**Python Part 3 – Regular Expressions**

|  |  |
| --- | --- |
| The ability to match patterns in texts has many uses in systems programming. As with sed and awk, **regular expressions** can be used in Python for many of our pattern matching needs.  The **re** module contains all the necessary functions/methods for regular expressions.  **Note:** Tools such as <http://regex101.com> are great for learning and/or validating regex. | Five major uses of regular expressions:  **parsing** read in data from canonical format  **searching** locating text within files/streams  **search/replace** locating texts and replacing it  **splitting** separating text into segments  **validation** ensuring that data is in an acceptable format  All of the above are easily implemented with Python's **re** module. |
| **Basic Pattern Matching**  re.search(*pattern, string, flags*)   * Returns a **match object\*** if *pattern* matches anywhere in *string*. Returns None otherwise. * *flags* is an optional argument to modify how the search is performed (we'll see some flags later) * **Note:** the re.match() method is similar, but matches *starting at the beginning of the string*.   \* Many re methods return match objects which include many useful methods (we'll see them later). For now, we can treat them as truthy/falsey. | **Example 3-1:** Searching for patterns  $ vi search.py  import re  string1 = "Hello world!"  string2 = "Hi Earth!"  # search for the fixed pattern "Hello" in strings  if re.search("Hello", string1):  print("'" + string1 + "' contains 'Hello'")  else:  print("'" + string1 + "' does not contain 'Hello'")  if re.search("Hello", string2):  print("'" + string2 + "' contains 'Hello'")  else:  print("'" + string2 + "' does not contain 'Hello'")  $ python3 ./search.py  'Hello world!' contains 'Hello'  'Hi Earth!' does not contain 'Hello' |
| **Raw Strings and Special Characters**  Since Python will automatically process escape sequences beginning with slashes, it is beneficial to use **raw strings** for regular expressions. Raw strings are preceded with an 'r' character and are not processed. If not using a raw string for a pattern, special symbols must be escaped (e.g., \\S).  re.search(r"\w+"*,* "hello world") | |
| **Characters and Classes**  Besides literal characters, the special classes below can be used to match character sets.  **.** Matches any character except **\n**  **\w** Matches any Unicode "word" character (letter, numeric digit or underscore)  **\W** Matches any character that is **not** a Unicode "word" character  **\d** Matches any Unicode numeric digit 0 thru 9  **\D** Matches any Unicode character that is **not** a numeric digit  **\s** Matches a Unicode whitespace character (space, tab or newline character)  **\S** Matches any Unicode character that is **not** a whitespace  **[*characters*]** Matches any single character within the brackets  **[^*characters*]** Matches any single character **not** within the brackets  **Assertions**  Assertions signify a characteristic about the string being matched.  **^** Matches the literal start. This is not the first character.  **$** Matches the literal end. This is not the last character. | **Example 3-2:** Using classes and assertions in patterns  $ vi classes.py  import re  string = "xzi415 is not my student ID. I wonder who it belongs to."  # find lines beginning with an abc123 id  if re.search(r"^[a-zA-Z][a-zA-Z][a-zA-Z]\d\d\d", string):  print("'" + string + "' begins with a valid student ID")  else:  print("'" + string + "' doesn't begin with a valid student ID")  $ python3 ./classes.py  'xzi415 is not my student ID. I wonder who it belongs to.' begins with a valid student ID  Why not use \w or \D instead of [a-zA-Z]?  Those don’t match only letters a-z |
