RegEx

Regular Expressions, often referred to as regex, are powerful pattern-matching tools used to search, extract, and manipulate text in Python. They provide a concise and flexible way to identify specific patterns within strings. First we need to import this to use in our code by using "import re"

Regular expressions offer a wide range of pattern-matching options, including character classes, anchors, modifiers, and more. They can be extremely powerful in various text processing tasks, such as data validation, parsing, and manipulation.The re module offers a set of functions that allows us to search a string for a match. And there are multiple Metacharcters, Special Sequences and Sets of these methods which can be used according to the requirements. Some of these are: '^' it is used when checking if the string starts with some specific keyword. '$' it is used when checking if the string ends with some specific keyword.

1. the re.search() method is used to search for a pattern within a string and return the first occurrence of a match. It scans the entire string and stops at the first match found.

In [11]:
```
#Example 1
import re

#Checking if the string starts with "The" and ends with "Spain":

txt = "The internship has been started"
x = re.search("^The.*started$", txt)

if x:
  print("YES! We have a match!")
else:
  print("No match")
```

```
YES! We have a match!
```

In [13]:
```
#Example 2
text = "The quick brown fox jumps over the lazy dog"
pattern = r'\bfox\b'

match = re.search(pattern, text)
if match:
    print("Match found:", match.group())
```

```
Match found: fox
```

In this case, the pattern \bfox\b ensures that only the word "fox" is matched and not partial matches like "foxy" or "foxes". The re.search() method finds the first occurrence of the word "fox" in the input string and returns the match object.

2. The findall() method in Python's re module is used to find all non-overlapping occurrences of a pattern in a string and return them as a list of strings. It searches the entire input string, unlike search() that stops at the first match. It is commonly used when you need to extract multiple occurrences of a pattern from a string.

In [9]:
```
text = "Learning the RegEx! and we can learn python easily"
pattern = r"(Learning).*(python)"
matches = re.findall(pattern, text)
print(matches)
```

```
[('Learning', 'python')]
```

In this example, the pattern (Learning).*(python) matches the substring "Learning the RegEx! and we can learn python" in the input text. The findall() method extracts and returns this matched substring as a list of tuples, where each tuple contains the captured groups. The output is [('Learning', 'python')], indicating that the pattern was found once, and the captured groups are "Learning" and "python".

In [14]:
```
import re

text = "Contact us at info@example.com or support@example.com"
pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b'

emails = re.findall(pattern, text)
print(emails)
```

```
['info@example.com', 'support@example.com']
```

In above example, the regular expression pattern is designed to match valid email addresses. It uses various components to define the structure of an email address, such as alphanumeric characters, special characters like dot and plus, and specific patterns for domain names. The re.search() function finds the first occurrence of an email address in the given text, and match.group() retrieves the matched email address from the match object.

3. The `split()` method in Python is used to split a string into multiple substrings based on a specified delimiter. It returns a list of substrings obtained by splitting the original string. The delimiter determines where the split occurs, and it is excluded from the resulting substrings.

In [17]:
```
#Example 1
sentence = "This is a sentence."
tokens = sentence.split(" ")
print(tokens)
```

```
['This', 'is', 'a', 'sentence.']
```

In [19]:
```
#Example 2
path = "/home/user/Documents/file.txt"
```

```
parts = path.split("/")
print(parts)
```

```
['', 'home', 'user', 'Documents', 'file.txt']
```

These examples demonstrate how the split() method can be used to split a string into substrings based on a specified delimiter, such as a comma, space, or any other character.

4. The sub() method in Python is used to substitute occurrences of a pattern in a string with a replacement value. It performs a search for the pattern in the string and replaces all occurrences with the provided replacement value.

In [22]:
```
#Example 1
text = "Hello, World!"
new_text = re.sub(r"World", "Python", text)
print(new_text)
```

```
Hello, Python!
```

In above example, the sub() method replaces all occurrences of "World" with "Python" in the text string.

