

Unit 5:Exception Handling and Regular Expressions



Python

Exceptions

Exception Handling

Try and Except

Nested try Block

Handling Multiple Exceptions in single Except Block

Raising Exception

Finally Block

User Defined Exceptions

What are Exceptions?

- An exception is an **event that occurs during the execution of programs that disrupt the normal flow of execution** (e.g., `KeyError` Raised when a key is not found in a dictionary.) An exception is a Python object that represents an error.
- Exception are useful to indicate different types of possible failure condition.
- For example, bellow are the few standard exceptions

`FileNotFoundError`

`ImportError`

`RuntimeError`

`NameError`

`TypeError`

Why use Exception

- **Standardized error handling:** Using built-in exceptions or creating a custom exception with a more precise name and description, you can adequately define the error event, which helps you debug the error event.
- **Cleaner code:** Exceptions separate the error-handling code from regular code, which helps us to maintain large code easily.
- **Robust application:** With the help of exceptions, we can develop a solid application, which can handle error event efficiently
- **Exceptions propagation:** By default, the exception propagates the call stack if you don't catch it. For example, if any error event occurred in a nested function, you do not have to explicitly catch-and-forward it; automatically, it gets forwarded to the calling function where you can handle it.
- **Different error types:** Either you can use built-in exception or create your custom exception and group them by their generalized parent class, or Differentiate errors by their actual class

What are Errors?

- An **error** is an action that is incorrect or inaccurate. For example, syntax error. Due to which the program fails to execute.
- The errors can be broadly classified into two types:
 1. Syntax errors
 2. Logical errors

Exception

- When writing a program, we, more often than not, will encounter errors.
- Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error
- Errors can also occur at runtime and these are called exceptions.
- They occur, for example, when a file we try to open does not exist (`FileNotFoundError`), dividing a number by zero (`ZeroDivisionError`)
- Whenever these type of runtime error occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Syntax error

- The syntax error occurs when we are not following the proper structure or syntax of the language. A syntax error is also known as a **parsing error**.
- When Python parses the program and finds an incorrect statement it is known as a syntax error. When the parser found a syntax error it exits with an error message without running anything.

Common Python Syntax errors:

- Incorrect indentation
- Missing colon, comma, or brackets
- Putting keywords in the wrong place.

```
print("Welcome to PYnative")  
    print("Learn Python with us..")
```

```
print("Learn Python with us..")  
    ^  
IndentationError: unexpected indent
```

Logical errors (Exception)

- Even if a statement or expression is syntactically correct, the error that occurs at the runtime is known as a **Logical error or Exception**. In other words, **Errors detected during execution are called exceptions**.

Common Python Logical errors:

- Indenting a block to the wrong level
- using the wrong variable name
- making a mistake in a boolean expression

```
a = 10
b = 20
print("Addition:", a + c)
```

```
print("Addition:", a + c)
NameError: name 'c' is not defined
```


Built-in Exceptions

- Python automatically generates many exceptions and errors. Runtime exceptions, generally a result of programming errors, such as:
- Reading a file that is not present
- Trying to read data outside the available index of a list
- Dividing an integer value by zero

Exception	Description
<code>AssertionError</code>	Raised when an <code>assert</code> statement fails.
<code>AttributeError</code>	Raised when attribute assignment or reference fails.
<code>EOFError</code>	Raised when the <code>input()</code> function hits the end-of-file condition.
<code>FloatingPointError</code>	Raised when a floating-point operation fails.
<code>GeneratorExit</code>	Raise when a generator's <code>close()</code> method is called.
<code>ImportError</code>	Raised when the imported module is not found.
<code>IndexError</code>	Raised when the index of a sequence is out of range.

KeyError

Raised when a key is not found in a dictionary.

KeyboardInterrupt

Raised when the user hits the interrupt key (Ctrl+C or Delete)

MemoryError

Raised when an operation runs out of memory.

NameError

Raised when a variable is not found in the local or global scope.

OSError

Raised when system operation causes system related error.

