Season 2

**Season’s Project:** A Note Book, Database

# Introduction

**Goal**: Develop a “notes” program which will, when provided a information file, and a search pattern of our choice, display the parts of the file containing that pattern.

The files we use may be filled with information we create such as of friends, family, coworkers and associates, or data we collect from others e.g., US States, historical data from the Internet.

We will begin by looking at an example program, data file, and patterns. We plan to proceed step-by-step, with a simple part of such a program, adding parts to create **your** functional notes program.

We’ll stop along the way reviewing, adding, and examining programming and python concepts and tools.

### File

A central programming concept in our note book project is the file.

File attributes – not all need apply every time

* Place(s) to store information so we can retrieve it or parts of it easily/quickly
* Produced and / or consumed by several sources
* Persistent - Stays around
* Large data.

Working example of a notes program: notes\_3e.py

Run program, using my defaults

Try different patterns

Students choose patterns.

1. Use us\_states.txt as an example of a database that can be created from the Internet.
2. Using Notepad, create my\_notes.txt with simple data

Example:

ray programmer Common St

prasha student Perkins

…

Run notes\_3e.py, use “my\_notes.txt” as file, “student” as pattern

Add a few more lines to my\_notes.txt

Try different patterns

# Iterations

To lay down a rough path for implementing our notes project program Following is an approximate list of program steps. Each step hopes to add a useful idea, concept, or feature to the existing code, moving us toward our goal. The order is a bit arbitrary but the plan is to provide a doable path that balances “big bang” features with doable coding steps.

## Reading computer file, Listing file lines

list\_file.py

Computer file – name like the traditional file - a storage place

Often, just listing the contents is useful enough.

Program plan:

* Prompt for file name
* Open file
* List lines in file

### Enhancements:

* Add line numbers to listing
* Repeat for new file

## Listing lines containing a Text pattern

simple\_notes.py from list\_file.py

### Minor extension, supporting changing patterns

simple\_notes\_loop.py.

## Searching variations

Often it is useful to vary on what we chose to match

* Case insensitive – treat upper case and lower case letters the same
* Accept matches only if they start at the beginning of the line
* Match only if the text is a compete word

### Case Insensitive – A capital idea

How to treat “M” and “m” the same?

Case sensitive (A != a) vs case insensitive (A == a)

simple\_notes\_case\_insensitive.py

## General Patterns – Regular expressions

### About Regular Expressions

Regular expression – special language for text matching to help Python or other programming languages with text matching. This is similar to the “arithmetic” language Python uses to express and evaluate arithmetic statements. Being reasonably understandable, regular expressions can be presented to the user as a method of specifying desired patterns. Like python, you don’t need to know everything about regular expressions to get a lot from them.

VERY SHORT description of regular expressions:

Letters(a-z, A-Z), digits(0-1), anything else that is not a “special” symbol, match themself

“a” matches “a”, “ray” matches ”ray” but not “Ray”

A handful of modifiers help to extent the pattern

(…) groups – “(abc)” matches “abc”

+ following *something* matches one or more of that something

“a+” matches “a”, “aa”, “aaa”

“(abc)+” matches “abc”, “abcabc”, “abcabcabc”

\* following *something* matches **zero** or more of that something

“a\*” matches “”, “a”, “aa”

“ba\*” matches “b”, “ba”, “baa”, “baaa”

There are numerous special or meta-characters which start with “\”

\s matches “white space” - space, tab, newline

\S matches NON “white space” any NON space, tab, newline such as “a”,”z”…

\S+ matches any contiguous list of non-space characters

\d matches a digit (0-9)

\D matches a NON-digit e.g. a,b,c, space, tab

\w matches a “word” character “a-z,A-Z,\_0

\w+ matches any string of “word” chracters

\W matches a Non-word character anything not a-z, A-Z, \_, 0-9

There are additional special characters

^ - matches the beginning of the search string/line

“^abc” matches “abcdef” but not “babcdef”

$ - matches the end of the search string/line

There is A LOT more, but the above can go a long way

Try some regular expression patterns, using notes\_3e.py.

### Programming python, using regular expressions

In our notes python program, we will use regular expressions to match strings by importing the “re” module, via the “import re” statement. To find strings we use statements of the form “re.search(*pattern*, *string*)” or “re.find(*pattern*, *string*)”. Our program will support regular expressions in user patterns. For example, to only match “M” at the beginning of a line, the user can enter **^M** (upper carrot followed by M).

re.search(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the **first location where the regular expression *pattern* produces a match**, and return a corresponding [match object](mk:@MSITStore:C:\Users\raysm\AppData\Local\Programs\Python\Python310\Doc\Python3102.chm::/library/re.html#match-objects). Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

re.match(*pattern*, *string*, *flags=0*)

If **zero or more characters at the beginning of *string* match** the regular expression *pattern*, return a corresponding [match object](mk:@MSITStore:C:\Users\raysm\AppData\Local\Programs\Python\Python310\Doc\Python3102.chm::/library/re.html#match-objects). Return None if the string does not match the pattern; note that this is different from a zero-length match.

re.I

re.IGNORECASE

Perform case-insensitive matching; expressions like [A-Z] will also match lowercase letters. Full Unicode matching (such as Ü matching ü) also works unless the [re.ASCII](mk:@MSITStore:C:\Users\raysm\AppData\Local\Programs\Python\Python310\Doc\Python3102.chm::/library/re.html#re.ASCII) flag is used to disable non-ASCII matches. The current locale does not change the effect of this flag unless the [re.LOCALE](mk:@MSITStore:C:\Users\raysm\AppData\Local\Programs\Python\Python310\Doc\Python3102.chm::/library/re.html#re.LOCALE) flag is also used. Corresponds to the inline flag (?i).

Example: rex\_patterns.py – experiment using regular expression patterns

Simple text patterns such as Mass or Main or California

Simple regular expressions: Ma\w+ or ^Ma\w+

## Program Control – command line args

Sometimes it is convenient to make program selections before the program is running.

One popular method is to use command line arguments. Often programs are written to support command line arguments. In our notes program, these will be optionally the file name and the search pattern. When these values are not present in the calling command line, the user will be prompted for one or both.

### file name on command line

list\_file\_cmd.py

### file and pattern specification on command line

simple\_notes\_cmd.py

Program loops only when there is no pattern specified on the command line.

### Named options, order independent

While quite useful, the standard command line arguments are somewhat limiting. One has to remember what position they are. To use subsequent options the first option must be present. A simple addition is to use flagged or named options. By preceding the value with a flag e.g. “-f” to indicate the option such as “-f” for “file name” provides a lot of flexibility. Options can be in any order. Options do not require previous options to be present. Our example is simplified by supporting only flagged options where the general case would allow a mixture of flagged and un-flagged options.

simple\_notes\_cmd\_flags.py