Regular Expressions

- Background
- Sets of strings
- Stating a regular expression (simple)
- Python **`re`** module (simple)
- A bit of theory

String patterns

- We all use searches where we provide strings or substrings to some module or mechanism
  - Google search terms
  - Filename completion
  - Command-line wildcards
  - Browser URL completion
  - Python string routines `find()`, `index()`, etc.
- Quite often these searches are simply expressed as a particular pattern
  - An individual word
  - Several words where some are strictly required while some are not
  - The start or end of particular words -- or perhaps just the string appearing within a larger string
- This works well if strings follow the format we expect…

String patterns

- Sometimes, however, we want to express a more complex pattern
  - The set of all files ending with either `.c` or `.h`
  - The set of all files starting with `in` or `out`.
  - The set of all strings in which `FREQ` appears as a string (but not `FREQUENCY` or `INFREQUENT`, but `fReQ` is fine)
  - The set of all strings containing dates in `MM/DD/YYYY` format.
- Such a variety of patterns used to require language-specific operations
  - SNOBOL
  - Pascal
- More troubling was that most non-trivial patterns required several lines of code to express (i.e., a series of "if-then-else" statements)
  - This is a problem as the resulting code can obscure the patterns for which we are searching
  - Even worse, changing the pattern is tedious and error-prone as it means changing the structure of already written code.

C code to check for DD/MM/YYYY format

```
int is_date_format(char *check) {

    if (!isdigit(check[0]) || !isdigit(check[1])) {
        return 0;
    }

    if (!isdigit(check[3]) || !isdigit(check[4])) {
        return 0;
    }

    for (i = 6; i < 10; i++) {
        if (!isdigit(check[i])) {
            return 0;
        }
    }

    if (check[2] != '/' || check[5] != '/') {
        return 0;
    }

    /* Still haven't even figured out if the DD makes sense, let alone
     * the MM!!!!
     */

    return 1;
}
```

Regular expressions

- Needed: a language-independent approach to expressing such patterns
- Solution: a **regular expression**
  - Sometimes called a **regex** or **regexp**
- They are written in a formal language and have the property that we can build very fast recognizers for them
- Part of a hierarchy of languages
  - Type 0: unrestricted grammars
  - Type 1: context-sensitive grammars
  - **Type 2: context-free grammars**
  - **Type 3: regular grammars**
- Type 2 and 3 grammars are used in Computer Science
  - Type 2 is used in parsers for computer languages (i.e., compilers)
  - Type 3 is used in regular expressions and lexical analyzers for compilers

grep

- We already can use regular expressions in Unix at the command line
- The `grep` utility accepts two sets of arguments
  - `grep`: **global regular expression print**
  - argument 1: A regular expression
  - argument 2: A set of files through which `grep` will try to find strings matching the regex
- The syntax for a regex is `grep` is somewhat similar to what we will use in Python
  - `grep` is a Very Old Tool (i.e., from 1973)
  - superseded somewhat in Modern Times by `fgrep` (fixed-string grep)
  - a variety of extensions, optimizations, etc. exist
- Example: search for variants on `apple`

grep

contents of `fruitstuff.txt`

```
apple
apples
Apple Pie
APPLE SUX!
apple-
apple-fruit
"Apple is the greatest!"
My best friend is an apple.
pineapple
Crabapple
fruit-apple
```

```
unix$ grep -i ^apple fruitstuff.txt
apple
apples
Apple Pie
APPLE SUX!
apple-
apple-fruit
```

```
unix$ grep apple fruitstuff.txt
apple
apples
apple-
apple-fruit
My best friend is an apple.
pineapple
Crabapple
fruit-apple
```

```
unix$ grep ^a.ple fruitstuff.txt
apple
apples
apple-
apple-fruit
```

```
unix$ grep -w apple fruitstuff.txt
apple
apple-
apple-fruit
My best friend is an apple.
fruit-apple
```

```
unix$ grep apple$ fruitstuff.txt
apple
pineapple
Crabapple
fruit-apple
```

