

Python for scientific research

Pattern matching and text manipulation

Bram Kuijper

University of Exeter, Penryn Campus, UK

February 24, 2020



Researcher
Development



Basic features of strings

First, some **basic features** of working with strings of text in Python:

- Using quotes within strings:

```
1 str1 = "Text with 'embbded' single quotes"
```

Basic features of strings

First, some **basic features** of working with strings of text in Python:

- Using quotes within strings:

```
1 str1 = "Text with 'embbded' single quotes"  
2 str2 = 'Text with "embedded" double quotes'
```

Basic features of strings

First, some **basic features** of working with strings of text in Python:

- Using quotes within strings:

```
1 str1 = "Text with 'embbded' single quotes"
2 str2 = 'Text with "embedded" double quotes'
3 str3 = "Text with \"escaped\" double quotes" # Text
        with "escaped" double quotes
```

Basic features of strings

First, some **basic features** of working with strings of text in Python:

- Using quotes within strings:

```
1 str1 = "Text with 'embbded' single quotes"
2 str2 = 'Text with "embedded" double quotes'
3 str3 = "Text with \"escaped\" double quotes" # Text
        with "escaped" double quotes
```

- Multiline strings demarcated by triple quotes

Basic features of strings

First, some **basic features** of working with strings of text in Python:

- Using quotes within strings:

```
1 str1 = "Text with 'embbded' single quotes"
2 str2 = 'Text with "embedded" double quotes'
3 str3 = "Text with \"escaped\" double quotes" # Text
        with "escaped" double quotes
```

- Multiline strings demarcated by triple quotes

```
1 multiline = """This is a
2 multiline string"""
```

Basic features of strings

First, some **basic features** of working with strings of text in Python:

- Using quotes within strings:

```
1 str1 = "Text with 'embbeded' single quotes"
2 str2 = 'Text with "embedded" double quotes'
3 str3 = "Text with \"escaped\" double quotes" # Text
        with "escaped" double quotes
```

- Multiline strings demarcated by triple quotes

```
1 multiline = """This is a
2 multiline string"""
3 multiline2 = '''Another
4 multiline string'''
```

String literals

Different **string literals** identifying different types of string:

- By default, any string is *encoded* as **UTF-8**, allowing for international characters:

```
1 str_normal = "Let's go to Gijón!"
```


String literals

Different **string literals** identifying different types of string:

- By default, any string is *encoded* as **UTF-8**, allowing for international characters:

```
1 str_normal = "Let's go to Gijón!"  
2 str_normal = u"Let's go to Gijón!" # u-prefix, now  
   redundant (Python2)
```

String literals

Different **string literals** identifying different types of string:

- By default, any string is *encoded* as **UTF-8**, allowing for international characters:

```
1 str_normal = "Let's go to Gijón!"
2 str_normal = u"Let's go to Gijón!" # u-prefix, now
   redundant (Python2)
3 type(str_normal) # <class 'str'>
```

String literals

Different **string literals** identifying different types of string:

- By default, any string is *encoded* as **UTF-8**, allowing for international characters:

```
1 str_normal = "Let's go to Gijón!"
2 str_normal = u"Let's go to Gijón!" # u-prefix, now
   redundant (Python2)
3 type(str_normal) # <class 'str'>
```

- Byte strings (written as `b"..."`) only contain **ASCII** characters (no international characters):

```
1 str_ascii = b"Let's go to Gijón!" # Error
```

String literals

Different **string literals** identifying different types of string:

- By default, any string is *encoded* as **UTF-8**, allowing for international characters:

```
1 str_normal = "Let's go to Gijón!"
2 str_normal = u"Let's go to Gijón!" # u-prefix, now
   redundant (Python2)
3 type(str_normal) # <class 'str'>
```

- Byte strings (written as `b"..."`) only contain **ASCII** characters (no international characters):

```
1 str_ascii = b"Let's go to Gijón!" # Error
2 str_ascii = b"Let's go to Gijon!" # only ASCII
```

String literals

Different **string literals** identifying different types of string:

- By default, any string is *encoded* as **UTF-8**, allowing for international characters:

```
1 str_normal = "Let's go to Gijón!"
2 str_normal = u"Let's go to Gijón!" # u-prefix, now
   redundant (Python2)
3 type(str_normal) # <class 'str'>
```

- Byte strings (written as `b"..."`) only contain **ASCII** characters (no international characters):

```
1 str_ascii = b"Let's go to Gijón!" # Error
2 str_ascii = b"Let's go to Gijon!" # only ASCII
3 type(str_ascii) # <class 'bytes'>
```

String literals

Different **string literals** identifying different types of string:

- By default, any string is *encoded* as **UTF-8**, allowing for international characters:

```
1 str_normal = "Let's go to Gijón!"
2 str_normal = u"Let's go to Gijón!" # u-prefix, now
   redundant (Python2)
3 type(str_normal) # <class 'str'>
```

- Byte strings (written as `b"..."`) only contain **ASCII** characters (no international characters):

```
1 str_ascii = b"Let's go to Gijón!" # Error
2 str_ascii = b"Let's go to Gijon!" # only ASCII
3 type(str_ascii) # <class 'bytes'>
```

- **UTF-8** and **ASCII** are encodings which specify how characters translate into 0s and 1s

