UNIT-3

UNIT-3

Regular Expressions: Introduction, Special Symbols and Characters, Res and Python

Multithreaded Programming: Introduction, Threads and Processes, Python, Threads, and the Global Interpreter Lock, Thread Module, Threading Module, Related Modules

Regular Expressions:

Introduction

✓The regular expressions can be defined as the **sequence of characters which are used to search for a pattern in a string.**

✓It is extremely **useful for extracting information from text** such as code, files, log, spreadsheets or even documents.

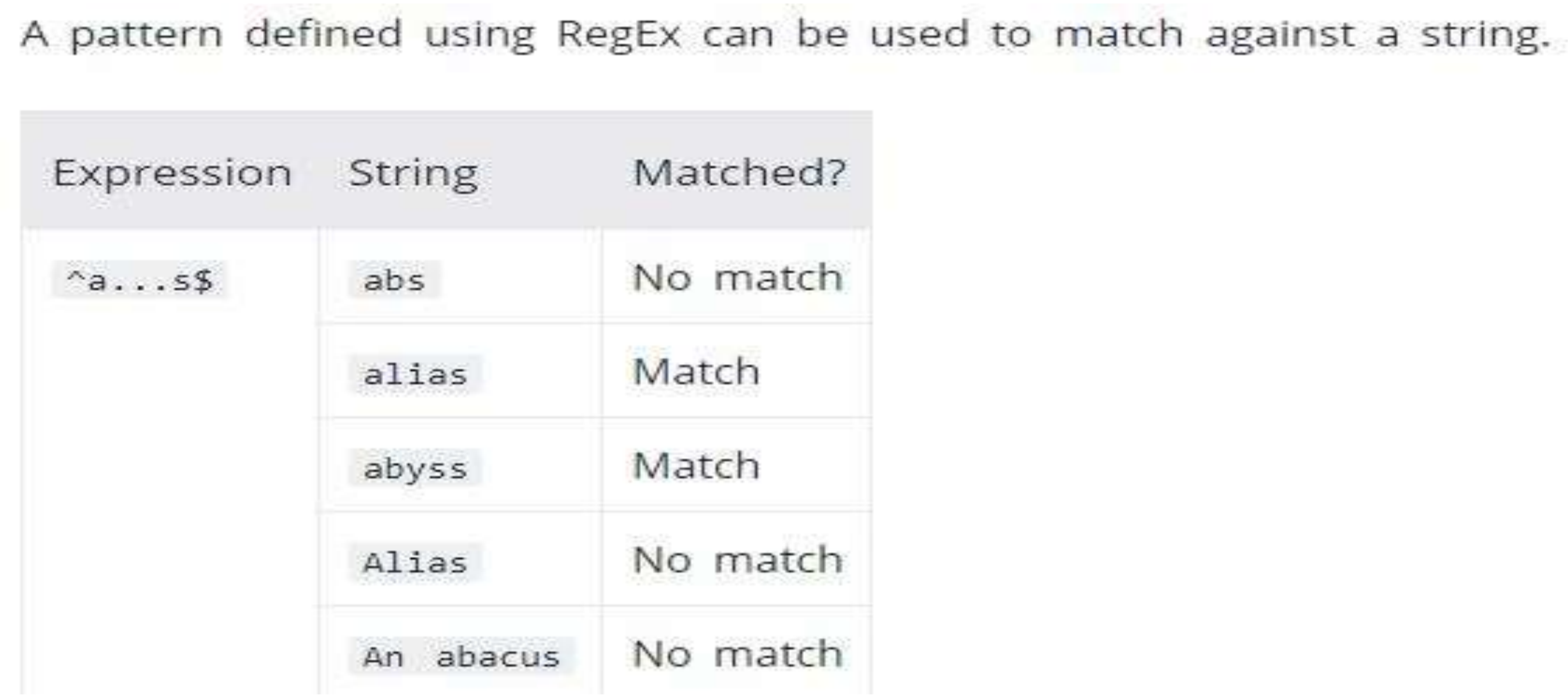
✓While using the regular expression the first thing is to recognize is that **everything is a character, and we are writing patterns to match a specific sequence of characters** also referred as **string.**

✓It is also called **RegEx or re.**

For example,

^a...s$

- ✓The above code defines a **RegEx pattern.**
- ✓ The pattern is: **any five letter string starting with** a **and ending with** s.

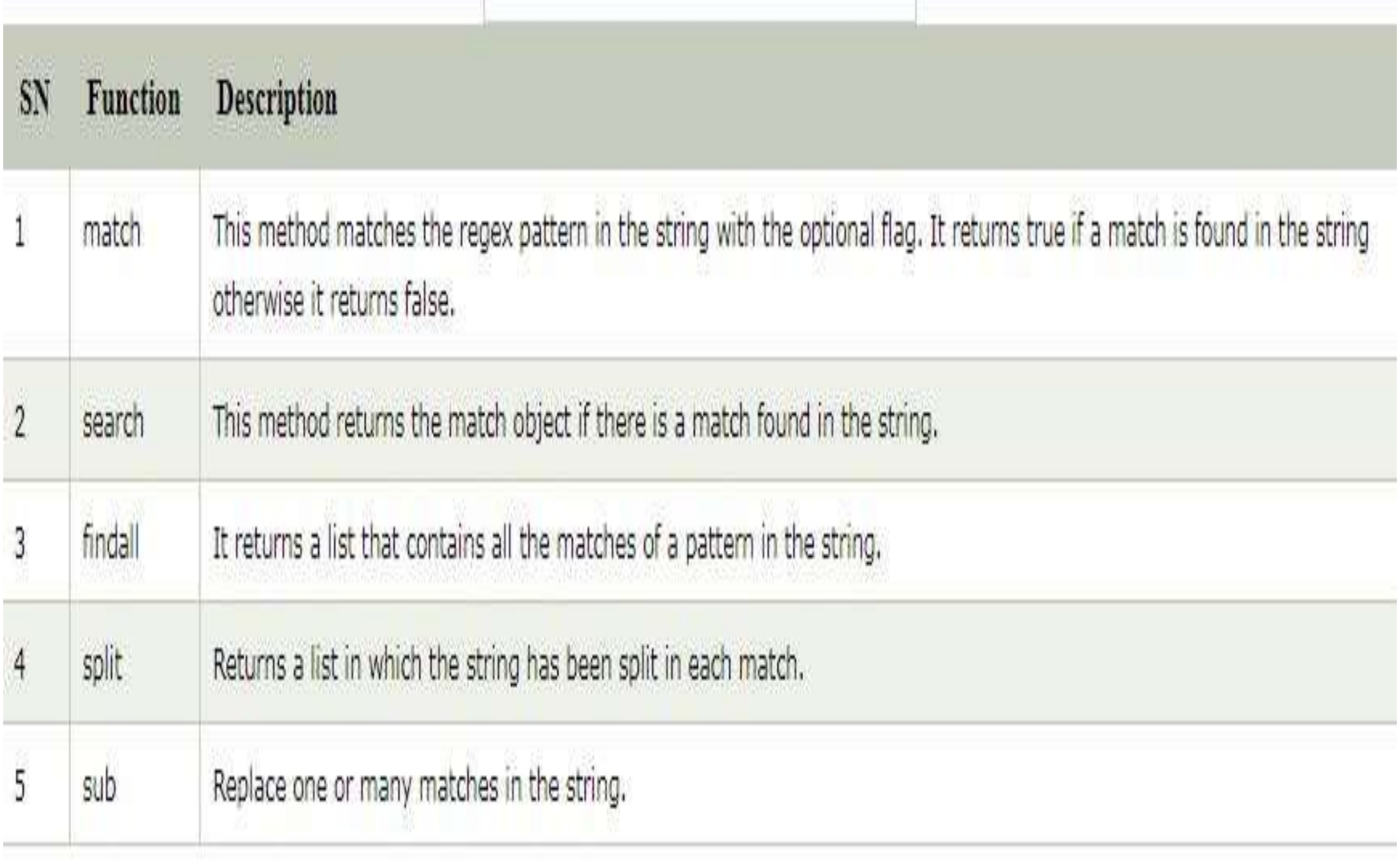


A pattern defined using RegEx can be used to match against a string.

Expression	String	Matched?
`^a...s$`	`abs`	No match
	`alias`	Match
	`abyss`	Match
	`Alias`	No match
	`An abacus`	No match

Regex Functions

The following regex functions are used in the python.



SN	Function	Description
1	match	This method matches the regex pattern in the string with the optional flag. It returns true if a match is found in the string otherwise it returns false.
2	search	This method returns the match object if there is a match found in the string.
3	findall	It returns a list that contains all the matches of a pattern in the string.
4	split	Returns a list in which the string has been split in each match.
5	sub	Replace one or many matches in the string.

1. re.match():

✓ This method matches the regex pattern in the string .It returns true if a match is found in the string otherwise it returns false.

```
import re
pattern = '^a...s$'
test_string = 'abyss'
result = re.match(pattern, test_string)
if result:
  print("Search successful.")
else:
  print("Search unsuccessful.")
```

Here, we used **re.match()** function to search pattern within the test_string. The method returns a match object if the search is successful. If not, it returns None.

2. re.search()

✓The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

✓If the search is successful, re.search() returns a match object; if not, it returns None.

Syntax:

```
match = re.search(pattern, str)
```

Ex:

```
import re
string = "Python is fun“
match = re.search(‘fun', string)
if match:
  print("pattern found inside the string")
else:
  print("pattern not found")
```

3. The findall() function

✓This method returns a **list containing a list of all matches of a pattern within the string.**

✓ It returns the patterns in the order they are found. If there are no matches, then an empty list is returned.

Example

```
>>> import re
>>> str="How are you. How is everything"
>>> matches=re.findall("How",str)
>>> print(matches)
['How', 'How']
```

4. re.split(*pattern*, *string*, [*maxsplit=0*]):

This methods helps to split *string* by the occurrences of given *pattern*.

Ex:

```
import re
result=re.split('y','Analytics')
Result
```

Output:

```
['Anal', 'tics']
```

ex:

```
>>> result=re.split('i','Analytics information',maxsplit=3)
>>> result
['Analyt', 'cs ', 'nformat', 'on']
```

5. re.sub(*pattern*, *repl*, *string*):

✓It helps to search a pattern and replace with a new sub string. If the pattern is not found, *string* is returned unchanged.

<u>Ex;</u>

```
import re
result=re.sub('India','the World','AV is largest Analytics community of India')
result
```

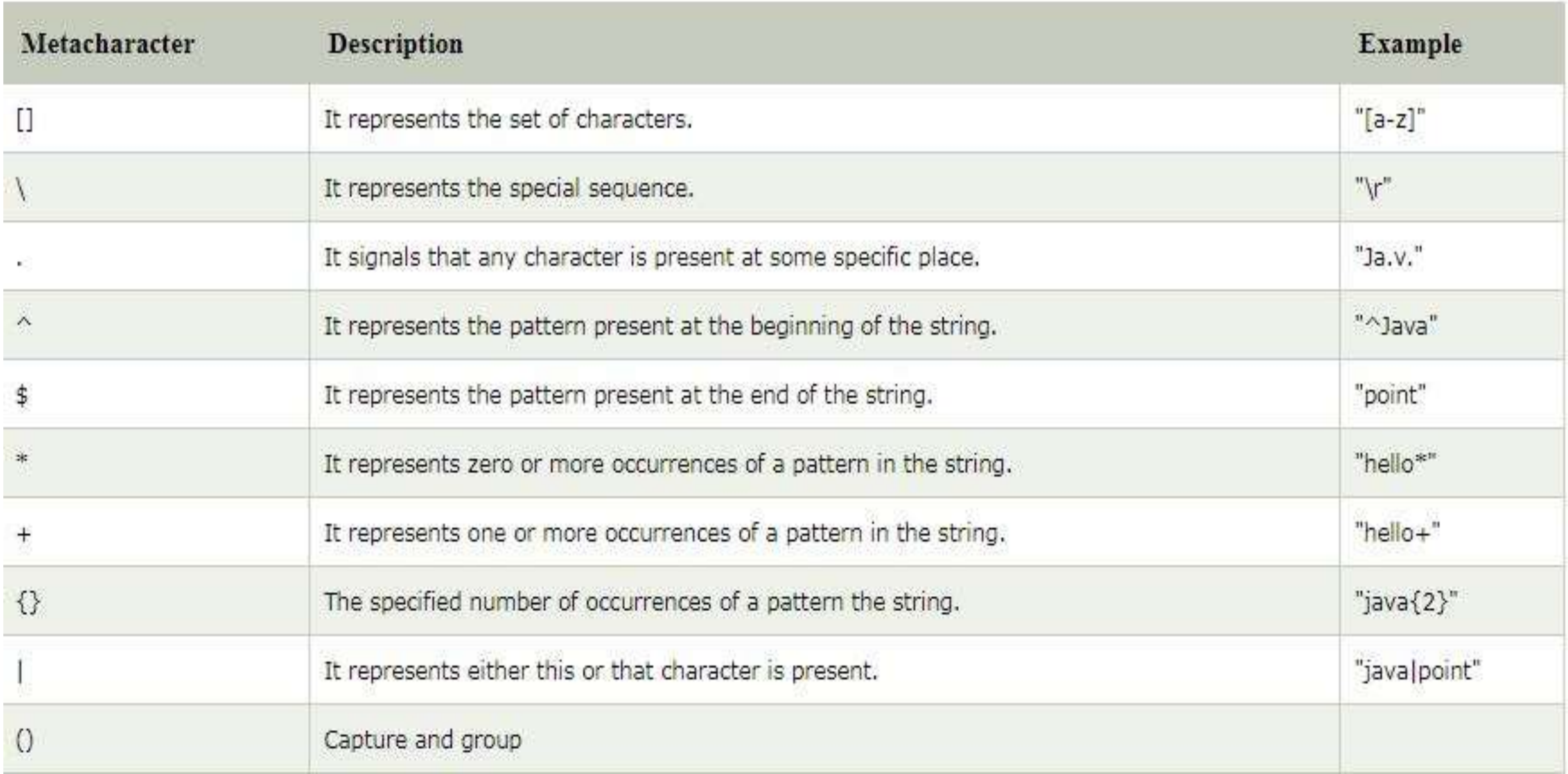
Output: 'AV is largest Analytics community of the World'

