

# Project 1

Lyell Read, CS321H, F2020

Prompt 2: Research the different uses and implementations of regular expressions. Given examples of the notation in at least three different programming languages/scripts. Write code that uses regular expressions to validate user input for example email addresses, passwords or scientific notation.

## Regular Expression in Python

Regular Expression comparisons in Python are achieved using the `re` library, which comes preinstalled with `python3`. `re` supports the use of Perl-like regular expressions, which feature more easily readable (and more elegant) syntax than the more conventional POSIX or POSIX Extended regular expression syntaxes[1].

### Usage

To use regular expression comparisons in Python, the `re` module must first be imported into the program:

```
import re
```

Once this is taken care of, the regex can either be compiled, then matched against strings, or matched straight away, without explicitly compiling first. To compile first, the documentation in [2] suggests:

```
prog = re.compile(pattern)
result = prog.match(string)
```

If `re.fullmatch()` is used, however, there becomes no need to compile explicitly, and instead, [3] indicates that the following will match `string` against `pattern`:

```
result = re.fullmatch(pattern, string)
```

### Example Regex

The documentation ([2] or [3]) contain pages upon pages of detailed syntactical guides to developing patterns for the Python `re` library. The site [regexr.com](http://regexr.com) provides an informative view of the process of developing regular expressions for the PCRE regex type, which is for all intents and purposes what the `re` library uses.

The example regular expression that the program `password_acceptor.py` uses to match passwords in:

- At least 14 characters long
- Contains at least one from each {lowercase, uppercase, number, symbol}
  - Special characters are `@$!%*#?&`

Is as follows (adapted from [4]):

```
^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!%*#?&]{14,}$
```

This regular expression works as follows:

- The `^` matches start of string
- The `(?=.*[A-Z])` performs a forward search for A-Z in string
- The `(?=.*[a-z])` performs a forward search for a-z in string
- The `(?=.*\d)` performs a forward search for 0-9 in string
- The `(?=.*[@$!%*#?&])` performs a forward search for special in string
- The `[A-Za-z\d@$!%*#?&]{14,}` matches all valid, and asserts that characters must total 14 or more
- The `$` matches end of string.

## Regular Expression in Bash Script

Regular expression comparisons are supported by most modern shells (**zsh**, **bash**, **ksh**) [5] bash scripts. Regular expression comparisons in bash scripts take advantage of the `=~` operator, within the `[[ ]]` ‘keyword’ [6].

### Usage

To use regular expressions in a shell script, optionally define the regular expression string to be a local variable to the script, for neatness:

```
EMAIL_REGEX='^[A-Za-z_.-\d]+@[A-Za-z]+\.[a-z.]+$'
```

The single quotes are used to prevent expansion of (for example) `$` symbols and such. Next, make a comparison against the object to check against the regular expression:

```
if [[ $1 =~ $EMAIL_REGEX ]]
then
    #success
else
    #failure
```

In this example, the `EMAIL_REGEX` variable that was defined earlier is compared against the first command line argument to the program.

### Example Regex

In the above example, the goal was to accept any valid email address. This comes with some associated challenges, which the below regular expression will make obvious:

```
^[A-Za-z_.-\d]+@[A-Za-z]+\.[a-z.]+$
```

To explain how this regular expression works, let’s break each part down:

- The `^` matches start of string
- The `[A-Za-z_.-\d]+` matches any number of numbers and letters and `._-`
- The `@` matches a single `@`
- The `[A-Za-z]+` matches any number of upper or lowercase alphabetic characters
- The `\.` matches a period, must be escaped with `\`, as the `.` is a regular expression special character.
- The `[a-z.]+` matches the domain extension, i.e com, co.nz
- The `$` matches end of string

In this way, the regular expression supports all usernames (the prefix in the address, before the `@`), as well as uncommon domain extensions, such as `.space`, `.co.nz`, and others. That said, it is not a perfect regular expression.

