

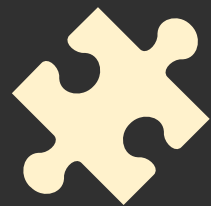
The Smartest Way to Learn Python Regex



1. Read a book
chapter to get
the concepts



2. Watch course
video online to
make it stick



3. Solve a code
puzzle at
finxter.com



Dr. Christian Mayer,
Lukas Rieger, Dr. Zohaib Riaz

The Smartest Way to Learn Python Regular Expressions

Learn the Best-Kept Productivity Secret
of Code Masters

Christian Mayer, Lukas Rieger, and Zohaib Riaz

Table of Contents

Table of Contents	ii
1 Introduction	1
2 Applications	7
3 About	24
4 Puzzle Learning	28
5 Basics	35
6 Special Symbols	44
7 Character Sets	53

<i>TABLE OF CONTENTS</i>	iii
8 Dot Regex	68
9 Asterisk Quantifier	81
10 Plus Quantifier	87
11 ? Quantifier	94
12 Quantifier Differentiation	102
13 Greediness	106
14 Line and String Boundaries	119
15 OR Regex	132
16 AND Regex	140
17 NOT Regex	148
18 Matching Groups	151
19 Split Method	167
20 Substitution Method	175
21 Compile Method	185
22 Bonus Puzzles	195

23 Final Remarks**224**

Long Table of Contents

Table of Contents	ii
1 Introduction	1
2 Applications	7
3 About	24
4 Puzzle Learning	28
5 Basics	35
5.1 What's a Regex Pattern?	36
5.2 The Method to Find All Matches . . .	40
5.3 Summary	42
6 Special Symbols	44

6.1	Literal Characters	44
6.2	Summary	51
7	Character Sets	53
7.1	What's a Character Set?	53
7.2	Negative Character Set	55
7.3	Summary	56
7.4	Intermezzo: Match Function	58
7.5	Methods <code>re.match()</code> vs <code>re.findall()</code>	62
7.6	<code>re.fullmatch()</code>	63
7.7	Methods <code>re.fullmatch()</code> vs <code>re.match()</code> .	66
8	Dot Regex	68
8.1	What's the Dot Regex?	68
8.2	Intermezzo: Flags	71
8.3	Match Newline	75
8.4	Match the Dot Character	76
8.5	Summary	78
9	Asterisk Quantifier	81
9.1	What's the Asterisk Quantifier?	81
9.2	Asterisk Example	82
9.3	Match the Asterisk Character	84
9.4	Summary	85
10	Plus Quantifier	87
10.1	What's the Plus Quantifier?	87

10.2 Plus (+) Quantifiers Examples	88
10.3 Match the Plus (+) Symbol	90
10.4 Summary	91
11 ? Quantifier	94
11.1 Definition Question Mark	94
11.2 Summary	98
12 Quantifier Differentiation	102
12.1 Asterisk vs Plus	103
12.2 Asterisk vs Question Mark	103
Question Mark vs Plus	104
13 Greediness	106
13.1 Regex Quantifiers	106
13.2 Greedy Match	109
13.3 Non-Greedy Match	110
13.4 Non-Greedy Question Mark (??)	111
13.5 Non-Greedy Asterisk (*?)	112
13.6 Non-Greedy Plus (+?)	112
13.7 Greedy vs Non-Greedy	113
13.8 Which is Faster?	114
13.9 Summary	117
14 Line and String Boundaries	119
14.1 Start-of-String (^)	119
14.2 Start-of-Line (^)	120

14.3	End of String (\$)	122
14.4	End of Line (\$)	122
14.5	Intermezzo: Regex Methods	124
14.6	Match Caret (^) or Dollar (\$) Symbols	128
14.7	Summary	129
15	OR Regex	132
15.1	What's the OR Operator?	132
15.2	Examples	133
15.3	Nested OR	135
15.4	Character Set OR	136
15.5	Match the Vertical Line ' '	137
15.6	Summary	138
16	AND Regex	140
16.1	Ordered Regex AND Operator	141
16.2	Unordered Regex AND Operator	142
16.3	Summary	145
17	NOT Regex	148
18	Matching Groups	151
18.1	Simple Matching Groups	151
18.2	First Matching Group	153
18.3	Other Matching Groups	154
18.4	Named Groups	156
18.5	Non-Capturing Groups	157

18.6	Positive Lookahead	159
18.7	Negative Lookahead	161
18.8	Group Flags	162
18.9	Summary	164
19	Split Method	167
19.1	re.split()	167
19.2	Minimal Example	169
19.3	Maxsplit Argument	170
19.4	Optional Flag Argument	171
19.5	Regex Versus String Split	172
19.6	Summary	173
20	Substitution Method	175
20.1	re.sub()	175
20.2	Minimal Example	177
20.3	count Argument	179
20.4	Optional Flag Argument	180
20.5	Regex Sub Versus String Replace . . .	181
20.6	Remove Regex Pattern	182
20.7	Summary	183
21	Compile Method	185
21.1	re.compile()	185
21.2	Regular Expression Object	187
21.3	Discussion	188
21.4	Summary	192

22 Bonus Puzzles	195
23 Final Remarks	224
Where to go from here?	225

— 1 —

Introduction

The human brain is a marvelous creation. Even with today's cutting-edge scientific technology, we understand very little about its detailed working. Putting aside the numerous body functions controlled by the brain, let's think for a moment about a single one: the ability to recall information from our memory. We know very well that the brain does this remarkably fast. However, it is not just the speed that is remarkable. It's also astounding how easy does the brain make it for us: we just have to think about the topic of interest. As soon as we do that, the relevant thoughts and memories start popping up in our minds.

Now compare this functionality of the human brain

with modern-day computers. For instance, if you wish to search a single word “accommodation” in a text file, you have to be careful in spelling it correctly in the search box of the underlying application (e.g., Notepad). If you happen to misspell it, the search may fail even if the text file contains numerous occurrences of “accommodation”. The point we want to highlight here is that, unlike the brain, computers always need precise instructions. This also explains why we have coding-languages, such as Python, as a medium of instruction from humans to machines.

However, it’s a shame that we must always be so “mechanical” when communicating with computers. It’s just not natural. In stark contrast to machines, we as humans can speak to each other quite *imprecisely*. Our high intelligence caters to the inaccuracies of our expression such that we can successfully exchange information and ideas by even talking *roughly* about them. For example, imagine traveling to a country whose language you do not know. You can still reasonably expect the local people to, for example, guide you to nearby restaurants if you approach them with some approximate expression of your need (e.g., a few words in the local language or even some sign language).

For computers to do the same, extra work is re-

quired to make them aware of the possible inaccuracies or deficiencies of human expression and to equip them to handle these appropriately. You can see a simple example of this extra work in today's online search engines, such as Google. These search-engines are generally intelligent enough to look up relevant web pages while handling misspelled or out-of-order search terms. On a high level, they present a much-desired, human-like interface where approximate search terms are mapped to quite precise topics occurring in relevant web pages. This human-like functionality has a cost that not everyone can afford: it must be programmed into these search engines. If you and I want to implement such human-like functionality for offline searches on our PCs, its quite infeasible given the high levels of the required expertise as well as the effort.

