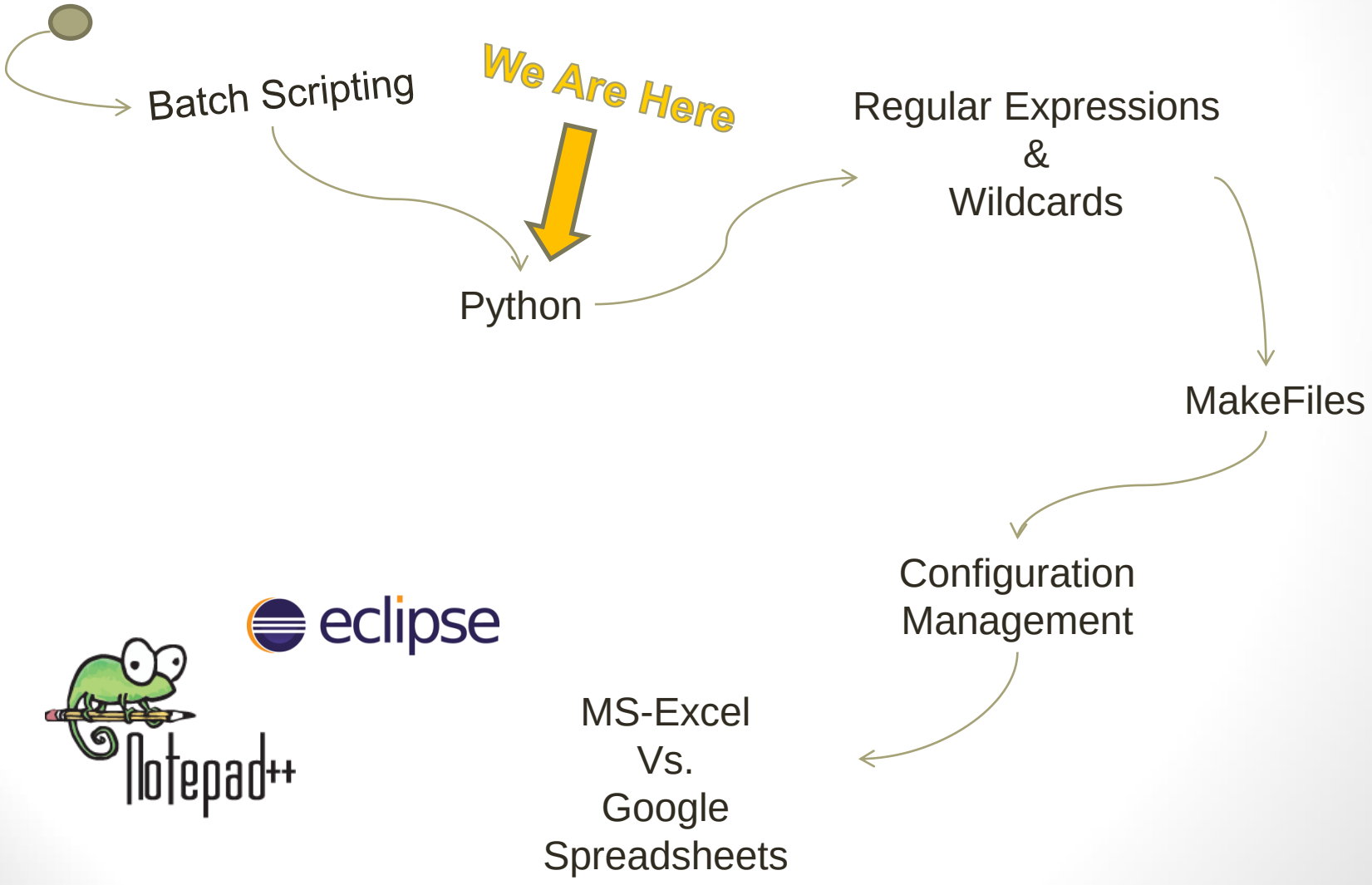
Python

Course Map


Batch Scripting
We Are Here
Python
Regular Expressions
&
Wildcards
MakeFiles
Configuration
Management
MS-Excel
Vs.
Google
Spreadsheets
eclipse
Notepad++