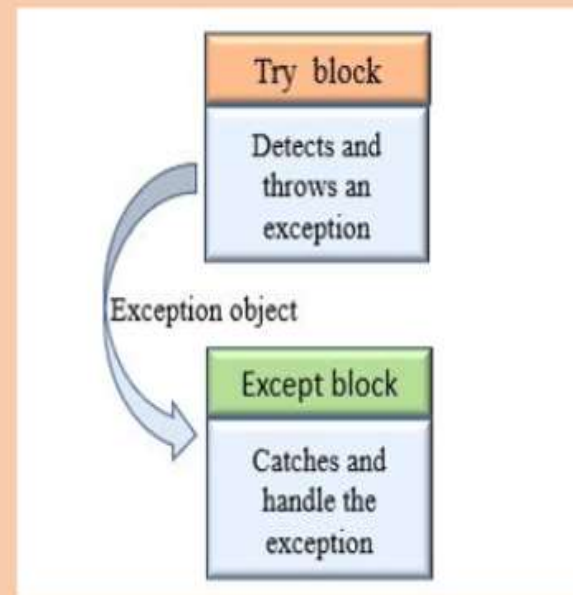
ReferenceError

Raised when a weak reference proxy is used to access a garbage collected referent.

Python Exception Handling

An exception is an event that occurs during the execution of programs that **disrupt the normal flow of execution**

1. Try-except-finally
2. Raise an exception
3. Exception chaining
4. Built-in exceptions
5. Custom exceptions



The try and except Block to Handling Exceptions

- When an exception occurs, Python stops the program execution and generates an exception message. It is highly recommended to handle exceptions. The doubtful code that may raise an exception is called risky code.
- To handle exceptions we need to use try and except block. Define risky code that can raise an exception inside the `try` block and corresponding handling code inside the `except` block.

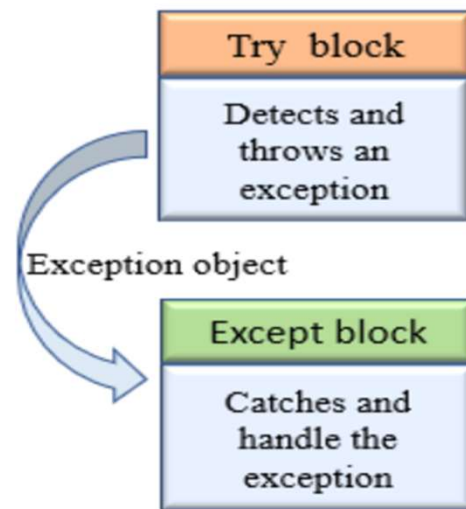


Exception Handling

- To handle exceptions, and to call code when an exception occurs, we can use a **try/except** statement.
- The **try** block contains code that might throw an exception.
- If that exception occurs, the code in the **try** block stops being executed, and the code in the **except** block is executed.
- If no error occurs, the code in the **except** block doesn't execute.

Syntax

```
try :  
    # statements in try block  
except :  
    # executed when exception occurred in try block
```



try-except block

The try block is for risky code that can raise an exception and the except block to handle error raised in a try block.

For example, if we divide any number by zero, try block will throw **ZeroDivisionError**, so we should handle that exception in the except block.

```
a = 10
b = 0
c = a / b
print("a/b = %d" % c)
```

```
Traceback (most recent call last):
  File "E:/demos/exception.py", line 3, in <module>
    c = a / b
ZeroDivisionError: division by zero
```

We can see in the above code when we are divided by 0; Python throws an exception as **ZeroDivisionError** and the program terminated abnormally.

We can handle the above exception using the **try...except** block. See the following code.

```
try:
    a = 10
    b = 0
    c = a/b
    print("The answer of a divide by b:", c)
except:
    print("Can't divide with zero. Provide different number")
```

```
Can't divide with zero. Provide different number
```


Catching Specific Exceptions

```
try:
    a = int(input("Enter value of a:"))
    b = int(input("Enter value of b:"))
    c = a/b
    print("The answer of a divide by b:", c)
except ValueError:
    print("Entered value is wrong")
except ZeroDivisionError:
    print("Can't divide by zero")
```

```
Enter value of a:Ten
Entered value is wrong
```

```
Enter value of a:10
Enter value of b:0
Can't divide by zero
```

```
Enter value of a:10
Enter value of b:2
The answer of a divide by b: 5.0
```

Handle multiple exceptions with a single except clause

```
try:
    a = int(input("Enter value of a:"))
    b = int(input("Enter value of b:"))
    c = a / b
    print("The answer of a divide by b:", c)
except(ValueError, ZeroDivisionError):
    print("Please enter a valid value")
```

