Multiple Function Arguments and Regular Expressions

Session 09

Session Overview

In this session, you will be able to:

- Explain the mechanism of passing arguments to Python functions
- Describe the use of `*args` and `**kwargs`
- Describe the significance of Regular Expressions
- Identify the different special characters used in Regular Expressions
- Explain how to extract and match using Regular Expressions

Argument Passing in Python (1-2)

- In a programming language, a developer passes an argument either by value or by reference.
- The most common method is to pass by value in which the compiler evaluates the caller's argument expression and binds its result to the corresponding variable in the function.
- If a variable forms this expression, a local copy of its value is in use.
- In other words, within the caller's scope, this variable shall remain intact when the function returns.

Argument Passing in Python (2-2)

Developers pass arguments by assignment, which has the following implications:

- Assignment occurs by allocating objects to local names.
- Allocating to argument names within a function has no impact on the caller.
- Altering a mutable object argument can affect the caller.

Use of *args (1-2)

Non-keyworded arguments:

```
>>>def function_two(one, two,
*multiple):
      print ("first: ", one)
      print ("second: ", two)
      print ("rest of arguments",
list(multiple))
```

Use of *args (2-2)

Calling the function given in Code Snippet A:

```
>>>function_two ("Sam", 28,
"Newyork", "Boston", "Texas")
```

Output:

```
first:  Sam
second:  28
rest of arguments ['Newyork', 'Boston',
'Texas']
```

Use of **kwargs

Handling a keyworded argument list:

Code Snippet:

```
>>> def get_details(**profile):
        if profile is not None:
            for key, value in
profile.items():
            print (key, ":", str(value))
>>> get_details(name = "Sam", age = 28, city =
"Newyork")
```

Output:

```
name : Sam
age : 28
city : Newyork
```

Using *args, **kwargs, and Formal Arguments

An example following order:

Code Snippet:

```
>>>def combined_func(one, two, *arg_list, **kw_arg_list):
        print ("Formal argument one: ",one)
        print ("Formal argument two: ",two)
        print ("Arbitrary Argument list: ",list(arg_list))
        print ("Keyword Argument list: ", dict(kw_arg_list))
>>>combined_func("Sam", 28, "Newyork", "Boson", height=167,
weight=89)
```

Output:

```
Formal argument one:  Sam
Formal argument two:  28
Arbitrary Argument list:  ['Newyork', 'Boson']
Keyword Argument list:  {'height': 167, 'weight': 89}
```

Defining Regular Expressions (RegEx)

- In Python code, it is a special string that represents a search pattern for fetching information from data in text format.
- Primarily, everything in the pattern is a character.
- A pattern helps in matching a particular sequence of characters (string).
- It can include letters, digits, special characters such as %!#$@ and punctuation.
- Regular expressions are tiny but highly specialized language patterns incorporated in Python.

RegEx Syntax

As regular expressions operate on strings, they are commonly in use for matching characters.

REs can have ordinary and special characters.

Ordinary characters such as 'B', 'b', or '2', are the simplest REs and they match to themselves.

However, a few characters do not match to themselves. They are called special metacharacters and are often a part of an RE.

They define how other RE sections should be interpreted.

Using Regular Expressions

Python compiles regular expressions into pattern objects.

These objects have methods for different operations such as substituting strings or searching for pattern matches.

Following is the syntax of compiling regular expressions in Python:

Syntax: `re.compile(pattern, flags)`

Extracting and Matching Regular Expressions

- A Python object denoting a compiled regular expression has many methods and attributes.
- Using these methods, developers can initiate the pattern matching process in Python.
- Typically, the match() or search() methods are useful here.
- They both take two arguments.
- The first argument is an expression, while the second argument is a string in which Python looks for the expression.