More general regular expressions

- Our `grep` examples were relatively simple
- Sometimes we want to denote more complex sets of strings
  - strings where the beginning and end match a pattern, while everything in-between can vary
  - all possible spellings of a particular name
  - match non-printable characters
  - catch possible misspellings of a particular word
  - match Unicode code points
- And we may want even more:
  - when matching patterns to strings, extract the actual match itself
  - look for strings where the matched pattern repeats exactly later in the same string
  - extract multiple matches from one string

Metasymbols

- Fully-fledged regexes initially look intimidating because of the metasymbols
- However, all that is required to understand them is patience
- Regexes never loop…
- … nor are they ever recursive
- Understanding them means reading from left-to-right!
- However, first some metasymbols

symbol/example	meaning
`.`	match any char except `\n`
`a*`	zero or more reps of `'a'`
`a+`	one or more reps of `'a'`
`a?`	zero or one rep of `'a'`
`a{5}`	exactly 5 reps of `'a'`
`a{3,7}`	3 to 7 reps of `'a'`
`[abc]`	any one character in the set `{'a','b','c'}`
`[^abc]`	any one character not in the set of `{'a','b','c'}`
`a|b`	match `'a'` or `'b'`
`(...)`	group a component of symbols in the regex
`\`	escape any metasymbol (caution!)

Special pattern elements

symbol	meaning
`\d`	Any decimal digit character
`\w`	Any alphanumeric character
`\s`	Any whitespace character (`\t \n \r \f \v`)
`\b`	Empty string at a word boundary
`^`	match 0 characters at the start of the string
`$`	match 0 characters at the end of the string
`\D`	match any non-digit character (opposite of `\d`)
`\W`	match any non-alphanumeric character (opposite of `\w`)
`\S`	match any non-whitespace character
`\B`	empty string (i.e., 0 characters) not at a word boundary
`\number`	matches text of group `number`

Python regular expressions

- The `re` module
  - Introduced into Python in version 1.5
  - (Don't use the **`regex`** module which is an older release of a regular-expression library)
  - Used to be slower than regex, but is now as fast if not faster
  - Supports **`named groups`**
  - Supports **`non-greedy matches`** (we'll cover this later)
- Note:
  - Regular expression syntax is generally the same from language to language and library to library (e.g., Python, Perl, Ruby)
  - However, sometimes there are differences in the way some features are expressed (e.g., groups, escaped characters)
  - Whenever you move to different implementations, always have the library reference nearby.

Simple example

```
>>> text1 = 'Hello spam...World'
>>> text2 = 'Hello spam...other'


>>> import re
>>> matchobj = re.match('Hello.*World', text2)
>>> print (matchobj)
None


>>> if re.match('Hello.*World', text2):
...     print ("It's the end of the World")
... else:
...     print ("The end of the world is nowhere in sight")
...
The end of the world is nowhere in sight
>>
```

Previous example

- The regular-expression match was applied to string `text2`
  - Regex specified a string with `"Hello"` followed by 0 or many characters followed by `"World"`
  - The match did not succeed, therefore the value `None` was returned
  - In Python, None may be used as part of a conditional expression (i.e., has similar meaning to `"False"`.
- Even though the name of the RE method was `match()`, we did not use any syntax to extract out some result of the match
  - Which is just as well as there was no match.
  - However, if we wanted to extract out some result, we must use parentheses.
- Let's look at the example again, but this time include the other string in our use of match
  - Note that in the following example the `"import re"` is left out (i.e., we assume it was executed earlier in the session)

Simple example

```
>>> text1 = 'Hello spam...World'
>>> text2 = 'Hello spam...other'

>>> matchobj = re.match('Hello(.*)World', text1)
>>> print (matchobj)
<_sre.SRE_Match object at 0x10043b8a0>

>>> hello_list = [text1, text2]
>>> for t in hello_list:
...     matchobj = re.match('Hello(.*)World', t)
...     if matchobj:
...             print (t, " --> match --> ", matchobj.group(1))
...     else:
...             print (t, " --> no matches")
...
Hello spam...World  --> match -->   spam...
Hello spam...other  --> no matches
```