| **Quantifiers**  Quantifiers place bounds on how many times the *preceding* character/class should be matched.  **?** 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  **Grouping**  **(*pattern*)** Parentheses around a pattern defines a **group**.  Parentheses define a group from which the value matched can be extracted, or ***captured***, from the source string (see "Capturing").  Grouping can also be used to apply quantifiers to an entire expression.  **Alternation**  ***pattern1*|pattern*2*** Matches either the left **or** right pattern. Often used with groups. | **Example 3-3:** classes.py but with quantifiers  $ vi classes.py  import re  string = "xzi415 is not my student ID. I wonder who it belongs to."  # find lines beginning with an abc123 id  if re.search(r"^[a-zA-Z]{3}\d{3}", string):  print("'" + string + "' begins with a valid student ID")  else:  print("'" + string + "' doesn't begin with a valid student ID")  $ python3 ./classes.py  'xzi415 is not my student ID. I wonder who it belongs to.' begins with a valid student ID  **Example 3-4:** Alternation to match synonyms  $ vi plane.py  import re  # check for valid synonym for aiplane  def is\_airplane(string):  if re.search(r"(air(craft|plane)|jet)", string):  print("'" + string + "' has a plane in it!")  else:  print("'" + string + "' has no plane in it.")  is\_airplane("The airplane flew high")  is\_airplane("The aircraft flew high")  is\_airplane("The jet flew high")  is\_airplane("The air is high")  $ python3 ./plane.py  'The airplane flew high' has a plane in it!  'The aircraft flew high' has a plane in it!  'The jet flew high' has a plane in it!  'The air is high' has no plane in it. |
| **Substitution**  Substitution allows for searching over an input string for a pattern and replacing any matches with a specified replacement.  re.sub(*pattern, replacement, string, max, flags*)   * Substitutes the text matched with pattern with replacement in the string string up to a maximum of max times (optional). * **Note** that, unlike substitution in sed, this will substitute *all* matches by default. Modify the *max* parameter to change this. * Returns a **copy** of the resulting string with substitutions. * Influenced by flags (optional). | **Example 3-5:** Redacting IDs with substitution  $ vi substitution.py  import re  string = "My student ID is aaa999. Yours is bbb222"  # redact all abc123 IDs  substituted = re.sub(r"[a-zA-Z]{3}\d{3}", "[REDACTED]", string)  # the original string remains unchanged  print("string is '" + string + "'")  print("substituted is '" + substituted + "'")  $ python3 ./substitution.py  string is 'My student ID is aaa999. Yours is bbb222'  substituted is 'My student ID is [REDACTED]. Yours is [REDACTED]'  How could we modify the code above to substitute only the first occurrence of an ID on a line?  Use a quantifier |
| **Capturing**  Recall that parentheses can be used to define a group. If a group is matched, the actual string can be captured and extracted from the corresponding **MatchObject.** Note that mo represents a match object for the following methods.  *mo*.group(*group\_number*)   * Returns the **string** matched for the corresponding *group\_number.*   *mo*.groups()   * Returns a **tuple**of all the groups which can be referenced by indices (e.g., my\_groups[2])*.*   **Naming Groups**  When dealing with multiple groups, it can be useful to *name* each group rather than referencing them by number. This can be done with **?P<*name*>** where *name* is whatever you want to name the group.  mo = re.search(r"(?P<some\_name>\w+)\s\w", input)  *mo*.groupdict()   * Returns a **dictionary** of all named groups in the match object which can be referenced by their names (e.g., my\_groups["some\_name"])*.