UNIT 4: REGULAR EXPRESSIONS

Regular Expressions in Python

Regular expressions are used to identify whether a pattern exists in a given sequence of characters (string) or not. They help in manipulating textual data, which is often a pre-requisite for data science projects that involve text mining. You must have come across some application of regular expressions: they are used at the server side to validate the format of email addresses or password during registration, used for parsing text data files to find, replace or delete certain string, etc.

In Python, regular expressions are supported by the re module. That means that if you want to start using them in your Python scripts, you have to import this module with the help of import.

“import re”

Basic Patterns: Ordinary Characters

You can easily tackle many basic patterns in Python using the ordinary characters. Ordinary characters are the simplest regular expressions. They match themselves exactly and do not have a special meaning in their regular expression syntax.

Examples are 'A', 'a', 'X', '5'.

Ordinary characters can be used to perform simple exact matches:

Example:

```
import re
pattern = r"Cookie"
sequence = "Cookie"
if re.match(pattern, sequence):
    print("Match!")
else:
    print("Not a match!")
```

output:

```
Match!
```

The match() function returns a match object if the text matches the pattern. Otherwise it returns None. The re module also contains several other functions.

The “r” at the start of the pattern Cookie is called a “raw string literal”. It changes how the string literal is interpreted. Such literals are stored as they appear.

Wild Card Characters: Special Characters

Notation(Symbol)	Description	Example Regex
.	Match *any character* (except \n)	b.b
^	Match start of a string	^Dear
$	Match end of string	Hello$
*	Match 0 or more occurrences of preceding regex	[A-Za-z0-9]*
+	Match 1 or more occurrences of	[a-z]+\.com

	preceding regex	
?	Match 0 or 1 occurrence of preceding regex	Goo?
{N}	Match N occurrence of preceding regex	[0-9]{3}
{M,N}	Match from M to N occurrence of preceding regex	[0-9]{5,9}
[…]	Match any single character from character class	[aeiou]
[..x-y..]	Match any single character in the range from x to y	[0-9],[A-Za-z]
[^…]	*Do not match* any character from character class, including any ranges, if present	[^aeiou]
re1 | re2	Match regular expressions re1 or re2	Foo | bar
\d	Match any decimal *digit*, same as [0-9] (\D is inverse of \d: do not match any numeric digit)	data\d+.txt
\w	Match any *alphanumeric* character, same as [A-Za-z0-9_] (\W is inverse of \w)	[A-Za-z_]\w+
\s	Match *any whitespace* character, same as [\n\t\r\v\f] (\S is inverse of \s)	of\sthe
\b	Match any *word boundary* (\B is inverse of \b)	\bThe\b

Groups and Grouping using Regular Expressions

Suppose that, when you're validating email addresses and want to check the user name and host separately.

This is when the group feature of regular expression comes in handy. It allows you to pick up parts of the matching text.

Parts of a regular expression pattern bounded by parenthesis() are called groups. The parenthesis does not change what the expression matches, but rather forms groups within the matched sequence. You have been using the group() function all along in this tutorial's examples. The plain match.group() without any argument is still the whole matched text as usual.

Greedy vs Non-Greedy Matching

When a special character matches as much of the search sequence (string) as possible, it is said to be a "Greedy Match". It is the normal behavior of a regular expression but sometimes this behavior is not desired:

```
pattern = "cookie"
sequence = "Cake and cookie"

heading  = r'<h1>TITLE</h1>'
re.match(r'<.*>', heading).group()
'<h1>TITLE</h1>'
```

The pattern <.*> matched the whole string, right up to the second occurrence of >.

However, if you only wanted to match the first <h1> tag, you could have used the greedy qualifier *? that matches as little text as possible.

Adding ? after the qualifier makes it perform the match in a non-greedy or minimal fashion; That is, as few characters as possible will be matched. When you run <.*>, you will only get a match with <h1>.

```
heading  = r'<h1>TITLE</h1>'
re.match(r'<.*?>', heading).group()
'<h1>'
```

The re Module: Core Functions and Methods

The re library in Python provides several functions that makes it a skill worth mastering.

