**Python Part 4 - Pattern Matching**

|  |  |
| --- | --- |
| **Pattern Matching with regex**  Python's pattern matching with regular expressions is based on Perl's regular expressions.   * To use these capabilities, we must import the regular expression module:   **import re**   * To search for a pattern within a string:   **re.search**(*pattern,string,flags*)  returns a **match object** or **None.** *flags* are an optional argument.   * To match only at the beginning of the string:   **re.match**(*pattern,string,flags*)  returns a **match object** or **None.** *flags* are an optional argument.   * When a **match object** is returned (i.e., it isn't None), it provides these methods:   group() - returns the string for the entire match  group(*k*) - returns the *k*th subgroup. *k* begins at 1.  groups() - returns a tuple containing all matching subgroups   * We usually save the results of one of those functions in a match object   *matchObj* = re.search(*pattern,string*) | **Example 1: introduction to search and match**  # we will assume that re has been imported in all subsequent  # regex examples  import re  text1 = "Please call me at 210-555-1234"  matchObj = re.search( r'\d{3}-\d{3}-\d{4}', text1)  if matchObj != None:  print( matchObj.group())  210-555-1234  text2 = "Please call me as soon as possible."  matchObj = re.search( r'\d{3}-\d{3}-\d{4}', text2)  if matchObj != None:  print( matchObj.group())  else:  print("not found")  not found  text3 = "210-555-6666"  matchObj = re.match( r'\d{3}-\d{3}-\d{4}', text3)  if matchObj != None:  print("matched")  matched  matchObj = re.match( r'\d{3}-\d{3}-\d{4}', text1)  if matchObj != None:  print("matched")  else:  print("not matched")  not matched  text4 = "210-555-6666 is my number"  matchObj = re.match( r'\d{3}-\d{3}-\d{4}', text4)  if matchObj != None:  print("matched")  else:  print("not matched")  matched |
| **Pattern Matching with regex continued**   * For efficiency, we can define a regular expression object using **re.compile**(*regEx*)   *regExObj* = re.**compile**(r'\d\d\d\d')   * We can use the regex object to search for a regular expression in a string   *regExObj*.**search**(*string*)   * We can use that regex object to match from the beginning of a string   *regExObj*.**match**(*string*)   * We can find all occurrences that match that regular expression using   *regExObj*.**findall**(*string*)  This returns a tuple containing all the matches. | **Example 2: compiling a regex for efficiency**  # we will assume that re has been imported in all subsequent  # regex examples  import re  # define a regular expression to match a phone number 999-999-9999  phoneRE = re.compile(r'\d{3}-\d{3}-\d{4}')  text5 = "Please call me at 210-555-1234 or 830-555-6789"  matchObj = phoneRE.search(text5)  if matchObj != None:  print( matchObj.group())  210-555-1234  ### return all matches  resultT = phoneRE.findall(text5)  print(resultT)  resultT = phoneRE.findall(text2)  print (resultT)  ['210-555-1234', '830-555-6789']  [] |
| **Saving Values Using Grouping**  It is often necessary to save particular portions of what is matched. This can be done by placing parentheses around the portions to be saved. You can reference each group*k* by specifying  *matchObj*.**group**(*k*)  The first matched group will be  *matchObj*.group(1).  *matchObj* will be **None** if the regular expression did not match.  We can get a tuple containing all the matched groups by specifying:  *matchObj*.**groups**() | **Example 3: groups**  # define a regular expression to match an phone number 999-999-9999  # saving the area code and number  text1 = "Please call me at 210-555-1234"  phoneRE = re.compile(r'(\d{3})-(\d{3}-\d{4})')  matchObj = phoneRE.search(text1)  if matchObj != None:  print( "Area Code=", matchObj.group(1)  , "Phone=", matchObj.group(2))  area, phone = matchObj.groups()  print( "Area Code=", area  , "Phone=", phone)  Area Code= 210 Phone= 555-1234  Area Code= 210 Phone= 555-1234 |