The `match()` Method (1-2)

How to use match() method:

Code Snippet:

```
>>> import re
>>> stList = ['sam sanfransico',
'sid santafe', 'max bos-ton']
>>> for string in stList:
        success =
re.match("(s\w+)\W(s\w+)",string)
```

The `match()` Method (2-2)

```
if success:
            print (success)
            print ("This set starts with s:",
success.groups())
```

Output:

```
<_sre.SRE_Match object; span=(0, 15), match='sam sanfransico'>
This set starts with s: ('sam', 'sanfransico')
<_sre.SRE_Match object; span=(0, 11), match='sid santafe'>
This set starts with s:('sid', 'santafe')
```

The `search()` Method

How to use search() method?

Code Snippet:

```
>>> import re
>>> text = 'Python tutorials are
helpful'
>>> patterns_to_search = ['python
tutorial', 'perl']
>>> for pattern in
patterns_to_search:
        print ("searching for ",
pattern, " in ", text)
        if re.search(pattern, text):
                print ("found it")
        else:
                print("not found")
```

Output:

```
searching for python
tutorial in Python
tutorials are helpful
found it
searching for perl in
Python tutorials are
helpful
not found
```

The `findall()` Method

How to use this findall() method:

Code Snippet:

```
>>> import re
>>> sample = 'sam@gmail.com,
dsyahoo.com, dk@ymail.com'
>>> mail_matches = re.findall(r'[\w\.-
]+@[\w\.-]+', sample)
>>> for mail in mail_matches :
      print ("found e-mail: ", mail)
```

```
found e-mail:
sam@gmail.com
found e-mail:
dk@ymail.com
```

The `split()` Method (1-2)

How to use split() method?

Code Snippet:

```
>>> re.split('\W+', 'Chars, chars, chars.')
>>> re.split('(\W+)', 'Chars, chars, chars.')
>>> re.split('\W+', 'Chars, chars, chars.', 1)
```

Output:

```
['Chars', 'chars', 'chars', '']
['Chars', ', ', 'chars', ', ', 'chars', '.', '']
['Chars', 'chars, chars.']
```

The `split()` Method (2-2)

```
>>> re.split('[a-f]+', '0a4F8',
flags=re.IGNORECASE)
>>> result=re.split(r'y','python analysis')
>>> result
>>> result = re.split(r'e', 'the charecter e
appears many times', 2)
>>> result
```

Output:

```
['0', '4', '8']
['p', 'thon anal', 'sis']
['th', ' char', 'cter e appears many times']
```

The `sub()` Method

How to use sub() method?

Code Snippet:

```
>>> result = re.sub(r'world','the
Universe','End of world')
>>> result
```

Output:

```
End of the Universe
```

MatchObject Methods

A developer can query a `MatchObject` instance for retrieving details about the matching string.

This is possible by employing methods of that instance.

The most important ones are `group()`, `span()`, `start()`, and `end()`.

Summary (1-5)

- Python implements pass by object or by object reference for passing arguments.
- Immutable arguments implement pass by value mode in Python.
- Mutable arguments implement pass by reference.
- The single asterisk syntax `(*args)` allows specifying a list of non-keyworded arguments of variable length.

Summary (2-5)

- The double asterisk syntax `(**kwargs)` allows passing a list of keyworded arguments of variable length. It handles named arguments.
- `*args` and `**kwargs` always come after formal arguments in a function header.
- A regular expression is a special string that represents a search pattern for fetching information from the targeted string or file data.

Summary (3-5)

- REs can have ordinary and special characters, of which the latter ones do not match themselves and are called metacharacters.
- Some of the most useful special characters are $, ., *, +, ?, |, (...), [ ], { }, and \.
- The re module offers functions for running regular expressions as strings.

Summary (4-5)

- Python compiles regular expressions into `RegexObject` instances that come with methods for operations such as substituting strings or searching for pattern matches.
- The `match()` method looks for a match only at the start of the string, while `search()` looks for it anywhere in the string.
- The `findall()` method allows iterating over the lines and returns a list of all matches at once.

Summary (5-5)

- The `split()` method divides a string by the pattern's occurrences.
- The `sub()` method searches for all occurrences of a pattern and replace them with a new sub string.
- Some of the most useful methods of a `MatchObject` instance are `group()`, `span()`, `start()`, and `end()`.