Strings and Regular Expressions

September 20, 2024

1 Software Construction and Development Lab

Muhammad Saood Sarwar

Instructor (CS), National University Of Computer and Emerging Sciences, Peshawar.

2 String Manipulation

In **Software Construction**, string manipulation plays a crucial role in real-world applications, particularly when processing textual data, handling user inputs, or working with file contents. Understanding how to manipulate strings effectively is essential for building robust and efficient software.

2.0.1 String Manipulation

String manipulation refers to the process of modifying, parsing, and handling strings to extract useful information, format data, or generate dynamic content. In Python, strings are sequences of characters, and various operations can be performed on them, such as concatenation, slicing, replacing, and searching.

2.0.2 String Manipulation in Software Construction

In software construction, strings are often manipulated for several purposes:

1. **Input Validation**: When building applications, ensuring that user inputs are valid is critical. This often involves trimming whitespace, checking for valid characters, or transforming input into a consistent format. For example, converting email addresses to lowercase or stripping out unwanted spaces.
2. **Parsing Text Files**: Many software systems process text files, configuration files, or logs. String manipulation techniques allow developers to extract relevant information from large text datasets, parse structured files like CSV or JSON, and even filter out unnecessary content.
3. **Data Formatting**: Software applications often need to display data in a user-friendly format. Formatting strings, such as generating dynamic reports, creating readable logs, or formatting messages with placeholders, is key to presenting information clearly.
4. **Search and Replace**: In some applications, especially those dealing with documents or web pages, strings need to be searched for specific keywords or patterns and replaced with new content. This is commonly used in text processing tools, document generators, or automated workflows.

Strings in Python can be defined using either single or double quotations (they are functionally equivalent):

[1]:
```
x = 'a string'
y = "a string"
x == y
```

[1]:
```
True
```

In addition, it is possible to define multi-line strings using a triple-quote syntax:

[2]:
```
multiline = """
one
two
three
"""
```

With this, let's take a quick tour of some of Python's string manipulation tools.

2.1 Simple String Manipulation in Python

For basic manipulation of strings, Python's built-in string methods can be extremely convenient. If you have a background working in C or another low-level language, you will likely find the simplicity of Python's methods extremely refreshing. We introduced Python's string type and a few of these methods earlier; here we'll dive a bit deeper

2.1.1 Formatting strings: Adjusting case

Python makes it quite easy to adjust the case of a string. Here we'll look at the `upper()`, `lower()`, `capitalize()`, `title()`, and `swapcase()` methods, using the following messy string as an example:

[3]:
```
fox = "tHe qUICk bROWn fOx."
```

To convert the entire string into upper-case or lower-case, you can use the `upper()` or `lower()` methods respectively:

[4]:
```
fox.upper()
```

[4]:
```
'THE QUICK BROWN FOX.'
```

[5]:
```
fox.lower()
```

[5]:
```
'the quick brown fox.'
```

A common formatting need is to capitalize just the first letter of each word, or perhaps the first letter of each sentence. This can be done with the `title()` and `capitalize()` methods:

[6]:
```
fox.title()
```

[6]:
```
'The Quick Brown Fox.'
```

[7]:
```
fox.capitalize()
```

[7]:
```
'The quick brown fox.'
```

The cases can be swapped using the `swapcase()` method:

[8]:
```
fox.swapcase()
```

[8]:
```
'ThE QuicK BrowN FoX.'
```

2.1.2 Formatting strings: Adding and removing spaces

Another common need is to remove spaces (or other characters) from the beginning or end of the string. The basic method of removing characters is the `strip()` method, which strips whitespace from the beginning and end of the line:

[9]:
```
line = '         this is the content         '
line.strip()
```

[9]:
```
'this is the content'
```

To remove just space to the right or left, use `rstrip()` or `lstrip()` respectively:

[10]:
```
line.rstrip()
```

[10]:
```
'         this is the content'
```

[11]:
```
line.lstrip()
```

[11]:
```
'this is the content         '
```

To remove characters other than spaces, you can pass the desired character to the `strip()` method:

[12]:
```
num = "000000000000435"
num.strip('0')
```

[12]:
```
'435'
```

The opposite of this operation, adding spaces or other characters, can be accomplished using the `center()`, `ljust()`, and `rjust()` methods.

