CH-8: Strings: A Deeper Look

Introduction

Introduction to Strings

- Strings support sequence operations like lists and tuples.
- Strings are immutable, similar to tuples.
- Python provides basic string formatting, operators, and methods.

Regular Expressions

- Regular expressions are tools for matching patterns in text.
- Crucial for text processing in data-rich applications.
- Python's `re` module provides powerful regular expression functionalities.

Advanced Topics and Applications

Relevance in Advanced Topics

- Mastering string processing and regular expressions is essential for:
  - **Natural Language Processing (NLP):** Manipulating and understanding text.
  - **Data Cleaning and Wrangling:** Preparing data using Pandas Series and DataFrames.

Applications

- Common applications include:
  - Text analysis and sentiment analysis.
  - Data cleaning and preparation for analytics.

Formatting Strings

- Text formatting is crucial for making data readable and understandable.
- Python offers powerful text formatting options, particularly through f-strings.

  Presentation Types

  Basic f-Strings

  - F-strings display values as strings unless another type is specified.
  - Example: Formatting a float rounded to two decimal places:

```
In [1]: f'{12.3457:.2f}'
Out[1]: '12.35'
```

Formatting Strings

Floating-Point and Decimal Values

- `f` **Presentation Type**: Formats numbers as standard decimal values.
- `e` **Presentation Type**: Formats numbers in exponential (scientific) notation.
- For a capital `E` in the exponent, use the E presentation type.

```
In [1]: from decimal import Decimal

In [2]: f'{Decimal("1000000.56007"):.3f}'
Out[2]: '1000000.560'

In [3]: f'{Decimal("1000000.56007"):.3e}'
Out[3]: '1.000e+6'
In [4]: f'{Decimal("1000000.56007"):.3E}'
Out[4]: '1.000E+6'
```

Formatting Strings

Precision Rules

- Precision is supported only for **floating-point** and **Decimal values**.
- Attempting to use f on incompatible types (e.g., strings) raises a `ValueError`.
- For a capital `E` in the exponent, use the E presentation type.

```
In [5]: f'{"hello":0.3f}'
---------------------------------------------------------
ValueError          Traceback (most recent call last)
```

Formatting Strings

Common Presentation Types

Type	Purpose	Example	Output
`f`	Floating-point numbers	`f'{17.489:.2f}'`	`'17.49'`
`e`	Exponential notation	`f'{1.23e5:.2e}'`	`'1.23e+05'`
`E`	Exponential notation (caps)	`f'{1.23e5:.2E}'`	`'1.23E+05'`
`d`	Decimal integers	`f'{10:d}'`	`'10'`
`b`	Binary integers	`f'{10:b}'`	`'1010'`
`o`	Octal integers	`f'{10:o}'`	`'12'`
`x/X`	Hexadecimal integers	`f'{255:x}'`	`'ff'` / `'FF'`
`c`	Character codes	`f'{65:c}'`	`'A'`
`s`	Strings (default type)	`f'"hello":s'`	`'hello'`

Formatting Strings

Special Cases

1. **Characters:**
   - Format integers as characters using c.

   ```
   f'{65:c} {97:c}'  # Output: 'A a'
   ```

2. **Strings**
   - Default presentation type is s.
   - Non-string values are automatically converted to strings.

   ```
   In [6]: f'{"hello":s} {10}'
   Out[6]: 'hello 10'
   ```

   - Double quotes can be used to include single quotes in the string.

Formatting Strings

Field Widths and Alignment

Field Widths

- Field widths define the total number of characters allocated for a value in the formatted output.
- If the value's width is less than the field width, Python fills the remaining spaces with blanks

```
f'[{27:10d}]'  # Number, right-aligned by default
# Output: '[        27]'

f'[{3.5:10f}]' # Float, right-aligned by default
# Output: '[  3.500000]'

f'[{"hello":10}]'   # String, left-aligned by default
# Output: '[hello     ]'
```

Alignment Options

- $<$ for left aligned
- $>$ for right aligned
- ˆfor centered

Formatting Strings

Numeric Formatting in Python

Forcing Positive Signs (+):

```
f'{27:+10d}'  # Output: '       +27'
f'{27:+010d}' # Output: '+000000027'
```