This leads us (and probably you too) to ask the following questions: Is there an “easier” way of generally improving our day-to-day communication with computers, making it more human-like? We surely don't want to implement complex natural-language-processing algorithms¹ by hand. Even if we somehow

¹Modern day machine-learning techniques that enable computers to understand text information produced by humans, such as books or articles.

implement these algorithms, do we really understand how to benefit from their strengths maximally? Is it something that anybody could do? Or does it boil down to a “nerdy” topic that no non-expert should touch? And most of all, what’s bad about the mechanical nature of computers? For example, aren’t we already used to instructing them precisely (e.g., for searching stuff)? What improvement can we really make there, if any?

This book will try to address the above questions with the following motivation. We, the authors of this book, have a strong feeling that most people are not aware of existing tools that can remarkably (and really very remarkably!) improve human to computer communication. Tools that are not intended for coding-experts only, but rather represent the low hanging fruits that everyone can benefit from with little effort. In our opinion, these tools can boost a novice coder’s skill-set so much so that they can make expert coders look like fools. In other words, an expert coder who is ignorant of these tools may need to write numerous lines of code to achieve the same result as a novice coder (who knows these tools) would achieve with a single line of code!

Without further ado, the specific tools that we introduce in this book are called Regular Expressions.

A regular expression, also referred to as “regex” or “regexp” for short, is basically a string that defines a search pattern. In comparison to the usual “exact” search-strings, regexes are highly flexible and can trivially accommodate *in-exact* and *imprecise* search patterns. At the same time, they are very powerful in that they allow you to perform searches for complex patterns that otherwise would take significant, often iterative effort when using exact search-strings.

Learning to write these one-liner regexes should tremendously improve how fast you can look up information. Note that the ability to look up the right information (that you really want to find) is not a trivial one. It’s a distinguishing gift!

For anyone dealing with any form of text data, be it a data-scientist collecting and analyzing data-sets, or a software engineer writing or even just exploring huge code-bases, knowing how to use regexes can boost their efficiency and capability drastically.

We, the authors of this book, would like to emphasize again that we firmly believe that regexes are one the most under-rated tools whose usefulness has been unjustifiably overlooked. In our humble opinion, we rate regexes as the second most important skill to learn, right after basic typing skills.

Throughout this book, we will try to keep convincing you in favor of using regexes in your daily work by presenting you with diverse and motivating use-cases. With the multi-faceted, engaging style of teaching adopted in this book (write-up, code-puzzles, and videos), we hope that you will find it easier to appreciate the usefulness of regexes.

To stretch your imagination already, we will now describe several applications where regular expressions are employed.

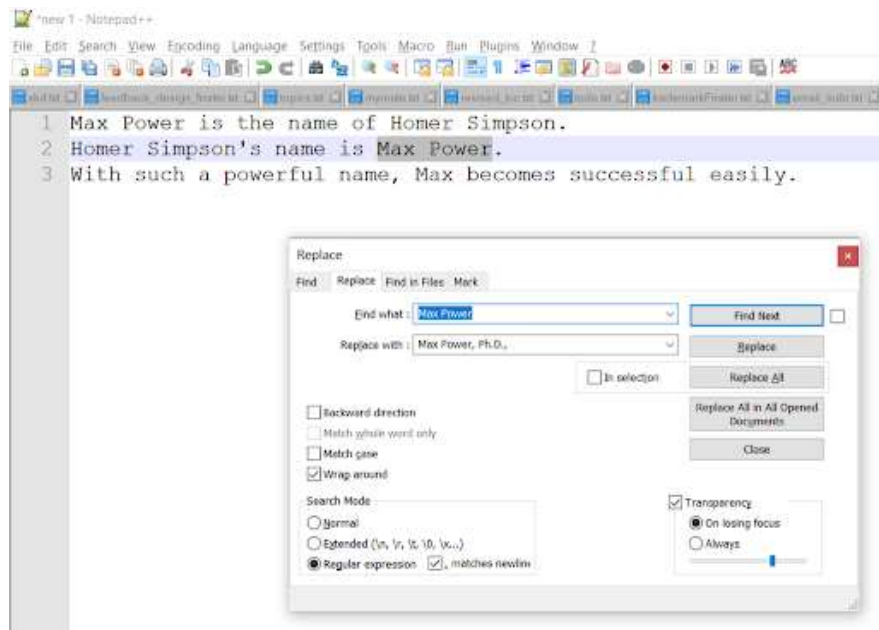
Applications

Regexes are widely used in many practical applications. Let's study some of them in this section.

Search and Replace in a Text Editor: You'll use regular expressions to search a given text in a text editor. Say, your boss asks you to replace all occurrences of a customer **Max Power** with the name **Max Power, Ph.D.,**. How'd you do this? In figure 2.1, we show how it would look like in the popular text editor Notepad++ (recommended for coders).

At the bottom of the “Replace” window, you can see the “Regular expression” radio-button. When developing your regular expression, you realize that you also want to replace all occurrences of **Max** (without the surname **Power**). To accomplish that, you use a

Figure 2.1: Screenshot of a normal text search.

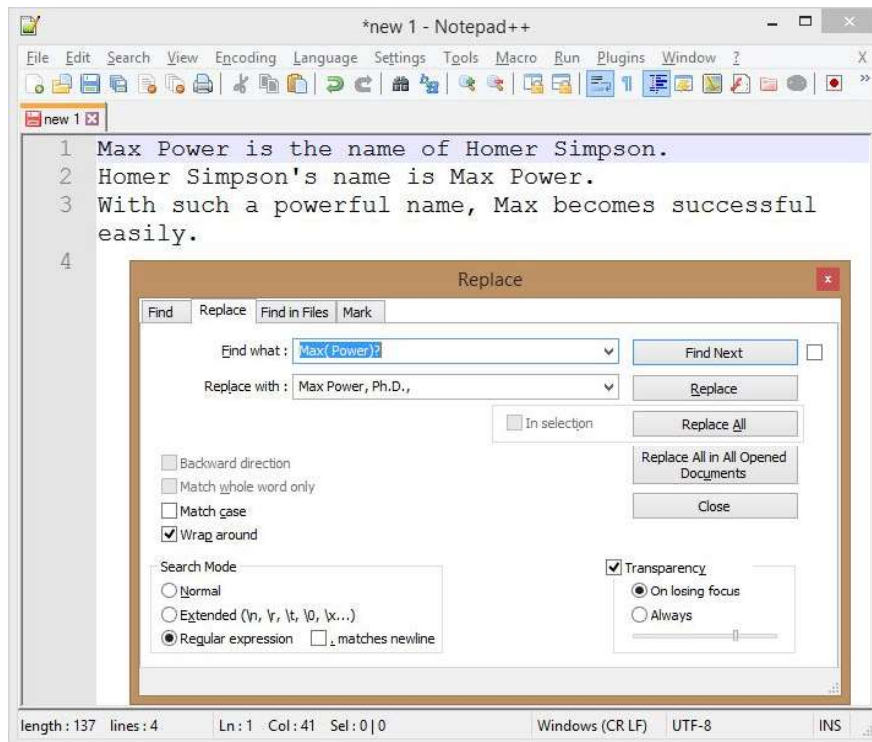


regex in the search field: `Max(Power)?` rather than only `Max Power` (see Figure 2.2).