For example, we can use the `center()` method to center a given string within a given number of spaces:

[13]:
```
line = "this is the content"
line.center(30)
```

[13]:
```
'     this is the content      '
```

Similarly, `ljust()` and `rjust()` will left-justify or right-justify the string within spaces of a given length:

[14]:
```
line.ljust(30)
```

[14]:
```
'this is the content           '
```

[15]:
```
line.rjust(30)
```

[15]:
```
'           this is the content'
```

All these methods additionally accept any character which will be used to fill the space. For example:

[16]:
```
'435'.rjust(10, '0')
```

[16]:
```
'0000000435'
```

Because zero-filling is such a common need, Python also provides `zfill()`, which is a special method to right-pad a string with zeros:

[17]:
```
'435'.zfill(10)
```

[17]:
```
'0000000435'
```

2.1.3 Finding and replacing substrings

If you want to find occurrences of a certain character in a string, the `find()`/`rfind()`, `index()`/`rindex()`, and `replace()` methods are the best built-in methods.

`find()` and `index()` are very similar, in that they search for the first occurrence of a character or substring within a string, and return the index of the substring:

[18]:
```
line = 'the quick brown fox jumped over a lazy dog'
line.find('fox')
```

[18]:
```
16
```

[19]:
```
line.index('fox')
```

[19]:
```
16
```

The only difference between `find()` and `index()` is their behavior when the search string is not found; `find()` returns `-1`, while `index()` raises a `ValueError`:

[20]:
```
line.find('bear')
```

[20]:
```
-1
```

[21]:
```
line.index('bear')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[21], line 1
----> 1 line.index('bear')
```

```
ValueError: substring not found
```

The related `rfind()` and `rindex()` work similarly, except they search for the first occurrence from the end rather than the beginning of the string:

[22]:
```
line.rfind('a')
```

[22]:
```
35
```

For the special case of checking for a substring at the beginning or end of a string, Python provides the `startswith()` and `endswith()` methods:

[23]:
```
line.endswith('dog')
```

[23]:
```
True
```

[24]:
```
line.startswith('fox')
```

[24]:
```
False
```

To go one step further and replace a given substring with a new string, you can use the `replace()` method. Here, let's replace `'brown'` with `'red'`:

[25]:
```
line.replace('brown', 'red')
```

[25]:
```
'the quick red fox jumped over a lazy dog'
```

The `replace()` function returns a new string, and will replace all occurrences of the input:

[26]:
```
line.replace('o', '--')
```

[26]:
```
'the quick br--wn f--x jumped --ver a lazy d--g'
```

For a more flexible approach to this `replace()` functionality, see the discussion of regular expressions in Flexible Pattern Matching with Regular Expressions.

2.1.4 Splitting and partitioning strings

If you would like to find a substring *and then* split the string based on its location, the `partition()` and/or `split()` methods are what you're looking for. Both will return a sequence of substrings.

The `partition()` method returns a tuple with three elements: the substring before the first instance of the split-point, the split-point itself, and the substring after:

[27]:
```
line.partition('fox')
```

[27]:
```
('the quick brown ', 'fox', ' jumped over a lazy dog')
```

The `rpartition()` method is similar, but searches from the right of the string.

The `split()` method is perhaps more useful; it finds *all* instances of the split-point and returns the substrings in between. The default is to split on any whitespace, returning a list of the individual words in a string:

[28]:
```
line.split()
```

[28]:
```
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'a', 'lazy', 'dog']
```

A related method is `splitlines()`, which splits on newline characters. Let's do this with a Haiku, popularly attributed to the 17th-century poet Matsuo Bashō:

[29]:
```
haiku = """matsushima-ya
aah matsushima-ya
matsushima-ya"""

haiku.splitlines()
```

[29]:
```
['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']
```

Note that if you would like to undo a `split()`, you can use the `join()` method, which returns a string built from a splitpoint and an iterable:

[30]:
```
'--'.join(['1', '2', '3'])
```

[30]:
```
'1--2--3'
```

A common pattern is to use the special character `"\n"` (newline) to join together lines that have been previously split, and recover the input:

[31]:
```
print("\n".join(['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']))
```

```
matsushima-ya
aah matsushima-ya
matsushima-ya
```

2.2 Format Strings

In the preceding methods, we have learned how to extract values from strings, and to manipulate strings themselves into desired formats. Another use of string methods is to manipulate string *representations* of values of other types. Of course, string representations can always be found using the `str()` function; for example:

[32]:
```
pi = 3.14159
str(pi)
```

[32]:
```
'3.14159'
```

[33]:
```
"The value of pi is " + str(pi)
```

[33]:
```
'The value of pi is 3.14159'
```

A more flexible way to do this is to use *format strings*, which are strings with special markers (noted by curly braces) into which string-formatted values will be inserted. Here is a basic example:

[34]:
```
"The value of pi is {}".format(pi)
```

[34]:
```
'The value of pi is 3.14159'
```

Inside the `{}` marker you can also include information on exactly *what* you would like to appear there. If you include a number, it will refer to the index of the argument to insert:

[35]:
```
"""First letter: {0}. Last letter: {1}.""".format('A', 'Z')
```

[35]:
```
'First letter: A. Last letter: Z.'
```

If you include a string, it will refer to the key of any keyword argument:

[36]:
```
"""First letter: {first}. Last letter: {last}.""".format(last='Z', first='A')
```

[36]:
```
'First letter: A. Last letter: Z.'
```

Finally, for numerical inputs, you can include format codes which control how the value is converted to a string. For example, to print a number as a floating point with three digits after the decimal point, you can use the following:

[37]:
```
"pi = {0:.3f}".format(pi)
```

[37]:
```
'pi = 3.142'
```

As before, here the "0" refers to the index of the value to be inserted. The ":" marks that format codes will follow. The ".3f" encodes the desired precision: three digits beyond the decimal point, floating-point format.

3 Introduction to Regular Expressions

Regular expressions (regex) are powerful tools used for pattern matching and text manipulation in strings. They provide a concise and flexible way to search, match, and replace text based on specific patterns. Regular expressions are used across various programming languages and tools, including Python, JavaScript, Java, and many others.

3.1 What is a Regular Expression?

A regular expression is a sequence of characters that defines a search pattern. This pattern can be used to perform operations such as:

- **Searching** for specific substrings within a larger string.
- **Validating** input formats (e.g., email addresses, phone numbers).
- **Replacing** or **extracting** parts of a string based on patterns.

Regular expressions use special characters to denote different types of patterns, such as:

- `.`: Matches any single character except newline.
- `*`: Matches zero or more of the preceding element.

- `+`: Matches one or more of the preceding element.
- `?`: Matches zero or one of the preceding element.
- `[]`: Defines a character class to match any one of the characters inside the brackets.
- `^`: Anchors the pattern to the start of the string.
- `$`: Anchors the pattern to the end of the string.

3.2 Regular Expressions in Software Construction

In software construction, regular expressions are essential for various tasks that involve text processing. Here's how they relate to different aspects of software development:

3.2.1 1. Input Validation

Ensuring that user inputs meet certain criteria is crucial for software reliability. Regular expressions can validate formats for email addresses, phone numbers, and other structured data.

Example: A regex pattern for validating an email address.

3.2.2 2. Text Processing

Regular expressions simplify the task of searching and manipulating text data. They allow developers to find patterns in logs, extract relevant information, and replace content dynamically.

Example: Using regex to extract all dates from a text string.

3.2.3 3. Data Extraction

Regular expressions are useful for extracting specific pieces of information from a large dataset or text file, such as parsing configuration files or logs.

Example: Extracting version numbers or URLs from a text.

3.2.4 4. Search and Replace

When transforming text data, regular expressions enable complex search and replace operations that would be cumbersome with basic string operations.

Example: Replacing all occurrences of "apple" or "Apple" with "fruit" in a document.

3.2.5 5. Data Cleaning

Data often needs to be cleaned and standardized before analysis. Regular expressions help in removing unwanted characters, whitespace, or formatting inconsistencies.

Example: Removing all non-numeric characters from a string of digits.

3.3 Basic Regex Patterns

- By default, all major regex engines match in case-sensitive mode.
- **Anyone from list `[]`:**
  - Matches any of a set of characters inside square brackets.
  - At least one character from the list given in `[]`.

3.3.1 For Example:

- `[abc]d` – any character from `abc` and then `d`.
- `[cC]ake` – any character from `cC` and then `ake`.
- `[0123456789].00` – any digit and then `.00`.

[38]:
```
import re
```

[39]:
```
#'[abc]d' - any character from 'abc' followed by 'd'
pattern1 = '[abc]d'
test_string1 = "d ab ad bd cd dd abd"
matches1 = re.findall(pattern1, test_string1)
print("Matches for '[abc]d':", matches1)
```

```
Matches for '[abc]d': ['ad', 'bd', 'cd', 'bd']
```

[40]:
```
#'[cC]ake' - any character from 'cC' followed by 'ake'
pattern2 = '[cC]ake'
test_string2 = "ake cake Cake bake"
matches2 = re.findall(pattern2, test_string2)
print("Matches for '[cC]ake':", matches2)
```

```
Matches for '[cC]ake': ['cake', 'Cake']
```

[41]:
```
#'[0123456789].00' - any digit followed by '.00'
pattern3 = '[0123456789]\.00'
test_string3 = "5.00 6.00 10.00 3.99"
matches3 = re.findall(pattern3, test_string3)
print("Matches for '[0123456789].00':", matches3)
```

```
Matches for '[0123456789].00': ['5.00', '6.00', '0.00']
```

3.3.2 Shortened Sequence (-):

In regex, when specifying a range of characters that follow a sequential order (such as alphabets or digits), the dash (-) can be used to shorten the pattern. This helps create more compact expressions for matching a continuous sequence of characters.