Example encoding: ASCII

USASCII code chart

<div> <div> <div>b7</div> <div>b6</div> <div>b5</div> <div>b4</div> <div>b3</div> <div>b2</div> <div>b1</div> </div> <div>Bits</div> </div> <div> <div>Column</div> <div>Row</div> </div>					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
					0	1	2	3	4	5	6	7
0	0	0	0	0	0	NUL	DLE	SP	@	P	\	p
0	0	0	1	1	1	SOH	DC1	!	A	Q	a	q
0	0	1	0	0	2	STX	DC2	"	B	R	b	r
0	0	1	1	1	3	ETX	DC3	#	C	S	c	s
0	1	0	0	0	4	EOT	DC4	\$	D	T	d	t
0	1	0	1	1	5	ENQ	NAK	%	E	U	e	u
0	1	1	0	0	6	ACK	SYN	&	F	V	f	v
0	1	1	1	1	7	BEL	ETB	'	G	W	g	w
1	0	0	0	0	8	BS	CAN	(H	X	h	x
1	0	0	1	1	9	HT	EM)	I	Y	i	y
1	0	1	0	0	10	LF	SUB	*	J	Z	j	z
1	0	1	1	1	11	VT	ESC	+	K	[k	{
1	1	0	0	0	12	FF	FS	,	L	\	l	
1	1	0	1	1	13	CR	GS	-	M]	m	}
1	1	1	0	0	14	SO	RS	.	N	^	n	~
1	1	1	1	1	15	SI	US	/	O	_	o	DEL

String literals continued

- International characters sometimes problematic, think web addresses or old filesystems/databases

String literals continued

- International characters sometimes problematic, think web addresses or old filesystems/databases
- To overcome this, you can use `str.encode()` to encode into bytes

```
1 str_var = "Let's go to Gijón!" # utf-8 string
```

String literals continued

- International characters sometimes problematic, think web addresses or old filesystems/databases
- To overcome this, you can use `str.encode()` to encode into bytes

```
1 str_var = "Let's go to Gijón!" # utf-8 string
2 str_ascii = str_var.encode() # bytes
```

String literals continued

- International characters sometimes problematic, think web addresses or old filesystems/databases
- To overcome this, you can use `str.encode()` to encode into bytes

```
1 str_var = "Let's go to Gijón!" # utf-8 string
2 str_ascii = str_var.encode() # bytes
3 # b"Let's go to Gij\xc3\xb3n!"
```

String literals continued

- International characters sometimes problematic, think web addresses or old filesystems/databases
- To overcome this, you can use `str.encode()` to encode into bytes

```
1 str_var = "Let's go to Gijón!" # utf-8 string
2 str_ascii = str_var.encode() # bytes
3 # b"Let's go to Gij\xc3\xb3n!"
```

- Here, `\xc3` and `\xb3` are [escape sequences](#) that together encode ç as a hex number (see [UTF-8 tool](#))

String literals continued

- International characters sometimes problematic, think web addresses or old filesystems/databases
- To overcome this, you can use `str.encode()` to encode into bytes

```
1 str_var = "Let's go to Gijón!" # utf-8 string
2 str_ascii = str_var.encode() # bytes
3 # b"Let's go to Gij\xc3\xb3n!"
```

- Here, `\xc3` and `\xb3` are [escape sequences](#) that together encode `ç` as a hex number (see [UTF-8 tool](#))
- The original UTF-8 string can be recovered from `bytes.decode()`

```
1 back_2_utf8 = str_ascii.decode('utf-8') # back to UTF-8
2 # "Let's go to Gijón!"
```

Escaping the escape sequences

- Sometimes we do not want `\` to be interpreted as an escape sequence:

```
1 windows_path = "C:\new_file.csv"  
2 # C:  
3 # ew_file.csv
```

Escaping the escape sequences

- Sometimes we do not want `\` to be interpreted as an **escape sequence**:

```
1 windows_path = "C:\new_file.csv"  
2 # C:  
3 # ew_file.csv
```

- `\n` is interpreted as a **newline character**

Escaping the escape sequences

- Sometimes we do not want `\` to be interpreted as an **escape sequence**:

```
1 windows_path = "C:\new_file.csv"  
2 # C:  
3 # ew_file.csv
```

- `\n` is interpreted as a **newline character**
- We can prevent this by writing another backslash:

```
1 windows_path = "C:\\new_file.csv"  
2 # C:\new_file.csv
```


Escaping the escape sequences

- Sometimes we do not want `\` to be interpreted as an **escape sequence**:

```
1 windows_path = "C:\new_file.csv"  
2 # C:  
3 # ew_file.csv
```

- `\n` is interpreted as a **newline character**
- We can prevent this by writing another backslash:

```
1 windows_path = "C:\\new_file.csv"  
2 # C:\new_file.csv
```

- or by using a raw string literal (prefix: `r"..."`)

```
1 windows_path = r"C:\new_file.csv"  
2 # C:\new_file.csv
```

Various string methods

- Lots of [string methods](#) available. Some examples:

Various string methods

- Lots of [string methods](#) available. Some examples:
 - Split strings in words

```
1 string1 = "Split this string up"  
2 string1.split()  
3 #["Split","this","string","up"]
```

Various string methods

- Lots of **string methods** available. Some examples:
 - Split strings in words

```
1 string1 = "Split this string up"
2 string1.split()
3 #["Split","this","string","up"]
```

- Join a list of words

```
1 list_of_words = ["Join","me","together!"]
2 "--".join(list_of_words)
3 # Join--me--together!
```

