**Python Regular Expressions**

**Introduction:**

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the [re](https://docs.python.org/3/library/re.html#module-re) module.

The Python module ‘**re’** provides full support for Perl-like regular expressions in Python. The re module raises the exception **re.error** if an error occurs while compiling or using a regular expression.

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern. RegEx can be used to check if a string contains the specified search pattern.

Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as “Does this string match the pattern?”, or “Is there a match for the pattern anywhere in this string?”. You can also use REs to modify a string or to split it apart in various ways. The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that *can* be done with regular expressions, but the expressions turn out to be very complicated.

**Simple Patterns**

We’ll start by learning about the simple possible regular expressions. Since regular expressions are used to operate on strings, we’ll begin with the most common task: matching characters.

For a detailed explanation of the computer science underlying regular expressions (deterministic and non-deterministic finite automata), you can refer to almost any textbook on writing compilers.

### 

### 

### **Matching Characters**

Most letters and characters will simply match themselves. For example, the regular expression **simple** will match the string **simple** exactly. (You can enable a case-insensitive mode that would let this RE match Simple or SIMPLE as well; more about this later.)

There are exceptions to this rule; some characters are special *metacharacters*, and don’t match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them or changing their meaning. Much of this document is devoted to discussing various metacharacters and what they do.

The function below attempts to match **RE** *pattern* to *string* with optional *flags*.

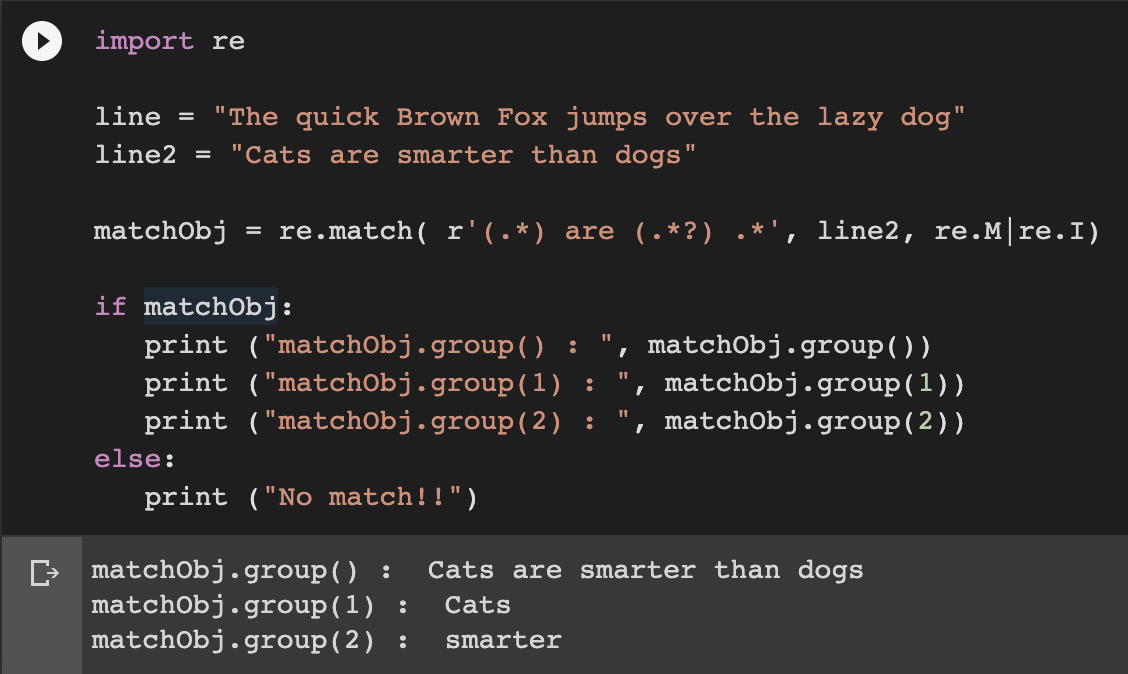
Here is the syntax for this function −

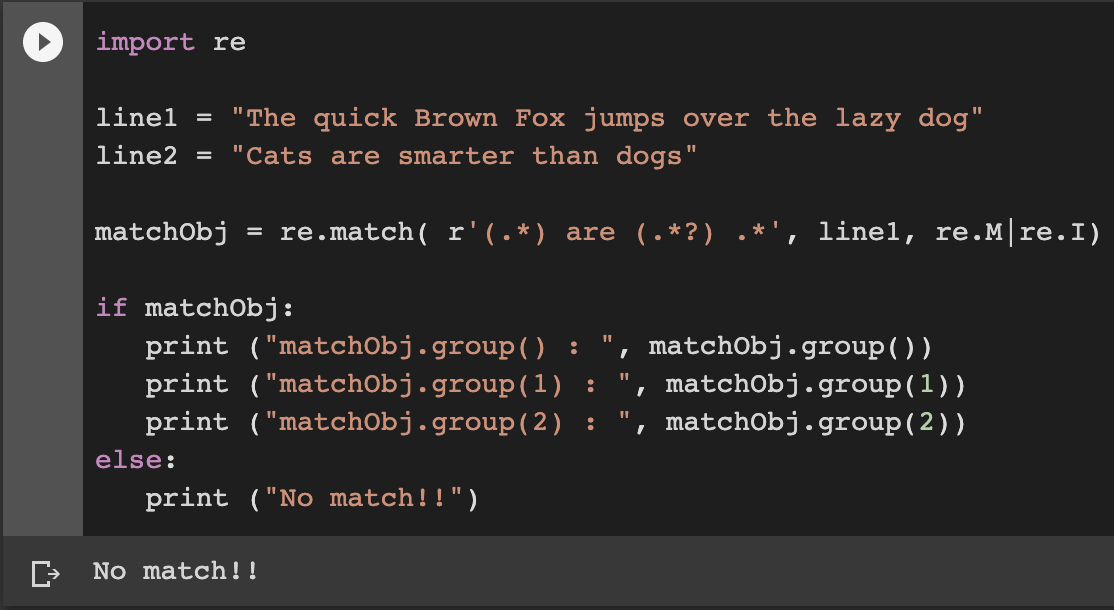
**re.match(pattern, string, flags=0)**

|  |  |
| --- | --- |
| # | Parameter & Description |
| i) | **pattern**  This is the regular expression to be matched. |
| ii) | **string**  This is the string, which would be searched to match the pattern at the beginning of the string. |
| iii) | **flags**  You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below. |

The *re.match* function returns a match object on success, **None** on failure. We use*group(num)* or *groups()* function of the match object to get a matched expression.

|  |  |
| --- | --- |
| **#** | **Match Object Method & Description** |
| 1 | **group(num=0)**  This method returns entire match (or specific subgroup num) |
| 2 | **groups()**  This method returns all matching subgroups in a tuple (empty if there weren't any) |

****

If we were to run the first line for regular expression, the following will be the result, there will be no match.****

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

**import re**

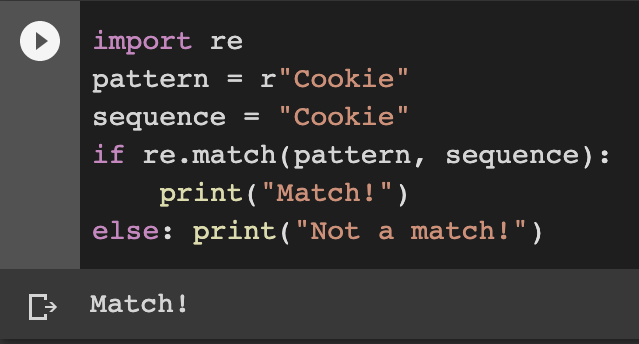
The re library in Python provides several functions that make it a skill worth mastering.