2. Special Symbols and Characters:

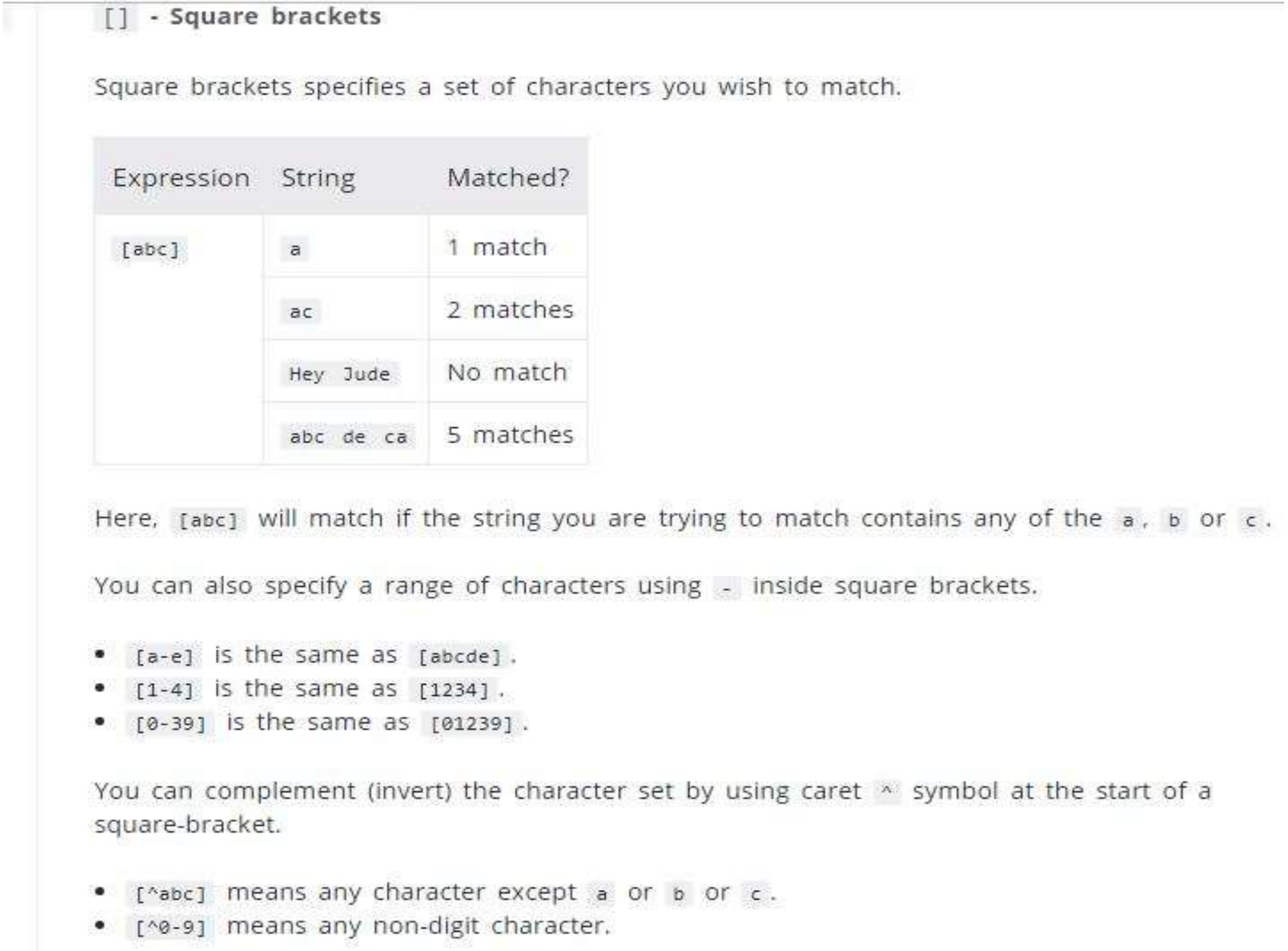
Forming a regular expression

✓A regular expression can be formed by using the mix of **meta-characters, special sequences, and sets.**

1. Meta-Characters



Metacharacter	Description	Example
[]	It represents the set of characters.	"[a-z]"
\	It represents the special sequence.	"\r"
.	It signals that any character is present at some specific place.	"Ja.v."
^	It represents the pattern present at the beginning of the string.	"^Java"
$	It represents the pattern present at the end of the string.	"point"
*	It represents zero or more occurrences of a pattern in the string.	"hello*"
+	It represents one or more occurrences of a pattern in the string.	"hello+"
{}	The specified number of occurrences of a pattern the string.	"java{2}"
|	It represents either this or that character is present.	"java|point"
()	Capture and group	



`[]` - Square brackets

Square brackets specifies a set of characters you wish to match.

Expression	String	Matched?
`[abc]`	`a`	1 match
	`ac`	2 matches
	`Hey Jude`	No match
	`abc de ca`	5 matches

Here, `[abc]` will match if the string you are trying to match contains any of the `a`, `b` or `c`.

You can also specify a range of characters using `-` inside square brackets.

- `[a-e]` is the same as `[abcde]`.
- `[1-4]` is the same as `[1234]`.
- `[0-39]` is the same as `[01239]`.

You can complement (invert) the character set by using caret `^` symbol at the start of a square-bracket.

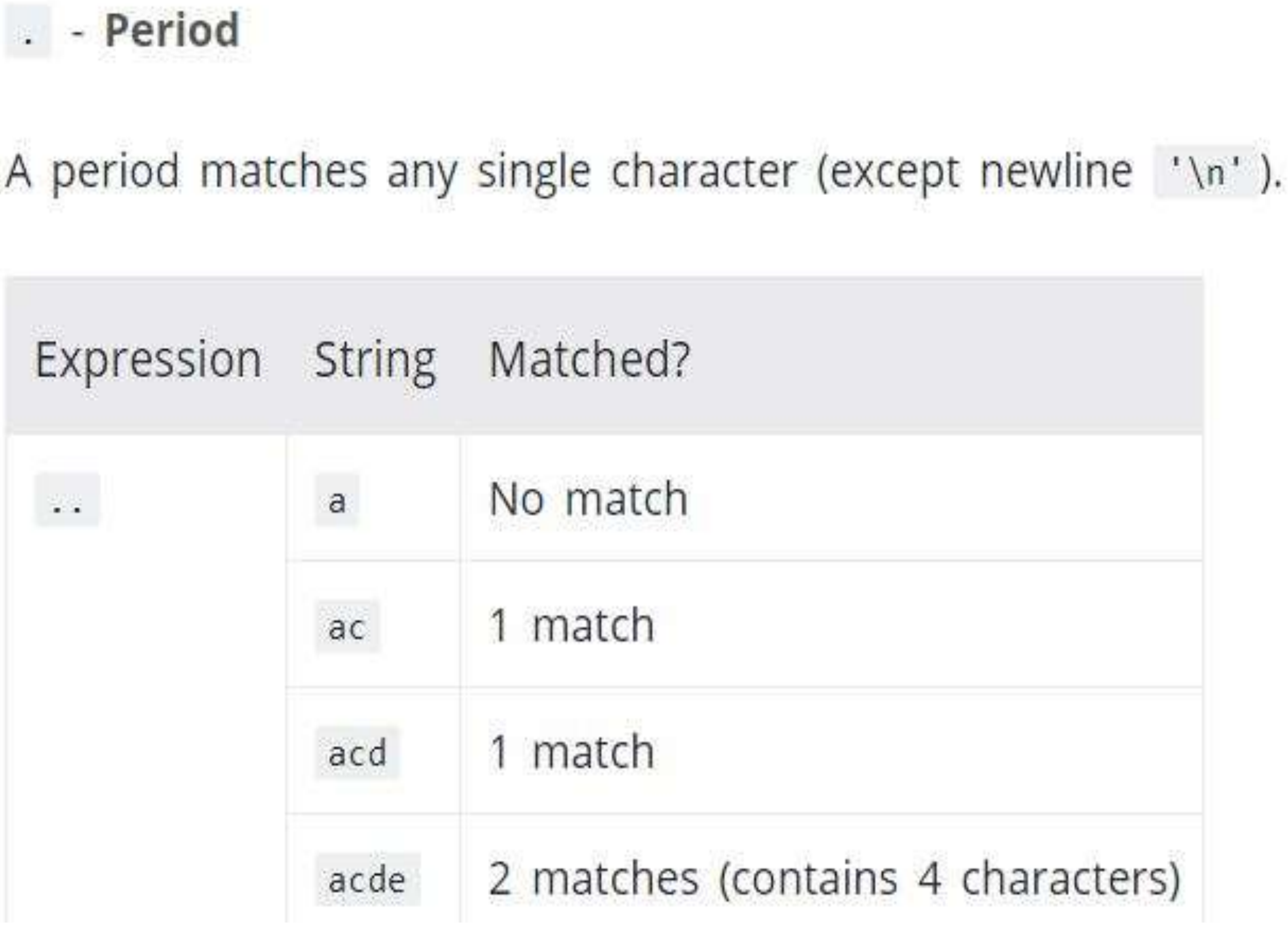
- `[^abc]` means any character except `a` or `b` or `c`.
- `[^0-9]` means any non-digit character.

EX:

```
import re
match=re.match('[abc]','a')
if match:
  print("matched")
else:
  print("not matched")
```

```
import re
match=re.match('[a-e]','a')
if match:
  print("matched")
else:
  print("not matched")
```

```
import re
match=re.match('[^a-e]','a')
if match:
  print("matched")
else:
  print("not matched")
```



`.` - Period

A period matches any single character (except newline `'\n'`).

Expression	String	Matched?
`..`	`a`	No match
	`ac`	1 match
	`acd`	1 match
	`acde`	2 matches (contains 4 characters)

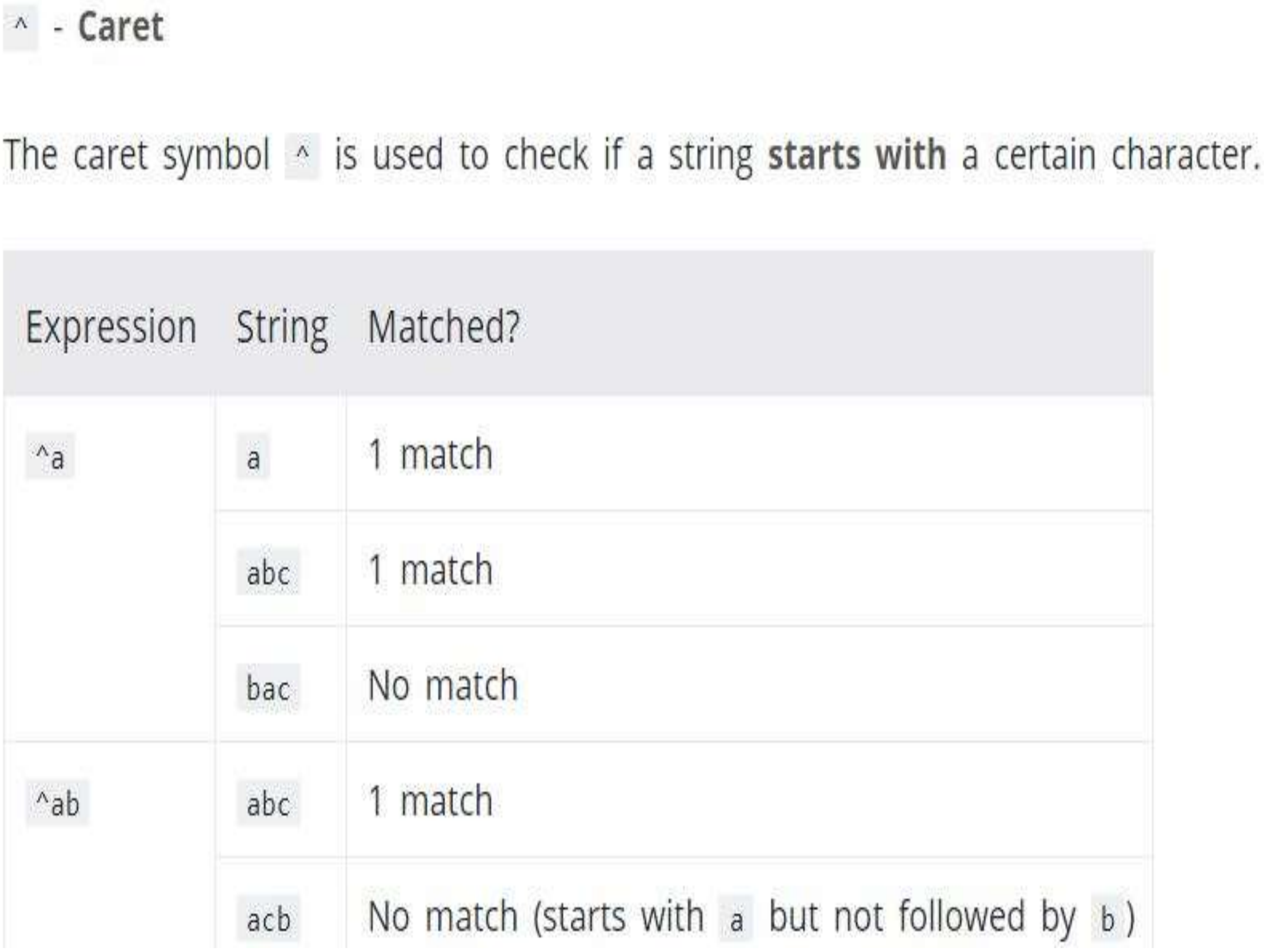
Ex:

```
import re
match=re.match('..',‘a')
if match:
  print("matched")
else:
  print("not matched“)
```

Output: not matched

```
import re
match=re.match('..','ac')
if match:
  print("matched")
else:
  print("not matched")
```

Output: matched



`^` - Caret

The caret symbol `^` is used to check if a string **starts with** a certain character.

Expression	String	Matched?
`^a`	`a`	1 match
	`abc`	1 match
	`bac`	No match
`^ab`	`abc`	1 match
	`acb`	No match (starts with `a` but not followed by `b`)

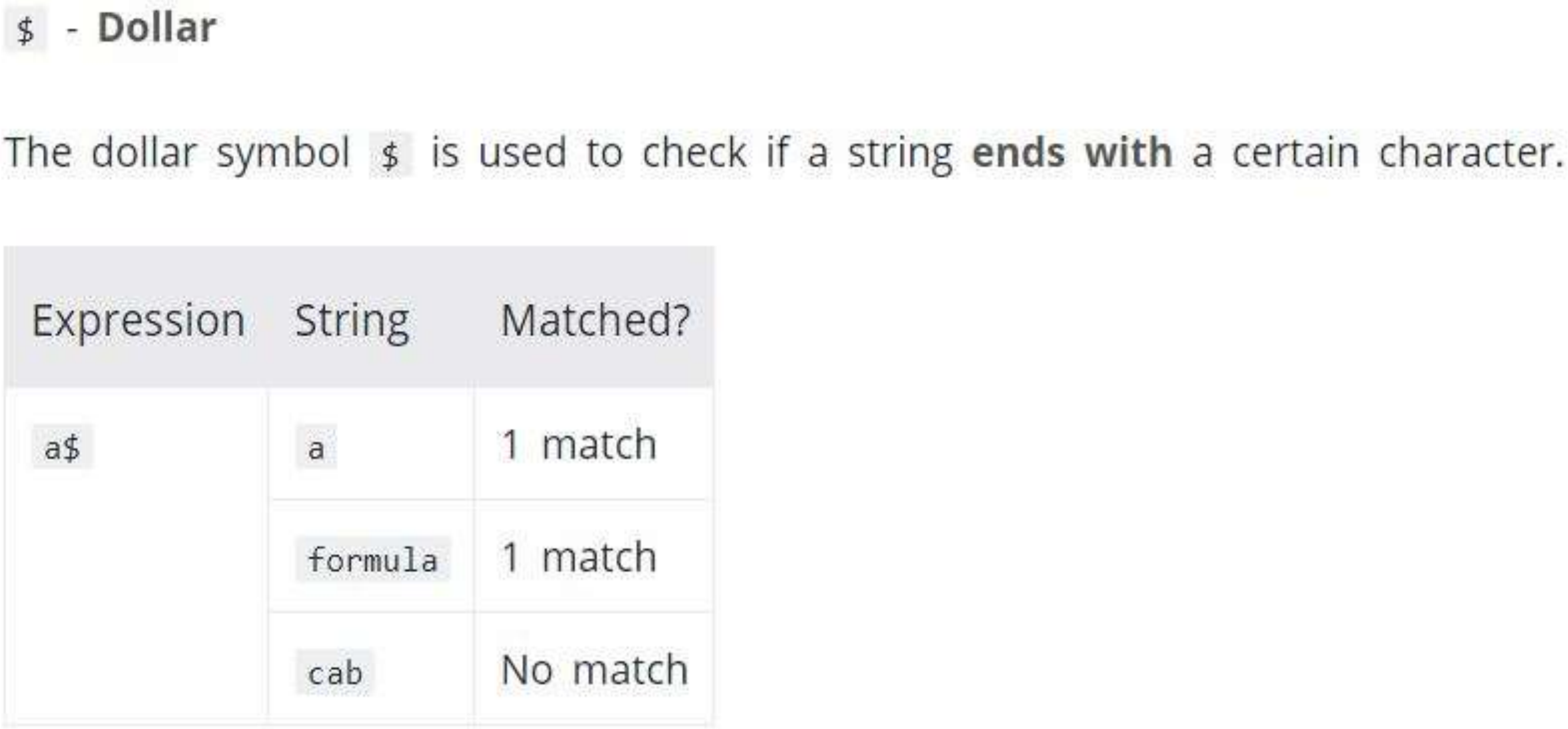
Ex:

```
import re
match=re.match('^a','ba')
if match:
  print("matched")
else:
  print("not matched")
```

