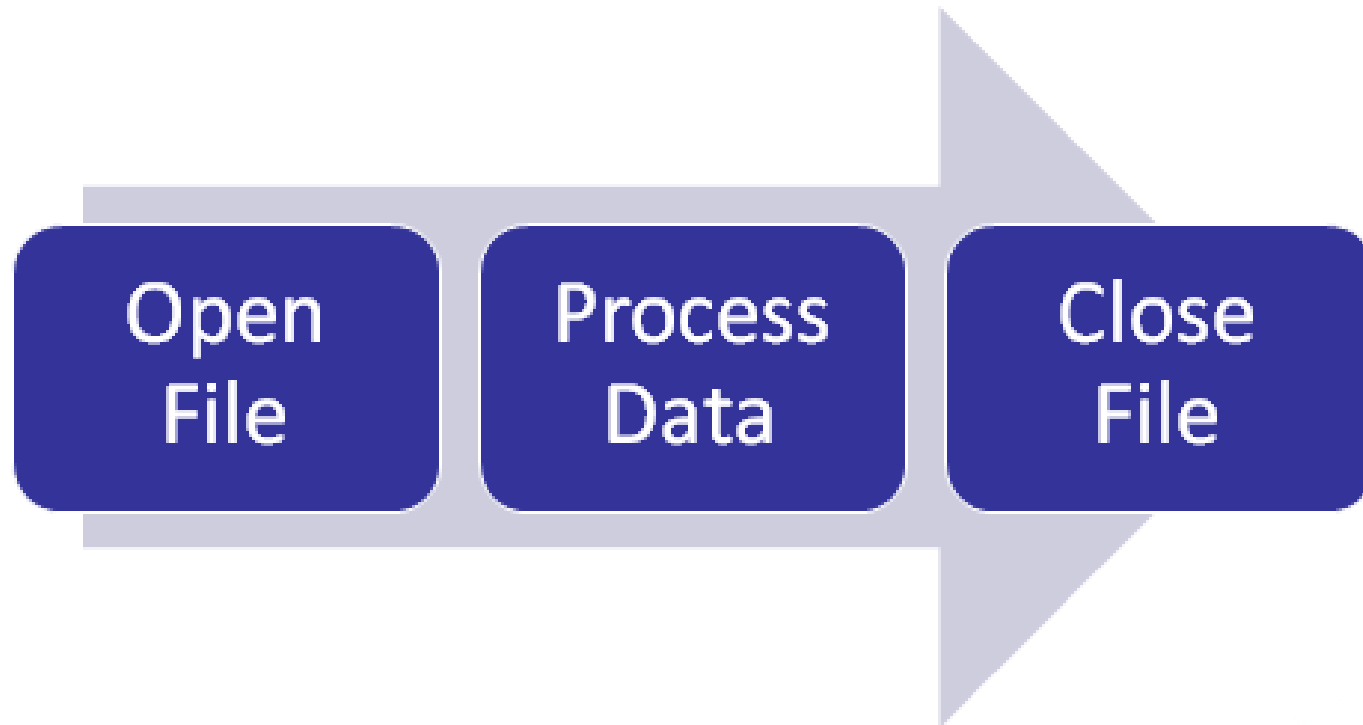
**dict.keys()** returns a list of keys  
**dict.values()** returns a list of values  
**dict.items()** returns a list of pairs (key,value)  
**dict.get(k)** returns the value associated to the key k  
**dict.pop()** removes the item associated to the key and returns its value  
**dict.update(D)** adds keys-values (D) to dictionary  
**dict.clear()** removes all keys-values from the dictionary  
**dict.copy()** returns a copy of the dictionary

**Legend:** x,y stand for any kind of data values, s for a string, n for a number, L for a list where i,j are list indexes, D stands for a dictionary and k is a dictionary key.

# Files

(Review Python OOP Chapter 2)

LECTURE 1



File  
Access

---



# Opening Files

---

- In Python the 'open()' function accepts a path to the file that you'd like to open along with a mode in which the file will be opened.
- The most commonly used modes are **read**, **write**, and **append**.
- This function creates a new **File** object which can then be iterated to extract or write information.



# Open function

---

File Object Created



Path to File



Mode



```
f = open('c:\\temp\\data.txt', 'r')
```



# Python File **Open** Operation

---

```
f = open("test.txt")
```

```
f = open("test.txt", "r")
```

```
f = open("test.txt", mode = 'r', encoding = 'utf-8')
```

file name



file access mode



file text encoding



# File Access Mode

LECTURE 2





# File Access Mode

---

read	<ul style="list-style-type: none"><li>• r</li><li>• r+ (read/write) – contents preserved</li></ul>
write	<ul style="list-style-type: none"><li>• w</li><li>• w+ (read/write) - contents deleted</li></ul>
append	<ul style="list-style-type: none"><li>• a</li><li>• a+ (read/write) – contents preserved</li></ul>
binary	<ul style="list-style-type: none"><li>• b</li><li>• Opens file in Binary mode. Addition to <u>r,w</u>, or a</li></ul>
universal	<ul style="list-style-type: none"><li>• U</li><li>• Addition to <u>r,w</u>, or a. Applies universal newline translator.</li></ul>



# File Access Mode

---

The second parameter of the open function corresponds to a mode which is typically read ('r'), write ('w'), or append ('a').

- **Read Mode:** A value of 'r' indicates that you'd like to open the file for read only operations,
- **Write Mode:** A value of 'w' indicates you'd like to open the file for write operations.
  - Note: In the event that you open a file that already exists for write operations this will overwrite any data currently in the file so you must be careful with write mode.
- **Append Mode:** ('a') will open a file for write operations, but instead of overwriting any existing data it will append data to the end of the file.



# File Access Mode

---

- Below you will find a list of all the available file modes. As I mentioned in a previous slide the most commonly used are read, write, and append.
- Read/Write Capability: However, you can also add the “+” to each of the modes to enable read/write capability. The contents of a file can be preserved or deleted depending upon the combination that you use.
- For example, w+ will open a file for read/write but the contents of the file are **deleted** while r+ **preserves** the contents of the file.
- **Binary Mode:** Adding a ‘b’ to r, w, or a will open a file in Binary mode.
- **Universal Mode:** Finally, the universal or ‘U’ character applies a universal newline translator.

# File Read

(char, token, line, block)

LECTURE 3



# read()

read function with a specific number of bytes.

---

- **file.read(n)** - This method reads n number of characters from the file, or if n is blank it reads the entire file.
- **file.readline()** - This method reads an entire line from the text file.
- **file.readlines()** - This method reads an entire file into a list of line strings from the text file.
- **file.read()** – This method reads the entire file from a text file.
- Return 0 (False) when end of file.