#Program to perform division and understand exception handling

```
a = int ( input('Enter the first number ' ) )  
b = int ( input('Enter the second number ' ) )
```

```
try:  
    res = a / b  
    print('result = ' , res)  
except:  
    print('Exception Handled ')
```

```
print('End of program')
```

Nested Try Block

```
try:
    num = int ( input('Enter the numerator ' ) )
    den = int ( input('Enter the denominator ' ) )
    try:
        result = num / den;
        print('Result = ' , result)
    except:
        print('Divide by Zero Error')
except:
    print('Invalid Input')

print('End of Program ')
```



~

.

```
#Program to know the type of exception
```

```
import sys
```

```
a = int ( input('Enter the first number ' ) )
```

```
b = int ( input('Enter the second number ' ) )
```

```
try:
```

```
    res = a / b
```

```
    print('result = ' , res)
```

```
except:
```

```
    print('Exception Handled ')
```

```
    print("Oops!",sys.exc_info()[0],"occured.")
```

```
print('End of program')
```

```
|
```

- A **try** statement can have multiple different **except** blocks to handle different exceptions.

```
import sys
try :
    num = int ( input('Enter the numerator ' ) )
    den = int ( input('Enter the denominator ' ) )
    result = num / den;
    print('Result = ' , result)
except ValueError:
    print('Invalid Input ')
except ZeroDivisionError:
    print('Divide by Zero Error')

print('End of Program ')
```

```
#Program to demonstrate multiple except(catch) blocks
```

```
try:
```

```
    a = input('Enter the first number ' )
```

```
    b = input('Enter the second number ' )
```

```
    a = int(a)
```

```
    res = a + b
```

```
    print('result = ' , res)
```

```
except ValueError:
```

```
    print('Invalid Input Error')
```

```
except TypeError:
```

```
    print('Type Error')
```

```
except ZeroDivisionError:
```

```
    print('Divide by Zero Error')
```

```
print('End of program')
```


- Multiple exceptions can also be put into a single **except** block using parentheses, to have the **except** block handle all of them.

#Program to demonstrate handling multiple exception types in a single block

```
import sys

try:

    a = input('Enter the first number ' )
    b = input('Enter the second number ' )
    a = int(a)

    sum = a + b
    print('sum = ' , sum)
    quotient = a // b
    print('quotient = ' , quotient)

except (ValueError,TypeError):
    print('Invalid Input Error',sys.exc_info()[0])

except ZeroDivisionError:
    print('Divide by Zero Error')

print('End of program')
```

```
def divide(num,den):  
    res = num / den  
    return res  
  
try :  
    num = int ( input('Enter the numerator ' ) )  
    den = int ( input('Enter the denominator ' ) )  
  
    result = divide(num , den)  
    print('Result = ' , result)  
  
except ValueError:  
    print('Main Block : Invalid Input ' )  
except ZeroDivisionError:  
    print('Main Block : Divide by Zero error')  
  
print('End of Program ' )  
■
```

Raising Exceptions

exception_raise.py - D:/sikander/python/exception_raise.py (3.6.3)

File Edit Format Run Options Window Help

```
try:
    print('Enter the marks ' )
    marks = int( input() )

    if marks < 0 or marks > 100:
        raise ValueError

    #write code to calculate grade

except ValueError:
    print('Input out of range')
```

```
def divide(num,den):
    try:
        res = num / den
        return res
    except ZeroDivisionError:
        print('Divide Function : Divide by Zero Error')
        return 0

try :
    num = int ( input('Enter the numerator ') )
    den = int ( input('Enter the denominator ') )

    result = divide(num , den)
    print('Result = ' , result)

except ValueError:
    print('Main Block : Invalid Input ')
except ZeroDivisionError:
    print('Main Block : Divide by Zero error')

print('End of Program ')
```