Output: not matched

```
import re
match=re.match('^a',’ab')
if match:
  print("matched")
else:
  print("not matched“)
```

Output: matched

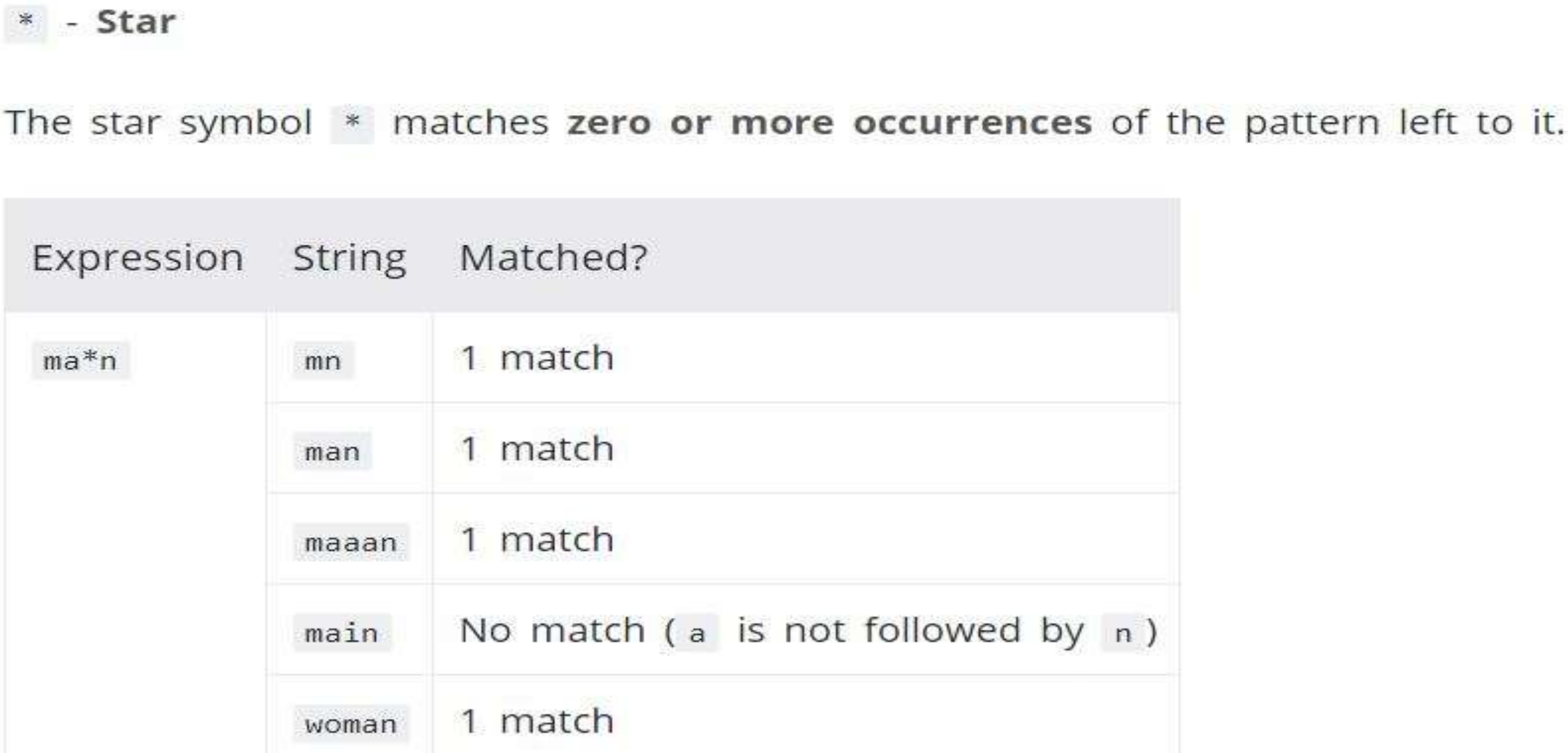


`$` - Dollar

The dollar symbol `$` is used to check if a string **ends with** a certain character.

Expression	String	Matched?
`a$`	`a`	1 match
	`formula`	1 match
	`cab`	No match

```
import re
match=re.match('......a$','formula')
if match:
  print("matched")
else:
  print("not matched")
```



`*` - Star

The star symbol `*` matches **zero or more occurrences** of the pattern left to it.

Expression	String	Matched?
`ma*n`	`mn`	1 match
	`man`	1 match
	`maaan`	1 match
	`main`	No match (`a` is not followed by `n`)
	`woman`	1 match

Ex:

```
 import re
match=re.match('ma*n','man')
if match:
  print("matched")
else:
  print("not matched")
```

Output: matched

Ex:

```
 import re
match=re.match('ma*n','main')
if match:
  print("matched")
else:
  print("not matched")
```

Output: not matched

`+` - Plus

The plus symbol `+` matches **one or more occurrences** of the pattern left to it.

Expression	String	Matched?
`ma+n`	`mn`	No match (no `a` character)
	`man`	1 match
	`maaan`	1 match
	`main`	No match (a is not followed by n)
	`woman`	1 match

Ex:

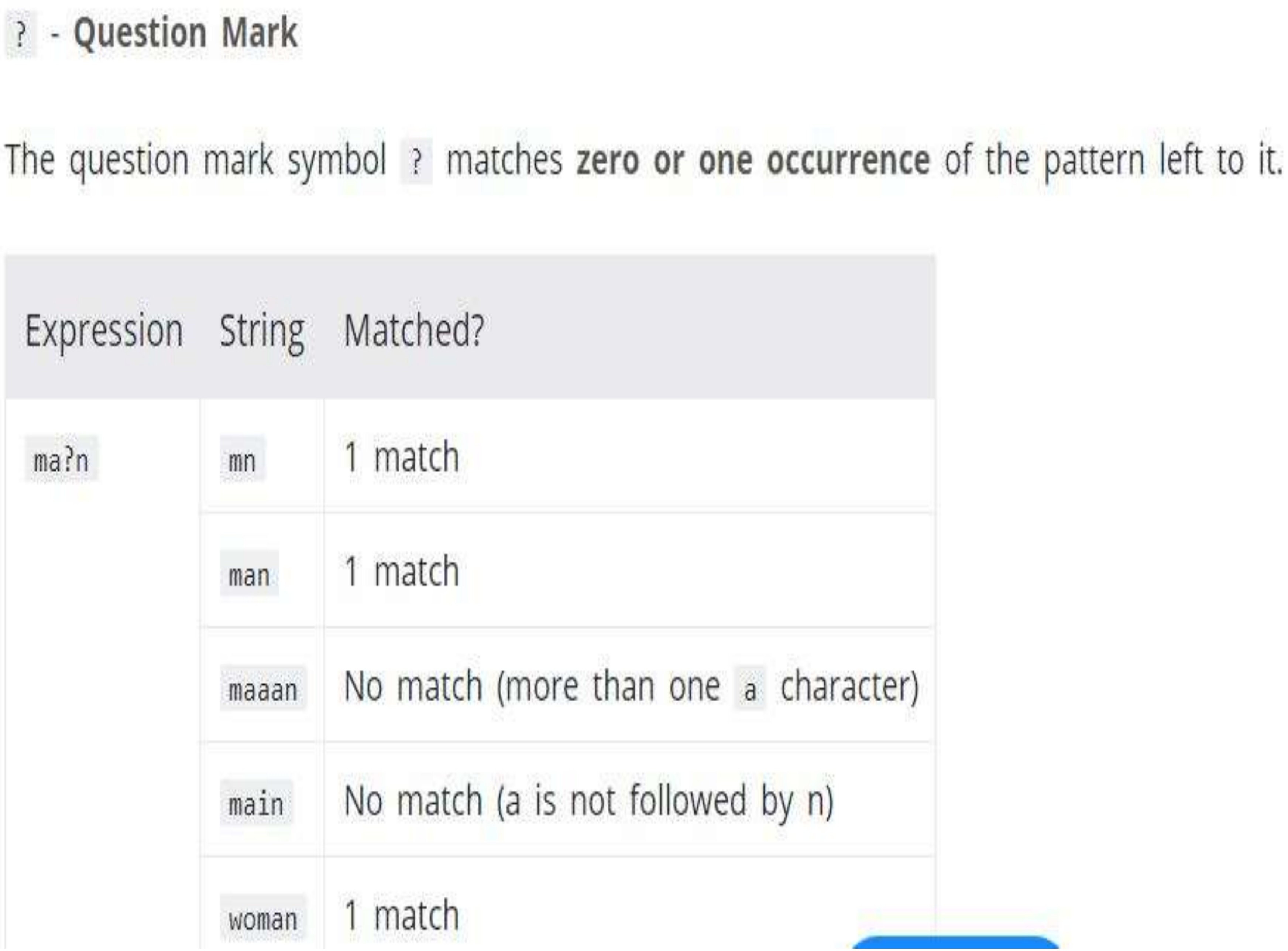
```
import re
match=re.match('ma+n','mn')
if match:
  print("matched")
else:
  print("not matched")
```

Output: not matched

Ex:

```
import re
match=re.match('ma+n','man')
if match:
  print("matched")
else:
  print("not matched“)
```

Output: matched



`?` - Question Mark

The question mark symbol `?` matches **zero or one occurrence** of the pattern left to it.

Expression	String	Matched?
`ma?n`	`mn`	1 match
	`man`	1 match
	`maaan`	No match (more than one `a` character)
	`main`	No match (a is not followed by n)
	`woman`	1 match

Ex:

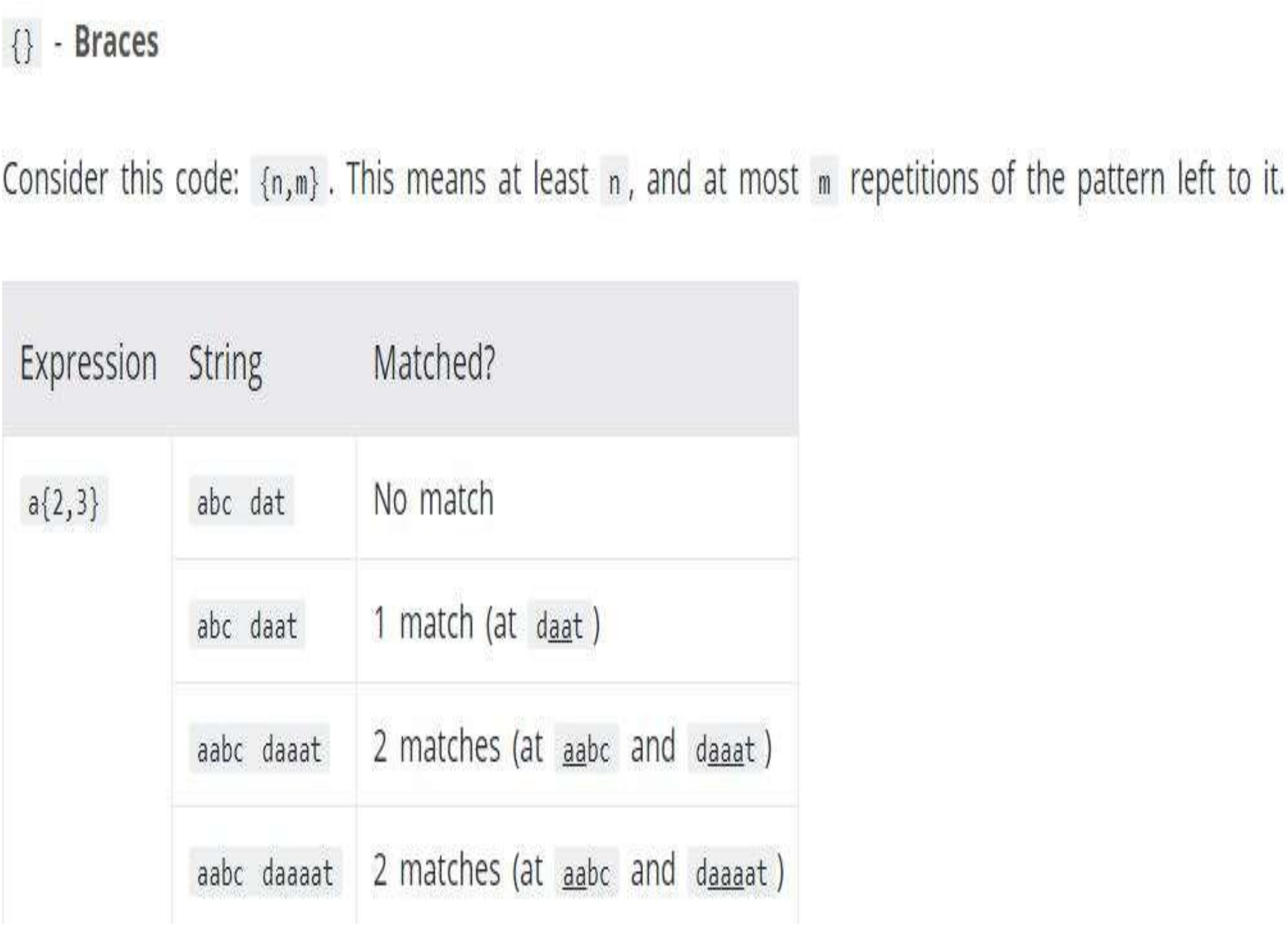
```
import re
match=re.match('ma?n','mn')
if match:
  print("matched")
else:
  print("not matched")
```

Output: matched

Ex:

```
import re
match=re.match('ma?n','maaan')
if match:
  print("matched")
else:
  print("not matched")
```

Output: not matched



`{}` - Braces

Consider this code: `{n,m}`. This means at least `n`, and at most `m` repetitions of the pattern left to it.

Expression	String	Matched?
`a{2,3}`	`abc dat`	No match
	`abc daat`	1 match (at `daat`)
	`aabc daaat`	2 matches (at `aabc` and `daaat`)
	`aabc daaaat`	2 matches (at `aabc` and `daaaat`)

Ex:

```
import re
match=re.match('a{2,3}','abc dat')
if match:
  print("matched")
else:
  print("not matched“)
```

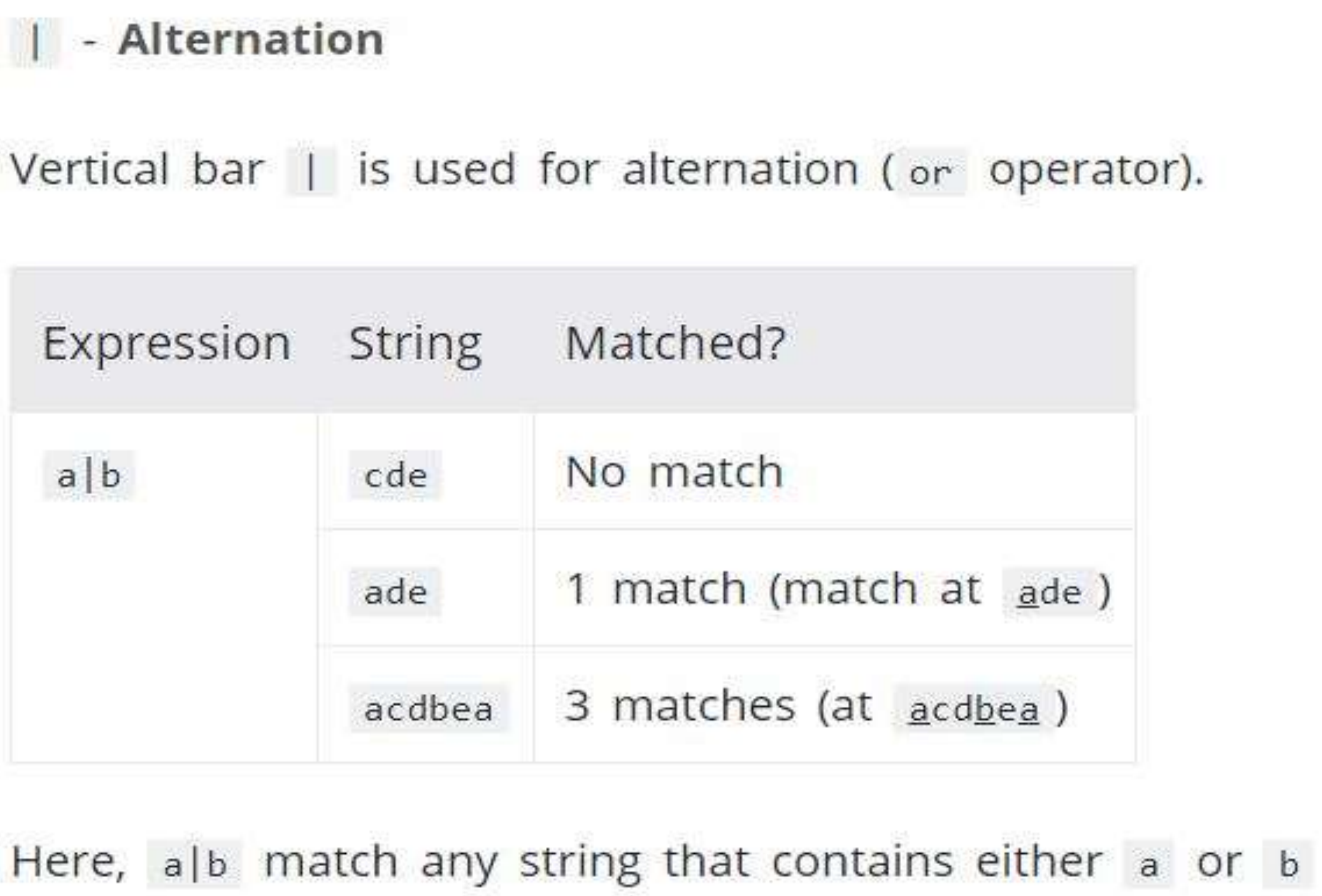
Output: not matched

```
import re
match=re.match('a{2,3}','aabc daaat')
if match:
  print("matched")
else:
  print("not matched“)
```

Output: matched

Let's try one more example. This RegEx `[0-9]{2, 4}` matches at least 2 digits but not more than 4 digits

Expression	String	Matched?
`[0-9]{2,4}`	`ab123csde`	1 match (match at `ab123csde`)
	`12 and 345673`	2 matches (at `12 and 345673`)
	`1 and 2`	No match



`|` - Alternation

Vertical bar `|` is used for alternation (`or` operator).