# File Access Pattern

---

## File Access Pattern Algorithm:

Open **File**

Read data from file to a **buffer**

**while** checking the **buffer** is valid:  
    working on the **buffer**

    Read data from file to a **buffer**

## Terminology:

**File f**: file handler

**Buffer**: `ch`, `token(string)`, `line`, `lines(list)`

**Read Functions**: `read(1)`, `read()`, `readline()`, `readlines()`



# File Access Code in Python and Java

---

## Python Pseudo Code:

```
f = open("filename.txt", "r")  
buffer=f.read_function()  
while buffer:  
    processing(buffer)  
    buffer=f.read_function()
```

```
# null return from the  
# read_function is used to  
# check the end_of_file  
# condition
```

## Java Pseudo Code:

```
File f = new File("filename.txt" );  
Scanner in = new Scanner();  
while (in.hasNext()) {  
    buffer=in.nextReading();  
    processing(buffer);  
}
```



# Data File to be Read in

aa.txt

---

alpha\n

beta\n

gamma\n

delta\n

epsilon\n

null (0, or EOF)





# Read File Character by Character: read(1)

Demo Program: file1.py

```
f = open("aa.txt", "r")
ch=f.read(1)
while ch:
    print(ch, end=" ")
    ch=f.read(1)
f.close()
```

Run fs4

C:\Python\Python36\python.exe

a l p h a  
b e t a  
g a m m a  
d e l t a  
e p s i l o n

Use space as separator so that we know the characters are read in one by one.



# Read File Token by Token: readlines()

Demo Program: file2.py (One Token Per Line File)

```
f = open("aa.txt", "r")
lines = f.readlines()
for line in lines:
    line = line.strip()
    print(line, end=" ")
f.close()
```

Run fs2

```
C:\Python\Python36\python.exe
alpha beta gamma delta epsilon
```

Equivalent to .trim() in Java

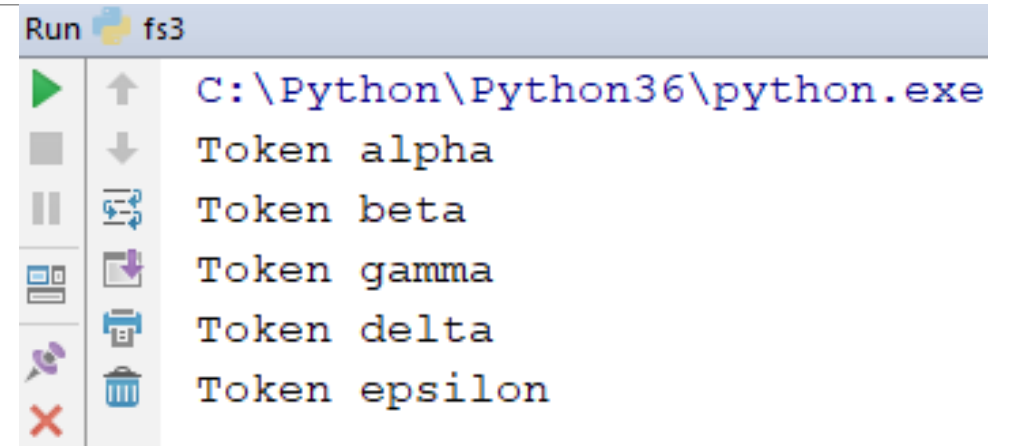
Take out all of the white space characters (\n, \t, \f, space)



# Read File Token by Token: read().split()

Demo Program: file3.py (Read the whole file into a string and split)

```
f = open("aa.txt", "r")
tokens=f.read().split()
for token in tokens:
    print("Token", token)
f.close()
```



```
Run fs3
C:\Python\Python36\python.exe
Token alpha
Token beta
Token gamma
Token delta
Token epsilon
```

Used to identify it is a token.

## Note:

**read():** read the whole file into a string.

**read().split():** read the whole file into a string. Then split the string into a list of tokens (string)

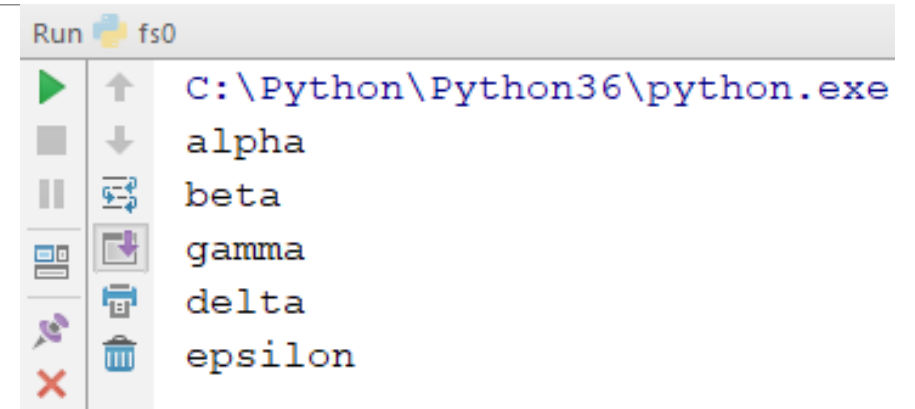
**readlines():** read the whole file into a list of lines.



# Read File Line by Line: readline()

Demo Program: file4.py

```
f = open("aa.txt", "r")
line = f.readline()
while line:
    print(line, end="")
    line = f.readline()
f.close()
```





# Read File as a whole: readlines()

Demo Program: file5.py (Read the whole file into a list of line strings.)

---

```
f = open("aa.txt", "r")
lines = f.readlines()    # lines is a list of line strings
print("Print file in list format: ")
print(lines)
```

```
all_lines = ""
for line in lines:
    all_lines += line
```

```
print("Print file as a long string: ")
print(all_lines)
f.close()
```

## Output:

```
Print file in list format:
['alpha\n', 'beta\n', 'gamma\n', 'delta\n', 'epsilon\n', '\n', '\n']
Print file as a long string:
alpha
beta
gamma
delta
epsilon
```

# Token Preprocessing

LECTURE 4



# Step 1: Remove Non-Letter Characters

Demo Program: `file_remove1.py`

---

Goal: read all words from a file. Remove the white space characters, punctuation marks, numbers, and “’s” possessive specifier ‘s.

1. Read the whole file into a text string.
2. remove ‘s
3. remove non-letter characters.
4. trim white space characters.

```
f = open("usdeclar.txt", "r") # file_remove1.py
① text = f.read()
② text = text.replace("'s", " ") # remove all 's
tokens = text.split()
for i in range(len(tokens)):
    new_token = tokens[i]
    for j in range(len(new_token)):
        ch = new_token[j]
        if not (ch.isalpha()):
            # must assign the result back to new_token
            new_token = new_token.replace(ch, " ")
    ③ tokens[i] = new_token.strip()
    ④ count = 0 # this part just to print out the words
for token in tokens:
    if (count % 20 == 0): print(token, end="")
    elif (count % 20 == 19): print(" "+token)
    else: print(" "+token, end=" ")
    count = count + 1
f.close()
```





# Step 2: Remove all the Empty Tokens

Demo Program: `file_remove2.py`

---

Goal: remove all the tokens that is either " " or `length == 0`.