For now, you don't need to worry about the specific regex `Max(Power)?` and why it works for this task. You just need to know that it's possible to match all occurrences of either `Max Power` or `Max`.

Counting words: How to count the number words in a text document? You could copy the text into a new Microsoft Word document and check the word count on the bottom left. However, what if you don't

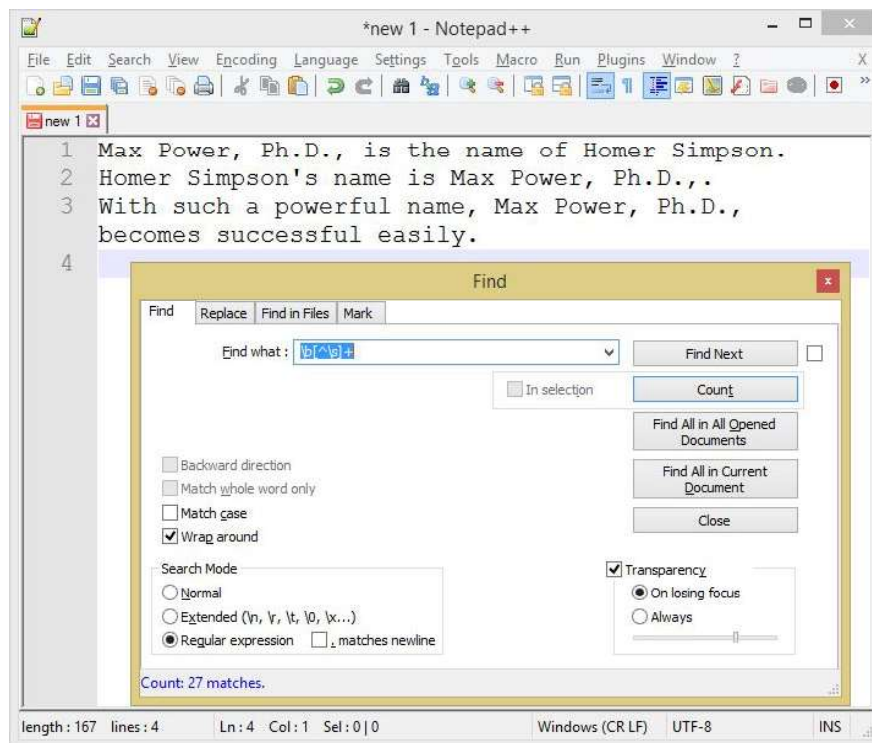
Figure 2.2: Screenshot of a regex text search.



have a Microsoft Word license for your machine, or you simply want to do it somehow inside the Notepad++ editor. Well, as you may have guessed, there is a way by using regular expressions. With the “Regular expression” option selected under the “find” tab, you enter the following regex in the search field: `\b[^\s]+`. Then press the “Count” button. Voila!, the number of words (27) is displayed at the bottom of the search window (see Figure 2.3). Again, you don’t need to

worry about the working of the regex `\b[^\s]+`. We will make sure that the later chapters of this book make things crystal clear!

Figure 2.3: Screenshot of a regex for word-count.



Searching Your Operating System for Files:

You can also use regular expressions to search (and find) specific files on your operating system. For example, a forum user tried to find all files with the following filename patterns:

```
abc.txt.r12222  
tjy.java.r9994
```

You can find all those files on Windows using the command:

```
dir * /s/b | findstr \.r[0-9]+$
```

Large parts of the commands are a regular expression. In the final part, you require the file to end with `.r` and an arbitrary number of numeric symbols. As soon as you've mastered regular expressions, this will cost you no time at all, and your productivity with your computer will skyrocket.

Searching Your Files for Text

But what if you don't want to find files with a specific filename but with a particular file content? Isn't this much harder? As it turns out, it isn't! Well, if you use regular expressions and `grep`. Here's what a `grep` guru would do to find all lines in a file `'haiku.txt'` that contain the word `'not'`.

```
$ grep not haiku.txt
```

This may lead to the following output of all lines that contain the word `'not'`:

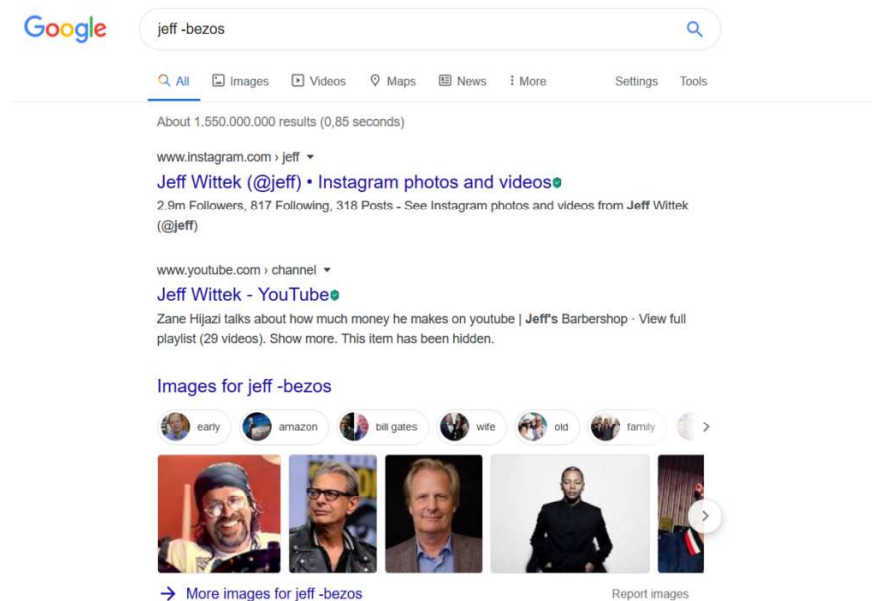
```
Is not the true Tao, until  
"My Thesis" not found  
Today it is not working
```

Grep is an age-old file search tool written by famous computer scientist Ken Thompson. And it's even more powerful than that: you can also search a bunch of files for specific content. A Windows version of grep is the find command.

Search Engines: Using regular expressions to find content on the web is considered the holy grail of search. But the internet is a huge beast, and supporting a full-fledged regex engine would be too demanding for Google's servers. It costs a lot of computational resources. Therefore, nobody actually provides a search engine that allows all regex commands. However, web search engines such as Google support a limited number of regex commands. For example, you can search queries that do NOT contain a specific word (see Figure 2.4).

The search "Jeff -Bezos" will give you all the Jeff's that do not end with Bezos. If a first name is dominated like this, using advanced search operators is quite a useful extension. Mastering search is a critical skill in the 21st century, after all.

Figure 2.4: A Google Search using the NOT operator.



Validate User Input in Web Applications: If you're running a web application, you need to deal with user input. Often, users can put anything in the input fields, even cross-site scripts to hack your webserver. Therefore, your application must validate the user input—otherwise, you're guaranteed to crash your backend application or database. How can you validate user input? Regex to the rescue!

