Custom Python for MMS Supplement

John Strickler

Version 1.0, March 2021

Table of Contents

Chapter 1: Serializing Data

Objectives

- Have a good understanding of the XML format
- Know which modules are available to process XML
- Use lxml ElementTree to create a new XML file
- Parse an existing XML file with ElementTree
- Using XPath for searching XML nodes
- Load JSON data from strings or files
- Write JSON data to strings or files
- Read and write CSV data
- Read and write YAML data

Which module to use?

- Bewildering array of XML modules
- Some are SAX, some are DOM
- Use xml.etree.ElementTree

When you are ready to process Python with XML, you turn to the standard library, only to find a number of different modules with confusing names.

To cut to the chase, use **lxml.etree**, which is based on **ElementTree** with some nice extra features, such as pretty-printing. While not part of the core Python library, it is provided by the Anaconda bundle.

If `lxml.etree` is not available, you can use **xml.etree.ElementTree** from the core library.

Getting Started With ElementTree

- Import xml.etree.ElementTree (or lxml.etree) as ET for convenience
- Parse XML or create empty ElementTree

ElementTree is part of the Python standard library; lxml is included with the Anaconda distribution.

Since putting "xml.etree.ElementTree" in front of its methods requires a lot of extra typing , it is typical to alias xml.etree.ElementTree to just ET when importing it: import xml.etree.ElementTree as ET

You can check the version of ElementTree via the VERSION attribute:

```
import xml.etree.ElementTree as ET
print(ET.VERSION)
```

How ElementTree Works

- ElementTree contains root Element
- Document is tree of Elements

In ElementTree, an XML document consists of a nested tree of Element objects. Each Element corresponds to an XML tag.

An ElementTree object serves as a wrapper for reading or writing the XML text.

If you are parsing existing XML, use ElementTree.parse(); this creates the ElementTree wrapper and the tree of Elements. You can then navigate to, or search for, Elements within the tree. You can also insert and delete new elements.

If you are creating a new document from scratch, create a top-level (AKA "root") element, then create child elements as needed.

```
element = root.find('sometag')
for subelement in element:
    print(subelement.tag)
print(element.get('someattribute'))
```

Elements

- Element has
  - Tag name
  - Attributes (implemented as a dictionary)
  - Text
  - Tail
  - Child elements (implemented as a list) (if any)
- SubElement creates child of Element

When creating a new Element, you can initialize it with the tag name and any attributes. Once created, you can add the text that will be contained within the element's tags, or add other attributes.

When you are ready to save the XML into a file, initialize an ElementTree with the root element.

The **Element** class is a hybrid of list and dictionary. You access child elements by treating it as a list. You access attributes by treating it as a dictionary. (But you can't use subscripts for the attributes – you must use the get() method).

The Element object also has several useful properties: **tag** is the element's tag; **text** is the text contained inside the element; **tail** is any text following the element, before the next element.

The **SubElement** class is a convenient way to add children to an existing Element.

TIP	Only the tag property of an Element is required; other properties are optional.

Table 1. Element properties and methods

Property	Description
append(element)	Add a subelement element to end of subelements
attrib	Dictionary of element's attributes
clear()	Remove all subelements
find(path)	Find first subelement matching path
findall(path)	Find all subelements matching path
findtext(path)	Shortcut for find(path).text
get(attr)	Get an attribute; Shortcut for attrib.get()
getiterator()	Returns an iterator over all descendants
getiterator(path)	Returns an iterator over all descendants matching path
insert(pos,element)	Insert subelement element at position pos
items()	Get all attribute values; Shortcut for attrib.items()
keys()	Get all attribute names; Shortcut for attrib.keys()
remove(element)	Remove subelement element
set(attrib,value)	Set an attribute value; shortcut for attr[attrib] = value
tag	The element's tag
tail	Text following the element
text	Text contained within the element

Table 2. ElementTree properties and methods

Property	Description
find(path)	Finds the first toplevel element with given tag; shortcut for getroot().find(path).
findall(path)	Finds all toplevel elements with the given tag; shortcut for getroot().findall(path).
findtext(path)	Finds element text for first toplevel element with given tag; shortcut for getroot().findtext(path).
getiterator(path)	Returns an iterator over all descendants of root node matching path. (All nodes if path not specified)
getroot()	Return the root node of the document
parse(filename) parse(fileobj)	Parse an XML source (filename or file-like object)
write(filename,encoding)	Writes XML document to filename, using encoding (Default us-ascii).

Creating a New XML Document

- Create root element
- Add descendants via SubElement
- Use keyword arguments for attributes
- Add text after element created
- Create ElementTree for import/export

To create a new XML document, first create the root (top-level) element. This will be a container for all other elements in the tree. If your XML document contains books, for instance, the root document might use the "books" tag. It would contain one or more "book" elements, each of which might contain author, title, and ISBN elements.