## **Basic Patterns: Ordinary Characters**

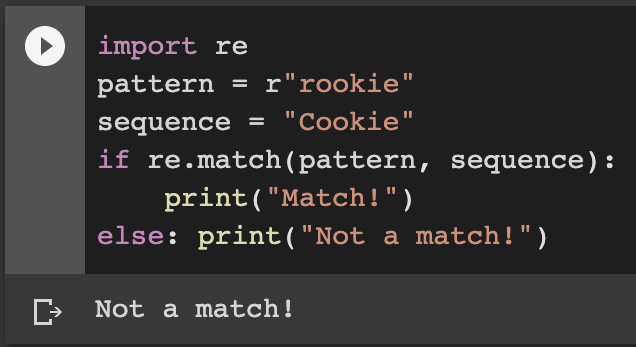
You can easily tackle many basic patterns in Python using ordinary characters. Ordinary characters are the simplest regular expressions. They match themselves exactly and do not have a special meaning in their regular expression syntax.

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

Ordinary characters can be used to perform simple exact matches:



Now change the pattern to the word rookie and let us see what result we get below.



Most alphabets and characters will match themselves, as you saw in the example.

The **match()** function returns a match object if the text matches the pattern. Otherwise, it returns None. The **re** module also contains several other functions, and you will learn some of them later on in this lesson.

Do you notice the **r** at the start of the pattern **Cookie or rookie** in the above examples? This is called a **raw string literal**. It changes how the string literal is interpreted. Such literals are stored as they appear.

For example, **\** is just a backslash when prefixed with an r rather than being interpreted as an escape sequence. You will see what this means with special characters. Sometimes, the syntax involves backslash-escaped characters, and to prevent these characters from being interpreted as escape sequences; we use the raw **r** prefix.

**Note:** *We don't actually need it for this example; however, it is a good practice to use it for consistency.*

## **Wild Card Characters: Special Characters**

Special characters are characters that do not match themselves as seen but have a special meaning when used in a regular expression. For simple understanding, they can be thought of as reserved metacharacters that denote something else and not what they look like.

Let's check out some examples to see the special characters in action…

But before you do, the examples below make use of two functions namely: **search()** and **group()**.

With the [search function](https://www.datacamp.com/community/tutorials/python-regular-expression-tutorial#search), we scan through a given string/sequence, looking for the first location where the regular expression produces a match.

The [group function](https://www.datacamp.com/community/tutorials/python-regular-expression-tutorial#group) returns the string matched by the **re**. You will see both these functions in more detail later.

**Back to the special characters now.**

**1] . - A period. Matches any single character except the newline character.**

**Example:** **re.search(r'Co.k.e', 'Cookie').group()**

**Returns ‘Cookie’**

**2] ^ - A caret. Matches the start of the string.**

This is helpful if we want to make sure a document/sentence starts with certain characters.

**Example:** **re.search(r'^Eat', "Eat cake!").group()**

**Returns ‘Eat’**

**On the contrary, the code below will not yield the same result.**

**## However, the code below will not give the same result. Try it for yourself:**

**# re.search(r'^eat', "Let's eat cake!").group()**

**##This code below will give the same result. Try it for yourself:**

**# re.search(r'^meat', "meat and cake!").group()**

**Returns ‘meat’**

**3] $ - Matches the end of string.**

This is helpful if you want to make sure a document/sentence ends with certain characters.

**Example: re.search(r'cake$', "Cake! Let's eat cake").group()**

**Returns ‘cake’**

**## The next search will return the NONE value, try it:**

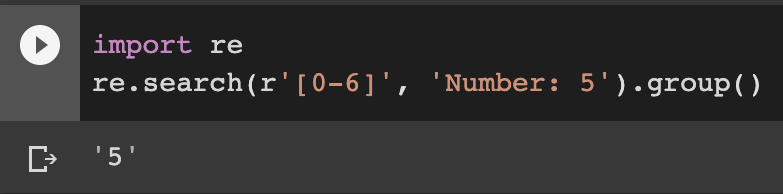
**# re.search(r'cake$', "Let's get some cake on our way home!").group()**

**[abc] - Matches a or b or c.**

**[123] - Matches 1 or 2 or 3.**

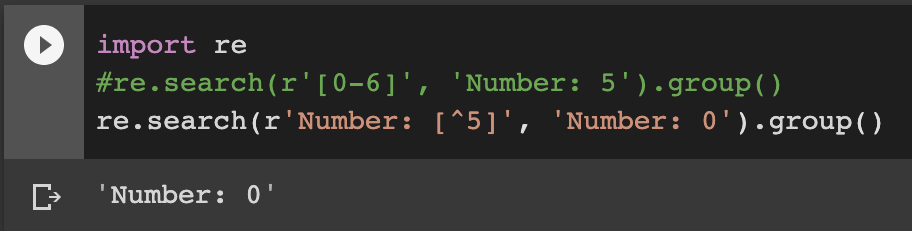
**[a-zA-Z0-9] - Matches any letter from (a to z) or (A to Z) or (0 to 9).**

**Note:** *Characters that are not within a range can be matched by complementing the set. If the first character of the set is ^, all the characters that are not in the set will be matched.*



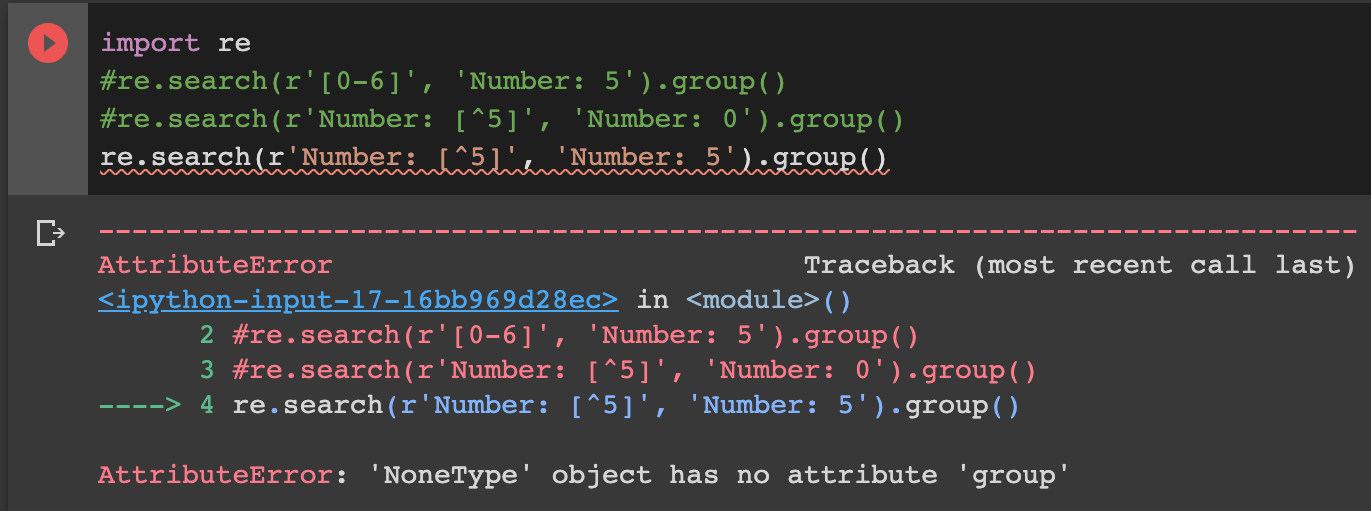
## This below Matches any character except 5

**re.search(r'Number: [^5]', 'Number: 0').group()**

****

## This below will not match and hence a NONE value will be returned

**#re.search(r'Number: [^5]', 'Number: 5').group()**

****

\ - Backslash.