| **Special Characters in Regular Expressions**  **\d** matches any numeric digit 0 thru 9  **\D** matches any character that is **not** a numeric digit  **\w** matches any letter, numeric digit or underscore  **\W** matches any character that is **not** a letter, numeric digit or underscore  **\s** matches a space, tab or newline character  **\S** matches any character that is **not** a space, tab or newline character  **. a dot** any character other than a newline character  **(*ex*)** defines a group for *ex*  **[*values*]** matches any character listed between the brackets  **[^*values*]** matches any character that is **not** listed between the brackets | **Example 4: special characters in regex**  # define a regular expression to match an ABC123 ID  abc123RE = re.compile(r'[a-z]{3}\d{3}')  text4 = "My ID is xyz123."  matchObj = abc123RE.search(text4)  if matchObj != None:  abc123Id = matchObj.group()  else:  abc123Id = None  >>> abc123Id  xyz123  # define a regular expression which matches Ho Ho Ho with  # any of these characters in between the words:  # comma or exclamation point  # It also has a space between those characters  # and the next Ho.  # The last Ho must be immediately followed by a !  # It should be case insensitive (specify re.I)  hohohoRE = re.compile(r'ho[,!]\sho[,!]\sho!', re.I)  matchObjHo = hohohoRE.search('Santa said, "Ho! Ho! Ho!"') |
| **Special Characters Applied to a Preceding Expression**  The following special characters are applied to a preceding expression:  **?** matches **zero** or **one** of the preceding expression  **\*** matches **zero** or **more** of the preceding expression  **+** matches **one** or **more** of the preceding expression  **{*n*}** matches **exactly *n*** of the preceding expression  **{*n,*}** matches ***n* or more** of the preceding expression  **{*,m*}** matches ***0* to *m*** of the preceding expression  **{*n,m*}** matches **at least *n* and at most *m*** of the preceding expression  **$** preceding expression must match at the end of the string  **| *ex*** matches the preceding expression or the following one | **Example 5: more special characters in regex**  # define a regex that searches for Mickey Mouse or Minnie Mouse  # Mouse is optional  mouseRE = re.compile(r'(Mickey|Minnie)\s?(Mouse)?')  mouseMO = mouseRE.search("Goofy yelled at Mickey Mouse")  >>> mouseMO.group(1)  Mickey  >>> mouseMO.group(2)  Mouse  >>> mouseMO = mouseRE.search("Pluto licked Minnie's hand")  >>> mouseMO.group(1)  Minnie  >>> mouseMO.group(2)  None |
| **Values that Must be Escaped to be Matched**  There are several symbols that have special meanings to regular expressions. To match the literal values, these must be escaped:  **\.** matches a period  **\\** matches a backslash  **\(** matches a left parenthesis  **\)** matches a right parenthesis  **\[** matches a left bracket  **\]** matches a right bracket  **\{** matches a left curly brace  **\}** matches a right curly brace  **\^** matches a carat  **\$** matches a dollar sign | **Example 6: escaping characters**  # define a regular expression for a phone number (999)-999-9999  # saving the area code and number  import re  phoneRE = re.compile(r'\((\d{3})\)-(\d{3}-\d{4})')  text1 = "Please call me at (210)-555-1234"  phoneMO = phoneRE.search(text1)  if phoneMO!= None:  area, phone = phoneMO.groups()  print( "Area Code=", area  , "Phone=", phone)  Area Code= 210 Phone= 555-1234 |
| **Splitting a string using re.split()**  **re.split(***regExpr, string***)**  *regExpr* is a regular expression. *string* is the string to be split. | **Example 7: regex split**  # Split on semicolon, comma, period or space. Also ignore 0 to many  # spaces after the delimiter.  import re  text3 = "He loved playing basketball; however, he hated watching it on TV."  wordM = re.split("[;,\s\.]\s\*", text3)  >>> wordM  ['He', 'loved', 'playing', 'basketball', 'however', 'he', 'hated', 'watching', 'it', 'on', 'TV', ''] |
