3 Regular Expressions

Lab Objective: *Cleaning and formatting data are fundamental problems in data science. Regular expressions are an important tool for working with text carefully and efficiently, and are useful for both gathering and cleaning data. This lab introduces regular expression syntax and common practices, including an application to a data cleaning problem.*

A *regular expression* or *regex* is a string of characters that follows a certain syntax to specify a pattern. Strings that follow the pattern are said to *match* the expression (and vice versa). A single regular expression can match a large set of strings, such as the set all valid email addresses.

> ACHTUNG!
>
> There are some universal standards for regular expression syntax, but the exact syntax varies slightly depending on the program or language. However, the syntax presented in this lab (for Python) is sufficiently similar to any other regex system. Consider learning to use regular expressions in vim or your favorite text editor, keeping in mind that there are bound to be slight syntactic differences from what is presented here.

Regular Expression Syntax in Python

The `re` module implements regular expressions in Python. The function `re.compile()` takes in a regular expression string and returns a corresponding *pattern* object, which has methods for determining if and how other strings match the pattern. For example, the `search()` method returns `None` for a string that doesn't match, and a *match* object for a string that does (more on these later).

```
>>> import re
>>> pattern = re.compile("cat")     # Make a pattern for finding 'cat'.
>>> bool(pattern.search("cat"))     # 'cat' matches 'cat', of course.
True
>>> bool(pattern.search("catfish")) # 'catfish' also contains 'cat'.
True
>>> bool(pattern.search("hat"))     # 'hat' does not contain 'cat'.
False
```

Achtung!

The poorly named `match()` method for pattern objects only matches strings that satisfy the pattern **at the beginning** of the string. To answer the question "does any part of my target string match this regular expression?" always use the `search()` method.

```
>>> pattern = re.compile("cat")
>>> bool(pattern.match("catfish"))
True
>>> bool(pattern.match("fishcat"))
False
>>> bool(pattern.search("fishcat"))
True
```

Note

Most of the functions in the `re` module are shortcuts for compiling a pattern object and calling one of its methods. For example, the following lines of code are equivalent.

```
>>> bool(re.compile("cat").search("catfish"))
True
>>> bool(re.search("cat", "catfish"))
True
```

Using `re.compile()` is good practice because the resulting object is reusable, while each call to `re.search()` compiles a new (but redundant) pattern object.

Problem 1. Write a function that compiles and returns a regular expression pattern object with the pattern string `"python"`.

Construct positive and negative test cases to test your object. Having good test cases will be important later, so be thorough. Your verification process might start as follows.

```
>>> pattern = re.compile("cat")
>>> positive = ["cat", "catfish", "fish cat", "your cat ran away"]
>>> assert all(pattern.search(p) for p in positive)
```

Literal Characters and Metacharacters

The following string characters (separated by spaces) are *metacharacters* in Python's regular expressions, meaning they have special significance in a pattern string: `. ^ $ * + ? { } [ ] \ | ( )`.

To construct a regular expression that matches strings with one or more metacharacters in them requires two things. First, use *raw strings* instead of regular Python strings by prefacing the string with an `r`, such as `r"cat"`. The resulting string interprets backslashes as actual backslash characters, rather than the start of an escape sequence like `\n` or `\t`. Second, preface any metacharacters with a backslash to indicate a literal character. For example, the following code constructs a regular expression to match the string `"$3.99? Thanks."`.

```
>>> dollar = re.compile(r"\$3\.99\? Thanks\.")
>>> bool(dollar.search("$3.99? Thanks."))
True
>>> bool(dollar.search("$3\.99? Thanks."))
False
>>> bool(dollar.search("$3.99?"))   # Doesn't contain the entire pattern.
False
```

Without raw strings, every backslash in has to be written as a double backslash, which makes many regular expression patterns hard to read (`r"\\$3\\.99\\? Thanks\\."`). Readability counts.

Problem 2. Write a function that compiles and returns a regular expression pattern object that matches the string `"^{@}(?)[%]{.}(*)[_]{&}$"`.

The regular expressions of Problems 1 and 2 only match strings that are or include the exact pattern. The metacharacters allow regular expressions to have much more flexibility and control so that a single pattern could match a wide variety of strings, or a very specific set of strings.