Various string methods

- Lots of [string methods](#) available. Some examples:

- Split strings in words

```
1 string1 = "Split this string up"
2 string1.split()
3 # ["Split", "this", "string", "up"]
```

- Join a list of words

```
1 list_of_words = ["Join", "me", "together!"]
2 "--".join(list_of_words)
3 # Join--me--together!
```

- Find/replace substrings

```
1 str_subject = "Great rockpools at Swanpool beach"
2 str_subject.find("pool") # 10
3 str_subject.rfind("pool") # 23
4 str_subject.replace("pool", "puddle") # "Great
    rockpuddles at Swanpuddle beach"
```

Various string methods

- Lots of [string methods](#) available. Some examples:

- Split strings in words

```
1 string1 = "Split this string up"
2 string1.split()
3 # ["Split", "this", "string", "up"]
```

- Join a list of words

```
1 list_of_words = ["Join", "me", "together!"]
2 "--".join(list_of_words)
3 # Join--me--together!
```

- Find/replace substrings

```
1 str_subject = "Great rockpools at Swanpool beach"
2 str_subject.find("pool") # 10
3 str_subject.rfind("pool") # 23
4 str_subject.replace("pool", "puddle") # "Great
   rockpuddles at Swanpuddle beach"
```

String `isX` methods

- Identifying string contents using various `str.isX()` functions

String isX methods

- Identifying string contents using various `str.isX()` functions
 - All characters in the string are numeric

```
1 string1 = "899898"  
2 string1.isnumeric() # True
```


String isX methods

- Identifying string contents using various `str.isX()` functions
 - All characters in the string are numeric

```
1 string1 = "899898"  
2 string1.isnumeric() # True  
3 string2 = "8998.98"  
4 string2.isnumeric() # False
```

String isX methods

- Identifying string contents using various `str.isX()` functions
 - All characters in the string are numeric

```
1 string1 = "899898"  
2 string1.isnumeric() # True  
3 string2 = "8998.98"  
4 string2.isnumeric() # False
```

- All characters in the string are alphabetic

```
1 string1 = "Thisisallalphabetic"  
2 string1.isalpha() # True
```

String isX methods

- Identifying string contents using various `str.isX()` functions
 - All characters in the string are numeric

```
1 string1 = "899898"  
2 string1.isnumeric() # True  
3 string2 = "8998.98"  
4 string2.isnumeric() # False
```

- All characters in the string are alphabetic

```
1 string1 = "Thisisallalphabetic"  
2 string1.isalpha() # True  
3 string2 = "Now with whitespace"  
4 string2.isalpha() # False
```

String isX methods

- Identifying string contents using various `str.isX()` functions

- All characters in the string are numeric

```
1 string1 = "899898"  
2 string1.isnumeric() # True  
3 string2 = "8998.98"  
4 string2.isnumeric() # False
```

- All characters in the string are alphabetic

```
1 string1 = "Thisisallalphabetic"  
2 string1.isalpha() # True  
3 string2 = "Now with whitespace"  
4 string2.isalpha() # False
```

- Lots of other `str.isX()` functions available. As we see later, however, regular expressions often preferable to search for patterns in text

Finding patterns of text without regular expressions

- Imagine one wants to convert various date formats to YYYY-MM-DD

Finding patterns of text without regular expressions

- Imagine one wants to convert various date formats to YYYY-MM-DD

```
1 s = "23.01.1980,08.09.1990,15-03-2019"
```

Finding patterns of text without regular expressions

- Imagine one wants to convert various date formats to YYYY-MM-DD

```
1 s = "23.01.1980,08.09.1990,15-03-2019"
2
3 for i in range(len(s)):
4
5     if i + 10 <= len(s):
6         if s[i:i+2].isdigit() and s[i+2] in ".-" and s[
            i+3:i+5].isdigit() and s[i+5] in ".-" and s[
                i+6:i+10].isdigit():
```

Finding patterns of text without regular expressions

- Imagine one wants to convert various date formats to YYYY-MM-DD

```
1 s = "23.01.1980,08.09.1990,15-03-2019"
2
3 for i in range(len(s)):
4
5     if i + 10 <= len(s):
6         if s[i:i+2].isdigit() and s[i+2] in ".-" and s[
7             i+3:i+5].isdigit() and s[i+5] in ".-" and s[
8                 i+6:i+10].isdigit():
9             day = s[i:i+2]
10            month = s[i+3:i+5]
11            year = s[i+6:i+10]
12            print(year + "-" + month + "-" + day)
```


Finding patterns of text without regular expressions

- Imagine one wants to convert various date formats to YYYY-MM-DD

```
1 s = "23.01.1980,08.09.1990,15-03-2019"
2
3 for i in range(len(s)):
4
5     if i + 10 <= len(s):
6         if s[i:i+2].isdigit() and s[i+2] in ".-" and s[
7             i+3:i+5].isdigit() and s[i+5] in ".-" and s[
8                 i+6:i+10].isdigit():
9             day = s[i:i+2]
10            month = s[i+3:i+5]
11            year = s[i+6:i+10]
12            print(year + "-" + month + "-" + day)
```

- Gets complicated quickly
- Breaks down for single digit months/days, e.g., 8.9.1980

Finding patterns of text with regular expressions

```
1 # load the regular expression module
2 import re
```