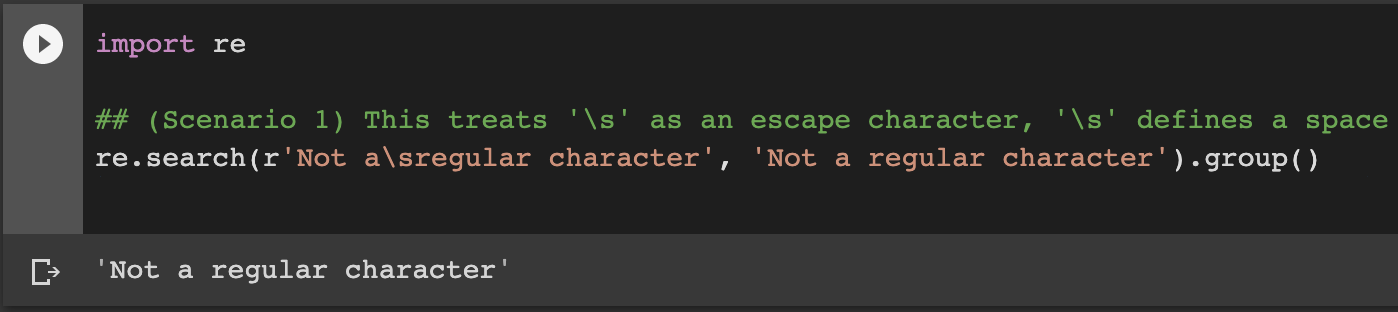
Perhaps, the most diverse metacharacter!!

* If the character following the backslash is a recognized escape character, then the special meaning of the term is taken (Scenario 1)

**## (Scenario 1) This treats '\s' as an escape character, '\s' defines a space**

**re.search(r'Not a\sregular character', 'Not a regular character').group()**

**Result: 'Not a regular character'**

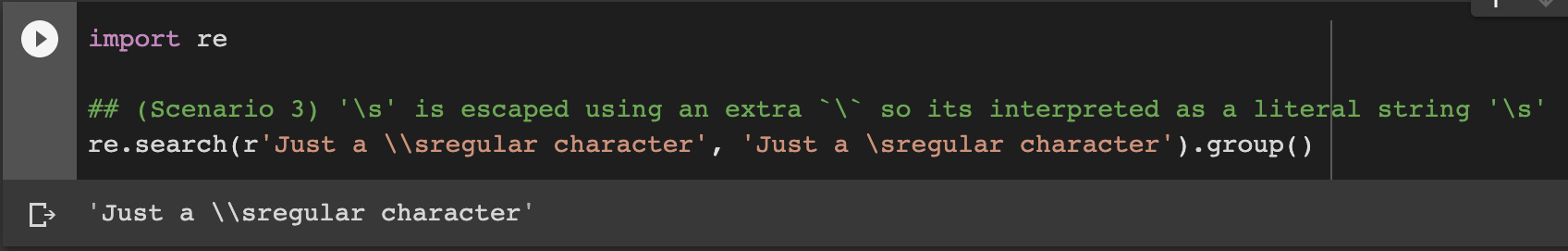
****

* Else if the character following the \ is not a recognized escape character, then the \ is treated like any other character and passed through (Scenario 2).

## (Scenario 2) '\' is treated as an ordinary character, because '\r' is not a recognized escape character

re.search(r'Just a \regular character', 'Just a \regular character').group()

**Result: 'Just a \regular character'**

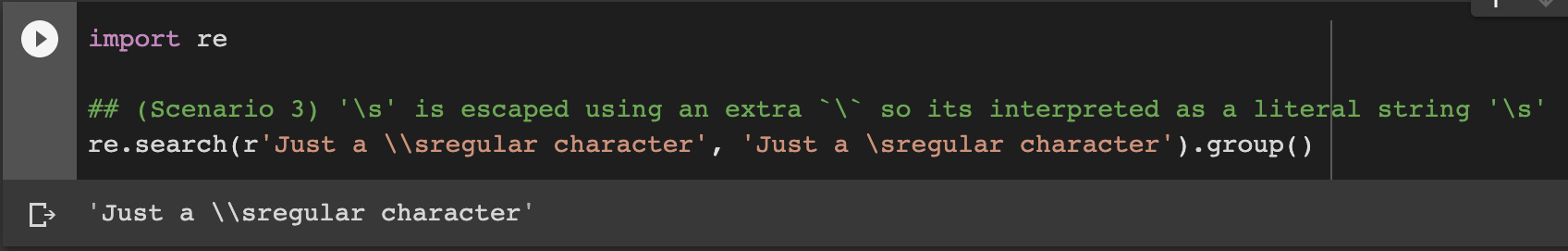
****

* \ can be used in front of all the metacharacters to remove their special meaning (Scenario 3).

**## (Scenario 3) '\s' is escaped using an extra `\` so its interpreted as a literal string '\s'**

**re.search(r'Just a \\sregular character', 'Just a \sregular character').group()**

**Result: 'Just a \\sregular character'**

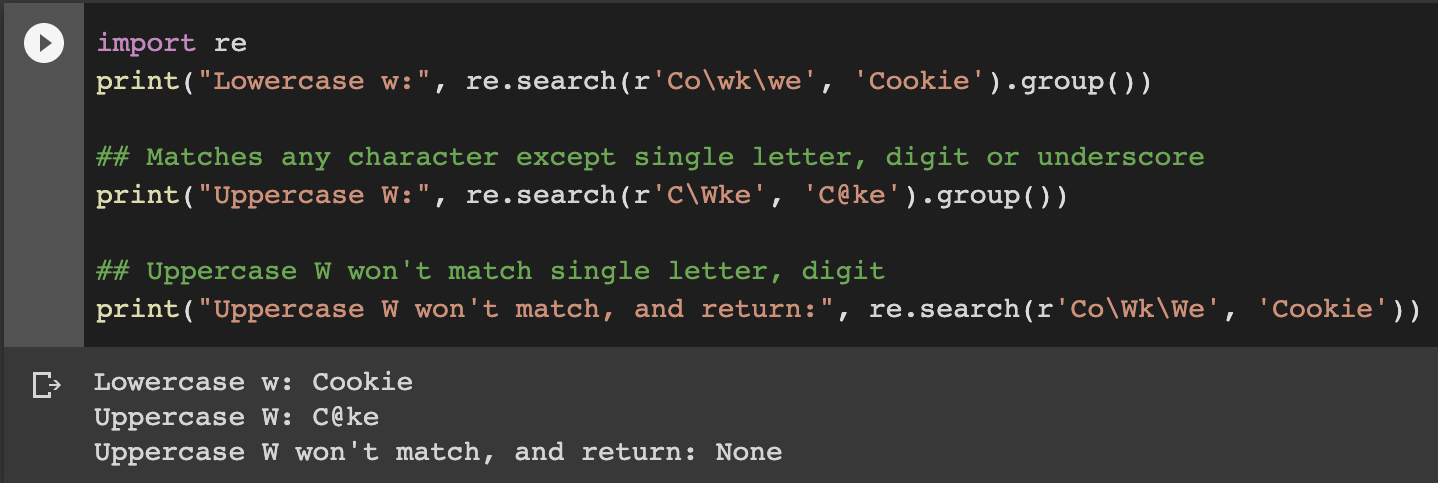
****

There is a predefined set of special sequences that begin with '\' and are also very helpful when performing search and match. Let's look at some of them up close…