To begin, the *line anchor* metacharacters `^` and `$` are used to match the **start** and the **end** of a line of text, respectively. This shrinks the matching set, even when using the `search()` method instead of the `match()` method. For example, the only single-line string that the expression `^x$` matches is `'x'`, whereas the expression `x` can match any string with an `x` in it.

```
>>> has_x, just_x = re.compile(r"x"), re.compile(r"^x$")
>>> for test in ["x", "xabc", "abcx"]:
...     print(test + ':', bool(has_x.search(test)), bool(just_x.search(test)))
...
x: True True
xabc: True False                 # Starts with 'x', but doesn't end with it.
abcx: True False                 # Ends with 'x', but doesn't start with it.
```

The *pipe* character `|` is like a logical `OR` in a regular expression: `A|B` matches `A` or `B`.

```
>>> rb, rbg = re.compile(r"^red$|^blue$"), re.compile(r"^red$|^blue$|^green$")
>>> for test in ["red", "blue", "green", "redblue"]:
...     print(test + ":", bool(rb.search(test)), bool(rbg.search(test)))
red: True True
blue: True True
green: False True
redblue: False False             # The line anchors prevent matching here.
```

The parentheses `()` create a *group* in a regular expression. Among other things, a group establishes an order of operations in an expression, much like how parentheses work in an arithmetic expression such as $3 \cdot (4 + 5)$.

```
>>> fish = re.compile(r"^(one|two) fish$")
>>> for test in ["one fish", "two fish", "red fish", "one two fish"]:
...     print(test + ':', bool(fish.search(test)))
...
one fish: True
two fish: True
red fish: False
one two fish: False
```

Problem 3. Write a function that compiles and returns a regular expression pattern object that matches the following strings (and no other strings, even with `re.search()`).

```
"Book store"      "Mattress store"     "Grocery store"
"Book supplier"   "Mattress supplier"  "Grocery supplier"
```

Character Classes

The hard bracket metacharacters `[` and `]` are used to create *character classes*, a part of a regular expression that can match a variety of characters. For example, the pattern `[abc]` matches any of the characters `a`, `b`, or `c`. This is different than a group delimited by parentheses: a group can match multiple characters, while a character class matches only one character. For instance, `[abc]` does not match `ab` or `abc`, and `(abc)` matches `abc` but not `ab` or even `a`.

Within character classes, there are two additional metacharacters. When `^` appears **as the first character** in a character class, right after the opening bracket `[`, the character class matches anything **not** specified instead. In other words, `^` is the set complement operation on the character class. Additionally, the dash `-` specifies a range of values. For instance, `[0-9]` matches any digit, and `[a-z]` matches any lowercase letter. Thus `[^0-9]` matches any character **except** for a digit, and `[^a-z]` matches any character **except** for a lowercase letters

```
>>> p1, p2 = re.compile(r"^[a-z][^0-7]$"), re.compile(r"^[^abcA-C][0-27-9]$")
>>> for test in ["d8", "aa", "E9", "EE", "d88"]:
...     print(test + ':', bool(p1.search(test)), bool(p2.search(test)))
...
d8: True True
aa: True False                    # a is not in [^abcA-C] or [0-27-9].
E9: False True                    # E is not in [a-z].
EE: False False                   # E is not in [a-z] or [0-27-9].
d88: False False                  # Too many characters.
```

Note that `[0-27-9]` acts like `[(0-2)|(7-9)]`.

There are also a variety of shortcuts that represent common character classes, listed in Table 3.1. Familiarity with these shortcuts makes some regular expressions significantly more readable.

Character	Description
\b	Matches the empty string, but only at the start or end of a word.
\d	Matches any decimal digit; equivalent to [0-9].
\D	Matches any non-digit character; equivalent to [^\d].
\s	Matches any whitespace character; equivalent to [\t\n\r\f\v].
\S	Matches any non-whitespace character; equivalent to [^\s].
\w	Matches any alphanumeric character; equivalent to [a-zA-Z0-9_].
\W	Matches any non-alphanumeric character; equivalent to [^\w].

Table 3.1: Character class shortcuts.

Any of the character class shortcuts can be used within other custom character classes. For example, [_A-Z\s] matches an underscore, capital letter, or whitespace character.

Finally, a period . matches **any** character except for a line break, and is therefore equivalent to [^\n] on UNIX machines and [^\r\n] on Windows machines. This is a very powerful metacharacter; be careful to only use it when part of the regular expression really should match **any** character.

```
# Match any three-character string with a digit in the middle.
>>> pattern = re.compile(r"^.\d.$")
>>> for test in ["a0b", "888", "n2%", "abc", "cat"]:
...     print(test + ':', bool(pattern.search(test)))
...
a0b: True
888: True
n2%: True
abc: False
cat: False

# Match two letters followed by a number and two non-newline characters.
>>> pattern = re.compile(r"^[a-zA-Z][a-zA-Z]\d..$")
>>> for test in ["tk421", "bb8!?", "JB007", "Boba?"]:
...     print(test + ':', bool(pattern.search(test)))
..
tk421: True
bb8!?: True
JB007: True
Boba?: False
```

Character	Description
.	Matches any character except a newline.
^	Matches the start of the string.
$	Matches the end of the string or just before the newline at the end of the string.
|	A|B creates an regular expression that will match either A or B.
[...]	Indicates a set of characters. A ^ as the first character indicates a complementing set.
(...)	Matches the regular expression inside the parentheses. The contents can be retrieved or matched later in the string.

