

Python Programming for Data Science

WEEK 38, FRIDAY

REGULAR EXPRESSIONS

REGULAR EXPRESSIONS IN PYTHON

Recap: modules: a popquiz

- How do you create a Python module?
- What is the difference between:
`import ...`
and
`from ... import ...?`
- What does the `as` keyword do?
- What is a *package*?

A few comments regarding handin2 (1)

Common mistakes:

- Forgetting to close the file
- Using built-in functions as variable names (e.g. input)
- Looking for first character in line before checking it is not empty
- Hardcoding assumptions about the data into the code

```
for index in range(13, 119):  
    ...
```

and

```
container2 = {container1[0][0]: container1[0][1], ...}
```

and

```
print('Error: ... not found. Available names: ["Paul  
McCartney", "John Lennon", ...])'
```

A few comments regarding handin2 (2)

Some advice:

- Default arguments of `split()` and `strip()` are often sufficient
- Are you sure you need indices when iterating over a loop?

```
for index in range(len(lines)):  
    print(lines[index])
```

- To check if an element is in a dictionary, you can write

```
if "Mick Jagger" in my_dict
```

instead of

```
if "Mick Jagger" in my_dict.keys()
```

Regular expressions

Scenario: you receive some data...

```
# Measurements started
Sep 9, 9:05, T=22deg
SEP 9, 10:15, T=25deg
# Taking a coffee break
Sep 9, 11:15, T=-10deg
# Weekend
Sept 12, 09:00AM, T=32deg
Oct12 13:00, T=32degr
```

The data file names contain a temperature measurement and when it was recorded.

How do we select only entries before September 10 for which the temperature was positive?

Can this be done with our current Python knowledge?

Parsing text

```
data = '''# Measurements started
Sep 9, 9:05, T=22deg
SEP 9, 10:15, T=25deg
# Taking a coffee break
Sep 9, 11:15, T=-10deg
# Weekend
Sept 12, 09:00AM, T=32deg
Oct12 13:00, T=32degr'''

data_lines = data.split('\n')

for data_line in data_lines:
    if ???:
        print(data_line)
```

How do we proceed?

Parsing text (2)

```
...
for data_line in data_lines:
    if data_line[0] == "#":
        continue                                # Skip lines starting
with #
    index = 0
    while data_line[index].isalpha():
        index += 1                            # Move ahead on
letters
    month_name = data_line[:index]      # extract month name
    if data_line[index] == " ":
        index += 1                            # Jump passed space
    data_line = data_line[index:]      # Remove month name
from data_line
    index = 0
    while data_line[index].isdigit():
        index += 1                            # Move ahead on
letters
    day_number = int(data_line[:index])  # extract day
number
    ...
...
```

Complicated. Can we do better?

Idea: we specify a single *text pattern* that matches exactly what we are interested in.

Regular expressions

Regular expressions can be used to specify *text patterns*. You can then use this pattern to *match* a string, or *search* in a string.

The good news: You've already used regular expressions: Any normal search string is automatically also a regular expression.

Regular expressions - Ingredients 1

.	Match any character
a	Match 'a' (normal text)
ab	Match 'a' followed by 'b' (normal text)
[abc]	Character class. Match one character: either 'a' or 'b' or 'c'
[^abc]	Character class. Match one character: anything except 'a' or 'b' or 'c'
[a-z]	Character class. Match one character: anything between 'a' and 'z'
^ and \$	Matches beginning and end of line
\A and \Z	Matches beginning and end of entire string

Regular expressions - Examples

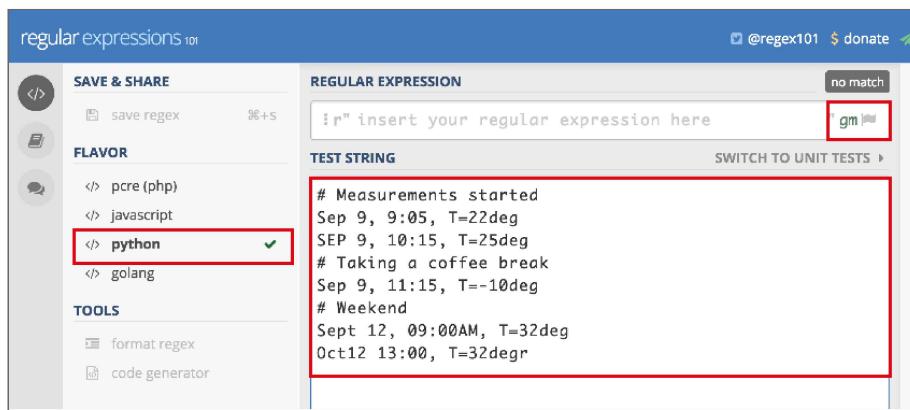
<u>pattern</u>	<u>examples of matching strings</u>
[0-9]	'0' '2'
[a-zA-Z]	'H' 'f'
.ellow	'yellow' 'mellow'
.[ei]llow	'pillow' 'yellow'
[ab0-9]	'a' 'b' '4'

regular expressions - exercise 1

There is an free regular expression tester online:

regex101.com

Click on Python in the left menu, make sure the option field states "gm", and copy the data lines into the TEST STRING field:



Type in a regular expression. Can you match only the data entries?

