Daily Notes - Input and Output

Regular expressions, often referred to as "regexes," are a powerful and compact way of representing collections of strings. In Python, these regular expressions are managed using the re module. They have a wide range of applications, and they are used for the following main purposes:

Parsing: Regular expressions are used to identify and extract pieces of text that meet specific criteria within a larger body of text.

Searching: They are employed to locate substrings that can take various forms while avoiding mismatches. For example, finding files with different extensions like 'pet.png', 'pet.gif', 'pet.mpg,' but not 'carpet.gif.'

Searching and Replacing: Regular expressions enable you to find substrings and replace specific content within the matched strings. For instance, you can replace phone numbers in a standardized format, like changing '071 234 5678' to '+2771 234 5678.'

Splitting Strings: You can split a string based on a certain character or pattern, allowing you to divide a string into smaller parts. For example, you can split a comma-delimited list into individual elements whenever a ',' is encountered.

Validation: Regular expressions are used to check whether a string conforms to specific criteria. For instance, you can use a regex pattern to validate whether an email address follows the standard format.

However, it's important to note that regular expressions have limitations:

Recursion: They can only handle recursive (repeating) structured text if the maximum number of recursions is known. Complex, nested structures can be challenging to work with.

Maintenance: Large and complex regular expressions can become difficult to maintain and understand over time.

Daily Notes - Reading and Writing Files

Regular expressions are a powerful tool for pattern matching and text manipulation in Python, and they are managed using the re-module. The section is divided into four subsections:

Matching Individual Characters or Groups of Characters: You will learn about basic regex syntax, including the dot (.), caret (^), dollar sign ($), asterisk (*), plus sign (+), question mark (?), and more. These symbols allow you to match specific characters or character patterns within text.

Quantifying Matches: You will discover how to quantify matches using symbols like '*', '+', '{m}', '{m,n}', and their non-greedy versions. These quantifiers help you specify how many occurrences of a character or group of characters you want to match.

Creating and Grouping Sub-Expressions: You will understand how to create and group sub-expressions, allowing you to apply quantifiers and match more complex patterns. Parentheses () are used for grouping and capturing parts of a regex pattern.

Using Lookahead and Flags: You will explore lookahead assertions, which allow you to define conditions that must be met for a match to occur, without consuming the matched text. You will also learn about flags that modify how regular expressions work, such as the DOTALL and MULTILINE flags.

Additionally, you are provided with a table that lists the functions available in the re-module, along with their descriptions. These functions help you perform various regex-related tasks, from finding matches to replacing text and more.

The section also explains the importance of using raw strings (prefixing strings with 'r') when working with regular expressions in Python, as it helps avoid the need for double-esc backslashes. Raw strings are recommended for regex patterns to simplify and clarify the expressions.

In summary, this section serves as a comprehensive introduction to regular expression concepts and syntax in Python, laying the foundation for your understanding and usage of regex patterns in your programming endeavors.

Daily Notes - Activity 1 - Read Methods

```
import re

# Get the input string from the user
input_string = input("Enter a string that contains numbers: ")

# Remove all numbers except '5'
result_string = re.sub(r'[^5]', '', input_string)

# Print the new string
print("\nNew string:\n" + result_string)
```

Daily Notes - Write Methods

Matching Individual Characters: A single character in a regular expression, such as '5' or 'b', matches only one occurrence of that character. If no quantifier is specified, only one occurrence is matched. For example, the regex 'you' matches 'your', 'YouTube', and 'fromMe2you'.

Special Characters: Some characters have special meanings in regular expressions, such as '.', '^', '$', '?', '+', '*', '[]', '()', etc. To match these special characters as literals, they need to be preceded by a backslash ('\'). For example, '\.' matches a literal period ('.'). Python's escape characters, such as '\t', '\n', '\'', and '\"', have special meanings and can also be used in regex.

Character Classes: Character classes are enclosed in square brackets '[]' and allow you to define sets of characters to match. For example, '[ea]' matches 'hundred' and 'radar' but not 'read'. You can define ranges within character classes, like '[0-9]' for digits. To exclude characters, use '^' inside a character class. To match any character except digits, use '[^0-9]'.

Character Class Shortcuts: Python's regular expressions provide shortcuts for common character classes. These include '\d' (matches any Unicode digit), '\D' (matches any character that is not a Unicode decimal digit), '\s' (matches Unicode whitespace characters), '\S' (matches characters that are not Unicode whitespace characters), '\w' (matches Unicode word characters, including letters, digits, and underscores), and '\W' (matches characters that are not Unicode word characters). For example, '\d' is equivalent to '[0-9]' and '\s' matches spaces, tabs, and line breaks.

The provided example demonstrates how to change the format of dates that use different separators like '-', '.', and '/'. By using the regular expression '[-./]', you can match any of these separators and replace them with '/' to standardize the date format.