Space for Positive Numbers:

```
print(f'{27:d}\n{27: d}\n{-27: d}')
# Output:
# 27
#  27
# -27
```

Thousands Separator (,):

```
f'{12345678:,d}'     # Output: '12,345,678'
f'{123456.78:,.2f}' # Output: '123,456.78'
```

Formatting Strings

- String Format Method
- **Overview:**
  - Used for string formatting before Python 3.6.
  - F-strings are based on its capabilities but are preferred now.
- **Basic Formatting:**

```
'{:.2f}'.format(17.489)  # Output: '17.49'
```

- **Multiple Placeholders:**

```
'{} {}'.format('Amanda', 'Cyan')  # Output: 'Amanda
    Cyan'
```

- **Referencing by Position:**

```
'{0} {0} {1}'.format('Happy', 'Birthday')
# Output: 'Happy Happy Birthday'
```

Formatting Strings

- **Referencing Keyword Arguments:**

```
'{first} {last}'.format(first='Amanda', last='Gray')
# Output: 'Amanda Gray'
'{last} {first}'.format(first='Amanda', last='Gray')
# Output: 'Gray Amanda'
```

- **Key Notes:**
  - Use placeholders ‘‘ in the format string.
  - Arguments can be passed by position or keyword.
  - Format specifiers (e.g., ‘:.2f‘) can control output.

Concatenating and Repeating Strings

Concatenation with +=:

- The += operator appends a string to an existing string.
- Each operation creates a new string because strings are immutable.

Repetition with *=:

- The *= operator repeats a string a specified number of times.
- Useful for creating patterns or repeated characters like a bar of asterisks.

Example:

```python
# Concatenate first and last name
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name)
# Repeat asterisks
bar = "*" * len(full_name)
print(bar)
print(full_name)
print(bar)
```

Stripping Whitespace from Strings

String Methods:

- `strip`: Removes both leading and trailing whitespace.
- `lstrip`: Removes only leading whitespace.
- `rstrip`: Removes only trailing whitespace.
- These methods return a new string, leaving the original unchanged.

Example:

```
# Using strip, lstrip, and rstrip
sentence = '\t  \n  This is a test string. \t\t \n'
print(sentence.strip())   # 'This is a test string.'
print(sentence.lstrip())  # 'This is a test string. \t\t \n'
print(sentence.rstrip())  # '\t  \n  This is a test string.'
```

Note: These methods remove all types of whitespace, including spaces, newlines, and tabs.

Changing Character Case

Methods to Change Case:

- `lower`: Converts the string to all lowercase letters.
- `upper`: Converts the string to all uppercase letters.
- `capitalize`: Capitalizes only the first letter of the string.
- `title`: Capitalizes the first letter of every word in the string.

Capitalizing Only the First Character:

```
# Capitalize the first character of the string
'happy birthday'.capitalize()  # 'Happy birthday'
```

Capitalizing the First Character of Every Word:

```
# Capitalize the first character of each word
'strings: a deeper look'.title()  # 'Strings: A Deeper
    Look'
```

Comparison Operators for Strings

Comparison of Strings:

- Strings are compared based on their underlying integer values.
- Uppercase letters compare as less than lowercase letters because they have lower integer values.
- For example, `'A'` is 65 and `'a'` is 97.
- The `ord` function can be used to check the character code of a string.

Example:

```
# Compare the strings 'Orange' and 'orange'
print(f"A: {ord('A')}; a: {ord('a')}")
print('Orange' == 'orange')  # False
print('Orange' != 'orange')  # True
print('Orange' < 'orange')   # True
print('Orange' <= 'orange')  # True
print('Orange' > 'orange')   # False
print('Orange' >= 'orange')  # False
```

Searching for Substrings: Counting Occurrences

Counting Occurrences:

- The `count` method returns the number of occurrences of a substring in a string.
- You can specify `start_index` and `end_index` to search within a substring.
- `count` searches the slice `string[start_index:end_index]`.