(Try to write an expression that matches the entire entries - we'll see why later)

regular expressions - exercise 1 - solution

The screenshot shows a regular expression testing interface. At the top, there is a "REGULAR EXPRESSION" input field containing the pattern `r" ^[a-zA-Z][a-zA-Z][a-zA-Z]`. To the right of this is a green button that says "5 matches, 72 steps (~1ms)". Below the input field is a "TEST STRING" input area containing the following text:

```
# Measurements started
Sep 9, 9:05, T=22deg
SEP 9, 10:15, T=25deg
# Taking a coffee break
Sep 9, 11:15, T=-10deg
# Weekend
Sept 12, 09:00AM, T=32deg
Oct12 13:00, T=32degr
```

On the right side of the interface, there is a "SWITCH TO UNIT TESTS" button.

Conclusion: We cannot really solve the exercise with the ingredients we've seen so far...

More expression power: Ingredients 2

Quantifiers: How often should the preceding character or group be repeated?

*	zero or more times
+	one or more times
?	zero or one time
{n}	n times
{n,}	at least n times
{,n}	at most n times
{n,m}	between n and m times

Regular expressions - Examples

<u>pattern</u>	<u>examples of matching strings</u>
[0-9] +	'01' '10035'
[0-9] *	'01' '10035' "
.?el[lb]ow	'yellow' 'mellow' 'elbow'

Regular expressions - Exercise 2

Using regex101.com, can we match all our data file names now?

Regular expressions - Exercise 2 - solution

Success!

```
^ [a-zA-Z]+ ? [0-9]+ [, ]* [0-9:] + [^,]+, ?T=[-0-9]+degr?
```

REGULAR EXPRESSION

5 matches, 99 steps (~0ms)

```
: r" ^ [a-zA-Z]+ ? [0-9]+ [, ]* [0-9:] + [^,]+, ?T=[-0-9]+degr?" gm
```

TEST STRING

SWITCH TO UNIT TESTS ▾

```
# Measurements started
Sep 9, 9:05, T=22deg
SEP 9, 10:15, T=25deg
# Taking a coffee break
Sep 9, 11:15, T=-10deg
# Weekend
Sept 12, 09:00AM, T=32deg
Oct12 13:00, T=32degr
```

but... we wanted only the entries before September 10 that had positive temperature. Can we extract this information?

A few details: Ingredients 3

(expr)	Group an expression
(expr1 expr2)	Match either expr1 or expr2
\.	Matches a literal '.' (escaping)
\?	Matches a literal '?' (escaping)
^ and \$	Matches beginning and end of line
\A and \Z	Matches beginning and end of entire string

As we will see later, *groups* are accessible for extraction in Python.

Regular expressions - Exercise 3

See if you can add groups in your regular expression, so that we can extract both the temperature value and the month and day values.

Regular expressions - Exercise 3 - solution

```
^([a-zA-Z]+) ?([0-9]+)[, ]*[0-9:]+[^, ]+, ?T=(-[0-9]+)degr?
```

REGULAR EXPRESSION 5 matches, 132 steps (~1ms)

```
: r"^(?P<text>[a-zA-Z]+) ?(?P<time>[0-9]+)[, ]*(?P<temp>[0-9:]+)(?P<unit>degr?)"
```

TEST STRING SWITCH TO UNIT TESTS ▾

```
# Measurements started
Sep 9, 9:05, T=22deg
SEP 9, 10:15, T=25deg
# Taking a coffee break
Sep 9, 11:15, T=-10deg
# Weekend
Sept 12, 09:00AM, T=32deg
Oct 12 13:00, T=32degr
```

EXPLANATION

- asserts position at start of a line
- 1st Capturing Group `([a-zA-Z]+)`
 - Match a single character present in the set
 - + Quantifier — Matches between one and many occurrences
- ?T=(-[0-9]+)degr?

MATCH INFORMATION

Match 1	Full match	23-43 `Sep 9, 9:05, T=22deg`
Group 1.	23-26 `Sep`	
Group 2.	27-28 `9`	
Group 3.	38-40 `22`	

Hurray!

