Regular Expressions

^Reg[a-z]*Ex[a-z]*$

1. INTRODUCTION

What are Regular Expressions?

Introduction

Regular expressions are a compact way of searching for text in a document. So let's imagine we have a document that has millions of lines of text in it, including a number of different email addresses, and we want to extract those addresses from the document. Unfortunately, the addresses come in different formats, including:

- `DamianTGordon@MyMail.com`
- `Damian.Gordon@MyMail.com`
- `Damian.T.Gordon@MyMail.co.uk`

So they all have the "@" symbol in their text, and they also have zero, one, or more full stops (".") before the "@" symbol. They also have at least one, but maybe more full stops after the "@" symbol.

To add to the complexity, there are also other phrases in the document that look like emails but aren't, that we don't want to extract, including incorrect email addresses:

- `DamianTGordon@MyMail` (No domain name, e.g. `.COM`)
- `@MyMail.com` (No name before the @ symbol)

As well as some typical text that looks like an email address:

- `Can we meet@2pm?`
- `We are meeting@boardroom.`

So a regular expression is a special code we can use to describe the rules of what defines a valid email address (and we know there's a few accepted email address formats), and also how to recognise something that isn't a valid address. The good news is … the regular expression below describes a valid email address according to the rules we stated above, and I know it looks complicated, but don't worry we'll be explaining each part of this code over the following weeks:

```
^[A-Za-z+.]+@[A-Za-z.]+[A-Za-z]+$
```

This code should work in almost any programming language, but it is worth noting that different programming languages implement regular expressions in slightly different ways, so a regular expression that works in one language might not always work exactly the same way in another, but we'll specify them to work in as many languages as possible, and we'll note when different languages behave differently.

Some Terminology

A Regular Expression is often called a ***RegEx***, and a collection of Regular Expressions are called ***RegExes***. There are two types of characters used in regular expressions:

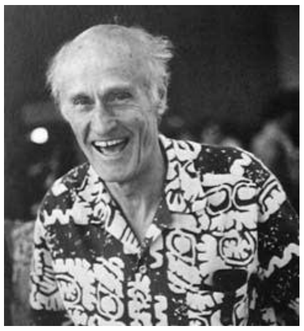
- **Metacharacters**: These are characters that have a special meaning.
- **Literal characters**: These are the actual characters we want to match.

History of Regular Expressions

History of Regular Expressions

In 1943, Warren McCulloch (an American cybernetician) and Walter Pitts (an American logician) developed a computer-based model of learning, modelled on how human brains learn. These types of models are generally referred to as artificial neural networks (ANNs), and their specific model is called a McCulloch-Pitts neural network, whose goal was to learn and recognise patterns. In 1951, American mathematician Stephen Cole Kleene created a formal mathematical language to describe these neural networks, and this language eventually evolved into a general pattern matching notation, which we know as Regular Expressions. This notation can be used to explore the structure of any type of text-based patterns.

BIOGRAPHY: Stephen Cole Kleene



Stephen Cole Kleene was born on January 5th, 1909 in Hartford, Connecticut, and died on January 25th, 1994 in Madison, Wisconsin. He is a notable mathematician who helped develop some of the foundations of theoretical computer science (often together with his thesis supervisor, Alonzo Church). He is a founder of the branch of mathematical logic known as recursion theory, and as we know, he invented Regular Expressions in 1951 to describe the McCulloch-Pitts neural net.

The Church–Turing thesis

In the 1930s, Alonzo Church (Kleene's dissertation supervisor) developed a general system of logic, that he called Lambda calculus (λ-calculus), to explore the limits of what it is possible to calculate. At around the same time, British mathematician, Alan Mathison Turing, was working on the same problem using his computation model, that later became known as the "Turing Machine", and he identified an approach that would also help explore the limits of what can be calculated, or computed. In 1952, Stephen Cole Kleene published "Introduction to Metamathematics" where he showed that Lambda calculus and Turing Machines are strictly equivalent, and that they are also equivalent to a third approach by Kurt Gödel called "Recursive Functions". He called this the "Church–Turing" thesis, and it serves as a foundational principle in computer science and helps to establish the limits of computability.

The Church–Turing thesis is part of the broader theory of Computation, which looks at the general question of whether a problem can be solved by a computer; and if it can be solved, is it an approximate solution or a very precise one? We can express these questions in a mathematical language, and when looking at problems that examine pattern matching, we can use Regular Expressions to represent them.

Software that uses Regular Expressions

RegEx in Software Tools

One of the first occurrences of the incorporation of regular expressions in software tools was in 1968, when Ken Thompson built Kleene's regex notation into the text editor ***QED*** that ran on an operating system called ***CTSS*** (***Compatible Time-Sharing System***), which allowed users to search for, and replace, specific formats of text.

Thompson went on to co-develop the ***Unix*** operating system in 1969, which proved to be a very popular operating system for many years. He and others incorporated regexes into many software tools, including text editors such as ***ed***, ***sed***, ***vi*** and ***Emacs***. He also developed other tools that uses regexes such as ***grep***, which can be used to search files for text that matches a particular pattern, and this led to the development of ***AWK***, which is a scripting language for extracting data from files.

BIOGRAPHY: Ken Thompson

Ken Thompson was born on February 4th, 1943, in New Orleans, Louisiana. He is a notable Computer Scientist who helped design and implement the original Unix operating system. He also invented a number of programming languages (including B, Bon, and Go), operating systems (Unix, Plan 9, and Inferno), and utilities such as QED, ed, and grep.

By the 1980s regular expressions were incorporated into a series of formal standards including work by the International Organization for Standardization to create the ***ISO SGML*** standard. And in 1992 they were incorporated into a standard for operating systems by the IEEE Computer Society in their ***POSIX*** family of standards.

In 1996, the ***PostgreSQL*** RDBMS (relational database management system) was developed as a free and open-source RDBMS emphasizing extensibility and its compliance with regular SQL. It incorporates regexes as part of its key features.

In the late 2010s, computer companies started to offer hardware implementations of regexes, including ones that are incorporated into ***FPGAs*** (***Field-Programmable Gate Arrays***) and ***GPUs*** (***Graphics Processing Units***).

RegEx in Computer Programming Languages

We can use regexes in many programming languages, and their functions are either built directly into the programming languages (like in ***Perl*** and ***ECMAScript***) or they can be included into the language by using a software library (which is a collection of additional features that can be optionally included into a programming language to extend the functionality of the language), in languages like ***C***, ***C++***, ***Java***, and ***Python***.

1. INTRODUCTION

Regular Expressions in Python

Introduction

The Python programming language was developed by Guido van Rossum starting in 1989, and the first version was released in 1991. It is one of the most widely used and popular programming languages, and is considered one of the easiest programming languages to read, because it uses indentation to show blocks of code.

RegExes in Python

Python doesn't have regular expressions built into the programming language, but it does have a library (or package) called `re` that can be imported into Python programs, and that gives those programs a range of RegEx functions.

The program below begins with stating the program name `RegEx_email`, and next it imports the regular expression package (`import re`). Following that the specific regular expression is declared, as well as the text we are going to test it against (these are `RegEx_Pattern` and `Test_Message` respectively). The pattern is then compared with the specific regular expression using `re.match` function, and since they will match each other, this program will print out the following message: `The pattern matches.`

```
# PROGRAM RegEx_email:

import re

RegEx_Pattern = "^[A-Za-z+.]+@[A-Za-z.]+[A-Za-z]+$"
Test_Message = "Damian.T.Gordon@mymail.com"

if (re.match(RegEx_Pattern, Test_Message)):
#then
    print("The pattern matches")
else:
    print("The pattern does not match")
# ENDIF;


# END.
```

Other Functions in the `re` Package

Importing the `re` package means a range of functions related to regular expressions are available to the program. We've seen one function already, `re.match()`, which compares a regular expression with some text. Two other important functions are:

- `re.search()`: This will search some text for a pattern, and it will return the first occurrence of that pattern within the text.
- `re.findall()`: This will find all occurrences of a pattern in some text.

1. INTRODUCTION

Regular Expressions in Java

Introduction

The Java programming language was developed by James Gosling, Mike Sheridan, and Patrick Naughton starting in 1991, and the first version was released in 1996. It is one of the most widely used and popular programming languages, and was designed with a C/C++-style syntax that programmers would be familiar with.

RegExes in Java

Java doesn't have regular expressions built into the programming language, but it does have a library (or package) called `java.util.regex` that can be imported into Java programs, and that gives those programs a range of RegEx functions.

The program below is similar to the Python program we have seen already, with a few differences; it begins by importing two classes from the `java.util.regex` package (`java.util.regex.Pattern` and `java.util.regex.Matcher`), those are `Pattern` and `Matcher`. Next the Main class is declared, as is the Main method. The regular expression is specified using the `Pattern` class, and the test text is specified using the `Matcher` class. The test text is compared with the specific regular expression using the `Test_Message.find()` function, and since the two match each other, this program will print out the following message:
`The pattern matches.`

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Main {
  public static void main(String[] args) {


  Pattern RegEx_Pattern =
     Pattern.compile("^[A-Za-z+.]+@[A-Za-z.]+[A-Za-z]+$");
  Matcher Test_Message =
     RegEx_Pattern.matcher("Damian.T.Gordon@mymail.com");


  if(Test_Message.find()) {
      System.out.println("The pattern matches");
   } else {
      System.out.println("The pattern does not match");
   }
  }
}
```

The `java.util.regex` package has over 4000 classes in it, so we won't cover them all here, but we'll see some more of them over the coming weeks.

2. SIMPLE MATCHING

Strings and Things

Introduction

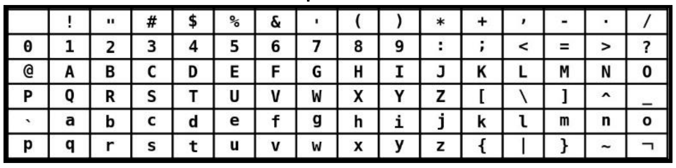
In computers we often use special terminology to represent fairly common things, the goal of this isn’t to make things more complicated at all, but instead it is to be very precise about what we are talking about (because when dealing with computers we know that they want us to be very precise about what we say).

Character

A ***character*** is any single letter, number, or symbol. They are often enclosed in single quotation marks (‘). Examples of characters include any of the following:

```
Test_Message  =  ‘H’
Test_Message  =  ‘X’
Test_Message  =  ‘d’
Test_Message  =  ‘j’
Test_Message  =  ‘4’
Test_Message  =  ‘8’
Test_Message  =  ‘@’
Test_Message  =  ‘%’
```

Further Examples of Characters:

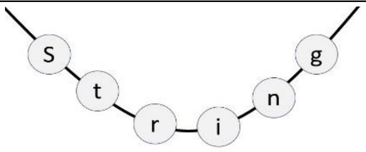


	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	¬

String

The term ***string*** is used to describe text, because text is made up of a chain of characters (like a string of pearls, where each pearl is a character). They are often enclosed in double quotes (“). Examples of Strings include any of the following:

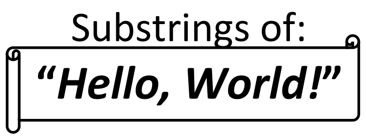
```
Test_Message  =  “Hello, World!”
Test_Message  =  “The rain in Spain”
Test_Message  =  “This answer is 42”
Test_Message  =  “3.14159265359”
Test_Message  =  “@£$%%^$^£$%$”
```



Substring

If we are referring to part of a specific string, we call it a ***substring***. They are often enclosed in double quotes (“). If the String is “*<u>Hello, World!</u>*”, then the following are examples of Substrings of that string:

```
Test_Message  =  “Hello”
Test_Message  =  “World!”
Test_Message  =  “llo”
Test_Message  =  “Wor”
Test_Message  =  “llo, Wor”
```



Reminder

We will remember that the key purpose of regular expressions is to create a pattern to locate a specific string, or to locate a collection of strings with a specific format (as we saw with the email example on Page 1). And we can use them for things like searching and replacing text in a Word Processing editor, and validating fields in an online form. We will also learn about other uses for regular expressions, including things such as Web Scraping and Data Mining, that we will discuss in the future.

Using the Period to Match

Introduction

This will be our first use of special characters ("metacharacters") to match a String with a regular expression pattern, and the special character we are going to look at is the period (".") character, or the *Full Stop* character. ***The period metacharacter represents any other single character, which can be a letter, number, or symbol***.

A Simple Example

Sometimes I get emails from people who spell my name wrong, and they call me "Damien" instead of "Damian", which isn't a big deal, but if I had a big document full of emails to me, and I wanted to check how often my name is used in emails, I'd have to search for the string "Damian" and the string "Damien". Regular Expressions give us a nice, simple way of doing this using the period character. We could create a Regular Expression using the period metacharacter as follows:

```
RegEx_Pattern = "Dami.n"
```

So, we can read this Regular Expression as to try to match Strings with the following pattern: "D", "a", "m", "I", any character, and "n".

So we would get the following matches:

`Test_Message = "Damian"`	MATCH ✓
`Test_Message = "Damien"`	MATCH ✓

And it won't match with names that don't fit the pattern:

`Test_Message = "Damani"`	NO MATCH ✗
`Test_Message = "Dameon"`	NO MATCH ✗

But, it is worth noting, that the pattern will match many other Strings, including:

`Test_Message = "Damixn"`	MATCH ✓
`Test_Message = "Dami5n"`	MATCH ✓
`Test_Message = "Dami=n"`	MATCH ✓
`Test_Message = "Dami n"`	MATCH ✓

Since the period can represent any character, including the space character (" ").

We should also remember, our pattern is six characters long ("D", "a", "m", "I", any character, and "n"), so the following Strings will not be considered a match:

`Test_Message = "Damin"`	NO MATCH ✗
`Test_Message = "Damiaan"`	NO MATCH ✗

As they don't follow the pattern.

More Details on the Period

Introduction

We will remember that the period metacharacter represents any other single character, which can be a letter, number, or symbol. Now let's look at more advanced uses of the period when searching for patterns in a String (i.e. in a text).

More than One Period

Sometimes when people are emailing me, instead of "Damian" they type "Damain", so they swap around the "I" and the "a". An easy way to describe this is as follows:

```
RegEx_Pattern = "Dam..n"
```

So, we can read this Regular Expression as to search for a String with the following pattern: "D", "a", "m", any character, any character, and "n". So we would get the following matches:

`Test_Message  = "Damian"`	MATCH ✓
`Test_Message  = "Damain"`	MATCH ✓

However, we know this will also match other Strings including any of the following:

`Test_Message  = "Damxxn"`	MATCH ✓
`Test_Message  = "Dam35n"`	MATCH ✓
`Test_Message  = "Dam&=n"`	MATCH ✓
`Test_Message  = "Dam  n"`	MATCH ✓

So, we might match to more things than we want.

But it will not match to these (as the pattern must be six characters long):

`Test_Message  = "Damn"`	NO MATCH ✗
`Test_Message  = "Dami an"`	NO MATCH ✗

Matching the Period Character

Finding the actual period character (not the metacharacter) in a String is a bit tricky. So, if we wanted to find something easy like the letter "D" in a String, we could simply set the Regular Expression to be equal to the literal character "D" as follows:

```
RegEx_Pattern = "D"
```

However, if we wanted to find the period character ".", the following won't work:

```
RegEx_Pattern = "."   ✗ ✗ ✗
```

As it would match any character in the String. So, we use a specific code to represent the actual period character. Mathematically we can represent it as follows: `\.`
However, most programming languages typically prefer we state it as follows:

```
RegEx_Pattern = "\\."
```

And this will allow us to locate a period character (".") in a String, e.g. in an email.

Using the Vertical Bar to Match

Introduction

Our next special character or "metacharacter" is the vertical bar ("|"). The vertical bar allows us to choose from a list of alternatives, and is thus also called the "alternation" operator, and also the "Logical OR" operator. ***The vertical bar metacharacter is used with a list of alternatives, from which any will be a match***.

Why Use the Vertical Bar?

We will remember that sometimes when people are emailing me, instead of "Damian" they type "Damien", and that we can use the period metacharacter to match both of these, using the following Regular Expression:

```
RegEx_Pattern = "Dami.n"
```

However, this is an *approximate solution*, as we know this will also match other variations, including: "Damisn", "Dami7n", "Dami@n", "Dami n", and many others. If we want a *precise solution*, that will match only to the Strings "Damian" or "Damien", we can use the vertical bar with the following Regular Expression:

```
RegEx_Pattern = "Damian|Damien"
```

This Regular Expression matches with EITHER "Damian" OR "Damien".

My friend Rebecca has a lot more trouble than I do, in her emails people address her as "Rebecca", "Becca", "Becks", and "Becky", so if she asked us to search for her name, we could use the following Regular Expression:

```
RegEx_Pattern = "Rebecca|Becca|Becks|Becky"
```

This reads as EITHER "Rebecca" OR "Becca" OR "Becks" OR "Becky".

Combining Metacharacters

We notice that two of Rebecca's nicknames are very similar: "Becks" and "Becky". So we could use the Period metacharacter to represent these using the following Regular Expression: "`Beck.`", with the Vertical Bar for the other alternatives:

```
RegEx_Pattern = "Rebecca|Becca|Beck."
```

And this is a good approximate solution that will match any of the required names, but will match a few others as well, like "Beckr" or "Becko". We could use an even shorter Regular Expression: "`Bec..`" to cover three alternatives:

```
RegEx_Pattern = "Rebecca|Bec.."
```

And this will match any of the required names, but also match lots of other Strings like "Becss", "Bechg" and "Becdh", but it's a more compact Regular Expression.

Matching the Vertical Bar Character

If we are searching for the actual vertical bar character (not the metacharacter), then mathematically we represent the actual vertical bar as follows: `\|`

However, most programming languages typically prefer we state it as follows:

```
RegEx_Pattern = "\\|"
```

And this will allow us to locate the vertical bar character ("|") in a String.

Using the Brackets to Group

Introduction

Our next special characters or "metacharacters" are the brackets - ( ) or [ ]. We'll be mainly focusing on Round Brackets (also known as *Parentheses*), but we'll also look at the square brackets [ ] a little bit as well. The brackets allow us to group together operations, and is thus also called the "Grouping" operator. ***The Brackets metacharacters are used to group together parts of a Regular Expression***.

Why Use the Round Brackets?

We will remember that if we are searching for "Damian" or "Damien", we can use the period metacharacter to give us an *approximate solution*:

```
RegEx_Pattern = "Dami.n"
```

And for a *precise solution*, we can use the vertical bar:

```
RegEx_Pattern = "Damian|Damien"
```

Using the brackets we can restate the *precise solution* more compactly, as follows:

```
RegEx_Pattern = "Dami(a|e)n"
```

Which can be read as creating a Regular Expression to match to any String with the letters: "D", "a", "m", "i", ("a" or "e"), and "n". So, the brackets allow specific parts of the Regular Expression that can be grouped together.

In the example above we are creating a choice between two single characters ("a" and "e"), however, we will see in later examples that the brackets can also be used to create choices between multicharacter Strings as well.

Character Class

A Character Class (or Character Set) is a list of characters enclosed in square brackets, and any one of those characters will represent a match to the class. So for the moment we can say that the Regular Expression with the single characters above, and Character Class below are equivalent in terms of their functionality:

```
Character_Class = "Dami[ae]n"
```

This can be read as creating a Character Class to match to any String with the letters "D", "a", "m", "I", ("a" or "e"), and "n".

Matching the Brackets Character

If we are searching for the Round Brackets characters (not the metacharacter), then mathematically we represent the Open Bracket as follows: `\(`
and for the Close Bracket we can mathematically represent it as follows: `\)`
However, most programming languages prefer we state the Open Bracket as:

```
RegEx_Pattern = "\\("
```

and most programming languages prefer we state the Close Bracket as:

```
RegEx_Pattern = "\\)"
```

And it is the same for the Square Brackets: "`\\[`" and "`\\]`".

Examples with the Period

Introduction

We will remember that the period metacharacter ***represents any other single character, which can be a letter, number, or symbol***. And that we can match "Damian" and "Damien" using the following Regular Expression:

```
RegEx_Pattern = "Dami.n"
```

And, we note that it does match to other Strings as well including: "`Damixn`", "`Dami5n`", "`Dami=n`", and "`Dami n`", so it's an *approximate solution*.

String Length

If we were looking for a String that has to be at least 5 characters long, we could create a regular expression of 5 periods, and that will only match with Strings that are at least 5 characters long, as follows:

```
RegEx_Pattern = "....."
```

So we would get the following matches (and many others):

`Test_Message = "abcde"`	MATCH ✓
`Test_Message = "H3llo"`	MATCH ✓
`Test_Message = "a   z"`	MATCH ✓

And it wouldn't match with a String that was 4 characters (or less) long:

`Test_Message = "wxyz"`	NO MATCH ✗
`Test_Message = "1234"`	NO MATCH ✗

String Format

An ISBN number (International Standard Book Number) is a code that is used to uniquely identify books. It takes the format of a series of digits separated by dashes, with a total of 13 digits (3 digits followed by 2 digits, followed by 5 digits, followed by 2 digits, followed by 1 digit). For example:

- `978-02-41953-53-2`
- `978-03-74537-14-2`
- `978-04-25054-71-0`

The digits in the ISBN can be represented by the period, but the dash character is a special character in using Regular Expressions, so it needs to be proceeded with two slashes (as we've seen before with other special characters), as follows: `\\-`

So bringing it all together, a potential Regular Expression for ISBNs is as follows:

```
RegEx_Pattern = "...\\-..\\-.....\\-..\\-."
```

This will only match a collection of digits (or numbers or symbols) in the ISBN format.

Another Example with the Period

Introduction

We remember that the period metacharacter ***represents any other single character, which can be a letter, number, or symbol***.

String Format

Mobile telephone numbers in Ireland have a prefix of either 085, 086 or 087, and then are followed by seven digits. For example:

- `(085) 1234567`
- `(086) 2345678`
- `(087) 3456789`

So let's look at each individual element of that format:

Element	Regular Expression
`Open Bracket`	We can express this as: `\\(`
`Prefix`	It's either 085, 086 or 087, so let's express it as: `08.`
`Close Bracket`	We can express this as: `\\)`
`Space`	Let's use the space character itself to match with space
`Seven Digits`	We can use seven periods to match the seven digits

So bringing it all together, the Regular Expression is the following:

```
RegEx_Pattern = "\\(08.\\) ......."
```

And, this will match to any of the phone numbers presented above, as follows:

`Test_Message = "(085) 1234567"`	MATCH ✓
`Test_Message = "(086) 2345678"`	MATCH ✓
`Test_Message = "(087) 3456789"`	MATCH ✓

And it wouldn't match with a String that doesn't match that format:

`Test_Message = "(085)1234567"`	NO MATCH ✗
`Test_Message = "(08)5 1234567"`	NO MATCH ✗
`Test_Message = "(085) 123456"`	NO MATCH ✗

But we will remember that this Regular Expression also matches many other Strings that might be invalid phone numbers, for example:

`Test_Message = "(08A) ABCDEFG"`	MATCH ✓
`Test_Message = "(08@) !£$%^&*"`	MATCH ✓
`Test_Message = "(08 )        "`	MATCH ✓

So the Period will match letters and symbols as well as numbers.

3. EXAMPLES 1

Examples with the Vertical Bar

Introduction

We will remember that the vertical bar metacharacter ***is used to choose from a list of alternatives, where any will be a match***. And we saw that if we want to match with the Strings "Damian" or "Damien", we can use the following:

```
RegEx_Pattern = "Damian|Damien"
```

This Regular Expression matches with EITHER "Damian" OR "Damien". And we saw that the same thing can be done as follows:

```
RegEx_Pattern = "Dami(a|e)n"
```

And we noted that sometimes when people are emailing me, instead of "Damian" they type "Damain"; they swap around the "I" and "a", and we used this RegEx:

```
RegEx_Pattern = "Dam..n"
```

However, we could also do the same as follows (and get a more *precise solution*):

```
RegEx_Pattern = "Dam(ia|ai)n"
```

And if we wanted to take in account both the people who confuse the spelling of the name, and those who confuse the order of letters, we could do the following:

```
RegEx_Pattern = "Dam(ia|ai|ie|ei)n"
```

This Regular Expression matches with any of the following four Strings, EITHER "Damian" OR "Damain" OR "Damien" OR "Damein".

Clear Regexes

It is really important that we make our Regular Expressions as clear as possible, so that if we have to modify them in a few months, or someone else has to, it will be easy to understand the purpose of the RegEx. This is not an exact science because something that is very clear to you might be less so to others, but it is important to be aware that we should try to make our RegExes as easy to understand as possible. If we can add comments in to explain to purpose of the RegEx, that will help a lot, but we should also think about how we present the RegEx. As an example, we'll look at searching for a Period or a Vertical Bar character.

So, if we are looking for the Period character, we put two slashes in front of it:

```
RegEx_Pattern = "\\. "
```

And it's the same for a Vertical Bar character:

```
RegEx_Pattern = "\\|"
```

So to search for one or the other of them, we can express it in one of two ways, either as Vertical Bar character or Period character ("|" OR "."):

```
RegEx_Pattern = "\\||\\."
```

Or as Period character or Vertical Bar character ("." OR "|"):

```
RegEx_Pattern = "\\.|\\|"
```

I, personally, find the second of these Regexes easier to read, the first one (`slash, slash, bar, bar, slash, slash, period`) is harder for me to find the OR symbol, than the second one (`slash, slash, period, bar, slash, slash, bar`), but it really is a matter of perspective, and it is always worth getting someone else to check if your Regexes are clear to them.

#RegExThursday © Damian Gordon

Examples with Brackets

Introduction

We will remember that the brackets metacharacters ***are used to group together parts of a Regular Expression***. And we saw that if we want to match with the Strings "Damian" or "Damien", we can do it as follows

```
RegEx_Pattern = "Dami(a|e)n"
```

Which can be read as creating a Regular Expression to match to any String with the letters: "D", "a", "m", "i", ("a" or "e"), and "n".

Telephone Numbers

We saw that mobile telephone numbers in Ireland have a prefix of either 085, 086 or 087, and then are followed by seven digits. For example:

- `(085) 1234567`
- `(086) 2345678`
- `(087) 3456789`

And we suggested a Regular Expression as follows:

```
RegEx_Pattern = "\\(08.\\) ......."
```

However, given that there are only three choices for the prefix, we can do refine this Regular Expression using the Vertical Bar and Brackets as follows:

```
RegEx_Pattern = "\\(08(5|6|7)\\) ......."
```

Which searches for either 085, 086 or 087. And as we noted before, a Character Class (or Character Set) uses Square Brackets to achieve the same thing as the above selection, but without the need for the Vertical Bar notation, so the above RegEx can equivalently be expressed as follows:

```
RegEx_Pattern = "\\(08[567]\\) ......."
```

Movie Titles

The "Jaws" series of movies (1975-1987) concerns the saga of a great white shark and its attacks on people, the titles of the first four movies are:

- `Jaws`
- `Jaws II`
- `Jaws 3D`
- `Jaws The Revenge`

If we wanted to search for any of these titles in a text, we could do so, as follows:

```
RegEx_Pattern = "Jaws($| II| 3D| The Revenge)"
```

In this case we are looking for a title starting with the String "`Jaws`" followed by either: Nothing OR "` II`" or "` 3D`" or "` The Revenge`". So the interesting thing here is that to represent nothing, we use the Dollar symbol (`$`), and this means "*End of String*", so, in other words, that there is no further text after the word "`Jaws`".

3. EXAMPLES 1

Examples with Letter Cases

Introduction

When we are working with Strings, it's important to note that Capital Letters (Uppercase Letters) are treated as being different from Lowercase Letters. So, for example, if the Regular Expression is the Uppercase Letter "D", as follows:

```
RegEx_Pattern = "D"
```

This will match with the String "D", but not to the String "d". We say that Regular Expressions are *Case Sensitive* to describe this property of Regular Expressions.

Uppercase and Lowercase Letters

So, the following three Strings should be considered as three different Strings:

- hello
- Hello
- HELLO

And to detect these three Strings, we could use the following Regular Expression:

```
RegEx_Pattern = "hello|Hello|HELLO"
```

And we can reduce the length of that Regular Expression by doing the following:

```
RegEx_Pattern = "(h|H)ello|HELLO"
```

Lowercase Letters

If we had a Notepad file full of text and we wanted to read all of that text into a computer program, and then only print out the letters that are in Lowercase, there is a really easy way to do that using Regular Expressions. We just read in the file one character at a time, and then check if that is a Lowercase character (or not) by comparing it to the following Regular Expression:

```
RegEx_Pattern =
"(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)"
```

So, if the character read in is a Lowercase letter, it will match the RegEx pattern above, and we can print it out, and if it is anything other than a Lowercase letter, it will not match and therefore we should not print it out. So, for example:

`Test_Message = "a"`	MATCH ✓
`Test_Message = "z"`	MATCH ✓

And any Uppercase letters, Numbers and Symbols will not match, for example:

`Test_Message = "A"`	NO MATCH ✗
`Test_Message = "4"`	NO MATCH ✗
`Test_Message = "@"`	NO MATCH ✗

We also note that the same Pattern can be created by using a Character Class:

```
RegEx_Pattern = "[abcdefghijklmnopqrstuvwxyz]"
```

And this will work exactly the same way, but the Pattern is more compact.

Errors in Regular Expressions

Introduction

Creating Regular Expressions is like every other skill we learn in life, the more we practice doing them, the better we will get at them. When we start writing them, we might construct them wrong sometimes, so we need to check whether or not the Regular Expression is doing what we want it to do, or not.

A Simple Example

Let's imagine we were doing our example of searching for "Damian" or "Damien", but we put the period in the wrong place, so instead of "`Dami.n`", we did this:

```
RegEx_Pattern = "Dam.an"
```

we can easily see there is an error (or "bug") in the RegEx, because it's a simple example. However, if we tested it on the word "Damian", it would work, but if we tested it on the word "Damien", it wouldn't work. So, we need to test our Regular Expressions on things we expect it to match, and things it shouldn't match with.

How long will it really take?

When people are learning to write Regular Expressions by themselves, a lot of them experience a similar cycle; in the beginning, it takes them about 5 minutes to write a Regular Expression, and 25 minutes to fix it (to get it working exactly the way they want). After a few weeks, one of two things happen; either they give up and decide Regular Expressions are too hard for them, or they hang in there, and after a few more weeks they become *Regular Expression Wizards*, which doesn't mean that they get everything right first time, it just means it takes a much shorter period of time to fix any issues, so, maybe 5 minutes to write them, and 5 minutes to fix them.

The lessons we are doing here will make it a lot easier to understand how to write Regular Expressions, and because we are doing it one page at a time, we are learning in a slow and steady manner, so we should never get to a place where the content is too hard for us, but it is also important that we aren't too hard on ourselves. So, we can't expect ourselves to be brilliant at Regular Expressions immediately, and even once we become proficient using them, we should still treat them like writing an essay or an e-mail, we should have the expectation that we will have to draft and redraft a few times to make the content better quality. So, this is my golden rule:

We will spend at least

of our time redrafting our Regexes

So as long as we don't expect that it will all work out first time, everything is easier.

4. DEBUGGING

What is Debugging?

Thomas Edison, Isaac Asimov, J. Robert Oppenheimer, Grace Hopper

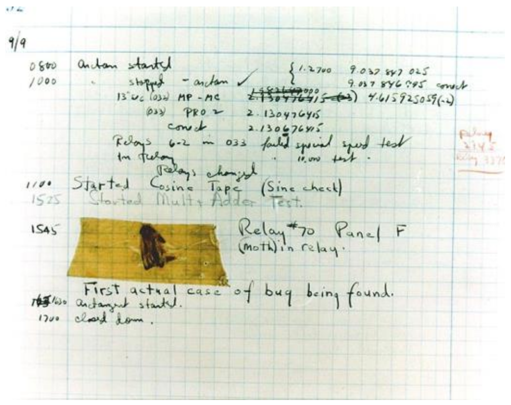
Debugging is …
Sometimes errors in programs are called "bugs", so we have a special name for finding and fixing errors in computer programs, we call it "debugging" (in other words, taking the bugs out). These terms are not exclusive to computers, as far back as the 1870s, Thomas Edison uses the term in a letter, where he says: "*then that "Bugs"—as such little faults and difficulties are called—show themselves*".

This term was used extensively in the 1930s and 1940s to describe *flaws* or *glitches* in mechanical devices. In 1944, the writer Isaac Asimov used the term in a fictional context describing potential errors in robots, in his short story "*Catch That Rabbit*", published in 1944: "*U. S. Robots had to get the bugs out of the multiple robots, and there were plenty of bugs, and there are always at least half a dozen bugs left for the field-testing*."

That same year, on October 27th, in a letter from theoretical physicist, J. Robert Oppenheimer, when he was discussing the building of the first atomic bomb, he mentions, when discussing staff recruitment, that most of the existing staff are "*occupied in getting into operation and debugging*" the bomb.

On September 9th, 1947, computer developer Grace Hopper was tracing an error on the Harvard Mark II electromechanical computer.

Along with one of the operators, Bill Burke, they found a moth trapped in a relay that was the cause of the error, so they taped the moth into the logbook, and recorded it as the first actual bug.


9/9

0800 Anctan started
1000 " stopped - anctan ✓ {1.2700 9.037 847 025 / 9.037 846 995 conect
13" uc (033) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
In relay 11.000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

The notation reads:
"*First actual case of bug being found*."

4. DEBUGGING

Patient Debugging

Fixing Regular Expressions Sometimes Isn't Easy
Now that we have seen some Regular Expressions, it's important to note that writing the Regular Expression often isn't the hard bit, it's when we try to run it, and it gives us lots and lots of wrong matches, and sometimes it can be hard to figure out exactly what the problem is. So we have to have the patience and persistence to review each part of the Regular Expression (as well as the rest of the program that the RegEx is in) to see if we can identify the problem. And often we have to stare at the RegEx for a few minutes before we will, in a flash, figure out what's wrong. This is not easy, and it requires a lot of determination, because often when we have figured out (and fixed) one bug, another one follows. So, we have to type in our Regexes very carefully, and review each part as we write it. And when we are finished writing a RegEx, and go to run it, we need to accept that it may not work correctly the first time, and when we fix the initial bug, there may be another, and another.

Breathing
Fixing RegExes can be a bit stressful, and the longer we are working on a single one, the more frustrated we can get. This can lead to short-term thinking, where we move parts of the RegEx around at random in hopes that it will fix itself. Try to avoid doing this, and try to avoid getting stressed, by breathing. There are a variety of breathing techniques that can be used to calm down, including the following:

- ***Left Nostril Breathing***: As the name suggests, just close your right nostril off, and breath in and out through your left nostril slowly, with your eyes closed. This creates a calming effect in your nervous systems within minutes.
- ***7-2-11 Breathing****:* Breath in through your nose for 7 seconds, hold the breath for 2 seconds, and exhale through your mouth for 11 seconds. This takes a bit of practice, but after a few days of 4-8 sessions a day, you will master it.

Take a Break
One important trick to know is when to take a break; so if I am staring at an bug and I can't figure it out, my rule of thumb is after 7 minutes I walk away from the computer and get a glass of water and stop thinking about it for 2-3 minutes. More often than not as soon as I return to the computer, I know exactly the bug is, because I gave my unconscious mind time to work on it, and can fix it in no time.

Cardboard Helper
Sometimes the easiest way to fix a bug is to ask someone for help, and you will find that as soon as you say to someone "*Excuse me, can you help me with this problem…*", even before you have even outlined the problem, you know what the solution is, because you got a chance to think about it in a different way. In fact, you don't really need another person, just get a cardboard cutout and ask them for help.

How to Find a Bug

Debugging Approaches

There are two parts to debugging, the locating, and fixing, of bugs. We'll look at locating the bugs first, so if the RegEx (or RegExes) runs but doesn't give us the results we are expecting, there is some bug in the RegEx (or RegExes) that we need to find (we should also check the rest of the program). The computer is only doing what it is told, so there must be a wrong instruction somewhere. There are a number of debugging approaches that can be taken to find that instruction:

- ***Brute Force Approach***: The is probably the most common approach to debugging, and it typically involves splitting the Regular Expression into parts from the start, and running each part separately to make sure we understand what it is doing. There are also tools that can be used in this approach, these include both tracing tools and debugging tools, e.g. https://regex101.com/
- ***Backtracking Approach***: The backtracking approach is exactly what it sounds like, you start at the end of the RegEx, manually reviewing each part of the RegEx, or each metacharacter in the RegEx to see if it is correctly written, until the incorrect instruction is found.
- ***Cause Elimination Approach***: This approach involves creating a list of possible causes (or hypothesis) for the bug, and initial tests are carried out to eliminate each hypothesis. Of the ones that cannot be eliminated in the initial testing, further tests are carried out to eliminate more and more hypotheses, until there is only one cause left. The bug is then located.

One More Thing….

This may be just me, but when I'm trying to debug a program, and I don't feel like I'm making progress; sometimes if I recite a verse of poetry, or part of a song, and I do that a couple of times, and it gives me the fortitude to continue and succeed. The two verses below are the ones I most commonly use.

There's nothing you can do that can't be done;
Nothing you can sing that can't be sung;
Nothing you can say, but you can learn how to play the game;
It's easy.

From the song "All You Need Is Love" by John Lennon and Paul McCartney (1967)

Great bugs have little bugs upon their backs to bite 'em,
And little bugs have lesser bugs, and so ad infinitum.
And the great bugs themselves, in turn, have greater bugs to go on;
While these again have greater still, and greater still, and so on.

"Siphonaptera" from Augustus De Morgan's *A Budget of Paradoxes* (1872)

How to Fix a Bug

Read the Error Message
If the bug is producing an error message, that's great, read it carefully. The error messages generally can give us a clue as to what the issue might be, or at least where to start looking from. It can also be incredibly useful for us to put the error message into Google, and search for it; we will often find that someone else has experienced this same bug and has found a solution.

Beware of Side-Effects
Fixing a bug should be done very carefully, if the fix is done poorly, it can introduce other issues into the RegEx, and do more harm than good, so it's important to be careful when fixing bugs. American software engineer Tom Van Vleck outlined three simple questions that we should ask ourselves before making a fix:

1. *Is this bug (or a similar bug) likely to appear in another RegEx?*
   In many cases a programmer will use the same type of logic throughout multiple RegExes, so if an error is found in one RegEx, it may be worth reflecting on whether or not there are other RegExes that has similar functionality.

2. *What new issue might be introduced into the RegEx when fixing the bug?*
   Before the bug is fixed, it is a good idea to explore the design of the RegEx (and the program around it), to check the context in which the RegEx is being used in. If it depends on other programs or RegExes, then it is important to carefully check what the consequences could be of any changes made.

3. *What can be done to prevent this same bug from happening again?*
   This is the first step in creating a good Software Quality Assurance Process going forward. If there was a problem in the development process that caused the bug to occur, fixing that issue with the process will prevent similar bugs from occurring in the future.

Other Terms for Bugs
Depending on who is discussing bugs, they may use different terms to describe them, so for example, someone in IT Sales might call them *features*, whereas an IT Tester might call them *issues*. Here are some other terms for bugs:

Defects	***Faults***	***Problems***	***Incidents***
Anomalies	***Inconsistencies***	***Variances***	***Failures***
Mistakes	***Exceptions***	***Errors***	***Side Effects***

This is a small sampling of the range of terms used for bugs.

Character Classes: Ranges

Introduction

We are already seen the idea of a Character Class (or Character Set), it's **a list of characters enclosed in square brackets, and any one of those characters will represent a match to the class**. So, for example, the following Regular Expression:

```
RegEx_Pattern = "Dami(a|e)n"
```

Can be more compactly represented as follows:

```
Character_Class = "Dami[ae]n"
```

Character Ranges

If we wanted to match only with Lowercase letters, we could do either this:

```
RegEx_Pattern =
"(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)"
```

or by using a Character Class:

```
RegEx_Pattern = "[abcdefghijklmnopqrstuvwxyz]"
```

However, we can also represent the same range of letters as follows:

```
RegEx_Pattern = "[a-z]"
```

Which means to match to any character in the range "a" to "z". So, in other words, the Regular Expression will match with any single Lowercase letter.

We can create a similar Regular Expression for Uppercase letter, as follows:

```
RegEx_Pattern = "[A-Z]"
```

Which means to match to any character in the range "A" to "Z".

And we can create a similar Regular Expression for numbers, as follows:

```
RegEx_Pattern = "[0-9]"
```

Which means to match to any number in the range "0" to "9".

Combining Ranges

So, if we wanted to search for a String of five characters long, with the first one being Uppercase and the rest being Lowercase, we could do that as follows:

```
RegEx_Pattern = "[A-Z][a-z][a-z][a-z][a-z]"
```

And that Pattern will match to Upper, Lower, Lower, Lower, Lower, for example:

`Test_Message = "Hello"`	MATCH ✓
`Test_Message = "Aaaaa"`	MATCH ✓
`Test_Message = "Abcde"`	MATCH ✓

And it wouldn't match with a String that doesn't match that format:

`Test_Message = "AAaaa"`	NO MATCH ✗
`Test_Message = "aaaaa"`	NO MATCH ✗

And we will see that Character Ranges are very common and very useful in RegExes.

Using the Question Mark

Introduction
We are now moving onto looking at "qualification metacharacters" which are metacharacters that allow us to specific the number of instances of a single character or a grouping of characters in a Regular Expression. Our first qualification metacharacters is the question mark ("?"). ***The question mark metacharacter is matched if there is zero or one instances of the preceding character (or grouping) in a Regular Expression***.

Why Use the Question Mark?
If we were searching for the word "colour", but we are dealing with a collection of documents some of which are from the USA and some from Europe; we know that in the USA we will be looking for the word "color", so we can create a Regular Expression to describe this as follows:

```
RegEx_Pattern = "colour|color"
```

But a more compact way of saying the same thing is to state that every time we find the word "colour", the "u" may appear either zero or one times in that word. So, if it appears zero times, then the word is "color" and if it appears one time, then the word is "colour". This can be stated using the Question Mark as follows:

```
RegEx_Pattern = "colou?r"
```

So it's a more compact way of matching with either "colour" and "color", as the character preceding the Question Mark ("u") can appear zero or one times.

Matching with Multiple Characters
If we were looking for documents about Regular Expressions, and we were looking for the phrase "Regular Expression" or "RegEx" in them, we could do it as follows:

```
RegEx_Pattern = "Regular Expression|RegEx"
```

We can achieve the same result using the Question Mark metacharacter in conjunction with the round brackets as follows:

```
RegEx_Pattern = "Reg(ular )?Ex(pression)?"
```

So it will match with either "Regular Expression" or "RegEx", as the two strings preceding the question marks ("ular " and "pression") can appear zero or one times.

Matching the Question Mark Character
If we are searching for the actual question mark character (not the metacharacter), then mathematically we represent it as follows: `\?`
However, most programming languages prefer we state the Question Mark as:

```
RegEx_Pattern = "\\?"
```

And this will allow us to locate the question mark character in a String.

Using the Wildcard Star

Introduction

Another “qualification metacharacter” is the Wildcard Star (“*”), which is commonly also known as the Kleene Star, or the Asterisk. it is similar to the Question Mark metacharacter but looks for zero or more instances of a character or group of characters. ***The Wildcard Star metacharacter is matched if there is zero or more instances of the preceding character (or grouping) in a Regular Expression***.

Why Use the Wildcard?

If we are trying to match with either “colour” and “color”, just for fun, we could do by placing the wildcard after the letter ‘u’, as follows:

```
RegEx_Pattern = "colou*r"
```

So, this would match with either “colour” and “color”, but it would also match with “colouur”, “colouuur”, “colouuuur”, “colouuuuur”, etc.

If I get an email with good news, I will reply with the phrase “Yippeeeee”, the number of e’s varies, depending on how good the news is, so if I had a collection of emails that we needed to search for these, we could do so as follows:

```
RegEx_Pattern = "Yippee*"
```

And this would match with “Y”, “I”, “p”, “p”, “e”, and zero or more “e”’s.

If we had an online form, and one of the fields in the form could be left blank, or it could be filled in with any number of characters, we could that as follows:

```
RegEx_Pattern = ".*"
```

Which means any character (period), any number of times (wildcard star).

If we wanted the field to be either blank or any amount of lowercase characters:

```
RegEx_Pattern = "[a-z]*"
```

Which means any lowercase character ([a-z]), any number of times (wildcard star).
If we wanted the field to be either blank or any amount of uppercase characters:

```
RegEx_Pattern = "[A-Z]*"
```

Which means any uppercase character ([A-Z]), any number of times (wildcard star).
And if we wanted the field to be either blank or any amount of numerical characters:

```
RegEx_Pattern = "[0-9]*"
```

Which means any numerical character ([0-9]), any number of times (wildcard star).

Matching the Wildcard Star Character

If we are searching for the actual star character (not the metacharacter), then mathematically we represent it as follows: *
However, most programming languages prefer we state the star as:

```
RegEx_Pattern = "\\*"
```

And this will allow us to locate the star character in a String.

Using the Plus Sign

Introduction

Another “qualification metacharacter” is the Plus Sign (“+”). It is also similar to the Question Mark and the Wildcard Star but looks for one or more instances of a character or group of characters. ***The Plus metacharacter is matched if there is one or more instances of the preceding character (or grouping) in a Regular Expression***.

Why Use the Plus?

If we are trying to match with either “colour” and “color”, and we tried to do it by placing the plus after the letter ‘u’ (it would be *wrong*), as follows:

```
RegEx_Pattern = "colou+r"    ✗
```

This is wrong because it would match with “colour”, but it would not match with “color”, as the “u” must appear at least once. And we remember that It would also match with “colouur”, “colouuur”, “colouuuur”, “colouuuuur”, etc.

However, if we wanted to search a text file for all instances of “Yippee”, including “Yippe” and “Yippeeeeeee” (so the number of e’s varies), we could do so as follows:

```
RegEx_Pattern = "Yippe+"
```

And this would match with “Y”, “I”, “p”, “p”, and one or more “e”’s. This is slightly more compact than the equivalent Wildcard version we have seen (“`Yippee*`”).

If we had an online form, and one of the fields in the form had to have at least one character (or more) in it, we could that as follows:

```
RegEx_Pattern = ".+"
```

Which means any character (period), at least one times or more times (plus sign).

If we wanted the field to have one or more lowercase characters:

```
RegEx_Pattern = "[a-z]+"
```

Which means any lowercase character ([a-z]), one or more times (plus sign).
If we wanted the field to have one or more uppercase characters:

```
RegEx_Pattern = "[A-Z]+"
```

Which means any uppercase character ([A-Z]), one or more times (plus sign).
And if we wanted the field to one or more numerical characters:

```
RegEx_Pattern = "[0-9]+"
```

Which means any numerical character ([0-9]), one or more times (plus sign).

Matching the Plus Character

If we are searching for the actual plus character (not the metacharacter), then mathematically we represent it as follows: `\+`
However, most programming languages prefer we state the plus sign as:

```
RegEx_Pattern = "\\+"
```

And this will allow us to locate the plus character in a String.

5. QUALIFICATIONS

Using Curly Braces

Introduction

Another "qualification metacharacter" is the curly braces { } characters, which are commonly also known as braces, or curly brackets. They are used to specify the number of instances of a character or grouping of characters. ***The Curly Braces metacharacters are a repetition qualification that specifies the number of instances of the preceding character (or grouping) in a Regular Expression***.

Why Use the Curly Braces?

If we know the number of times a character, or a group of characters is going to repeat, or even if we know a minimum and/or maximum number of occurrences of a character or grouping, we should use the Curley Braces. We can use it in four ways, to indicate that the preceding character (or grouping) appears the following times:

`{num}`	Appears exactly `num` times.
`{min,}`	Appears at least `min` times.
`{,max}`	Appears up to `max` times.
`{min,max}`	Appears between `min` and `max` times.

For example, if we are looking for the phrase "aaaaa", we can do:

```
RegEx_Pattern = "a{5}"
```

If we are trying to match no less than 4 "a" characters in a row, we can do:

```
RegEx_Pattern = "a{4,}"
```

And this will match with "aaaa", "aaaaa", "aaaaaa", etc., but not "a", "aa" or "aaa".

If we are trying to match no more than 4 "a" characters in a row, we can do:

```
RegEx_Pattern = "a{,4}"
```

And this will match with "a", "aa", "aaa", and "aaaa".

If we aren't sure exactly how many times "a" will appear in a row, but we know it will be at least 2 times, but no more than 5 times, we can do:

```
RegEx_Pattern = "a{2,5}"
```

And this will match with "aa", "aaa", "aaaa" and "aaaaa", but nothing else.

Matching the Curly Braces Character

If we are searching for the Curly Braces characters (not the metacharacter), then mathematically we represent the Open Curly Brace as follows: `\{`
and for the Close Bracket we can mathematically represent it as follows: `\}`
However, most programming languages prefer we state the Open Curly Brace as:

```
RegEx_Pattern = "\\{"
```

and most programming languages prefer we state the Close Curly Brace as:

```
RegEx_Pattern = "\\}"
```

And this will allow us to locate Curly Braces in a String.

6. EXAMPLES 2

Examples with Character Ranges

Introduction

As we have already seen a Character Class (or Character Set) is **a list of characters enclosed in square brackets, and any one of those characters will represent a match to the class**.

And a range is represented using the dash symbol, for example:

[a-z]	Any character from “a” to “z”, so, all the lowercase letters.
[A-Z]	Any character from “A” to “Z”, so, all the uppercase letters.
[0-9]	Any character from “0” to “9”, so, all the numbers.

Combining Character Ranges

We will also remember that we can represent “Dami(a|e)n” as follows:

```
RegEx_Pattern = "Dami[ae]n"
```

Therefore, if we want to match any letter that is uppercase or lowercase, we can do:

```
RegEx_Pattern = "[a-zA-Z]"
```

Which means any letter that is lowercase or uppercase, e.g. “A”, “a”, “z”, or “Z”.

If we want to match one or more letters that are uppercase or lowercase:

```
RegEx_Pattern = "[a-zA-Z]+"
```

Which means any letter that is lowercase or uppercase ([a-zA-z]) , one or more times (plus sign), e.g., “A”, “a”, “z”, “Z”, “Dog”, “Cat”, “fish”, “BIRD”, etc.

To match any combination of lowercase and numerical characters:

```
RegEx_Pattern = "[a-z0-9]+"
```

Which means any lowercase character ([a-z]) or any numerical character ([0-9]), one or more times (plus sign), e.g. “r2d2” or “c3po”.

To match any combination of lowercase, uppercase and numerical characters:

```
RegEx_Pattern = "[a-zA-Z0-9]+"
```

Which means any lowercase character ([a-z]), or any uppercase character ([A-Z]) or any numerical character ([0-9]), one or more times (plus sign).

Matching with Multiple Characters

We have seen already that if we are trying to find matches for the phrase “RegEx” or “Regular Expression” in a document, we could use a vertical bar, as follows:

```
RegEx_Pattern = "RegEx|Regular Expression"
```

Or use the Question Mark metacharacter, as follows:

```
RegEx_Pattern = " Reg(ular )?Ex(pression)?"
```

And we can also get an approximate match as follows:

```
RegEx_Pattern = " Reg[a-z]*Ex[a-z]*"
```

“Reg” followed by zero or more lowercase letters, and “Ex” followed by the same.

6. EXAMPLES 2

Examples with the Question Mark

Introduction

As we have already seen the Question Mark is ***matched if there is zero or one instances of the preceding character (or grouping) in a Regular Expression***.

And we will remember that if we are looking for either the word "colour" or "color", then we can match these using the Question Mark as follows:

```
RegEx_Pattern = "colou?r"
```

So the character preceding the Question Mark ("u") can appear zero or one times.

And we can do the same for multiple characters with the round brackets as follows:

```
RegEx_Pattern = "Reg(ular )?Ex(pression)?"
```

So it will match with either "Regular Expression" or "RegEx".

Some Common Examples

If we are looking for the singular or plural of a word, e.g. "dog" or "dogs", we can do:

```
RegEx_Pattern = "dogs?"
```

So the character preceding the Question Mark ("s") can appear zero or one times.

If we are looking for a word with a prefix or not, e.g. "dependent" or "independent":

```
RegEx_Pattern = "(in)?dependent
```

So the characters preceding the Question Mark ("in") can appear zero or one times.

Greedy and Lazy Matching

The question mark ("?") has additional meanings in Regular Expressions, and one relates to how much of a particular String matches to a regular expression, and this is called *greedy* and *lazy* matching. So, we can define them as follows:

- **Greedy Matching**: The regex will match to as much of the String as possible.
- **Lazy Matching**: The regex will match to as little of the String as possible.

So, for example, if we had a String "`XLazyXGreedyX`", we could match using:

```
RegEx_Pattern = "[a-zA-Z]*"
```

So, this matches any number of characters ("`*`") which are uppercase or lowercase characters ("`a-zA-Z`"). However, this does match a lot of other Strings, so if we wanted to make the search a bit more specific to our string, we could state that the string we are looking for starts with an "`X`" and ends with an "`X`", as follows:

```
RegEx_Pattern = "X[a-zA-Z]*X"
```

And this will match to the entire String "`XLazyXGreedyX`", because a RegEx will typically try to match to as much of a String as possible (it's *greedy* by default), but we can tell the pattern to do a *lazy* match instead, and just match with as much of the String as needed to get a match (so to be *lazy*) using the question mark:

```
RegEx_Pattern = "X[a-zA-Z]*?X"
```

And this will match to the String "`XLazyX`" (the first substring to match the RegEx).

6. EXAMPLES 2

Examples with the Wildcard Star

Introduction

As we have already seen the Wildcard Star is ***matched if there are zero or more instances of the preceding character (or grouping) in a Regular Expression.***

And we will remember that if we are looking for either the word "colour" or "color", then we can match these using the Question Mark as follows:

```
RegEx_Pattern = "colou*r"
```

So the character preceding the Question Mark ("u") can appear zero or more times, so it would also match with "colouur", "colouuur", "colouuuur", etc.

So the following Regular Expression:

```
RegEx_Pattern = "(abc)*"
```

will match any of these "", "abc", "abcabc", "abcabcabc", "abcabcabcabc", etc.

File Names

If we are looking for a file, and we know its name, but we are not sure of the file type, we can use a wildcard, so, if the file could be:

```
Example.docx
Example.pdf
Example.pptx
```

We can do the following:

```
RegEx_Pattern = "Example\\.*"
```

If we are looking for a file, and we don't know its name, but we do know the file type, we can use a wildcard, so, if the file could be:

```
Greetings.docx
Hello.docx
Hi.docx
```

We can do the following:

```
RegEx_Pattern = "*\\.docx"
```

Character Classes

If we want to create a Regular Expression to match any String that has a mix of uppercase and lowercase characters, we could do the following:

```
RegEx_Pattern = "[a-zA-Z]*"
```

And this would match "", "a", "x", "A", "X", "Ax", "Xa", "XXX","xxx","Xxx", etc.

And, for example, if we had a String "`XStringX`", we can match it as follows:

```
RegEx_Pattern = "X[a-zA-Z]*X"
```

So, this matches a String that starts with "X" followed by any combination of uppercase and lowercase characters ("`a-zA-Z`"), and it has to end with an "`X`", so it will also match with "XX", "XaX", "XAX", "XxX", "XXX", "XHelloX", "XGoodbyeX", etc.

6. EXAMPLES 2

Examples with the Plus Sign

Introduction

As we have already seen the Plus Sign is ***matched if there is one or more instances of the preceding character (or grouping) in a Regular Expression***.

And we will remember that if we wanted to search a text file for all instances of "Yippee", including "Yippe" and "Yippeeeeeee" (so the number of e's varies), we do:

```
RegEx_Pattern = "Yippe+"
```

And this would match with "Y", "I", "p", "p", and one or more "e"'s. This is slightly more compact than the equivalent Wildcard version we have seen ("`Yippee*`").

And the following Regular Expression:

```
RegEx_Pattern = "(abc)+"
```

will match any of these "abc", "abcabc", "abcabcabc", "abcabcabcabc", etc.

File Names

If we are looking for a file, and we don't know its name, but we do know the file type, we can use a wildcard, so, if the file could be:

```
Greetings.docx
Hello.docx
Hi.docx
```

And if we know it can't be "`.docx`", we can do the following:

```
RegEx_Pattern = "+\\.docx"
```

Character Classes

If we want to create a Regular Expression to match any String that has a mix of uppercase and lowercase characters, and has to at least one character, we do:

```
RegEx_Pattern = "[a-zA-Z]+"
```

And this would match "a", "x", "A", "X", "Ax", "Xa", "XXX","xxx","Xxx", etc.

If we want to create a Regular Expression to match any String that has a mix of numbers, uppercase, and lowercase characters, and has at least one character:

```
RegEx_Pattern = "[a-zA-Z0-9]+"
```

And this would match "a", "x", "A", "X", "5", "Ax", "Xa", "XXX","c3po","R2D2", etc.

And, for example, if we had a String "`XStringX`", we can match it as follows:

```
RegEx_Pattern = "X[a-zA-Z]+X"
```

So, this matches a String that starts with "X" followed by any combination of uppercase and lowercase characters ("`a-zA-Z`"), and it has to end with an "`X`", so it will also match with "XaX", "XAX", "XxX", "XXX", "XHelloX", "XGoodbyeX", etc.

Finally, if we were looking for HTML tags, e.g. "`<P>`" or "`<HEAD>`", we can do:

```
RegEx_Pattern = "<[a-zA-Z]+>"
```

So, this matches any String with at least one character, and enclosed in "<" and ">".

6. EXAMPLES 2

Examples with the Curly Braces

Introduction

As we have already seen the curly braces ***are a repetition qualification that specifies the number of instances of the preceding character (or grouping) in a RegEx.***

And we will remember that we can use it in four ways, to indicate that the preceding character (or grouping) appears the following times:

`{num}`	Appears exactly `num` times.
• e.g. `a{5}` matches "`aaaaa`".	
`{min,}`	Appears at least `min` times.
• e.g. `a{4,}` matches "`aaaa`", "`aaaaa`", "`aaaaaa`", "`aaaaaaa`", etc.	
`{,max}`	Appears up to `max` times.
• e.g. `a{,4}` matches "`a`", "`aa`", "`aaa`", and "`aaaa`".	
`{min,max}`	Appears between `min` and `max` times.
• e.g. `a{2,5}` matches "`aa`", "`aaa`", "`aaaa`" and "`aaaaa`".	

Some Common Examples

If we were searching my emails for "`Yippee`", "`Yippeee`", "`Yippeeee`", and all the way up to "`Yippeeeeeeee`", we could do it simply as:

```
RegEx_Pattern = "Yippe{2,8}"
```

So this will match with "`Yipp`" and between two (2) to eight (8) "`e`"s.

If we were looking for "`CamelCaseCamelCaseCamelCaseCamelCase`", we can avoid a lot of typing (and avoid potential typos) by simply do the following:

```
RegEx_Pattern = "(CamelCase){4}"
```

And this will look for the phrase "`CamelCase`" repeated four (4) times.

Password Length

Let imagine we were writing a computer program to check if a user creates a password that adheres to the following rules:

- *The password has to have between 9 and 15 characters.*
- *The password cannot contain special characters (e.g. "%", "@", etc.).*
- *The password can be any combination of lowercase letters, uppercase letters, and numbers.*

We can express that as a Regular Expression as follows:

```
RegEx_Pattern = "[a-zA-Z0-9]{9,15}"
```

So, this matches any combination of lowercase letters, uppercase letters and numbers, in a range of nine (9) to fifteen (15) characters. Additional rules for passwords that might be added could be things like "*the password should be hard to guess*", and "*the password should not be one you have used before*", but these rules wouldn't necessarily be implemented using Regular Expressions.

Natural Language Processing (NLP)

Introduction

Natural Language Processing (NLP) focuses on enabling computers to understand language by analysing the structure of sentences and tagging different words.

Traditional Parts of Speech

The term "parts of speech" is used to describe the role that each word plays in a sentence. We'll have a look at a few of these parts of speech, and we'll see how they can be used to help analyse the structure of a sample sentence.

- *Noun*: A word to describe people, places, or things.

Sample Sentence	"*The **cat** sat on the **mat***."
Tagged words	"*The* [**NOUN**] *sat on the* [**NOUN**]."

- *Verb*: A word to describe an action.

Sample Sentence	"*The cat **sat** on the mat*."
Tagged words	"*The* [**NOUN**] [**VERB**] *on the* [**NOUN**]."

- *Preposition*: A word to describe the relationship between two nouns.

Sample Sentence	"*The cat sat **on** the mat*."
Tagged words	"*The* [**NOUN**] [**VERB**] [**PREPOSITION**] *the* [**NOUN**]."

Natural Language Processing

We have almost all of the words in our sample sentence categorised, so for the rest of them, we can use two additional structures from Natural Language Processing.

- *Noun Phrase*: A group of words that function as a noun, it can include a Main Noun, Adjectives, Determiners, and Modifiers.

Sample Sentence	"***The cat** sat on **the mat***."
Tagged words	"[**NOUN PHRASE**] [**VERB**] [**PREPOSITION**] [**NOUN PHRASE**]."

- *Verb Phrase*: A group of words that function as a verb, it can include a Main Verb, Adverbs, Prepositions, Direct Objects, and Indirect Objects.

Sample Sentence	"*The cat **sat on** the mat*."
Tagged words	"[**NOUN PHRASE**] [**VERB PHRASE**] [**NOUN PHRASE**]."

Once the structure is defined, Regular Expressions can be used to help analyse it.

RegExs in Natural Language Processing

Stages in Natural Language Processing

There are many stages in Natural Language Processing where RegExes can be used:

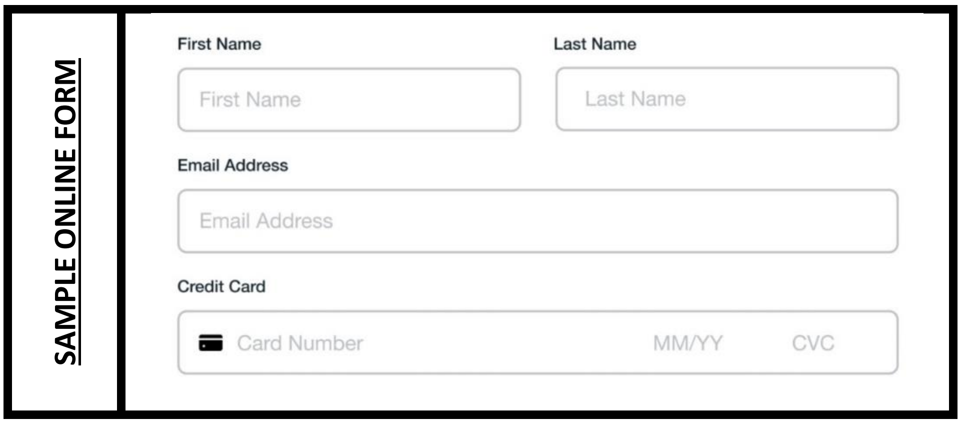
- ***Text Cleaning***: This is typically the first step in NLP, where a text file is taken into the NLP system, and unwanted characters are removed. These can include things such as special symbols, control characters, HTML tags, unwanted punctuation, and extra whitespaces. Regular Expressions can used extensively at this stage to remove such content, simply by only allowing content that matches "`[a-zA-Z0-9]*`" to proceed onto the next stage of the process.
- ***Stop Word Removal***: This stage involves the removal of "Stop words", which are words that are so widely used that they carry very little useful information, so, for example, "`a`", "`the`", "`with`", "`is`", "`are`", "`be`" can be put in a list for the RegEx as follows: `StopWords = (a|the|with|is|are|be).`
- ***Tokenization***: As the name suggests, this stage involves breaking the text into smaller units (called "Tokens"), this can include the following:
  - *Word Tokenization*, so for example, the sentence "`ChatGPT is a ChatBot`" becomes [`"ChatGPT","is","a","ChatBot"`].
  - *Character Tokenization*, so for example, the sentence "`ChatGPT is a ChatBot`" becomes [`"C", "h", "a", "t", "G", "P", "T", " ", "i", "s", " ", "a", " ", "C", "h", "a", "t", "B", "o", "t"`].
  - *Subword Tokenization*, so for example, the sentence "`ChatGPT is a ChatBot`" becomes [`"Chat", "GPT", "is", "a", "Chat", "Bot"`].
- ***Pattern Matching***: This stage involves locating specific patterns within the text, for example, identifying dates, times, names, addresses, and other predefined patterns. So, for example, if the sentence "`Jane Smith is a professor of Computer Science`" is scanned to search for names, it should return "`Jane Smith`".
- ***Entity Recognition***: This stage involves locating special patterns within the text, for example, identifying email addresses, phone numbers, bank account numbers, and social security numbers. We have already seen a RegEx for detecting emails: "`[A-Za-z+.]+@[A-Za-z.]+[A-Za-z]+`".
- ***Text Validation***: This stage involves validating input text against some predefined standard, so for example, validating user input like email addresses, passwords, or other structured data. And we have seen a RegEx for validating passwords: "`[a-zA-Z0-9]{9,15}`".
- ***Text Normalization:*** This stage involves standardizing the text by converting different representations of the same information into a single unified format. So, for example, dates may be in the same document in the following formats: "23rd January 2021", "`04/11/2019`", "`15-NOV-2002`".

These are just some of the stages that Regular Expressions can be used in NLP.

What is Form Validation?

Introduction

When a user fills out an online form, there are checks that can be done using Regular Expressions to ensure that the input adheres to a predetermined set of criteria.



First Name

Assuming that this field requires a first name starting with a capital letter, and followed by lower case characters, this could be checked as follows:

```
RegEx_Pattern = "[A-Z][a-z]+"
```

Last Name

Assuming that this field requires a last name with the same criteria as "First Name":

```
RegEx_Pattern = "[A-Z][a-z]+"
```

Email Address

As we have seen before, the email address can be checked as follows:

```
RegEx_Pattern = "[A-Za-z+.]+@[A-Za-z.]+[A-Za-z]+"
```

Credit Card Number

Credit Card numbers are typically 16 digits long, so we can do the following:

```
RegEx_Pattern = "[0-9]{16}"
```

Date on Card

The date follows the pattern, MM/YY:

```
RegEx_Pattern = "(0[1-9]|1[0-2])/[0-9](2)"
```

CVC

CVC typically has 3 digits long, so we can do the following:

```
RegEx_Pattern = "[0-9]{3}"
```

A RegEx approach can do a great deal, but for more complex validation issues, other approaches can be used in conjunction with RegExes.

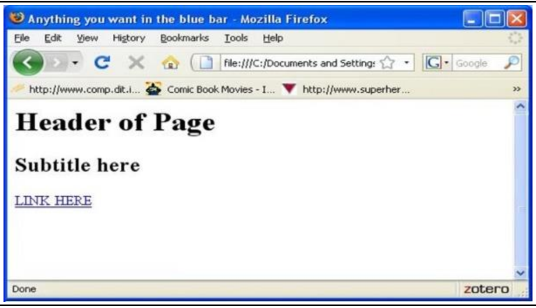
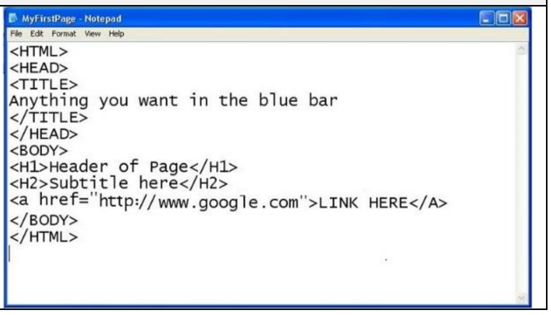
What is Web Scraping?

Introduction

Web scraping is when we write a computer program to extract information off a web page. This isn't as easy as it sounds, because when we view a web page in a web browser, one of the things that the browser is doing is, first, reading the webpage in, and, then, using the HTML instructions on the page to understand how to display the information on the screen correctly, including: which headers to put in, which images to put in, and which hyperlinks to put in. So often when we are scaping a webpage, we are trying to locate just the text on the webpage (and none of the formatting information), so we want to ignore all the HTML instructions.

Simple Example

Below is a sample webpage on the left, with the associated HTML code on the right.

Webpage	HTML
	

Anything you want in the blue bar - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Documents and Setting:
http://www.comp.dit.i...
Comic Book Movies - I...
http://www.superher...
Done
zotero

Header of Page

Subtitle here

LINK HERE

MyFirstPage - Notepad
File Edit Format View Help

```
<HTML>
<HEAD>
<TITLE>
Anything you want in the blue bar
</TITLE>
</HEAD>
<BODY>
<H1>Header of Page</H1>
<H2>Subtitle here</H2>
<a href="http://www.google.com">LINK HERE</A>
</BODY>
</HTML>
```

A web scraping process will be reading the HTML on the right, and may, for example, be attempting to extract the following text for that HTML:

- "Header of Page"
- "Subtitle here"
- "LINK HERE"

So, if we were looking for simple HTML tags, e.g. "<P>" or "<HEAD>", we can do:

```
RegEx_Pattern = "<[a-zA-Z]+>"
```

This matches a String with at least one character, and enclosed in "<" and ">".

If we are looking for closing HTML tags, e.g. "</P>" or "</HEAD>", we can do:

```
RegEx_Pattern = "<[\/a-zA-Z]+>"
```

This matches a String with a slash and at least one character, enclosed in "<" and ">".

If we are looking for the HTML for a link tag, it is typically structured as follows:

```
<A HREF="URL">TEXT</A>
```

So this can be described as follows:

```
RegEx_Pattern = "<[A-Z] [A-Z]+=\"[A-Z]+\">[A-Z]+<\/[A-Z]>"
```

This matches a String that starts with a "<", followed by a single letter, a space, a word, an equals sign, a double quote, another string, another double quote, a ">", another string, another "<", a forward slash, a single letter, and another ">".

What is Data Mining?

Introduction
Data Mining is a lot like web scraping, except instead of trying to extract data off a webpage, we are trying to extract data out of a large dataset. The specific goal is to explore datasets to uncover hitherto undiscovered patterns in the data.

There is a (possibly apocryphal) story that is often used to illustrate data mining, and it’s called the “Beers and Nappies” story. The story goes that a large American supermarket, usually it’s Walmart, was exploring its sales data from their cash registers. The data is stored one customer’s purchase after another, but when the supermarket mined the dataset, they looked at each product to see if it is commonly associated with any other products, they found an unexpected pattern between the purchase of beers and the purchase of nappies. The supermarket starting to place those two products right beside each other on the supermarket floor and they made lots of money. The explanation for the association between the products could not be deduced from the dataset, but the cashiers explain that if a couple with a baby have one partner at home minding the baby, and one going to work; the partner who is going to work will pop into the supermarket after work to buy some nappies, and will decide that they need to get themselves some beers as well ;-)

So the types of activities we would use in Data Mining are quite similar to the ones we would be using in web scraping:

- ***Data Extraction***: Locating specific information from the datasets, including structured information like email addresses, phone numbers, and URLs.
- ***Pattern Matching***: Identifying patterns or sequences within the data, for example, looking for specific sequences of characters or words that might indicate trends, common phrases, or anomalies in the data.
- ***Data Cleaning***: Dealing with messy or inconsistent data, by cleaning and standardizing the data. This usually involves identifying and replacing or removing unwanted characters, symbols, or patterns.
- ***Named Entity Recognition (NER)***: When identifying domain-specific entities, such as products in a supermarket, gene names in bioinformatics, or financial symbols in stock market analysis.
- ***Language Detection and Classification***: Regexes can be used as a part of language identification systems to identify the language of a given dataset by analyzing character patterns or specific linguistic features.
- ***Fraud Detection***: In fraud detection systems Regexes can be used to identify suspicious patterns in transactions, and to create rules to flag potentially fraudulent activities such as identity theft, or abnormal usage patterns.
- ***Log Analysis***: Parsing log files generated by systems, applications, or servers, by extracting relevant information such as timestamps, error codes, IP addresses, or user activities, enabling analysts to identify trends, troubleshoot issues, and improve system performance.

8. ADVANCED FEATURES

Character Slashes

Introduction

We have already seen the use of the slash to match special characters:

Code	Description	Code	Description
\.	Matches the period.	**\?**	Matches the question mark.
\\\|	Matches the vertical bar.	*****	Matches the wildcard star.
\(	Matches the open round bracket.	**\+**	Matches the plus symbol.
\)	Matches the close round bracket.	**\/**	Matches the front slash.
\[	Matches the open square bracket.	**\{**	Matches the open curly braces.
\]	Matches the close square bracket.	**\}**	Matches the close curly braces.

There are other slash codes to match with character types and white spaces:

Code	Description
\d	Matches any of the digit characters (0-9)
\D	Matches any character other than the digit ones (a-z, A-Z, @, $, etc.)
\w	Matches any of the word characters (a-z, A-Z, and "_")
\W	Matches any character other than the word ones (0-9, @, $, etc.)
\s	Matches any whitespace character (space, tab, newline, etc.).
\S	Matches any character other than the whitespace ones (a-z, A-Z, 0-9, etc.)
\t	Matches the tab character
\r	Matches the return character
\n	Matches the new line character

Examples

- "X" matches pattern "\w"
- "XXX" matches pattern "\w\w\w"
- "XXX" matches pattern "\w{3}"
- "3" matches pattern "\d"
- "3X3" matches pattern "\d\w\d"
- "12.345" matches pattern "\d+\.\d+"
- "[34]" matches pattern "\[\d\d\]"
- "Array[5] " matches pattern "\w+\[\d+\]"
- "(abc)" matches pattern "\(\w\w\w\)"
- "3D " matches pattern "\d\w\s"

Raw Notation

We mentioned that programming languages prefer to have two backslashes instead of one, so in programming the question mark character "`\?`" becomes "`\\?`". The backslash character itself is matched to "`\\`" which when programming becomes "`\\\\`", which is just too much. If we put the letter "`r`" at the start of a computer program RegEx, it goes into "raw notation" and we don't need for the extra slashes.

- "`\\?`" is the same as `r"\?"` and "`\\\\`" is the same as `r"\\"`

Text Boundaries

Introduction

Text Boundaries refer to the beginning and end of strings, or the beginning and end of words within a string. Sometimes if we are looking for a pattern in a String, we might only want to match it if it comes at the start or at the end of a String or word.

Word Boundaries

Code	Description
\b	Matches to a pattern if it's at the start or at the end of a single word.
\B	Matches to a pattern if it's in the middle of a single word.

- If the word we are considering is "`Hello`", then "**`\b`**`He`" will match, as well as "`llo`**`\b`**", but "**`\b`**`el`" will not match, and neither will "`ll`**`\b`**".
- And with the "**`\B`**", it's the opposite, if the word is "`Hello`", then "**`\B`**`el`" will match, and so will "`ll`**`\B`**", but "**`\B`**`He`" and "`llo`**`\B`**" will not match.

String Boundaries

Code	Description
^	Matches the beginning of an input String.
$	Matches the end of an input String.

- If, for example, the String is "`Hello, World!`", then the expression "**`^`**`Hell`" will match, but "**`^`**`ello`" will not match.
- And with the "**$**", the expression "`rld!`**`$`**" will match, but the expression "`world`**`$`**" will not match.
- We can use both together, "**`^`**`X{3}`**`$`**" matches the standalone String "`XXX`".

Why do we need Text Boundaries?

If we don't use Text Boundaries, and the Regular Expression pattern matches the middle, or end of the String it is being compared to, the output will differ depending on which programming language being used, e.g.:

Position of Text	**RegEx Code**	***C***	***Python***	***Java***

Text from the middle of the String	`RegEx_Pattern = "llo"`
	`Test_Message  = "Hello, World!"`
		✓	✗	✓

Text from the end of the String	`RegEx_Pattern = "World!"`
	`Test_Message  = "Hello, World!"`
		✓	✗	✓

So in ***Python***, if the pattern is from the middle or end of the string, it would indicate that the two Strings are not an exact match (which is correct), whereas in either ***Java*** or ***C***, the outcome would be that the two Strings are an exact match (they really aren't an ***exact*** match), so they will ignore the extra characters.

8. ADVANCED FEATURES

Character Classes: Advanced

Introduction

We are already seen the idea of a Character Class (or Character Set), it's **a list of characters enclosed in square brackets, and any one of those characters will represent a match to the class**. So, for example, the following Regular Expression:

```
RegEx_Pattern = "Dami(a|e)n"
```

Can be more compactly represented as follows:

```
Character_Class = "Dami[ae]n"
```

For a String of five characters long, with the first Uppercase and the rest Lowercase:

```
RegEx_Pattern = "[A-Z][a-z][a-z][a-z][a-z]"
```

And that Pattern will match to Upper, Lower, Lower, Lower, Lower character,

Chemical Symbols

The chemical elements are represented by a one-letter or two-letter symbol, so for example, Oxygen is O, Hydrogen is H, Sodium is Na, and Lead is Pb. As a RegEx:

```
RegEx_Pattern = "[A-Z][a-z]?"
```

So the pattern is one uppercase letter, and one optional lowercase letter. When people are trying to develop patterns like the pattern above `[A-Z][a-z]?`, sometimes they get confused and write it as `[A-Za-z]?`, which has a very different meaning, and it's saying match with or of two things:

- Match with a single character either uppercase or lowercase, or
- Match with a string with no characters in it (a "*Null String*").

Negation in Character Classes

We have already seen the caret symbol (**^**) being used to indicate the beginning of a String boundary, but if this symbol is used inside a Character Class (i.e. inside the square brackets) it has another meaning; it means that a String will match with anything except the pattern in the Character Class. So, for example:

```
RegEx_Pattern = "[A-Z]"
```

Means match any single uppercase character, and:

```
RegEx_Pattern = "[^A-Z]"
```

Means match anything single character except uppercase characters, so it will match to any of the lowercase characters, any numbers, any symbols, and any whitespace. So, if we wanted to remove all the vowels from an input String, we could do:

```
RegEx_Pattern = "[^aeiou]"
```

And that would match everything except vowels.

Non-Literal Characters ^ -] \

Some symbols inside Character Classes are treated literally, so for example, if we were looking for a period character outside of a Character Class, we would do `\.` but inside a Character Class we can just do `[.]` All symbols are treated literally in Character Classes except for the following symbols: the caret (`^`), the hyphen (`-`), the close square bracket (`]`), and the backslash (`\`), which all have special meanings.

8. ADVANCED FEATURES

Assertions and Capturing Groups

Assertations

These are like `IF-THEN` statements in programming languages.

Lookahead Assertion	**`x(?=y)`**

Matches "x" but only if it is followed by "y", so for example, `Hello(?=World)` means we will match "`Hello`" only if followed by "`World`". And the RegEx `Hello(?=World|You)` matches "`Hello`" if followed by "`World`" or "`You`".

Negative Lookahead Assertion	**`x(?!y)`**

Matches "x" but only if it is not followed by "y", so for example, the RegEx `Hello(?!World)` means we will match "`Hello`" if not followed by "`World`".

Lookbehind Assertion	**`(?<=y)x`**

Matches "x" but only if it is proceeded by "y", so for example, the RegEx `(?<=John|Tom)Smith` means we will match "`Smith`" but only if it is proceeded by either "`John`" or "`Tom`".

Negative Lookbehind Assertion	**`(?<!y)x`**

Matches "x" but only if it is not proceeded by "y" , so for example, the RegEx `(?<!John|Tom)Smith` means we will match "`Smith`" but only if it is not proceeded by either "`John`" or "`Tom`".

Capturing Groups and Backreferences

These are way of remembering (or forgetting) previous searches,

Capturing Groups	**`(x)`**

Matches "x" and remembers all the matches to "x" in a given String, so for example, if we say `(aB)` with the String "`aBa`" it will match and print out `['aB']`

Non-capturing Groups	**`(?:x)`**

Matches "x", but will not remember the matches to "x" in a given String, so for example, if we say `(a)(?:B)` to the String "`aBa`" then it will look to match with an "a" followed by "B", but prints out the "a" and not the "B", so we get `['a']`

Named Capturing Groups	**`(?<Name>x)`**

Matches "x" and remembers all the matches to "x" in a given String, and labels those matches with the label `Name`, so for example, if we say `(?<MySearch>aB)` with the String "`aBa`" it will match and it will save `['aB']` with the label `MySearch`

Backreferences	**`\k<Name>`**

Identifies the last String that matched to the Label `Name`, so for example, the RegEx `\k<MySearch>` will match to `['aB']` assuming we have just run the example from the previous section on "Named Capturing Groups".

RegExes Summary

Simple Matching

Code	Description
.	Single Character Match
|	Logical OR from a List
( )	Grouping Content

Qualifications

Code	Description
?	Zero or one occurrences
*****	Zero or more occurrences
+	One or more occurrences
{ }	**{num}** Appears exactly num times **{min,}** Appears at least min times **{,max}** Appears up to max times **{min,max}** between min and max

Symbol Slashes

Code	Description
\\.	Matches the period
\\\|	Matches the vertical bar
\\(	Matches the open round bracket
\\)	Matches the close round bracket
\\[	Matches the open square bracket
\\]	Matches the close square bracket
\\?	Matches the question mark
\*	Matches the wildcard star
\\+	Matches the plus symbol
\\/	Matches the front slash
\\{	Matches the open curly braces
\\}	Matches the close curly braces

Character Classes

Code	Description
[XY]	Matches **X** or **Y**
[a-z]	Matches any lowercase character
[A-Z]	Matches any uppercase character
[0-9]	Matches any number
[.]	Matches to period character (It is equivalent to \\ .)
[^X]	Matches everything except X
The non-Literal Characters are ^ -] \\	

Character Slashes

Code	Description
\\d	Matches any of the digit characters (0-9)
\\D	Matches any character other than the digit ones (a-z, A-Z, @, $, etc.)
\\w	Matches any of the word characters (a-z, A-Z, and "_")
\\W	Matches any character other than the word ones (0-9, @, $, etc.)
\\s	Matches any whitespace character (space, tab, newline, etc.).
\\S	Matches any character other than whitespace ones (a-z, A-Z, 0-9, etc)
\\t	Matches the tab character
\\r	Matches the return character
\\n	Matches the new line character

Boundaries

Code	Description
\\b	Matches the start or end of a word
\\B	Matches the middle of a word
^	Matches the start of a String
$	Matches the end of a String

Assertations

Code	Description
X(?=Y)	Matches X if followed by Y
X(?!Y)	Matches X if not followed by Y
(?<=Y)X	Matches X if Y is proceeding
(?<!Y)X	Matches X if Y not proceeding

Groups and Backreferences

Code	Description
(X)	Remembers matches to X
(?:X)	Doesn't remember when matching with X
(?<Name>X)	Matches to X, and stores the match as Name
\\k<Name>	Remembers the last match to stored Name

#RegExThursday © Damian Gordon