**\w** - Lowercase 'w'. Matches any single letter, digit, or underscore. [A-Za-z][0-9][\_]

**\W** - Uppercase 'W'. Matches any character *not* part of \w (lowercase w).

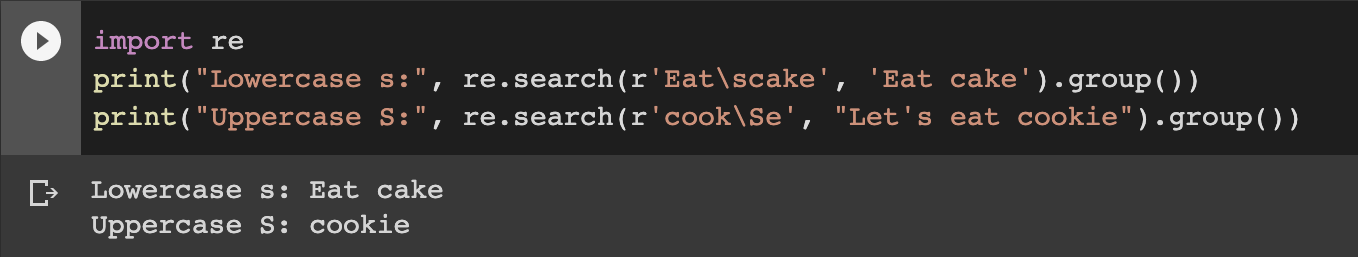
Have a look at the below example of the program when run



**\s** - Lowercase '**s**'. Matches a single whitespace character like: space, newline, tab, return.

**\S** - Uppercase '**S**'. Matches any character *not* part of **\s** (lowercase s).

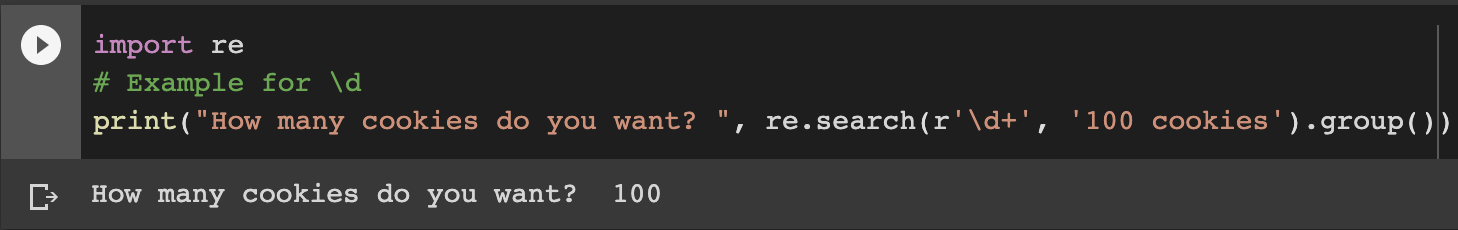
Have a look at the below example of the program when run



**\d** - Lowercase d. Matches decimal digit 0-9.

**\D** - Uppercase d. Matches any character that is *not* a decimal digit.

Have a look at the below example of the program when run



The **+** symbol used after the **\d** in the example above is used for repetition. You will see this in some more detail in the repetition section later on...

**\t** - Lowercase t. Matches tab.

**\n** - Lowercase n. Matches newline.

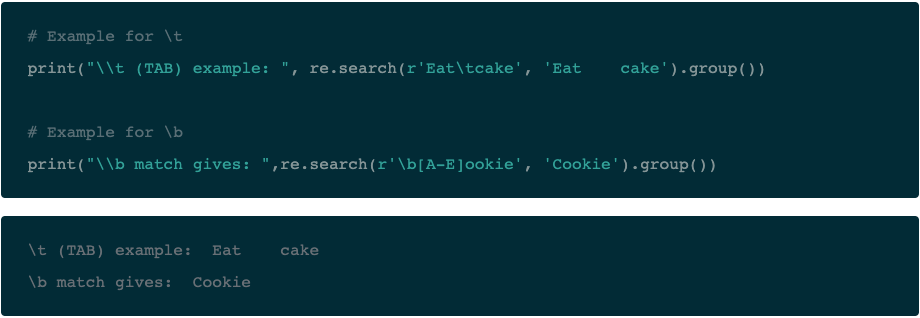
**\r** - Lowercase r. Matches return.

**\A** - Uppercase a. Matches only at the start of the string. Works across multiple lines as well.

**\Z** - Uppercase z. Matches only at the end of the string.

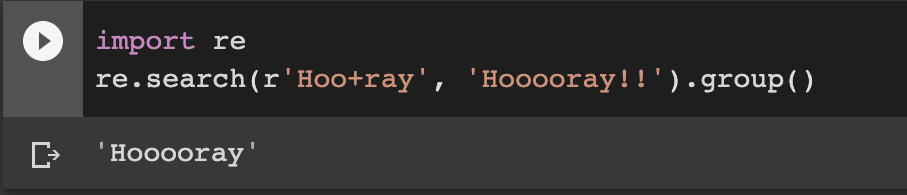
**Note:** **^** and **\A** are effectively the same, and so are **$** and **\Z**. Except when dealing with MULTILINE mode. Learn more about it in the flags section.

**\b** - Lowercase b. Matches only the beginning or end of the word.

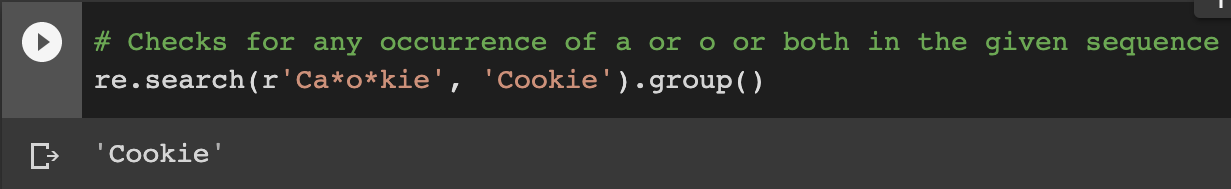


**Repetitions:** It becomes quite monotonous if you are looking to find long patterns in a sequence. Fortunately, the **re** module handles repetitions using the following special characters:

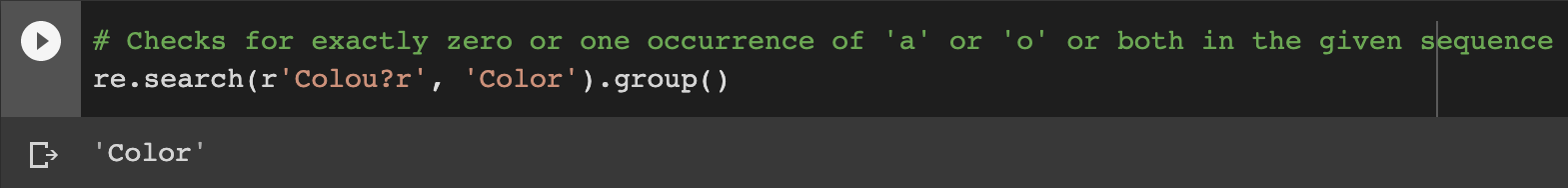
**+** - Checks if the preceding character appears one or more times starting from that position.



**\* -** Checks if the preceding character appears zero or more times starting from that position.



**?** - Checks if the preceding character appears exactly zero or one time starting from that position.



But what if you want to check for an exact number of sequence repetition?

For example, checking the validity of a phone number in an application. re module handles this very gracefully as well using the following regular expressions:

**{x}** - Repeat exactly **x** number of times.

**{x,}** - Repeat at least **x** times or more.

**{x, y}** - Repeat at least **x** times but no more than y times.



The **+** and **\*** qualifiers are said to be greedy. You will see what this means later on.

## **Grouping in Regular Expressions**

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

Let's understand this concept with a simple example. Imagine you were validating email addresses and wanted to check the user **name** and **host**. This is when you would want to create separate groups within your matched text.