Here's how you'd check whether

- The user input consists only of lowercase letters:

`[a-z]+`,

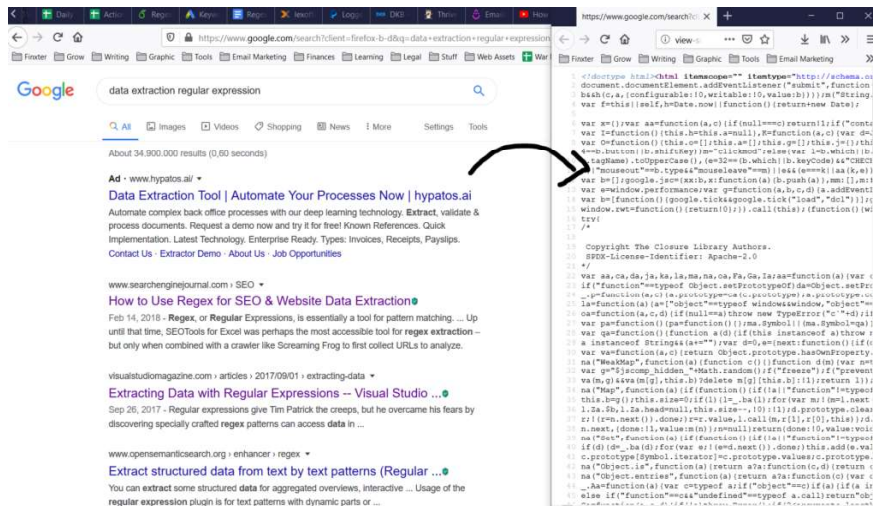
- The username consists of only lowercase letters, underscores, or numbers: `[a-z_0-9]+`, or
- The input does not contain any parentheses: `[^\\(\\)]+`.

With regular expressions, you can validate any user input—no matter how complicated it may seem. Think about this: any web application that processes user input needs regular expressions. Google, Facebook, Baidu, WeChat—all of those companies work with regular expressions to validate their user input. This skill is wildly important for your success as a developer working for those companies (or any other web-based company for that matter).

Web Crawlers to Extract Useful Information: Okay, you can validate user input with regular expressions. But is there more? You bet there is. Regular expressions are not only great to validate textual data but to extract information from textual data. For example, say you want to gain some advantage over your competition. You decide to write a web crawler that works 24/7 exploring a subset of webpages. A webpage links to other webpages. By going from webpage to webpage, your crawler can explore

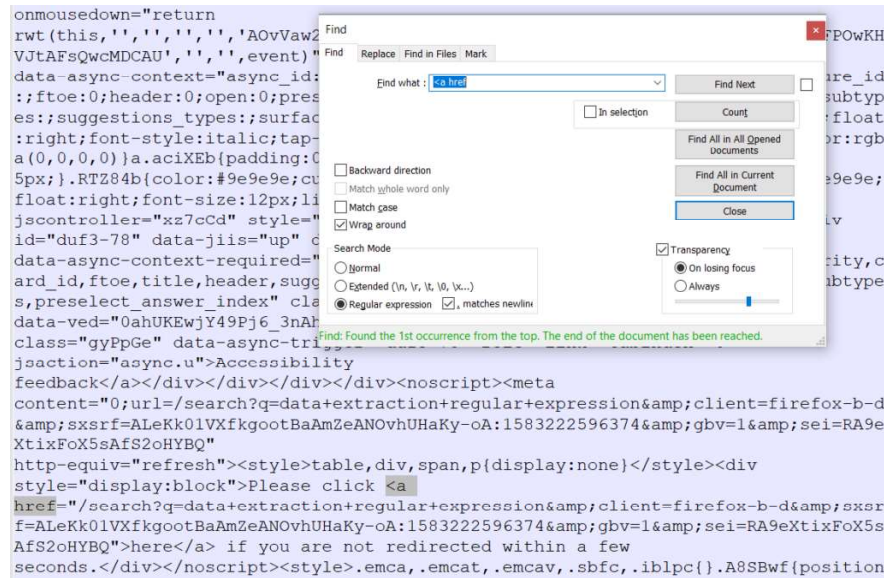
large parts of the web—fully automatized. Imagine the potential! Data is the asset class of the 21st century, and you can collect this valuable asset with your own web crawler. A web crawler can be a Python program that downloads the HTML content of a website (see Figure 2.5).

Figure 2.5: A Google Search.



Now, your crawler can use regular expressions to extract all outgoing links to other websites (starting with "[A simple regular expression can now automatically get the stuff that follows—which is the outgoing URL. You can store this URL in a list and visit it at a later point in time. As you're extracting links, you can](, see Figure 2.6).</p>
</div>
<div data-bbox=)

Figure 2.6: A Google Search using the NOT operator.



build a web graph, extract other information (e.g., embedded opinions of people), and run complicated subroutines on parts of the textual data (e.g., sentiment analysis). Don't underestimate the power of web crawlers when used in combination with regular expressions!

Data Scraping and Web Scraping: In the previous example, you've already seen how web crawlers benefit from regular expressions. But often, the first step is to simply download a specific type of data from a large number of websites to store it in a database (or a spreadsheet). The process of extracting a spe-

cific type of data from a set of websites and converting it to the desired data format is called *web scraping*. Web scrapers are needed in finance startups, analytics companies, law enforcement, eCommerce companies, and social networks. Regular expressions help much in processing the messy textual data. There are many different applications, such as finding titles of a bunch of blog articles (e.g., for search engine optimization). A minimal example of using Python's regex library `re` for web scraping is the following:

```
from urllib.request import urlopen
import re

url =
↪ "https://blog.finxter.com/start-learning-python/"
html = urlopen(url).read()

print(str(html))
titles = re.findall("<title>(.*?)</title>",
↪ str(html))

print(titles)
# ["What's The Best Way to Start Learning Python? A
↪ Tutorial in 10 Easy Steps! | Finxter"]
```

You extract all data that's enclosed in opening and closing title tags: `<title>...</title>`.

Data Wrangling: Data wrangling is the process of transforming raw data into a more useful format to simplify the processing of downstream applications. Such formatting of data and its subsequent filtering or “cleaning” is essential for nearly all data science and machine learning applications. As you may have guessed already, data wrangling is highly dependent on tools such as regular expression engines. In the upcoming chapters of this book, we will show you how to convert a string into a new one where a replacement string replaces each occurrence of pattern. This way, you can transform currencies, dates, or stock prices into a standard format with regular expressions.