Raising Exception from Except Block

user@varsity-OptiPlex-320: ~/sikander/python/exception

```
def divide(num,den) :  
    try:  
        res = num / den  
        return res  
    except ZeroDivisionError:  
        print('Divide Function : Divide by Zero Error')  
        raise  
  
try :  
    num = int ( input('Enter the numerator ' ) )  
    den = int ( input('Enter the denominator ' ) )  
  
    result = divide(num , den)  
    print('Result = ' , result)  
  
except ValueError:  
    print('Main Block : Invalid Input ' )  
except ZeroDivisionError:  
    print('Main Block : Divide by Zero error')  
  
print('End of Program ' )
```

finally

- To ensure some code runs no matter what errors occur, you can use a **finally** statement.
- The **finally** statement is placed at the bottom of a **try/except** statement.
- Code within a **finally** statement always runs after execution of the code in the **try**, and possibly in the **except**, blocks.

```
#Program to demonstrate finally blocks

try:

    a = int ( input('Enter the first number ' ) )
    b = int( input('Enter the second number ' ) )

    res = a / b
    print('result = ' , res)
except (ValueError,TypeError):
    print('Invalid Input Error')
except ZeroDivisionError:
    print('Divide by Zero Error')
finally:
    print('This code will run no matter what')

print('End of program')
```



```

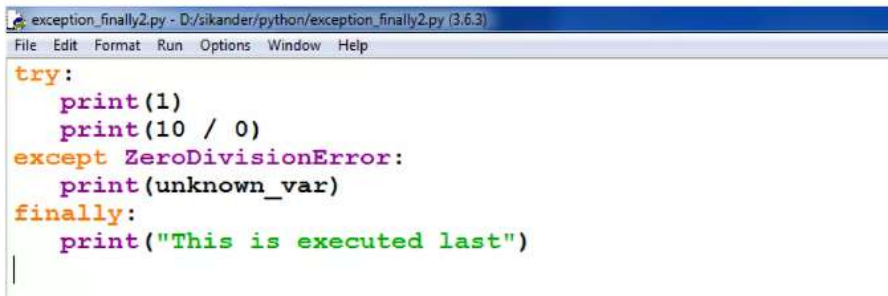
//
===== RESTART: D:/sikander/python/exception_finally.py =====
Enter the first number 4
Enter the second number 2
result = 2.0
This code will run no matter what
End of program
>>>

===== RESTART: D:/sikander/python/exception_finally.py =====
Enter the first number 4
Enter the second number a
Invalid Input Error
This code will run no matter what
End of program
>>>

===== RESTART: D:/sikander/python/exception_finally.py =====
Enter the first number 4
Enter the second number 0
Divide by Zero Error
This code will run no matter what
End of program
>>> |

```

- Code in a **finally** statement even runs if an uncaught exception occurs in one of the preceding blocks.



The screenshot shows a Python IDE window titled "exception_finally2.py - D:/sikander/python/exception_finally2.py (3.6.3)". The menu bar includes "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The code in the editor is as follows:

```
try:
    print(1)
    print(10 / 0)
except ZeroDivisionError:
    print(unknown_var)
finally:
    print("This is executed last")
|
```


Raising Exception

- Raising exception is similar to throwing exception in C++/Java.
- You can raise exceptions by using the **raise** statement

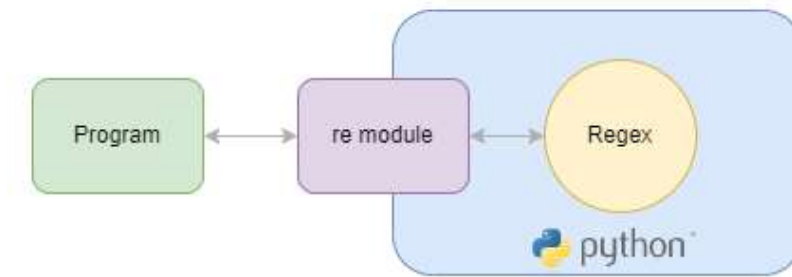
Introduction to the Python regular expressions