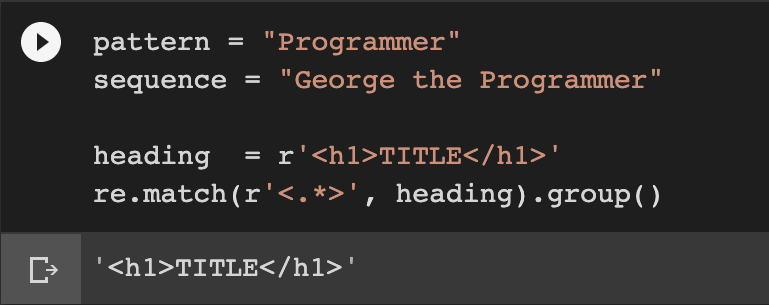
Another way of doing the same is with the usage of **<>** brackets instead. This will let you create **named groups**. Named groups will make your code more readable. The syntax for creating a named group is: **(?P<name>...)**. Replace the **name** part with the name you want to give to your group. The **...** represent the rest of the matching syntax. See this in action using the same example as before…



**Note:** *You can always access the named groups using numbers instead of the name. But as the number of groups increases, it gets harder to handle them using numbers alone. So, always make it a habit to use named groups instead.*

## ***Greedy vs. Non-Greedy Matching***

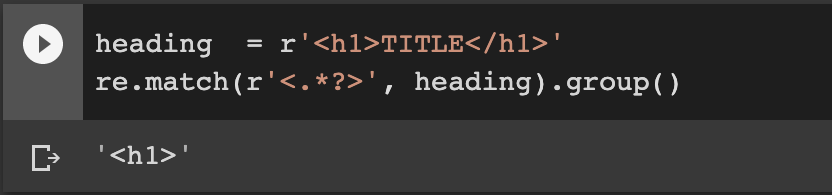
When a special character matches as much of the search sequence (string) as possible, it is said to be a "Greedy Match". It is the normal behavior of a regular expression, but sometimes this behavior is not desired:



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

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

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

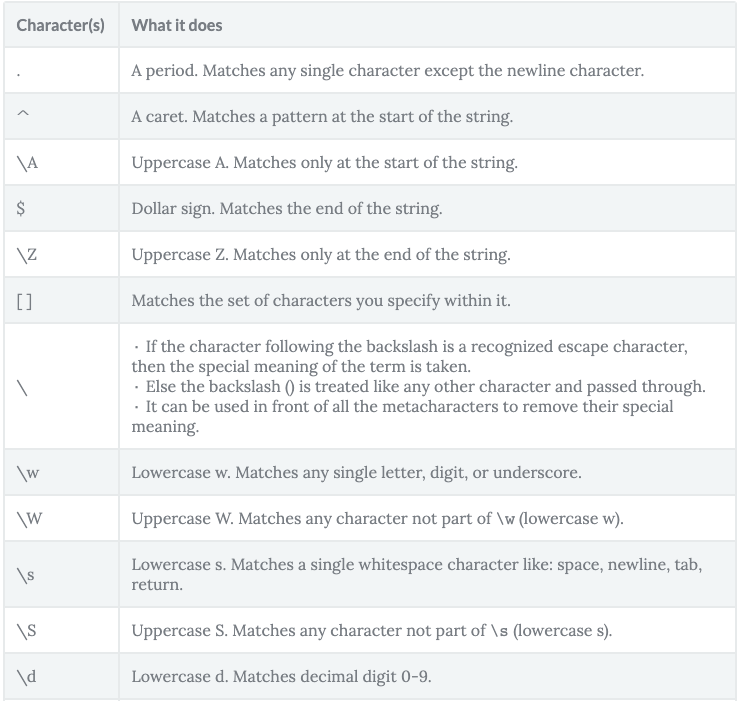


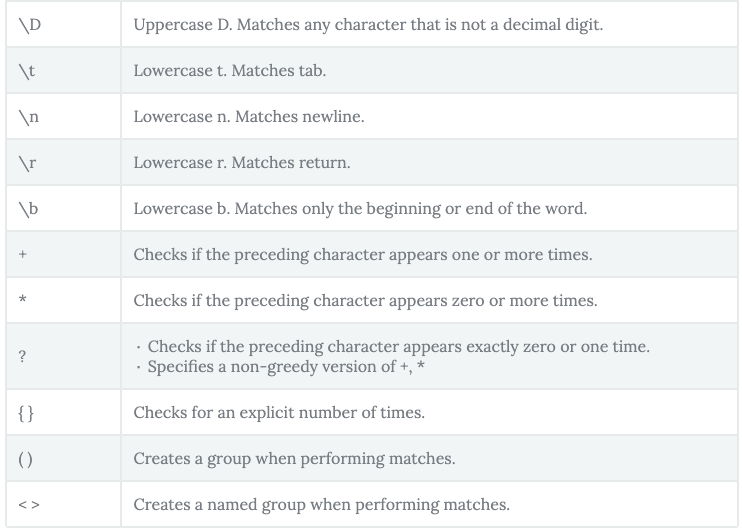
## **Summary table**

We’ve already come a long way with regular expressions. It is a lot of information and concepts to grasp! The following table summarizes all that you've seen so far in this tutorial. Don't worry if you can't wrap your head around all the metacharacters just yet. With time and practice, you will be able to see the uniqueness of these characters and learn when to use what...

This tutorial does not discuss all the special sequences provided in Python. Check out the [Standard Library reference](https://docs.python.org/3/library/re.html#re-syntax) (https://docs.python.org/3/library/re.html#re-syntax) for a complete list.

**Warning:** *Although regular expressions are very powerful and helpful, be wary of long, confusing expressions that are hard for others, and also you to understand and maintain over time.*



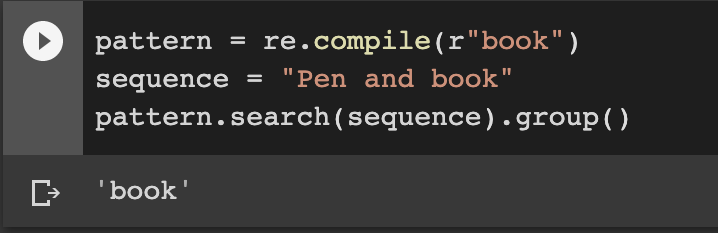


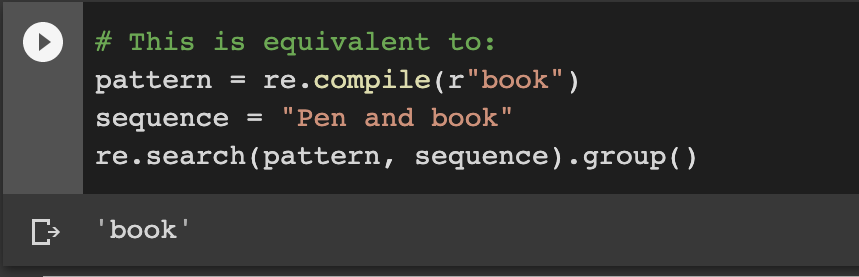
## **Function provided by 're'**

The re library in Python provides several functions to make your tasks easier. You have already seen some of them, such as the re.search(), re.match(). Let's check out more...

**compile(pattern, flags=0)**

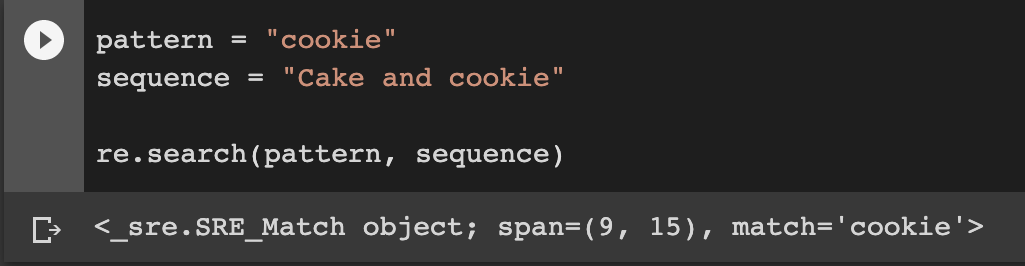
Regular expressions are handled as strings by Python. However, with **compile()**, you can computer a regular expression pattern into a [regular expression object](https://docs.python.org/3/library/re.html#re-objects). When you need to use an expression several times in a single program, using **compile()** to save the resulting regular expression object for reuse is more efficient than saving it as a string. This is because the compiled versions of the most recent patterns passed to **compile()** and the module-level matching functions are cached.