Function/Method	**Description**
compile(pattern,flags=0)	Compile regex pattern with any optional flags and return a regex object
match(pattern,string, flags=0)	Attempt to match *pattern* to *string* with optional *flags*; return match object on success, None on failure
search(pattern,string, flags=0)	Search for first occurrence of *pattern* within *string* with optional *flags*; return match object on success, None on failure
findall(pattern,string[,flags])	Look for all (non-overlapping) occurrences of *pattern* in *string*; return a list of matches
sub(pattern, repl,string, count=0)	Replace all occurrences of the regex *pattern* in *string* with *repl*, substituting all occurrences unless *count* provided (see also subn(), which, in addition, returns the number of substitutions made)

Common Module Attributes (flags for most regex functions)

re.I, re.IGNORECASE	Case-insensitive matching
re.L, re.LOCALE	Matches via \w, \W, \b, \B, \s, \S depends on locale
re.M, re.MULTILINE	Respectively causes ^ and $ to match the beginning and end of each line in target string rather than strictly the beginning and end of the

	entire string itself
re.S, re.DOTALL	The . normally matches any single character except \n; this flag says . should match them, too
re.X, re.VERBOSE	All whitespace plus # (and all text after it on a single line) are ignored unless in a character class or backslash-escaped, allowing comments and improving readability

search(pattern, string, flags=0)

With this function, you scan through the given string/sequence looking for the first location where the regular expression produces a match. It returns a corresponding match object if found, else returns None if no position in the string matches the pattern. Note that None is different from finding a zero-length match at some point in the string.

```
pattern = "cookie"
sequence = "Cake and cookie"

re.search(pattern, sequence).group()
'cookie'
```

match(pattern, string, flags=0)

Returns a corresponding match object if zero or more characters at the beginning of string match the pattern. Else it returns None, if the string does not match the given pattern.

```
pattern = "C"
sequence1 = "IceCream"

# No match since "C" is not at the start of "IceCream"
re.match(pattern, sequence1)
sequence2 = "Cake"

re.match(pattern,sequence2).group()
'C'
```

search() versus match()

The match() function checks for a match only at the beginning of the string (by default) whereas the search() function checks for a match anywhere in the string.

findall(pattern, string, flags=0)

Finds all the possible matches in the entire sequence and returns them as a list of strings. Each returned string represents one match.

```
email_address = "Please contact us at: support@datacamp.com, xyz@datacamp.com"

#'addresses' is a list that stores all the possible match
```

```
addresses = re.findall(r'[\w\.-]+@[\w\.-]+', email_address)
for address in addresses:
    print(address)
support@datacamp.com
xyz@datacamp.com
```

sub(pattern, repl, string, count=0, flags=0)

This is the substitute function. It returns the string obtained by replacing or substituting the leftmost non-overlapping occurrences of pattern in string by the replacement repl. If the pattern is not found then the string is returned unchanged.

```
email_address = "Please contact us at: xyz@datacamp.com"
new_email_address = re.sub(r'([\w\.-]+)@([\w\.-]+)', r'support@datacamp.com', email_address)
print(new_email_address)
Please contact us at: support@datacamp.com
```

compile(pattern, flags=0)

Compiles a regular expression pattern into a regular expression object. When you need to use an expression several times in a single program, using the compile() function to save the resulting regular expression object for reuse is more efficient. This is because the compiled versions of the most recent patterns passed to compile() and the module-level matching functions are cached.

```
pattern = re.compile(r"cookie")
sequence = "Cake and cookie"
pattern.search(sequence).group()
'cookie'
# This is equivalent to:
re.search(pattern, sequence).group()
'cookie'
```

Text Processing

What Is a CSV File?

A CSV file (Comma Separated Values file) is a type of plain text file that uses specific structuring to arrange tabular data. Because it's a plain text file, it can contain only actual text data—in other words, printable ASCII or Unicode characters.