Agenda - 5

- Python Utilities
  - Miscellaneous operating system interfaces
  - Common pathname manipulations
  - High-level file operations
- Regular Expressions
  - Introduction
  - Python Regex
  - Related Topics
- **Lab-7**

Python Utilities

Miscellaneous operating system interfaces

- The **os** module provides a portable way of using operating system dependent functionality.
  - **os.listdir**(***dir***) ← List of filenames in that directory path (not including . and ..). The filenames are just the names in the directory, not their absolute paths.
  - **os.chdir**(***path***) ← Change the current working directory to *path*.
  - **os.getcwd()** ← Get current working directory.
  - **os.mkdir**(***dir_path***) ← Makes one dir.
  - **os.rmdir**(***path***) ←Remove (delete) the directory path. Only works when the directory is **empty**, otherwise, OSError is raised. In order to remove whole directory trees, **shutil.rmtree**() can be used.

Python Utilities

Miscellaneous operating system interfaces

- The **os** module provides a portable way of using operating system dependent functionality.
  - **os.rename**(***src***, ***dst***) ← Rename the file or directory src to dst. If dst is a directory, OSError will be raised.
  - **os.remove**(***path***) ← Remove (delete) the file path. If path is a directory, OSError is raised.
  - **os.getenv**(***varname***) ← Return the value of the environment variable *varname* if it exists, or None if it doesn't.
  - **os.system**(**cmd**) ← Runs an external command and dumps its output onto your output and returns its error code.
  - For more info on **os** module:
    - https://docs.python.org/2/library/os.html#module-os

Python Utilities

Common pathname manipulations

- The **os.path** module implements some useful functions on pathnames.
  - **os.path.exists**(***path***) ← Return True if it exists.
  - **os.path.isfile**(***path***) ← Return True if *path* is an existing regular file.
  - **os.path.isdir**(***path***) ← Return True if *path* is an existing directory.

Python Utilities

Common pathname manipulations

- The **os.path** module implements some useful functions on pathnames.
  - **os.path.getsize**(***path***) ← Return the size, in bytes, of path. Raise os.error if the file does not exist or is inaccessible.
  - **os.path.abspath**(***path***) ← Given a path, return an absolute form, e.g. d:\\Scripts\\myscript.py .

Python Utilities

Common pathname manipulations

- The **os.path** module implements some useful functions on pathnames.
  - **os.path.dirname**(***path***) ← Given dir\foo\bar.html, return the dirname "**dir\foo**"
  - **os.path.basename**(***path***) ← Given dir\foo\bar.html, return the basename "**bar.html** "
  - For more on **os.path** module**:**
    - https://docs.python.org/2/library/os.path.html

Python Utilities

High-level file operations

- The **shutil** module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal.
  - **shutil.copy**(***src***, ***dst***) ← Copy the file *src* to the file or directory *dst*.
  - **shutil.copytree(src, dst)** ← Recursively copy an entire directory tree rooted at *src*. The destination directory, named by *dst*, must not already exist; it will be created as well as missing parent directories..

Python Utilities

High-level file operations

- The **shutil** module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal.
  - **shutil.rmtree**(***path***) ← Delete an entire directory tree.
  - **shutil.move**(***src***, ***dst***) ← Recursively move a file or directory (*src*) to another location (*dst*).
  - For more on **shutil** module:
    - https://docs.python.org/2/library/shutil.html

Self-Study

EXCEPTION HANDLING

Exception Handling

- An **exception** represents a run-time error that halts the normal execution at a particular line and transfers control to error handling code.
- For example a run-time error might be that a variable used in the program does not have a value (ValueError .. you've probably seen that one a few times), or a file open operation error because that a does not exist (IOError).

Exception Handling

- Without any error handling code (as we have done thus far), a run-time exception just halts the program with an error message. That's a good default behavior, and you've seen it many times. You can add a "**try/except**" structure to your code to handle exceptions, like this:

```
try:
    ## Either of these two lines could throw an IOError, say
    ## if the file does not exist or the read() encounters a low level error.
    f = open(filename, 'rU')
    text = f.read()
    f.close()
  except IOError:
    ## Control jumps directly to here if any of the above lines throws IOError.
    sys.stderr.write('problem reading:' + filename)
  ## In any case, the code then continues with the line after the try/except
```