Once the root element is created, use SubElement to add elements to the root element, and then nested Elements as needed. SubElement returns the new element, so you can assign the contents of the tag to the **text** attribute.

Once all the elements are in place, you can create an ElementTree object to contain the elements and allow you to write out the XML text. From the ElementTree object, call write.

To output an XML string from your elements, call ET.tostring(), passing the root of the element tree as a parameter. It will return a bytes object (pure ASCII), so use .decode() to convert it to a normal Python string.

For an example of creating an XML document from a data file, see **xml_create_knights.py** in the EXAMPLES folder

Example

xml_create_movies.py

```
#!/usr/bin/env python

# from xml.etree import ElementTree as ET
import lxml.etree as ET

movie_data = [
    ('Jaws', 'Spielberg, Stephen'),
    ('Vertigo', 'Alfred Hitchcock'),
    ('Blazing Saddles', 'Brooks, Mel'),
    ('Princess Bride', 'Reiner, Rob'),
    ('Avatar', 'Cameron, James'),
]

movies = ET.Element('movies')

for name, director in movie_data:
    movie = ET.SubElement(movies, 'movie', name=name)
    ET.SubElement(movie, 'director').text = director

print(ET.tostring(movies, pretty_print=True).decode())

doc = ET.ElementTree(movies)

doc.write('movies.xml')
```

xml_create_movies.py

```
<movies>
  <movie name="Jaws">
    <director>Spielberg, Stephen</director>
  </movie>
  <movie name="Vertigo">
    <director>Alfred Hitchcock</director>
  </movie>
  <movie name="Blazing Saddles">
    <director>Brooks, Mel</director>
  </movie>
  <movie name="Princess Bride">
    <director>Reiner, Rob</director>
  </movie>
  <movie name="Avatar">
    <director>Cameron, James</director>
  </movie>
</movies>
```

Parsing An XML Document

- Use ElementTree.parse()
- returns an ElementTree object
- Use get* or find* methods to select an element

Use the parse() method to parse an existing XML document. It returns an ElementTree object, from which you can find the root, or any other element within the document.

To get the root element, use the getroot() method.

Example

```
import xml.etree.ElementTree as ET

doc = ET.parse('solar.xml')

root = doc.getroot()
```

Navigating the XML Document

- Use find() or findall()
- Element is iterable of it children
- findtext() retrieves text from element

To find the first child element with a given tag, use find('tag'). This will return the first matching element. The findtext('tag') method is the same, but returns the text within the tag.

To get all child elements with a given tag, use the findall('tag') method, which returns a list of elements.

to see whether a node was found, say

```
if node is None:
```

but to check for existence of child elements, say

```
if len(node) > 0:
```

A node with no children tests as false because it is an empty list, but it is not None.

TIP The ElementTree object also supports the find() and findall() methods of the Element object, searching from the root object.

Example

xml_planets_nav.py

```
#!/usr/bin/env python
'''Use etree navigation to extract planets from solar.xml'''
import lxml.etree as ET


def main():
    '''Program entry point'''
    doc = ET.parse('../DATA/solar.xml')  ①

    solar_system = doc.getroot()  ②

    print(solar_system)
    print()

    inner = solar_system.find('innerplanets')  ③
    print('Inner:')

    for planet in inner:  ④
        if planet.tag == 'planet':
            print('\t', planet.get("planetname", "NO NAME"))

    outer = solar_system.find('outerplanets')
    print('Outer:')

    for planet in outer:
        print('\t', planet.get("planetname"))

    plutoids = solar_system.find('dwarfplanets')
    print('Dwarf:')

    for planet in plutoids:
        print('\t', planet.get("planetname"))


if __name__ == '__main__':
    main()
```

xml_planets_nav.py

```
<Element solarsystem at 0x7f9e900575f0>

Inner:
     Mercury
     Venus
     Earth
     Mars
Outer:
     Jupiter
     Saturn
     Uranus
     Neptune
Dwarf:
     Pluto
```

Example

xml_read_movies.py

```
#!/usr/bin/env python

# import xml.etree.ElementTree as ET
import lxml.etree as ET

movies_doc = ET.parse('movies.xml')  ①

movies = movies_doc.getroot()  ②

for movie in movies:  ③
    print('{} by {}'.format(
        movie.get('name'),  ④
        movie.findtext('director'),  ⑤
    )
    )
```

① read and parse the XML file

② get the root element (<movies>)

③ loop through children of root element

④ get 'name' attribute of movie element

⑤ get 'director' attribute of movie element

xml_read_movies.py

```
Jaws by Spielberg, Stephen
Vertigo by Alfred Hitchcock
Blazing Saddles by Brooks, Mel
Princess Bride by Reiner, Rob
Avatar by Cameron, James
```