Example:

```
sentence = 'this is a test sentence to test the string methods'
print(sentence.count('to', 12))  # 1
print(sentence.count('that', 12, 25))  # 1
```

Locating a Substring: Index and Rindex

Locating a Substring:

- `index`: Returns the first index where a substring is found.
- `rindex`: Returns the last index of a substring, searched from the end.
- `find` and `rfind`: Like `index` and `rindex`, but return −1 if not found (instead of raising an error).

Example:

```
print(sentence.index('be'))  # 3
print(sentence.rindex('be'))  # 16
```

Checking Substring Presence

Checking Substring Presence:

- Use the `in` and `not in` operators to check if a substring exists in a string.

Example:

```
print('that' in sentence)  # True
print('THAT' in sentence)  # False
print('THAT' not in sentence)  # True
```

Locating Substrings at the Beginning or End

Locating Substrings at the Beginning or End:

- `startswith`: Returns True if the string starts with a given substring.
- `endswith`: Returns True if the string ends with a given substring.

Example:

```
print(sentence.startswith('to'))  # True
print(sentence.startswith('be'))  # False
print(sentence.endswith('question'))  # True
print(sentence.endswith('quest'))  # False
```

Replacing Substrings

Replacing Substrings:

- The `replace` method searches for a substring in a string and replaces it with another substring.
- The method returns a new string containing the replaced values, leaving the original string unchanged.
- You can optionally specify the maximum number of replacements using a third argument.

Example:

```
values = '1\t2\t3\t4\t5'
print(values.replace('\t', ','))  # '1,2,3,4,5'
```

Optional Argument for Maximum Replacements:

- The third argument to `replace` specifies the maximum number of replacements.

Example:

```
print(values.replace('\t', ',', 2))  # '1,2,3\t4\t5'
```

Tokenization and Delimiters

Concept:

- Strings are tokenized by breaking them into smaller components (tokens).
- Delimiters separate tokens; common delimiters include spaces, commas, and newline characters.

String Method `split`:

- Tokenizes strings based on whitespace or a custom delimiter.
- Example:

```
letters = 'A, B, C, D'
letters.split(', ')  # ['A', 'B', 'C', 'D']
letters.split(', ', 2)  # ['A', 'B', 'C, D']
```

- Second argument specifies maximum number of splits.

Reverse Splitting

String Method `rsplit`:

- Similar to `split`, but performs splits starting from the end of the string.
- Useful when you want to prioritize splitting from the right.
- Example:

```
letters = 'A, B, C, D'
letters.rsplit(', ', 2)  # ['A, B', 'C', 'D']
```

Joining Strings

Concept:

- Method `join` concatenates strings in an iterable, using the string it is called on as the separator.

Examples:

- Joining a list of strings:

```
letters_list = ['A', 'B', 'C', 'D']
','.join(letters_list)  # 'A,B,C,D'
```

- Joining with a list comprehension:

```
','.join([str(i) for i in range(10)])  # '0,1,2,3,4,5,6,7,8,9'
```

Note:

- The iterable must contain only strings; otherwise, a `TypeError` occurs.

Partitioning Strings

String Method `partition`:

- Splits a string into three parts: before the separator, the separator, and after the separator.
- Example:

```
'Amanda: 89, 97, 92'.partition(': ')
# ('Amanda', ': ', '89, 97, 92')
```

String Method `rpartition`:

- Similar to `partition`, but searches for the separator from the end of the string.
- Example:

```
url = 'http://www.deitel.com/books/PyCDS/table_of_contents.html'
rest_of_url, separator, document = url.rpartition('/')
# rest_of_url: 'http://www.deitel.com/books/PyCDS'
# document: 'table_of_contents.html'
```

Splitting Strings into Lines

String Method `splitlines`:

- Splits a string into a list of lines at newline characters (`\n`).
- **Default behavior:**
  - Removes newline characters from the resulting lines.
- **Optional argument:**
  - Passing `True` keeps newline characters in the output.
- Example:

```
lines = """This is line 1
This is line2
This is line3"""

lines.splitlines()
# Output: ['This is line 1', 'This is line2', 'This is line3']

lines.splitlines(True)
# Output: ['This is line 1\n', 'This is line2\n', 'This is line3']
```

Characters and Character-Testing Methods

Concept:

- Characters (digits, letters, symbols) are the fundamental building blocks of programs.
- In Python, a character is a one-character string.
- Python provides methods to test strings for specific characteristics.