## Regular Expressions in C

Regular expressions in the C language make use of the POSIX `<regex.h>` regular expressions library [7]. The support for this library varies across platforms, and is not part of ANSI C, however it is commonly found on most `*nix` systems [8].

### Usage

To use the POSIX C Regular Expressions library, first, it must be included (as well as some other helpful libraries for output and functions like `exit()`):

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
```

Then, optionally for neatness, the regular expression string must be defined, as well as other variables that will be used to handle parts of the compilation and execution process below:

```
regex_t regex;
int ret;
char message[100];
char input[100];

const char * regex_string = "(^[A-Z][[:alpha:]]+)([[:space:]] [A-Z]\\\.?)?\\"
    ([[:space:]] [A-Z][[:alpha:]]+)([[:space:]] (Jr\\\.|I{1,3}|Sr\\\.))?"
```

Next, it is necessary to compile our regular expression into something that can be evaluated. This is done as follows:

```
ret = regcomp(&regex, regex_string, REG_EXTENDED);
if (ret){
    puts("Error Compiling Regex");
    exit(1);
}
```

Once compiled into the variable `regex`, it can then be executed against a string to accept or deny the string:

```
ret = regexec(&regex, input, 0, NULL, 0);
if (!ret){
    puts("Name matched regex");
}
else if (ret==REG_NOMATCH){
    puts("Name did not match regex");
}
else{
    regerror(ret, &regex, message, sizeof(message));
    fprintf(stderr, "Regex match failed: %s\n", message);
    exit(1);
}
```

This process was inspired by [9].

## Example Regex

The regular expression in the above C example is designed to accept all full names, ranging from simple “John Smith” to more elaborate “Mary C. Smith II”.

The regular expression that is used is:

```
(^[A-Z][[:alpha:]]+)([[:space:]] [A-Z]\\\.?)?\\"
    ([[:space:]] [A-Z][[:alpha:]]+)([[:space:]] (Jr\\\.|I{1,3}|Sr\\\.))?
```

This regular expression is POSIX syntax regular expression, which uses a slightly different syntax from the PCRE/Perl regular expressions above. It is grouped using `()`. Notably, this regular expression does not work perfectly in the C program, because of `regex.h`’s inability to handle the string termination match character, `$`.

This regular expression can be explained with the following groups:

- The `(^[A-Z][[:alpha:]]+)` matches one or more capitalized, alphabetic words at the start of a string
- The `([[:space:]] [A-Z]\\\.?)?` matches zero or one of a space, followed by a capital letter, optionally followed by a period
- The `([[:space:]] [A-Z][[:alpha:]]+)` matches one or more last names with a preceding space, and a capital letter at the start of the name

- The `([:space:~](Jr~|.I{1,3}|Sr~|.))?` matches either one or zero of {Jr., Sr., I, II, III}

## Testing These Regular Expressions

Documentation can be found in `README.md` for running the basic tests using the `Makefile`, compiling using the `Makefile`, and running the programs individually.

## Sources Cited:

- [1] [https://en.wikipedia.org/wiki/Regular\\_expression#Perl\\_and\\_PCRE](https://en.wikipedia.org/wiki/Regular_expression#Perl_and_PCRE)
- [2] <https://docs.python.org/3/library/re.html#re.compile>
- [3] <https://docs.python.org/3/library/re.html#re.fullmatch>
- [4] <https://stackoverflow.com/a/21456918/8704864>
- [5] <https://serverfault.com/questions/52034/what-is-the-difference-between-double-and-single-square-brackets-in-bash>
- [6] <https://www.gnu.org/software/bash/manual/bash.html#Conditional-Constructs>
- [7] <https://pubs.opengroup.org/onlinepubs/7908799/xsh/regex.h.html>
- [8] <https://stackoverflow.com/a/1085120/8704864>
- [9] <https://stackoverflow.com/questions/1085083/regular-expressions-in-c-examples>
- [10] <https://www.regextester.com/99203>
- [11] <https://www.regular-expressions.info/posixbrackets.html>