Exception Handling

- The **try**: section includes the code which might throw an exception.
- The **except**: section holds the code to run if there is an exception. If there is no exception, the except: section is skipped (that is, that code is for error handling only, not the "normal" case for the code).
- You can get a pointer to the exception object itself with syntax "except IOError, e: .. (e points to the exception object)".
- For more about exceptions:
  - https://docs.python.org/2/tutorial/errors.html#exceptions

Self-Study

EXCEPTION HANDLING

Course Map

We Are Here

Batch Scripting

Python

Regular Expressions
&
Wildcards

MakeFiles

Configuration
Management

MS-Excel
Vs.
Google
Spreadsheets

Regular Expressions
Introduction

- Regular expressions originated in 1956, when mathematician Stephen Cole Kleene described regular languages using his mathematical notation called regular sets.

- Regular expressions entered popular use from 1968 in two uses:
  - Pattern matching in a text editor.
  - Lexical analysis in a compiler.

- Ken Thompson later added this capability to the Unix editor **ed**, which eventually led to the popular search tool **grep**'s use of regular expressions ("**grep**" is a word derived from the command for regular expression searching in the ed editor: g/re/p meaning "**G**lobal search for **R**egular **E**xpression and **P**rint matching lines").

Regular Expressions
Introduction

- Starting in 1997, Philip Hazel developed PCRE (**P**erl **C**ompatible **R**egular **E**xpressions), which attempts to closely mimic Perl's regexp functionality

- The **IEEE** POSIX standard has three sets of compliance:
  - **SRE** = **S**imple **R**egular **E**xpressions. SRE is deprecated in favor of BRE.
  - **BRE** = **B**asic **R**egular **E**xpressions
    - Uses the following meta-characters:
      - . ^ $ [ ] *
    - The tool 'grep' uses BRE by default
  - **ERE** = **E**xtended **R**egular **E**xpressions
    - Uses the following in addition to the basic set:
      - ( ) { } ? + |
    - To access ERE in grep use 'egrep' or 'grep –E'

Regular Expressions
Introduction

Regular Expressions

- A **regular expression** is a text pattern used in **text matching**, **search and replace** and **splitting**.
- Mainly used inside scripting languages such as Perl, Python, JavaScript and etc .
- **Consists of:**
  - Alphanumeric characters
  - Special characters known as meta-characters.
  - Character classes

Wild Cards

- A **wildcard** is a generic term referring to something that can be substituted for all possibilities.
- This is like a **joker** being a wildcard in poker.
- Mainly used inside shells for **file searching and management**.
- **Consists of:**
  - "*" matches multiple characters.
  - "?" matches a single character.
  - "[]" a range of characters

Regular Expressions
Introduction

- Many programming/scripting languages provide regular expression (**regex** or **regexp** for short) capabilities:
  - Some **built-in**:
    - Perl, JavaScript, Ruby, AWK, and Tcl.
  - Others via a **standard library**:
    - .NET languages, Java, Python, POSIX C and C++ (since C++11).

- Also **regex** is supported in:
  - Notepad++
  - Eclipse
  - Many other tools and editors.

Regular Expressions
Variations

- There are **different regex implementations**, which differs in the way special characters **. { } ( )[ ] ^ $** are handled (escaping rules etc.), and occasionally **substituted**, the handling/availability of **POSIX character classes** e.g. **[:digit:]**, and the use of **options**, e.g. **g i** etc.

- **Who uses what?**
  - Perl ← Perl style
  - Python ← Python style (Modeled on Perl)
  - Java ← Java style (POSIX ERE variant).
  - PHP ← POSIX ERE, PCRE
  - JavaScript ← uses ECMA style.
  - Grep ← POSIX BRE / ERE
  - Eclipse ← Search uses Java style regex.
  - Notepad++ ← PCRE

Regular Expressions
Introduction

- Here is a link for an online tool that will help you while working with regex:
  - https://regex101.com

Regular Expressions
Python Regex

- The Python "**re**" module provides regular expression support.

```
import re
```

- In Python a regular expression search is typically written as:

```
match = re.search(pat, str)
```

  - The **re**.**search**() method takes a regular expression pattern and a string and searches for that pattern within the string.
  - If the search is successful, search() returns a match object or None otherwise.