Table 3.2: Standard regular expression metacharacters in Python.

Repetition

The remaining metacharacters are for matching a specified number of characters. This allows a single regular expression to match strings of varying lengths.

Character	Description
`*`	Matches 0 or more repetitions of the preceding regular expression.
`+`	Matches 1 or more repetitions of the preceding regular expression.
`?`	Matches 0 or 1 of the preceding regular expression.
`{m,n}`	Matches from m to n repetitions of the preceding regular expression.
`*?`, `+?`, `??`, `{m,n}?`	Non-greedy versions of the previous four special characters.

Table 3.3: Repetition metacharacters for regular expressions in Python.

```
# Match 0 or more 'a' characters, ending in a 'b'.
>>> pattern = re.compile(r"^a*b$")
>>> for test in ["b", "ab", "aaaaaaaaaab", "aba"]:
...     print(test + ':', bool(pattern.search(test)))
...
b: True                             # 0 'a' characters, then 1 'b'.
ab: True
aaaaaaaaaab: True                   # Several 'a' characters, then 1 'b'.
aba: False                          # 'b' must be the last character.

# Match an 'h' followed by at least one 'i' or 'a' characters.
>>> pattern = re.compile(r"^h[ia]+$")
>>> for test in ["ha", "hii", "hiaiaa", "h", "hah"]:
...     print(test + ':', bool(pattern.search(test)))
...
ha: True
hii: True
hiaiaa: True                        # [ia] matches 'i' or 'a'.
h: False                            # Need at least one 'i' or 'a'
hah: False                          # 'i' or 'a' must be the last character.

# Match an 'a' followed by 'b' followed by 0 or 1 'c' characters.
>>> pattern = re.compile(r"^abc?$")
>>> for test in ["ab", "abc", "abcc", "ac"]:
...     print(test + ':', bool(pattern.search(test)))
...
ab: True
abc: True
abcc: False                         # Only up to one 'c' is allowed.
ac: False                           # Missing the 'b'.
```

Each of the repetition operators acts on the expression immediately preceding it. This could be a single character, a group, or a character class. For instance, `(abc)+` matches `abc`, `abcabc`, `abcabcabc`, and so on, but not `aba` or `cba`. On the other hand, `[abc]*` matches any sequence of `a`, `b`, and `c`, including `abcabc` and `aabbcc`.

The curly braces `{}` specify a custom number of repetitions allowed. `{,n}` matches **up to** n instances, `{m,}` matches **at least** m instances, `{k}` matches **exactly** k instances, and `{m,n}` matches from m to n instances. Thus the `?` operator is equivalent to `{,1}` and `+` is equivalent to `{1,}`.

```
# Match exactly 3 'a' characters.
>>> pattern = re.compile(r"^a{3}$")
>>> for test in ["aa", "aaa", "aaaa", "aba"]:
...     print(test + ':', bool(pattern.search(test)))
...
aa: False                           # Too few.
aaa: True
aaaa: False                         # Too many.
aba: False
```

Achtung!

Line anchors are especially important when using repetition operators. Consider the following (bad) example and compare it to the previous example.

```
# Match exactly 3 'a' characters, hopefully.
>>> pattern = re.compile(r"a{3}")
>>> for test in ["aaa", "aaaa", "aaaaa", "aaaab"]:
...     print(test + ':', bool(pattern.search(test)))
...
aaa: True
aaaa: True                          # Should be too many!
aaaaa: True                         # Should be too many!
aaaab: True                         # Too many, and even with the 'b'?
```

The unexpected matches occur because `"aaa"` is at the beginning of each of the test strings. With the line anchors `^` and `$`, the search truly only matches the exact string `"aaa"`.

Problem 4. A *valid Python identifier* (a valid variable name) is any string staring with an alphabetic character or an underscore, followed by any (possibly empty) sequence of alphanumeric characters and underscores.

Define a function that compiles and returns a regular expression pattern object that matches any valid Python identifier.
(Hint: Use the `\w` character class shortcut to keep your regular expression clean.)

Check your regular expression against the following words. These test cases are a good start, but are not exhaustive.