Expression	String	Matched?
`a|b`	`cde`	No match
	`ade`	1 match (match at `ade`)
	`acdbea`	3 matches (at `acdbea`)

Here, `a|b` match any string that contains either `a` or `b`

Ex:

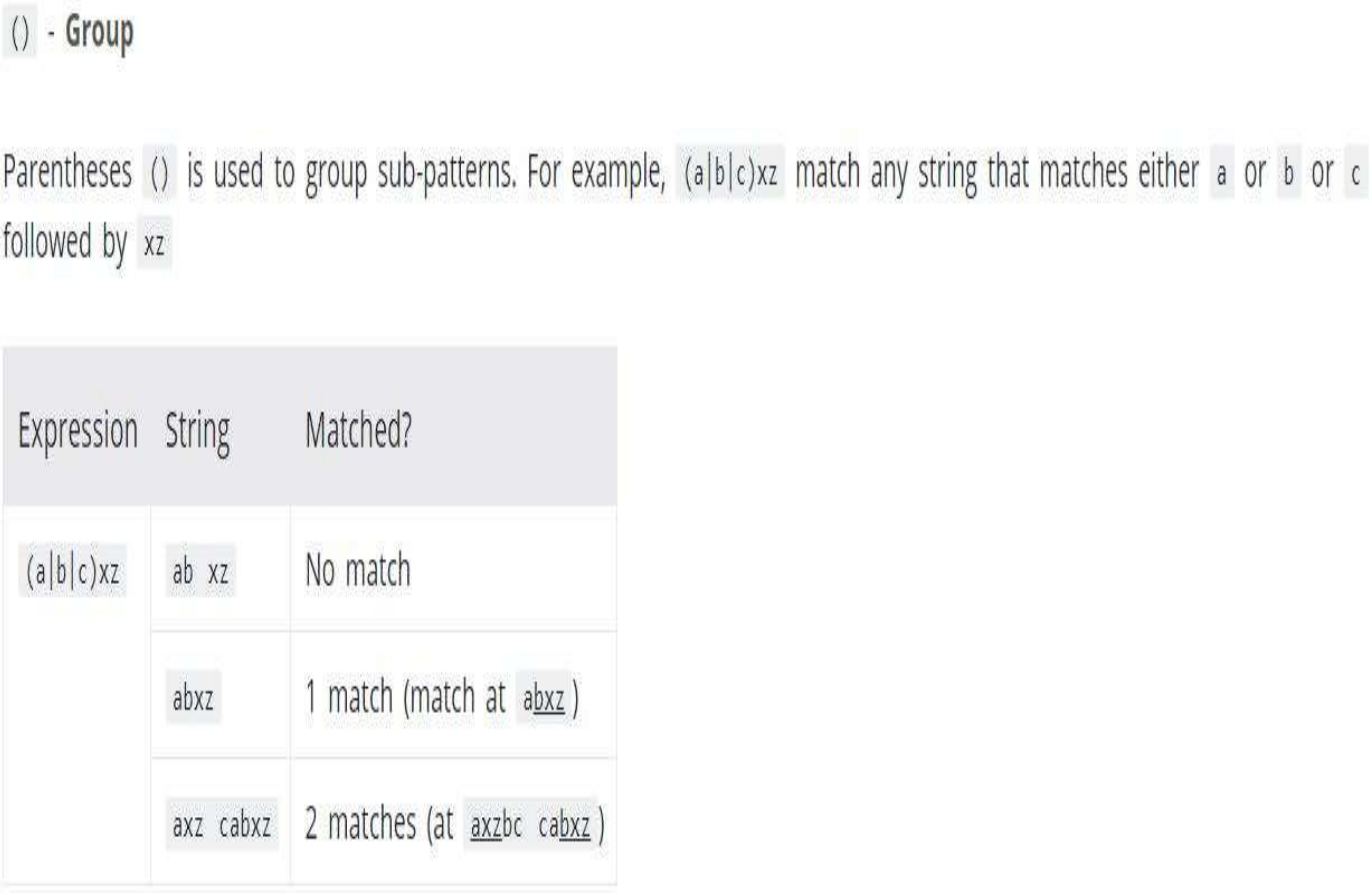
```
import re
match=re.match('a|b','ade')
if match:
  print("matched")
else:
  print("not matched“)
```

Output: matched

Ex:

```
import re
match=re.match('a|b',‘cde')
if match:
  print("matched")
else:
  print("not matched")
```

Output: not matched



`()` - Group

Parentheses `()` is used to group sub-patterns. For example, `(a|b|c)xz` match any string that matches either `a` or `b` or `c` followed by `xz`

Expression	String	Matched?
`(a|b|c)xz`	`ab xz`	No match
	`abxz`	1 match (match at `abxz`)
	`axz cabxz`	2 matches (at `axzbc cabxz`)

Ex:

```
import re
match=re.match('(a|b)xz','axz cabxz')
if match:
  print("matched")
else:
  print("not matched")
```

Output: matched

```
import re
match=re.match('(a|b)xz','ab xz')
if match:
  print("matched")
else:
  print("not matched")
```

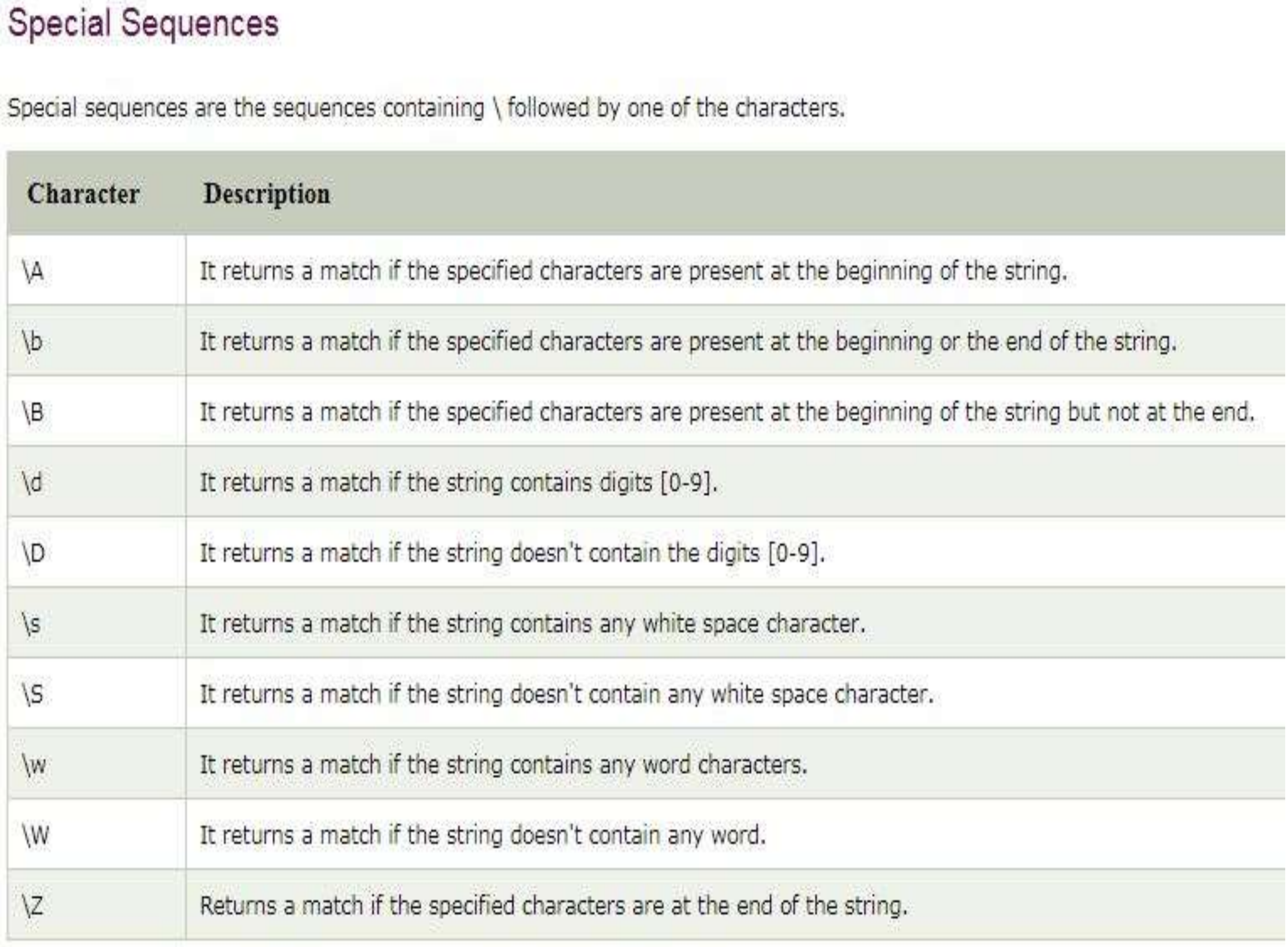
Output: not matched

\ - **Backslash**

✓Backlash \ is used to escape various characters including all metacharacters. For example,

✓\$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.

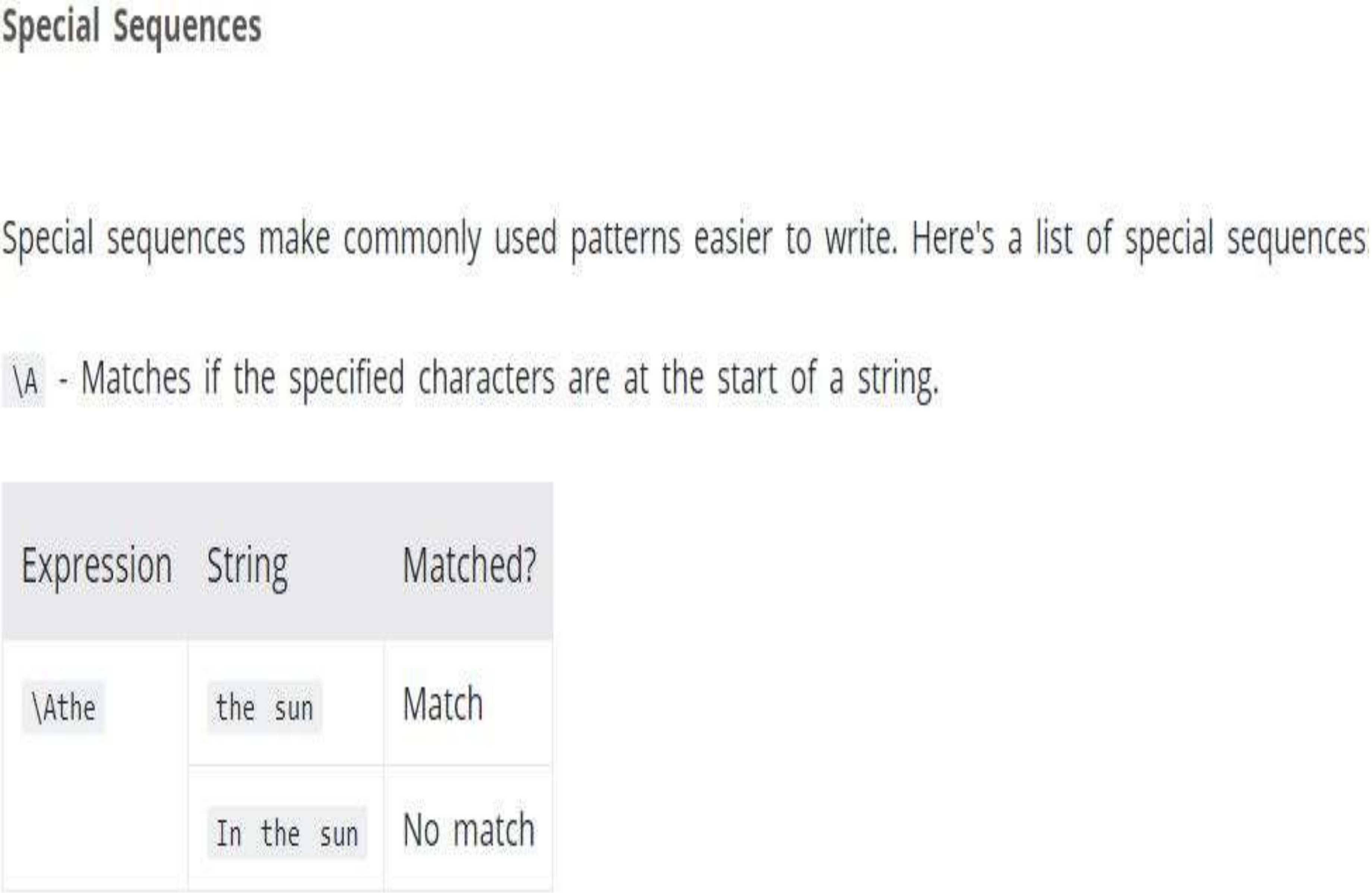
✓If you are unsure if a character has special meaning or not, you can put \ in front of it. This makes sure the character is not treated in a special way.



Special Sequences

Special sequences are the sequences containing \ followed by one of the characters.

Character	Description
\A	It returns a match if the specified characters are present at the beginning of the string.
\b	It returns a match if the specified characters are present at the beginning or the end of the string.
\B	It returns a match if the specified characters are present at the beginning of the string but not at the end.
\d	It returns a match if the string contains digits [0-9].
\D	It returns a match if the string doesn't contain the digits [0-9].
\s	It returns a match if the string contains any white space character.
\S	It returns a match if the string doesn't contain any white space character.
\w	It returns a match if the string contains any word characters.
\W	It returns a match if the string doesn't contain any word.
\Z	Returns a match if the specified characters are at the end of the string.