Using XPath

- Use simple XPath patterns Works with find* methods

When a simple tag is specified, the find* methods only search for subelements of the current element. For more flexible searching, the find* methods work with simplified **XPath** patterns. To find all tags named 'spam', for instance, use `.//spam`.

```
.//movie
presidents/president/name/last
```

Example

xml_planets_xpath1.py

```
#!/usr/bin/env python

# import xml.etree.ElementTree as ET
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml')  ①

inner_nodes = doc.findall('innerplanets/planet')  ②

outer_nodes = doc.findall('outerplanets/planet')  ③

print('Inner:')
for planet in inner_nodes:  ④
    print('\t', planet.get("planetname"))  ⑤

print('Outer:')
for planet in outer_nodes:  ④
    print('\t', planet.get("planetname"))  ⑤
```

① parse XML file

② find all elements (relative to root element) with tag "planet" under "innerplanets" element

③ find all elements with tag "planet" under "outerplanets" element

④ loop through search results

⑤ print "name" attribute of planet element

xml_planets_xpath1.py

```
Inner:
     Mercury
     Venus
     Earth
     Mars
Outer:
     Jupiter
     Saturn
     Uranus
     Neptune
```

Example

xml_planets_xpath2.py

```
#!/usr/bin/env python

# import xml.etree.ElementTree as ET
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml')

jupiter = doc.find('.//planet[@planetname="Jupiter"]')

if jupiter is not None:
    for moon in jupiter:
        print(moon.text)  # grab attribute
```

xml_planets_xpath2.py

```
Metis
Adrastea
Amalthea
Thebe
Io
Europa
Gannymede
Callista
Themisto
Himalia
Lysithea
Elara
```

Table 3. ElementTree XPath Summary

Syntax	Meaning
tag	Selects all child elements with the given tag. For example, “spam” selects all child elements named “spam”, “spam/egg” selects all grandchildren named “egg” in all child elements named “spam”. You can use universal names (“{url}local”) as tags.
*	Selects all child elements. For example, “*/egg” selects all grandchildren named “egg”.
.	Select the current node. This is mostly useful at the beginning of a path, to indicate that it’s a relative path.
//	Selects all subelements, on all levels beneath the current element (search the entire subtree). For example, “.//egg” selects all “egg” elements in the entire tree.
..	Selects the parent element.
[@attrib]	Selects all elements that have the given attribute. For example, “.//a[@href]” selects all “a” elements in the tree that has a “href” attribute.
[@attrib=’value’]	Selects all elements for which the given attribute has the given value. For example, “.//div[@class=’sidebar’]” selects all “div” elements in the tree that has the class “sidebar”. In the current release, the value cannot contain quotes.
parent_tag[*child_tag*]	Selects all parent elements that has a child element named *child_tag*. In the current version, only a single tag can be used (i.e. only immediate children are supported). Parent tag can be *****.

About JSON

- Lightweight, human-friendly format for data
- Contains dictionaries and lists
- Stands for JavaScript Object Notation
- Looks like Python
- Basic types: Number, String, Boolean, Array, Object
- White space is ignored
- Stricter rules than Python

JSON is a lightweight and human-friendly format for sharing or storing data. It was developed and popularized by Douglas Crockford starting in 2001.

A JSON file contains objects and arrays, which correspond exactly to Python dictionaries and lists.

White space is ignored, so JSON may be formatted for readability.

Data types are Number, String, and Boolean. Strings are enclosed in double quotes (only); numbers look like integers or floats; Booleans are represented by true or false; null (None in Python) is represented by null.

Reading JSON

- json module in standard library
- json.load() parse from file-like object
- json.loads() parse from string
- Both methods return Python dict or list

To read a JSON file, import the json module. Use json.loads() to parse a string containing valid JSON. Use json.load() to read JSON from a file-like object0.

Both methods return a Python dictionary containing all the data from the JSON file.

Example

json_read.py

```
#!/usr/bin/env python

import json

with open('../DATA/solar.json') as solar_in:  ①
    solar = json.load(solar_in)  ②

# json.loads(STRING)
# json.load(FILE_OBJECT)

# print(solar)

print(solar['innerplanets'])  ③
print('*' * 60)
print(solar['innerplanets'][0]['name'])
print('*' * 60)
for planet in solar['innerplanets'] + solar['outerplanets']:
    print(planet['name'])

print("*" * 60)
for group in solar:
    if group.endswith('planets'):
        for planet in solar[group]:
            print(planet['name'])
```

① open JSON file for reading

② load from file object and convert to Python data structure

③ solar is just a Python dictionary

json_read.py

```
[{'name': 'Mercury', 'moons': None}, {'name': 'Venus', 'moons': None}, {'name': 'Earth',
'moons': ['moon']}, {'name': 'Mars', 'moons': ['Deimos', 'Phobos']}]
************************************************************
Mercury
************************************************************
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
************************************************************
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto
```

Writing JSON

- Use json.dumps() or json.dump()

To output JSON to a string, use json.dumps(). To output JSON to a file, pass a file-like object to json.dump(). In both cases, pass a Python data structure as the data to be output.