In a more complex example, a regex pattern is used to match words in a sentence. The pattern 'r'^[\W][\w][\W]*' uses character classes to match and extract words from the sentence. The regex removes words and any non-alphanumeric characters surrounding them, counting the words as they go.

This section provides a fundamental understanding of character matching and character classes in regular expressions, which are essential tools for text processing and pattern matching in Python.

Daily Notes - Activity 2 - Write Methods

```
 # Prompt the user to enter a file name
file_name = input("Enter a file name: ")

# Get the file extension
file_extension = file_name.split('.')[-1].lower()

# Define a dictionary mapping file extensions to file types and applications
file_types = {
"txt": ("Text Document", "Notepad"),
"docx": ("Word Document", "MS Word"),
"pdf": ("PDF Document", "Adobe Acrobat"),
"jpg": ("Image File", "Image Viewer"),
"xlsx": ("Excel Spreadsheet", "MS Excel"),
}
```

```
# Check if the extension is in the dictionary
if file_extension in file_types:
file_type, application = file_types[file_extension]
print(f"{file_name} is a {file_type} and can be opened using {application}")
else:
print("File type not recognized or unsupported.")
```

My own views on lists and methods

Flags are important modifiers in regular expressions that control how the regex engine behaves when matching patterns in text. They allow you to fine-tune the matching process to suit your needs. In Python, flags are provided as optional arguments in the re.compile() function or can be set as part of the regular expression pattern. Here's a summary of the commonly used flags in regular expressions:

re.A or re.ASCII: This flag makes special sequences like \b, \B, \s, \S, \w, and \W assume that the input strings are ASCII. It's used to ensure that these special sequences behave as expected when working with ASCII characters.

re.I or re.IGNORECASE: The IGNORECASE flag makes the regex pattern case-insensitive. It enables the regex engine to match characters regardless of their case. For example, when matching [aeiou] with the IGNORECASE flag, it will match both uppercase and lowercase vowels.

re.M or re.MULTILINE: The MULTILINE flag alters the behavior of the ^ and $ anchors. Without this flag, ^ and $ match the start and end of the string. With the MULTILINE flag, they also match the start and end of each line within a multi-line string.

re.S or re.DOTALL: The DOTALL flag makes the . (dot) character match any character, including newline characters (\n). By default, the dot matches any character except newline. DOTALL enables you to work with multi-line strings more effectively.

re.X or re.VERBOSE: The VERBOSE flag allows you to write regular expressions with added white space and comments. This makes complex regular expressions more readable by allowing you to include explanatory comments and line breaks within the regex pattern.

Daily Notes - Python Tutorial: File Objects - Reading and Writing to Files

In the examples provided, two flags are demonstrated: IGNORECASE and VERBOSE.

In the IGNORECASE example, the flag is used to perform a case-insensitive match, making it easier to match characters regardless of their case. In this case, the regex pattern [aeiou] is used to match vowels, and the IGNORECASE flag ensures that it matches both uppercase and lowercase vowels.

In the VERBOSE example, the VERBOSE flag is used to create a more readable and well-documented regular expression for matching octal and hexadecimal numbers. By using the VERBOSE flag, you can

include comments and whitespace within the regex pattern to explain and break down complex expressions.

Flags are valuable tools for customizing the behavior of regular expressions to meet specific requirements and simplify the writing and understanding of complex patterns.

My Views on the Day

What were the important features of the day?

Today's information focused on various aspects of regular expressions, including their syntax, character classes, quantifiers, flags, and practical examples of their use.
Key features included understanding regular expression syntax and using flags to modify the behavior of regular expressions.
Practical examples, such as removing specific numbers from a string or validating phone numbers, were important features of the day.

What activities were beneficial?

Activities that demonstrated practical applications of regular expressions, such as removing specific numbers from a string, were beneficial in helping understand how to use regular expressions in real scenarios.
The explanations about character classes and flags, along with their respective examples, were also valuable in gaining a comprehensive understanding of regular expressions.

What activities (if any) were too easy?

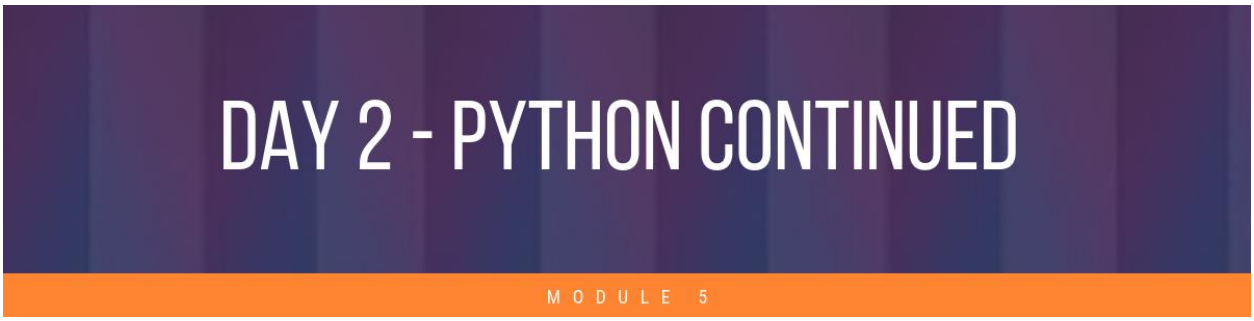
The activity that asked about the "import re" statement was relatively easy as it involved recognizing a common Python import statement.