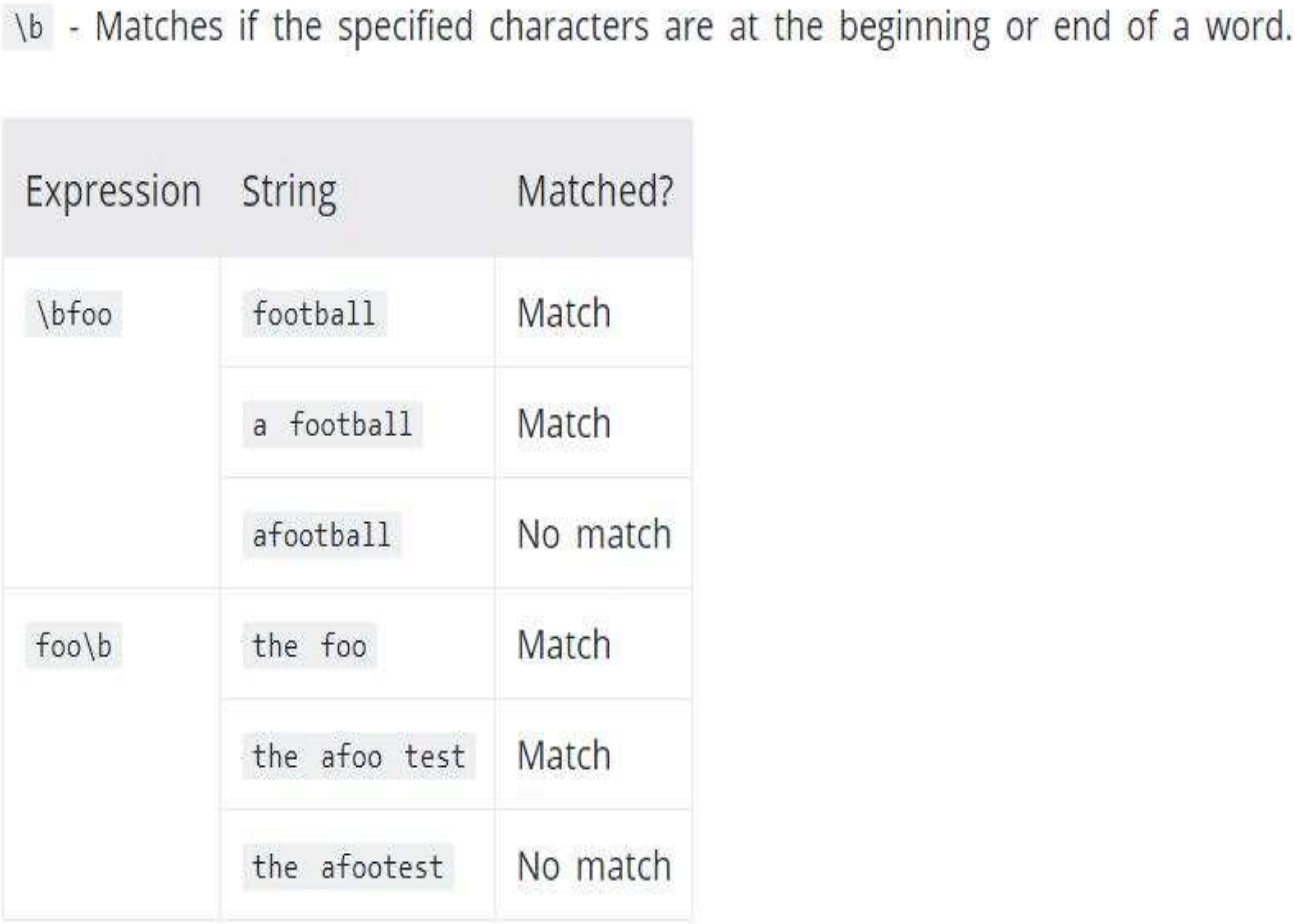


Special Sequences

Special sequences make commonly used patterns easier to write. Here's a list of special sequences:

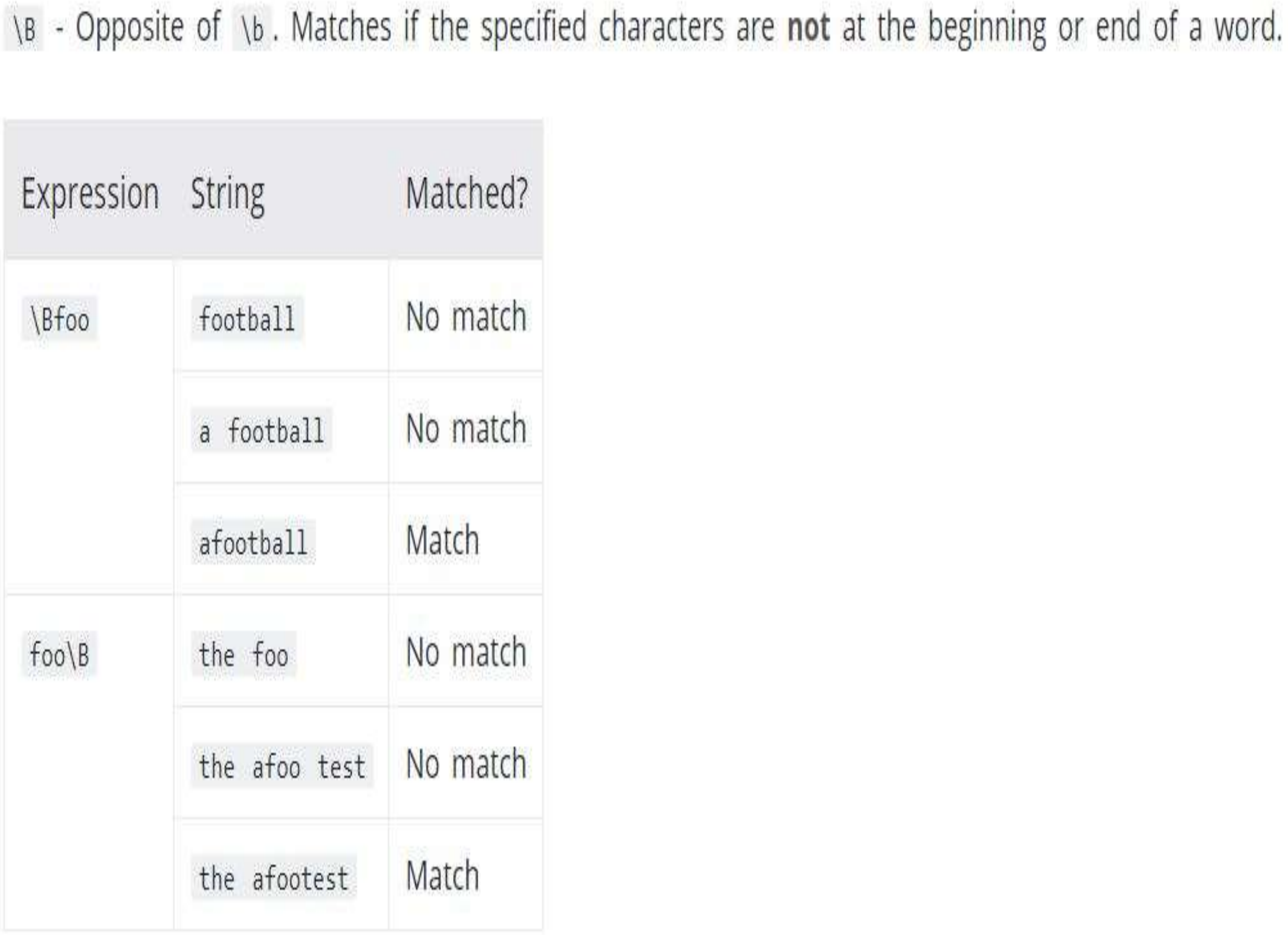
`\A` - Matches if the specified characters are at the start of a string.

Expression	String	Matched?
`\Athe`	`the sun`	Match
	`In the sun`	No match



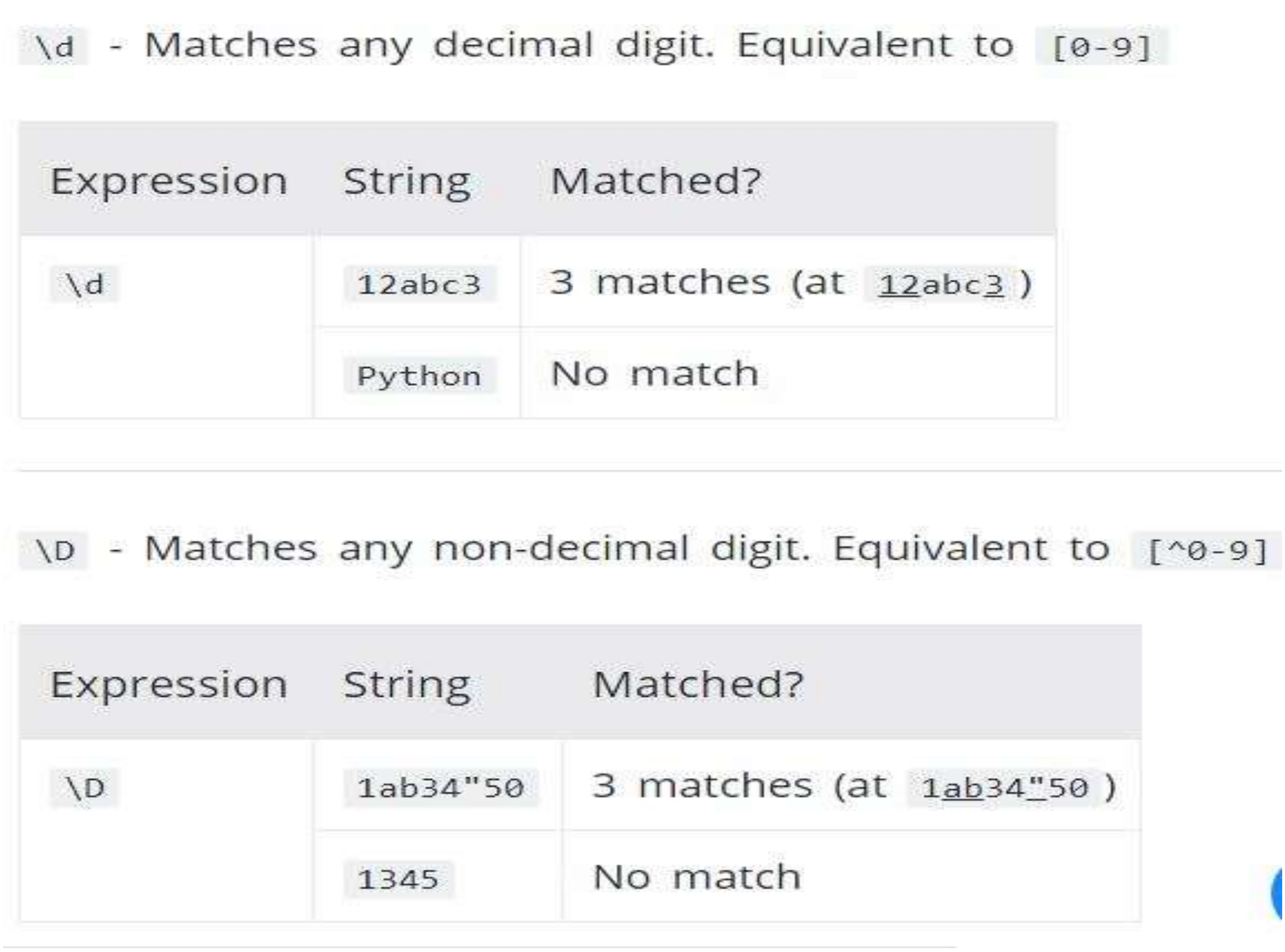
`\b` - Matches if the specified characters are at the beginning or end of a word.

Expression	String	Matched?
`\bfoo`	`football`	Match
	`a football`	Match
	`afootball`	No match
`foo\b`	`the foo`	Match
	`the afoo test`	Match
	`the afootest`	No match



`\B` - Opposite of `\b`. Matches if the specified characters are **not** at the beginning or end of a word.

Expression	String	Matched?
`\Bfoo`	`football`	No match
	`a football`	No match
	`afootball`	Match
`foo\B`	`the foo`	No match
	`the afoo test`	No match
	`the afootest`	Match

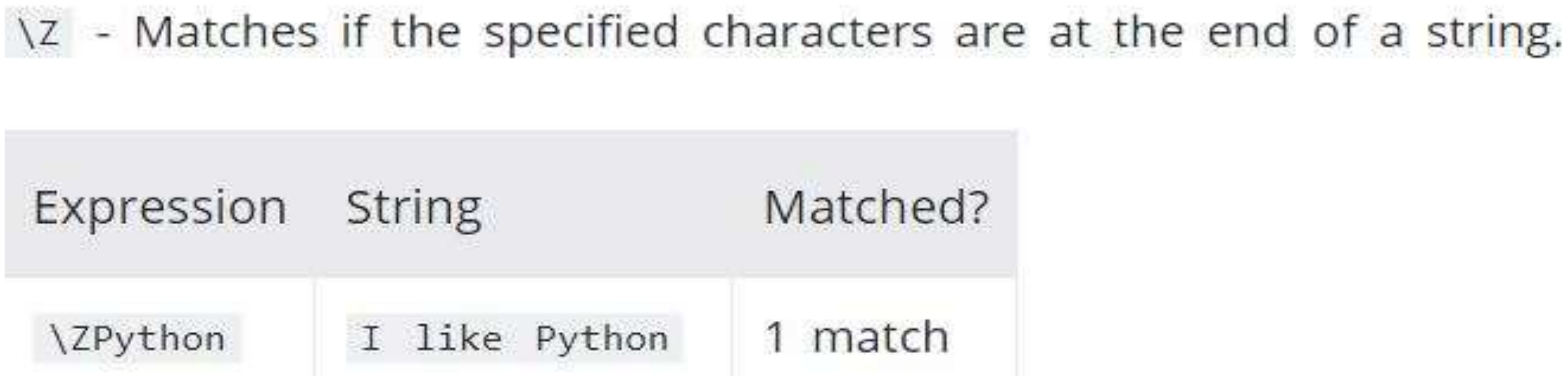


`\d` - Matches any decimal digit. Equivalent to `[0-9]`

Expression	String	Matched?
`\d`	`12abc3`	3 matches (at `12abc3`)
	`Python`	No match

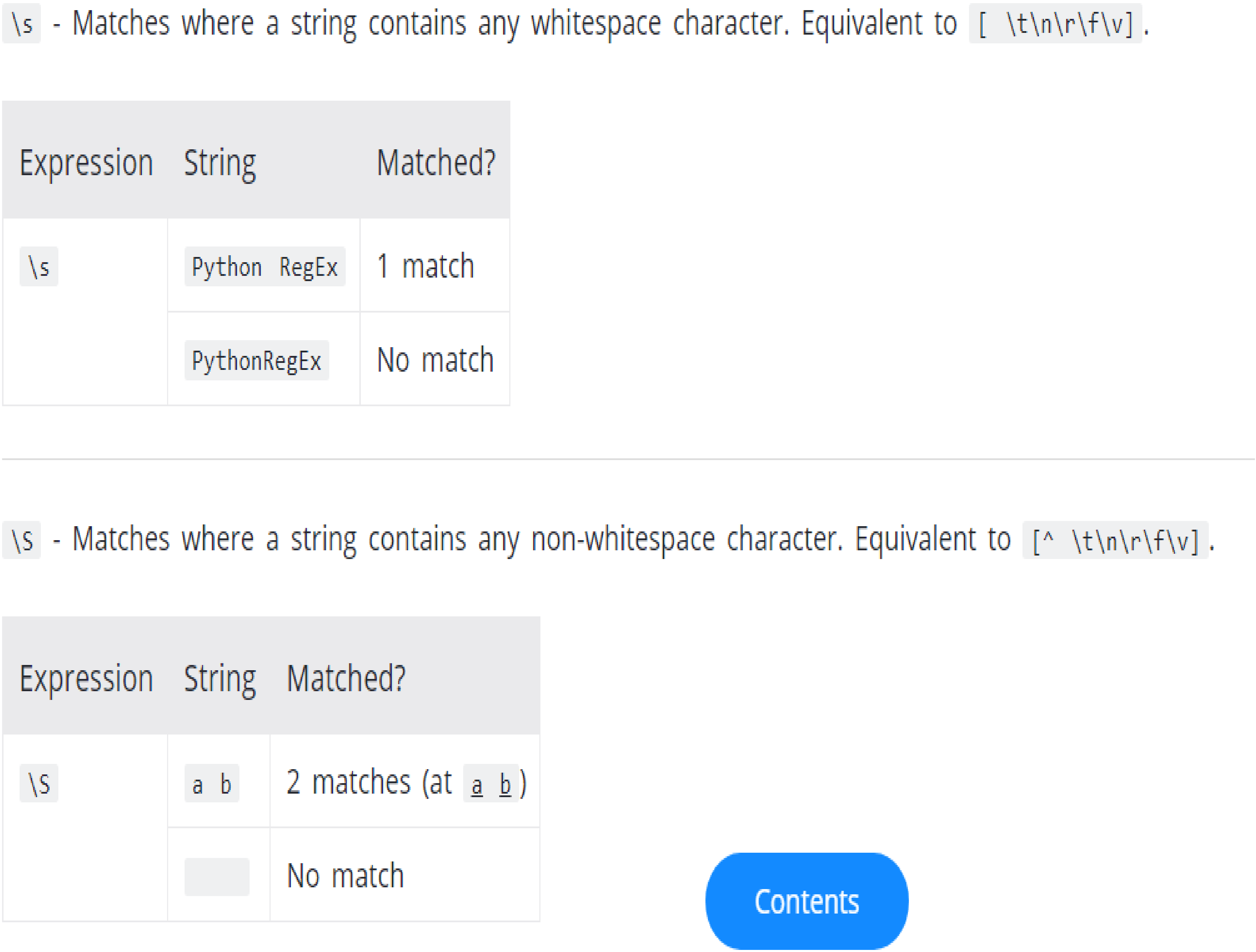
`\D` - Matches any non-decimal digit. Equivalent to `[^0-9]`

Expression	String	Matched?
`\D`	`1ab34"50`	3 matches (at `1ab34"50`)
	`1345`	No match



`\Z` - Matches if the specified characters are at the end of a string.