Parsing: Any parser leverages hundreds of regular expressions to process the input quickly and effectively. You may ask: what’s a parser anyway? And you’re right to ask. A parser translates a string of symbols into a higher-level abstraction such as a formalized language, often using an underlying grammar to “understand” the symbols. You’ll need a parser to write your own programming language, syntax system, or text editor. For example, if you write a program in the Python programming language, it’s just a bunch of characters. Python’s parser brings order into the chaos and translates the characters from your code file into more meaningful abstractions (e.g., keywords,

variable names, or function definitions). This is then used as an input for further processing stages, such as the execution of your program. If you're looking at parser implementations, you'll see that they heavily rely on regular expressions. This makes sense because a regular expression can easily analyze and catch parts of your text. For example, to extract function names you can use the following regex in your parser:

```
import re

code = '''
def f1():
    return 1

def f2()
    return 2
'''

print(re.findall('def ([a-zA-Z0-9_]+)', code))
# ['f1', 'f2']
```

You can see that our mini parser extracts all function names in the code. Of course, it's only a minimal example, and it wouldn't work for all instances. For example, you can use more characters than the given ones to define a function name. If you're interested in writing parsers or learning about compilers, reg-

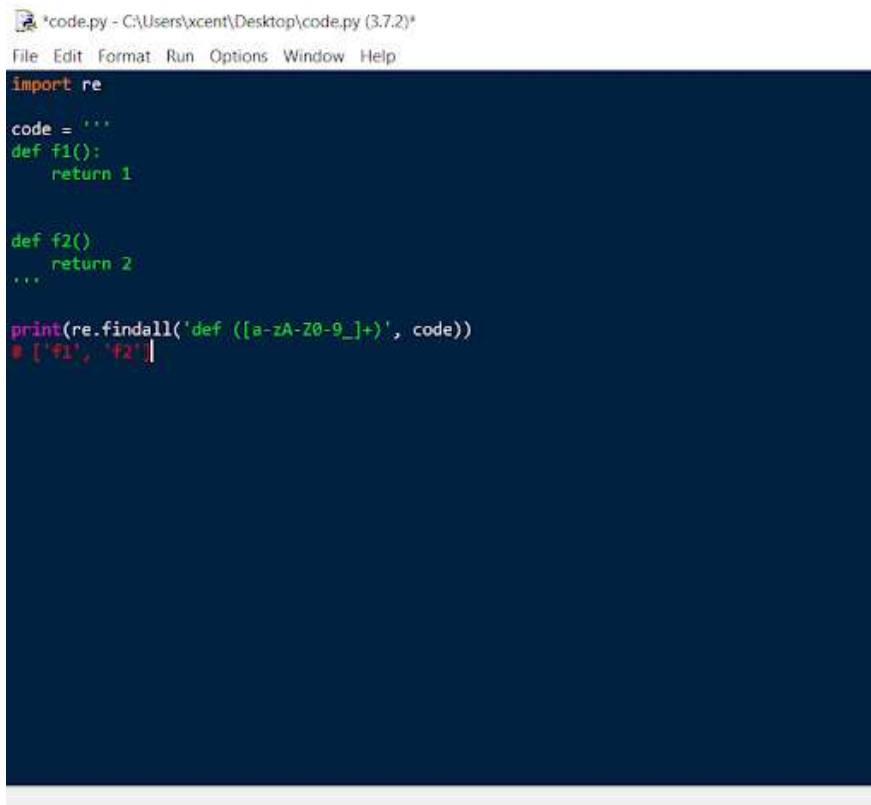
ular expressions are among the most useful tools in existence!

Programming Languages: Yes, you’ve already learned about parsers in the previous point. And parsers are needed for any programming language. Nearly all programming languages rely on regular expressions for their own implementation. For compiled languages, the compiler also uses regular expressions for lexical analysis to scan valid tokens that belong to the programming language to be compiled. The regular expressions define the grammar of the programming language (denoted as *regular grammar*. But regular expressions are also prevalent when writing code *within* the programming language. Some programming languages such as Perl provide built-in regex functionality: you don’t even need to import an external library. If you’re going to become an expert coder, you will use regular expressions in many coding projects. And the more you use it, the more you’ll learn to love and appreciate the power of regular expressions.

Syntax Highlighting Systems: Figure 2.7 shows how your standard coding environment may look like.

Any code editor provides syntax highlighting capabilities. Here’s an example:

Figure 2.7: Syntax Highlighting.



The screenshot shows a Python IDE window titled '*code.py - C:\Users\xcent\Desktop\code.py (3.7.2)*'. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code editor has a dark blue background and displays the following Python code with syntax highlighting: 'import re' (white), 'code = ''' (white), 'def f1():' (white), ' return 1' (white), 'def f2()' (white), ' return 2' (white), '...' (white), 'print(re.findall('def ([a-zA-Z0-9_]+)', code))' (white), and a comment '# ['f1', 'f2']' (red).

```
import re

code = '''
def f1():
    return 1

def f2()
    return 2
...

print(re.findall('def ([a-zA-Z0-9_]+)', code))
# ['f1', 'f2']
```

- Function names are blue.
- Strings are yellow.
- Comments are red.
- And normal code are white.

This way, reading and writing code becomes far more convenient. More advanced IDEs such as PyCharm provide dynamic tool-tips as an additional feature. Those are implemented with regular expressions to locate keywords, function names, and normal code snippets—in order to, ultimately, highlight them and enrich them with additional information, e.g., for tool-tips.

Formal Language Theory: Theoretical computer science is the foundation of all computer science. The great names in computer science, Alan Turing, Alonzo Church, and Steven Kleene spent significant time and effort studying and developing regular expressions. If you want to become a great computer scientist, you need to know your fair share of theoretical computer science. You should know the basics of formal language theory and regular expressions that are at the heart of these theoretical foundations.

How do regular expressions relate to formal language theory? Each regular expression defines a “language” of acceptable words. All words that match the regular expression are in this language. All words that do not match the regular expression are not in this language. This way, you can create a precise set of rules to describe any formal language—by expressing it in terms of the right regular expressions.

The applications of regexes that we described in this chapter are only a small subset of the ones used in practice. We hope to have given you a glance into how vital regular expressions have been, are, and will remain in the future.¹

Watch the short chapter video on YouTube:



<https://youtu.be/v6oi3M117sw>

¹If you want to learn how to apply some of your newly acquired regex skills, check out the Finxter Freelancer webinar at <https://blog.finxter.com/webinar-freelancer/> for free.

— 3 —

About This Book: Structure and Instruction Methodology

The main driver for mastery is neither a character trait nor talent. Mastery comes from intense, structured training. The author Malcolm Gladwell formulated the famous rule of 10,000 hours after collecting research from various fields such as psychology and neurological science.¹ The rule states that if you have average talent, you will reach mastery in any discipline by investing approximately 10,000 hours of intense training. Bill Gates, the founder of Microsoft, reached mastery at a young age as a result of coding for more than 10,000 hours. He was committed and passionate about coding and worked long nights to

¹Malcolm Gladwell *Outliers: The Story of Success*

develop his skills. He was anything but an overnight success.

If you are reading this book, you are an aspiring coder, and you seek ways to advance your coding skills. Nurturing your ambition to learn will pay a rich stream of dividends to you and your family as long as you live. It will make you a respectable member of society, providing unique value to information technology, automation, and digitization. Ultimately, it will give you strong confidence. So keeping your ambition to learn intact is the one thing you must place above all else.

The Smartest Way to Learn Python Regex is the sixth book in the Finxter book series. But as a regular expression introduction, it stands on its own. The other books in the Finxter book series are:

- *Coffee Break Python*
- *Coffee Break Python Workbook*
- *Coffee Break NumPy*
- *Brain Games Python*
- *Python One-Liners*
- *The Smartest Way to Learn Python Regex*

This book aims to be a stepping stone on your path to becoming a Python master. It helps you to learn faster by making use of the established principles of good teaching. It offers you 15 to 25 hours of thorough Python training using one of the most efficient learning techniques, called *practice testing*. Investing this time will improve your skills to write, read, and understand Python source code even further.