In [24]:
```
#Example 2
text = "I love cats, but I'm allergic to dogs."
new_text = re.sub(r"(cats|dogs)", "birds", text)
print(new_text)
```

```
I love birds, but I'm allergic to birds.
```

In above example, the sub() method replaces occurrences of either "cats" or "dogs" with the value "birds" in the text string. The regular expression (cats|dogs) matches either "cats" or "dogs".

5. A Match Object is an object containing information about the search and the result. The Match object has properties and methods used to retrieve information about the search, and the result:

.span() returns a tuple containing the start-, and end positions of the match. .string returns the string passed into the function .group() returns the part of the string where there was a match

In [28]:
```
#example 1
txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())

#example 2
txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

```
#example 3
txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

```
(12, 17)
The rain in Spain
Spain
```

In example 1, the pattern \bS\w+ matches the word "Spain" in the given text. The print(x.span()) statement will output (12, 17), indicating that the matched substring starts at index 12 and ends at index 17 in the original text.

In example 2, the pattern \bS\w+ matches the word "Spain" in the given text. The print(x.string) statement will output "The rain in Spain", which is the original text that was searched.

In example 3, the pattern \bS\w+ matches the word "Spain" in the given text. The print(x.group()) statement will output "Spain", which is the substring that matched the pattern.

2. Lambdas Functions

A lambda function is an anonymous function in Python defined using the lambda keyword. It allows you to create small, one-line functions without a name. Lambda functions are defined using the lambda keyword followed by the input arguments and a colon. They can take any number of arguments, but can only have a single expression. They are typically used when you need a small function for a specific task and don't want to define a separate named function. Lambda functions are often used in conjunction with higher-order functions like map(), filter(), and reduce().

In [31]:
```
#example 1
add = lambda x, y: x + y
print(add(2, 3))

square = lambda x,y: x**y
print(square(2,2))
```

```
5
4
```

In [30]:
```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

```
22
33
```

The above code defines a function myfunc that returns a lambda function which multiplies its input by a given number n. It then creates two instances of the lambda function, mydoubler with n as 2 and mytripler with n as 3. Finally, it calls the lambda functions with an input of 11, resulting in the output of 22 and 33 respectively.

3. List Comprehensions

List comprehensions in Python provide a concise way to create lists based on existing lists or other iterable objects. They allow you to combine loops and conditional statements into a single line of code, making it easier to create new lists with specific criteria. The basic syntax of a list comprehension is [expression for item in iterable if condition], where the expression defines the operation on each item, the item is the variable representing elements in the iterable, and the condition is an optional filtering condition.

In [38]:
```
#example 1
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = [x for x in numbers if x % 2 == 0]
print(evens)

#example 2
names = ['John', 'Jane', 'Alice', 'Jack', 'Mark']
j_names = [name for name in names if name.startswith('J')]
print(j_names)
```

```
[2, 4, 6, 8, 10]
['John', 'Jane', 'Jack']
```

Example 1 extracts even numbers from the list given. numbers is a list containing numbers from 1 to 10. The list comprehension [x for x in numbers if x % 2 == 0] creates a new list called evens by iterating over each element x in the numbers list and including it in the evens list if x is divisible by 2

In Example 2, names is a list of strings containing names. The list comprehension [name for name in names if name.startswith('J')] creates a new list called j_names by iterating over each element name in the names list and including it in the j_names list if the name starts with the letter 'J'

4. Nested List Comprehensions

Nested list comprehensions refer to the usage of one or more list comprehensions within another list comprehension. It allows you to create complex nested structures or perform transformations on nested data.

In [34]:
```
#example 1
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
flattened = [num for sublist in nested_list for num in sublist]
print(flattened)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The above code takes a nested list and flattens it into a single list. It uses nested list comprehensions to iterate over each sublist in the nested list and extract the individual elements. The resulting flattened list is then printed.

