8.12 Introduction to Regular Expressions

- A **regular expression** describes a **search pattern** for **matching** characters in other strings
- Can help you extract data from unstructured text
- Can help you ensure that data is in the correct format before processing it

Validating Data

- Often use regular expressions to **validate the data**
  - A U.S. ZIP Code consists of five digits (such as 02215) or five digits followed by a hyphen and four more digits (such as 02215-4775)
  - A string last name contains only letters, spaces, apostrophes and hyphens
  - An e-mail address contains only the allowed characters in the allowed orde
  - A U.S. Social Security number contains three digits, a hyphen, two digits, a hyphen and four digits, and adheres to other rules about the specific numbers that can be used in each group of digits
- Rarely need to create your own regular expressions
- Repositories of existing regular expressions that you can copy and use
  - `https://regex101.com`
  - `http://www.regexlib.com`
  - `https://www.regular-expressions.info`

Other Uses of Regular Expressions

- Extract data from text (sometimes known as **scraping**)
  - e.g., locating all URLs in a web page
  - You might prefer tools like BeautifulSoup, XPath and lxml for this
- Clean data
  - Removing data that's not required, removing duplicate data, handling incomplete data, fixing typos, ensuring consistent data formats, dealing with outliers and more
- Transform data into other formats
  - Reformatting data that was collected as tab-separated or space-separated values into comma-separated values (CSV) for an application that requires data to be in CSV format

---

8.12.1 re Module and Function fullmatch

- **`fullmatch`** checks whether the **entire** string in its second argument matches the pattern in its first argument

In [26]:

```
import re
```

Matching Literal Characters

In [27]:

```
pattern = '02215'
```

In [28]:

```
'Match' if re.fullmatch(pattern, '02215') else 'No match'
```

Out[28]:

```
'Match'
```

In [29]:

```
'Match' if re.fullmatch(pattern, '51220') else 'No match'
```

Out[29]:

```
'No match'
```

- First argument is the regular expression pattern to match
  - Any string can be a regular expression
  - **literal characters** match **themselves** in the specified order
- Second argument is the string that should entirely match the pattern.
- If the second argument matches the pattern in the first argument, `fullmatch` returns an object containing the matching text, which evaluates to `True`
- For no match, `fullmatch` returns `None`, which evaluates to `False`

Metacharacters, Character Classes and Quantifiers

- Regular expressions typically contain various special symbols called **metacharacters**:

Regular expression metacharacters
`[] {} () \ * + ^ $ ? . |`

- **`\` metacharacter** begins each predefined **character class**
- Each matches a specific set of characters

In [30]:

```
'Valid' if re.fullmatch(r'\d{5}', '02215') else 'Invalid'
```

Out[30]:

```
'Valid'
```

In [31]:

```
'Valid' if re.fullmatch(r'\d{5}', '9876') else 'Invalid'
```

Out[31]:

```
'Invalid'
```

- In `\d{5}` , **`\d`** is a character class representing a digit (0–9)
- A character class is a **regular expression escape sequence** that matches **one** character
- To match more than one, follow the character class with a **quantifier**
- `{5}` repeats `\d` five times to match five consecutive digits

Other Predefined Character Classes

- To match any metacharacter as its **literal** value, precede it by a backslash (\ )
    - For example, `\\` matches a backslash (\ ) and `\$` matches a dollar sign ($)

Character class	Matches
`\d`	Any digit (0–9).
`\D`	Any character that is *not* a digit.
`\s`	Any whitespace character (such as spaces, tabs and newlines).
`\S`	Any character that is *not* a whitespace character.
`\w`	Any **word character** (also called an **alphanumeric character**)—that is, any uppercase or lowercase letter, any digit or an underscore
`\W`	Any character that is *not* a word character.