The idea is that you break the big task of learning a topic into a series of small interactive steps. In particular, you'll follow the 3-step interactive training method:

- Step 1: You read a book section that explains all the needed concepts and information. After reading the book section, you'll know the theory and have a profound understanding of the basics.
- Step 2: You watch the associated video after completing each section. We'll give you a link to the <https://finxter.com> app where you'll also find the supporting free video. This way, you're strengthening your intuitive understanding of the material through repetition and multi-modal learning. To see and hear the same material explained by your personal teacher

will help you gain deeper insights into the theory.

Step 3: You solve the associated code puzzle on the <https://finxter.com> app link after watching the video. This will commit your theoretical understanding of the material into your memory through the scientifically proven method of *practice testing*.

The next chapter explains and motivates the advantages of the Finxter method of puzzle-based learning. If you already know about the benefits of puzzle-based learning from previous books, and you want to dive right into the technical content, feel free to skip the next chapter.

A Case for Puzzle-based Learning

Definition: A *code puzzle* is an educative snippet of source code that teaches a single computer science concept by activating the learner’s curiosity and involving them in the learning process.

Before diving into practical puzzle-solving, let’s first study some reasons why puzzle-based learning accelerates your learning speed and improves retention of the learned material. There is robust evidence in psychological science for each of these reasons.

Overcome the Knowledge Gap: The great teacher Socrates delivered complex knowledge by asking a sequence of questions. Each question built on

answers to previous questions provided by the student.

A good teacher opens a gap between their knowledge and the learner's. The knowledge gap makes the learner realize that they do not know the answer to a burning question. This creates tension in the learner's mind. To close this gap, the learner awaits the missing piece of knowledge from the teacher. Better yet, the learner starts developing their own answers. The learner *craves knowledge*.

Code puzzles open an immediate knowledge gap. When looking at the code, you first do not understand the meaning of the puzzle. The puzzle's semantics are hidden. But only you can transform the unsolved puzzle into a solved one.

Embrace the Eureka Moment: Humans are unique because of their ability to learn. Fast and thorough learning has increased our chances of survival. Thus, evolution created a brilliant biological reaction to reinforce learning. Your brain is wired to seek new information; it is wired to always process data, to always learn.

Did you ever feel the sudden burst of happiness after experiencing a eureka moment? Your brain releases endorphins, the moment you close a knowledge

gap. The instant gratification from learning is highly addictive, but this addiction makes you smarter. Solving a puzzle gives your brain instant gratification. Easy puzzles open small, hard puzzles, which open large knowledge gaps. Overcome any of them and learn in the process.

Divide and Conquer: Learning to code is a complex task. You must learn a myriad of new concepts and language features. Many aspiring coders are overwhelmed by complexity. They seek a clear path to mastery.

As any productivity expert will tell you: break a big task or goal into a series of smaller steps. Finishing each tiny step brings you one step closer to your big goal. *Divide and conquer* makes you feel in control, pushing you one step closer toward mastery. This book is the embodiment of divide and conquer: the big task of learning regular expressions is broken down into a series of smaller steps. In addition to that, each puzzle is a step toward your bigger goal of mastering computer science. Keep solving puzzles and you keep improving your skills.

Improve From Immediate Feedback: As a child, you learned to walk by trial and error—try, receive feedback, adapt, and repeat. Unconsciously, you

will minimize negative and maximize positive feedback. You avoid falling because it hurts, and you seek the approval of your parents. To learn anything, you need feedback such that you can adapt your actions.

An excellent learning environment provides you with *immediate* feedback for your actions. If you were to slap your friend each time he lights a cigarette—a not overly drastic measure to save his life—he would quickly stop smoking. Puzzle-based learning offers you an environment with immediate feedback to make learning to code easy and fast.

Small is Beautiful: The 21st century has seen a rise in microcontent. Microcontent is a short and accessible piece of valuable information such as the weather forecast, a news headline, or a cat video. Social media giants like Facebook and Twitter offer a stream of never-ending microcontent. Microcontent has many benefits: the consumer stays engaged and interested, and it is easily digestible in a short time. Each piece of microcontent pushes your knowledge horizon a bit further.

This book and its backend Finxter offers a stream of self-contained microcontent in the form of hundreds of small code puzzles. But instead of just being unrelated microcontent, each puzzle is a tiny stimulus that

teaches a coding concept or language feature. Hence, each puzzle pushes your knowledge *in the same direction*.

Active Beats Passive Learning: Robust scientific evidence shows that active learning improves students' learning performance. A popular study shows that one of the best active learning techniques is *practice testing*.¹ In this learning technique, you test your knowledge even if you have not learned everything yet. The study argues that students must feel safe during these tests. Therefore, the tests must be low-stake, i.e., students have little to lose. After the test, students get feedback about the correctness of the tests. The study shows that practice testing boosts long-term retention of the material by almost a factor of 10. As it turns out, solving a daily code puzzle is not just another learning technique—it is one of the best.

Although active learning is twice as effective, most books focus on passive learning. The author delivers information; the student passively consumes it. Puzzle-based learning with this book fixes this mismatch between research and everyday practice. In contrast to other books, this book makes active learning a first-class citizen. Solving code puzzles is an in-

¹<http://journals.sagepub.com/doi/abs/10.1177/1529100612453266>

herent active learning technique. You must develop the solution yourself, in every single puzzle.

Make Code a First-class Citizen: Each grandmaster of chess has spent tens of thousands of hours looking into a nearly infinite number of chess positions. Over time, they develop a powerful skill: the intuition of the expert. When presented with a new position, they can name a small number of strong candidate moves within seconds. For ordinary people, the position of a single chess piece is one chunk of information. Hence they can only memorize the position of about six chess pieces. But chess grandmasters view a whole position or a sequence of moves as a single chunk of information. The extensive training and experience have burned strong patterns into their biological neural networks. Their brain can hold much more information—a result of the good learning environment they have put themselves in. Chess grandmasters learn by practicing isolated stimuli again and again until they have mastered them. Then they move on to more complex stimuli.

Puzzle-based learning is code-centric: code is the isolated stimuli. You will find yourself staring at the code for a long time until the insight strikes. This creates new synapses in your brain that help you understand, write, and read code fast. Placing code in

the center of the whole learning process creates an environment in which you will develop the powerful intuition of the expert. *Maximize the learning time you spend looking at code rather than at other stimuli.*

In the following chapters, we will show you how to create your own regular expressions patterns using Python's regex library `re`. You'll need 20 minutes or so to read each chapter. After each chapter, you'll have the opportunity to watch the related regex video on YouTube and solve the practical code puzzle (interactive or in the book). We'll give you two links for the video and the puzzle which you can use for interactive learning.

— 5 —

Basics

Are you motivated to build your regex superpower? If you are not until now, please revisit Chapter 2 to recall the broad range of applications that thrive on top of regular expressions. It's highly likely that one day you will find yourself a professional in one of those application domains and your knowledge of regular expressions will definitely make a difference in how well you fare.

Now, for this chapter, let's start slowly with the very basics of regular expressions.