Matches:	`"Mouse"`	`"compile"`	`"_123456789"`	`"__x__"`	`"while"`
Non-matches:	`"3rats"`	`"err*r"`	`"sq(x)"`	`"sleep()"`	`" x"`

Manipulating Text with Regular Expressions

So far we have been solely concerned with whether or not a regular expression and a string match, but the power of regular expressions comes with what can be done with a match. In addition to the `search()` method, regular expression pattern objects have the following useful methods.

Method	Description
`match()`	Match a regular expression pattern to the beginning of a string.
`fullmatch()`	Match a regular expression pattern to all of a string.
`search()`	Search a string for the presence of a pattern.
`sub()`	Substitute occurrences of a pattern found in a string.
`subn()`	Same as sub, but also return the number of substitutions made.
`split()`	Split a string by the occurrences of a pattern.
`findall()`	Find all occurrences of a pattern in a string.
`finditer()`	Return an iterator yielding a match object for each match.

Table 3.4: Methods of regular expression pattern objects.

```python
# Find words that start with 'cat'.
>>> expr = re.compile(r"\bcat\w*")  # \b is the shortcut for a word boundary.

>>> target = "Let's catch some catfish for the cat"
>>> bool(expr.search(target))       # Check to see if there is a match.
True

>>> expr.findall(target)            # Get all matching substrings.
['catch' 'catfish', 'cat']

>>> expr.sub("DOG", target)         # Substitute 'DOG' for the matches.
"Let's DOG some DOG for the DOG"

>>> expr.split(target)              # Split the target by the matches.
["Let's ", ' some ', ' for the ', '']
```

Some substitutions require remembering part of the text that the regular expression matches. Groups are useful here: each group in the regular expression can be represented in the substitution string by `\n`, where n is an integer (starting at 1) specifying which group to use.

```python
# Find words that start with 'cat', remembering what comes after the 'cat'.
>>> pig_latin = re.compile(r"\bcat(\w*)")
>>> target = "Let's catch some catfish for the cat"

>>> pig_latin.sub(r"at\1clay", target)  # \1 = (\w*) from the expression.
"Let's atchclay some atfishclay for the atclay"
```

Note

The repetition operators ?, +, *, and {m,n} are *greedy*, meaning that they match the largest string possible. On the other hand, the operators ??, +?, *?, and {m,n}? are *non-greedy*, meaning they match the smallest strings possible. This is very often the desired behavior for a regular expression.

```
>>> target = "<abc> <def> <ghi>"

# Match angle brackets and anything in between.
>>> greedy = re.compile(r"^<.*>$")  # Greedy *
>>> greedy.findall(target)
['<abc> <def> <ghi>']               # The entire string matched!

# Try again, using the non-greedy version.
>>> nongreedy = re.compile(r"<.*?>")# Non-greedy *?
>>> nongreedy.findall(target)
['<abc>', '<def>', '<ghi>']         # Each <> set is an individual match.
```

Finally, there are a few customizations that make searching larger texts manageable. Each of these *flags* can be used as keyword arguments to `re.compile()`.

Flag	Description
`re.DOTALL`	. matches any character at all, including the newline.
`re.IGNORECASE`	Perform case-insensitive matching.
`re.MULTILINE`	^ matches the beginning of lines (after a newline) as well as the string; $ matches the end of lines (before a newline) as well as the end of the string.

Table 3.5: Regular expression flags.

```
# Match any line with 3 consecutive 'a' characters somewhere.
>>> pattern = re.compile("^.*a{3}.*$", re.MULTILINE)    # Search each line.
>>> pattern.findall("""
This is aaan example.
This is not an example.
Actually, it's an example, but it doesn't match.
This example does maaatch though.""")
['This is aaan example.', 'This example does maaatch though.']

# Match anything instance of 'cat', ignoring case.
>>> catfinder = re.compile("cat", re.IGNORECASE)
>>> catfinder.findall("cat CAT cAt TAC ctacATT")
['cat', 'CAT', 'cAt', 'cAT']
```

Problem 5. A Python *block* is composed of several lines of code with the same indentation level. Blocks are delimited by key words and expressions, followed by a colon. Possible key words are `if`, `elif`, `else`, `for`, `while`, `try`, `except`, `finally`, `with`, `def`, and `class`. Some of these keywords require an expression of some sort to follow before the colon (`if`, `elif`, `for`, etc.), some require no expressions to follow before the colon (`else`, `finally`), and `except` may or may not have an expression following before the colon.