In [37]:
```
#example 2
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transposed = [[row[i] for row in matrix] for i in range(len(matrix[0]))]
print(transposed)
```

```
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

The above code creates a transposed matrix by rearranging the rows and columns of a given matrix. It uses nested list comprehensions to iterate over the elements of the original matrix and construct the transposed matrix. The resulting transposed matrix is then printed.

5. Decorators

Decorators are a powerful tool in Python that allow programmers to modify the behavior of a function or class. They enable us to wrap a function, extending its behavior without permanently modifying it. This is achieved by using some key concepts such as functions as objects, nested functions, and closures. Understanding these concepts will provide a solid foundation for understanding and implementing decorators effectively.

In [39]:
```
#Example 1
def uppercase_decorator(func):
    def wrapper():
        result = func()
        return result.upper()
    return wrapper

@uppercase_decorator
def say_hello():
    return "Hello, World!"

print(say_hello())
```

```
HELLO, WORLD!
```

The uppercase_decorator function is a decorator that takes a function and returns a new function that wraps around the original function. When the decorated function say_hello is called, it invokes the wrapper function. The wrapper function first calls the original function, stores the result, converts it to uppercase, and returns the uppercase result. The say_hello function is decorated with @uppercase_decorator syntax, indicating that the decorator is applied to it. The final output will be the uppercase version of "Hello, World!".

6. Iterators

In Python, an iterator is an object that implements the iterator protocol, which consists of the iter() and next() methods. Iterators are used to iterate over a collection of elements or to generate a sequence of values on-the-fly.

The iter() method returns the iterator object itself. It is called when an iterator is initialized or reset. The next() method returns the next element in the sequence. It is called repeatedly until it raises the StopIteration exception, indicating that there are no more elements to be returned

In [41]:
```
class MyIterator:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.end:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

my_iterator = MyIterator(1, 5)

for num in my_iterator:
    print(num)
```

```
1
2
3
4
5
```

The code defines a class MyIterator that implements the iterator protocol in Python. It has an **init** method to initialize the starting and ending values of the iterator. The **iter** method returns the iterator object itself. The **next** method is called to retrieve the next value from the iterator. It raises StopIteration when it reaches the end. An instance of MyIterator is created with a start value of 1 and an end value of 5. The for loop iterates over the my_iterator object, calling **next** on each iteration and printing the returned value. The output will be the numbers from 1 to 5.

7. Yield Keyword

The yield keyword in Python is used in the context of defining generator functions. When a function contains the yield keyword, it becomes a generator function instead of a regular

function. The yield keyword allows the function to produce a sequence of values that can be iterated over.

This allows for lazy evaluation of values, where the values are generated on-demand as they are requested, rather than generating the entire sequence upfront. It is memory-efficient and suitable for handling large or infinite sequences.

In [43]:
```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

# Using the generator function
my_generator = count_up_to(5)

# Iterating over the generator to retrieve values
for num in my_generator:
    print(num)
```

```
1
2
3
4
5
```

In the above code, the count_up_to function is a generator function that yields numbers from 1 up to a given n. Each time the yield statement is encountered, the current value of i is yielded, and the function's state is saved. The for loop iterates over the my_generator object, calling **next**() on each iteration and printing the yielded values. The output will be the numbers from 1 to 5.

8. Generators Expression

Generator expressions in Python are similar to list comprehensions, but instead of generating a list, they create an iterator that produces the values on-the-fly. Generator expressions allow you to generate values dynamically without storing them in memory all at once.

The syntax for a generator expression is similar to that of a list comprehension, but with parentheses instead of square brackets. Instead of creating a list, it creates an iterator object that produces values when iterated over.

In [46]:
```
my_generator = (x for x in range(5))

# Iterating over the generator to retrieve values
for num in my_generator:
    print(num)
```

```
#example 2
# Generator expression
even_numbers = (x for x in range(10) if x % 2 == 0)

# Iterating over the generator to retrieve even numbers
for num in even_numbers:
    print(num)
```

```
0
1
2
3
4
0
2
4
6
8
```

In the above code, the generator expression (x for x in range(5)) generates values from 0 to 4. When the for loop iterates over the my_generator object, it calls **next**() on each iteration and prints the yielded values. The output will be the numbers from 0 to 4.

Generator expressions are useful when you don't need to store all the values in memory at once and want to iterate over a large or infinite sequence. They provide a memory-efficient and concise way to generate values on-the-fly.