Regular expressions in Python

The re module

To use regular expressions in Python, you have to import the `re` module.

```
import re
```

Creating a pattern object

You can create a regular expression object using the `compile` function:

```
re_pattern.py
import re
pattern = re.compile('[a-zA-Z]+')
print(pattern)
```

```
output
<_sre.SRE_Pattern object at 0xb74db7a0>
```

match, search, and.findall

Most important methods in a regular expression object:

Check whether the pattern matches from
the start of the string. Returns match
object.

search() Search for substrings where pattern
matches. Returns match object.

.findall() Search for substrings where pattern
matches. Returns list of strings.

re_pattern2.py

```
pattern = re.compile('[0-  
9]+')  
print(pattern.match('13'))  
print(pattern.match('hello'))
```

output

```
<re.Match object; span=(0, 2), match='13'> # pattern  
matched  
None  
# pattern not  
matched
```

Regular expressions - Exercise 4

Start with the code on the next slide

1. Construct (this time in Python) a regular expression that matches the data entries
2. Inside the loop, apply the regular expression, and only print out the line if it matches the regular expression

Regular expressions - Exercise 4 - code

```
data = '''# Measurements started
Sep 9, 9:05, T=22deg
SEP 9, 10:15, T=25deg
# Taking a coffee break
Sep 9, 11:15, T=-10deg
# Weekend
Sept 12, 09:00AM, T=32deg
Oct12 13:00, T=32degr'''

data_lines = data.split('\n')

for data_line in data_lines:
    if ???:
        print(data_line)
```

Regular expressions - Exercise 4 - solution

```
# Import regular expression module
import re

data = '... ...' # data removed here to fit on slide

# Create regular expression
pattern = re.compile("^( [a-zA-Z]+ ) ?([0-9]+)[, ]*[0-9:]+[^\n,]+, ?T=(-[0-9]+)degr?")

for data_line in data.split('\n'):

    # Make attempt to match
    match = pattern.match(data_line)

    # If successful, print line
    if match:
        print(data_line)
```

But how do we access our groups?

The match object

The match object has methods that can extract information from the string that was matched:

Some of the most common ones:

group()	Return one or more groups of the match
groups()	Return a tuple with all groups
start()	Return start index of group
end()	Return end index of group

The group, start, and end methods all default to group index 0, which corresponds to the entire matching string.

The match object - example

```
import re
match_entries.py

data_lines = ['# Measurements started',
              'Sep 9, 9:05, T=22deg']

pattern = re.compile('^[a-zA-Z]+ ?([0-9]+)')

match = pattern.search(data_lines[1])
if match:
    print(match.groups()) # all specified groups
    print(match.group()) # Group 0 -> entire matching
str
    print(match.group(1)) # Group 1 -> first set of ()
```

output

```
('9')
Sep 9
9
```

Regular expressions - Exercise 5

Start with the previous exercise

1. Inside the loop, extract the temperature value, and only print entries with positive temperatures.
2. BONUS exercise - can you solve the full task as I originally phrased it: print only entries before September 10 for which the temperature is positive.
Hint: `datetime.strptime()`

Regular expressions - Exercise 5 - solution

match_entries2.py

```
...
# Same regular expression as before
for data_line in data.split('\n'):
    # Make attempt to match
    match = pattern.match(data_line)

    if match:
        # Extract temperature
        temperature = int(match.group(3))

        if temperature > 0:
            print(data_line)
```

output
Sep 9, 9:05, T=22deg
Sep 9, 10:15, T=25deg
Sept 12, 09:00AM, T=32deg
Oct12 13:00, T=32degr

Regular expressions - Exercise 5 - solution (BONUS)

```
match_entries3.py
from datetime import datetime
for data_line in data.split('\n'):
    # Make attempt to match
    match = pattern.match(data_line)
    if match:
        temperature = int(match.group(3)) # Extract
        temperature
        - Shorten to 3 characters
        month = match.group(1).lower()[:3] # Extract month
        day = int(match.group(2)) # Extract day
        # Create datetime object from month and day
        date = datetime.strptime("{} {}".format(month,
        day), "%b %d")
        if temperature > 0 and date <
        datetime.strptime("Sep 10", "%b %d"):
            print(data_line)
```

output
 Sep 9, 9:05, T=22deg
 Sep 9, 10:15, T=25deg

Regular expression details

Special characters (character classes)

For convenience, many often-used character classes have shortcuts

\d	digit character	[0-9]
\D	non-digit character	[^0-9]
\w	alphanumeric character	[a-zA-Z0-9_]
\W	non-alphanumeric character	[^a-zA-Z0-9_]
\s	white-space character	[\t\n\r\f\v]
\S	non-white-space character	[^\t\n\r\f\v]

The backslash mess

Both Python strings and Regular expressions use the \ character for characters with special meaning. This can be rather confusing.

