Lab 3

Basic Regular Expressions

Lab Objective: *Learn the basics of using regular expressions to find text*

Regular expressions allow for quick searching and replacing of general patterns of text. While nearly all text editors have a feature that will find and replace exact strings of text, regular expressions are used to find text in a much more general way. For example, using a single regular expression, you can find every email address in a text file without having to sift through it by hand.

Terminology and Basics

A "regular expression" is basically just a string of characters that follow a certain syntax. Computer programs can then interpret these expressions as instructions to search for certain kinds of text. We will often call regular expressions "patterns", and we will say that certain patterns "match" certain strings. The general idea is that a regular expression represents a large set of strings (for example, all valid email addresses), and if a specific string is in that set, we say that the regular expression matches that string.

> ACHTUNG!
>
> Regular expression libraries have been implemented and are a part of the standard distribution of nearly every programming language, and many text editors have a find-and-replace mode that uses regular expressions. Unfortunately, the syntax for regular expressions may be slightly different in each implementation. There is no universal standard for all regular expressions across all platforms. However, the orginal syntax and a few variants are very widespread, so the basic regular expression techniques we learn in this lab should be virtually the same in almost every situation you will encounter them.

The simplest use of regular expressions is to match text literally. For example, the pattern `"cat"` matches the string `"cat"` but does not match the strings `"dog"` or `"bat"`.

Now that we have a general idea of what regular expressions are, we will see how to use them in Python.

Regular Expressions in Python

The python package `re` contains the functionality for using regular expressions. To use it, simply run the command `import re`.

The following Python code demonstrates what we said earlier about the regular expression `"cat"`:

```
>>> bool(re.match("cat", "cat"))
True
>>> bool(re.match("cat", "dog"))
False
>>> bool(re.match("cat", "bat"))
False
```

The main functions we will use are `re.match(pattern, string_to_test)` and `re.compile(pattern)`. You can think of `re.match` as returning a boolean value representing whether the given pattern matched the given string. The function `re.compile` returns a compiled object that represents a regular expression. You can then call the `match` function on this compiled object to get a boolean value. There is a similar function, `re.search`, which will match the regular expression anywhere inside a given string. We will see one example shortly where `re.search` is preferred in multiline matching.

The following code shows an example of how to use `re.compile`:

```
>>> pattern = re.compile("any regular expression")
>>> result = pattern.match("any string")
```

The above code is equivalent to the following:

```
>>> result = re.match("any regular expression", "any string")
```

Most programs use the compiled form (the first of the above two examples) for efficiency.

When constructing a regular expression, it is best to construct your pattern string using Python's syntax for raw strings by prefacing the string with the `'r'` character. This causes the constructed string to treat backslashes as actual backslash characters, rather than the start of an escape sequence.

For example:

```
>>> normal = "hello\nworld"
>>> raw = r"hello\nworld"
>>> print normal
hello
world
>>> print raw
hello\nworld
>>> type(normal), normal
(str, 'hello\nworld')
>>> type(raw), raw
(str, 'hello\\nworld')
```

Note that `raw` and `normal` are both python strings; one was just constructed differently. Also notice that when we constructed `raw`, it inserted an extra backslash before the existing backslash.

We use raw strings because the backslash character is a very important special character in regular expressions. If we wanted to use backslash characters as part of a normally-constructed Python string, we would need to either escape every single backslash by using two backslashes each time, or we could take the much easier and less confusing route of using Python's raw strings. To demonstrate this effect, suppose we wanted to know whether the regular expression `"\$3\.00"` matched the string `"$3.00"`. We could get our answer in either of the following ways:

```
>>> bool(re.match("\\$3\\.00", "$3.00"))
True
>>> bool(re.match(r"\$3\.00", "$3.00"))
True
```

(You will see why this pattern matches this string soon)

Remember, readability counts.

Literal Characters and Metacharacters

The following characters are used as metacharacters in regular expressions:

```
. ^ $ * + ? { } [ ] \ | ( )
```

These characters mean special things when used in regular expressions, making the vast power of regular expressions possible. We will get to using these characters later. For now, what do we do if want to match these characters literally? We simply escape these characters using the metacharacter `'\'`:

```
>>> pattern = re.compile(r"\$2\.95, please")
>>> bool(pattern.match("$2.95, please"))
True
>>> bool(pattern.match("$295, please"))
False
>>> bool(pattern.match("$2.95"))
False
```

Problem 1. Define the variable `pattern_string` using literal characters and escaped metacharacters in such a way that the following python program prints `True`:

```
import re
pattern_string = r""  # Edit this line
pattern = re.compile(pattern_string)
print bool(pattern.match("^{(!%.*_)}&"))
```

A little misleadingly, the `re.match` method isn't actually checking whether the given regular expression matches entire strings. Rather, it checks whether the regular expression matches *at* the *beginning* of the string, even if the string continues on afterward. For example:

```
>>> pattern = re.compile(r"x")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("xabc"))
True
>>> bool(pattern.match("abcx"))
False
```

You might not expect the pattern `'x'` to match the string `"xabc"`, but it does. This can cause confusion and headache, so we'll have to be a little more precise with the help of metacharacters.