Examples of Character-Testing Methods:

- `isdigit`: Checks if the string contains only digits.

```
'-27'.isdigit()  # False
'27'.isdigit()   # True
```

- `isalnum`: Checks if the string contains only letters and digits.

```
'A9876'.isalnum()  # True
'123 Main Street'.isalnum()  # False
```

Character-Testing Methods Overview

Other Methods Overview:

- `isalpha`: Only alphabetic characters.
- `isdecimal`: Only decimal integer characters.
- `islower`: All alphabetic characters are lowercase.
- `isupper`: All alphabetic characters are uppercase.
- `isspace`: Only whitespace characters.
- `istitle`: Each word starts with an uppercase letter.
- `isidentifier`: Valid identifier for Python variables.
- `isnumeric`: Represents a numeric value without signs or decimal points.

Note:

- Each method returns `True` only if the condition described is satisfied; otherwise, it returns `False`.
- Character-testing methods are useful for validating user input and processing data.

Raw Strings in Python

Introduction to Raw Strings:

- Backslash (\\) characters in strings introduce escape sequences, such as:
  - `\n` for a newline.
  - `\t` for a tab.
- To include a literal backslash in a string, you must escape it with another backslash (\\\\), which can make strings harder to read.
- For example, Windows file paths use backslashes, requiring extra effort to write correctly:

```
file_path = 'C:\\MyFolder\\MySubFolder\\MyFile.txt'
print(file_path)
# Output: C:\MyFolder\MySubFolder\MyFile.txt
```

Benefits of Raw Strings

What Are Raw Strings?

- Prefixing a string with `r` makes it a raw string, treating backslashes as regular characters.
- Raw strings are particularly helpful for:
  - File paths.
  - Regular expressions, which often include multiple backslashes.

Example with Raw Strings:

```
# Using a raw string for a file path
file_path = r'C:\MyFolder\MySubFolder\MyFile.txt'
print(file_path)
# Output: C:\MyFolder\MySubFolder\MyFile.txt
```

Key Insight:

- Internally, Python still represents the backslashes as \\, even in raw strings.

Introduction to Regular Expressions

- **Pattern Recognition:** Regular expressions help identify patterns in text (e.g., phone numbers, email addresses, ZIP codes, and Social Security numbers).
- **Data Extraction:** They can be used to extract information from unstructured text like social media posts.
- **Data Validation:** Regular expressions validate data formats, such as:
  * ZIP codes (5 digits or 5 digits + hyphen + 4 digits)
  * Last names (letters, spaces, apostrophes, hyphens only)
  * Email addresses (valid character order)
  * Social Security numbers (specific format with rules for digits)
- **Reuse of Existing Patterns:** Common regular expressions for these patterns can be found on websites like:
  * regex101.com
  * regexlib.com
  * regular-expressions.info

Other usage of Regular Expressions

- **Data Extraction (Scraping):** Used to extract specific data from text, such as locating all URLs on a web page. Tools like BeautifulSoup, XPath, and lxml may be preferred for complex scraping tasks.
- Data Cleaning: Helps clean data by:
  - Removing unnecessary or duplicate data
  - Handling incomplete or incorrect data
  - Fixing typos
  - Ensuring consistent data formats
  - Managing outliers and other anomalies
- **Data Transformation:** Used to transform data into different formats, such as converting tab-separated or space-separated values into CSV format for applications that require CSV input.

re Module and Function fullmatch

- To use regular expressions, import the Python Standard Library's `re` module:

```
In [1]: import re
```