For instance, the boundary character \b in regular expressions also means *backspace* in python strings. Therefore, the you have to escape the backslash

```
pattern = re.compile("\d+\b")    # \b doesn't work
```

This is annoying. Instead, use *raw strings* to write your regular expressions:

```
pattern = re.compile(r"\d+\b")    # Note the r""
```

Greedy vs non-greedy

Per default, repetitions are *greedy* - they will match as much as possible.

Sometimes, this is not what you want.

```
text = "Barack Obama"
pattern = re.compile("([A-Za-z]+).*( [A-Za-z]+)")
print(pattern.match(text).groups())    # what does this
give us?
```

The regular expression <.*> will match as much as possible instead of just the spaces between the words.

Solutions:

- Improve the regular expression (`\s*` or `[^A-Za-z]*`)
- Use non-greedy repetition operators

Non-greedy repetition

The non-greedy repetition operators match as little as possible

*? Match zero or more times (non-greedy)

+? Match one or more times (non-greedy)

?? Match zero or one time (non-greedy)

{n}? Match n times (non-greedy)

{n,}? Match at least n times (non-greedy)

{,n}? Match at most n times (non-greedy)

{n,m}? Match between n and m times (non-greedy)

Non-greedy repetition - example

```
re_non_greedy.py
```

```
text = "Barack Obama"
pattern1 = re.compile("[A-Za-z]+.*([A-Za-z]+)")      # greedy
pattern2 = re.compile("[A-Za-z]+.*?([A-Za-z]+)")     # non-greedy
print(pattern1.match(text).groups())
print(pattern2.match(text).groups())
```

```
output
```

```
{'Barack', 'a'}
{'Barack', 'Obama'}
```

Boundaries

^ and \$ match beginning and end of line

\A and \Z match beginning and end of entire string

\b Matches at boundary of word

\B Matches anywhere except at boundary of word

Note that none of the above actually corresponds to a character in the string we are matching against.

```
re_boundaries.py
```

```
pattern = re.compile(r"\d+\b")      # Note use of raw
string!
print(pattern.match("22 "))
print(pattern.match("22a"))
```

```
output
<_sre.SRE_Match object at 0x10ae8c440>
None
```

Compile flags

So far, we have used the default settings for
`re.compile()`

It is possible to alter the behavior by specifying *compile flags*

```
re.compile(pattern,flags)
```

<code>re.DOTALL</code>	make . also match newlines
<code>re.IGNORECASE</code>	Case insensitive matching
<code>re.MULTILINE</code>	Multiline matching
<code>re.VERBOSE</code>	Allow verbose regular expressions

Compile flags - example

re_compile_flags.py

```
import re
text = 'linux\npython'
pattern1 = re.compile('.*')
pattern2 = re.compile('.*', re.DOTALL)
print(pattern1.findall(text))
print(pattern2.findall(text))
```

```
output
['linux', '', 'python', '']
['linux\npython', '']
```

Sub and split

In addition to the `search`, `match` and `findall` methods,

the pattern object has two more important methods:

Substitution. Replace any substring in *string* that matches *pattern* with the string *replacement*.

```
pattern.sub(replacement, string)
```

Split. Split up string using matches as delimiters.

```
pattern.split(string)
```

Sub and split - example

SPLIT

```
re_split.py  
my_string = 'one|two@three$four'  
pattern = re.compile('[|@$]')  
print(pattern.split(my_string))
```

```
output  
['one', 'two', 'three', 'four']
```

SUB

```
re_sub.py  
my_string = 'one|two@three$four'  
pattern = re.compile('[|@$]')  
print(pattern.sub(" - ", my_string))
```

```
output  
one - two - three - four
```

Backreferencing

Inside a regular expression, you can refer to groups matched in a previous part of an expression, using \1 \2, etc

This also works in the *replacement* string of `sub`:

```
re_sub_backref.py
import re
my_string = 'Hello bla bla bla hello.'
pattern = re.compile('([Hh]ello)')
print(pattern.sub(r"\1ey", my_string))      # \1 refers to
first group
```

output

```
Hey bla bla bla hey.                      # Note, the case of h
is preserved
```

Regular expressions - Exercise 6

1. Within Python, open the british-english file (download from
https://wouterboomsma.github.io/ppds2023/data/british_english)
2. Iterate over the lines, and use a regular expression to print only those words that start and end with the same letter

Regular expressions - Exercise 6 - solutions

1. Within Python, open the /usr/share/dict/british-english file

```
dict_filename = "/usr/share/dict/british-english"  
# Open file  
dict_file = open(dict_filename)
```

2. Iterate over the lines, and use a regular expression to print only those words that start and end with the same letter

```
# Define regular expression
pattern = re.compile(r"^(.).*\1$")

for line in dict_file.readlines():
    if pattern.match(line): # No match => None =>
        interpreted as False
        print(line)
```

Speaker notes