*   **Matching All Occurrences**  If multiple occurrences of a group are expected in the same text string, re.findall(pattern, string) and re.finditer(pattern, string) return a list of matching strings and an iterator of MatchObjects respectively. | **Example 3-6:** Capture values from passwd file to use later  $ vi capturing.py  import sys  import re  # skipping validation for brevity  pattern = r"^([^:]\*):[^:]\*:[^:]\*:[^:]\*:([^:]\*):[^:]\*:([^:]\*)$"  try:  with open(sys.argv[1], 'r') as passwd:  for line in passwd:  matches = re.search(pattern, line.rstrip())  if matches:  print(matches.group(2) + "'s username is " +  matches.group(1) + " and they use " +  matches.group(3) + " as their shell.")  except IOError:  print("Error opening file")  sys.exit(1)  $ cat passwd.sample  rslavin:x:123:1000:Rocky Slavin:/home/rslavin:/bin/zsh  maynard:x:124:1000:Hugh B. Maynard:/home/maynard:/bin/tcsh  clark:x:125:1000:Larry Clark:/home/clark:/bin/bash  abc123:x:5035:1001:Some Student:/home/abc123:/bin/tcsh  $ python3 ./capturing.py ./passwd.sample  Rocky Slavin's username is rslavin and they use /bin/zsh as their shell.  Hugh B. Maynard's username is maynard and they use /bin/tcsh as their shell.  Larry Clark's username is clark and they use /bin/bash as their shell.  Some Student's username is abc123 and they use /bin/tcsh as their shell.  Why is rstrip() being used?  Get rid of the white space at the end of the line |
| **Backreferencing**  Captured groups can be used immediately in the *same pattern* using backreferencing. If a group is defined, that group can be referenced later in the expression with \g where g is the group number.  To illustrate, consider a situation where you want to match only lines where a variable is used in assignment to itself immediately after the equals sign.  sum = sum + 1  We might use backreferencing to match the above line as follows.  re.search(r"\s\*(\w+)\s\*=\s\*\1.\*", input)  The above would work for the given example, but how might it run into problems for other situations we may want to match?  ??  Does the following fix the problems? Why or why not?  re.search(r"\s\*(\w+)\s\*=\s\*\1\W\*", input)  no  How do we solve the problem?  re.search(r"\s\*(\w+)\s\*=\s\*\1\W", input) | **Example 3-7:** Using backreferencing to use a captured value in the same pattern  $ vi backrefs.py  #!/usr/bin/env python3  import re  def test\_regex(regex, string):  if re.search(regex, string):  print("MATCH:\t\t" + regex + " \t'" + string + "'")  else:  print ("NO MATCH:\t" + regex + " \t'" + string + "'")  input1 = "sum = sum + 1"  input2 = "sum = sumthing + 1"  input3 = "sumthing = sum + 1"  regex1 = r"\s\*(\w+)\s\*=\s\*\1.\*"  regex2 = r"\s\*(\w+)\s\*=\s\*\1\W\*"  regex3 = r"\s\*(\w+)\s\*=\s\*\1\W"  test\_regex(regex1, input1)  test\_regex(regex1, input2)  test\_regex(regex1, input3)  print("---")  test\_regex(regex2, input1)  test\_regex(regex2, input2)  test\_regex(regex2, input3)  print("---")  test\_regex(regex3, input1)  test\_regex(regex3, input2)  test\_regex(regex3, input3)}  $ backrefs.py  MATCH: \s\*(\w+)\s\*=\s\*\1.\* 'sum = sum + 1'  MATCH: \s\*(\w+)\s\*=\s\*\1.\* 'sum = sumthing + 1'  NO MATCH: \s\*(\w+)\s\*=\s\*\1.\* 'sumthing = sum + 1'  ---  MATCH: \s\*(\w+)\s\*=\s\*\1\W\* 'sum = sum + 1'  MATCH: \s\*(\w+)\s\*=\s\*\1\W\* 'sum = sumthing + 1'  NO MATCH: \s\*(\w+)\s\*=\s\*\1\W\* 'sumthing = sum + 1'  ---  MATCH: \s\*(\w+)\s\*=\s\*\1\W 'sum = sum + 1'  NO MATCH: \s\*(\w+)\s\*=\s\*\1\W 'sum = sumthing + 1'  NO MATCH: \s\*(\w+)\s\*=\s\*\1\W 'sumthing = sum + 1' |