Regular Expressions
Python Regex

- The search for a pattern is usually immediately followed by an **if-statement** to test if the search succeeded, as shown in the following example which searches for the pattern '**word:**' followed by a **3 letter word**:

```
str = 'an example word:cat!!'
match = re.search(r'word:\w\w\w', str)
# If-statement after search() tests if it succeeded
if match:
  print 'found', match.group() ## 'found word:cat'
else:
  print 'did not find'
```

- The '**r**' at the start of the pattern string designates a python "**raw**" string. It is recommend to always write pattern strings with the 'r' to avoid any unwanted expansion.

Regular Expressions
Python Regex

- The power of regular expressions is that they can specify **patterns**, not just fixed characters. Here are the most basic patterns which match single chars:
  - a, X, 9 ← ordinary characters just match themselves exactly (literals).
  - . ^ $ * + ? { [ ] \ | ( ) ← meta-characters which do not match themselves because they have special meanings.
    - To treat meta-characters as literals **escape** them with a \
      - \^ \{ \$ \\

Regular Expressions
Python Regex

- The power of regular expressions is that they can specify **patterns**, not just fixed characters. Here are the most basic patterns which match single chars:
  - \w (lowercase w) ← matches a single word character: a letter or digit or underbar [a-zA-Z0-9_].
  - \W ← matches any non-word character.
  - \d ← decimal digit [0-9]
  - \D ←matches any non-digit character.

Regular Expressions
Python Regex

- The power of regular expressions is that they can specify **patterns**, not just fixed characters. Here are the most basic patterns which match single chars:
  - . (a period) ← matches any single character except newline '\n‘
  - ^ = start, $ = end ← match the start or end of the string
  - \t, \n, \r ← tab, newline, return
  - \s (lowercase s) ← matches a single whitespace character [\t\n\r\f\v].
  - \S (upper case S)← matches any non-whitespace character [^ \t\n\r\f\v]
  - \b ← boundary between word and non-word character
    - E.x r'\bfoo\b' matches 'foo', 'foo.', '(foo)', 'bar foo baz' but not 'foobar' or 'foo3'

Regular Expressions
Python Regex

- The basic rules of regular expression search for a pattern within a string are:
  - The search proceeds through the string from start to end, stopping at the first match found.
  - All of the pattern must be matched, but not all of the string.

```
## Search for pattern 'iii' in string 'piiig'.
## All of the pattern must match, but it may appear anywhere.
## On success, match.group() is matched text.
match = re.search(r'iii', 'piiig') =>  found, match.group() == "iii"
match = re.search(r'igs', 'piiig') =>  not found, match == None
```

Regular Expressions
Python Regex

- The basic rules of regular expression search for a pattern within a string are:
  - The search proceeds through the string from start to end, stopping at the first match found.
  - All of the pattern must be matched, but not all of the string.

```
## . = any char but \n
match = re.search(r'..g', 'piiig') =>  found, match.group() == "iig"

## \d = digit char, \w = word char
match = re.search(r'\d\d\d', 'p123g') =>  found, match.group() == "123"
match = re.search(r'\w\w\w', '@@abcd!!') =>  found, match.group() == "abc"
```

Regular Expressions
Python Regex

- **Repetition**
  - Things get more interesting when you use + and * to specify repetition in the pattern
    - + ← 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's (**Greedy**).
    - * ← 0 or more occurrences of the pattern to its left (**Greedy**).
    - ? ← match 0 or 1 occurrences of the pattern to its left.

```
## i+ = one or more i's, as many as possible.
match = re.search(r'pi+', 'piiig') =>  found, match.group() == "piii"

## Finds the first/leftmost solution, and within it drives the +
## as far as possible (aka 'leftmost and largest').
## In this example, note that it does not get to the second set of i's.
match = re.search(r'i+', 'piigiiii') =>  found, match.group() == "ii"
```

Regular Expressions
Python Regex

- **Repetition**
  - Things get more interesting when you use + and * to specify repetition in the pattern
    - + ← 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's (**Greedy**).
    - * ← 0 or more occurrences of the pattern to its left (**Greedy**).
    - ? ← match 0 or 1 occurrences of the pattern to its left.

```
## \s* = zero or more whitespace chars
## Here look for 3 digits, possibly not separated or separated by whitespace.
match = re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx') =>  found, match.group() == "1 2   3"
match = re.search(r'\d\s*\d\s*\d', 'xx12  3xx') =>  found, match.group() == "12  3"
match = re.search(r'\d\s*\d\s*\d', 'xx123xx') =>  found, match.group() == "123"
```