What activities need more time?

The activity related to file types and their associated software could be expanded to cover more file types and applications. This would provide a broader understanding of how to identify and work with different types of files.
Additionally, more advanced regular expression examples and complex patterns could be explored for those who want to delve deeper into this topic. Regular expressions can become quite intricate in more advanced use cases, and additional practice would be helpful.

Daily Notes - Day 2 Reflections

this was a productive day and i have figured out how to use these regular expressions on our banking app/ investment calculator.



Many programs need to carry out the same operation repeatedly, with only slight differences in the calculation or processing. To repeat the code over and over is clumsy and consumes memory. One way of avoiding this is to put the repeated code into a function, which is called each time the operation needs to be carried out. A function can take zero or multiple inputs, which are called parameters.

In Python, functions are created by specifying a name, a list of parameters (which are local variables within the function), and an optional block of code to provide a return value. There are three types of functions: ordinary functions (perform calculations and return results), procedure functions (execute procedures without returning results), and factory functions (generate values without taking parameters).

In the example provided, a function named `calculateTax` is defined to calculate and print the tax based on a given salary. It takes a `salary` parameter and uses conditional statements to calculate the tax rate and tax amount. The local variables within the function, such as `rate` and `tax`, are not accessible outside the function. To make the value of a local variable available outside the function, a `return` statement can be used.

Example 2 demonstrates the use of default parameters in functions. The `num` function has a parameter `y` with a default value of 15. Even if the value of the global variable `x` is changed, the default value of `y` remains 15 unless explicitly overridden in a function call.

In summary, functions in Python are defined with a name, parameters, and a code block. They serve different purposes, such as performing calculations, executing procedures, or generating values. Default parameters can be used to assign default values to function parameters, which are only used if no specific value is provided when calling the function.

Daily Notes - Reading and Writing Files

In Python, functions are created by specifying a name, a list of parameters (which are local variables within the function), and an optional block of code to provide a return value. There are three types of functions: ordinary functions (perform calculations and return results), procedure functions (execute procedures without returning results), and factory functions (generate values without taking parameters).

In the example provided, a function named calculateTax is defined to calculate and print the tax based on a given salary. It takes a salary parameter and uses conditional statements to calculate the tax rate and tax amount. The local variables within the function, such as rate and tax, are not accessible outside

the function. To make the value of a local variable available outside the function, a return statement can be used.

Example 2 demonstrates the use of default parameters in functions. The num function has a parameter y with a default value of 15. Even if the value of the global variable x is changed, the default value of y remains 15 unless explicitly overridden in a function call.

In summary, functions in Python are defined with a name, parameters, and a code block. They serve different purposes, such as performing calculations, executing procedures, or generating values. Default parameters can be used to assign default values to function parameters, which are only used if no specific value is provided when calling the function.

Quantifying Matches: You will discover how to quantify matches using symbols like '*', '+', '{m}', '{m,n}', and their non-greedy versions. These quantifiers help you specify how many occurrences of a character or group of characters you want to match.

Creating and Grouping Sub-Expressions: You will understand how to create and group sub-expressions, allowing you to apply quantifiers and match more complex patterns. Parentheses () are used for grouping and capturing parts of a regex pattern.

Using Lookahead and Flags: You will explore lookahead assertions, which allow you to define conditions that must be met for a match to occur, without consuming the matched text. You will also learn about flags that modify how regular expressions work, such as the DOTALL and MULTILINE flags.

Additionally, you are provided with a table that lists the functions available in the re-module, along with their descriptions. These functions help you perform various regex-related tasks, from finding matches to replacing text and more.

The section also explains the importance of using raw strings (prefixing strings with 'r') when working with regular expressions in Python, as it helps avoid the need for double-esc backslashes. Raw strings are recommended for regex patterns to simplify and clarify the expressions.

In summary, this section serves as a comprehensive introduction to regular expression concepts and syntax in Python, laying the foundation for your understanding and usage of regex patterns in your programming endeavors.