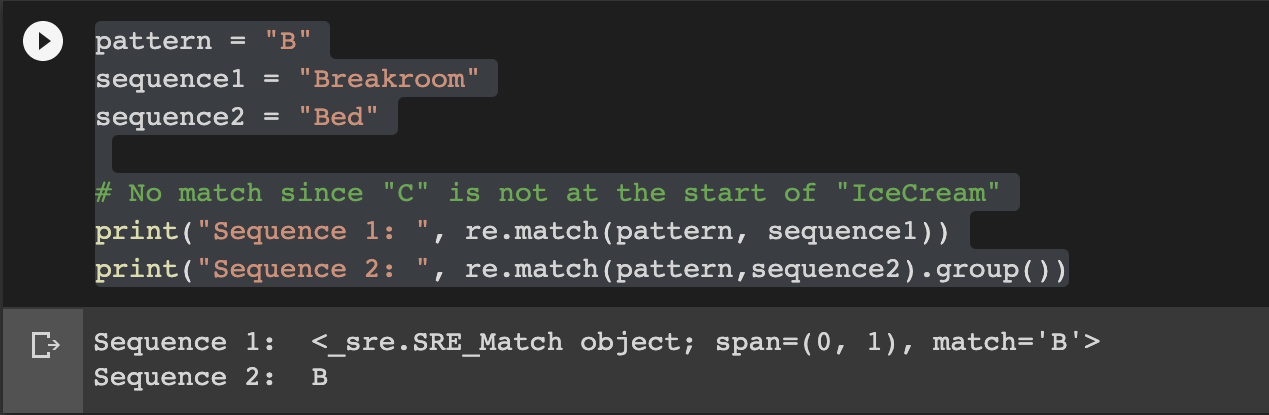
**search(pattern, string, flags=0)**

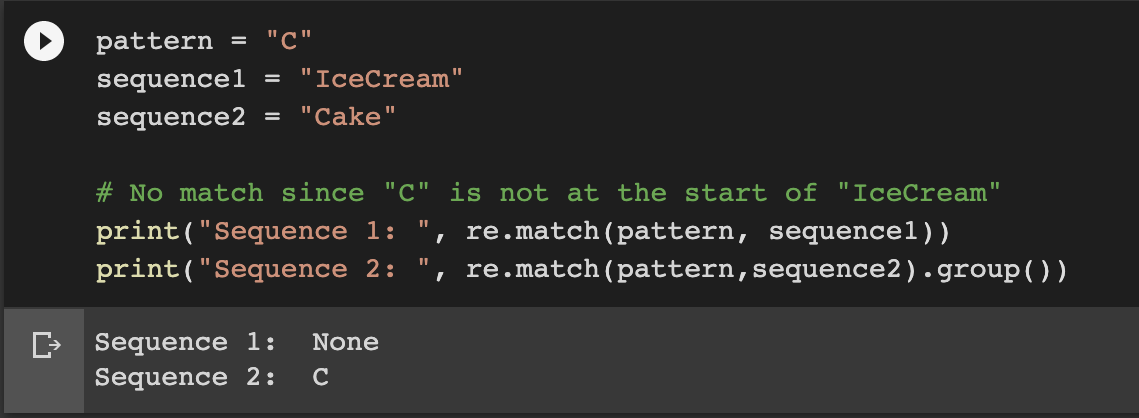
With this function, you scan through the given string/sequence, looking for the first location where the regular expression produces a match. It returns a corresponding match object if found, else returns **None** if no position in the string matches the pattern. Note that **None** is different from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



**match(pattern, string, flags=0)**

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



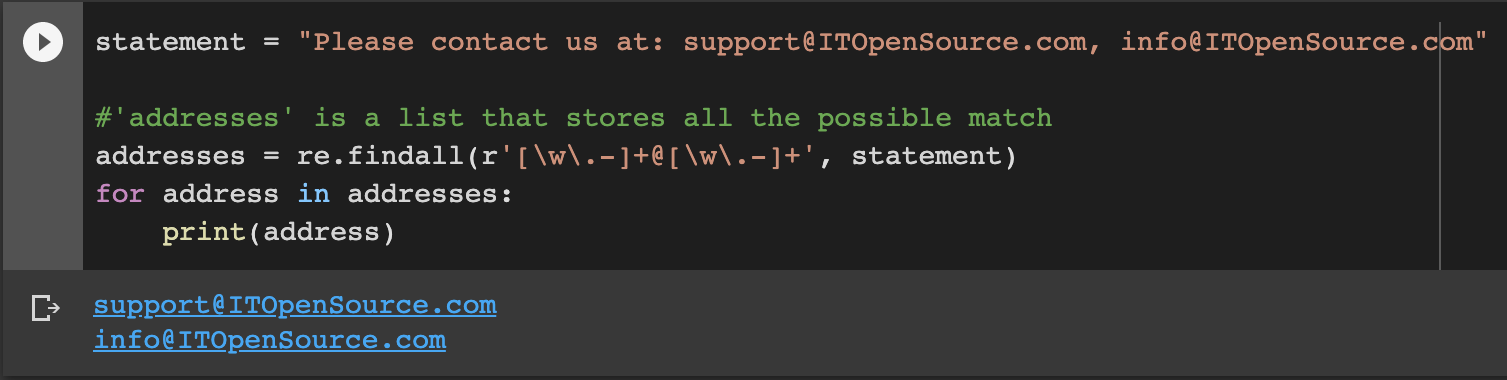


**search() versus match()**

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

**findall(pattern, string, flags=0)**

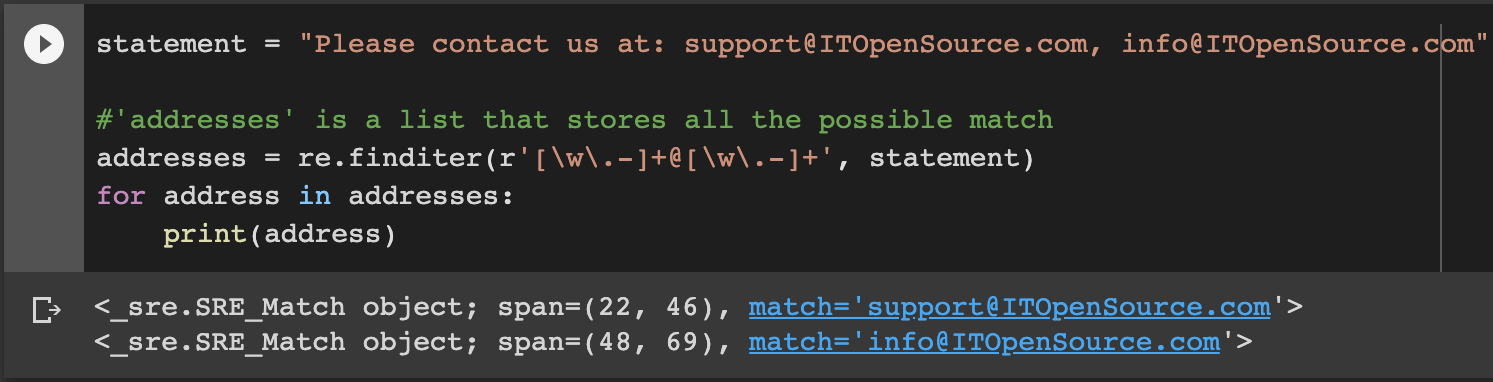
Finds all the possible matches in the entire sequence and returns them as a list of strings. Each returned string represents one match.



**finditer(string, [position, end\_position])**

Similar to **findall()** - it finds all the possible matches in the entire sequence but returns regex match objects as an iterator.

**Note:** **finditer()** might be an excellent choice when you want to have more information returned to you about your search. The returned regex match object holds not only the sequence that matched but also their positions in the original text.