The structure of a CSV file is given away by its name. Normally, CSV files use a comma to separate each specific data value. Here's what that structure looks like:

```
column 1 name,column 2 name, column 3 name
first row data 1,first row data 2,first row data 3
second row data 1,second row data 2,second row data 3
...
```

Notice how each piece of data is separated by a comma. Normally, the first line identifies each piece of data—in other words, the name of a data column. Every subsequent line after that is actual data and is limited only by file size constraints.
In general, the separator character is called a delimiter, and the comma is not the only one used. Other popular delimiters include the tab (\t), colon (:) and semi-colon (;) characters. Properly parsing a CSV file requires us to know which delimiter is being used.

Parsing CSV Files With Python’s Built-in CSV Library

The csv library provides functionality to both read from and write to CSV files.

Reading CSV Files With csv

Reading from a CSV file is done using the reader object. The CSV file is opened as a text file with Python’s built-in open() function, which returns a file object. This is then passed to the reader, which does the heavy lifting.

Here’s the employee_birthday.txt file:

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here’s code to read it:

```
import csv

f=open('employee_birthday.txt')
csv_reader = csv.reader(f, delimiter=',')
line_count = 0
for row in csv_reader:
    if line_count == 0:
        print(f'Column names are {", ".join(row)}')
        line_count += 1
    else:
        print(f'\t{row[0]} works in the {row[1]} department, and was born in {row[2]}.')
        line_count += 1
print(f'Processed {line_count} lines.')
```

This results in the following output:

```
Column names are name, department, birthday month
    John Smith works in the Accounting department, and was born in November.
    Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.
```

Each row returned by the reader is a list of String elements containing the data found by removing the delimiters. The first row returned contains the column names, which is handled in a special way.

Reading CSV Files Into a Dictionary With csv

Rather than deal with a list of individual String elements, you can read CSV data directly into a dictionary as well.

Again, our input file, employee_birthday.txt is as follows:

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here’s the code to read it in as a dictionary this time:

```
import csv

f=open('employee_birthday.txt')
csv_reader = csv.DictReader(f)
line_count = 0
for row in csv_reader:
      if line_count == 0:
          print(f'Column names are {", ".join(row)}')
          line_count += 1
      print(f'\t{row["name"]} works in the {row["department"]} department, and was born in
{row["birthday month"]}.')
      line_count += 1
 print(f'Processed {line_count} lines.')
```

This results in the same output as before:

```
Column names are name, department, birthday month
    John Smith works in the Accounting department, and was born in November.
    Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.
```

Where did the dictionary keys come from? The first line of the CSV file is assumed to contain the keys to use to build the dictionary. If you don’t have these in your CSV file, you should specify your own keys by setting the fieldnames optional parameter to a list containing them.

Writing CSV Files With csv

You can also write to a CSV file using a writer object and the .write_row() method:

```
import csv

with open('employee_file.csv', mode='w') as employee_file:
    employee_writer = csv.writer(employee_file, delimiter=',', quotechar=’”’,
quoting=csv.QUOTE_MINIMAL)

    employee_writer.writerow(['John Smith', 'Accounting', 'November'])
    employee_writer.writerow(['Erica Meyers', 'IT', 'March'])
```

The quotechar optional parameter tells the writer which character to use to quote fields when writing. Whether quoting is used or not, however, is determined by the quotingoptional parameter:

- If quoting is set to csv.QUOTE_MINIMAL, then .writerow() will quote fields only if they contain the delimiter or the quotechar. This is the default case.
- If quoting is set to csv.QUOTE_ALL, then .writerow() will quote all fields.
- If quoting is set to csv.QUOTE_NONNUMERIC, then .writerow() will quote all fields containing text data and convert all numeric fields to the float data type.
- If quoting is set to csv.QUOTE_NONE, then .writerow() will escape delimiters instead of quoting them. In this case, you also must provide a value for the escapechar optional parameter.

Reading the file back in plain text shows that the file is created as follows:

```
John Smith,Accounting,November
Erica Meyers,IT,March
```

Writing CSV File From a Dictionary With csv

Since you can read our data into a dictionary, it’s only fair that you should be able to write it out from a dictionary as well:

```
import csv

with open('employee_file2.csv', mode='w') as csv_file:
    fieldnames = ['emp_name', 'dept', 'birth_month']
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'emp_name': 'John Smith', 'dept': 'Accounting', 'birth_month': 'November'})
    writer.writerow({'emp_name': 'Erica Meyers', 'dept': 'IT', 'birth_month': 'March'})
```

Unlike DictReader, the fieldnames parameter is required when writing a dictionary. This makes sense, when you think about it: without a list of fieldnames, the DictWriter can’t know which keys to use to retrieve values from your dictionaries. It also uses the keys in fieldnames to write out the first row as column names.