5.1 What's a Regex Pattern?

The English language consists of a set of words that are grammatical. All other words are non-grammatical. Both sets are infinitely large for all practical matters. For example, the word *programming* is grammatically correct while the word *qwertzuiopü* is not. For more than one hundred years, researchers have tried to find rules that describe whether a word is grammatical or ungrammatical.

Think of a regular expression as describing a language, too. If you create a regex pattern, you've just created your own language—a fascinating thought, isn't it? In fact, regular expressions and *formal language theory* are closely related fields¹.

You use regular expressions to define the set of “grammatical” words—grammatical for the little language you're designing. How do you do this? You define a set of one or more words through regex patterns (see Figure 5.1). The pattern can be a simple string

¹For instance, mathematician Stephen Cole Kleene developed regular expressions in the 1950s as a means to create *regular languages*. But don't worry, this is as far as we'll get into formal language theory. Well, if you read this whole book, you'll understand the basics of formal language theory—although you won't realize it if you don't decide to pursue a computer science degree



	Regex pattern	
	<code>[\w\.\.]*\w+\ (..*?\)</code>	
<code>re.match(a, b, c)</code>		Hello world
<code>match(a, b, c)</code>		match (
<code>re.re.exec()</code>		(noop)
Matching words		Non-matching words

Figure 5.1: Each regex pattern defines a set of grammatical and a set of ungrammatical words. You call them *matching* and *non-matching words*.

of characters like 'alice' or a complicated string of special symbols, quantifiers, and meta-characters like '`[\w\.\.]*\w+\ (..*?\)`'. The first pattern defines a single word (the word 'alice') while the second pattern defines an infinite number of words (because of the infinite combinations of characters it an match). We used the second pattern—that describes valid Python method calls with their arguments—to style Python method calls with a special code font in this book, which is yet another application of regular expres-

sions!

Of course, we don't expect that you already understand what the complicated pattern does and how it works. But just so you know: if you've read this book, understanding it will be a breeze for you!

Many teachers shy away from giving the regex beginner too much technical detail about how the regex processing internally works. But we believe it'll help you make regular expressions more accessible. Figure 5.2 shows the process of matching the pattern `'hello'` in the string `'hello world'`. The *regex engine* internally goes over the string and keeps track of partial matches that have occurred so far. As soon as the match is done, the engine moves on to either further matching the pattern in the remaining string or it terminates if there's no further match possible.

The regex pattern defines a whole language of valid words. To put it differently, all words in your defined language *match* your pattern. The regex engine is responsible for finding occurrences of the words in your language—it finds the *matches* of the pattern in a given text.

As you've seen in the applications chapter, finding those matches is very useful. For instance, you can extract interesting parts of the text (e.g., the Python

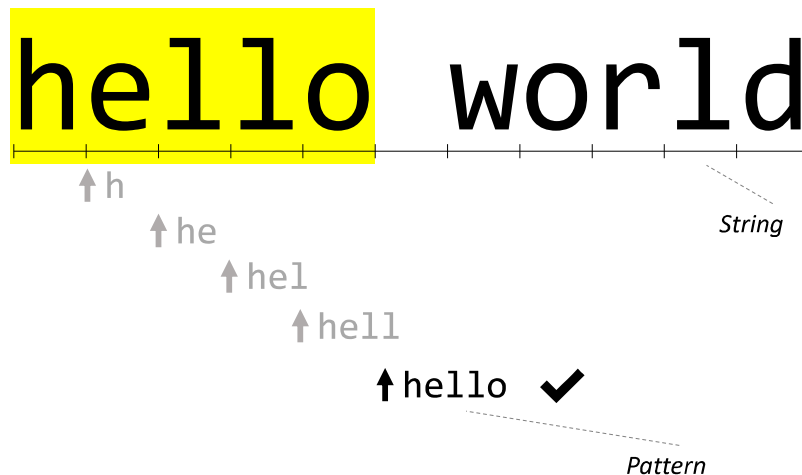


Figure 5.2: Matching the simple pattern 'hello' in the string 'hello world'.

method calls in this book), or even substitute certain parts of the text (e.g., translating Python method calls to, say, C++ method calls).

Throughout this book, we will use the methods of Python's regex library `re`, as a way of practically familiarizing you with regular expressions. From this point on in the book, you'll need a quick introduction to the single most important method of the `re` library. Don't worry, we describe this method here and now.

5.2 The Method to Find All Matches

The `re.findall()` method is the most basic way of using regular expression patterns in Python.

How Does It Work?

The `re.findall(pattern, string, flags=0)` method finds all occurrences of the `pattern` in the `string` and returns a list of all matching substrings. It has up to three arguments.

- **pattern**: the regular expression pattern that you want to match.
- **string**: the string in which you want to search the pattern.
- **flags** (optional argument): a more advanced modifier that allows you to customize the behavior of the function. You can forget about this argument for now. We'll come back to it later in the book.

Return Value: The `re.findall()` method returns a list of strings. Each string element is a matching substring of the string argument. You say, the substring *matches* your `pattern`.

5.2. THE METHOD TO FIND ALL MATCHES 41

Let's check out a few examples.

You import the `re` module and create the text string to be searched for the regex patterns:

```
import re

# Shakespeare
text = '''
    Ha! let me see her: out, alas! he's cold:
    Her blood is settled, and her joints are stiff;
    Life and these lips have long been separated:
    Death lies on her like an untimely frost
    Upon the sweetest flower of all the field.
'''
```

Let's say, you want to search the text for the string `'her'`:

```
>>> re.findall('her', text)
['her', 'her', 'her']
```

The first argument is the pattern. In your case, it's the string `'her'`. The second argument is the text to be analyzed stored as a multi-line string in the variable `text`.

5.3 Summary

- A pattern defines one or more words.
- The set of words defined by your pattern is your personal language.
- If you go through all the trouble creating your personal language, you'd love to see how famous writers such as Shakespeare used *your* words in their texts, right?
- If one of your words appear in the text, you've found a *match*!
- The `re.findall(pattern, text)` method gives you all occurrences of the found pattern in a list of strings.

Video

Watch the short chapter video on YouTube:



https://youtu.be/Lj_QGc7zWUA

Puzzle

Can you solve the puzzle?

Guess the output and check your solution for this puzzle on the Finxter app:



<https://app.finxter.com/learn/computer/science/565>

Special Symbols

Regular expressions are built from characters. There are two types of characters: *literal characters* and *special characters*.

6.1 Literal Characters

Let's start with what you already know: you write regular expressions to match a given pattern in a given string.

Note: We use the term *regular expression* when we talk about the larger field and the term *pattern* when we talk about the concrete matching process. But this is only a preference because these terms are really

interchangeable. A pattern is a regular expression. And a regular expression is a pattern. For example, some coders use the term *pattern matching* when they talk about regular expressions.

In its most basic form, a pattern is a literal character. For example, if you search for the regex pattern 'd' in the string 'hello world', it is matched in the last position. Similarly, a regex pattern with a combination of literal characters, say 'ld', is valid and matches the last two characters in 'hello world'.

Special Characters

The power of regular expressions comes from their abstraction capability. Like you need air to breathe, regular expressions need special symbols for their construction. For instance, instead of writing the character set [abcdefghijklmnopqrstuvwxyz], you'd write [a-z] (see Chapter 7) or even \w (as we will discuss in detail in this chapter). The latter is a special regex character—and every regex master uses it often. In fact, regex experts seldom match literal characters. They often use more advanced constructs or special characters for brevity, expressiveness, or generality.