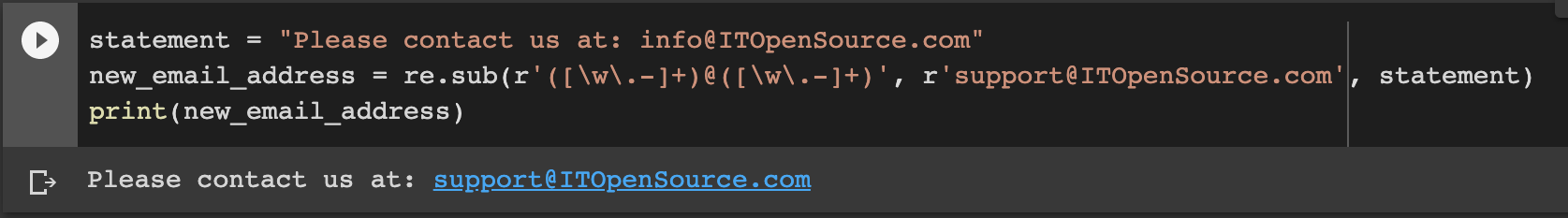


**sub(pattern, repl, string, count=0, flags=0)**

**subn(pattern, repl, string, count=0)**

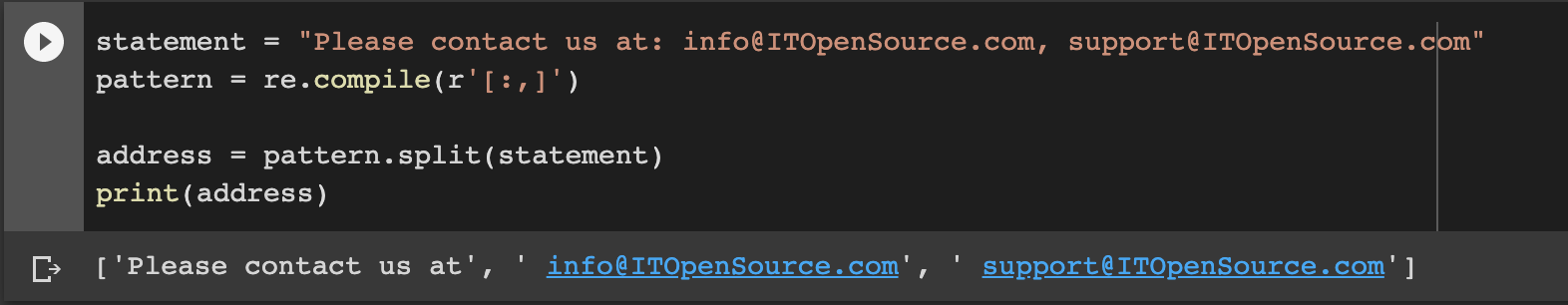
**sub()** is the **substitute** function. It returns the string obtained by replacing or substituting the leftmost non-overlapping occurrences of pattern in string by the replacement **repl**. If the pattern is not found, then the string is returned unchanged.

The **subn()** is similar to **sub()**. However, it returns a tuple containing the new string value and the number of replacements that were performed in the statement.



**split(string, [maxsplit = 0])**

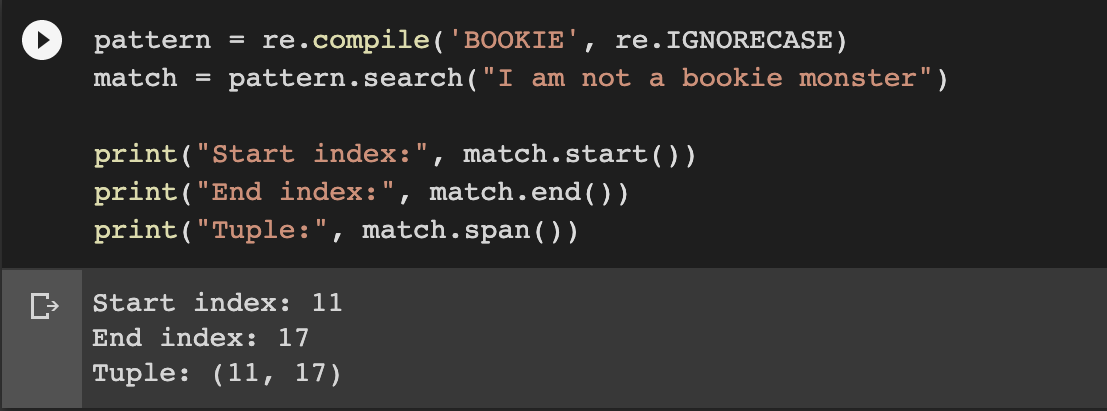
This splits the strings wherever the pattern matches and returns a list. If the optional argument maxsplit is nonzero, then the maximum 'maxsplit' number of splits are performed.



**start()** - Returns the starting index of the match.

**end()** - Returns the index where the match ends.

**span()** - Return a tuple containing the (start, end) positions of the match.



## **Compilation Flags**

Did you notice the term re.IGNORECASE in the last example? Did you figure out its importance?

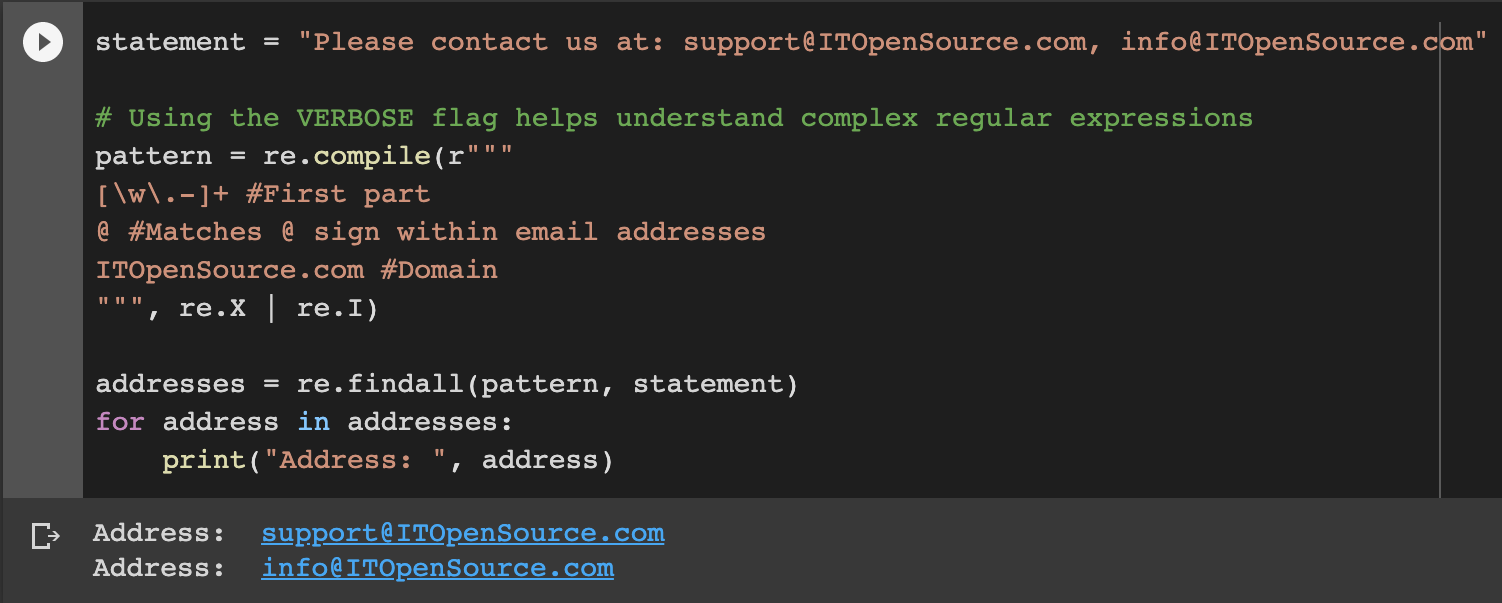
An expression's behavior can be modified by specifying a flag value. You can add flags as an extra argument to the different functions that you have seen in this tutorial. Some of the more useful ones are:

**IGNORECASE (I)** - Allows case-insensitive matches.

**DOTALL (S)** - Allows . to match any character, including newline.

**MULTILINE (M)** - Allows start of string (^) and end of string ($) anchor to match newlines as well.

**VERBOSE (X)** - Allows you to write whitespace and comments within a regular expression to make it more readable.



**TIP:** *You can also combine multiple flags by using bitwise OR |*

Compiled regular expression objects support the following methods and attributes:

Pattern.**search**(*string*[, *pos*[, *endpos*]])

Scan through *string* looking for the first location where this regular expression produces a match, and return a corresponding match object. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional second parameter *pos* gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the '^' pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter *endpos* limits how far the string will be searched; it will be as if the string is *endpos* characters long, so only the characters from *pos* to endpos - 1 will be searched for a match. If *endpos* is less than *pos*, no match will be found; otherwise, if *rx* is a compiled regular expression object, **rx.search(string, 0, 50)** is equivalent to **rx.search(string[:50], 0)**.

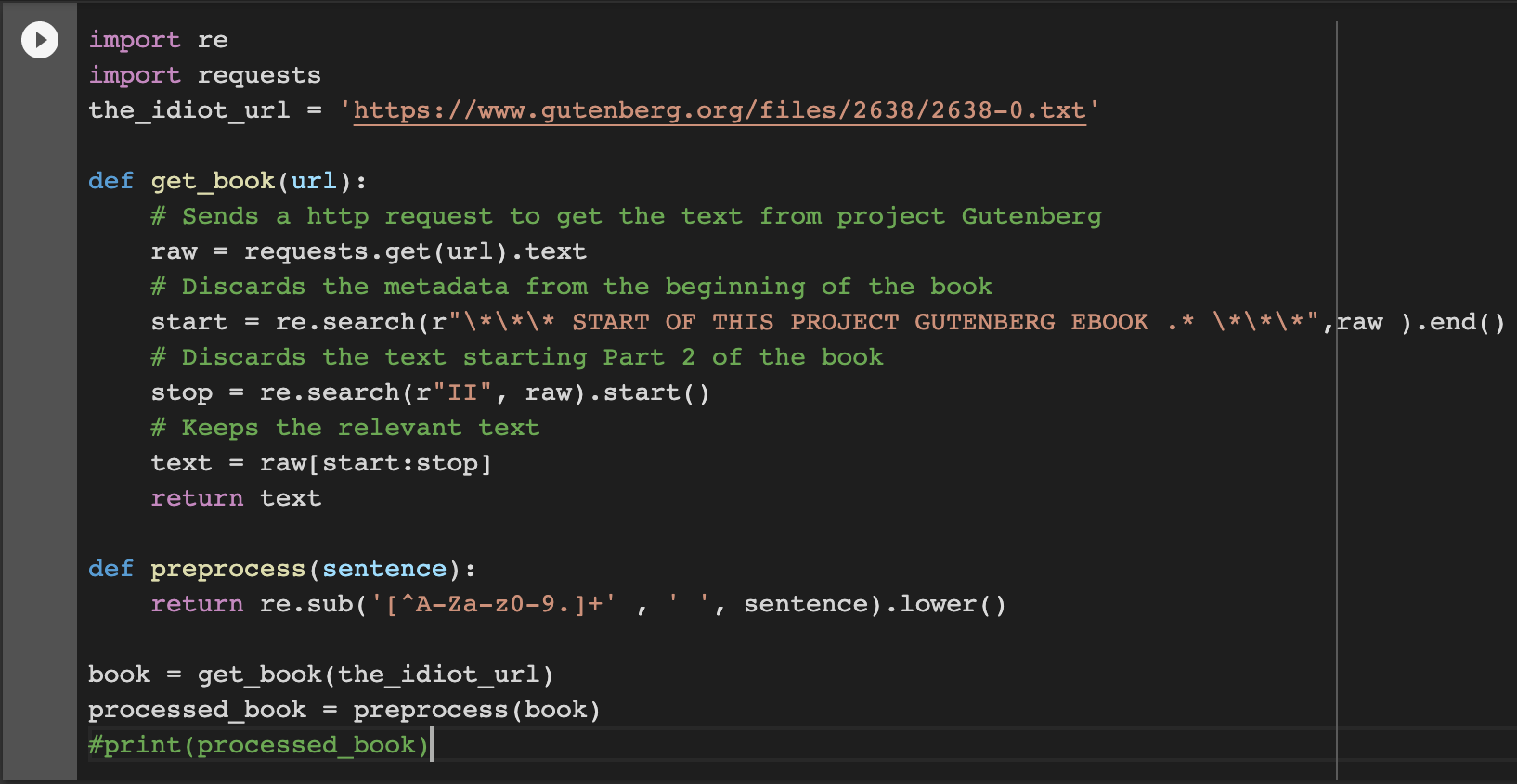
## **Case Study: Working with Regular Expressions**

Now that you have seen how regular expressions work in Python by studying some examples, it's time to get your hands dirty! In this case study, you'll put all your knowledge to work.

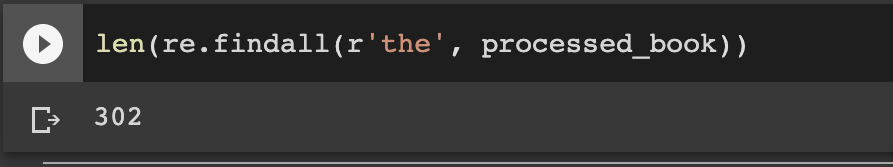
You will work with the first part of a free e-book titled "[The Idiot](https://www.gutenberg.org/files/2638/2638-0.txt)", written by Fyodor Dostoyevsky from the Project Gutenberg. The novel is about Prince (Knyaz) Lev Nikolayevich Myshkin, a guileless man whose good, kind, simple nature mistakenly leads many to believe he lacks intelligence and insight. The title is an ironic reference to this young man.

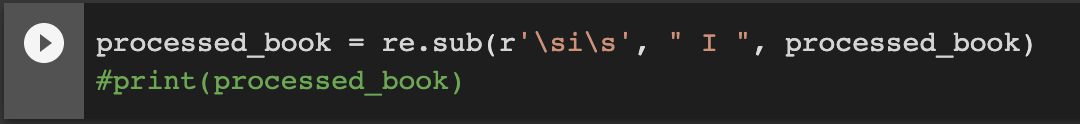
You shall be writing some regular expressions to parse through the text and complete some exercises.

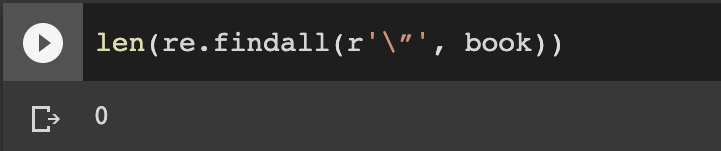
**Url:** <https://www.gutenberg.org/files/2638/2638-0.txt>



* **Exercise:** Find the number of the pronoun "the" in the corpus.
* **Hint:** Use the len() function.
* **Solution:** **len(re.findall(r'the', processed\_book))**

****

* **Exercise:** Try to convert every single stand-alone instance of 'i' to 'I' in the corpus. Make sure not to change the 'i' occurring within a word:
* **Exercise**: Find the number of times anyone was quoted ("") in the corpus.

****

* **Exercise:** What are the words connected by '**--**' in the corpus?  
  *Try this out yourself! Feel free to share your answer in the comments below.*