Syntax: - `[a-z]` : Matches any lowercase letter from 'a' to 'z'. - `[A-Z]` : Matches any uppercase letter from 'A' to 'Z'. - `[0-9]` : Matches any digit from '0' to '9'.

[42]:
```
#Match any lowercase letter from 'a' to 'z'
pattern1 = '[a-z]'
test_string1 = "hello World 123"
matches1 = re.findall(pattern1, test_string1)
print("Matches for '[a-z]':", matches1)
```

```
Matches for '[a-z]': ['h', 'e', 'l', 'l', 'o', 'o', 'r', 'l', 'd']
```

[43]:
```
#Match any uppercase letter from 'A' to 'Z'
pattern2 = '[A-Z]'
```

```
test_string2 = "hello World 123"
matches2 = re.findall(pattern2, test_string2)
print("Matches for '[A-Z]':", matches2)
```

```
Matches for '[A-Z]': ['W']
```

[44]:
```
#Match any digit from '0' to '9'
pattern3 = '[0-9]'
test_string3 = "hello World 123"
matches3 = re.findall(pattern3, test_string3)
print("Matches for '[0-9]':", matches3)
```

```
Matches for '[0-9]': ['1', '2', '3']
```

3.4 Caret (^)

The caret (^) symbol in regex is used to negate a set of characters, meaning it will match any character *other than* the ones specified inside the square brackets. However, if the caret appears in the middle or end of the set, it is treated as a literal caret rather than a negation symbol.

3.4.1 Syntax:

- `[^apt]`: Matches any character *except* `'a'`, `'p'`, or `'t'`.
- `[^a-z]`: Matches any character *except* lowercase letters (i.e., anything that is not between `'a'` and `'z'`).
- `[e^]`: Matches either the character `'e'` or the literal caret (^).
- `a^b`: Matches the sequence `'a^b'`.

3.4.2 Important Notes:

- The caret (^) *must* be the first character inside the square brackets for negation. If placed elsewhere, it is treated as a literal character.

[45]:
```
#Match any character other than 'a', 'p', or 't'
pattern1 = '[^apt]'
test_string1 = "apple bat top"
matches1 = re.findall(pattern1, test_string1)
print("Matches for '[^apt]':", matches1)
```

```
Matches for '[^apt]': ['l', 'e', ' ', 'b', ' ', 'o']
```

[46]:
```
#Match any character other than lowercase letters
pattern2 = '[^a-z]'
test_string2 = "Hello 123 World!"
matches2 = re.findall(pattern2, test_string2)
print("Matches for '[^a-z]':", matches2)
```

```
Matches for '[^a-z]': ['H', ' ', '1', '2', '3', ' ', 'W', '!']
```

[47]:
```
#Match either 'e' or '^'
pattern3 = '[e^]'
test_string3 = "e ^ f g"
matches3 = re.findall(pattern3, test_string3)
print("Matches for '[e^]':", matches3)
```

```
Matches for '[e^]': ['e', '^']
```

[48]:
```
#Match the sequence 'a^b'
pattern4 = '[a^b]'
test_string4 = "a^b abc aaa"
matches4 = re.findall(pattern4, test_string4)
print("Matches for '[a^b]':", matches4)
```

```
Matches for '[a^b]': ['a', '^', 'b', 'a', 'b', 'a', 'a', 'a']
```

3.5 Optional Characters (?)

In regex, the question mark (?) is used to indicate that the preceding character or group is optional, meaning it can appear either once or not at all.

3.5.1 Syntax:

- `colou?r`: Matches `'color'` or `'colour'` (the character `'u'` is optional).
- `Books?`: Matches `'Book'` or `'Books'` (the character `'s'` is optional).

[49]:
```
#Match 'color' or 'colour'
pattern1 = 'colou?r'
test_string1 = "color, colour, colouur"
matches1 = re.findall(pattern1, test_string1)
print("Matches for 'colou?r':", matches1)
```

```
Matches for 'colou?r': ['color', 'colour']
```

[50]:
```
#Match 'Book' or 'Books'
pattern2 = 'Books?'
test_string2 = "Book, Books, Bookshelf, bookshelf"
matches2 = re.findall(pattern2, test_string2)
print("Matches for 'Books?':", matches2)
```

```
Matches for 'Books?': ['Book', 'Books', 'Books']
```

3.6 Kleene Star (*)

The Kleene star (*) in regex allows for zero or more occurrences of the preceding character or group. This is useful when you need to match a pattern where the previous character can appear any number of times, including not at all.