Example

json_write.py

```
#!/usr/bin/env python

import json

george = [
    {
        'num': 1,
        'lname': 'Washington',
        'fname': 'George',
        'dstart': [1789, 4, 30],
        'dend': [1797, 3, 4],
        'birthplace': 'Westmoreland County',
        'birthstate': 'Virginia',
        'dbirth': [1732, 2, 22],
        'ddeath': [1799, 12, 14],
        'assassinated': False,
        'party': None,
    },
    {
        'spam': 'ham',
        'eggs': [1.2, 2.3, 3.4],
        'toast': {'a': 5, 'm': 9, 'c': 4},
    }
]  ①

js = json.dumps(george, indent=4)  ②
print(js)

with open('george.json', 'w') as george_out:  ③
    json.dump(george, george_out, indent=4)  ④
```

① Python data structure

② dump structure to JSON string

③ open file for writing

④ dump structure to JSON file using open file object

json_write.py

```
[
    {
        "num": 1,
        "lname": "Washington",
        "fname": "George",
        "dstart": [
            1789,
            4,
            30
        ],
        "dend": [
            1797,
            3,
            4
        ],
        "birthplace": "Westmoreland County",
        "birthstate": "Virginia",
        "dbirth": [
            1732,
            2,
            22
        ],
        "ddeath": [
            1799,
            12,
            14
        ],
        "assassinated": false,
        "party": null
    },
    {
        "spam": "ham",
        "eggs": [
            1.2,
            2.3,
            3.4
        ],
        "toast": {
            "a": 5,
            "m": 9,
            "c": 4
        }
    }
]
```

Customizing JSON

- JSON data types limited
- simple cases — dump dict
- create custom encoders

The JSON spec only supports a limited number of datatypes. If you try to dump a data structure contains dates, user-defined classes, or many other types, the json encoder will not be able to handle it.

You can a custom encoder for various data types. To do this, write a function that expects one Python object, and returns some object that JSON can parse, such as a string or dictionary. The function can be called anything. Specify the function with the **default** parameter to json.dump().

The function should check the type of the object. If it is a type that needs special handling, return a JSON-friendly version, otherwise just return the original object.

Table 4. Python types that JSON can encode

Python	JSON
dict	object
list	array
str	string
int	number (int)
float	number (real)
True	true
False	false
None	null

NOTE see the file **json_custom_singledispatch.py** in EXAMPLES for how to use the **singledispatch** decorator (in the **functools** module to handle multiple data types.

Example

json_custom_encoding.py

```
#!/usr/bin/env python
#
import json
from datetime import date


class Parrot():  ①
    def __init__(self, name, color):
        self._name = name
        self._color = color

    @property
    def name(self):  ②
        return self._name

    @property
    def color(self):
        return self._color


parrots = [  ③
    Parrot('Polly', 'green'),  #
    Parrot('Peggy', 'blue'),
    Parrot('Roger', 'red'),
]


def encode(obj):  ④
    if isinstance(obj, date):  ⑤
        return obj.ctime()  ⑥
    elif isinstance(obj, Parrot):  ⑦
        return {'name': obj.name, 'color': obj.color}  ⑧
    return obj  ⑨


data = {  ⑩
    'spam': [1, 2, 3],
    'ham': ('a', 'b', 'c'),
    'toast': date(2014, 8, 1),
    'parrots': parrots,
}

print(json.dumps(data, default=encode, indent=4))  ⑪
```

① sample user-defined class (not JSON-serializable)

② JSON does not understand arbitrary properties

③ list of Parrot objects

④ custom JSON encoder function

⑤ check for date object

⑥ convert date to string

⑦ check for Parrot object

⑧ convert Parrot to dictionary

⑨ if not processed, return object for JSON to parse with default parser

⑩ dictionary of arbitrary data

⑪ convert Python data to JSON data; 'default' parameter specifies function for custom encoding; 'indent' parameter says to indent and add newlines for readability

json_custom_encoding.py

```
{
    "spam": [
        1,
        2,
        3
    ],
    "ham": [
        "a",
        "b",
        "c"
    ],
    "toast": "Fri Aug  1 00:00:00 2014",
    "parrots": [
        {
            "name": "Polly",
            "color": "green"
        },
        {
            "name": "Peggy",
            "color": "blue"
        },
        {
            "name": "Roger",
            "color": "red"
        }
    ]
}
```

Reading and writing YAML

- yaml module from PYPI
- syntax like **json** module
- yaml.load(), dump() parse from/to file-like object
- yaml.loads(), dumps() parse from/to string

YAML is a structured data format which is a superset of JSON. However, YAML allows for a more compact and readable format.