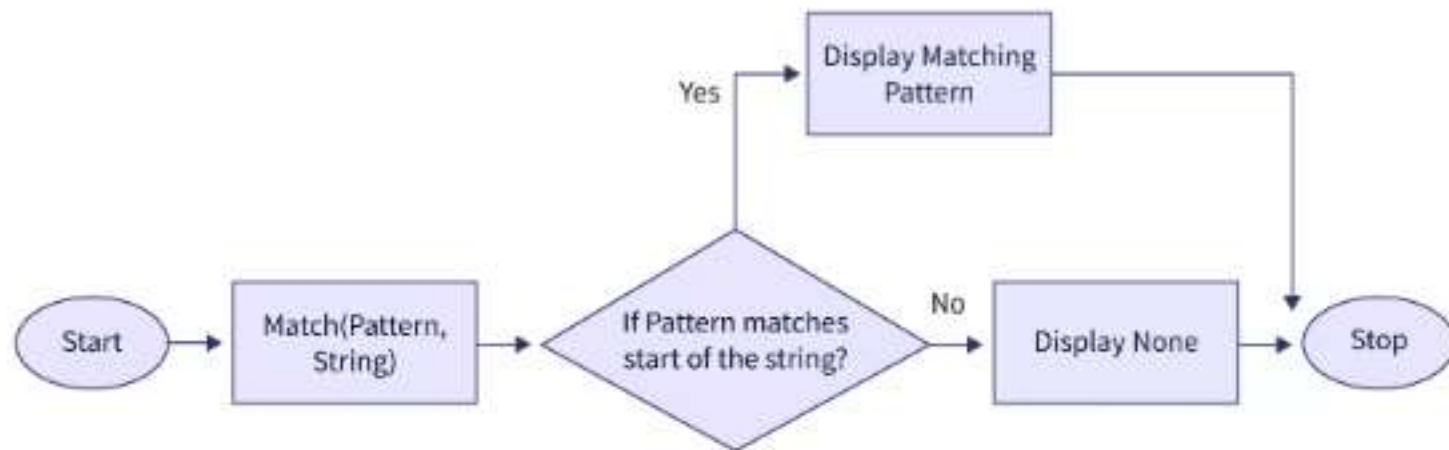
Introduction to the Python regular expressions

- A **Regular Expressions (RegEx)** is a special sequence of characters that uses a search pattern to find a string or set of strings.
- It can detect the presence or absence of a text by matching it with a particular pattern, and also can split a pattern into one or more sub-patterns.
- Python provides a **re** module that supports the use of regex in Python. Its primary function is to offer a search, where it takes a regular expression and a string. Here, it either returns the first match or else none.

Introduction to the Python regular expressions

- Regular expressions (called regex or regexp) specify search patterns. Typical examples of regular expressions are the patterns for matching email addresses, phone numbers, and credit card numbers.
- Regular expressions are essentially a specialized programming language embedded in Python. And you can interact with regular expressions via the built-in re module in Python.





Special Sequences

Special sequences do not match for the actual character in the string instead it tells the specific location in the search string where the match must occur. It makes it easier to write commonly used patterns.

Special Sequence	Description	Examples	
\A	Matches if the string begins with the given character	\Afor	for geeks
			for the world
\b	Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word.	\bge	geeks
			get
\B	It is the opposite of the \b i.e. the string should not start or end with the given regex.	\Bge	together
			forge
\d	Matches any decimal digit, this is equivalent to the set class [0-9]	\d	123
			gee1

\D	Matches any non-digit character, this is equivalent to the set class [^0-9]	\D	geeks
			geek1
\s	Matches any whitespace character.	\s	gee ks
			a bc a
\S	Matches any non-whitespace character	\S	a bd
			abcd
\w	Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].	\w	123
			geeKs4
\W	Matches any non-alphanumeric character.	\W	>\$
			gee<>
\Z	Matches if the string ends with the given regex	ab\Z	abcdab
			abababab

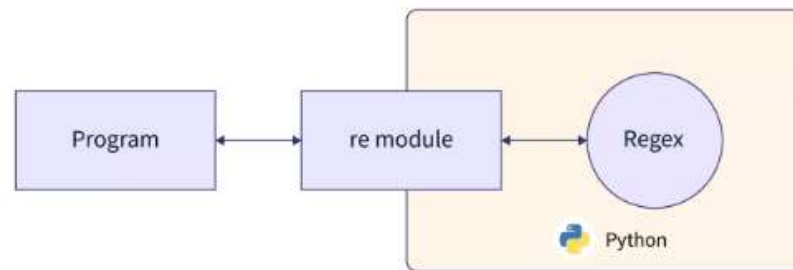
MetaCharacters

- To understand the RE analogy, MetaCharacters are useful, important, and will be used in functions of module re. Below is the list of metacharacters.

MetaCharacters	Description
\	Used to drop the special meaning of character following it
[]	Represent a character class
^	Matches the beginning
\$	Matches the end
.	Matches any character except newline
	Means OR (Matches with any of the characters separated by it.
?	Matches zero or one occurrence
*	Any number of occurrences (including 0 occurrences)
+	One or more occurrences
{}	Indicate the number of occurrences of a preceding regex to match.
()	Enclose a group of Regex

"re" Module in Python

- Regular expressions are a powerful language for matching text patterns. In python, we have a built-in package called "re" to work with regular expressions. The Python "re" module provides regular expression support.



