basic text processing

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

```
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"]
"XX".join(["hello", "world"])  #returns "helloXXworld"
```

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 and text extraction

```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'(\.)\2(?=.\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’
```

- Useful for data problems that require extracting data from a corpus
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:
  import re
  p = re.compile("ece (264|20875|368)"")

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

• Can then look for the regular expression in other strings:
  p.match("ece 264") #returns a match object
  p.match("hello ece 20875") #returns None
  p.search("hello ece 368") #returns a match object

• match checks only at the beginning of the string, while search looks throughout, and both only return the first occurrence
inspecting a match object

• We want to see what the match is, so we can set it to a variable:
  
x = p.search("hello ece 368")

• If we print x, we will see the match object (more on objects later)
  
  print(x)  # Returns <re.Match object; span=(6, 13)
             #   match=‘ece 368’>

• To see the actual match string, we use group():
  
x.group()  # Returns "ece 368"

• To see the index of the match, we use span():
  
x.span()  # Returns (6, 13)
extra syntax for regex

• . #wildcard, 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 0 or one ‘a’
• a* #matches zero or more ‘a’s
• a+ #matches one or more ‘a’s
• [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 digit
- \D #matches non-digit
- \w #matches any word character, which is alphanumeric and the underscore (equivalent to [a-zA-Z0-9_])
- \W #matches any non-word character

```python
s = "hello 12 hi 89. Howdy 34"
p = re.compile("\d+")
result = p.findall(s)
print(result)
#Output: ['12', '89', '34']
```
lookahead characters

- \b : matches the empty string at the beginning or end of a word
- \B : matches the empty string not at the beginning or end of a word
- \(?=abc\) : matches if “abc” is what comes next
- \(?!abc\) : matches if “abc” is not what comes next
- These are **zero-width assertions**: They don’t cause the engine to advance through the string, and they are not part of the resulting match

Other regex examples: [https://www.pythonsheets.com/notes/python-rexp.html](https://www.pythonsheets.com/notes/python-rexp.html)
groups

- 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
    - \texttt{match.group(k)} returns the contents of the \(k\)th group in the matched text
    - Group 0 is always the whole matched regex
    - \texttt{match.groups()} returns all subgroups in a list
groups

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

- Groups can be named:
  \[
  \text{re.compile(‘(\?P<foo>abc)’)}
  \]

- 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

```python
x = "dog = (?P<pet>\w+), cat = (?P=pet)"
y = "random_text  dog = sammy, cat = sammy"
z = re.compile(x).search(y)
print(z.group("pet"))
# prints sammy
```
substitution

- There is also a replacement command `sub()`
  - `p.sub(a,b)` rewrites `b` with any match to `p` replaced by `a`
- For example, we can generate the following regex, with groups:
  - `p = re.compile(r'hello (\w*)')` #match “hello …”
  - Note that prefixing a string with `r` makes it a raw string literal that tells Python not to process it (useful when trying to match characters like “\n”)
- We can write the following replacements, using the groups if we want:
  - `p.sub(r'goodbye \1', 'well hello ece')` #returns ‘well goodbye ece’
  - `p.sub(r'\1 goodbye \1', 'well hello X')` #return ‘well X goodbye X’