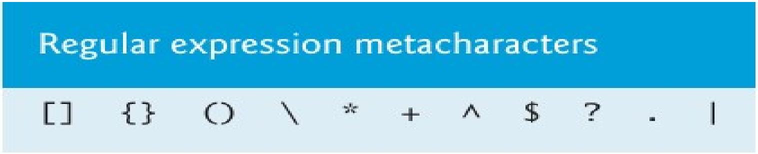
- Using `fullmatch`: The `fullmatch` function checks if the entire string matches the regular expression pattern. It returns a match object if the entire string matches, or None if there's no match.

```
In [2]: pattern = 'aja123'
In [3]: print('match' if re.fullmatch(pattern,'aja12345') else 'no match')
no match

In [4]: print('match' if re.fullmatch(pattern,'aja123') else 'no match')
match
```

* The key point is that the string must entirely match the pattern for fullmatch to return a match object.

Metacharacters, Character Classes and Quantifiers

- Regular expressions typically contain various special symbols called metacharacters, which are shown in the table below:



Regular expression metacharacters										
[]	{}	()	\	*	+	^	$	?	.	|

- Regular Expression Metacharacters:
  - [ ]: Matches any one of the characters inside the brackets.
  - {}: Specifies the number of repetitions (quantifier).
  - (): Groups patterns together.
  - \: Escapes a metacharacter (e.g., \\d).
  - *: Matches 0 or more repetitions of the preceding pattern.
  - +: Matches 1 or more repetitions of the preceding pattern.
  - ^ : Matches the start of a string.
  - $: Matches the end of a string.

Metacharacters, Character Classes and Quantifiers

- Regular Expression Metacharacters:
  - ?: Matches 0 or 1 repetition of the preceding pattern.
  - .: Matches any character except a newline.
  - |: Acts as a logical OR between two patterns.
- Character Classes:
  - \d: Matches any digit (0-9).
  - \w: Matches any alphanumeric character (letters and digits) and underscores.
  - \s: Matches any whitespace character (spaces, tabs, newlines).
- Quantifiers:
  - {n}: Matches exactly n repetitions of the preceding pattern.
  - {n,}: Matches n or more repetitions of the preceding pattern.
  - {n,m}: Matches between n and m repetitions of the preceding pattern.

Examples

```
# Square Brackets []
In [1]: import re

In [2]: pattern = '[aeiou]'  # Matches any vowel

In [3]: print('match' if re.fullmatch(pattern, 'e') else 'no match')
match

In [4]: print('match' if re.fullmatch(pattern, 'x') else 'no match')
no match

# Curly Braces {}
pattern = 'a{3}'  # Matches exactly three 'a's
In [9]: print('match' if re.fullmatch(pattern, 'aaa') else 'no match')
match

In [10]: print('match' if re.fullmatch(pattern, 'aaaa') else 'no match')
no match

# Parentheses ()
pattern = '(ab)+'  # Matches one or more repetitions of 'ab'
In [12]: print('match' if re.fullmatch(pattern, 'abab') else 'no match')
match

In [13]: print('match' if re.fullmatch(pattern, 'ab') else 'no match')
match

In [14]: print('match' if re.fullmatch(pattern, 'aba') else 'no match')
no match
```

Examples

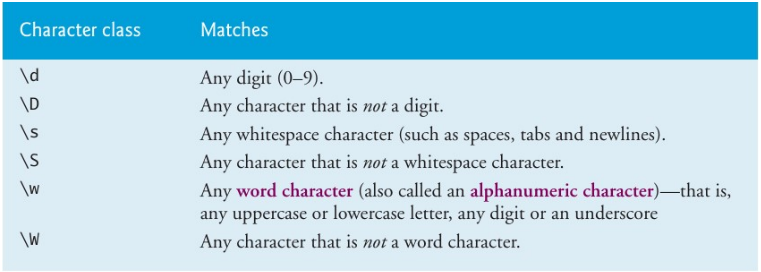
```
# Backslash \
pattern = r'\d{3}'  # Matches exactly three digits
In [16]: print('match' if re.fullmatch(pattern, '123') else 'no match')
match

In [17]: print('match' if re.fullmatch(pattern, '12') else 'no match')
no match
#The r tells Python to treat the string as a raw string, meaning backslashes (\)
      are not treated as escape characters.

# Asterisk *: Matches 0 or more repetitions of the preceding pattern
pattern = 'a*b'  # Matches 'b', 'ab', 'aab', etc.
In [19]: print('match' if re.fullmatch(pattern, 'c') else 'no match')
match

In [20]: print('match' if re.fullmatch(pattern, 'aaa') else 'no match')
no match

In [21]: print('match' if re.fullmatch(pattern, 'aaac') else 'no match')
match

#Plus +: Matches 1 or more repetitions of the preceding pattern
pattern = 'a+b'  # Matches 'ab', 'aab', 'aaab', etc.
In [23]: print('match' if re.fullmatch(pattern, 'aad') else 'no match')
match

In [24]: print('match' if re.fullmatch(pattern, 'd') else 'no match')
no match
```