Write a function that accepts a string of Python code and uses regular expressions to place colons in the appropriate spots. You may assume that every colon is missing in the input string. See the following for an example.

```
"""
k, i, p = 999, 1, 0
while k > i
    i *= 2
    p += 1
    if k != 999
        print("k should not have changed")
    else
        pass
print(p)
"""

# The string given above should become this string.
"""
k, i, p = 999, 1, 0
while k > i:
    i *= 2
    p += 1
    if k != 999:
        print("k should not have changed")
    else:
        pass
print(p)
"""
```

Problem 6. The file `fake_contacts.txt` contains poorly formatted contact data for 5000 fictitious individuals. Each line of the file contains data for one person, including their name and possibly their birthday, email address, and/or phone number. The formatting of the data is not consistent, and much of it is missing. For example, not everyone has their birthday listed, and those who do may have it listed in the form `1/1/99`, `1/01/1999`, or some other format.

Use regular expressions to parse the data and format it uniformly, writing birthdays as `mm/dd/yyyy` and phone numbers as `(xxx)xxx-xxxx`. Return a dictionary where the key is the name of an individual and the value is another dictionary containing their information. Each of these inner dictionaries should have the keys `"birthday"`, `"email"`, and `"phone"`. In the case of missing data, map the key to None. The first two entries of the completed dictionary are given below.

```
{
    "John Doe": {
        "birthday": "01/01/1990",
        "email": "john_doe90@hopefullynotarealaddress.com",
        "phone": "(123)456-7890"
        },
    "Jane Smith": {
        "birthday": None,
        "email": None,
        "phone": "(222)111-3333"
        },
# ...
}
```

Additional Material

Regular Expressions in the Unix Shell

As we have seen thusfar, regular expressions are very useful when we want to match patterns. Regular expressions can be used when matching patterns in the Unix Shell. Though there are many Unix commands that take advantage of regular expressions, we will focus on `grep` and `awk`.

Regular Expressions and grep

Recall from Lab 1 that `grep` is used to match patterns in files or output. It turns out we can use regular expressions to define the pattern we wish to match.

In general, we use the following syntax:

```
$ grep 'regexp' filename
```

We can also use regular expressions when piping output to `grep`.

```
# List details of directories within current directory.
$ ls -l | grep ^d
```

Regular Expressions and awk

As in Lab 2, we will be using `awk` to format output. By incorporating regular expressions, `awk` becomes much more robust. Before GUI spreedsheet programs like Microsoft Excel, `awk` was commonly used to visualize and query data from a file.

Including `if` statements inside `awk` commands gives us the ability to perform actions on lines that match a given pattern. The following example prints the filenames of all files that are owned by `freddy`.

```
$ ls -l | awk ' {if ($3 ~ /freddy/) print $9} '
```

Because there is a lot going on in this command, we will break it down piece-by-piece. The output of `ls -l` is getting piped to `awk`. Then we have an `if` statement. The syntax here means if the condition inside the parenthesis holds, print field 9 (the field with the filename). The condition is where we use regular expressions. The ~ checks to see if the contents of field 3 (the field with the username) matches the regular expression found inside the forward slashes. To clarify, `freddy` is the regular expression in this example and the expression must be surrounded by forward slashes.

Consider a similar example. In this example, we will list the names of the directories inside the current directory. (This replicates the behavior of the Unix command `ls -d */`)

```
$ ls -l | awk ' {if ($1 ~ /^d/) print $9} '
```

Notice in this example, we printed the names of the directories, whereas in one of the example using `grep`, we printed all the details of the directories as well.

Achtung!

Some of the definitions for character classes we used earlier in this lab will not work in the Unix Shell. For example, `\w` and `\d` are not defined. Instead of `\w`, use `[[:alnum:]]`. Instead of `\d`, use `[[:digit:]]`. For a complete list of similar character classes, search the internet for *POSIX Character Classes* or *Bracket Character Classes.*

Problem 7. You have been given a list of transactions from a fictional start-up company. In the `transactions.txt` file, each line represents a transaction. Transactions are represented as follows:

```
# Notice the semicolons delimiting the fields. Also, notice that in between the last and first name, that is a comma, not a semicolon.
<ORDER_ID>;<YEAR><MONTH><DAY>;<LAST>,<FIRST>;<ITEM_ID>
```

Using this set of transactions, produce the following information using regular expressions and the given command:

- Using `grep`, print all transactions by either Nicholas Ross or Zoey Ross.
- Using `awk`, print a sorted list of the names of individuls that bought item 3298.
- Using `awk`, print a sorted list of items purchased between June 13 and June 15 of 2014 (inclusive).

These queries can be produced using one command each.

We encourage the interested reader to research more about how regular expressions can be used with `sed`.