Previous example

- The match did succeed when applied to **`text1`**
  - The result is a match object
  - This has an interface which is used to extract matched substrings
  - In this case, we extracted the substring matching the pattern in the parentheses
- The parameter passed to **`group`** corresponds to the order of left parenthesis
  - A regular expression can have several such groups given the use of parentheses
  - Groups can even be nested (i.e., nested parentheses)…
  - … but **they can never overlap.**
  - Programmers make extensive use of groups in regular expressions
  - It helps make code more robust and less dependent on an exact format.

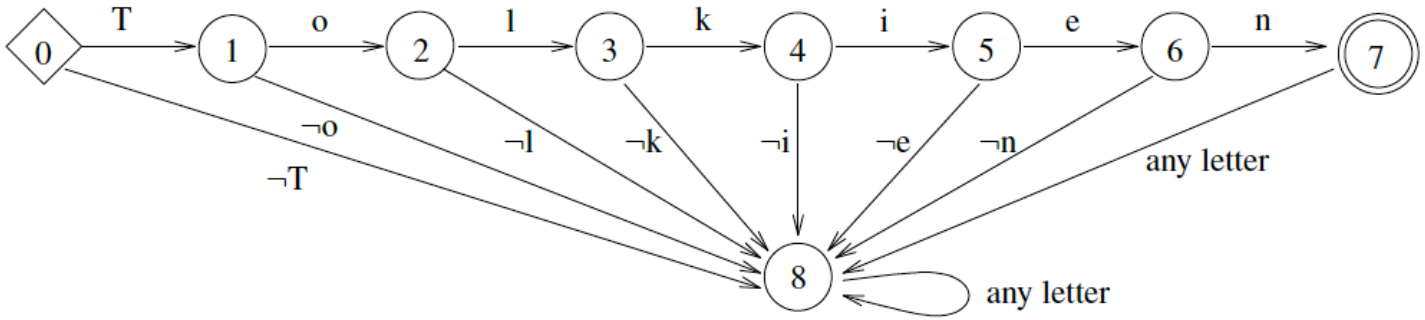
Speed concerns

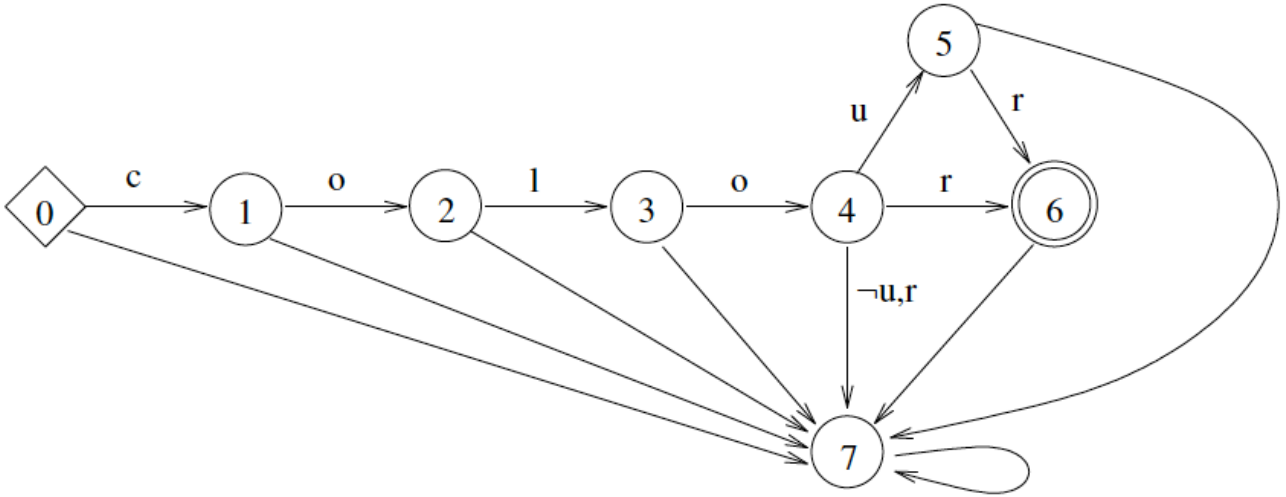
- So far we have specified the regular expression for every use of an **`re`** operation
- For occasional regex matching this is fine
- However, each time the match is performed the Python interpreter must re-interpret the regex
  - This means the regex must be re-parsed and the state machine re-constructed.
  - If we want to search many strings using the same regex, it makes sense to eliminate the overhead of repeating this work.
  - To eliminate the repeated work, we must **compile the pattern**