| **Greedy vs Nongreedy Matches**  Like regular expression matching in other languages and tools, Python uses a greedy matching mode for multi-occurring patterns unless told to be nongreedy. With **greedy matching mode**, it tries to match as much text as possible.  In the first example on the right, our pattern is matching anything followed by a space. The second group then matches anything.  Because the ".\*\s" in the first group is greedy, it matched  "hello there " instead of just "hello ".  Placing a ? after a multi-occurring pattern, causes **re** to use a nongreedy matching mode. This is not the same meaning of ? as above. With **nongreedy matching mode**, it tries to match as little text as possible. | **Example 8: greedy vs nongreedy default is greedy (1)**  Some examples using greedy and non-greedy matches  str = “hello there world”;   |  |  |  | | --- | --- | --- | | **pattern** | **group(1)** | **group(2)** | | r’(.\*\s)(.\*)’ | ‘helloΔthereΔ’ | ‘world' | | r'(.\*?\s)(.\*)' | 'helloΔ' | 'thereΔworld' |   str = "Ho! Ho! Ho! Ho! Ho! "; {atleast, atmost}   |  |  | | --- | --- | | **pattern** | **group(1)** | | r'((Ho!\s){2,5})' | 'Ho! Ho! Ho! Ho! Ho! ' | | r'((Ho!\s){2,5}?)' | 'Ho! Ho!Δ' | |
|  | **Example 9: greedy vs nongreedy**  # Greedy matching lab.cs.utsa.edu  # The matching pattern matches any character until a dot is found.  # With greedy, the .\* matches as much text as possible.  location = "lab.cs.utsa.edu"  gDotRE = re.compile(r'(.\*)\.')  gDotMO = gDotRE.search(location)  print (gDotMO.group())  **lab.cs.utsa.**  # Nongreedy matching lab.cs.utsa.edu  location = "lab.cs.utsa.edu"  ngDotRE = re.compile(r'(.\*?)\.')  ngDotMO = ngDotRE.search(location)  print (ngDotMO.group())  **lab.** |
| **More Examples of Greedy vs Nongreedy** | **Example 10: more greedy vs nongreedy**  # Greedy matching of Ho!  text = "Ho! Ho! Ho! Ho! Ho! "  gSantaRE = re.compile(r'(Ho!\s){2,5}')  gSantaMO = gSantaRE.search(text)  print (gSantaMO.group())  **Ho! Ho! Ho! Ho! Ho!**  # Nongreedy matching of Ho!  text = "Ho! Ho! Ho! Ho! Ho! "  ngSantaRE = re.compile(r'(Ho!\s){2,5}?')  ngSantaMO = ngSantaRE.search(text)  print (ngSantaMO.group())  **Ho! Ho!**  # Matching of ho, ho, ho, kid  text = "ho, ho, ho, kid"  ngSantaRE = re.compile(r'(ho,\s){1,5}?kid')  ngSantaMO = ngSantaRE.search(text)  print (ngSantaMO.group())  **ho, ho, ho, kid**  Why?  **??** |
| **Returning Multiple of the same Match**  Suppose we have the recipients of medals in any order on a text line. Example:  GOLD Fred SILVER Barney  SILVER Shaggy GOLD Scooby BRONZE Fred  We can use  *regExObj*.**findall**(*string*). If a single match has multiple groups (e.g., medal, recipient), **findall** will return a list of multiple tuples. | **Example 11: using findall to return multiple tokens**  import re  # what does this pattern do?  # **matches leading space, matches type of metal;**  medalRE = re.compile(r'\s\*(GOLD|SILVER|BRONZE)\s(\w\*)')  textLine = "SILVER Shaggy GOLD Scooby BRONZE Fred"  resultM = medalRE.findall(textLine)  print(resultM)  for oneMatchRes in resultM :  medal, recipient = oneMatchRes  print(medal, recipient)  [('SILVER', 'Shaggy'), ('GOLD', 'Scooby'), ('BRONZE', 'Fred')]  SILVER Shaggy  GOLD Scooby  BRONZE Fred |
| **How could split be used to separate those?**  **split**(delim,n) Creates a list of n tokens with the remaining text placed in the last element of the list. The resulting list usually has n+1 entries unless there isn't any remaining text. | **Example 12: using multiple calls of split to return multiple tokens**  # loop while there are at least two parts  import re  textLine = "SILVER Shaggy GOLD Scooby BRONZE Fred"  rest = textLine  **tokenM = rest.split(‘ ’, 2)** # tokenM[0] is medal, tokenM[1] is recipient  while len(tokenM) >= 2:  medal = tokenM[0]  recipient = tokenM[1]  print (medal, recipient)  if len(tokenM) <= 2:  break  rest = tokenM[2]  tokenM = rest.split(' ',2) |