Regular Expressions
Python Regex

- **Repetition**
  - Things get more interesting when you use + and * to specify repetition in the pattern
    - + ← 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's (**Greedy**).
    - * ← 0 or more occurrences of the pattern to its left (**Greedy**).
    - ? ← match 0 or 1 occurrences of the pattern to its left.

```
## ^ = matches the start of string, so this fails:
match = re.search(r'^b\w+', 'foobar') =>  not found, match == None
## but without the ^ it succeeds:
match = re.search(r'b\w+', 'foobar') =>  found, match.group() == "bar"
```

Regular Expressions
Python Regex

- Suppose you want to find the email address inside the string 'xyz alice-b@google.com purple monkey'.

```
str = 'purple alice-b@google.com monkey dishwasher'
match = re.search(r'\w+@\w+', str)
if match:
  print match.group()  ## 'b@google'
```

  - The search does **not get the whole email address** in this case because the \w does not match the '-' or '.' in the address.

Regular Expressions
Python Regex

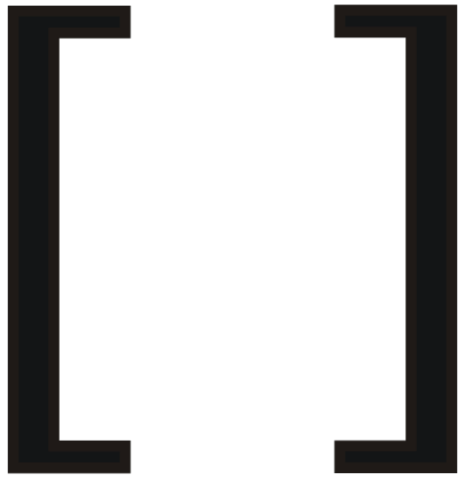
- **Square brackets** can be used to indicate a **set of chars**, so [abc] matches 'a' or 'b' or 'c'. Special characters **lose** their **special meaning** inside sets. For example, [**(+*)**] will match any of the **literal characters** '(', '+', '*', or ')' . The codes \w, \s work inside square brackets.

- Square brackets are an easy way to add '.' and '-' to the set of chars which can appear around the @ with the pattern r'**[\w.-]+@[\w.-]+**' to get the **whole email address**:

```
match = re.search(r'[\w.-]+@[\w.-]+', str)
if match:
  print match.group()  ## 'alice-b@google.com'
```

Regular Expressions
Python Regex

- **Square brackets** can be used also to indicate a **range**, so **[a-z]** matches **all lowercase letters**.
  - To use a **dash** without indicating a range (as a **literal**), put the dash last, e.g. **[abc-]**.
  - An up-hat (**^**) at the start of a square-bracket set **inverts** it, so [**^ab**] means any char except 'a' or 'b'.



Regular Expressions
Python Regex

- **Group Extraction:**
  - The "**group**" feature of a regular expression allows you to pick out parts of the matching text.
  - Suppose for the emails problem that we want to extract the **username** and **host** separately.
    - To do this, add parenthesis ( ) around the username and host in the pattern, like this: r'**(**[\w.-]+**)**@**(**[\w.-]+**)**'. In this case, pattern is not changed, instead **logical "groups"** are established inside of the match text.

```
str = 'purple alice-b@google.com monkey dishwasher'
match = re.search('([\w.-]+)@([\w.-]+)', str)
if match:
  print match.group()   ## 'alice-b@google.com' (the whole match)
  print match.group(1)  ## 'alice-b' (the username, group 1)
  print match.group(2)  ## 'google.com' (the host, group 2)
```

Regular Expressions
Python Regex

- **findall:**
  - findall() is probably the single most powerful function in the **re** module.
    - findall() finds *all* the matches and returns them as a **list of strings**, with each string representing one match.

```
## Suppose we have a text with many email addresses
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'

## Here re.findall() returns a list of all the found email strings
emails = re.findall(r'[\w\.-]+@[\w\.-]+', str) ##['alice@google.com','bob@abc.com']
for email in emails:
  # do something with each found email string
  print email
```