Speed concerns

- When using this style of regex matching, we work with a pattern object
  - Resulting code is much, much faster
  - Note, however, the compilation itself takes up some cycles.
- For now, just be aware there exist the two styles of invoking re operations
  - One directly specifying the regex in call to `match()`, `search()`, etc.
  - The other using a pattern object returned from `re.compile()` for which we call `match()`, `search()`, etc.).

Regex as a state machine





Compiled pattern

```
>>> pattobj = re.compile('Hello(.*)World')
>>> matchobj = pattobj.match(text1)
>>> print (matchobj)
<_sre.SRE_Match object at 0x10043b8a0>

>>> hello_list = [text1, text2]
>>> for t in hello_list:
...     matchobj = pattobj.match(t)
...     if matchobj:
...             print (t, " --> match --> ", matchobj.group(1))
...     else:
...             print (t, " --> no matches")
...
Hello spam...World  --> match -->   spam...
Hello spam...other  --> no matches
```

Lots in the `re` module

- Python's `re` module has methods for:
  - matching (i.e., finding a match that must start at the beginning of the string)
  - searching (i.e., finding a match that may occur anywhere in the string)
  - substituting
  - precompiling
  - splitting
  - iterating through matches
- Match objects also have several methods
  - We've already seen `group()`
  - There are also `groups()`, `groupdict()`
- Let us look at a few examples, this time with a few more metasymbols included

More complex pattern

```
>>> datetime1 = "20220211T110000"
>>> datetime2 = "20211225T000000Z"
>>> datetime3 = "11/06/2022"
>>> datetime4 = "22/4/14"

>>> matchobj = re.match("(\d{4})(\d{2})(\d{2})T.*", datetime1)
>>> if matchobj:
...     (year, month, day) = matchobj.groups()
...     print (year, month, day)
... else:
...     print ("Error")
...
2022 02 11
```

More complex pattern

```
>>> dates = ["20220211T110000", "20211225T000000Z", "11/06/2022",
             "22/4/14"]

>>> pattobj = re.compile( "(\d\d?)/(\d\d?)/(\d\d(\d\d)?)" )

>>> for d in dates:
...     matchobj = pattobj.match(d)
...     if matchobj:
...             (day, month, year, _) = matchobj.groups()
...             print ("%4d%02d%02d" % (int(year), int(month),
int(day)))
...     else:
...             print (d, "doesn't match")
...
20220211T110000 doesn't match
20211225T000000Z doesn't match
20220611
  140422
```

Another pattern

```
>>> line1 = ".LM +5"
>>> line2 = ".LM filled"
>>> line3 = ".LM 10x"
>>> line4 = ".LM 22"
>>> lines = [line1, line2, line3, line4]

>>> for line in lines:
...     matchobj = re.match("\.LM (\d+)\s*$", line)
...     if matchobj:
...             values = matchobj.groups()
...             print (line, ": matches with value ", values[0])
...     else:
...             print (line, ": DOESN'T match")

.LM +5 : DOESN'T match
.LM filled : DOESN'T match
.LM 10x : DOESN'T match
.LM 22 : matches with value  22
```

Notes from previous example

- Although grouping may be used to control the regular-expression match, not all results need to be extracted
  - Notice that sometimes part of the extracted matches is ignored
  - Always be aware the extracted matches are indexed by opening left parenthesis (i.e., not by your intent as a programmer to extract out particular parts of the match)
- There is often more than one way to phrase the same regular expression
  - Note that "\d\d" is the same as "\d{2}"
  - Which one is better? Depends perhaps on style of programmer, amount of change expected with code, etc. etc.