| **Symbols that are delimiters and are also needed**  Suppose we have an expression like  perimiter = 2\*length + 2\*width  We want each token, but our tokens aren't always surrounded by spaces. | **Example 13: Returning Delimiters**  import re  # define a regex which  # - skips over white space  # - accepts either \w\* or one of our operators (jam everything after into variable)  tokenRE = re.compile(r'\s\*([+\*=]|\w\*)(.\*)’) |
| **Substituting Text**   * To return a string with a pattern replaced by *repl*, use:   **re.sub**(*pattern,repl,string,*max=0)  If max is defaulted to zero, all occurrences are replaced. If max=*k,* only *k* occurrences are replaced. | **Example 14: replacing text**  import re  text = "eight men lifted weights"  string1 = text  string1 = re.sub("eight", "seven", string1, 1)  print("modified: '"+ string1 +"' \n")  text = "eight men lifted weights"  string2 = text  string2 = re.sub("eight", "seven", string2)  print("modified: '"+ string2 +"' \n")  modified: 'seven men lifted weights'  modified: 'seven men lifted wsevens' |
|  |  |
| **Non-exact matches – difflib NEW**  Python has several libraries which support non-exact matches. The **difflib** module has several methods.  To get the difference of two strings separated as lists of text lines:  *dobj* = difflib.Differ()  *result*=list(*dobj.*compare(*list1, list2*))  Results contain lines with a two character code   |  |  | | --- | --- | | **Code** | **Meaning** | | '- ' | line unique to *list1* | | '+ ' | line unique to *list2* | | ' ' | line is in both lists | | '? ' | line shows where the two lines are different | | **Example 15: difflib to show differences**  import difflib  text1 = """This is an example where the first line is the same, but  the second line is different.  How about the third line? It is in this only.  Oh boy! It is the same here!  The last line in text1.""".splitlines()  text2 = """This is an example where the first line is the same, but  the second line is a little different.  Oh boy! It is the same here!  This line isn't in text1.  This line also isn't in text1.  The last line in text1.""".splitlines()  dobj = difflib.Differ()  result = list(dobj.compare(text1, text2))  from pprint import pprint  pprint(result)  **[' This is an example where the first line is the same, but',**  **'- the second line is different.',**  **'+ the second line is a little different.',**  **'? +++++++++\n',**  **'- How about the third line? It is in this only.',**  **' Oh boy! It is the same here!',**  **"+ This line isn't in text1.",**  **"+ This line also isn't in text1.",**  **' The last line in text1.']** |
| **Non-exact matches - difflib continued**  difflib also provides a function for checking sequences of matched characters.  from difflib import SequenceMatcher as SM  *result =* SM(None, *str1, str2*).ratio()  Returns a float, between 0 and 1, measuring the *similarity* of the sequences. In general, matches less than 0.6 are not good.  In some situations, you may prefer to specify the first parameter to eliminate characters of no interest:  *result =* SM(lambda x: x == " "  , *str1, str2*).ratio()  That would not consider blank spaces. | **Example 16: using SequenceMatcher**  from difflib import SequenceMatcher as SM  str1 = "SAN ANTONIO"  str2 = "SAN ANTONOI"  str3 = "SNA ANTONIO"  str4 = "SAN DIEGO"  str5 = "SAN FRANCISCO"  print (str1, str2, SM(None, str1, str2).ratio())  print (str1, str3, SM(None, str1, str3).ratio())  print (str2, str3, SM(None, str2, str3).ratio())  print (str1, str4, SM(None, str1, str4).ratio())  print (str1, str5, SM(None, str1, str5).ratio())  **SAN ANTONIO SAN ANTONOI 0.90909090909090906**  **SAN ANTONIO SNA ANTONIO 0.90909090909090906**  **SAN ANTONOI SNA ANTONIO 0.81818181818181823**  **SAN ANTONIO SAN DIEGO 0.5**  **SAN ANTONIO SAN FRANCISCO 0.58333333333333337** |
| **©2020 Larry W. Clark,** UTSA CS students may make copies for their personal use | |