Daily Notes - Activity 1 - Read Methods

```
import re

# Get the input string from the user
input_string = input("Enter a string that contains numbers: ")

# Remove all numbers except '5'
```

```
result_string = re.sub(r'[^5]', '', input_string)

# Print the new string
print("\nNew string:\n" + result_string)
```

Daily Notes - Write Methods

The provided information explains the usage of the `random` module in Python and provides an example of a program that uses this module to generate random lottery numbers. Additionally, it offers another example where two functions work together to calculate and print the product and average of three user-input numbers.

Here's a summary of the key points:

1. The `random` module in Python is used to generate random numbers. It employs the Mersenne Twister algorithm, which has a long period before repeating and is fast.

2. The `random` module functions are actually bound methods of an instance of the `random.Random` class.

3. The example program (`lotto_number`) demonstrates the use of the `random.randrange()` function to generate random numbers within a specified range. However, it does not check for duplicate numbers.

4. The program mentions other useful `random` module functions, such as `randint()`, `sample()`, `choice()`, and `randrange()`.

5. Example 4 illustrates a program that takes three user-input numbers and uses two functions, `determineProduct` and `determineAve`, to calculate and print the product and average of these numbers.

6. The main program first calls `determineProduct`, which calculates the product of the three numbers and prints it. Then, it calls `determineAve`, which calculates the average and returns it. Finally, it prints the calculated average.

7. The example demonstrates how functions can be used to encapsulate specific tasks, making the code more modular and readable.

In summary, the provided information gives an overview of using the `random` module in Python and showcases the concept of functions, as well as how multiple functions can work together in a program to perform various tasks.

Daily Notes - Activity 2 - Write Methods

```
# Prompt the user to enter a file name
file_name = input("Enter a file name: ")
```

```
# Get the file extension
file_extension = file_name.split('.')[-1].lower()

# Define a dictionary mapping file extensions to file types and applications
file_types = {
"txt": ("Text Document", "Notepad"),
"docx": ("Word Document", "MS Word"),
"pdf": ("PDF Document", "Adobe Acrobat"),
"jpg": ("Image File", "Image Viewer"),
"xlsx": ("Excel Spreadsheet", "MS Excel"),
}

# Check if the extension is in the dictionary
if file_extension in file_types:
file_type, application = file_types[file_extension]
print(f"{file_name} is a {file_type} and can be opened using {application}")
else:
print("File type not recognized or unsupported.")
```

My own views on lists and methods

Recursion in programming is a technique where a function calls itself to solve a problem. Recursive functions have two essential parts: a base case and a recursive case.

The base case is the condition that determines when the recursion should stop. It's necessary to prevent infinite recursion.

The recursive case is where the function calls itself with modified arguments to make progress toward the base case.

Example 5 presents a recursive function called Fibonacci, which calculates the Fibonacci sequence. In the Fibonacci sequence, each number is the sum of the previous two numbers, starting with 0 and 1.

In the Fibonacci function, the base case is defined as if num <= 1, and in this case, it returns 1. When num is greater than 1, the function calculates the Fibonacci number by calling itself with Fibonacci(num - 1) and multiplying the result by num.

The for loop at the end of the example demonstrates how the Fibonacci function is used to calculate and print the first 11 Fibonacci numbers, ranging from 0 to 10.

The output of the program shows the Fibonacci numbers and their corresponding values.

Daily Notes - Python Tutorial: File Objects - Reading and Writing to Files

In summary, recursive functions are a powerful concept in programming that allows functions to call themselves to solve problems. The Fibonacci sequence is a classic example to illustrate this concept,

where each number depends on the results of the previous two numbers. Recursive functions should always have a base case to ensure termination and prevent infinite recursion.

My Views on the Day

What were the important features of the day?

Today's information focused on various aspects of regular expressions, including their syntax, character classes, quantifiers, flags, and practical examples of their use.
Key features included understanding regular expression syntax and using flags to modify the behavior of regular expressions.
Practical examples, such as removing specific numbers from a string or validating phone numbers, were important features of the day.

What activities were beneficial?

Activities that demonstrated practical applications of regular expressions, such as removing specific numbers from a string, were beneficial in helping understand how to use regular expressions in real scenarios.
The explanations about character classes and flags, along with their respective examples, were also valuable in gaining a comprehensive understanding of regular expressions.

What activities (if any) were too easy?

The activity that asked about the "import re" statement was relatively easy as it involved recognizing a common Python import statement.

What activities need more time?

The activity related to file types and their associated software could be expanded to cover more file types and applications. This would provide a broader understanding of how to identify and work with different types of files.
Additionally, more advanced regular expression examples and complex patterns could be explored for those who want to delve deeper into this topic. Regular expressions can become quite intricate in more advanced use cases, and additional practice would be helpful.

Daily Notes - Day 2 Reflections

this was a productive day and i have figured out how to use these regular expressions on our banking app/ investment calculator.