Regular Expressions
Python Regex

- **findall With Files:**
  - For files, you may think of writing a **loop** to iterate over the lines of the file, and you could then call **findall**() on each line to find a certain pattern in a file.
  - Instead of looping, just **feed** the **whole file** text to **findall**():

```
# Open file
f = open('test.txt', 'r')
# Feed the file text into findall(); it returns a list of all the found strings
strings = re.findall(r'some pattern', f.read())
```

Regular Expressions
Python Regex

- **findall and Groups:**
  - The **parenthesis ( )** group mechanism can be **combined** with **findall**().
    - If the pattern includes 2 or more parenthesis groups, then instead of returning a list of strings, findall() returns a **list** of ***tuples***.

```
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
tuples = re.findall(r'([\w.-]+)@([\w.-]+)', str)
print tuples  ## [('alice', 'google.com'), ('bob', 'abc.com')]
for tuple in tuples:
  print tuple[0]  ## username
  print tuple[1]  ## host
```

- Writing a **?:** at the **start** of a paren () group will **not make it count** as a group result
  - e.g. (?:)

Regular Expressions
Python Regex

- **Options:**
  - The **re** functions take **options** to modify the behavior of the pattern match. The option flag is added as an extra argument to the **search**() or **findall**() etc.
    - e.g. **re.search(*pat*, *str*, *re.IGNORECASE*)**.
      - **IGNORECASE** ← ignore upper/lowercase differences for matching, so 'a' matches both 'a' and 'A'.
      - **DOTALL** ← allow dot (.) to match newline.
      - **MULTILINE** ← Within a string made of many lines, allow ^ and $ to match the start and end of each line. Normally ^/$ would just match the start and end of the whole string.

Regular Expressions
Python Regex

- **Greedy vs. Non-Greedy:**
  - A more advanced feature in regular expressions is the greedy and lazy quantifiers.
    - If there is a string = '**foo and so on**' and there is need to match the following pattern **<.*>**
      - The result here will be all the string '**foo and so on**' because **.*** goes as far as is it can, instead of stopping at the first **>** (aka it is "greedy").
      - To solve this issue there is an extension to regular expression where you add a **?** at the end, such as **.*?** or **.+?**, changing them to be non-greedy (aka it is "Lazy") so **<.*?>** will match **   ** in case of a global match.