Custom Character Classes

- Square brackets, `[]` , define a **custom character class** that matches a **single** character
- `[aeiou]` matches a lowercase vowel
- `[A-Z]` matches an uppercase letter
- `[a-z]` matches a lowercase letter
- `[a-zA-Z]` matches any lowercase or uppercase letter

In [32]:

```
'Valid' if re.fullmatch('[A-Z][a-z]*', 'Wally') else 'Invalid'
```

Out[32]:

```
'Valid'
```

In [33]:

```
'Valid' if re.fullmatch('[A-Z][a-z]*', 'eva') else 'Invalid'
```

Out[33]:

```
'Invalid'
```

In [34]:

```
'Match' if re.fullmatch('[^a-z]', 'A') else 'No match'
```

Out[34]:

```
'Match'
```

In [35]:

```
'Match' if re.fullmatch('[^a-z]', 'a') else 'No match'
```

Out[35]:

```
'No match'
```

In [36]:

```
'Match' if re.fullmatch('[*+$]', '*') else 'No match'
```

Out[36]:

```
'Match'
```

In [37]:

```
'Match' if re.fullmatch('[*+$]', '!') else 'No match'
```

Out[37]:

```
'No match'
```

`*` vs. `+` Quantifier

- `+` matches **at least one occurrence** of a subexpression
- `*` and `+` are **greedy**—they match as many characters as possible

In [38]:

```
'Valid' if re.fullmatch('[A-Z][a-z]+', 'Wally') else 'Invalid'
```

Out[38]:

```
'Valid'
```

In [39]:

```
'Valid' if re.fullmatch('[A-Z][a-z]+', 'E') else 'Invalid'
```

Out[39]:

```
'Invalid'
```

Other Quantifiers

- **`?` quantifier** matches **zero or one occurrences** of a subexpression

In [40]:

```
'Match' if re.fullmatch('labell?ed', 'labelled') else 'No match'
```

Out[40]:

```
'Match'
```

In [41]:

```
'Match' if re.fullmatch('labell?ed', 'labeled') else 'No match'
```

Out[41]:

```
'Match'
```

In [42]:

```
'Match' if re.fullmatch('labell?ed', 'labellled') else 'No match'
```

Out[42]:

```
'No match'
```

- `labell?ed` matches `labelled` (the U.K. English spelling) and `labeled` (the U.S. English spelling), but not the misspelled word `labellled`
- `l?` indicates that there can be **zero or one more** `l` characters before the remaining literal `ed` characters

Other Quantifiers (cont.)

- Can match **at least *n* occurrences** of a subexpression with the **`{n,}` quantifier**

In [43]:

```
'Match' if re.fullmatch(r'\d{3,}', '123') else 'No match'
```

Out[43]:

```
'Match'
```

In [44]:

```
'Match' if re.fullmatch(r'\d{3,}', '1234567890') else 'No match'
```

Out[44]:

```
'Match'
```

In [45]:

```
'Match' if re.fullmatch(r'\d{3,}', '12') else 'No match'
```

Out[45]:

```
'No match'
```

Other Quantifiers (cont.)

- Can match **between *n* and *m* (inclusive) occurrences** of a subexpression with the **`{n,m}` quantifier**

In [46]:

```
'Match' if re.fullmatch(r'\d{3,6}', '123') else 'No match'
```

Out[46]:

```
'Match'
```

In [47]:

```
'Match' if re.fullmatch(r'\d{3,6}', '123456') else 'No match'
```

Out[47]:

```
'Match'
```

In [48]:

```
'Match' if re.fullmatch(r'\d{3,6}', '1234567') else 'No match'
```

Out[48]:

```
'No match'
```

In [49]:

```
'Match' if re.fullmatch(r'\d{3,6}', '12') else 'No match'
```

Out[49]:

```
'No match'
```

8.12.2 Replacing Substrings and Splitting Strings

- `sub` function replaces patterns in a string
- `split` function breaks a string into pieces, based on patterns

Function `sub` —Replacing Patterns

- **`sub` function** replaces **all** occurrences of a pattern with the replacement text you specify

In [109]:

```
import re
```

In [110]:

```
re.sub(r'\t', ', ', '1\t2\t3\t4')
```

Out[110]:

```
'1, 2, 3, 4'
```

- Three required arguments:
  - the *pattern to match* (the tab character `'\t'`)
  - the *replacement text* (`', '`) and
  - the *string to be searched* (`'1\t2\t3\t4'`)
- Keyword argument `count` can be used to specify the maximum number of replacements