Reading and writing YAML uses the same syntax as JSON, other than using the **yaml** module, which is NOT in the standard library. To install the **yaml** module:

```
pip install pyyaml
```

To read a YAML file (or string) into a Python data structure, use `yaml.load(file_object)` or `yaml.loads(string)`.

To write a data structure to a YAML file or string, use `yaml.dump(data, file_object)` or `yaml.dumps(data)`.

You can also write custom YAML processors.

NOTE YAML parsers will parse JSON data

Example

yaml_read_solar.py

```
#!/usr/bin/env python
# (c) 2015 John Strickler
#
import yaml

PLANET_SECTIONS = "inner outer plutoid".split()

with open('../DATA/solar.yaml') as solar_in:
    solar_data = yaml.load(solar_in, Loader=yaml.FullLoader)

star = solar_data['star']
print("Our star is {}\n".format(star))

for section in PLANET_SECTIONS:
    for planet in solar_data[section]:
        print(planet['name'])
        for moon in planet['moons']:
            print("\t{}".format(moon))
```

yaml_read_solar.py

```
Our star is Sun

Mercury
    None
Venus
    None
Earth
    Moon
Mars
    Deimos
    Phobos
    Metis
Jupiter
    Adrastea
    Amalthea
    Thebe
    Io
    Europa
    Gannymede
    Callista
    Themisto
    Himalia
    Lysithea
    Elara
Saturn
    Rhea
    Hyperion
    Titan
    Iapetus
    Mimas
```

...

Example

yaml_create_file.py

```
import sys
from datetime import date
import yaml

potus = {
    'presidents': [
         {
            'lastname': 'Washington',
            'firstname': 'George',
            'dob': date(1732, 2, 22),
            'dod': date(1799, 12, 14),
            'birthplace': 'Westmoreland County',
            'birthstate': 'Virginia',
            'term': [ date(1789, 4, 30), date(1797, 3, 4) ],
            'assassinated': False,
            'party': None,
        },
        {
            'lastname': 'Adams',
            'firstname': 'John',
            'dob': date(1735, 10, 30),
            'dod': date(1826, 7, 4),
            'birthplace': 'Braintree, Norfolk',
            'birthstate': 'Massachusetts',
            'term': [date(1797, 3, 4), date(1801, 3, 4)],
            'assassinated': False,
            'party': 'Federalist',

        }
    ]
}

with open('potus.yaml', 'w') as potus_out:
    yaml.dump(potus, potus_out)

yaml.dump(potus, sys.stdout)
```

yaml_create_file.py

```
presidents:
- assassinated: false
  birthplace: Westmoreland County
  birthstate: Virginia
  dob: 1732-02-22
  dod: 1799-12-14
  firstname: George
  lastname: Washington
  party: null
  term:
  - 1789-04-30
  - 1797-03-04
- assassinated: false
  birthplace: Braintree, Norfolk
  birthstate: Massachusetts
  dob: 1735-10-30
  dod: 1826-07-04
  firstname: John
  lastname: Adams
  party: Federalist
  term:
  - 1797-03-04
  - 1801-03-04
```

Reading CSV data

- Use csv module
- Create a reader with any iterable (e.g. file object)
- Understands Excel CSV and tab-delimited files
- Can specify alternate configuration
- Iterate through reader to get rows as lists of columns

To read CSV data, use the reader() method in the csv module.

To create a reader with the default settings, use the reader() constructor. Pass in an iterable – typically, but not necessarily, a file object.

You can also add parameters to control the type of quoting, or the output delimiters.

Example

csv_read.py

```
#!/usr/bin/env python
import csv

with open('../DATA/knights.csv') as knights_in:
    rdr = csv.reader(knights_in)  ①
    for name, title, color, quest, comment, number, ladies in rdr:  ②
        print('{:4s} {:9s} {}'.format(
            title, name, quest
        ))
```

① create CSV reader

② Read and unpack records one at a time; each record is a list

csv_read.py

```
King Arthur    The Grail
Sir  Lancelot  The Grail
Sir  Robin     Not Sure
Sir  Bedevere  The Grail
Sir  Gawain    The Grail
```

...

Nonstandard CSV

- Variations in how CSV data is written
- Most common alternate is for Excel
- Add parameters to reader/writer

You can customize how the CSV parser and generator work by passing extra parameters to csv.reader() or csv.writer(). You can change the field and row delimiters, the escape character, and for output, what level of quoting.