The *line anchor* metacharacters, `'^'` and `'$'`, are used to match the start and the end of a line of text, respectively. Let's see them in action:

```
>>> pattern = re.compile(r"^x$")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("xabc"))
False
>>> bool(pattern.match("abcx"))
False
```

An added benefit of using `'^'` and `'$'` is that they allow you to search across multiple lines. For example, how would we match `"World"` in the string `"Hello\nWorld"`? Using `re.MULTILINE` in the `re.search` function will allow us to match at the beginning of each new line, instead of just the beginning of the string. Since we have seen two ways to match strings with regex expressions, the following shows two ways to implement multiline searching:

```
>>>bool(re.search("^W","Hello\nWorld"))
False
>>>bool(re.search("^W","Hello\nWorld", re.MULTILINE))
True
>>>pattern1 = re.compile("^W")
>>>pattern2 = re.compile("^W", re.MULTILINE)
>>>bool(pattern1.search("Hello\nWorld"))
False
>>>bool(pattern2.search("Hello\nWorld"))
True
```

For simplicity, the rest of the lab will focus on single line matching.

Let's move on to `'('`, `')'`, and `'|'`. The `'|'` character (the "pipe" character, usually found on the key below the backspace key) matches one of two or more regular expressions:

```
>>> pattern2 = re.compile(r"^red$|^blue$")
>>> pattern3 = re.compile(r"^red$|^blue$|^orange$")
>>> bool(pattern2.match("red")),     bool(pattern3.match("red"))
(True, True)
```

```
>>> bool(pattern2.match("blue")),    bool(pattern3.match("blue"))
(True, True)
>>> bool(pattern2.match("orange")),  bool(pattern3.match("orange"))
(False, True)
>>> bool(pattern2.match("redblue")), bool(pattern3.match("redblue"))
(False, False)
```

You can think of `'|'` as doing an "or" operation. How would we create a regular expression that matched both `"one fish"` and `"two fish"`? Although the regular expression `"one fish|two fish"` works, there is a better way, by using both the pipe character and parentheses:

```
>>> pattern = re.compile(r"^(one|two) fish$")
>>> bool(pattern.match("one fish"))
True
>>> bool(pattern.match("two fish"))
True
>>> bool(pattern.match("three fish"))
False
>>> bool(pattern.match("one two fish"))
False
```

As the above example demonstrates, parentheses are used to group sequences of characters together and change the order of precedence of the metacharacters, much like how parentheses work in an arithmetic expression such as `3*(4+5)`. In regular expressions, the `'|'` metacharacter has the lowest precedence out of all the metacharacters.

Parentheses actually have more uses, which we will learn later. For now, note that parentheses aren't matched literally:

```
>>> bool(re.match(r"r(hi)no(c(e)ro)s", "rhinoceros"))
True
```

Parentheses help give regular expressions higher precedence. For example, `"^one|two fish$"` gives precedence to the invisible string concatenation between `"two"` and `"fish"` while `"^(one|two) fish$"` gives precedence to the `'|'` metacharacter.

Problem 2. Define the variable `pattern_string` using the metacharacter `'|'` and parentheses in such a way that the following python program prints `True`:

```
import re
pattern_string = r""  # Edit this line
pattern = re.compile(pattern_string)
strings_to_match = [ "Book store", "Book supplier", "Mattress store", "↩
    Mattress supplier", "Grocery store", "Grocery supplier"]
print all(pattern.match(string) for string in strings_to_match)
```

Your regular expression should not match any other string, including strings such as `"Book store sale"`.

Character Classes

The metacharacters `'['` and `']'` are used to create *character classes*. Here they are in action:

```
>>> pattern = re.compile(r"[xy]")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("y"))
True
>>> bool(pattern.match("z"))
False
>>> bool(pattern.match("x: Why does this match? Were you paying attention?"))
True
```

In essence, a character class will match any one out of several characters.