1. strip all whitespace again (there might be spaces created in step 1)
2. convert all words to lower case. [`str.lower()`]
3. append only non-empty and non-space strings to a new word list.

```
f = open("usdeclar.txt", "r")
text = f.read()
text = text.replace("'s", " ")    # remove all 's
tokens = text.split()
for i in range(len(tokens)):
    new_token = tokens[i]
    for j in range(len(new_token)):
        ch = new_token[j]
        if not (ch.isalpha()):
            new_token = new_token.replace(ch, " ")    # must assign the result back to new_token
    tokens[i] = new_token.strip()
# remove empty or meaningless tokens
tokenb = []
for token in tokens:
    token = token.strip()
    token = token.lower()
    if (token != " " and len(token) != 0): tokenb.append(token)
tokens = tokenb
# print out part
count = 0
for token in tokens:
    if (count % 20 == 0): print(token, end="")
    elif (count % 20 == 19): print(" "+token)
    else: print(" "+token, end=" ")
    count = count + 1
print("\n")
print("Word count in File "+ "usdeclar.txt is "+str(len(tokens)))
f.close()
```



## Result at this Stage

---

- Each token is a lower-case string.
- Each token contains no symbol.
- Each token has no number.
- The length of the list tokens is the number of words in the source file.

# Text Processing After File Read-in

LECTURE 5



## Step 3: Calculate the count of individual words

Demo Program: [filecount.py](#)

---

- Convert the tokens list to a non-repeating word list.
  1. If a words has never shown up, add the word into the non-repeating word list.
  2. Sort the word list according to the alphabetical order

```
f = open("usdeclar.txt", "r")
text = f.read()
text = text.replace("'s", " ")    # remove all 's
tokens = text.split()            # using regular expression by one or more spaces
for i in range(len(tokens)):
    new_token = tokens[i]
    for j in range(len(new_token)):
        ch = new_token[j]
        if not (ch.isalpha()):
            new_token = new_token.replace(ch, " ")    # must assign the result back
to new_token
    tokens[i] = new_token.strip()

# remove empty or meaningless tokens
tokenb = []
for token in tokens:
    token = token.strip()
    token = token.lower()
    if (token != " " and len(token) != 0): tokenb.append(token)
tokens = tokenb
```

1 *# create non-repeating word list*

```
word_list = []  
for i in range(len(tokens)):  
    found = False  
    j=0  
    while not found and j<len(word_list):  
        if (tokens[i]==word_list[j]):  
            found = True  
            j = j + 1  
    if not found:  
        word_list.append(tokens[i])
```

*# sort the word\_list*

2

```
word_list.sort()  
# print out part  
tokens = word_list  
count = 0  
for token in tokens:  
    print(token)  
    count = count + 1  
print("\n")  
print("Word count in File "+"usdeclar.txt is "+str(len(tokens)))  
f.close()
```

# Create Occurrence Count List

LECTURE 6





## Step 4: Generating the Occurrent Count List

Demo Program: `filecount2.py`

---

1. When a word is first found, add the word into the `word_list` and append a one into the occurrence count list.
2. When a word is found later, increase the occurrence count number for the token.

# Creating the Occurrence List

```
# create non-repeating word list
word_list = []
occurrence = []
for i in range(len(tokens)) :
    found = False
    j=0
    while not found and j<len(word_list):
        if (tokens[i]==word_list[j]):
            ❷ occurrence[j] += 1
            found = True
            j = j + 1
    if not found:
        word_list.append(tokens[i])
        ❶ occurrence.append(1)
```



## Step 4: Sorting by Occurrence

---

1. Repeat number of Tokens times, each time find the token with maximum occurrence time.
  - Each time, remove the token and its occurrence count with maximum occurrence from word\_list
  - Append the token to wlist
  - Append the occurrence count to olist
2. Print out the token and occurrence pair

# Sorting the Word List by Occurrence Count

```
leng = len(word_list)
wlist = []
olist = []
for i in range(leng):
    ❶ max = -1
    ind = 0
    for j in range(len(occurrence)):
        if (occurrence[j] > max):
            max = occurrence[j]
            ind = j
    wlist.append(word_list[ind])
    olist.append(occurrence[ind])
    del (word_list[ind])
    del (occurrence[ind])
```

# Print out the Token and Occurrence Count

```
# print out part
tokens      = wlist
occurrence  = olist
for i in range(len(tokens)):
    2 t = "%-20s - %d" % (tokens[i], occurrence[i])
      print(t)
```

# Python String Functions

LECTURE 7

Python String Functions	Description
<u>capitalize()</u>	This method will convert the first character to Capitalize and following characters to Lowercase
casefold()	This method will return the given string in Lowercase.
<u>center()</u>	This method is used to Justify the string to Center and fill the remaining width with default white spaces
<u>count()</u>	This method <b>Counts</b> , How many times the string is occurred
encode()	This method returns the encoded version of a string object
<u>endswith()</u>	This method returns TRUE, if the string Ends with the specified substring
expandtabs()	This method returns a copy of the given string, where all the tab characters will be replaced with one or more spaces.
<u>find()</u>	It returns the <b>index</b> position of the first occurrence of a specified string. It will return -1, if the specified string is not found
format()	This method will be useful to <b>format</b> the string
format_map()	This method will be useful to format the string

Python String Functions	Description
<u><a href="#">index()</a></u>	It returns the index position of the first occurrence of a specified string. It will raise ValueError, if the specified string is not found
<u><a href="#">isalnum()</a></u>	This method returns TRUE, if the string contains <b>letters</b> and <b>numbers</b>
<u><a href="#">isalpha()</a></u>	This method returns TRUE, if the string has at least one letter and all the letters are <b>Alphabetic</b>
<u><a href="#">isdecimal()</a></u>	This method returns TRUE, if the string has at least one letter and all the letters are <b>Decimal</b>
<u><a href="#">isdigit()</a></u>	This method returns TRUE, if the string has at least one letter and all the letters are <b>Digits</b>
<u><a href="#">isidentifier()</a></u>	This method returns TRUE, if the string is valid identifier
<u><a href="#">islower()</a></u>	This method returns TRUE, if the string has at least one letter and all the letters are in <b>Lowercase</b>
<u><a href="#">isnumeric()</a></u>	This method returns TRUE, if the string has at least one letter and all the letters are <b>Numeric</b>
<u><a href="#">isprintable()</a></u>	This method returns TRUE, if all the letters are Printable
<u><a href="#">isspace()</a></u>	This method returns TRUE, if the string contains only white spaces
<u><a href="#">istitle()</a></u>	This method returns TRUE, if the string has at least one letter and it is a Title.
<u><a href="#">isupper()</a></u>	This method returns TRUE, if the string has at least one letter and all the letters are in <b>Uppercase</b>