Variety

- Sometimes our needs vary when working with regexes
  - Sets of strings may be best expressed by alternative strings
  - Regexes may need to be carefully crafted sets of characters
  - Matches may sometimes be required on word boundaries
  - Sometimes all we want is the starting location of the match.
- Python string rules can sometimes interfere with regular expressions
  - The problem is with backslashes
  - Sometimes you must double-up on them (e.g., "\\")

Using search()

```
>>> pattern, string = "A.C.", "xxABCDxx"
>>> matchobj = re.search(pattern, string)
>>> if matchobj:
...     print (matchobj.start())
...
2
>>> pattobj = re.compile("A.*C.*")
>>> matchobj = pattobj.search("xxABCDxx")
>>> if matchobj:
...     print (matchobj.start())
...
2

>>> print (re.search(" *A.C[DE][D-F][^G-ZE]G\t+ ?", "..ABCDEFG\t..").start())
2
>>> print (re.search("A|XB|YC|ZD", "..AYCD..").start())
2
>>> print (re.search("\bABCD", "..ABCD").start())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'start'
>>> print (re.search(r"\bABCD", "..ABCD").start())
2
>>> print (re.search(r"ABCD\b", "..ABCD").start())
2
```

Problem Solving

- We have seen a variety of **metasymbols** (sometimes referred to as **metacharacters**)
  - Most of them match one or more characters
  - Some, however, are meant to catch a particular position (i.e., they catch zero characters!)
- The simplest positional symbols are `^` and `$`
  - `^`: match beginning of string
  - `$`: match end of string
  - Note that `re.match("<pattern>", string)` is **exactly** the same as `re.search("^<pattern>", string)` if `string` is not multiline
- Another positional symbol is `\b`
  - Matches a word boundary (i.e., zero characters)
  - That is, it matches the position in between characters (one of which is a word character, the other a non-word character)
  - Word characters: `[a-zA-Z0-9_]`
- **Problem 1: Match the word "Chris" in a string, but not "Christmas", "Christine", etc.**

Problem Solving

```
#!/usr/bin/env python3

import re

lines = ['''I said to Chris, "Hey, watch out!"''',
    '''I'll be home for Christmas!''',
    'Christine Faulkner',
    'Chris Flynn',
    'Evert, Chris']

for li in lines:
    if (re.search(r'\bChris\b', li)):
        print (li)
```

```
$ ./prob01.py
I said to Chris, "Hey, watch out!"
Chris Flynn
Evert, Chris
```

Problem Solving

- Words need not be textual
  - They can also be numerical
  - Key point is that non-word characters are neither numbers nor letters (nor the underscore)
- Sometimes we are interested in the shape of number sequences
  - Course numbers
  - Room numbers
  - Serial numbers, product codes, etc.
- **Problem 2: Extract the last four digits from a North American phone number**
  - May be of the form "250-472-5000"…
  - or "250 472 5000"…
  - or "472-5000"
  - or perhaps "250.472.5000" or "+1 250 472 5000"

Problem Solving

```
#!/usr/bin/env python3

import re

lines = ["250-472-5000", "472-5000", "250.472.5000", \
    "+1 250 472 5000", "011 49 9602 4241", "2504725000", \
    "mom's number", "12345 678 90"]

for l in lines:
    matchobj = re.search(r"(\b\d{3}\b[- \.])?\b\d{3}\b[- \.](\b\d{4}\b)",
l)
    if matchobj:

    matchobj = re.search(r"\b(?:\d{3})?\d{3}(\d{4})\b", l)
    if matchobj:
        print (l, "-->", matchobj.group(1))
```

```
$ ./prob02.py
250-472-5000 --> 5000
472-5000 --> 5000
250.472.5000 --> 5000
+1 250 472 5000 --> 5000
2504725000 --> 5000
```