Finding patterns of text with regular expressions

```
1 # load the regular expression module
2 import re
3
4 # text with dates (with single digit days and months)
5 s = "23.01.1980,8.9.1990,15-03-2019"
```

Finding patterns of text with regular expressions

```
1 # load the regular expression module
2 import re
3
4 # text with dates (with single digit days and months)
5 s = "23.01.1980,8.9.1990,15-03-2019"
6
7 # regular expression (given as a r" [raw literal] string)
8 all_dates = re.findall(pattern=r"(\d{1,2})[-.](\d{1,2})
    [-.](\d{4})", string=s)
```

Finding patterns of text with regular expressions

```
1 # load the regular expression module
2 import re
3
4 # text with dates (with single digit days and months)
5 s = "23.01.1980,8.9.1990,15-03-2019"
6
7 # regular expression (given as a r"" [raw literal] string)
8 all_dates = re.findall(pattern=r"(\d{1,2})[-.](\d{1,2})
    [-.](\d{4})", string=s)
9
10 # print the result
11 for date in all_dates:
12     print(date[2] + "-" + date[1].zfill(2) + "-" + date[0].
        zfill(2))
```

What is a regular expression?

- A tiny, highly specialized programming language within Python

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module
- Specifies the rules to match (and replace) patterns in text

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module
- Specifies the rules to match (and replace) patterns in text
- One line of regex can replace 100s of lines of procedural code

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module
- Specifies the rules to match (and replace) patterns in text
- One line of regex can replace 100s of lines of procedural code
- More portable across different programming languages than `str` methods

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module
- Specifies the rules to match (and replace) patterns in text
- One line of regex can replace 100s of lines of procedural code
- More portable across different programming languages than `str` methods
- Easy to create using trial and error, for example on <https://regex101.com>

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module
- Specifies the rules to match (and replace) patterns in text
- One line of regex can replace 100s of lines of procedural code
- More portable across different programming languages than `str` methods
- Easy to create using trial and error, for example on <https://regex101.com>
- A simple example:

```
1 import re
```

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module
- Specifies the rules to match (and replace) patterns in text
- One line of regex can replace 100s of lines of procedural code
- More portable across different programming languages than `str` methods
- Easy to create using trial and error, for example on <https://regex101.com>
- A simple example:

```
1 import re
2 str_to_match = "Factors: 1foo, 2foo, foo, 4bar"
```

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module
- Specifies the rules to match (and replace) patterns in text
- One line of regex can replace 100s of lines of procedural code
- More portable across different programming languages than `str` methods
- Easy to create using trial and error, for example on <https://regex101.com>
- A simple example:

```
1 import re
2 str_to_match = "Factors: 1foo, 2foo, foo, 4bar"
3 # regex that matches '1foo', '2foo' etc but not 'foo'
```

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module
- Specifies the rules to match (and replace) patterns in text
- One line of regex can replace 100s of lines of procedural code
- More portable across different programming languages than `str` methods
- Easy to create using trial and error, for example on <https://regex101.com>
- A simple example:

```
1 import re
2 str_to_match = "Factors: 1foo, 2foo, foo, 4bar"
3 # regex that matches '1foo', '2foo' etc but not 'foo'
4 regex = r"\dfoo"
```

What is a regular expression?

- A tiny, highly specialized programming language within Python
- Made available in the `re` module
- Specifies the rules to match (and replace) patterns in text
- One line of regex can replace 100s of lines of procedural code
- More portable across different programming languages than `str` methods
- Easy to create using trial and error, for example on <https://regex101.com>
- A simple example:

```
1 import re
2 str_to_match = "Factors: 1foo, 2foo, foo, 4bar"
3 # regex that matches '1foo', '2foo' etc but not 'foo'
4 regex = r"\dfoo"
5 print(re.findall(pattern=regex, string=str_to_match)) #
    ['1foo', '2foo']
```


Testing regular expressions

- Practice regular expressions at <https://regex101.com/>

The screenshot shows the regex101.com website in a browser window. The page title is "Online regex tester and debugger: PHP, PCRE, Python, Golang and JavaScript - Mozilla Firefox". The URL bar shows "https://regex101.com". The page has a blue header with the text "regular expressions" and navigation links: "@regex101", "donate", "contact", "bug reports & feedback", and "wiki".

The main content area is divided into several sections:

- REGULAR EXPRESSION**: A text input field containing the regex "[0-9]". A red arrow points to this field with the text "Enter regex here".
- TEST STRING**: A text input field containing the test string "23.09.1988". A red arrow points to this field with the text "Provide test strings".
- SUBSTITUTION**: A text input field containing the substitution pattern "n|". A red arrow points to this field with the text "Provide substitutions".
- EXPLANATION**: A section on the right side of the page that provides details about the regex. It shows that "[0-9]" matches a digit (equal to "[0-9]"). It also lists "Global pattern flags" and explains the "modifier: global" and "modifier: multi line".
- MATCH INFORMATION**: A section on the right side of the page that shows the matches found. It lists "Match 1", "Match 2", and "Match 3", each with a "Full match" and a "Match" value.
- QUICK REFERENCE**: A section on the right side of the page that provides a quick reference for common regex tokens. It includes a search bar and a list of tokens: "All Tokens", "Common Tokens" (checked), "General Tokens", "Anchors", and "Meta Sequences".

The bottom of the page features a navigation bar with icons for navigation and search.