| **Splitting**  Grouping can be used to tokenize a string if the *tokens* have a known pattern. What if instead, we know only what the *delimiters* look like? The split() function can help with this.  re.split(*pattern, string, max, flags*)   * Returns a **list** of strings in *string* which are delimited by *pattern* up to a maximum of *max* splits (if *max* is not supplied, it will split as many times as possible). * Influenced by flags (optional).   **Example 3-8** is a different implementation of the passwd parser in Example 3-6. Instead of creating a long, tedious pattern defining the entire line to be parsed, we can just look for colons instead using split().  Here, the delimiter is a single character, but you can use more complex patterns if necessary (e.g., r"[:-,]"). | **Example 3-8:** Capture text from passwd file using splitting instead of grouping  $ vi splitting.py  import sys  import re  # skipping validation for brevity  # ~~pattern = r"^([^:]\*):[^:]\*:[^:]\*:[^:]\*:([^:]\*):[^:]\*:([^:]\*)$"~~  delimiter = ":"  try:  with open(sys.argv[1], 'r') as passwd:  for line in passwd:  tokens = re.split(delimiter, line.rstrip())  if tokens:  print(tokens[4] + "'s username is " +  tokens[0] + " and they use " +  tokens[6] + " as their shell.")  except IOError:  print("Error opening file")  sys.exit(1)  $ python3 ./splitting.py ./passwd.sample  Rocky Slavin's username is rslavin and they use /bin/zsh as their shell.  Hugh B. Maynard's username is maynard and they use /bin/tcsh as their shell.  Larry Clark's username is clark and they use /bin/bash as their shell.  Some Student's username is abc123 and they use /bin/tcsh as their shell.  In the grouping example, we used the following code to access the values we were interested in. Why are the indices 4, 0, and 6 now instead of 2, 1, and 3 as before?  print(matches.group(2) + "'s username is " +  matches.group(1) + " and they use " +  matches.group(3) + " as their shell.")  ?? |
| **Greedy vs Nongreedy**  By default, Python will try to match as much text as possible. This **greedy** matching can be a problem in situations where only the minimum matched text is desired. Quantifiers can be switched to nongreedy mode by following them with a "?".  In the first example on the right, the first group is trying to match anything followed by a space. The second group then matches anything after the first group. Because the ".\*\s" in the first group is greedy, it matched  "one two three " (i.e., everything up to the last space). The "\*" will match **as much as possible** while keeping the **entire expression true**.  If we place a "?" after the "\*" quantifier, it switches it to nongreedy. In the example, the first group now matches only "one " (i.e, everything up to the first space) and the second group matches the remainder of the text. **Note that it will start matching as far to the left as possible.** | **Example 3-9:** Greedy and nongreedy quantifiers  string = "one two three four";   |  |  |  | | --- | --- | --- | | **pattern** | **Group 1** | **Group 2** | | (.\*\s)(.\*) | 'one two three ' | four | | (.\*?\s)(.\*) | 'one ' | 'there world' |   string = "Hello Hello Hello Hello Hello Hello Hello";   |  |  | | --- | --- | | **pattern** | **Group 1** | | ((Hello\s){2,6}) | 'Hello Hello Hello Hello Hello Hello ' | | ((Hello\s){2,6}?) | 'Hello Hello ' |   string = "Goofy yelled at Mickey Mouse and the Moose";   |  |  | | --- | --- | | **pattern** | **Group 1** | | Mickey.+(Mo.se)/ | Moose | | Mickey.+?(Mo.se) | Moose | |
| **Exercise: Fill in the Chart** | |  |  |  | | --- | --- | --- | | **String** | **Pattern** | **Group 1** | | aooob | (ao+) | aooo | | aooob | (ao+?) | ao | | aooob | (ao+?)b | aooob | | aobaoob | (a.+)b | aobaooo | | aobaoob | (a.+?)b | ao | | aobaoob | (ao+)b | ao | |