3.6.1 Syntax:

- `baa*`: Matches `'b'` followed by one `'a'` and then zero or more `'a'` characters (e.g., `'ba'`, `'baa'`, `'baaa'`, etc.).

3.6.2 Explanation:

- The Kleene star means the character preceding the `*` can occur zero, one, or many times. In this case, it applies to the letter `'a'`.
- **Example**: In the pattern `baa*`, the string can contain:
  - `'ba'` (zero `'a'` after the first one),
  - `'baa'` (one `'a'` after the first one),
  - `'baaa'` (two or more `'a'` characters), and so on.

[51]:
```
# Example: Match 'b' followed by one or more 'a's
pattern = 'baa*'
test_string = "aaaa b ba bab baa baaa baaaa baaaab"
matches = re.findall(pattern, test_string)
print("Matches for 'baa*':", matches)
```

```
Matches for 'baa*': ['ba', 'ba', 'baa', 'baaa', 'baaaa', 'baaaa']
```

3.7 Repeating Patterns (`[ab]*`)

The expression `[ab]*` in regex allows for zero or more occurrences of any combination of characters specified inside the square brackets. This is useful when you need to match sequences consisting of a combination of characters repeatedly.

3.7.1 Syntax:

- `[ab]*`: Matches any sequence of the characters `'a'` and `'b'`, including sequences with no characters at all. Examples include `'ababab'`, `'abbbbb'`, `'bbbbbb'`, `'aaaaa'`, and the empty string.

3.7.2 Example:

- **Pattern**: `[ab]*`
  - This pattern matches sequences that consist of any combination of `'a'` and `'b'`, and can also match an empty string.
- **Example**: It can match:
  - `'ababab'`
  - `'abbbbb'`
  - `'bbbbbb'`
  - `'aaaaa'`
  - `''` (empty string)

3.7.3 Another Example:

- **Pattern**: `[1-9][0-9]*`

– This pattern matches a number where the first digit is between 1 and 9, followed by zero or more digits (0-9). It represents prices or numbers that can be one or more digits long.

[52]:
```
#Match seqindalluences of 'a' and 'b'
pattern1 = '[ab]*'
test_string1 = "ababab abbbbb bbbbbb aaaaa ab"
matches1 = re.findall(pattern1, test_string1)
print("Matches for '[ab]*':", matches1)
```

```
Matches for '[ab]*': ['ababab', '', 'abbbbb', '', 'bbbbbb', '', 'aaaaa', '',
'ab', '']
```

[53]:
```
#Match numbers starting with 1-9 followed by zero or more digits
pattern2 = r'[0-9][1-9]*'
test_string2 = "a 1 23 456 789 0 12345 10"
matches2 = re.findall(pattern2, test_string2)
print("Matches for '[1-9][0-9]*':", matches2)
```

```
Matches for '[1-9][0-9]*': ['1', '23', '456', '789', '0', '12345', '1', '0']
```

3.8 Kleene Plus (+)

The Kleene plus (+) in regex allows for matching one or more occurrences of the preceding character or group. This is similar to the Kleene star (*), but with the requirement that at least one occurrence must be present.

3.8.1 Syntax:

- `[1-9]+`: Matches one or more digits between 1 and 9. This can be used to represent prices or any number that is composed of one or more digits, excluding zero.

3.8.2 Explanation:

- The Kleene plus ensures that the preceding element occurs at least once, but it can appear multiple times.

[54]:
```
#Match one or more digits between 1 and 9
pattern = r'[1-9][1-9]+'
test_string = "1 23 456 789 10 12345"
matches = re.findall(pattern, test_string)
print("Matches for '[1-9]+':", matches)
```

```
Matches for '[1-9]+': ['23', '456', '789', '12345']
```

3.9 Wildcard (.)

In regex, the dot (.) is a wildcard character that matches any single character except for newline characters. This is useful for creating patterns where you need to match any character in a specific position.

3.9.1 Syntax:

- `beg.n`: Matches any string that starts with `'beg'`, followed by any single character, and ends with `'n'`. For example, it can match `'began'`, `'begin'`, or `'begxn'`.

[55]:
```
#Match 'beg' followed by any single character and ending with 'n'
pattern = r'beg.n'
test_string = "began begin begwn begn beg1n begiin begiinn"
matches = re.findall(pattern, test_string)
print("Matches for 'beg.n':", matches)
```

```
Matches for 'beg.n': ['began', 'begin', 'begwn', 'beg1n']
```

3.10 Anchors

Anchors in regex are used to specify the position in a string where the pattern should be matched. They help in anchoring the regular expressions to particular places in the string.