You can also create a "dialect", which is a custom set of CSV parameters. The csv module includes one extra dialect, **excel**, which handles CSV files generated by Microsoft Excel. To use it, specify the *dialect* parameter:

```
rdr = csv.reader(csvfile, dialect='excel')
```

Table 5. CSV reader()/writer() Parameters

Parameter	**Meaning**
quotechar	One-character string to use as quoting character (default: '"')
delimiter	One-character string to use as field separator (default: ',')
skipinitialspace	If True, skip white space after field separator (default: False)
lineterminator	The character sequence which terminates rows (default: depends on OS)
quoting	When should quotes be generated when writing CSV csv.QUOTE_MINIMAL – only when needed (default) csv.QUOTE_ALL – quote all fields csv.QUOTE_NONNUMERIC – quote all fields that are not numbers csv.QUOTE_NONE – never put quotes around fields
escapechar	One-character string to escape delimiter when quoting is set to csv.QUOTE_NONE
doublequote	Control quote handling inside fields. When True, two consecutive quotes are read as one, and one quote is written as two. (default: True)

Example

csv_nonstandard.py

```
#!/usr/bin/env python
import csv

with open('../DATA/computer_people.txt') as computer_people_in:
    rdr = csv.reader(computer_people_in, delimiter=';')  ①

    for first_name, last_name, known_for, birth_date in rdr:  ②
        print('{}: {}'.format(last_name, known_for))
```

① specify alternate field delimiter

② iterate over rows of data — csv reader is a generator

csv_nonstandard.py

```
Gates: Gates Foundation
Jobs: Apple
Wall: Perl
Allen: Microsoft
Ellison: Oracle
Gates: Microsoft
Zuckerberg: Facebook
Brin: Google
Page: Google
Torvalds: Linux
```

Using csv.DictReader

- Returns each row as dictionary
- Keys are field names
- Use header or specify

Instead of the normal reader, you can create a dictionary-based reader by using the DictReader class.

If the CSV file has a header, it will parse the header line and use it as the field names. Otherwise, you can specify a list of field names with the **fieldnames** parameter. For each row, you can look up a field by name, rather than position.

Example

csv_dictreader.py

```
#!/usr/bin/env python
import csv

field_names = ['term', 'firstname', 'lastname', 'birthplace', 'state', 'party'] ①

with open('../DATA/presidents.csv') as presidents_in:
    rdr = csv.DictReader(presidents_in, fieldnames=field_names)  ②
    for row in rdr:  ③
        print('{:25s} {:12s} {}'.format(row['firstname'], row['lastname'], row['party']))
④
        # string .format can use keywords from an unpacked dict as well:
        # print('{firstname:25s} {lastname:12s} {party}'.format(**row))
```

① field names, which will become dictionary keys on each row

② create reader, passing in field names (if not specified, uses first row as field names)

③ iterate over rows in file

④ print results with formatting

csv_dictreader.py

```
George                  Washington   no party
John                    Adams        Federalist
Thomas                  Jefferson    Democratic - Republican
James                   Madison      Democratic - Republican
James                   Monroe       Democratic - Republican
John Quincy             Adams        Democratic - Republican
Andrew                  Jackson      Democratic
Martin                  Van Buren    Democratic
William Henry           Harrison     Whig
John                    Tyler        Whig
James Knox              Polk         Democratic
Zachary                 Taylor       Whig
Millard                 Fillmore     Whig
Franklin                Pierce       Democratic
James                   Buchanan     Democratic
Abraham                 Lincoln      Republican
Andrew                  Johnson      Republican
Ulysses Simpson         Grant        Republican
Rutherford Birchard     Hayes        Republican
James Abram             Garfield     Republican
```

...

Writing CSV Data

- Use csv.writer()
- Parameter is file-like object (must implement write() method)
- Can specify parameters to writer constructor
- Use writerow() or writerows() to output CSV data

To output data in CSV format, first create a writer using csv.writer(). Pass in a file-like object.

For each row to write, call the writerow() method of the writer, passing in an iterable with the values for that row.

To modify how data is written out, pass parameters to the writer.

TIP On Windows, to prevent double-spaced output, add `lineterminator='\n'` when creating a CSV writer.

Example

csv_write.py

```
#!/usr/bin/env python
import sys
import csv

data = [
    ('February', 28, 'The shortest month, with 28 or 29 days'),
    ('March', 31, 'Goes out like a "lamb"'),
    ('April', 30, 'Its showers bring May flowers'),
]

with open('../TEMP/stuff.csv', 'w') as stuff_in:
    if sys.platform == 'win32':
        wtr = csv.writer(stuff_in, lineterminator='\n') ①
    else:
        wtr = csv.writer(stuff_in) ①
    for row in data:
        wtr.writerow(row) ②
```

① create CSV writer from file object that is opened for writing; on windows, need to set output line terminator to '\n'

② write one row (of iterables) to output file

Pickle

- Use the pickle module
- Create a binary stream that can be saved to file
- Can also be transmitted over the network

Python uses the pickle module for data serialization.

To create pickled data, use either pickle.dump() or pickle.dumps(). Both functions take a data structure as the first argument. dumps() returns the pickled data as a string. dump () writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use pickle.load(), which takes a file-like object that has been open for writing, or pickle.loads() which reads from a string. Both functions return the original data structure that had been pickled.