Python String Functions	Description
<code>join()</code>	This method will be useful to Join (Concatenate) a list of strings
<code>ljust()</code>	This method is used to Justify the string to Left hand side and fill the remaining width with default white spaces
<code>lower()</code>	This method will convert the given string into <b>Lowercase</b> letters and return new string
<code>lstrip()</code>	It will remove the white spaces from Left hand side of a string
<code>maketrans()</code>	It returns the transaction table. We can further use this transaction in <code>translate()</code> method.
<code>partition()</code>	It partition the given string at the first occurrence of the specified separator and return a tuple with three arguments.
<code>replace()</code>	This method will search for specified string and replace it with new string value
<code>rfind()</code>	It returns the index position of the Last occurrence of a specified string. It will return -1, if the specified string is not found
<code>rindex()</code>	It returns the index position of the Last occurrence of a specified string. It will raise <code>ValueError</code> , if the specified string is not found
<code>rjust()</code>	This method is used to Justify the string to Right hand side and fill the remaining width with default white spaces
<code>rpartition()</code>	This method will partition the given string using the specified separator and return a tuple with three arguments.
<code>rsplit()</code>	This method will be useful to Split the string into list of strings, based on the specified delimiter. This will done from right to left
<code>rstrip()</code>	It will remove the white spaces from <b>Right</b> hand side of a string

## Python String Functions

## Description

[split\(\)](#) This method will be useful to Split the string into list of strings, based on the specified **delimiter**

[splitlines\(\)](#) It returns a list of lines in the given string by breaking the given string at line boundaries.

[startswith\(\)](#) This method returns TRUE, if the string Starts with the specified substring

[strip\(\)](#) It will **remove** the white spaces from both ends. Performs both [lstrip\(\)](#) and [rstrip\(\)](#)

[swapcase\(\)](#) This method will convert the Lowercase letters into Uppercase and Uppercase letters into Lowercase

[title\(\)](#) This method will convert the first character in each word to Uppercase and following characters to Lowercase

[translate\(\)](#) Returns a Copy of the given string in which each character has been mapped with the transaction table.

[upper\(\)](#) This method will convert the given string into **Uppercase** letters and return new string

[zfill\(\)](#) It returns a copy of the string filled with [ASCII](#) '0' digits on the left hand side of the string to make a string length to specified width.

# Simple and Space Efficient Code

LECTURE 8



# Strip of Right-Hand-Side “\n”

---

- **file.readlines()** operation will read in a list of line strings. Each line has a “\n” newline mark.
- Therefore, we can remove the newline mark by  
for line in open(file\_name):  
    process(line.rstrip('\n'))  
or  
for line in open(file\_name):  
    line = line.rstrip('\n') # re-bind line (also rebind on next loop iteration)  
    process(line)



# Python is Immutable

All Returned Result must be Assigned to a String Reference Variable

---

## **CORRECT**

```
for line in open(file_name):  
    line = line.rstrip('\n')  
    process(line)
```

## **INCORRECT**

```
for line in open(file_name):  
    line.rstrip('\n') # does NOTHING!  
    process(line)
```



# List Element Creation by Iterator

---

```
line_list = [line.rstrip('\n') for line in open(file_name)]
```



# Alias for list of a File

---

- With the open context manager, we would write the above code

- fragments as

```
with open(file_name) as open_file:
```

```
    for line in open_file:
```

```
        process(line.rstrip('\n'))
```

- or

```
with open(file_name) as open_file:
```

```
    line_list = [line.rstrip('\n') for line in open_file]
```

Alias for the `open(file_name)` as an iterable object





# Simple and Space Efficient Code for `readlines()`

---

- If we wanted to call `.readlines()` and process every string in the file (without the `'\n'` characters at the end) we would write  
**`# open(file_name).readlines() is also iterable`**  
`for line in open(file_name).readlines():`  
 `process(line.rstrip('\n'))`
- or  
`for line in open(file_name).readlines():`  
 `line = line.rstrip('\n') # re-bind line (also rebind on next loop iteration)`  
 `process(line)`
- or (by list creator by iterator)  
`line_list = [line.rstrip('\n') for line in open(file_name).readlines()]`





# List of Tokens Generation `read().split()`

---

- calling `open_file.read()` returns the string

`'Line 1\nLine 2\nLine 3\n'`

- We can split this string into a list of strings by calling the `.split` method.

`line_list = open(file_name).read().split('\n')`

- and this code is simpler than what we have seen before, which is equivalent to

`line_list = [line.rstrip('\n') for line in open(file_name)]`



# `.read().split('\n')`

## Generating `line_list` not tokens

---

The `.read().split('\n')` code above is even less space efficient than the simpler comprehension, because it stores both the entire file string and a list of all the lines in the file at the same time; the comprehension stores the list of all the lines in the file, but not a string whose contents is the entire file itself.

```
for line in open(file_name).read().split('\n'):
    process(line)
```



# Summary

---

There is little to be gained by reading files by calling the **.readlines()** or the **.read()** method. Iterate directly over the "open" file with a standard for loop or a for loop in a comprehension.