Inside character classes, there are two additional metacharacters: `'-'` and `'^'`. Although we've already seen `'^'` as a metacharacter, it has a different meaning when used inside a character class. When `'^'` appears *as the first character* in a character class, the character class matches anything not specified instead. Think of `'^'` as performing a set complement operation on the character class. For example:

```
>>> pattern = re.compile(r"^[^ab]$")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("#"))
True
>>> bool(pattern.match("a"))
False
>>> bool(pattern.match("b"))
False
```

Note that the two `'^'` characters mean completely different things; the first `'^'` anchors us at the beginning of the line, while the second `'^'` performs a set complement operation on the character class `"[ab]"`.

The other character class metacharacter is `'-'`. This is used to specifiy a range of values. For example:

```
>>> pattern = re.compile(r"^[a-z][0-9][0-9]$")
>>> bool(pattern.match("a90"))
True
>>> bool(pattern.match("z73"))
True
>>> bool(pattern.match("A90"))
False
>>> bool(pattern.match("zs3"))
False
```

Multiple ranges or characters can be included in a single character class; in this case, the character class will match any character that fits either criterion:

```
>>> pattern = re.compile(r"^[abcA-C][0-27-9]$")
>>> bool(pattern.match("b8"))
True
>>> bool(pattern.match("B2"))
```

```
True
>>> bool(pattern.match("a9"))
True
>>> bool(pattern.match("a4"))
False
>>> bool(pattern.match("E1"))
False
```

Notice in the first line that `[abcA-C]` acts like `[a|b|c|(A-C)]` and `[0-27-9]` acts like `[(0-2)|(7-9)]`.

Finally, there are some built-in shorthands for certain character classes:

- `'\d'` (think "digit") matches any digit. It is equivalent to `"[0-9]"`.
- `'\w'` (think "word") matches any alphanumeric character or underscore. It is equivalent to `"[a-zA-Z0-9_]"`.
- `'\s'` (think "space") matches any whitespace character. It is equivalent to `"[ \t\n\r\f\v]"`.

The following character classes are the complements of those above:

- `'\D'` is equivalent to `"[^0-9]"` or `"[^D]"`
- `'\W'` is equivalent to `"[^a-zA-Z0-9_]"` or `"[^W]"`
- `'\S'` is equivalent to `"[^ \t\n\r\f\v]"` or `"[^S]"`

These character classes can be used in character classes; for example, `"[_A-Z\s]"` will match an underscore, any capital letter, or any whitespace character.

The `'.'` metacharacter, equivalent to `"[^\n]"` on UNIX and `"[^\r\n]"` on Windows, matches any character except for a line break. For example:

```
>>> pattern = re.compile(r"^.\d.$")
>>> bool(pattern.match("a0b"))
True
>>> bool(pattern.match("888"))
True
>>> bool(pattern.match("n2%"))
True
>>> bool(pattern.match("abc"))
False
>>> bool(pattern.match("m&m"))
False
>>> bool(pattern.match("cat"))
False
```

Problem 3. Define the variable `pattern_string` in such a way that the following python program prints `True`:

```
import re
pattern_string = r""  # Edit this line
pattern = re.compile(pattern_string)
```

```
strings_to_match = ["a", "b", "c", "x", "y", "z"]
uses_line_anchors = (pattern_string.startswith('^') and pattern_string.endswith('$'))
solution_is_clever = (len(pattern_string) == 8)
matches_list = all(pattern.match(string) for string in strings_to_match)

print uses_line_anchors and solution_is_clever and matches_list
```

Problem 4. A *valid python identifier* (aka a valid variable name) is defined as any string composed of an alphabetic character or underscore followed by any (possibly empty) sequence of alphanumeric characters and underscores.

Define the variable `identifier_pattern_string` that defines a regular expression that matches valid python identifiers that are exactly five characters long.

To help you test your pattern, the following program should print `True`. (This is necessary but not sufficient to show your regular expression is correct):

```
import re
identifier_pattern_string = r""  # Edit this line
identifier_pattern = re.compile(identifier_pattern_string)

valid = ["mouse", "HORSE", "_1234", "__x__", "while"]
not_valid = ["3rats", "err*r", "sq(x)", "too_long"]

print all(identifier_pattern.match(string) for string in valid) and not any(identifier_pattern.match(string) for string in not_valid)
```

Hint: Use the `'\w'` character class to keep your regular expression relatively short.

> **Note**
>
> As you might have noticed, using this definition, `"while"` is considered a valid python identifier, even though it really is a reserved word. In the following problems, we will make a few other simplifying assumptions about the python language.

Repetition

Suppose in the last problem we wanted the string to be 20 characters long. You wouldn't want to write `\w` 20 times. In fact, what if you wanted to match at most one instance of a character or a number with at least three digits? The metacharacters

`'*'`, `'+'`, `'{'`, and `'}'` are very useful for repetition.