So what are the special characters you can use in

your regex patterns?

Well, we will introduce them one-by-one now. Note that many of these characters are also available in other regex-supporting languages such as Perl. Thus, if you study the following list carefully, you'll undoubtedly improve your conceptual clarity in writing your own regular expressions—independent of the concrete coding-language you use.

`\n` The newline character is not a special character particular to regular expressions. It's one of the most widely-used, standard characters. However, leaving it out here would definitely leave our list of special characters incomplete. For example, the regex `'hello\nworld'` matches a string where the string `'hello'` is placed in one line and the string `'world'` is placed into the second line.

`\t` Just like the newline character, the tabular character is also not a “regex-specific” character. It just encodes the tabular space `' '` which is different to a sequence of whitespaces (even if it doesn't look different here). For example, the regex `'hello\n\tworld'` matches the string that consists of `'hello'` in the first line and `' world'` in the second line (with a leading tab character).

- `\s` Unlike the newline and the tabular character, the whitespace character is a special symbol of the regex library. You'll find it in many other programming languages, too. It solves an important problem: you often don't know which type of whitespace is used—tabular characters, simple whitespaces, or even newlines. The whitespace character `'\s'` matches any of them. For example, the regex `'\s*hello\s+world'` matches the string `' \t \n hello \n \n \t world'`, as well as `'hello world'`.
- `\S` The whitespace-negation character (with capital 'S') matches everything that does not match `\s`.
- `\w` The word character regex simplifies text processing significantly. It represents the class of all characters used in typical words (A-Z, a-z, 0-9, and '_'). This simplifies the writing of complex regular expressions significantly. For example, the regex `'\w+'` matches the strings `'hello'`, `'bye'`, `'Python'`, and `'Python_is_great'`.
- `\W` The word-character-negation (note the capital 'W'). It matches any character that is not a word character.

- `\b` The word boundary is also a special symbol used in many regex tools. You can use it to match, as the name suggests, the boundary between the a word character (`\w`) and a non-word (`\W`) character. But note that it matches only the empty string! You may ask: why does it exist if it doesn't match any character? The reason is that it doesn't "consume" the character right in front or right after a word. This way, you can search for whole words (or parts of words) and return only the word but not the delimiting characters that separate the words.
- `\d` The digit character matches all numeric symbols between 0 and 9. Use it to match integers with an arbitrary number of digits: the regex `'\d+'` matches integer numbers `'10'`, `'1000'`, `'942'`, and `'999999999999'`.
- `\D` Matches any non-digit character. This is the inverse of `\d` and it's equivalent to `[^0-9]` (see Chapter 7).

These are the most important special symbols and characters. But before you move on to the next chapter, let's try to understand them better by studying some examples!

```
import re

text = '''
    Ha! let me see her: out, alas! he's cold:
    Her blood is settled, and her joints are stiff;
    Life and these lips have long been separated:
    Death lies on her like an untimely frost
    Upon the sweetest flower of all the field.
'''

print(re.findall('\w+\W+\w+', text))
```

Matches each pair of words in the text. Here's the output:

```
['Ha! let', 'me see', 'her: out', 'alas! he',
's cold', 'Her blood', 'is settled', 'and her',
'joints are', 'stiff;\n Life', 'and these',
'lips have', 'long been', 'separated:\n Death',
'lies on', 'her like', 'an untimely',
'frost\n Upon', 'the sweetest', 'flower of', 'all
↪ the']
```

Note that it matches also across new lines: 'stiff;\n Life' also matches!

Note also that matched text is “consumed” and doesn't match again. That's why the combination 'let me' is not a matching substring.

```
print(re.findall('\d', text))
```

No integers in the text:

```
[]
```

The output is empty. What if you search for all occurrences of a newline character, followed by a tab character?

```
print(re.findall('\n\t', text))
```

Match all occurrences where a tab follows a newline:

```
[]
```

No match because each line starts with a sequence of four whitespaces rather than the tab character.

```
print(re.findall('\n ', text))
```

Next, you're going to match all occurrences where 4 whitespaces ' ' follow a newline:

```
['\n ', '\n ', '\n ', '\n ', '\n ']
```

Yes! It matches all five lines.

6.2 Summary

- Regular expressions are built from characters.
- An important class of characters is the class of literal characters. In principle, you can use all Unicode literal characters in your regex pattern.
- Another class of characters is the class of special characters. They make your life much easier, and your patterns more readable.

Video

Watch the short chapter video on YouTube:



<https://youtu.be/hSy0xea-8p8>

Puzzle

```
import re

# Shakespeare
text = ' ' ' ' ' '
```

```
Ha! let me see her: out, alas! he's cold:  
Her blood is settled, and her joints are stiff;  
Life and these lips have long been separated:  
Death lies on her like an untimely frost  
Upon the sweetest flower of all the field.  
...  
  
print(len(re.findall('\s\w\w\s', text)))
```

Can you solve the puzzle?

Guess the output and check your solution for this puzzle on the Finxter app:



<https://app.finxter.com/learn/computer/science/577>

Character Sets

There are many terms describing the same powerful concept: “character class”, “character range”, or “character group”. However, the most precise term is “character set” as introduced in the official Python regex docs. So in this book, we’ll use this term throughout.

7.1 What’s a Character Set?

The best way to describe a character set is, unsurprisingly, as a set of characters. If you use a character set in a regular expression pattern, you tell the regex engine to look for an arbitrary character from that set.

As you may know, a set is an *unordered collection of unique elements*. So each character in a character

set is unique, and the order doesn't matter (with a few minor exceptions).

Here's an example of a character set as used in a regular expression:

```
>>> import re
>>> re.findall('[abcde]', 'hello world!')
['e', 'd']
```

You use the `re.findall(pattern, string)` method to match the pattern `'[abcde]'` in the string `'hello world!'`. You can think of all characters a, b, c, d, and e as being in an OR relation: either of them is a valid match. This OR relation between the characters is represented by the square brackets around them (`[]`)

The regex engine moves from left to right, scanning over the string `'hello world!'` and simultaneously trying to match the character set. Two characters from the text `'hello world!'` are in the character set—they are valid matches and returned by the `re.findall()` method.

You can simplify many character sets by using the range symbol `'-'` that has a special meaning within square brackets: `[a-z]` reads “match any character from

a to z”, while `[0-9]` reads “match any character from 0 to 9”.

Here’s the previous example, simplified:

```
>>> re.findall('[a-e]', 'hello world!')
['e', 'd']
```

You can even combine multiple character ranges in a single character set:

```
>>> re.findall('[a-eA-E0-4]', 'hello WORLD 42!')
['e', 'D', '4', '2']
```

Here, you match three ranges: lowercase characters from 'a' to 'e', uppercase characters from 'A' to 'E', and numbers from '0' to '4'.

Note that the ranges are inclusive, *so both start and stop symbols are included in the range*.

7.2 Negative Character Set

But what if you want to match all characters—except some? You can achieve this with a negative character set. The negative character set works just like a (positive) character set, but with one difference: it matches all characters that are *not* in the character