Examples

```
# Caret ^: Matches the start of a string.

# Dollar Sign $: Matches the end of a string.

#Question Mark ?: Matches 0 or 1 repetition of the preceding pattern.
pattern = 'a?b'  # Matches 'b' or 'ab'
print(re.fullmatch(pattern, 'ab'))    # Match
print(re.fullmatch(pattern, 'b'))     # Match
print(re.fullmatch(pattern, 'aab'))   # No match

#Dot .: Matches any character except a newline
pattern = 'a.c'  # Matches 'a' followed by any character and then 'c'
print(re.fullmatch(pattern, 'abc'))  # Match
print(re.fullmatch(pattern, 'adc'))  # Match
print(re.fullmatch(pattern, 'ac'))   # No match

#Pipe |: Acts as a logical OR between two patterns.
pattern = 'cat|dog'  # Matches 'cat' or 'dog'
print(re.fullmatch(pattern, 'cat'))  # Match
print(re.fullmatch(pattern, 'dog'))  # Match
print(re.fullmatch(pattern, 'bat'))  # No match
```

Other Predefined Classes

- If you want to match any metacharacter literally (like \, *, +, etc.), you need to escape it using a backslash.
  - For example, \\matches a single backslash and \$ matches a dollar sign.



Character class	Matches
`\d`	Any digit (0–9).
`\D`	Any character that is *not* a digit.
`\s`	Any whitespace character (such as spaces, tabs and newlines).
`\S`	Any character that is *not* a whitespace character.
`\w`	Any **word character** (also called an **alphanumeric character**)—that is, any uppercase or lowercase letter, any digit or an underscore
`\W`	Any character that is *not* a word character.

Custom Character Classes in Regular Expressions

1. **Square Brackets for Custom Character Classes:**
   - Square brackets `[]` are used to define a custom character class that matches a single character from the list of characters inside the brackets.
   - Example:

     `[aeiou]`: Matches any lowercase vowel.
     `[A-Z]`: Matches any uppercase letter.
     `[a-z]`: Matches any lowercase letter.
     `[a-zA-Z]`: Matches any letter, whether lowercase or uppercase.
2. **Example of Validating a First Name:**
   - The pattern `[A-Z][a-z]*` ensures that the first name starts with an uppercase letter followed by any number of lowercase letters.
   - `[A-Z]`: Matches one uppercase letter (A-Z).
   - `[a-z]*`: Matches zero or more lowercase letters (a-z).

Example:

- `'Amanda'`: Matches (Valid)

Caret (ˆ) in Custom Character Classes

3. Caret (ˆ) in Custom Character Classes:

- The caret ˆ at the beginning of a custom character class negates the character class, meaning it will match any character except the ones listed in the brackets.
- Example:
  - `[^a-z]`: Matches any character that is not a lowercase letter.

In Python:

```
'Match' if re.fullmatch('[^a-z]', 'A') else 'No match'
# Output: 'Match'

'Match' if re.fullmatch('[^a-z]', 'a') else 'No match'
# Output: 'No match'
```

Replacing Substrings: sub Function

Function sub: Replacing Patterns
The sub function in the re module replaces all occurrences of a pattern with a replacement string.

- Example: Convert a tab-delimited string to a comma-delimited string:
- Use count to limit the number of replacements.