3.10.1 Caret (^)

The caret (^) anchor is used to indicate the start of a line or string.

Syntax: - `^The`: Matches any line or string that starts with `'The'`.

[56]:
```
#Match lines that start with 'The'
pattern = '^The'
test_string = """The quick brown fox.
                 Jumps over the lazy dog.
                 The end"""
matches = re.findall(pattern, test_string, re.MULTILINE)
print("Matches for '^The':", matches)
```

```
Matches for '^The': ['The']
```

3.11 Dollar ($)

The dollar ($) anchor is used to indicate the end of a line or string.

Syntax: - The dog.$: Matches any line or string that ends with 'The dog.'

[57]:
```
pattern = 'The dog\.$'
test_string = """The quick brown fox
                 Jumps over the lazy The dog.
                 The dog."""
matches = re.findall(pattern, test_string, re.MULTILINE)
print("Matches for 'The dog\.$':", matches)
```

```
Matches for 'The dog\.$': ['The dog.', 'The dog.']
```

3.12 The word boundary ()

The word boundary () anchor is used to match positions where a word character is not followed or preceded by another word character. This ensures the pattern matches whole words only.

Syntax: : Matches the word 'the' as a whole word, not as part of another word like 'other' or 'their'.

[58]:
```
#Match the whole word 'the'
pattern = '[tT]he'
test_string = "The their quick brown fox jumps over the there lazy dog."
matches = re.findall(pattern, test_string)
print("Matches for 'the':", matches)
```

```
Matches for 'the': ['The', 'the', 'the', 'the']
```

[59]:
```
#Match the whole word 'the'
pattern = r'\bThe\b'
test_string = "The their quick brown fox jumps over the there lazy dog."
matches = re.findall(pattern, test_string)
print("Matches for '\\bthe\\b':", matches)
```

```
Matches for '\bthe\b': ['The']
```

3.13 Disjunction (|)

In regex, the vertical bar (|) is used to represent a logical OR operation between patterns. This allows you to match one pattern or another within the same regex expression.

3.13.1 Syntax:

- `cat|dog`: Matches either `'cat'` or `'dog'`.

3.13.2 Explanation:

- The `|` operator acts as a disjunction, meaning it will match either of the patterns separated by the `|`. In this case, it will match strings that contain either `'cat'` or `'dog'`.

[60]:
```
# Match either 'cat' or 'dog'
pattern = r'cat|dog'
test_string = "I have a cat and a dog."
matches = re.findall(pattern, test_string)
print("Matches for 'cat|dog':", matches)
```

```
Matches for 'cat|dog': ['cat', 'dog']
```

3.14 Grouping and Precedence

In regex, parentheses `()` are used to group patterns and control the order of operations. This is useful when you want to apply disjunction (|) to a specific part of the pattern or to combine multiple characters into a single unit.

3.14.1 Syntax:

- `gupp(y|ies)`: Matches either `'guppy'` or `'guppies'`.

[61]:
```
#Match 'guppy' or 'guppies'
pattern = r'happ(y|ies)'
test_string = "I am happy and happies."
matches = re.findall(pattern, test_string)
print("Matches for 'happ(y|ies)':", matches)
```

```
Matches for 'happ(y|ies)': ['y', 'ies']
```

3.15 Referring to an Earlier Occurrence of the Instance

In regex, you can refer to a previously captured group within the same pattern using backreferences. This allows you to ensure that the same text appears in different parts of the string.

3.15.1 Syntax:

- `The (.*)er they were, the \1er they will be`
  - `\1` refers to the content captured in the first group `(.*)`.

3.15.2 Explanation:

- The parentheses `()` create a capturing group. The `.*` inside the parentheses matches any sequence of characters.
- `\1` is a backreference to the content captured by the first group. This means the same text matched by `(.*)` must appear again in the part of the pattern where `\1` is used.

[62]:
```
# Match a pattern with a backreference to an earlier captured group
pattern = r'The (.*)er they were, the \1er they will be'
test_string = "The bigger they were, the bigger they will be."
match = re.search(pattern, test_string)

# Check if a match was found and display the result
if match:
    print("Match found!")
    print("Captured group:", match.group(1))  # Print the content of the first␣
 ↪capturing group
else:
    print("No match found.")
```

```
Match found!
Captured group: bigg
```

3.16 Regex with Multiple Capturing Groups and Backreferences

Regular expressions (regex) allow for powerful text matching and manipulation. When dealing with patterns that involve repeated segments or need to refer back to previously matched text, capturing groups and backreferences become essential tools.