In [111]:

```
re.sub(r'\t', ', ', '1\t2\t3\t4', count=2)
```

Out[111]:

```
'1, 2, 3\t4'
```

Function `split`

- **`split` function** *tokenizes* a string, using a regular expression to specify the *delimiter*
- Returns a list of strings

In [112]:

```
re.split(r',\s*', '1,  2,  3,4,    5,6,7,8')
```

Out[112]:

```
['1', '2', '3', '4', '5', '6', '7', '8']
```

- Keyword argument `maxsplit` specifies maximum number of splits

In [113]:

```
re.split(r',\s*', '1,  2,  3,4,    5,6,7,8', maxsplit=3)
```

Out[113]:

```
['1', '2', '3', '4,    5,6,7,8']
```

8.12.3 Other Search Functions; Accessing Matches

Function `search` —Finding the First Match Anywhere in a String

- `search` looks in a string for the *first* occurrence of a substring that matches a regular expression and returns a **match object** (of type `SRE_Match`) that contains the matching substring
- Match object's `group` method returns that substring

In [114]:

```
import re
```

In [115]:

```
result = re.search('Python', 'Python is fun')
```

In [116]:

```
result.group() if result else 'not found'
```

Out[116]:

```
'Python'
```

- `search` returns `None` if the string does *not* contain the pattern

In [117]:

```
result2 = re.search('fun!', 'Python is fun')
```

In [118]:

```
result2.group() if result2 else 'not found'
```

Out[118]:

```
'not found'
```

- Can search for a match only at the *beginning* of a string with function `match`

Ignoring Case with the Optional `flags` Keyword Argument

- Many `re` module functions receive an optional `flags` keyword argument
- Changes how regular expressions are matched

In [119]:

```
result3 = re.search('Sam', 'SAM WHITE', flags=re.IGNORECASE)
```

In [120]:

```
result3.group() if result3 else 'not found'
```

Out[120]:

```
'SAM'
```

Metacharacters That Restrict Matches to the Beginning or End of a String

- `^` **metacharacter** at the beginning of a regular expression (and not inside square brackets) is an anchor
- Indicaties that the expression matches only the *beginning* of a string

In [121]:

```
result = re.search('^Python', 'Python is fun')
```

In [122]:

```
result.group() if result else 'not found'
```

Out[122]:

```
'Python'
```

In [123]:

```
result = re.search('^fun', 'Python is fun')
```

In [124]:

```
result.group() if result else 'not found'
```

Out[124]:

```
'not found'
```

- `$` **metacharacter** at the end of a regular expression is an anchor indicating that the expression matches only the *end* of a string

In [125]:

```
result = re.search('Python$', 'Python is fun')
```

In [126]:

```
result.group() if result else 'not found'
```

Out[126]:

```
'not found'
```

In [127]:

```
result = re.search('fun$', 'Python is fun')
```

In [128]:

```
result.group() if result else 'not found'
```

Out[128]:

```
'fun'
```

Function `findall` and `finditer` —Finding All Matches in a String

- **`findall`** finds *every* matching substring in a string
- Returns a list of the matching substrings

In [129]:

```
contact = 'Wally White, Home: 555-555-1234, Work: 555-555-4321'
```

In [130]:

```
re.findall(r'\d{3}-\d{3}-\d{4}', contact)
```

Out[130]:

```
['555-555-1234', '555-555-4321']
```

- **`finditer`** works like `findall` , but returns a lazy *iterable* of match objects

In [131]:

```
for phone in re.finditer(r'\d{3}-\d{3}-\d{4}', contact):
    print(phone.group())
```

```
555-555-1234
555-555-4321
```

Capturing Substrings in a Match

- Use **parentheses metacharacters**— **`(`** and **`)`** —to capture substrings in a match

In [132]:

```
text = 'Charlie Cyan, e-mail: demo1@deitel.com'
```

In [133]:

```
pattern = r'([A-Z][a-z]+ [A-Z][a-z]+), e-mail: (\w+@\w+\.\w{3})'
```