Example 1:

```
import re
# Replace all tabs with commas
re.sub(r'\t', ', ', '1\t2\t3\t4')  # Output: '1, 2, 3, 4'
```

Example 2:

```
# Replace only the first two tabs with commas
re.sub(r'\t', ', ', '1\t2\t3\t4', count=2)  # Output: '1, 2, 3\t4'
```

Splitting Strings: `split` Function

Function `split`: Tokenizing Strings

The `split` function breaks a string into pieces based on a regular expression pattern and returns a list of substrings.

- Example: Split a string at commas followed by optional whitespace.
- Use `maxsplit` to specify the maximum number of splits.

Example 1:

```
# Split at commas followed by optional whitespace
re.split(r',\s*', '1,  2,  3,4, 5,6,7,8')
# Output: ['1', '2', '3', '4', '5', '6', '7', '8']
```

Example 2:

```
# Split with a maximum of 3 splits
re.split(r',\s*', '1,  2,  3,4, 5,6,7,8', maxsplit=3)
# Output: ['1', '2', '3', '4, 5,6,7,8']
```

Function `search`: Finding the First Match

Function `search`: Finding the First Match
The `search` function looks for the first occurrence of a substring that matches a regular expression.

- Returns a match object containing the matching substring.
- Uses the `group` method to retrieve the matched substring.

Example 1:

```
import re
result = re.search('Python', 'Python is fun')
result.group() if result else 'not found'  # Output: 'Python'
```

Example 2:

```
result2 = re.search('fun!', 'Python is fun')
result2.group() if result2 else 'not found'  # Output: 'not found'
```

Function `match`: Match at the Beginning

Function `match`: Match Only at the Beginning

The `match` function searches for a match only at the beginning of the string.

- Returns a match object if the pattern matches the beginning.

Example:

```
result = re.match('Python', 'Python is fun')
result.group() if result else 'not found'  # Output: 'Python'
```

Ignoring Case with `flags`

Ignoring Case with `flags`
The `flags` keyword argument allows you to change how regular expressions are matched. For case-insensitive matching, use `re.IGNORECASE`.
Example:

```
result3 = re.search('Sam', 'SAM WHITE', flags=re.IGNORECASE)
result3.group() if result3 else 'not found'  # Output: 'SAM'
```

Using ^ and $ Anchors

Using ^ and $ Anchors

The ^ metacharacter matches the beginning, and $ matches the end of the string.

Example 1:

```
result = re.search('^Python', 'Python is fun')
result.group() if result else 'not found'  # Output: 'Python'
```

Example 2:

```
result = re.search('fun$', 'Python is fun')
result.group() if result else 'not found'  # Output: 'fun'
```

Function findall: Finding All Matches

Function `findall`: Finding All Matches
The `findall` function finds every matching substring in a string and returns a list of matches.
Example:

```
contact = 'Wally White, Home: 555-555-1234, Work: 555-555-4321'
re.findall(r'\d{3}-\d{3}-\d{4}', contact)
# Output: ['555-555-1234', '555-555-4321']
```

Function `finditer`: Finding Matches with Iterators

Function `finditer`: Finding Matches with Iterators
The `finditer` function works similarly to `findall`, but returns an iterator of match objects.

- This is memory-efficient for large datasets.

Example:

```
for phone in re.finditer(r'\d{3}-\d{3}-\d{4}', contact):
    print(phone.group())
# Output:
# 555-555-1234
# 555-555-4321
```

Capturing Substrings in a Match

Capturing Substrings in a Match
Use parentheses () to capture substrings in a match. These are returned as a tuple by the group method.
Example:

```
text = 'Charlie Cyan, e-mail: demo1@deitel.com'
pattern = r'([A-Z][a-z]+ [A-Z][a-z]+), e-mail: (\w+@\w+\.\w{3})'
result = re.search(pattern, text)
result.groups()  # Output: ('Charlie Cyan', 'demo1@deitel.com')
```

Accessing Individual Groups:

```
result.group(1)  # Output: 'Charlie Cyan'
result.group(2)  # Output: 'demo1@deitel.com'
```