NOTE The syntax of the **json** module is based on the **pickle** module.

Example

pickling.py

```
#!/usr/bin/env python
"""

@author: jstrick
Created on Sat Mar 16 00:47:05 2013

"""
import pickle
from pprint import pprint

①
airports = {
    'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
    'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'
}

colors = [
    'red', 'blue', 'green', 'yellow', 'black',
    'white', 'orange', 'brown', 'purple'
]

data = [  ②
    colors,
    airports,
]

with open('../TEMP/pickled_data.pic', 'wb') as pic_out:  ③
    pickle.dump(data, pic_out)  ④

with open('../TEMP/pickled_data.pic', 'rb') as pic_in:  ⑤
    pickled_data = pickle.load(pic_in)  ⑥

pprint(pickled_data)  ⑦
```

① some data structures

② list of data structures

③ open pickle file for writing in binary mode

④ serialize data structures to pickle file

⑤ open pickle file for reading in binary mode

⑥ de-serialize pickle file back into data structures

⑦ view data structures

pickling.py

```
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'}]
```

Chapter 1 Exercises

Exercise 1-1 (xwords.py)

Using ElementTree, create a new XML file containing all the words that start with 'x' from words.txt. The root tag should be named 'words', and each word should be contained in a 'word' tag. The finished file should look like this:

```
<words>
   <word>xanthan</word>
   <word>xanthans</words>
   and so forth
</words>
```

Exercise 1-2 (xpresidents.py)

Use ElementTree to parse presidents.xml. Loop through and print out each president's first and last names and their state of birth.

Exercise 1-3 (jpresidents.py)

Rewrite xpresidents.py to parse presidents.json using the json module.

Exercise 1-4 (cpresidents.py)

Rewrite xpresidents.py to parse presidents.csv using the csv module.

Exercise 1-5 (pickle_potus.py)

Write a script which reads the data from presidents.csv into an dictionary where the key is the term number, and the value is another dictionary of data for one president.

Using the pickle module, Write the entire dictionary out to a file named presidents.pic.

Exercise 1-6 (unpickle_potus.py)

Write a script to open presidents.pic, and restore the data back into a dictionary.

Then loop through the array and print out each president's first name, last name, and party.

Chapter 2: The Python Imaging Library

Objectives

- Get an overview of Pillow
- Open and save image files in various formats
- Create image thumbnails
- Crop, cut, and paste images
- Transform images with filters

Pillow

- Adds imaging processing to the Python interpreter
- Possible uses
  - Image Archives (thumbnails, resizing, etc.)
  - Image Display (interfaces with GUIs)
  - Image Processing (filtering, conversion, rotating, etc.)

Pillow is a fork of the original PIL (Python Imaging Library). It is a comprehensive package for working with images. It has support for a large number of image types.

If you have existing code that uses the original PIL, you can probably just change the line

```
import Image
```

to

```
from PIL import Image
```

to make your code work with Pillow.

Table 6. Supported Image File Types

BMP
BUFR (identify only)
CUR (read only)
DCX (read only)
EPS (write-only)
FITS (identify only)
FLI, FLC (read only)
FPX (read only)
GBR (read only)
GD (read only)
GIF
GRIB (identify only)
HDF5 (identify only)
ICO (read only)
IM
IMT (read only)
IPTC/NAA (read only)
JPEG
MCIDAS (read only)
MIC (read only)
MPEG (identify only)
MSP
PALM (write only)
PCD (read only)
PCX
PDF (write only)
PIXAR (read only)
PNG
PPM
PSD (read only)
SGI (read only)

SPIDER
TGA (read only)
TIFF
WAL (read only)
WMF (identify only)
XBM
XPM (read only)
XV Thumbnails

The Image class

- Primary class of Pillow
- Used to process and manipulate the image
- Use attributes to examine image
- Can load from/save to file

The Image class is the main object of Pillow. An image is typically loaded from a file into an Image object, and from there it can be processed.

An image has several attributes that provide detailed information – format, size, mode, etc.

Once the image is loaded, there are many methods for working with the image.

Reading and writing

- Open with Image.open(filename)
- Save with Image.save(filename, [filetype])
- Filename determines saved format

To get started, open an image with Image.open(). This returns an Image object, which is the basis for all processing.

The image can then be manipulated, and eventually saved in the same, or a different, format.

Open() only reads the file header, so opening a file is very fast. PIL only reads the actual content of the file if necessary.