3.16.1 Capturing Groups

Capturing groups are portions of a regex pattern enclosed in parentheses `()`. They are used to extract specific parts of the matched text. For example, in the pattern `(\d+)`, the `(\d+)` part captures one or more digits as a group.

3.16.2 Backreferences

Backreferences refer to a previously captured group within the same regex pattern. They allow you to ensure that a specific part of the pattern matches the same text as another part. For example, `(\d+)\1` matches a digit sequence followed by the same sequence again.

[63]:
```
# Define the pattern with two capturing groups and backreferences
pattern = r'The (.*)er they (.*), the \1er they \2 be'

# Define a test string that should match the pattern
test_string = "The bigger they were, the bigger they were be"

# Perform the regex search
match = re.search(pattern, test_string)

# Check if a match was found and display the result
if match:
    print("Match found!")
    print("Captured group 1:", match.group(1))  # Print the content of the first␣
 ↪capturing group
    print("Captured group 2:", match.group(2))  # Print the content of the␣
 ↪second capturing group
else:
    print("No match found.")
```

```
Match found!
Captured group 1: bigg
Captured group 2: were
```

3.17 Regex Shorthand Characters

Regex shorthand characters are used to match specific types of characters in a string. They simplify patterns by representing common character classes.

3.17.1 `\d` - Digit

- **Description**: Matches any digit character. Equivalent to `[0-9]`.
- **Example**: `\d` will match any single digit in a string.

[64]:
```
# Define the pattern to match digits
pattern = '\d'
test_string = "The year is 2024."
matches = re.findall(pattern, test_string)
```

```
print("Digits found:", matches)
```

```
Digits found: ['2', '0', '2', '4']
```

3.17.2 \D - Non-Digit

- **Description**: Matches any character that is not a digit. It is the opposite of \d, which matches digits. This shorthand is equivalent to [^0-9].
- **Example**: \D will match any single character that is not a digit.

[65]:
```
# Define the pattern to match non-digits
pattern = '\D'
test_string = "Year 2024"
matches = re.findall(pattern, test_string)
print("Non-digits found:", matches)
```

```
Non-digits found: ['Y', 'e', 'a', 'r', ' ']
```

3.17.3 \w - Word Character

- **Description**: Matches any alphanumeric character (letters and digits) and underscores. It is equivalent to [a-zA-Z0-9_]. This shorthand is used to find characters that are part of a word or identifier.
- **Example**: \w will match any single letter, digit, or underscore.

[66]:
```
# Define the pattern to match word characters
pattern = r'\w'
test_string = "Hello_2024!"
matches = re.findall(pattern, test_string)
print("Word characters found:", matches)
```

```
Word characters found: ['H', 'e', 'l', 'l', 'o', '_', '2', '0', '2', '4']
```

3.17.4 \W - Non-Word Character

- **Description**: Matches any character that is not a word character. It is the opposite of \w. This shorthand is equivalent to [^a-zA-Z0-9_]. \W will match any character that is not a letter, digit, or underscore.
- **Example**: \W will match any single character that is not part of a word.

[67]:
```
# Define the pattern to match non-word characters
pattern = r'\W'
test_string = "Hello_2024! "
matches = re.findall(pattern, test_string)
print("Non-word characters found:", matches)
```

```
Non-word characters found: ['!', ' ']
```

3.17.5 \s - Whitespace Character

- **Description**: Matches any whitespace character, which includes spaces, tabs, newlines, and other whitespace characters. It is equivalent to `[ \t\n\r\f\v]`.
- **Example**: `\s` will match any single whitespace character.

[68]:
```
# Define the pattern to match whitespace characters
pattern = r'\s'
test_string = """Hello 2024!
"""
matches = re.findall(pattern, test_string)
print("Whitespace characters found:", matches)
```

```
Whitespace characters found: [' ', '\n']
```

3.17.6 \S - Non-Whitespace Character

- **Description**: Matches any character that is not a whitespace character. It is the opposite of `\s` and is equivalent to `[^ \t\n\r\f\v]`. This shorthand is useful for finding characters that are part of a meaningful sequence or content, excluding spaces and other whitespace characters.
- **Example**: `\S` will match any single character that is not a space, tab, newline, or other whitespace character.