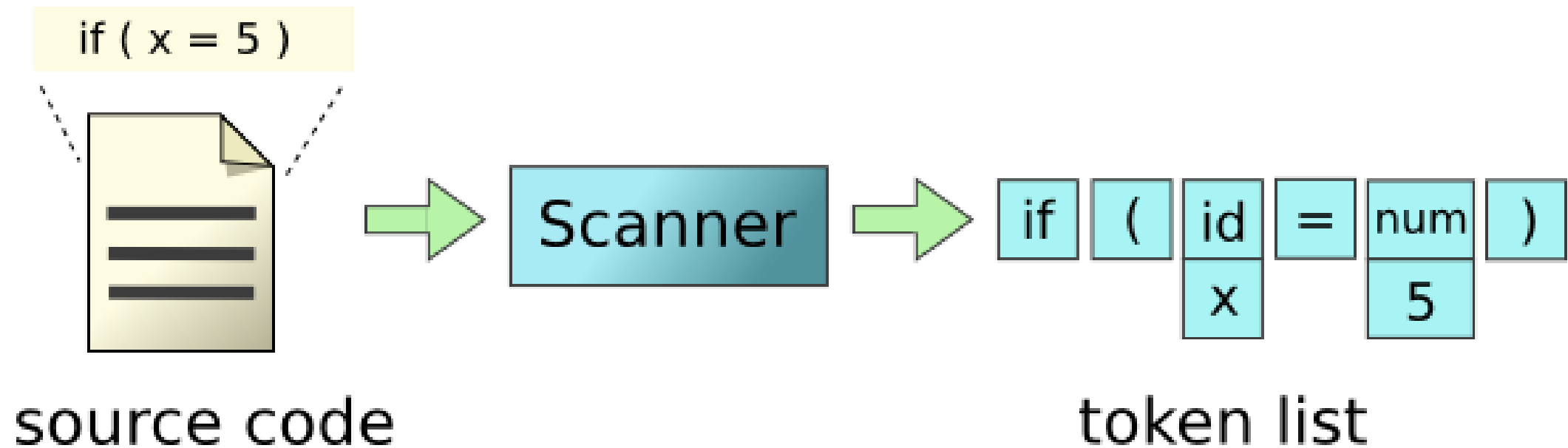
# Parsing of Tokens

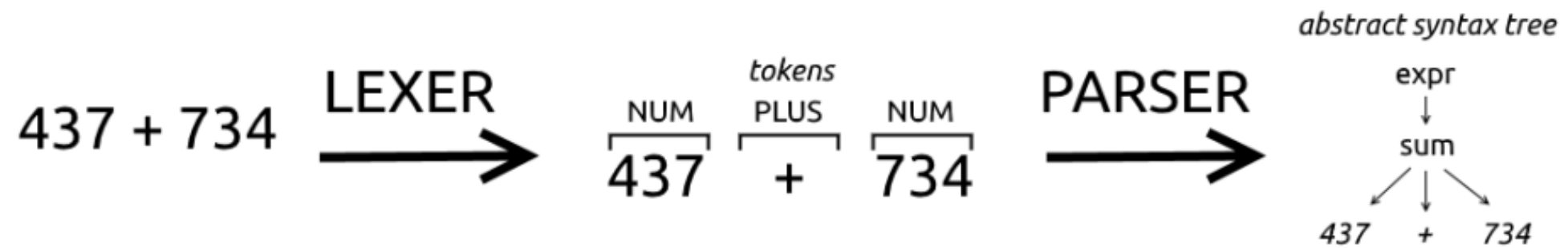
LECTURE 9




# Lexical Analysis

## From Source File to Token List





	Scanning	Parsing
<b>Task</b>	determining the structure of tokens	determining the syntax or structure of a program
<b>Describing Tools</b>	regular expression	context-free grammar 
<b>Algorithmic Method</b>	represent by DFA	top-down parsing bottom-up parsing
<b>Result Data Structure</b>	liner structure	parser tree or syntax tree, they are recursive



# Reading Files and Parsing their Contents

---

Step 1:

```
for line in open(file_name):  
    process( int(line.rstrip('\n')) )
```

Some text files contain lines that store other types or mixed-types of information. Suppose that we wanted to read a text file that stored strings representing numbers (one number per line).





# Parsing their Contents

---

Step 2: Simple Parsing (Pre-condition: Each line is a record.)

Here we are assuming process takes an integer value as an argument.

In some files each line is a "record": a fixed number of fields of values, with possibly different types, separated by some special character (often a space or punctuation character like a comma or colon). To process each record in a file, we must

1. read its line
2. separate its fields of values (still each value is a string)
3. call a conversion function for each string to get its value

# Preprocessing for Parsing

LECTURE 10



# zip Function

---

## **zip() Parameters**

The zip() function takes zero or more iterables (lists)

## **Return Value from zip()**

The zip() function returns an a of tuples based on the iterable object.



# zip Function

---

## Return Value from zip()

The zip() function returns an iterator of tuples based on the iterable object.

- If no parameters are passed, zip() returns an **empty iterator**
- If a single iterable is passed, zip() returns an **iterator of 1-tuples**.  
Meaning, the number of elements in each tuple is 1.
- If multiple iterables are passed, ith tuple contains ith Suppose, two iterables are passed; one iterable containing 3 and other containing 5 elements. Then, the returned iterator has 3 tuples. It's because iterator stops when shortest iterable is exhausted.



# Demo Program: zip.py

---

- Show how to combine many **iterables** (data collections like lists) into iterables of tuples.
- Purpose: Combine list of same records into one list. Each tuple will represent one record.

```
1 numberList = [1, 2, 3]
2 strList = ['one', 'two', 'three']
3
4 # No iterables are passed
5 result = zip()
6
7 # Converting itertor to list
8 resultList = list(result)
9 print(resultList)
10
11 # Two iterables are passed
12 result = zip(numberList, strList)
13
14 # Converting itertor to set
15 resultSet = set(result)
16 print(resultSet)
```

Run  zip



C:\Python\Python36\python.exe "C:\Eric\_Chou

[]

{(1, 'one'), (3, 'three'), (2, 'two')}



# Demo Program: filecount3.py

---

- Rewrite the filecount2.py to combine the word\_list and occurrence list into one list of records (each word record has two tuples: one for word string and one for integer occurrence count of that word.)
- We didn't use the default **zip** function. We used a custom designed zip function.



# Custom Designed zip Function

---

```
def zip(a, b):  
    zip_list = []  
    for i in range(len(a)):  
        zip_list.append([a[i], b[i]])  
    return zip_list
```



```
f = open("usdeclar.txt", "r") # filecount3.py (part 1: tokenization)
text = f.read()
text = text.replace("'s", " ") # remove all 's
tokens = text.split() # using regular expression by one or more spaces
for i in range(len(tokens)):
    new_token = tokens[i]
    for j in range(len(new_token)):
        ch = new_token[j]
        if not (ch.isalpha()):
            new_token = new_token.replace(ch, " ")
            # must assign the result back to new_token
    tokens[i] = new_token.strip()

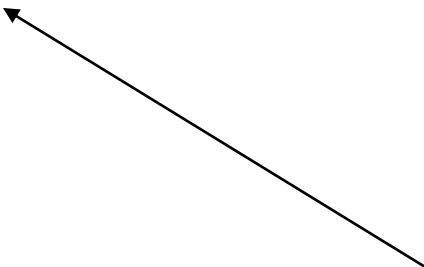
# remove empty or meaningless tokens
tokenb = []
for token in tokens:
    token = token.strip()
    token = token.lower()
    if (token != " " and len(token) != 0): tokenb.append(token)
tokens = tokenb
```

## # filecount3.py Part 2: create non-repeating word list

```
word_list = []
occurrence = []
for i in range(len(tokens)):
    found = False
    j=0
    while not found and j<len(word_list):
        if (tokens[i]==word_list[j]):
            occurrence[j] += 1
            found = True
        j = j + 1
    if not found:
        word_list.append(tokens[i])
        occurrence.append(1)
```

## # filecount3.py Part 3: selection sort

```
words = zip(word_list, occurrence)
print(words)
leng = len(words)
words2 = []
for i in range(leng):
    max = -1
    ind = 0
    for j in range(len(words)):
        if (words[j][1] > max):
            max = words[j][1]
            ind = j
    words2.append(words[ind])
    del (words[ind])
```



Zip function groups the word and its associated occurrence into a list (tuple). This makes sorting easier.

```
# filecount3.py Part 4: listing words in order
```

```
words = words2
```

```
# print out part
```

```
for i in range(len(words)):
```

```
    t = "%-20s - %d" % (words[i][0], words[i][1])
```

```
    print(t)
```

```
print("\n")
```

```
print("Word count in File "+ "usdeclar.txt is "+str(len(words)))
```



# Summary for Pre-Processing

---

1. Retrieve proper information before parsing the data.
2. Remove un-wanted symbols.
3. Data File format to be processed: Plain Text file (TXT), CSV (Comma Separated Values) file, JSON file, JSONP file, XML file, XLS (Excel), XLSX (Excel new).