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit
- `\s` matches any whitespace character (e.g., a space, a tab, a newline)

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit
- `\s` matches any whitespace character (e.g., a space, a tab, a newline)
- `\S` matches any character that is not a whitespace

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit
- `\s` matches any whitespace character (e.g., a space, a tab, a newline)
- `\S` matches any character that is not a whitespace
- `\b` matches a word boundary

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit
- `\s` matches any whitespace character (e.g., a space, a tab, a newline)
- `\S` matches any character that is not a whitespace
- `\b` matches a word boundary

```
1 str_to_match = "Factor levels are snafoo, foosna and  
    foo"
```

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit
- `\s` matches any whitespace character (e.g., a space, a tab, a newline)
- `\S` matches any character that is not a whitespace
- `\b` matches a word boundary

```
1 str_to_match = "Factor levels are snafoo, foosna and  
   foo"  
2 regex = r"\bfoo\b" # regex
```


Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit
- `\s` matches any whitespace character (e.g., a space, a tab, a newline)
- `\S` matches any character that is not a whitespace
- `\b` matches a word boundary

```
1 str_to_match = "Factor levels are snafoo, foosna and  
   foo"  
2 regex = r"\bfoo\b" # regex  
3 m = re.search(pattern=regex, string=str_to_match) #  
   Factor levels are snafoo, foosna and bar
```

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit
- `\s` matches any whitespace character (e.g., a space, a tab, a newline)
- `\S` matches any character that is not a whitespace
- `\b` matches a word boundary

```
1 str_to_match = "Factor levels are snafoo, foosna and  
   foo"  
2 regex = r"\bfoo\b" # regex  
3 m = re.search(pattern=regex, string=str_to_match) #  
   Factor levels are snafoo, foosna and bar  
4 m.group(0) # obtain the complete match
```

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit
- `\s` matches any whitespace character (e.g., a space, a tab, a newline)
- `\S` matches any character that is not a whitespace
- `\b` matches a word boundary

```
1 str_to_match = "Factor levels are snafoo, foosna and  
   foo"  
2 regex = r"\bfoo\b" # regex  
3 m = re.search(pattern=regex, string=str_to_match) #  
   Factor levels are snafoo, foosna and bar  
4 m.group(0) # obtain the complete match  
5 m.start() # Match position in string: 37
```

Regular expressions: syntax

The **syntax** for different patterns:

- `\d` matches any character that is a digit
- `\D` matches any character that is not a digit
- `\s` matches any whitespace character (e.g., a space, a tab, a newline)
- `\S` matches any character that is not a whitespace
- `\b` matches a word boundary

```
1 str_to_match = "Factor levels are snafoo, foosna and  
    foo"  
2 regex = r"\bfoo\b" # regex  
3 m = re.search(pattern=regex, string=str_to_match) #  
    Factor levels are snafoo, foosna and bar  
4 m.group(0) # obtain the complete match  
5 m.start() # Match position in string: 37
```

- `\B` does not match a word boundary

Regular expressions: syntax II

The **syntax** for different patterns:

- `.` matches any character (except a newline)

Regular expressions: syntax II

The **syntax** for different patterns:

- . matches any character (except a newline)
- ^ matches the start of a string

Regular expressions: syntax II

The **syntax** for different patterns:

- . matches any character (except a newline)
- ^ matches the start of a string

```
1 str1 = "foo snafoo funfoo"  
2 regex = r"^foo" # regex matching the first foo
```

Regular expressions: syntax II

The [syntax](#) for different patterns:

- `.` matches any character (except a newline)
- `^` matches the start of a string

```
1 str1 = "foo snafoo funfoo"
2 regex = r"^foo" # regex matching the first foo
3 m = re.search(pattern=regex, string=str1)
4 m.start() # match position in string: 0
```


Regular expressions: syntax II

The **syntax** for different patterns:

- `.` matches any character (except a newline)
- `^` matches the start of a string

```
1 str1 = "foo snafoo funfoo"
2 regex = r"^foo" # regex matching the first foo
3 m = re.search(pattern=regex, string=str1)
4 m.start() # match position in string: 0
```

- `$` matches end of a string

```
1 str1 = "foo snafoo funfoo"
2 regex = r"foo$" # regex matching the last foo
```

Regular expressions: syntax II

The **syntax** for different patterns:

- `.` matches any character (except a newline)
- `^` matches the start of a string

```
1 str1 = "foo snafoo funfoo"
2 regex = r"^foo" # regex matching the first foo
3 m = re.search(pattern=regex, string=str1)
4 m.start() # match position in string: 0
```

- `$` matches end of a string

```
1 str1 = "foo snafoo funfoo"
2 regex = r"foo$" # regex matching the last foo
3 m = re.search(pattern=regex, string=str1)
4 m.start() # match position in string: 14
```

Regular expressions: syntax II

The **syntax** for different patterns:

- `.` matches any character (except a newline)
- `^` matches the start of a string

```
1 str1 = "foo snafoo funfoo"
2 regex = r"^foo" # regex matching the first foo
3 m = re.search(pattern=regex, string=str1)
4 m.start() # match position in string: 0
```

- `$` matches end of a string

```
1 str1 = "foo snafoo funfoo"
2 regex = r"foo$" # regex matching the last foo
3 m = re.search(pattern=regex, string=str1)
4 m.start() # match position in string: 14
```

Regular expressions: syntax III

The [syntax](#) for different patterns:

- [...] matches a range of characters

```
1 str1 = "the number 60 is larger than 59"
2 regex = r"[0-5][0-9]" # matches 00 to 59
```