The `'*'` metacharacter means "Match zero or more times (as many as possible)" when it follows another regular expression. For instance:

```
>>> pattern = re.compile(r"^a*b$")
>>> bool(pattern.match("b"))
True
>>> bool(pattern.match("ab"))
True
>>> bool(pattern.match("aab"))
True
>>> bool(pattern.match("aaab"))
True
>>> bool(pattern.match("abab"))
False
>>> bool(pattern.match("abc"))
False
```

The `'+'` metacharacter means "Match one or more times (as many as possible)" when it follows another regular expression. As an example:

```
>>> pattern = re.compile(r"^h[ia]+$")
>>> bool(pattern.match("ha"))
True
>>> bool(pattern.match("hii"))
True
>>> bool(pattern.match("hiaiaa"))
True
>>> bool(pattern.match("h"))
False
>>> bool(pattern.match("hah"))
False
```

It's important to understand why `"hiaiaa"` *is* a match here; matching multiple times means matching the preceeding *expression* multiple times, not matching the *results* of the preceeding expression multiple times. We haven't yet learned how to construct a regular expression with that behavior.

The `'?'` metacharacter means "Match one time (if possible) or do nothing (i.e. match zero times)" when it follows another regular expression:

```
>>> pattern = re.compile(r"^abc?$")
>>> bool(pattern.match("abc"))
True
>>> bool(pattern.match("ab"))
True
>>> bool(pattern.match("abd"))
False
>>> bool(pattern.match("ac"))
False
```

The curly brace metacharacters are used to specify a more precise amount of repetition:

```
>>> pattern = re.compile(r"^a{2,4}$")
>>> bool(pattern.match("a"))
False
```

```
>>> bool(pattern.match("aa"))
True
>>> bool(pattern.match("aaa"))
True
>>> bool(pattern.match("aaaa"))
True
>>> bool(pattern.match("aaaaa"))
False
```

If two arguments `x` and `y` are given to the curly braces (i.e., `{x, y}`), the preceeding regular expression must appear between `x` and `y` times, inclusive, in order for the overall expression to match.

Achtung!

In this last example, line anchors can save us from a lot of confusion. Note the differences between the following example and the example immediately above:

```
>>> pattern = re.compile(r"a{2,4}")
>>> bool(pattern.match("a"))
False
>>> bool(pattern.match("aa"))
True
>>> bool(pattern.match("aaa"))
True
>>> bool(pattern.match("aaaa"))
True
>>> bool(pattern.match("aaaaa"))
True
```

If only one argument `x` is given and is followed by a comma, the preceeding regular expression must match `x` or more times. If only one argument `x` is given without a comma, the preceeding regular expression must match *exactly* `x` times. For example:

```
>>> exactly_three = re.compile(r"^a{3}$")
>>> three_or_more = re.compile(r"^a{3,}$")
>>> def test_both_patterns(string):
...     return bool(exactly_three.match(string)), bool(three_or_more.match(string↩
    ))
>>> test_both_patterns("a")
(False, False)
>>> test_both_patterns("aa")
(False, False)
>>> test_both_patterns("aaa")
(True, True)
>>> test_both_patterns("aaaa")
(False, True)
>>> test_both_patterns("aaaaa")
(False, True)
```

You can also test `{,x}` which will match the preceeding regular expression up to `x` times.

Problem 5. Modify your definition of `identifier_pattern_string` from the previous problem to match valid python identifiers of any length.

Cleaning Dirty Data with Regular Expressions

A common consensus among data scientists is that the majority of your time will be spent cleaning data. Throughout the remainder of this volume, you will have multiple opportunities to practice cleaning data.

Often times, cleaning data is as simple as changing the format of your data or filling missing values. However, with text-based data, additional work is often necessary. Using regular expressions to clean text-based data is often a good option.

Problem 6. The provided file `contacts.txt` contains poorly formatted contact data for 5000 (fictitious) individuals. This dataset contains birthdays, email addresses, and phone numbers for the individuals.

You will notice that much of this data is missing. To make things more complicated, the format of the data isn't consistent. For example, some birthdays are in the format `1/1/99`, some in the format `01/01/1999`, and some in the format `1/1/1999`. The formatting for phone numbers is not consistent either. Some phone numbers are of the form `(123)456-7890` while others are of the form `123-456-7890`.

Using regular expressions, create a Python dictionary where the key is the name of the individual and the value is a dictionary of data. For example, the resulting dictionary should look something like this:

```
{"John Doe":{"bday":"1/1/1990",
             "email":"john_doe90@gmail.com",
             "phone":"(123)456-7890"}}
```

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`.