ECE 20875
Python for Data Science
Milind Kulkarni and Chris Brinton

regular expressions
basic text processing

- Python lets you do a lot of simple text processing with strings:

```python
s = "hello world"
s.count("l")  # returns 3
s.endswith("rld")  # returns True
"ell" in s  # returns True
s.find("ell")  # returns 1
s.replace("o", "0")  # returns "hell0 w0rld"
s.split(" ")  # returns ['hello', 'world']
```

See [https://docs.python.org/3/library/stdtypes.html#string-methods](https://docs.python.org/3/library/stdtypes.html#string-methods) for more

- But what if we want to do fancier processing? More complicated substitutions or searches?
regular expressions

- Powerful tool to find/replace/count/capture patterns in strings: regular expressions (regex)

- Can do very sophisticated text manipulation

```python
import re
s = "hello cool world see"
#find all double letters that are one character from the end of a word
p = re.compile(r"((.)(?=(?=\b)))")
#replace those double letters with their capital version
s1 = p.sub(lambda match : match.group(1).upper(), s)
print(s1) #prints ‘heLLo c00l world see’
```

- So what are regular expressions?
regular expressions (regex)

• A means for defining regular languages

• A language is a set (possibly infinite) of strings

• A string is a sequence of characters drawn from an alphabet

• A regular language is one class of languages: those defined by regular expressions (ECE 369 and 468 go into more details, including what other kinds of languages there are)

• Use: find whether a string (or a substring) matches a regex (more formally, whether a substring is in the language)
regular expressions

• A single string is a regular expression: “ece 20875” “data science”

  • Note: the empty string is also a valid regular expression

• All other regular expressions can be built up from three operations:

  1. Concatenating two regular expressions: “ece 20875 data science”

  2. A choice between two regular expressions: “(ece 20875)||(data science)”

  3. Repeating a regular expression 0 or more times “(ece)*”
building regular expressions

• A regular expression in Python is *compiled*

• This creates special code for matching a regular expression (ECE 369/468 discusses the machinery behind this)

```python
p = re.compile("ece (264|20875|368)")
```

• Can then look for the regular expression in other strings:

```python
p.match("ece 264") #returns a match object
p.match("hello ece 20875") #returns None
p.search("hello ece 368") #returns a match object
```
extra syntax for regex

• . #matches any character (except newline)
• ^abc #matches ‘abc’ only at the start of the string
• abc$ #matches ‘abc’ only at the end of the string
• a+ #matches one or more ‘a’s
• a? #matches 0 or one ‘a’
• [abc] #character class, matches ‘a’ or ‘b’ or ‘c’
• [^abc] #matches any character except ‘a’ or ‘b’ or ‘c’
• [a–z] #character class, matches any letter between ‘a’ and ‘z’
extra syntax for regex

• \s : matches whitespace
• \S : matches non-whitespace
• \d : matches a digit
• \D : matches a non-digit
• \w : matches a word character
• \W : matches a non-word character
lookahead characters

• \b : matches the beginning or end of a word (without consuming the character)

• \B : matches not the beginning or end of a word (without consuming the characters)

• ( ?=abc ) : matches “abc” (without consuming the characters)

• ( ?!abc ) : matches not “abc” (without consuming the characters)

• zero-width assertions: don’t cause the engine to advance through the string
• Can use parentheses to capture **groups**
  • Groups together characters (like in math): `(abc)*` means repeat `abc`, but `abc*` means repeat `c`

• Groups are **captured** by regular expressions
  • `match.group(k)` returns the contents of the kth group in the matched text
  • Group 0 is always the whole matched regex

• Groups can be nested — count based on number of left parentheses

• Groups can be named: `re.compile("(\?P<foo>abc)\")`

• Can refer to groups within a regular expression (or a substitution):
  • `\k` refers to the content of the kth group
  • `(\?P=foo)` refers to the content of the group named `foo`
substitution

• Regexes are great for string rewriting

• Generate the regex, with some groups:

\[p = re.compile(r'hello (\w*)') \] #match “hello …”

• Sub rewrites the matched string with the replacement. The replacement can refer to the same groups:

\[p.sub(r'goodbye \1', 'hello ece') \] #return ‘goodbye ece’
\[p.sub(r'goodbye \1', 'hello X') \] #return ‘goodbye X’
• Regexes are great for string rewriting

• Generate the regex, with some groups:

```python
p = re.compile(r'hello \w*) #match “hello …”
```

• Sub rewrites the matched string with the replacement. The replacement can refer to the same groups:

```python
p.sub(r'goodbye \1', 'hello ece') #return ‘goodbye ece’
p.sub(r'goodbye \1', 'hello X') #return ‘goodbye X’
```

Prefixing a string with ‘r’ makes it a raw string: Python does not use “normal” rules for ‘\’ in strings
file I/O

- File operation takes place in the following order:
  1. Open a file
  2. Read or write (perform operation)
  3. Close the file

- Opening a file: `open()` method
  - Returns a file object (handle) used to read or write
  - Specify the mode: most common are read ‘r’, write ‘w’, append ‘a’
  - `f = open("test.txt", 'w')` # write in text mode
file I/O

- Closing a file: close() method
  - Free up resources that were tied up with the file
  - Exception handling: Use try...finally block

  ```python
  try:
      f = open("test.txt", 'w')
      # perform file operations
  finally:
      f.close()
  ```
Writing files: open in write or append mode

- ‘w’ will overwrite existing file
- The `write("text")` method will write text to the file

```python
with open("test.txt", 'w') as f:
    f.write("my first file\n")
    f.write("This file\n\n")
    f.write("contains three lines\n")
```
file I/O

- Reading files: open in read mode
  - write 'w' will overwrite existing file

  ```python
  f = open("test.txt","r")
  f.read(4)   # read the first 4 characters
  f.read(4)   # read the next 4 characters
  f.read()    # read in the rest until the end
  f.close()
  
  f = open("test.txt","r")
  f.readline() # reads the first line (delimited by \n)
  f.readlines() # reads the remaining lines, returns as list
  f.close()
  ```