Regular expressions: syntax III

The [syntax](#) for different patterns:

- [...] matches a range of characters

```
1 str1 = "the number 60 is larger than 59"
2 regex = r"[0-5][0-9]" # matches 00 to 59
3 m = re.search(pattern=regex, string=str1)
4 m.group(0) # '59'
```

- [^...] matches characters not in the range

```
1 seq1 = "cccggttaaccg"
```

Regular expressions: syntax III

The [syntax](#) for different patterns:

- [...] matches a range of characters

```
1 str1 = "the number 60 is larger than 59"
2 regex = r"[0-5][0-9]" # matches 00 to 59
3 m = re.search(pattern=regex, string=str1)
4 m.group(0) # '59'
```

- [^...] matches characters not in the range

```
1 seq1 = "cccgggtaaccg"
2 regex = r"[^cg]" # do not match c or g
```

Regular expressions: syntax III

The [syntax](#) for different patterns:

- [...] matches a range of characters

```
1 str1 = "the number 60 is larger than 59"
2 regex = r"[0-5][0-9]" # matches 00 to 59
3 m = re.search(pattern=regex, string=str1)
4 m.group(0) # '59'
```

- [^...] matches characters not in the range

```
1 seq1 = "cccgggtaaccg"
2 regex = r"[^cg]" # do not match c or g
3 m = re.search(pattern=regex, string=seq1)
4 m.group(0) # 't', first match when using re.search()
```

Regular expressions: repetitions

Specify **number of times** patterns are matched

Regular expressions: repetitions

Specify **number of times** patterns are matched

- * matches preceding regex 0 or more times

Regular expressions: repetitions

Specify **number of times** patterns are matched

- * matches preceding regex 0 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d*" # matches 0 or more occurrences of
   numbers
```

Regular expressions: repetitions

Specify **number of times** patterns are matched

- * matches preceding regex 0 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d*" # matches 0 or more occurrences of
   numbers
3 re.findall(pattern=regex, string=str1) # [' ', ' ', ' ',
   ' ', ' ', ' ', ' ', ' ', ' ', '60', ' ', '500', ' ', ' ', ' ', ' ',
   ' ', '3000', '']
```

Regular expressions: repetitions

Specify **number of times** patterns are matched

- * matches preceding regex 0 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d*" # matches 0 or more occurrences of
   numbers
3 re.findall(pattern=regex, string=str1) # [' ', ' ', ' ',
   ' ', ' ', ' ', ' ', ' ', ' ', '60', ' ', '500', ' ', ' ', ' ', ' ',
   ' ', '3000', ' ']
```

- + matches preceding regex 1 or more times

Regular expressions: repetitions

Specify **number of times** patterns are matched

- ***** matches preceding regex 0 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d*" # matches 0 or more occurrences of
   numbers
3 re.findall(pattern=regex, string=str1) # [' ', ' ', ' ',
   ' ', ' ', ' ', ' ', ' ', ' ', '60', ' ', '500', ' ', ' ', ' ', ' ',
   ' ', '3000', ' ']
```

- **+** matches preceding regex 1 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d+" # matches 1 or more occurrences of
   numbers
```

Regular expressions: repetitions

Specify **number of times** patterns are matched

- * matches preceding regex 0 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d*" # matches 0 or more occurrences of
   numbers
3 re.findall(pattern=regex, string=str1) # ['', '', '',
   '', '', '', '', '60', '', '500', '', '', '', '',
   '', '3000', '']
```

- + matches preceding regex 1 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d+" # matches 1 or more occurrences of
   numbers
3 re.findall(pattern=regex, string=str1) # ['60', '500',
   '3000']
```

Regular expressions: repetitions

Specify **number of times** patterns are matched

- * matches preceding regex 0 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d*" # matches 0 or more occurrences of
   numbers
3 re.findall(pattern=regex, string=str1) # ['', '', '',
   '', '', '', '', '60', '', '500', '', '', '', '',
   '', '3000', '']
```

- + matches preceding regex 1 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d+" # matches 1 or more occurrences of
   numbers
3 re.findall(pattern=regex, string=str1) # ['60', '500',
   '3000']
```

- ? matches preceding regex 0 or 1 times

Regular expressions: repetitions

Specify **number of times** patterns are matched

- ***** matches preceding regex 0 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d*" # matches 0 or more occurrences of
  numbers
3 re.findall(pattern=regex, string=str1) # [' ', ' ', ' ',
  ' ', ' ', ' ', ' ', ' ', ' ', '60', ' ', '500', ' ', ' ', ' ', ' ',
  ' ', '3000', ' ']
```

- **+** matches preceding regex 1 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d+" # matches 1 or more occurrences of
  numbers
3 re.findall(pattern=regex, string=str1) # ['60', '500',
  '3000']
```

- **?** matches preceding regex 0 or 1 times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d?" # matches 0 or 1 occurrences of numbers
```


Regular expressions: repetitions

Specify **number of times** patterns are matched

- ***** matches preceding regex 0 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d*" # matches 0 or more occurrences of
  numbers
3 re.findall(pattern=regex, string=str1) # ['', '', '',
  '', '', '', '', '60', '', '500', '', '', '', '',
  '', '3000', '']
```

- **+** matches preceding regex 1 or more times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d+" # matches 1 or more occurrences of
  numbers
3 re.findall(pattern=regex, string=str1) # ['60', '500',
  '3000']
```