Notice the "?:" used in the second search. It makes a set of parentheses "non-matching" but still useful to structure the regex. That is why the group number is still 1 even though the match we want is denoted by the second left-parenthesis

Problem Solving

- We can also use regexes to verify that the format provided as input matches what we expect
  - Example: Input string is in "DD/MM/YYYY" or "MM/DD/YYYY" format
  - Example: String provided is a URI (i.e., proper sets of characters)
- **Problem 3: Obtain a temperature (assumed to be Celsius) and return the number in Fahrenheit**
  - Number is to be an integer
  - There must be only one number in the string
  - No other characters (such as "C") should be at the end
  - Fahrenheit = (Celsius * 9 / 5) + 32

Input validation

```
#!/usr/bin/env python3

import re

celsius = input("Enter a temperature in Celsius: ")
celsius = celsius.rstrip("\n")

matchobj = re.search(r"^[0-9]+$", celsius)  # same as re.match("\d+",...)
if matchobj:
    celsius = int(celsius)
    fahrenheit = (celsius * 9 / 5) + 32
    print ("%d C is %d F" % (celsius, fahrenheit))
else:
    print ("Expecting a number, so I don't understand", celsius)
```

```
$ ./prob03.py
Enter a temperature in Celsius: 30
30 C is 86 F
```

Problem Solving

- However, we should do a bit more
  - The problem statement is perhaps a bit too restrictive.
  - Negative temperatures cannot be given as values.
  - Decimal temperatures also cannot be provided
- The regular expression should accept these
  - And the other code changed to suit (i.e., use `float()` instead of `int()`)

Input validation

```
#!/usr/bin/env python3

import re

celsius = input("Enter a temperature: ")
celsius.rstrip("\n")


matchobj = re.search(r"^([-+]?[0-9]+(\.[0-9]*)?)$", celsius)
if matchobj:
    celsius, _ = matchobj.groups()
    celsius    = float(celsius)

    fahrenheit = (celsius * 9 / 5) + 32
    print ("%.2f C is %.2f F" % (celsius, fahrenheit))
else:
    print ("Expecting a number, so I don't understand", celsius)
```

```
 ./prob03.py
Enter a temperature in Celsius: 12.2
12.20 C is 53.96 F
```

Problem Solving

- Our little script could be even more general
  - Rather than just convert from celsius to fahrenheit, it could convert the other direction
  - The starting value can be indicated by a "C" or "F" (or "c" or "f")
- **Problem 4: Obtain a temperature. If it is in celsius, return the number in fahrenheit; if in fahrenheit, return the number in celsius.**
  - Number can be an integer or a float, positive or negative
  - There must be only one number in the string
  - Character "C" or "F" implies what we are converting from and to.

Input validation plus more

```
#!/usr/bin/env python3

import re

input = input("Enter a temperature: ")
input.rstrip("\n")

matchobj = re.search(r"^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$", input, re.IGNORECASE)
if matchobj:
    input_num, _, type = matchobj.groups()
    input_num          = float(input_num)

    if type == "C" or type == "c":
        celsius    = input_num
        fahrenheit = (celsius * 9 / 5) + 32
    else:
        fahrenheit = input_num
        celsius = (fahrenheit - 32) * 5 / 9

    print ("%.2f C is %.2f F\n" % (celsius, fahrenheit))
else:
    print ('Expecting a number followed by "C" or "F",')
    print ('so I cannot interpret the meaning of', input)
```

Notice how we indicate that case is to be ignored. The `re` module contains a large number of these kinds of options.

Problem Solving

- Our solution to Problem 4 still has some flaws
  - Cannot enter a number less than one without a leading zero.
  - No leading spaces are permitted (i.e., we have general whitespace issues)
  - We are using `[0–9]` instead of `\d`
  - etc. etc.
- There are many ways to "skin" a regular expression
  - The lesson so far, however, is that coming up with a full regular expression for these kinds of matches can be an iterative process.
  - Must also be aware of how a language deals with metasymbols within strings (e.g., Perl and Ruby are a bit different than Python)