Expression	String	Matched?
`\ZPython`	`I like Python`	1 match

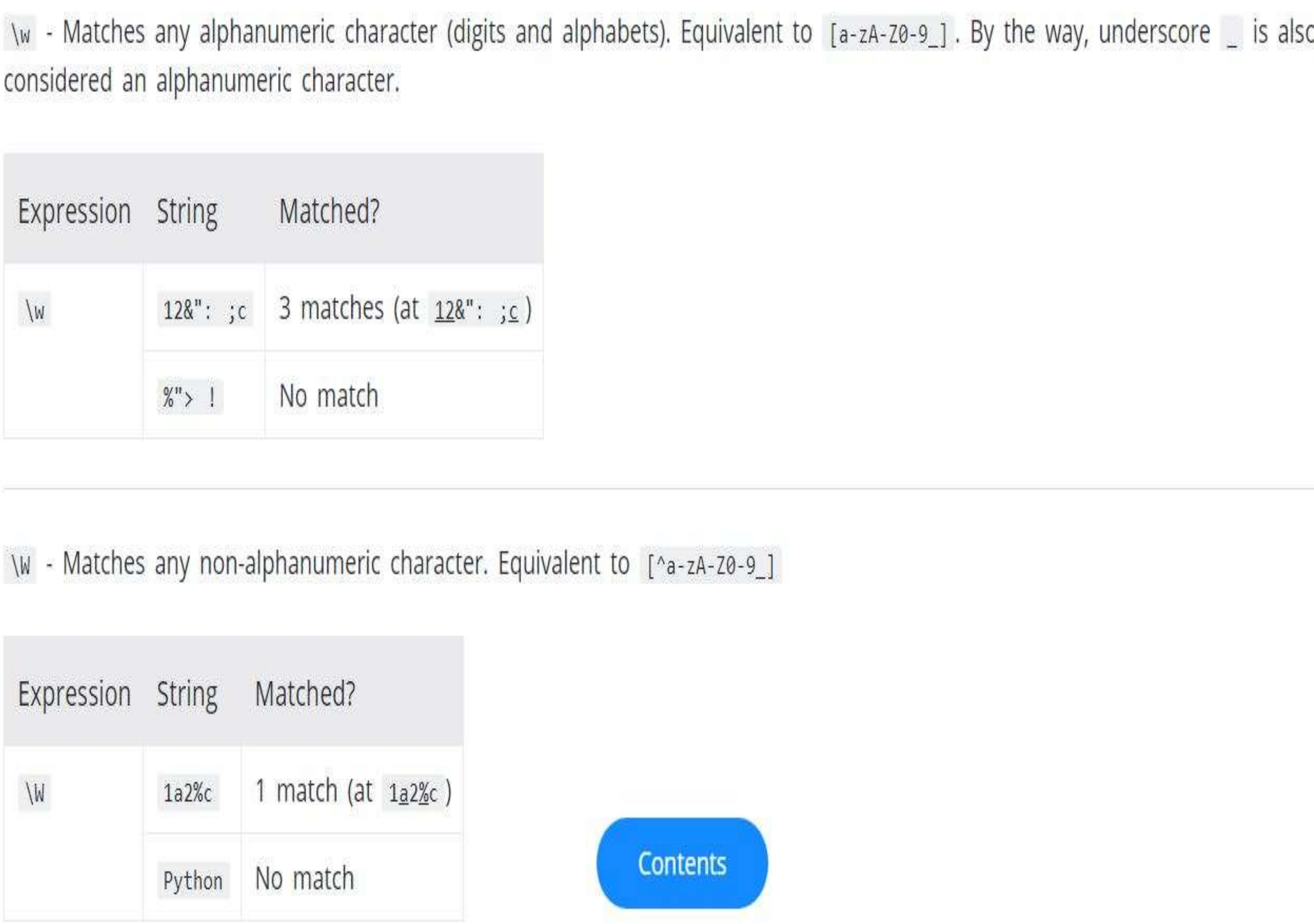


`\s` - Matches where a string contains any whitespace character. Equivalent to `[ \t\n\r\f\v]`.

Expression	String	Matched?
`\s`	`Python RegEx`	1 match
	`PythonRegEx`	No match

`\S` - Matches where a string contains any non-whitespace character. Equivalent to `[^ \t\n\r\f\v]`.

Expression	String	Matched?
`\S`	`a b`	2 matches (at `a b`)
	` `	No match

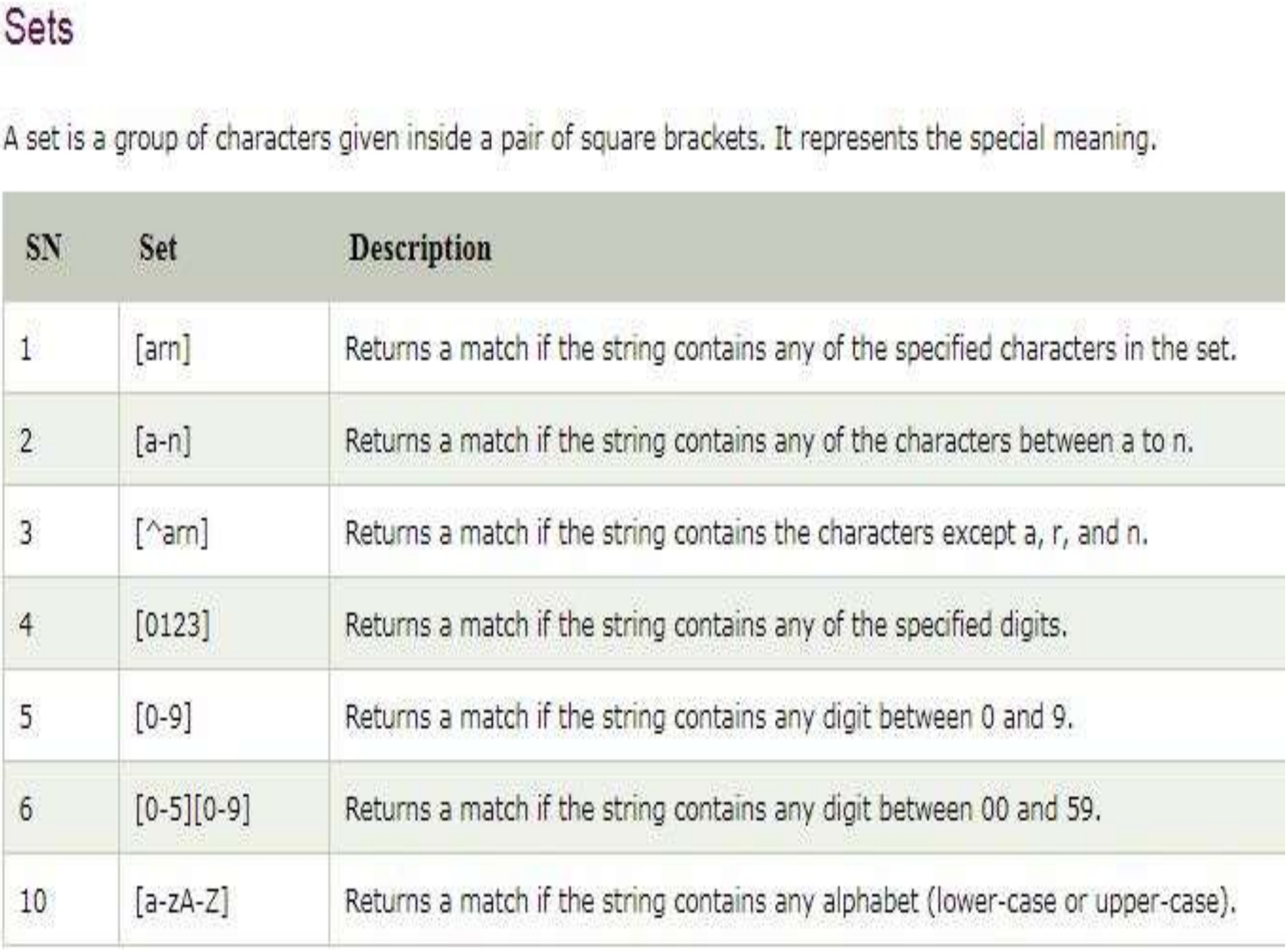


\w - Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]. By the way, underscore _ is also considered an alphanumeric character.

Expression	String	Matched?
\w	12&": ;c	3 matches (at 12&": ;c)
	%"> !	No match

\W - Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_]

Expression	String	Matched?
\W	1a2%c	1 match (at 1a2%c)
	Python	No match



Sets

A set is a group of characters given inside a pair of square brackets. It represents the special meaning.

SN	Set	Description
1	[arn]	Returns a match if the string contains any of the specified characters in the set.
2	[a-n]	Returns a match if the string contains any of the characters between a to n.
3	[^arn]	Returns a match if the string contains the characters except a, r, and n.
4	[0123]	Returns a match if the string contains any of the specified digits.
5	[0-9]	Returns a match if the string contains any digit between 0 and 9.
6	[0-5][0-9]	Returns a match if the string contains any digit between 00 and 59.
10	[a-zA-Z]	Returns a match if the string contains any alphabet (lower-case or upper-case).

group()

✓The group() method returns the part of the string where there is a match.

```
>>> import re
>>> string='324 45 656 65'
>>> pattern = '(\d{3}) (\d{2})'
>>> match = re.search(pattern, string)
>>> match.group()
'324 45‘
>>> match.group(0)
'324 45'
>>> match.group(1)
'324‘
>>> match.groups()
('324', '45')
```

match.start(), match.end(), match.string and match.span():

✓**The start()** function returns **the index of the start** of the matched substring.

✓Similarly, **end() returns** the **end index of the matched** substring.

✓**string():** returns a **string passed into the function**

```
>>> match.start()
0
>>> match.end()
6
>>> match.span()
(0, 6)
>>> match.string
'324 45 656 65'
```

re.compile():

Regular expressions are compiled **into pattern objects**, which have **methods** for various operations such as **searching** for pattern **matches** or performing string substitutions.

Ex:

```
import re
p = re.compile('[a-e]')
print(p.findall("Aye, said Mr. Gibenson Stark"))
```

Output:

```
['e', 'a', 'd', 'b', 'e', 'a']
```

Using r prefix before RegEx

✓When **r or R prefix is used before a regular expression**, it means raw string.

✓ For example, '\n' is a new line whereas r'\n' means two characters: a backslash \ followed by n.

✓Backlash \ is used to escape various characters including all metacharacters. However, using r prefix makes \ treat as a normal character.

Example : Raw string using r prefix

```
import re
string = '\n and \r are escape sequences.'
result = re.findall(r'[\n\r]', string)
print(result)
```

# Output: ['\n', '\r']

Multithreaded (MT) Programming:

What is Multithreading?

✓**Multithreading** is defined as the ability of a processor to execute multiple threads concurrently or parallelly.

✓MT is a running of computer programs consisted of a **single sequence of steps** that were executed in **synchronous order** by the host's central processing unit (**CPU).**

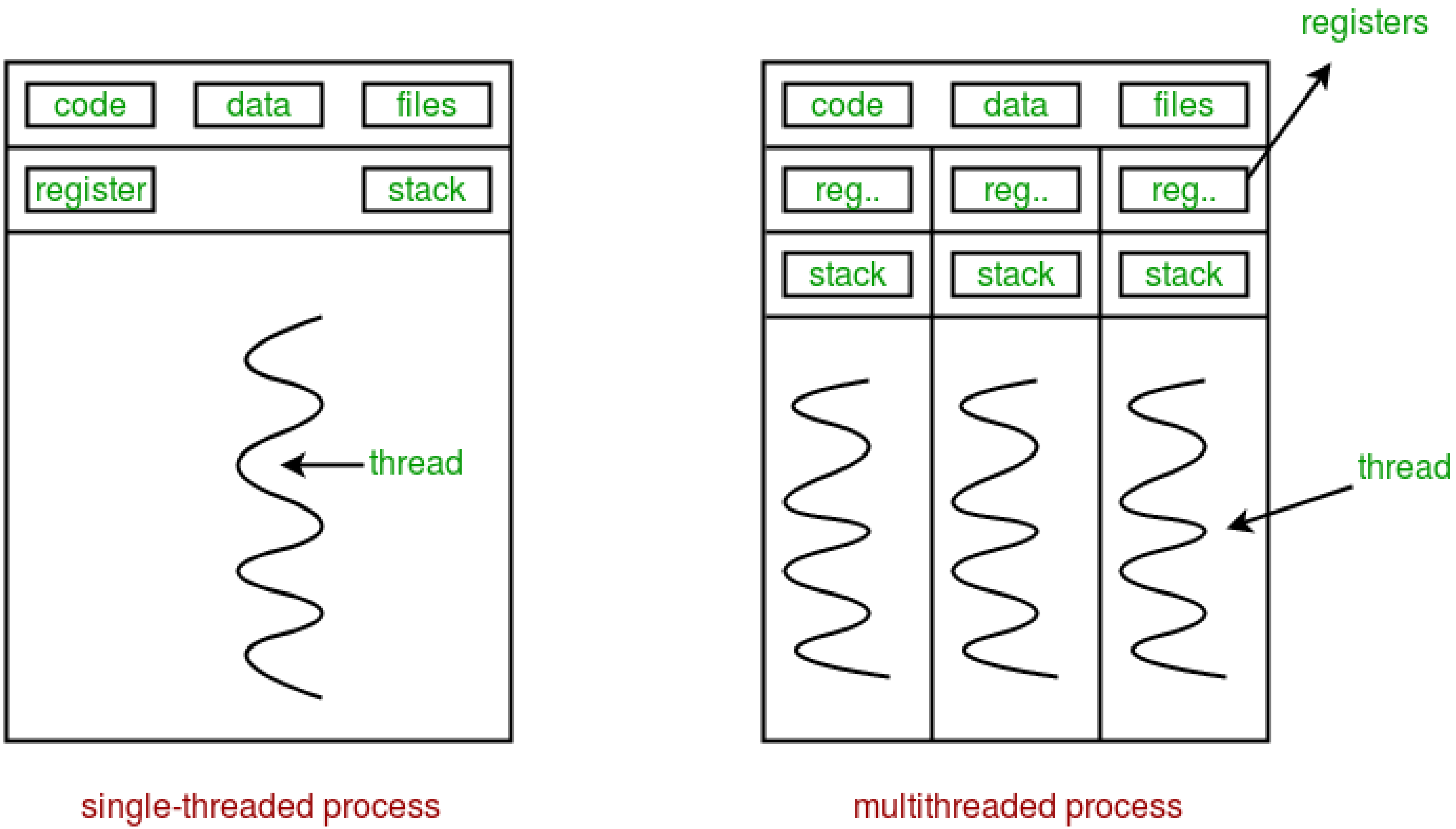
✓multithreading can significantly **improve** the performance of any program.