- **?** matches preceding regex 0 or 1 times

```
1 str1 = "numbers 60, 500 and 3000"
2 regex = r"\d?" # matches 0 or 1 occurrences of numbers
3 re.findall(pattern=regex, string=str1) # ['', '', '',
  '', '', '', '', '6', '0', '', '', '5', '0', '0',
  '', '', '', '', '', '3', '0', '0', '0', '']
```

Regular expressions: repetitions continued

Specify **number of times** patterns are matched (continued)

Regular expressions: repetitions continued

Specify **number of times** patterns are matched (continued)

- `{n}` match preceding regex exactly n times

Regular expressions: repetitions continued

Specify **number of times** patterns are matched (continued)

- $\{n\}$ match preceding regex exactly n times
- $\{n,m\}$ match preceding regex minimally n and maximally m times

Regular expressions: repetitions continued

Specify **number of times** patterns are matched (continued)

- $\{n\}$ match preceding regex exactly n times
- $\{n,m\}$ match preceding regex minimally n and maximally m times

```
1 str1 = "numbers 5, 60, 500, 3000, 50000"  
2 regex = r"\d{2,4}" # matches numbers of 2 to 4 digits
```

Regular expressions: repetitions continued

Specify **number of times** patterns are matched (continued)

- `{n}` match preceding regex exactly `n` times
- `{n,m}` match preceding regex minimally `n` and maximally `m` times

```
1 str1 = "numbers 5, 60, 500, 3000, 50000"
2 regex = r"\d{2,4}" # matches numbers of 2 to 4 digits
3 re.findall(pattern=regex, string=str1) # ['60', '500',
    '3000', '5000']
```

Regular expressions: repetitions continued

Specify **number of times** patterns are matched (continued)

- `{n}` match preceding regex exactly `n` times
- `{n,m}` match preceding regex minimally `n` and maximally `m` times

```
1 str1 = "numbers 5, 60, 500, 3000, 50000"
2 regex = r"\d{2,4}" # matches numbers of 2 to 4 digits
3 re.findall(pattern=regex, string=str1) # ['60', '500',
    '3000', '5000']
```

- `*?`, `+?`, `??`, `{m,n}?` minimize the number of times a pattern matches

Regular expressions: repetitions continued

Specify **number of times** patterns are matched (continued)

- `{n}` match preceding regex exactly `n` times
- `{n,m}` match preceding regex minimally `n` and maximally `m` times

```
1 str1 = "numbers 5, 60, 500, 3000, 50000"
2 regex = r"\d{2,4}" # matches numbers of 2 to 4 digits
3 re.findall(pattern=regex, string=str1) # ['60', '500',
    '3000', '5000']
```

- `*?`, `+?`, `??`, `{m,n}?` minimize the number of times a pattern matches

```
1 str1 = "numbers 5, 60, 500, 3000, 50000"
2 regex = r"\d{2,4}?" # matches numbers of 2 to 4 digits,
    favoring minimal numbers
```


Regular expressions: repetitions continued

Specify **number of times** patterns are matched (continued)

- `{n}` match preceding regex exactly `n` times
- `{n,m}` match preceding regex minimally `n` and maximally `m` times

```
1 str1 = "numbers 5, 60, 500, 3000, 50000"
2 regex = r"\d{2,4}" # matches numbers of 2 to 4 digits
3 re.findall(pattern=regex, string=str1) # ['60', '500',
    '3000', '5000']
```

- `*?`, `+?`, `??`, `{m,n}?` minimize the number of times a pattern matches

```
1 str1 = "numbers 5, 60, 500, 3000, 50000"
2 regex = r"\d{2,4}?" # matches numbers of 2 to 4 digits,
    favoring minimal numbers
3 re.findall(pattern=regex, string=str1) # ['60', '50',
    '30', '00', '50', '00']
```

Regular expressions: repetitions continued

Specify **number of times** patterns are matched (continued)

- `{n}` match preceding regex exactly `n` times
- `{n,m}` match preceding regex minimally `n` and maximally `m` times

```
1 str1 = "numbers 5, 60, 500, 3000, 50000"
2 regex = r"\d{2,4}" # matches numbers of 2 to 4 digits
3 re.findall(pattern=regex, string=str1) # ['60', '500',
    '3000', '5000']
```

- `*?`, `+?`, `??`, `{m,n}?` minimize the number of times a pattern matches

```
1 str1 = "numbers 5, 60, 500, 3000, 50000"
2 regex = r"\d{2,4}?" # matches numbers of 2 to 4 digits,
    favoring minimal numbers
3 re.findall(pattern=regex, string=str1) # ['60', '50',
    '30', '00', '50', '00']
```

Regular expressions: functions

- `re.compile()` compiles a regular expression before applying it. Speeds things up when same regex is used a lot of times

Regular expressions: functions

- `re.compile()` compiles a regular expression before applying it. Speeds things up when same regex is used a lot of times
- `re.findall()` matches all occurrences of a pattern

Regular expressions: functions

- `re.compile()` compiles a regular expression before applying it. Speeds things up when same regex is used a lot of times
- `re.findall()` matches all occurrences of a pattern
- `re.search()` matches the first occurrence of a pattern