[69]:
```
# Define the pattern to match non-whitespace characters
pattern = r'\S'
test_string = "Hello 2024!"
matches = re.findall(pattern, test_string)
print("Non-whitespace characters found:", matches)
```

```
Non-whitespace characters found: ['H', 'e', 'l', 'l', 'o', '2', '0', '2', '4',
'!']
```

3.17.7 Practice Questions

[70]:
```
test_string = (
    """It certainly was the thing I was looking for. It has aesthetic appearance
    and is very subtle at The controlling different channels. The other thing I
    appreciate about it is its themes the app and the255."""
)

# Pattern 1: 'the'
pattern1 = 'the'
matches1 = re.findall(pattern1, test_string)
print("Pattern 1: 'the'")
print("Regex:", pattern1)
print("Matches found:", matches1)
```

```
Pattern 1: 'the'
Regex: the
```

```
Matches found: ['the', 'the', 'the', 'the', 'the', 'the']
```

[71]:
```
# Pattern 2: '[tT]he'
pattern2 = '[tT]he'
matches2 = re.findall(pattern2, test_string)
print("Pattern 2: '[tT]he'")
print("Regex:", pattern2)
print("Matches found:", matches2)
```

```
Pattern 2: '[tT]he'
Regex: [tT]he
Matches found: ['the', 'the', 'The', 'The', 'the', 'the', 'the', 'the']
```

[72]:
```
# Pattern 3: '\b[tT]he\b'
pattern3 = r'\b[tT]he\b'
matches3 = re.findall(pattern3, test_string)
print("Pattern 3: '\\b[tT]he\\b'")
print("Regex:", pattern3)
print("Matches found:", matches3)
```

```
Pattern 3: '\b[tT]he\b'
Regex: \b[tT]he\b
Matches found: ['the', 'The', 'The', 'the']
```

[73]:
```
# Pattern 4: '[^a-zA-Z][tT]he[^a-zA-Z]'
pattern4 = r'[^a-zA-Z][tT]he[^a-zA-Z]'
matches4 = re.findall(pattern4, test_string)
print("Pattern 4: '[^a-zA-Z][tT]he[^a-zA-Z]'")
print("Regex:", pattern4)
print("Matches found:", matches4)
```

```
Pattern 4: '[^a-zA-Z][tT]he[^a-zA-Z]'
Regex: [^a-zA-Z][tT]he[^a-zA-Z]
Matches found: [' the ', ' The ', ' The ', ' the ', ' the2']
```

3.17.8 Curly Brace Quantifiers in Regular Expressions

Curly braces `{}` in regular expressions are used to define the number of occurrences of the preceding character or group. They give precise control over how many times a pattern should be repeated.

- `{n}`: Matches exactly `n` occurrences.
  - The pattern will match the preceding character or group exactly `n` times.
- `{n,m}`: Matches between `n` and `m` occurrences.
  - The pattern will match the preceding character or group at least `n` times and at most `m` times.
- `{n,}`: Matches at least `n` occurrences.
  - The pattern will match the preceding character or group at least `n` times, with no upper limit.
- `{,m}`: Matches up to `m` occurrences (including 0).
  - The pattern will match the preceding character or group from 0 up to `m` times.

These quantifiers give you flexibility in controlling how many times you want a pattern to match, making regular expressions more powerful for complex validations.

3.17.9 {n}: Exactly n occurrences

Matches the preceding character or group exactly n times.

[74]:
```
pattern = 'a{3}'
test_string = "aaa aa aaaa"
matches = re.findall(pattern, test_string)
print("Matches for 'a{3}':", matches)
```

```
Matches for 'a{3}': ['aaa', 'aaa']
```

3.17.10 {n,m}: Between n and m occurrences

Matches the preceding character or group between n and m times, inclusive.

[75]:
```
pattern = 'a{2,4}'
test_string = "a aa aaa aaaa aaaaa"
matches = re.findall(pattern, test_string)
print("Matches for 'a{2,4}':", matches)
```

```
Matches for 'a{2,4}': ['aa', 'aaa', 'aaaa', 'aaaa']
```

3.17.11 {n,}: At least n occurrences

Matches the preceding character or group at least n times, with no upper limit.

[76]:
```
pattern = 'a{2,}'
test_string = "a aa aaa aaaa aaaaa"
matches = re.findall(pattern, test_string)
print("Matches for 'a{2,}':", matches)
```

```
Matches for 'a{2,}': ['aa', 'aaa', 'aaaa', 'aaaaa']
```

3.17.12 {,m}: Up to m occurrences

Matches the preceding character or group up to m times. The pattern allows 0 to m occurrences.

[77]:
```
pattern = 'a{,3}'
test_string = "a aa aaa aaaa aaaaa"
matches = re.findall(pattern, test_string)
print("Matches for 'a{,3}':", matches)
```

```
Matches for 'a{,3}': ['a', '', 'aa', '', 'aaa', '', 'aaa', 'a', '', 'aaa', 'aa',
'']
```

[ ]:
```

```