On-line Converter (CSV to JSON): <http://www.csvjson.com/csv2json>

Free File Format Converter: <https://www.lifewire.com/free-file-converter-software-and-online-services-2626121>

# CSV (Comma Separated Values) Files

LECTURE 11



# What is a CSV File?

---

- A file with the **CSV** file extension is a **Comma Separated Values** file. All CSV files are plain text, can contain numbers and letters only, and structure the data contained within them in a tabular, or table, form.
- Files of this format are generally used to exchange data, usually when there's a large amount, between different applications. Database programs, analytical software, and other applications that store massive amounts of information (like contacts and customer data), will usually support the CSV format.
- A Comma Separated Values file might sometimes be referred to as a Character Separated Values or Comma Delimited file but regardless of how someone says it, they're talking about the same CSV format.

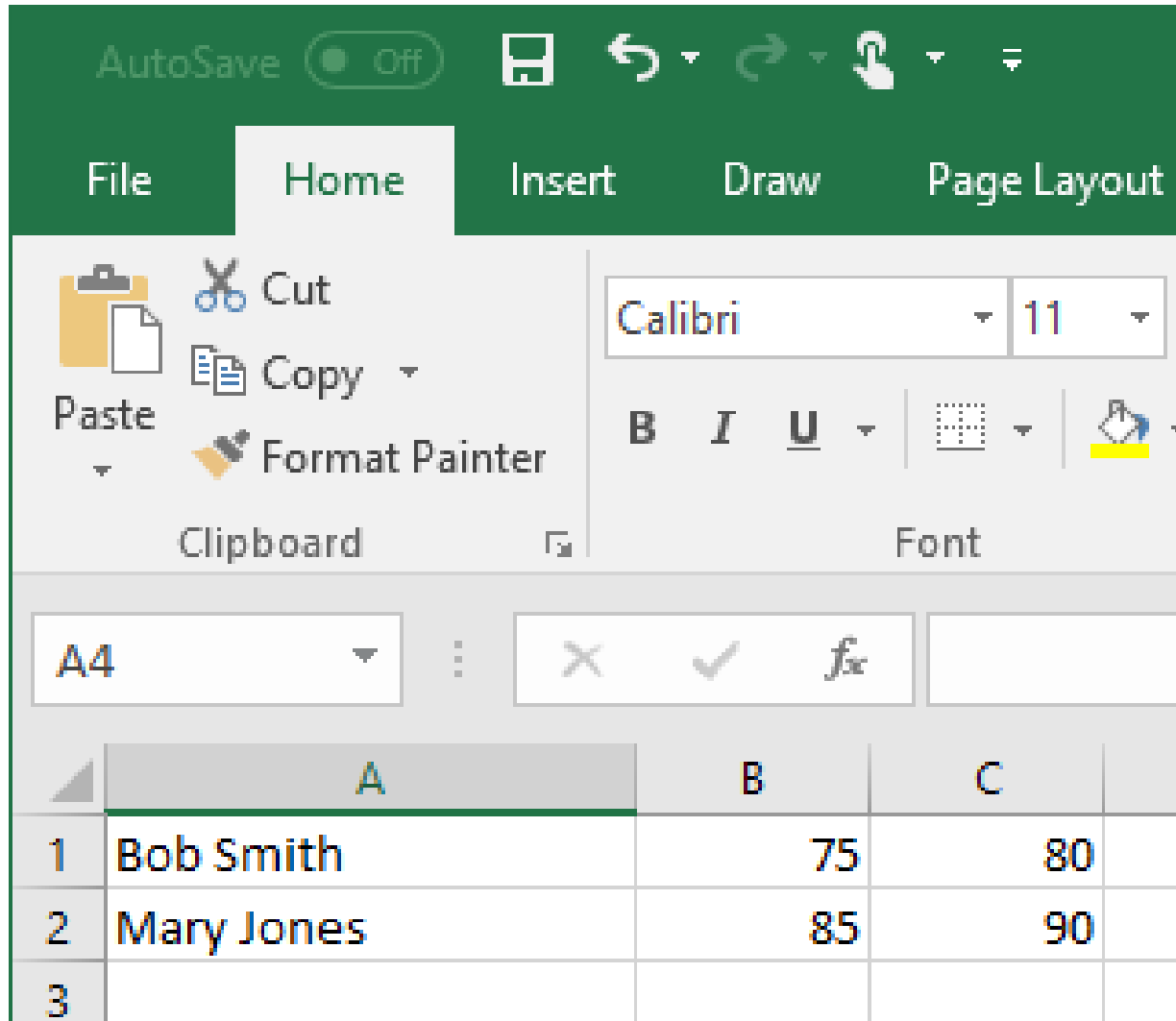


# How To Open a CSV File?

---

- **Spreadsheet** software (MS-Excel) is generally used to open and edit CSV files, such as the free **OpenOffice Calc** or **Kingsoft** Spreadsheets. Spreadsheet tools are great for CSV files because the data contained is usually going to be filtered or manipulated in some way after opening.
- **Text Editor**: [Notepad++](#) or [GenScriber](#) to open CSV files but large ones will be very difficult to work with in these types of programs. [Open Freely](#) is another alternative but with the same problem with larger CSVs.
- Considering the number of programs out there that support structured, text-based data like **CSV**, you may have more than one program installed that can open these types of files.





# Generation of CSV by MS-Excel

Demo Program:  
Student.XLSX

---

**GO MS-EXCEL!!!**



Info

New

Open

Save

Save As

Print

Share

Export

Publish

Close

Account 

Feedback

Options

# Export



Create PDF/XPS Document



Change File Type

## Change File Type

### Workbook File Types



Workbook

Uses the Excel Spreadsheet format



Excel 97-2003 Workbook

Uses the Excel 97-2003 Spreadsheet format



OpenDocument Spreadsheet

Uses the OpenDocument Spreadsheet format



Template

Starting point for new spreadsheets



Macro-Enabled Workbook

Macro enabled spreadsheet



Binary Workbook

Optimized for fast loading and saving

### Other File Types



Text (Tab delimited)

Text format separated by tabs



CSV (Comma delimited)

Text format separated by commas



Formatted Text (Space delimited)