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
```

Regular expressions: functions

- `re.compile()` compiles a regular expression before applying it. Speeds things up when same regex is used a lot of times
- `re.findall()` matches all occurrences of a pattern
- `re.search()` matches the first occurrence of a pattern

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
2 # pattern that matches '1foo', '2foo' etc but not 'foo'
3 regex = r"\dfoo" # regex
```

Regular expressions: functions

- `re.compile()` compiles a regular expression before applying it. Speeds things up when same regex is used a lot of times
- `re.findall()` matches all occurrences of a pattern
- `re.search()` matches the first occurrence of a pattern

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
2 # pattern that matches '1foo', '2foo' etc but not 'foo'
3 regex = r"\dfoo" # regex
4 m = re.search(pattern=regex, string=str_to_match) #
   returns a Match object
```

Regular expressions: functions

- `re.compile()` compiles a regular expression before applying it. Speeds things up when same regex is used a lot of times
- `re.findall()` matches all occurrences of a pattern
- `re.search()` matches the first occurrence of a pattern

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
2 # pattern that matches '1foo', '2foo' etc but not 'foo'
3 regex = r"\dfoo" # regex
4 m = re.search(pattern=regex, string=str_to_match) #
   returns a Match object
5
6 m.group(0) # '1foo'
7
8 regex = r"\dbar" # another regex which does not match
```


Regular expressions: functions

- `re.compile()` compiles a regular expression before applying it. Speeds things up when same regex is used a lot of times
- `re.findall()` matches all occurrences of a pattern
- `re.search()` matches the first occurrence of a pattern

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
2 # pattern that matches '1foo', '2foo' etc but not 'foo'
3 regex = r"\dfoo" # regex
4 m = re.search(pattern=regex, string=str_to_match) #
   returns a Match object
5
6 m.group(0) # '1foo'
7
8 regex = r"\dbar" # another regex which does not match
9 re.search(pattern=regex, string=str_to_match) # None
```

Regular expressions: functions II

- `re.sub()` replaces occurrences of patterns in a string

Regular expressions: functions II

- `re.sub()` replaces occurrences of patterns in a string

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
```

Regular expressions: functions II

- `re.sub()` replaces occurrences of patterns in a string

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
2 # pattern that matches '1foo', '2foo' etc but not 'foo'
3 # and remembers the digit using a group ()
4 regex = r"(\d)foo" # regex
```

Regular expressions: functions II

- `re.sub()` replaces occurrences of patterns in a string

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
2 # pattern that matches '1foo', '2foo' etc but not 'foo'
3 # and remembers the digit using a group ()
4 regex = r"(\d)foo" # regex
5 # replace foo by bar but keep the digit
6 replacement = r"\1bar" # regex
```

Regular expressions: functions II

- `re.sub()` replaces occurrences of patterns in a string

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
2 # pattern that matches '1foo', '2foo' etc but not 'foo'
3 # and remembers the digit using a group ()
4 regex = r"(\d)foo" # regex
5 # replace foo by bar but keep the digit
6 replacement = r"\1bar" # regex
7 re.sub(pattern=regex, repl=replacement, string=
    str_to_match) # Factor levels are 1bar, 2bar and foo
```

Regular expressions: functions II

- `re.sub()` replaces occurrences of patterns in a string

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
2 # pattern that matches '1foo', '2foo' etc but not 'foo'
3 # and remembers the digit using a group ()
4 regex = r"(\d)foo" # regex
5 # replace foo by bar but keep the digit
6 replacement = r"\1bar" # regex
7 re.sub(pattern=regex, repl=replacement, string=
    str_to_match) # Factor levels are 1bar, 2bar and foo
```

- Everything within `()` (a group) is stored in memory

Regular expressions: functions II

- `re.sub()` replaces occurrences of patterns in a string

```
1 str_to_match = "Factor levels are 1foo, 2foo and foo"
2 # pattern that matches '1foo', '2foo' etc but not 'foo'
3 # and remembers the digit using a group ()
4 regex = r"(\d)foo" # regex
5 # replace foo by bar but keep the digit
6 replacement = r"\1bar" # regex
7 re.sub(pattern=regex, repl=replacement, string=
    str_to_match) # Factor levels are 1bar, 2bar and foo
```

- Everything within `()` (a group) is stored in memory
- Group contents can be recalled in the replacement, using `\1, \2, \3`, etc

- Use <https://regex101.com/> to write dates in "23.01.1980,29-03-2019" as "1980-01-23,2019-03-29"

Regex exercise

- Use <https://regex101.com/> to write dates in "23.01.1980,29-03-2019" as "1980-01-23,2019-03-29"
- Then try to do it in Python

- Use <https://regex101.com/> to write dates in "23.01.1980,29-03-2019" as "1980-01-23,2019-03-29"
- Then try to do it in Python
- In Python, that is:

```
1 two_dates = "23.01.1980,29-03-2019"
2 date_regex = r"(\d{1,2})[. -](\d{1,2})[. -](\d{4})"
3 date_substitution = r"\3-\2-\1"
4 re.sub(pattern=date_regex, repl=date_substitution,
5         string=two_dates)
6 # 23-01-1980,29-3-2019
```

Another regex exercise

- Use <https://regex101.com/> to match the words 'pit', 'spot', 'spate', but not 'pt', 'Pot', 'peat', 'part'

Another regex exercise

- Use <https://regex101.com/> to match the words 'pit', 'spot', 'spate', but not 'pt', 'Pot', 'peat', 'part'

```
1 regex = r"s?p(i|o|a)te?"
```