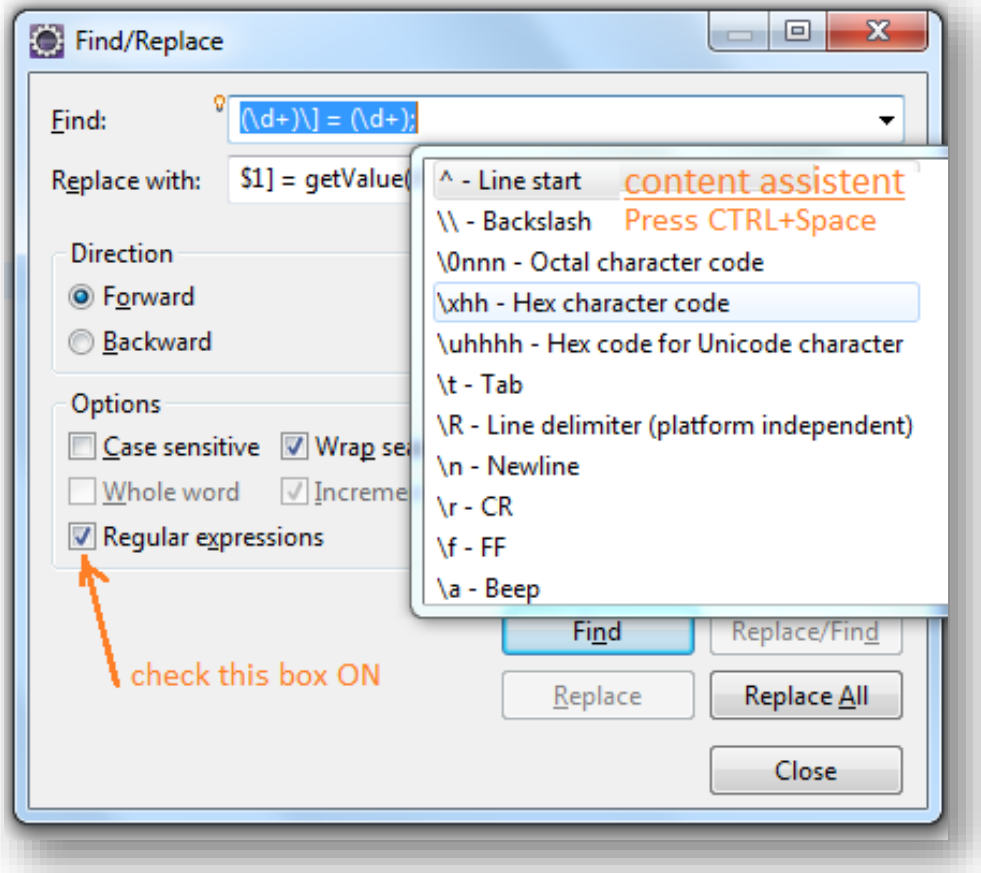
Regular Expressions
Python Regex

- **Substitution:**
  - The **re.sub**(***pat***, ***replacement***, ***str***) function searches for all the instances of pattern in the given string, and replaces them.
    - The replacement string can include '**\1**', '**\2**' which refer to the text from **group(1)**, **group(2)**, and so on from the original matching text.
    - E.x. Searches for all the email addresses, and changes them to keep the user (\1) but have **iti.com** as the **host**.

```
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
## re.sub(pat, replacement, str) -- returns new string with all replacements,
## \1 is group(1), \2 group(2) in the replacement
print re.sub(r'([\w\.-]+)@([\w\.-]+)', r'\1@iti.com', str)
## purple alice@iti.com, blah monkey bob@iti.com blah dishwasher
```

Regular Expressions
Related Topics

- **Eclipse Example:**
  - Eclipse search uses Java style regex, for more help -> LINK:

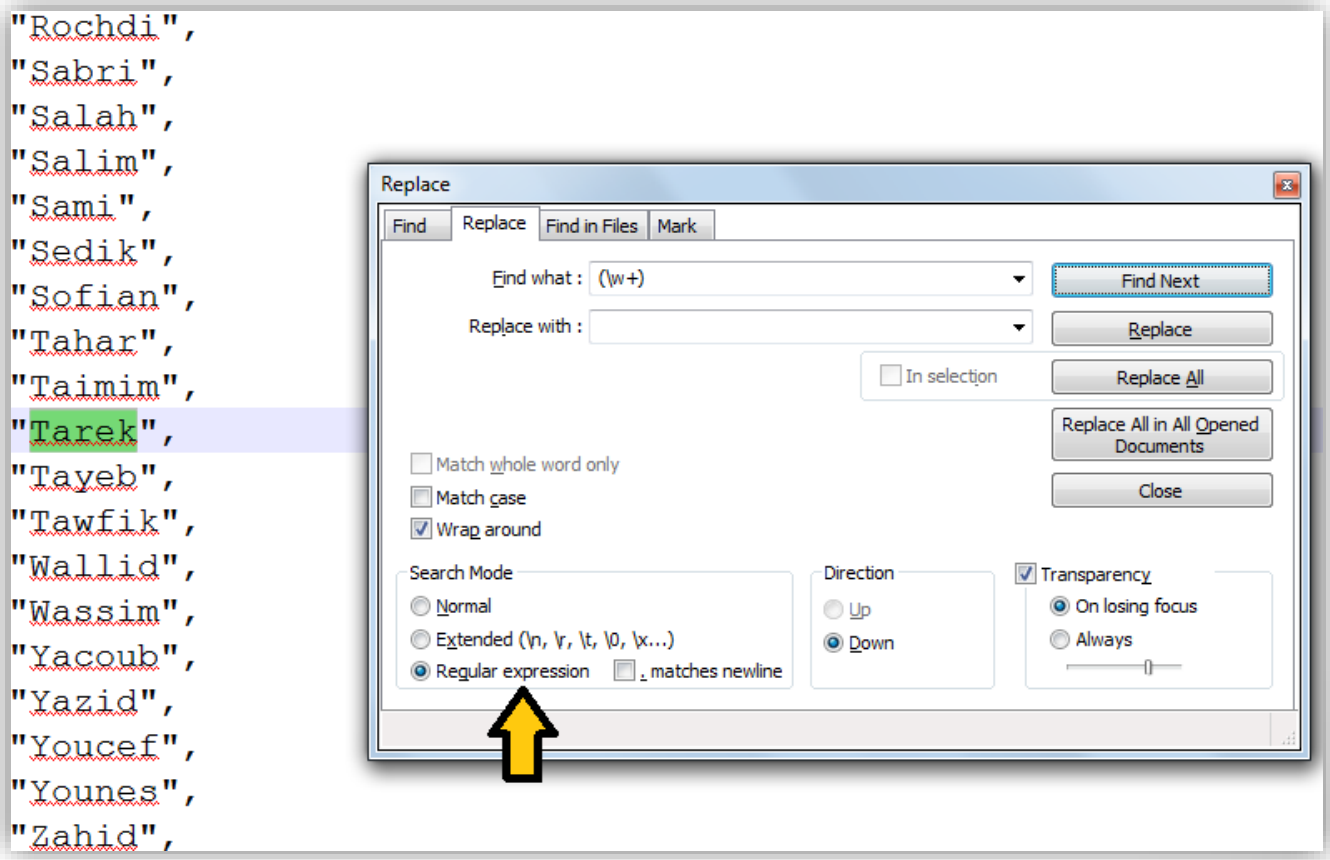


Regular Expressions
Related Topics

- **Eclipse Example:**
  - To practice a different flavor of regex try this exercise:
    - Download the following C file:
      - https://drive.google.com/file/d/0B6Hf8UvSSqTXYlBMb0VJeGdTSFk/view?usp=sharing
    - Using regex change the initialization of the students array of sturcts to be initialized using the struct fields names:
      - {.student_number=1,.student_name="Abida",.student_age=13}
    - Instead of:
      - {1,"Abida",13}

Regular Expressions
Related Topics

- **Notepad++ Example:**
  - Notepad++ search uses PCRE regex, for more help:
    - http://docs.notepad-plus-plus.org/index.php/Regular_Expressions



Regular Expressions
Related Topics

- **Useful References:**
  - For more help about regex, please see python docs:
    - https://docs.python.org/2.7/library/re.html?highlight=regular%20expressions
    - https://docs.python.org/2.7/howto/regex.html?highlight=regular%20expressions
  - Ahmed ElArabawy session about regex in general:
    - http://linux4embeddedsystems.com/courses/pluginfile.php/976/mod_label/intro/C_102_Lec_13_Regular_Expressions_updated.pdf
  - For Regex Cheat Sheet:
    - http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/pdf/

LAB – 7
REGEX LAB

Regex Lab

- Please download the lab from the following link:
  - https://drive.google.com/file/d/0B6Hf8UvSSqTXN0NseVI1V0duSTQ/view?usp=sharing
  - Complete the script **Regex_lab.py** in **60** mins and send your solution on the following email:
  - Omar.Soliman@imtSchool.com with the following subject :
    - If you are from ITI-Smart track ES:
      - [ITI_SV_39][PY-regex]yourfullname
    - If you are from ITI-Nasr City track ES:
      - [ITI_NC_39][PY-regex]yourfullname

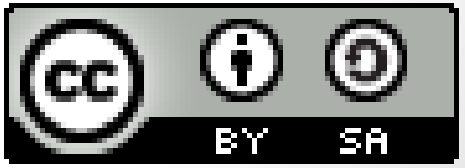
LAB – 7
REGEX LAB

What's Next ?

- Get Certified With:
  - https://www.edx.org/course/learn-program-using-python-utarlingtonx-cse1309x
  - https://www.coursera.org/course/interactivepython1
  - https://www.coursera.org/course/interactivepython2
- For more about python:
  - Python Cookbook, 2nd Edition
- For practicing python and ready made recipes:
  - https://automatetheboringstuff.com/
  - https://learnpythonthehardway.org/book/
  - http://code.activestate.com/recipes/langs/

Copy Rights

- This material is under the creative commons Attribution-ShareAlike license.
  - https://creativecommons.org/licenses/by-sa/4.0/



Eng. Mohammad A.Hekal: embeddedgeek.34@gmail.com
Omar Soliman: Omar.Soliman@imtSchool.com