Text format separated by spaces



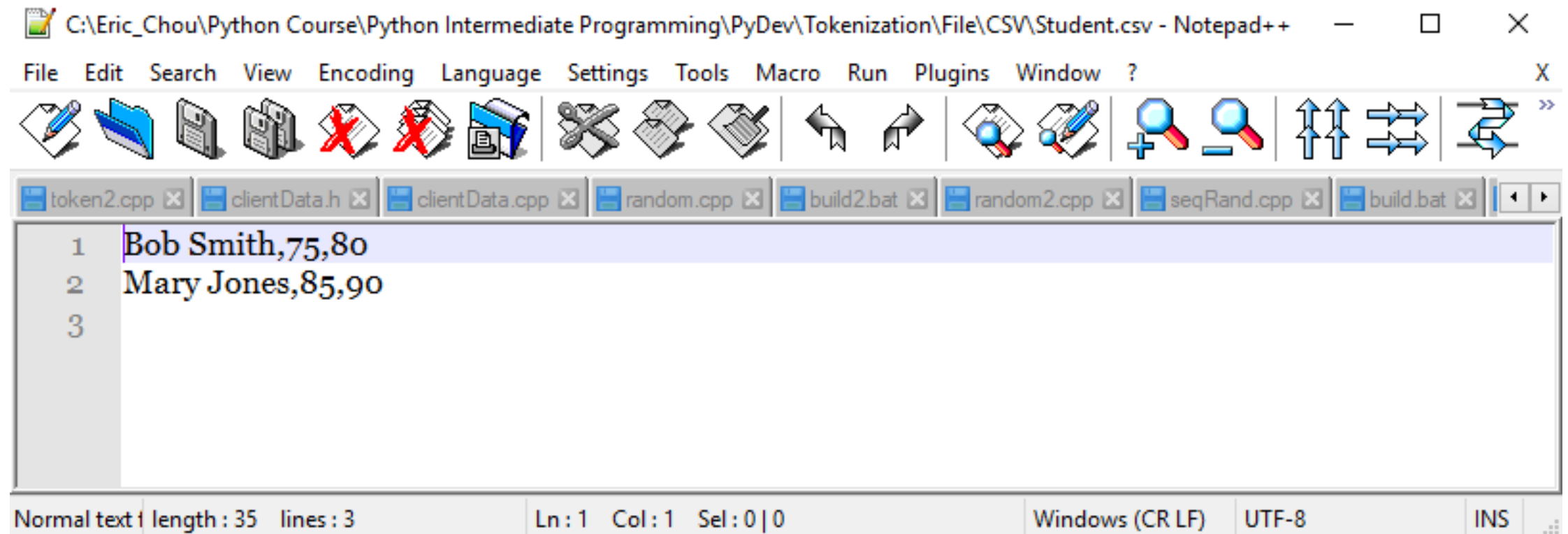
Save as Another File Type



Save As

Pick This One

# Open CVS File with Notepad++



# Parsing CSV Files

LECTURE 12



# parse\_lines function

Demo Program: parse\_lines.py

---

```
# parse_lines.py
# read a csv file and print the student record to screen.
#
def parse_lines(open_file, sep, conversions):
    for line in open_file:
        yield [conv(item) for conv, item in
               zip(conversions, line.rstrip('\n').split(sep))]

for fields in parse_lines(open("Student.csv"), ',', (str, int, int)):
    print(fields[0], (fields[1]+fields[2])/2)
```

Run		parse_lines
		C:\Python\Python36\python.exe
		Bob Smith 77.5
		Mary Jones 87.5



# parse\_lines function

Demo Program: `parse_lines.py`

---

- The module contains the **parse\_lines** function that easily supports reading records from files (similarly to how lines are read from "open" files).
- **sep** is a special character used to separate the fields in the record;
- **conversions** is a tuple of function objects: they are applied in sequence to the string values extracted from the separated fields.
- When we iterate over a call to **parse\_lines** (similar to iterating over a call to "open"), the index variable is bound to a list of the values of the fields in the record.



# zip function

---

**conversions:** (str,int,int)

**a line of data fields:** (Bob Smith,75,80)

zip(conversions, ('Bob Smith', 75, 80) →  
[(str, 'Bob Smith', (int, 75), (int, 80))] # zip object

Apply **str("Bob Smith"), int(75), int(80)**



# Apply str("Bob Smith"), int(75), int(80)

---

```
yield [conv(item) for conv, item in  
       zip(conversions, line.rstrip('\n').split(sep))]
```

**yield**  
return generator

generator is a kind of iterable which is only temporary iterable. It can only be used once (on the fly).





# Iterables

---

- When you create a list, you can read its items one by one. Reading its items one by one is called **iteration**:

```
>>> mylist = [1, 2, 3]
>>> for i in mylist:
...     print(i)
1
2
3
```

- **mylist** is an iterable.



# Iterables

---

- When you use a list comprehension, you create a list, and so an iterable:

```
>>> mylist = [x*x for x in range(3)]  
>>> for i in mylist:  
...     print(i)  
0  
1  
4
```

- Everything you can use "for... in..." on is an **iterable**; **lists**, **strings**, **files**...



# Generators

---

- Generators are iterators, a kind of iterable you can only iterate over once. Generators do not store all the values in memory, they generate the values **on the fly**:

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator:
...     print(i)
0
1
4
```



# Generators

---

- It is just the same except you used `()` instead of `[]`.
- BUT, you cannot perform `for i in mygenerator` a second time since generators can only be used once: they calculate 0, then forget about it and calculate 1, and end calculating 4, one by one.



# Yield

---

- yield is a keyword that is used like **return**, except the function will return a generator.

```
>>> def createGenerator():
...     mylist = range(3)
...     for i in mylist:
...         yield i*i ...
>>> mygenerator = createGenerator() # create a generator
>>> print(mygenerator) # mygenerator is an object!
<generator object createGenerator at 0xb7555c34>
>>> for i in mygenerator:
...     print(i)
0
1
4
```



# Yield

Return as a whole, Yield on each iteration

---

- To master **yield**, you must understand that when you call the function, the code you have written in the function body does not run. The function only returns the generator object, this is a bit tricky.
- Then, your code will be run each time the for uses the generator.
- The first time the for calls the generator object created from your function, it will run the code in your function from the beginning until it hits yield, then it'll return the **first** value of the loop. Then, each other call will run the loop you have written in the function one more time, and return the next value, until there is no value to return.