- In Python a regular expression search is typically written as:

```
match = re.search(pattern, string)
```

Let us understand each of the terminologies used above in depth:

- **re.search()** : In general, the re.search() method takes a regular expression pattern and a string and **searches for that pattern within the string**. In case the search is successful, the search() method returns a "**match object**". Otherwise, it returns **None**. Hence, the search() method is usually followed by an if-else statement to confirm whether our search operation was successful or not.
- **Pattern** : The pattern usually depicts the pattern we are trying to search for in our string. There are multiple notations we have while mentioning the pattern. We will learn a few of them in this article. To learn about this in detail, you may refer [Regular Expression in Python](#).
- **String** : Here we pass the string in which we will try to find our pattern

```
import re
str = 'Hey! How are you?'
match = re.search(r'How are you', str)
# If-statement after search() tests if it succeeded
if match:
    print('found the matching word: ', match.group())
else:
    print('did not find any matching word!!')
```

Output:

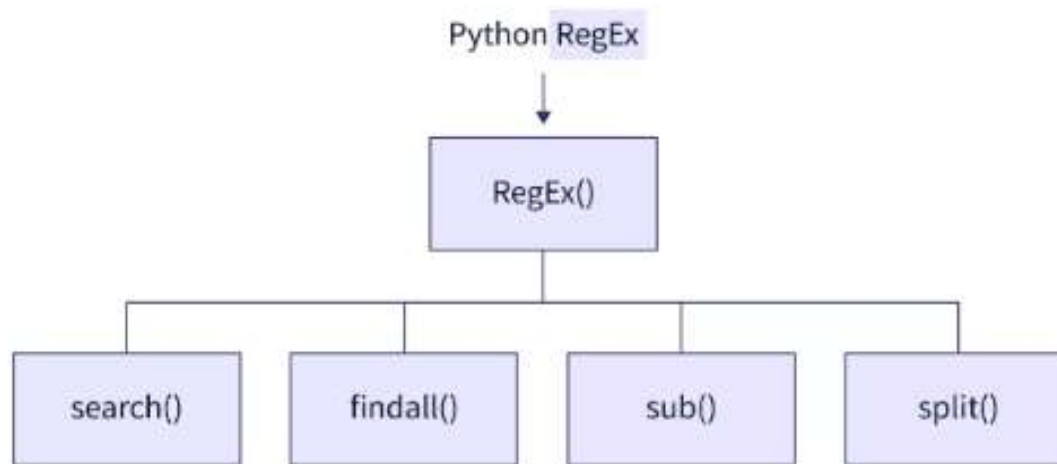
```
found the matching word:  How are you
```

```
import re
str = 'His Employee ID=A10D'
match = re.search(r'ID\=\w\d\d\w', str)
# If-statement after search() tests if it succeeded
if match:
    print('found the matching word: ', match.group())
else:
    print('did not find any matching word!!')
```

Output:

```
found the matching word: ID=A10D
```

Regular Expression Examples



re.findall()

The re.findall() method returns a list of strings containing all matches.

```
# Program to extract numbers from a string

import re

string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'

result = re.findall(pattern, string)
print(result)

# Output: ['12', '89', '34']
```


re.split()

The `re.split` method splits the string where there is a match and returns a list of strings where the splits have occurred.

```
import re

string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'

result = re.split(pattern, string)
print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']
```

If the pattern is not found, `re.split()` returns a list containing the original string.

re.sub()

The syntax of re.sub() is:

```
re.sub(pattern, replace, string)
```

The method returns a string where matched occurrences are replaced with the content of replace variable.

If the pattern is not found, re.sub() returns the original string.

```
# Program to remove all whitespaces
import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.sub(pattern, replace, string)
print(new_string)

# Output: abc12de23f456
```

re.search()

The `re.search()` method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, `re.search()` returns a match object; if not, it returns `None`.

```
match = re.search(pattern, str)
```

```
import re

string = "Python is fun"

# check if 'Python' is at the beginning
match = re.search('\APython', string)

if match:
    print("pattern found inside the string")
else:
    print("pattern not found")

# Output: pattern found inside the string
```