The code above generates the following output file:

```
emp_name,dept,birth_month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Reading and Writing JSON in Python

JSON (JavaScript Object Notation) is language-neutral data interchange format. JSON is a text-based format which is derived from JavaScript object syntax. JSON is commonly used by web applications to transfer data between client and server. If you are using a web service then there are good chances that data will be returned to you in JSON format, by default.

Serialization and Deserialization

Serialization: The process of converting an object into a special format which is suitable for transmitting over the network or storing in file or database is called **Serialization**.

Deserialization: It is the reverse of serialization. It converts the special format returned by the serialization back into a usable object.

In the case of JSON, when we serializing objects, we essentially convert a Python object into a JSON string and deserialization builds up the Python object from its JSON string representation.

Python provides a built-in module called json for serializing and deserializing objects.

The json module mainly provides the following functions for serializing and deserializing.

1. dump(obj, fileobj)
2. dumps(obj)
3. load(fileobj)
4. loads(s)

Serializing with dump()

The dump() function is used to serialize data. It takes a Python object, serializes it and writes the output (which is a JSON string) to a file like object.

The syntax of dump() function is as follows:

Syntax: dump(obj, fp)

ARGUMENT	DESCRIPTION
obj	Object to be serialized.
Fp	A file-like object where the serialized data will be written.

Writing JSON to a File

The easiest way to write your data in the JSON format to a file using Python is to use store your data in a dict object, which can contain other nested dicts, arrays, booleans, or other primitive types like integers and strings. You can find a more detailed list of data types supported here.

The built-in json package has the magic code that transforms your Python dict object in to the serialized JSON string.

```
import json

data = {}
data['people'] = []
data['people'].append({
    'name': 'Scott',
    'website': 'example.com',
    'from': 'America'
})
data['people'].append({
    'name': 'Larry',
    'website': 'google.com',
    'from': 'NewZealand'
})
data['people'].append({
    'name': 'Tim',
    'website': 'apple.com',
    'from': 'Africa'
})

with open('data.txt', 'w') as outfile:
    json.dump(data, outfile)
```

Deserializing with load()

The load() function deserializes the JSON object from the file like object and returns it.
Its syntax is as follows:
load(fp) -> a Python object

ARGUMENT	**DESCRIPTION**
fp	A file-like object from where the JSON string will be read

Reading JSON from a File

On the other end, reading JSON data from a file is just as easy as writing it to a file. Using the same json package again, we can extract and parse the JSON string directly from a file object. In the following example, we do just that and then print out the data we got:

```
import json

with open('data.txt') as json_file:
    data = json.load(json_file)
    for p in data['people']:
        print('Name: ' + p['name'])
        print('Website: ' + p['website'])
        print('From: ' + p['from'])
        print('')
```

Serializing and Deserializing with dumps() **and** loads()

The dumps() function works exactly like dump() but instead of sending the output to a file-like object, it returns the output as a string.
Similarly, loads() function is as same as load() but instead of deserializing the JSON string from a file, it deserializes from a string.

```
import json
data = {'people':[{'name': 'Scott', 'website': 'stackabuse.com', 'from': 'Nebraska'}]}
a=json.dumps(data,indent=4)
print(a)
person=json.loads(a)
print(person)
```

output:

```
{
    "people": [
        {
            "name": "Scott",
            "website": "stackabuse.com",
            "from": "Nebraska"
        }
    ]
}
{'people': [{'name': 'Scott', 'website': 'stackabuse.com', 'from': 'Nebraska'}]}
```

Reading and Writing XML in Python

What is XML?

XML stands for "Extensible Markup Language". It is mainly used in webpages, where the data has a specific structure and is understood dynamically by the XML framework.

XML creates a tree-like structure that is easy to interpret and supports a hierarchy. Whenever a page follows XML, it can be called an XML document.

- XML documents have sections, called *elements*, defined by a beginning and an ending *tag*. A tag is a markup construct that begins with < and ends with >. The characters between the start-tag and end-tag, if there are any, are the element's content. Elements can contain markup, including other elements, which are called "child elements".
- The largest, top-level element is called the *root*, which contains all other elements.
- Attributes are name–value pair that exist within a start-tag or empty-element tag. An XML attribute can only have a single value and each attribute can appear at most once on each element.

Introduction to ElementTree

The XML tree structure makes navigation, modification, and removal relatively simple programmatically. Python has a built-in library, ElementTree, that has functions to read and manipulate XMLs (and other similarly structured files).

First, import ElementTree. It's a common practice to use the alias of ET:

Syntax: import xml.etree.ElementTree as ET

Creating XML file

In this example, we will create a new XML file with an element and a sub-element. Let’s get started straight away:

```
import xml.etree.ElementTree as xml

def createXML(filename):
    # Start with the root element
    root = xml.Element("users")
    child1 = xml.Element("child_user")
    root.append(child1)

    tree = xml.ElementTree(root)
    with open(filename, "wb") as fh:
        tree.write(fh)

createXML("test.xml")
```

Once we run this script, a new file will be created in the same directory with file named as test.xml with following contents:

```
<users><child_user /></users>
```

There are two things to notice here:

- While writing the file, we used wb mode instead of w as we need to write the file in binary mode.
- The child user tag is a self-closing tag as we haven’t put any sub-elements in it.

Adding values to XML elements

Let’s improve the program by adding values to the XML elements:

```
import xml.etree.ElementTree as xml

def createXML(filename):
    # Start with the root element
    root = xml.Element("users")
    child1 = xml.Element("child_user")
    root.append(child1)

    userId1 = xml.SubElement(child1, "id")
    userId1.text = "123"

    userName1 = xml.SubElement(child1, "name")
    userName1.text = "John"

    tree = xml.ElementTree(root)
    with open(filename, "wb") as fh:
        tree.write(fh)

createXML("test.xml")
```

Once we run this script, we will see that new elements are added added with values. Here are the content of the file:

```
<users>
    <child_user>
        <id>123</id>
        <name>John</name>
    </child_user>
```

```
</users>
```

Editing XML data

We will use the same XML file we showed above. We just added some more data into it as:

```
<users>
   < child_user>
      <id>123</id>
      <name>John</name>
      <salary>0</salary>
   </ child_user>
   < child_user>
      <id>234</id>
      <name>Pankaj</name>
      <salary>0</salary>
   </ child_user>
   < child_user>
      <id>345</id>
      <name>Dev</name>
      <salary>0</salary>
   </ child_user>
</users>
```

Let’s try and update salaries of each user:

```
import xml.etree.ElementTree as xml

def updateXML(filename):
   # Start with the root element
   tree = xml.ElementTree(file=filename)
   root = tree.getroot()

   for salary in root.iter("salary"):
      salary.text = '1000'

   tree = xml.ElementTree(root)
   with open("updated_test.xml", "wb") as fh:
      tree.write(fh)

updateXML("test.xml")
```

Python XML Parser Example(Read XML file)

This time, let’s try to parse the XML data present in the file and print the data:

<u>Example1:</u>

```
import xml.etree.cElementTree as xml

def parseXML(file_name):
    # Parse XML with ElementTree
    tree = xml.ElementTree(file=file_name)
    root = tree.getroot()

    # get the information via the children!
    print("-" * 40)
    print("Iterating using getchildren()")
    print("-" * 40)
    #users = root.getchildren()
#getchildren() generates warning. So use the following syntax
     users=list(root)
    for user in users:
        user_children = list(user)
        for user_child in user_children:
            print(user_child.tag  + "=" + user_child.text)

parseXML("test.xml")
```

Example2:

```
from xml.etree import ElementTree
tree = ElementTree.parse("test1.xml")
root = tree.getroot()

children = list(root)
for child in children:
    ElementTree.dump(child)
```