```

# generator_examples.py
print("Iterable Example:")
mylist = [1, 2, 3]
for i in mylist:
    print(i)
print("\n")
print("List Creation by Iterator Example 2:")
mylist = [x*x for x in range(3)]
for i in mylist:
    print(i)
print("\n")
print("Generator Example:")
mygenerator = (x*x for x in range(3))
for i in mygenerator:
    print(i)
print("\n")
print("Yield Example:")
def createGenerator():
    mylist = range(3)
    for i in mylist:
        yield i*i
mygenerator = createGenerator() # create a generator
print(mygenerator) # mygenerator is an object!
for i in mygenerator:
    print(i)

```

Iterable Example:

```

1
2
3

```

List Creation by Iterator Example 2:

```

0
1
4

```

Generator Example:

```

0
1
4

```

Yield Example:

```

<generator object createGenerator at 0x0000023B39B25BF8>
0
1
4

```



# Understand parse\_lines.py

---

- For example, the following file contains fields of a name (str) followed by two test scores (ints) all separated by commas.

**Bob Smith,75,80**

**Mary Jones,85,90**

- We could read this file and print out the names of each student and their average test score by

```
for fields in parse_lines( open(file_name), ',' , (str,int,int) ):  
    print(fields[0], (fields[1]+fields[2])/2)
```

- Here fields is repeatedly bound to a 3-list containing a name (str) followed by two test scores (ints). fields is first bound to ['Bob Smith', 75, 80] and then to ['Mary Jones', 85, 90].



# Understand parse\_lines.py

## Note:

1. if we specified conversions as (str, int) it would return the 2-lists ['Bob Smith', 75] followed by ['Mary Jones', 85] (because looping over a zip stops when one of its arguments runs out of values: here each line contains more field values than conversion functions). Accessing fields[2] in the code above would raise an **IndexError** exception.
  2. Likewise (because looping over a zip stops when one of its arguments runs out of values), if the a line contains a name and 3 integer values, only the name and first two integers would be returned in the 3-list: the line  
Paul White,80,75,85  
returns only the 3-list ['Paul White', 80, 75]
- So parse\_lines would not raise any exceptions in the code above; instead it incorrectly reads the file contents with no warning.
  - We could define a more complicated parse\_lines function that **checked** and immediately **raised** an **exception** if the number of separated field values in a record was not equal to the length of the tuple of conversion functions.

# Simplification of parse\_lines.py

LECTURE 13



# Simplification by Unpacking

---

- A simpler way to write such code is to use multiple index variables and unpacking (as we do when we write: `for k,v in adict.items()`).

```
for name,test1,test2 in parse_lines(open(file_name),',',(str,int,int)):  
    print(name, (test1+test2)/2)
```

- With this for loop, the first error noted above would also raise an exception because there would not be three values to **unpack** into `name`, `test1`, and `test2`; the second error would again go unnoticed.



# Demo Program: parse\_lines2.py

---

```
# parse_lines2.py
```

```
# read a csv file and print the student record to screen.
```

```
#
```

```
def parse_lines(open_file, sep, conversions):
```

```
    for line in open_file:
```

```
        yield [conv(item) for conv,item in
```

```
                zip(conversions,line.rstrip('\n').split(sep))]
```

```
for name,test1,test2 in parse_lines(open("Student.csv"),',',(str,int,int)):
```

```
    print(name, (test1+test2)/2)
```



# Colon Separated Data List

---

- Finally, note that besides using the standard conversion function(s) like `str` and `int`, we can define our own conversion function(s).
- For example, suppose that each record in the file specified a string name, some number of int quiz results separated by colons, and an int final exam, with these three fields (name, quizzes, final) separated by commas.
- Such a file might look like

**Bob Smith,75:80,90**

**Mary Jones,85:90:77,85**



# Decoding Colon Separated Data\_List

---

Here Bob took two quizzes but Mary took three. We could define

```
def quiz_list(scores):
```

```
    return [int(q) for q in scores.split(':')]
```

and then write

```
for name,quizzes,final in parse_lines(open(file_name),',',(str,quiz_list,int)):
```

```
    print(name, sum(quizzes)/len(quizzes), final)
```

which would print

**Bob Smith 77.5 90**

**Mary Jones 84.0 85**

Note that 77.5 is  $(75+90)/2$  and 84 is  $(85+90+77)/3$ .



# Demo Program: parse\_lines3.py

---

```
# parse_lines3.py
```

```
# read a csv file and print the student record to screen.
```

```
#
```

```
def parse_lines(open_file, sep, conversions):
```

```
    for line in open_file:
```

```
        yield [conv(item) for conv,item in
```

```
                zip(conversions,line.rstrip('\n').split(sep))]
```

```
def quiz_list(scores):
```

```
    return [int(q) for q in scores.split(':')]
```

```
for name,quizzes,final in parse_lines(open("Student.csv"),',',(str,quiz_list,int)):
```

```
    print(name, sum(quizzes)/len(quizzes), final)
```



# Using Lambda Function

---

If we instead wrote

```
for fields in parse_lines(open(file_name),',',(str, quiz_list, int)):  
    print(fields)
```

it would print the 3-lists

```
['Bob Smith', [75, 80], 90]
```

```
['Mary Jones', [85, 90, 77], 85]
```

Of course, we can also use lambdas instead of named functions; below we have substituted a **lambda** for the **quiz\_list** function.

```
for fields in parse_lines(open(file_name),',', (str, lambda scores : [int(q) for q in scores.split(':')], int)):  
    print(fields)
```



- The lambda expression is the function itself
  - Apply the expression to one or more parameters

`(λ (x) x * x * x) (4)`

`# Python`  
`(lambda x:x*x*x) (4)`

**Formal Parameter**  
**Returned Value**  
**Actual Argument**

`(λ (x,y) x * y) (8,7)`

`# Python`  
`(lambda x,y:x*y) (8,7)`


# Python Lambda Function



# Demo Program: parse\_lines4.py

---

```
# parse_lines4.py  
# read a csv file and print the student record to screen.  
#  
def parse_lines(open_file, sep, conversions):  
    for line in open_file:  
        yield [conv(item) for conv,item in  
                zip(conversions,line.rstrip('\n').split(sep))]  
  
for fields in parse_lines(open("Student.csv"),",",(str, lambda scores : [int(q) for q in scores.split(":")],int)):  
    print(fields)
```

```
Run  parse_lines4  
  
▶ ↑ C:\Python\Python36\python.exe  
■ ↓ ['Bob Smith', [75], 80]  
|| ↕ ['Mary Jones', [85], 90]
```

# File Read-in Styles

LECTURE 14



# File Processing Styles

---

- Suppose that we want to process the lower case version of every word on every line (where the words on a line are separated by spaces) in a file named file.txt. Which of the following code fragments correctly does so? For those that don't, explain why they fail. For example, if the file contained the three lines:

**See spot**

**See spot run**

**Run spot run**

- it should process the following words in the following order:  
**'see', 'spot', 'see', 'spot', 'run', 'run', 'spot', 'run'.**



# Demo Program:

`spot.py` and `spot_error.py`

---

- Please check `spot_error.py` to see which examples will deliver the expected results and which examples are actually wrong.
- The `spot.py` is the corrected version.



# Summary of the Chapter

---

1. Text files read-in before tokenization may need to remove white space symbol, punctuation marks, digits, and many other things.
2. Text can be read-in character by character, token by token, line by line or as a block.
3. Line of text or block of text can be split into a list of tokens.
4. Text file of .txt and .csv are used as example in this chapter.
5. Basic parsing by validating each data field in a record from a line of text has been demonstrated.