| **Flags**  As with sed, you can apply flags to an expression to modify how it is matched. In Python, flags are passed as an optional parameter to functions/methods such as re.search().  **re.I or re.IGNORECASE** Ignore case  **re.M or re.MULTILINE** Makes ^ and $ match after and before newlines respectively instead of the beginning and end of the string.  Multiple flags can be combined with the "|" operator.  Why isn't there an re.G/re.GLOBAL flag (s/search/replace/**g** in sed)?  ?? | **Example 3-10:** Using flags to substitute over multi-line strings ignoring case  $ vi flags.py  import re  import sys  try:  with open(sys.argv[1], 'r') as infile:  # read the entire file into one string  # rstrip() to remove the empty string read() adds at EOF  all\_text = infile.read().rstrip()  print("File contents as a single variable:\n" + all\_text)  # add <h1> to the beginning of every line  subbed = re.sub(r"^", "<h1>", all\_text, flags=re.MULTILINE)  # add </h1> to the end of every line  subbed = re.sub(r"$", "</h1>", subbed, flags=re.M)  # redact all abc123s ignoring case  subbed = re.sub(r"[a-z]{3}\d{3}", "[REDACTED]", subbed, flags=re.M|re.I)  print("\nSubstitued:\n" + subbed)  except IOError:  print("Error opening file")  sys.exit(1)  $ cat abc123.txt  Soandso's ID is ZYX098  Tom's ID is xyz890  Sarah's ID is Cba321  $ python3 ./flags.py ./abc123.txt  File contents:  Soandso's ID is ZYX098  Tom's ID is xyz890  Sarah's ID is Cba321  Substitued:  <h1>Soandso's ID is [REDACTED]</h1>  <h1>Tom's ID is [REDACTED]</h1>  <h1>Sarah's ID is [REDACTED]</h1>  Why do we have to use "flags=" to pass the flags in?  ??.  **Note:** [Do not try to **p̡a̢rse̵** ͚̹̹H̵ͩ̇̉T̄͜Mͦ̕L̨ͭͬͦ̈ ̭ẃ̮̼̖̲̯͎͚i̸̙̣̫͕̘͙͖ț̷̩̩͖̟h̵̜͕ r̨e͝g͟ex̧.](https://stackoverflow.com/a/1732454) |
| **Compiling**  So far, we've used static functions from the re module to apply regular expressions (e.g., re.search()). This is useful when an expression only needs to be applied once or twice. However, each of these calls requires the pattern to be **compiled** to an internal format, repetition can be expensive.  If we know a regex is going to be used multiple times, it can be manually compiled into a regex object once and used subsequently without compilation, thus improving efficiency.  re.compile(*pattern, flags*)   * Returns a **compiled regex object** of *pattern* with flags applied optionally*.*   Most re functions have a corresponding **method** which can be called on a compiled regex (cr in this case).  cr.search(*string, start, end*)   * Returns a **match object** if *cr* matches anywhere in *string*. Returns None otherwise. * *start* and *end* denote an optional **slice** of string   cr.sub(*replacement, string, max*)   * Substitutes the text matched with cr with replacement in the string string up to a maximum of max times (optional). * **Note** that, unlike substitution in sed, this will substitute *all* matches by default. Modify the *max* parameter to change this. * Returns the resulting string with substitutions.   cr.split(*string, max*)   * Returns a **list** of strings in *string* which are delimited by crup to a maximum of *max* splits (if *max* is not supplied, it will split as many times as possible). | **Example 3-11:** Compiling an expression and using the split() method  $ vi compiling.py  import sys  import re  # skipping validation for brevity  compiled\_pattern = re.compile(":")  try:  with open(sys.argv[1], 'r') as passwd:  for line in passwd:  tokens = compiled\_pattern.split(line.rstrip())  if tokens:  print(tokens[4] + "'s username is " +  tokens[0] + " and they use " +  tokens[6] + " as their shell.")  except IOError:  print("Error opening file")  sys.exit(1)  $ python3 ./compiling.py ./passwd.sample  Rocky Slavin's username is rslavin and they use /bin/zsh as their shell.  Hugh B. Maynard's username is maynard and they use /bin/tcsh as their shell.  Larry Clark's username is clark and they use /bin/bash as their shell.  Some Student's username is abc123 and they use /bin/tcsh as their shell.  What's an example input for which this implementation will have a major benefit?  ?? |

**©2018 Rocky Slavin,** UTSA CS students may make copies for their personal use