✓Multithreading can improve **the speed of computation on multiprocessor or multi-core systems** because each processor or core handles a separate **thread concurrently.**

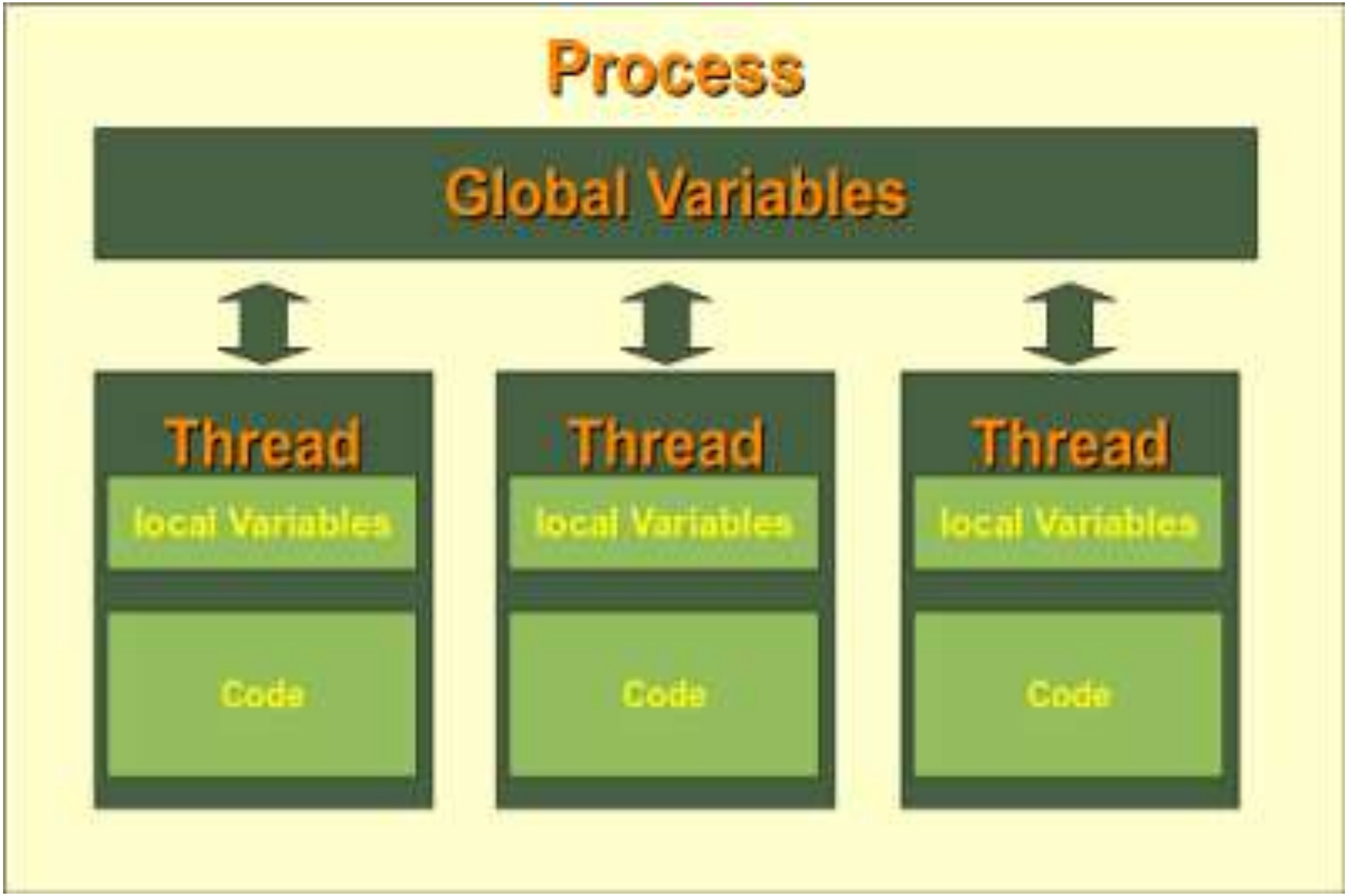
✓Multithreading allows a program to remain **responsive** while one thread waits for input, and **another** runs a GUI at the same time.

Multiple threads can exist within one process where:

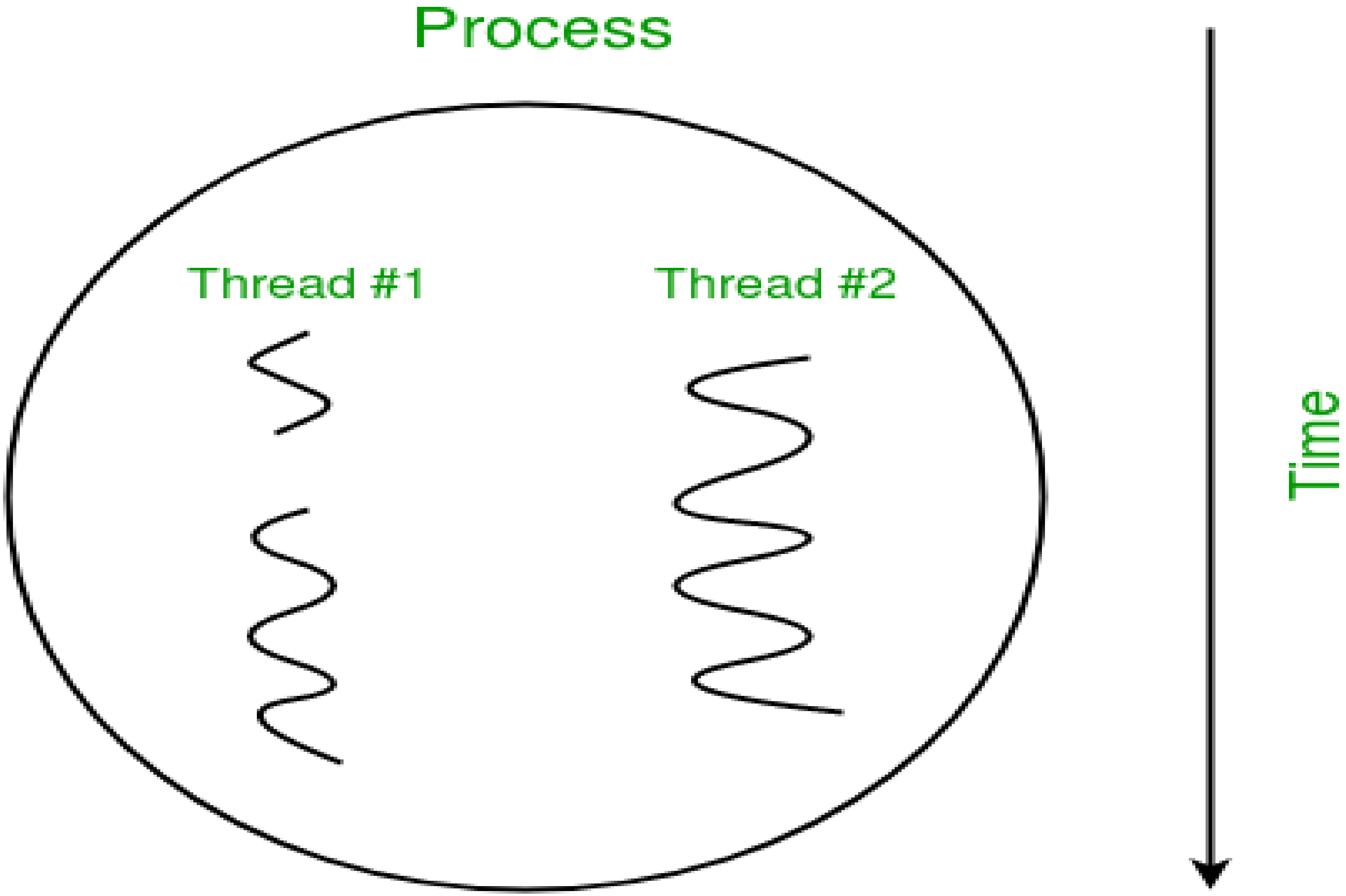
- ✓Each thread contains its own **register set** and **local variables (stored in stack)**.
- ✓All thread of a process share **global variables (stored in heap)** and the **program code**.



✓All the **threads** of a process have **access to its global variables**. If a global variable changes in one thread, it is visible to other threads as well. **A thread can have local variables.**



Consider the diagram below in which a process contains two active threads



Threads and Processes:

What Are Processes?

✓ A **process (heavyweight process)** is a program in execution. Each process has its own **address space, memory, a data stack, and other auxiliary data** to keep track of execution.

✓The operating system **manages the execution of all processes** on the system, dividing the time **fairly** between all processes.

✓Processes can also **fork or spawn new processes to perform other tasks**, but each new process has its **own memory, data stack,** etc., and cannot share information unless interprocess-communication (IPC) is employed.

A process is an instance of a computer program that is being executed.

Any process has **3 basic components:**

- ✓An executable program.
- ✓The associated data needed by the program (variables, work space, buffers, etc.)
- ✓The execution context of the program (State of process)

What Are Threads?

✓Threads is a **lightweight processes** are similar to processes except that **they all execute within the same process,** and thus **all share the same context.** They can be thought of as **"mini-processes“** running in parallel within a **main process** or "main thread.“

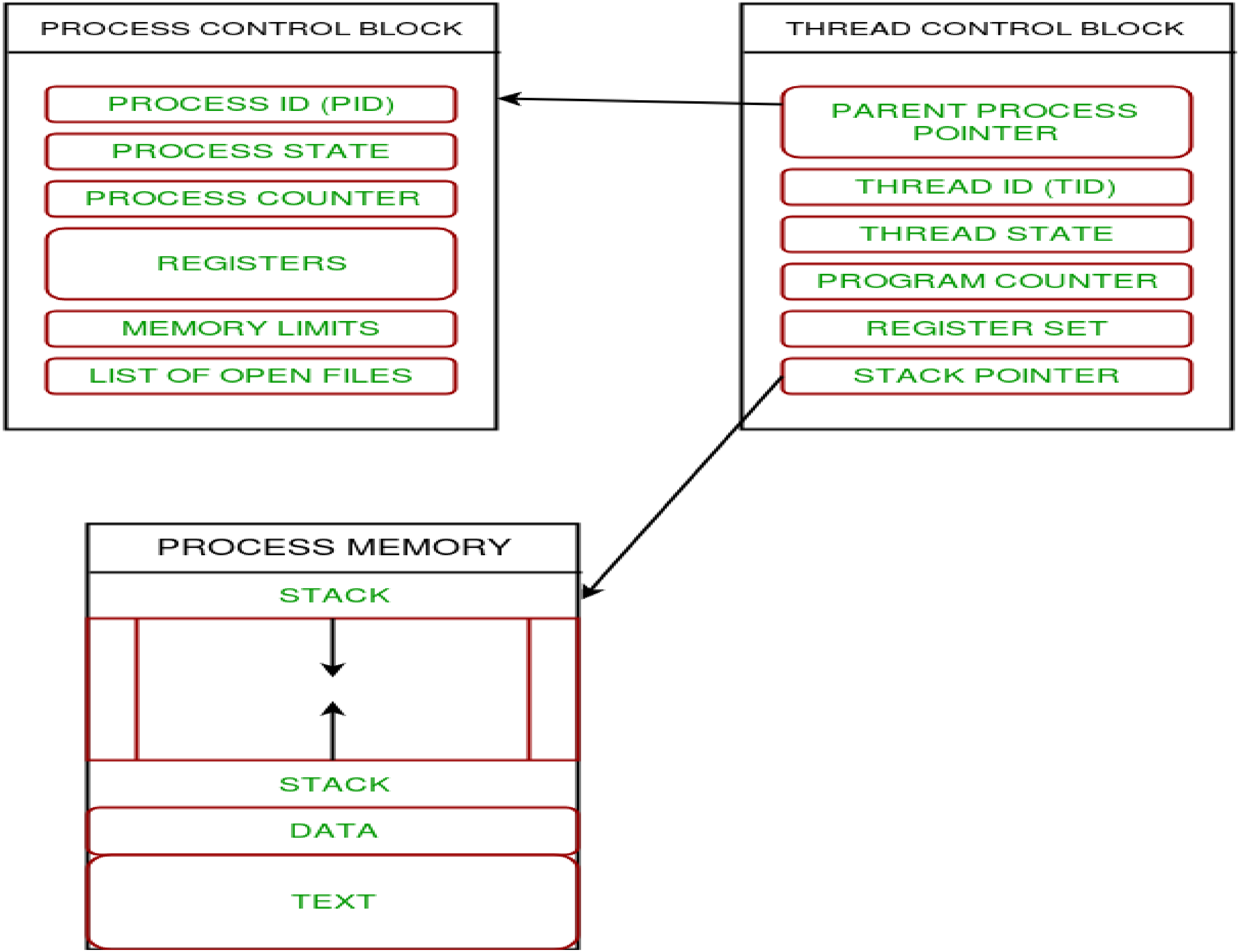
✓Thread is simply a **subset of a process**!

✓ **Thread** is a sequence of instructions within a program that can be executed **independently** of other code.

✓A thread has a **beginning, an execution sequence, and a conclusion**.

✓It has an **instruction pointer** that keeps track of where within its context it is currently running.

✓It can be preempted (**interrupted**) and temporarily put on hold (also known as **sleeping**) while other threads are running this is called **yielding.**


PROCESS CONTROL BLOCK
PROCESS ID (PID)
PROCESS STATE
PROCESS COUNTER
REGISTERS
MEMORY LIMITS
LIST OF OPEN FILES
THREAD CONTROL BLOCK
PARENT PROCESS POINTER
THREAD ID (TID)
THREAD STATE
PROGRAM COUNTER
REGISTER SET
STACK POINTER
PROCESS MEMORY
STACK
STACK
DATA
TEXT

A thread contains all this information in a **Thread Control Block (TCB)**:

✓**Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

✓**Thread Identifier:** Unique id (TID) is assigned to every new thread.

✓**Thread state:** can be running, ready, waiting, start or done.

✓**Program counter:** a register which stores the **address of the instruction** currently being executed by thread.

✓**Thread's register set:** registers **assigned to thread** for computations.

✓**Stack pointer:** Points to **thread's stack in the process memory** . Stack contains the local variables under thread's scope.

Python Multithreading Modules

Python provides several modules to support MT programming, important modules are

1. thread,
2. threading, and
3. Queue modules.

- ✓The **thread and threading modules** allow the programmer to create and manage threads.
- ✓The **thread module** provides basic **thread and locking support,** while **threading** provides **higher-level, fully featured thread management**.
- ✓The **Queue module** allows the user **to create a queue data structure** that can be shared across multiple threads.

<u>Note</u>: For your information, Python 2.x used to have the **<*thread*> module**. But it got deprecated in Python 3.x and renamed to **<_*thread*> module** for backward compatibility.

✓The difference between two modules is that module **<_*thread*>** implements a thread as a function. **<*threading*>** offers an object-oriented approach to enable thread creation.