In [134]:

```
result = re.search(pattern, text)
```

- The regular expression specifies two substrings to capture, each denoted by the metacharacters `(` and `)`
- `(` and `)` do *not* affect whether the `pattern` is found in the string `text`
- `match` function returns a match object *only* if the *entire* `pattern` is found in the string `text`
- `match` object's **`groups`** method returns a tuple of the captured substrings

In [135]:

```
result.groups()
```

Out[135]:

```
('Charlie Cyan', 'demo1@deitel.com')
```

- `match` object's `group` method returns the *entire* match as a single string

In [136]:

```
result.group()
```

Out[136]:

```
'Charlie Cyan, e-mail: demo1@deitel.com'
```

- Access each captured substring by passing an integer to the `group` method

In [137]:

```
result.group(1)
```

Out[137]:

```
'Charlie Cyan'
```

In [138]:

```
result.group(2)
```

Out[138]:

```
'demo1@deitel.com'
```

8.13 Intro to Data Science: Pandas, Regular Expressions and Data Munging

- Data does not always come in forms ready for analysis
- Data could be
    - wrong format
    - incorrect
    - missing
- Data scientists can spend as much as 75% of their time preparing data before they begin their studies
- Called **data munging** or **data wrangling**

8.13 Intro to Data Science: Pandas, Regular Expressions and Data Munging (cont.)

- Two of the most important steps in data munging are *data cleaning* and *transforming data* into optimal formats for database systems and analytics software
- Common data cleaning examples include:
    - deleting observations with missing values,
    - substituting reasonable values for missing values,
    - deleting observations with bad values,
    - substituting reasonable values for bad values,
    - tossing outliers (although sometimes you'll want to keep them),
    - duplicate elimination (although sometimes duplicates are valid),
    - dealing with inconsistent data,
    - and more.

8.13 Intro to Data Science: Pandas, Regular Expressions and Data Munging (cont.)

- Data cleaning is a difficult and messy process where you could easily make bad decisions that would negatively impact your results
- The actions data scientists take can vary per project, be based on the quality and nature of the data and be affected by evolving organization and professional standards
- Common data transformations include:
    - removing unnecessary data and *features* (we'll say more about features in the data science case studies),
    - combining related features,
    - sampling data to obtain a representative subset (we'll see in the data science case studies that *random sampling* is particularly effective for this and we'll say why),
    - standardizing data formats,
    - grouping data,
    - and more.

Cleaning Your Data

- Bad data values and missing values can significantly impact data analysis
- Some data scientists advise against any attempts to insert “reasonable values”
    - Instead, they advocate clearly marking missing data and leaving it up to the data analytics package to handle the issue

Cleaning Your Data (cont.)

- Consider a hospital that records patients’ temperatures (and probably other vital signs) four times per day
- Assume that the data consists of a name and four `float` values, such as

```
['Brown, Sue', 98.6, 98.4, 98.7, 0.0]
```

- Patient’s first three recorded temperatures are 99.7, 98.4 and 98.7
- Last temperature was missing and recorded as 0.0, perhaps because the sensor malfunctioned
- Average of the first three values is 98.57, which is close to normal
- If you calculate the average temperature *including* the missing value for which 0.0 was substituted, the average is only 73.93, clearly a questionable result
- Crucial to “get the data right.”
- One way to clean the data is to substitute a *reasonable* value for the missing temperature, such as the average of the patient’s other readings

Data Validation

- `Series` of five-digit ZIP Codes from a dictionary of city-name/five-digit-ZIP-Code key–value pairs
- Intentionally entered an invalid ZIP Code for Miami

In [1]:

```
import pandas as pd
```

In [2]:

```
zips = pd.Series({'Boston': '02215', 'Miami': '3310'})
```

In [3]:

```
zips
```

Out[3]:

```
Boston    02215
Miami      3310
dtype: object
```

Data Validation (cont.)

- The “second column” represents the `Series` ’ ZIP Code *values* (from the dictionary’s values)
- The “first column” represents their *indices* (from the dictionary’s keys)
- Can use regular expressions with Pandas to validate data
- The **`str` attribute** of a `Series` provides string-processing and various regular expression methods
- Use the `str` attribute’s **`match` method** to check whether each ZIP Code is valid:

In [4]:

```
zips.str.match(r'\d{5}')
```

Out[4]:

```
Boston     True
Miami     False
dtype: bool
```

- `match` applies the regular expression `\d{5}` to *each* `Series` element
- Returns a new `Series` containing `True` for each valid element

Data Validation (cont.)

- Several ways to deal with invalid data
- One is to catch it at its source and interact with the source to correct the value
  - Not always possible
- In the case of the bad Miami ZIP Code of `3310`, we might look for Miami ZIP Codes beginning with 3310
  - There are two— `33101` and `33109`
  - We could pick one of those

Data Validation (cont.)

- Sometimes, rather than matching an *entire* value to a pattern, you’ll want to know whether a value contains a *substring* that matches the pattern
- Use method **`contains`** instead of `match`

In [5]:

```
cities = pd.Series(['Boston, MA 02215', 'Miami, FL 33101'])
```

In [6]:

```
cities
```

Out[6]:

```
0    Boston, MA 02215
1     Miami, FL 33101
dtype: object
```

In [7]:

```
cities.str.contains(r' [A-Z]{2} ')
```

Out[7]:

```
0    True
1    True
dtype: bool
```

In [8]:

```
cities.str.match(r' [A-Z]{2} ')
```

Out[8]:

```
0    False
1    False
dtype: bool
```

Reformatting Your Data

- Consider munging data into a different format
- Assume that an application requires U.S. phone numbers in the format ###-###-####
- The phone numbers have been provided to us as 10-digit strings without hyphens

In [9]:

```
contacts = [['Mike Green', 'demo1@deitel.com', '5555555555'],
            ['Sue Brown', 'demo2@deitel.com', '5555551234']]
```

In [10]:

```
contactsdf = pd.DataFrame(contacts,
                          columns=['Name', 'Email', 'Phone'])
```

In [11]:

```
contactsdf
```

Out[11]:

	Name	Email	Phone
0	Mike Green	demo1@deitel.com	5555555555
1	Sue Brown	demo2@deitel.com	5555551234

Reformatting Your Data (cont.)

- Munge the data with functional-style programming
- Can *map* the phone numbers to the proper format by calling the `Series` method **`map`** on the `DataFrame`'s `'Phone'` column
- `map`'s argument is a *function* that receives a value and returns the *mapped* value
- Our function `get_formatted_phone` maps 10 consecutive digits into the format ###-###-####

In [12]:

```
import re
```

In [13]:

```
def get_formatted_phone(value):
    result = re.fullmatch(r'(\d{3})(\d{3})(\d{4})', value)
    return '-'.join(result.groups()) if result else value
```

Reformatting Your Data (cont.)

- Regular expression in the block's first statement matches *only* 10 consecutive digits
- Captures substrings containing the first three digits, the next three digits and the last four digits
- `return` statement:
  - If `result` is `None`, returns `value` unmodified
  - Otherwise, calls `result.groups()` to get a tuple containing the captured substrings and pass that tuple to string method `join` to concatenate the elements, separating each from the next with `'-'` to form the mapped phone number

Reformatting Your Data (cont.)

- `Series` method `map` returns a new `Series` containing the results of calling its function argument for each value in the column

In [14]:

```
formatted_phone = contactsdf['Phone'].map(get_formatted_phone)
```

In [15]:

```
formatted_phone
```

Out[15]:

```
0    555-555-5555
1    555-555-1234
Name: Phone, dtype: object
```

- Once you've confirmed that the data is in the correct format, you can update it in the original `DataFrame`

In [16]:

```
contactsdf['Phone'] = formatted_phone
```

In [17]:

```
contactsdf
```

Out[17]:

	Name	Email	Phone
0	Mike Green	demo1@deitel.com	555-555-5555
1	Sue Brown	demo2@deitel.com	555-555-1234