1. thread Module:

The thread module provides a **basic synchronization data structure** called a ***lock object*** (aka primitive lock, simple lock, mutual exclusion lock, mutex, binary semaphore).

thread *Module Functions:*

1. start_new_thread(*function, args, kwargs=None):*

It **spawns a new thread** and execute *function with the* given ***args*** *and optional* ***kwargs***

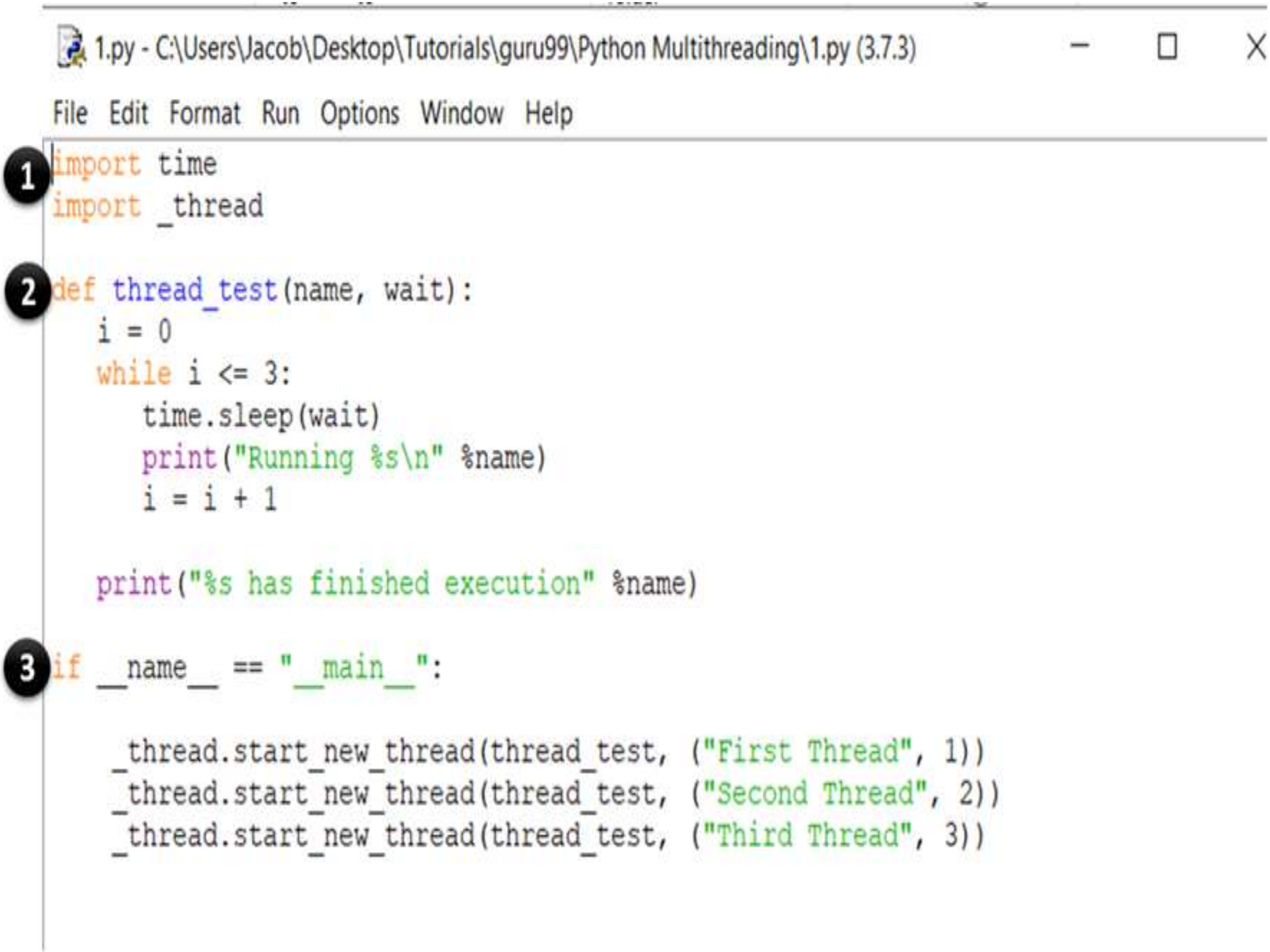
#Syntax

thread.start_new_thread (function, args[, kwargs])

✓This method starts a new thread and returns its **identifier.** It'll invoke the **function** specified as the "function" parameter with the passed list of arguments. When the <*function*> returns, the **thread would silently exit.**

✓Here, ***args*** is a **tuple of arguments**; use an empty tuple to call <*function*> without any arguments.

✓The optional **<*kwargs*> argument** specifies the dictionary of keyword arguments.



1.py - C:\Users\Jacob\Desktop\Tutorials\guru99\Python Multithreading\1.py (3.7.3)
File Edit Format Run Options Window Help
1
2
3

```python
import time
import _thread

def thread_test(name, wait):
   i = 0
   while i <= 3:
      time.sleep(wait)
      print("Running %s\n" %name)
      i = i + 1

   print("%s has finished execution" %name)

if __name__ == "__main__":

    _thread.start_new_thread(thread_test, ("First Thread", 1))
    _thread.start_new_thread(thread_test, ("Second Thread", 2))
    _thread.start_new_thread(thread_test, ("Third Thread", 3))
```

2. allocate_lock()

- ✓ allocates **LockType lock object.**
- ✓ it return a **new lock object.** Initially ,the **lock is unlocked.**

Syntax:

```
thread.allocate_lock()
```

3. exit():

✓Instructs a **thread to exit.**

✓Raise the **SystemExit exception.** When not caught, this will cause the **thread to exit** silently.

Syntax:

```
thread.exit()
```

LockType Lock Object Methods:

1. acquire(waitflag):

- ✓ Attempts to acquire lock object.

Syntax:

lock.acquire(*waitflag=1*, *timeout*= *-1*)

- ✓ If the integer ***waitflag*** argument is present, the action depends on its **value:** if it is zero, **the lock is only acquired if it can be acquired immediately without waiting**, while if it is nonzero**, the lock is acquired unconditionally** .

✓ the floating-point **positive *timeout*** argument is present, it specifies the **maximum wait time** in **seconds** before returning. A **negative *timeout*** argument specifies an **unbounded wait.** You cannot specify a *timeout* if *waitflag* is zero.

✓The if the lock is acquired successfully return value is **True** otherwise **False.**

2. locked():

returns True if lock acquired, False otherwise

syntax:

```
lock.locked()
```

3. release() :

- ✓ Releases lock.

Syntax:

```
lock.release( )
```

2. threading Module:

This module provides **rich features and better support for threads** than the legacy <*thread*> module .The <*threading*> module is an excellent example of Python Multithreading.

threading Module Objects

1. **Thread:**

Object that represents a single thread of execution.

Syntax: threading.Thread

2. **Lock:**

It returns Primitive lock object (same lock object as in the tHRead module)

Syntax: threading.Lock()

3. **RLock**

- ✓ It returns **a new reentrant lock object**. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock**, the same thread may acquire it again without blocking**; the thread must release it once for each time it has acquired it.

Syntax: threading.RLock()

4. Condition:

it returns a **new condition variable object**. A condition variable allows one or more threads to **wait** until they are notified by another thread.

Syntax:

```
threading.Condition()
```

5. Semaphore

✓It returns a **new semaphore object.** A semaphore manages a **counter** representing the number of release() calls minus the number of acquire() calls, plus an initial value

✓It provides a "**waiting area**"**-** like structure for threads waiting on a lock.

Syntax:

threading.Semaphore([*value*])

6. BoundedSemaphore:

It is similar to a Semaphore but **ensures it never exceeds its initial value**.

Syntax:

threading.BoundedSemaphore([*value*])

7. Timer:

It is similar to **Thread** except **that it waits for an allotted period of time before running.**

Syntax:

***class* threading.Timer**

```
# Python program to illustrate the concept  of threading
        import threading
        def print_cube(num):
            """
            function to print cube of given num
            """
            print("Cube: {}".format(num * num * num))

        def print_square(num):
            """
            function to print square of given num
            """
            print("Square: {}".format(num * num))
```

```
if __name__ == "__main__":
    # creating thread
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))
    # starting thread 1
    t1.start()
    # starting thread 2
    t2.start()
    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()
    # both threads completely executed
    print("Done!")
```

Output:

```
Square: 100 Cube: 1000 Done!
```

Thread Class:

✓<***threading*****>** module also presents the **<*Thread*> class** that you can try for implementing threads. **It is an object-oriented variant of Python multithreading**.

✓The Thread class of the threading is your **primary executive object**. It has a variety of functions not available to the thread module.

1. start()

✓ Start the **thread's activity/execution**.

✓It must be called **at most once per thread object**. It arranges for the object's **run() method** to be invoked in a separate thread of control.

✓This method will raise a **RuntimeErro**r if called more than once on the same thread object.

2. run() :

✓Method **defining thread functionality** (usually overridden by application writer in a subclass)

4. join(timeout = None):

Suspend until the started thread terminates; blocks unless timeout (in seconds) is given

5 .getName(): Return name of thread

6. setName(name): Set name of thread

7. isAlive(): Boolean flag indicating whether thread is still running.

8. daemon:

A boolean value indicating whether this **thread is a daemon thread (True) or not (False).** This must be set **before start()** is called, otherwise **RuntimeError** is raised. The main thread is not a daemon thread and therefore all threads created in the main thread default to **daemon = False.**

9. isDaemon() Return daemon flag of thread

10. setDaemon(daemonic):

Set the daemon flag of thread as per the Boolean daemonic (must be called before thread start()ed)

Daemon thread does not block the main thread from exiting and continues to run the background.

The <threading> module combines all the methods of the <thread> module and exposes few additional methods.

1. **threading.activeCount():** It finds the total no. of active thread objects.

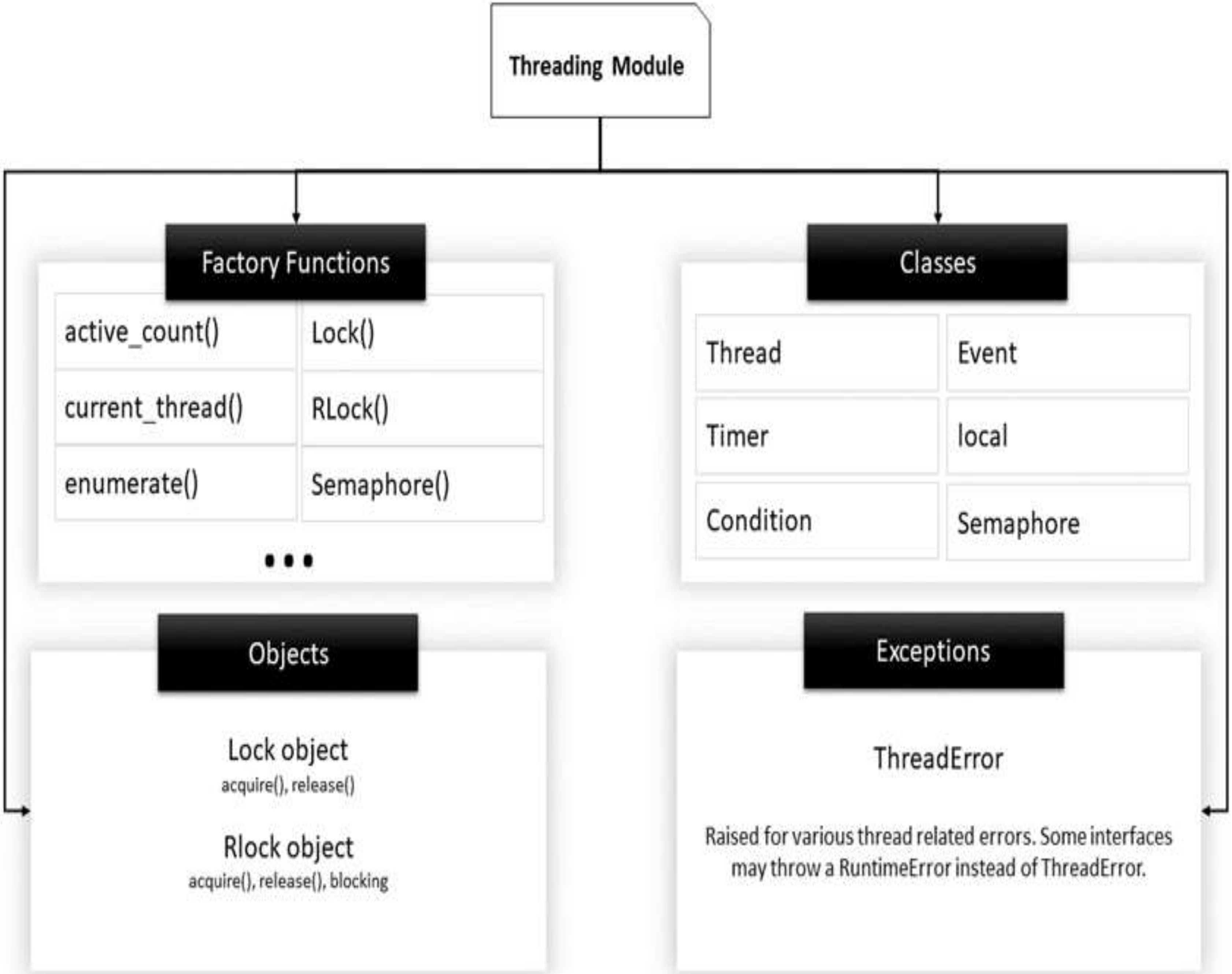
2. **threading.currentThread():** You can use it to determine the number of thread objects in the caller’s thread control.

3. **threading.enumerate():** It will give you a complete list of thread objects that are currently active

Exceptions :

ThreadError:

- ✓ Raised for various thread related errors. some interfaces may throw a RuntimeError instead of ThreadError



Threading Module

Factory Functions

active_count()	Lock()
current_thread()	RLock()
enumerate()	Semaphore()

• • •

Classes

Thread	Event
Timer	local
Condition	Semaphore

Objects

Lock object

acquire(), release()

Rlock object

acquire(), release(), blocking

Exceptions

ThreadError

Raised for various thread related errors. Some interfaces may throw a RuntimeError instead of ThreadError.

Global Interpreter Lock (GIL)

What is GIL?

✓The Python Global Interpreter Lock or GIL **is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.**

✓This means that **only one thread can be in a state of execution** at any point in time.

✓The impact of the **GIL isn’t visible to developers** who execute single-threaded programs, but it can be a performance bottleneck in **CPU-bound and multi-threaded code.**

✓Since the **GIL allows only one thread to execute at a time** even in a multi-threaded architecture with more than one CPU core.

✓A lock can be used to make sure that **only one thread** has access to a **particular resource** at a given time.

✓**every process** treats the python interpreter itself as a **resource.**

✓For example, suppose **you have written a python program** which uses **two threads** to perform **both CPU and 'I/O' operations**. When you execute this program, this is what happens:

1. The python interpreter creates a **new process and spawns the threads**
2. When **thread-1 starts running**, it will first **acquire the GIL** and lock it.
3. **If thread-2 wants to execute** now, it will have to **wait** for the GIL to be **released** even if **another processor is free.**
4. Now, **suppose thread-1 is waiting for an I/O operation**. At this time, it will **release the GIL,** and **thread-2 will acquire it.**
5. After completing the I/O ops, **if thread-1 wants to execute now**, it will again have to **wait** for the GIL to be released by thread-2.

- ✓ Due to this, **only one thread can access** the interpreter at any time, meaning that there will be only **one thread executing python code** at a given point of time.
- ✓ Execution of Python code is controlled by the **Python Virtual Machine.**
- ✓ **Access to the Python Virtual Machine** is controlled by the global interpreter lock **(GIL).** This lock is what ensures that **exactly one thread is running**.
- ✓ The **Python Virtual Machine executes** in the following manner in an MT environment:

  1. Set the GIL

  2. Switch in a thread to run

  3. Execute either ...

  a. For a specified number of bytecode instructions, or

  b. If the thread voluntarily yields control (can be accomplished time.sleep(0))

  4. Put the thread back to sleep (switch out thread)

  5. Unlock the GIL, and ...

  6. Do it all over again (lather, rinse, repeat)