NOTE	You can also use a file-like object in place of a filename

Example

pil_basics.py

```
#!/usr/bin/env python
from PIL import Image ①

im = Image.open('../DATA/felix_auto.jpeg')  ②
print(im.format)  ③
print(im.size)  ③
print(im.mode)  ③

im.save('felix_auto.png')
```

① import Image class from the python imaging library

② create an Image object from the file name

③ access properties of the image

pil_basics.py

```
JPEG
(3008, 2000)
RGB
```

Creating thumbnails

- Use IM.thumbnail(size)
- Size is tuple of height, width

To convert an image into a thumbnail, call the thumbnail() method. Pass in a tuple of height and width, then save in desired format.

Example

pil_thumb.py

```
#!/usr/bin/env python

from PIL import Image

size = 125, 125  ①
im = Image.open('../DATA/felix_auto.jpeg')
im.thumbnail(size)  ②
im.save('felix_auto_small.png')  ③
```

① set thumbnail size as tuple

② change image to thumbnail size

③ save image as new name

Coordinate System

- Points: (0,0) is upper left
- Rectangles: (0, 0, 800, 800) is entire 800x600 pixel image

Pillow uses a Cartesian pixel coordinate system, with (0,0) in the upper left corner. Note that the coordinates refer to the implied pixel corners; the center of a pixel addressed as (0, 0) actually lies at (0.5, 0.5).

Coordinates are usually passed to the library as 2-tuples (x, y). Rectangles are represented as 4-tuples, with the upper left corner given first. For example, a rectangle covering all of an 800x600 pixel image is written as (0, 0, 800, 600).

Cropping and pasting

- Use crop() to pull out a region from an image
- Use paste() to insert an image into another

The Image class contains methods allowing you to manipulate regions within an image. To extract a sub-rectangle from an image, use the crop() method. crop() requires a tuple containing the bounds of the region.

Use paste() to insert a previously extracted region.

Example

pil_crop.py

```
#!/usr/bin/env python
from PIL import Image

box = (1200, 100, 2000, 400)  ①
im = Image.open('../DATA/felix_auto.jpeg')
region = im.crop(box)  ②
region = region.transpose(Image.FLIP_LEFT_RIGHT)  ③
im.paste(region,box)  ④
im.save('felix_auto_cropped.jpg')  ⑤
```

① create tuple with boundaries of desired area

② grab section of image

③ flip image section left-to-right

④ paste flipped section back into picture

⑤ save as a new name

Rotating, resizing, and flipping

- Use IM.resize(), IM.rotate(), or im.transpose()
- These return new Image objects

The Image class contains several methods for transposing and resizing images.

resize() takes a tuple giving the new size. .rotate() takes the angle in degrees to rotate counter-clockwise. .transpose() is a convenience method for flipping or rotating an image.

To rotate the image in 90 degree steps, you can use transpose(), which can also be used to flip an image vertically or horizontally.

Example

pil_transpose.py

```
#!/usr/bin/env python

from PIL import Image

SMALLSIZE = 200, 200  ①

im = Image.open('../DATA/felix_auto.jpeg')
im_small = im.resize(SMALLSIZE)  ②

im1 = im_small.rotate(45)  ③
im1.save('felix_auto_45.jpg')

im2 = im_small.transpose(Image.FLIP_LEFT_RIGHT)  ④
im2.save('felix_auto_flipLR.jpg')

im3 = im_small.transpose(Image.FLIP_TOP_BOTTOM)  ⑤
im3.save('felix_auto_flipTB.jpg')
```

① make tuple for new image size

② resize image to 200x200

③ rotate image 45 degrees and save

④ flip image left-to-right and save

⑤ flip image top-to-bottom and save

Enhancing

- Many filters provided
- ImageFilter
  - filter(), point()
- ImageEnhance
  - contrast(), etc.
- New Image object is returned

Pillow has many filters to change the appearance of an image. These are in the ImageFilter module.

The point method can be used to translate the pixel values of an image (e.g. image contrast manipulation). Pass a function expecting one argument to this method. Each pixel is processed according to that function

You can quickly apply any simple conversion to an image, or combine point and paste to modify a region within an image.

The ImageEnhance module has functions to further process an image.

Example

pil_filter.py

```
#!/usr/bin/env python
from PIL import Image, ImageFilter, ImageEnhance  ①

SMALLSIZE = 240,160  ②
im = Image.open('../DATA/felix_auto.jpeg')
im_small = im.resize(SMALLSIZE)  ③

im_emboss = im_small.filter(ImageFilter.EMBOSS)  ④
im_emboss.save('felix_auto_emboss.jpg')  ⑤

im_blur = im_small.filter(ImageFilter.BLUR)  ⑥
im_blur.save('felix_auto_blur.jpg')

enh_bright = ImageEnhance.Brightness(im_small)
im_bright = enh_bright.enhance(2.5)
im_bright.save('felix_auto_bright.jpg')

enh_lowcontrast = ImageEnhance.Contrast(im_small)
im_lowcontrast = enh_bright.enhance(.5)
im_lowcontrast.save('felix_auto_lowcontrast.jpg')
```

① import filters from PIL

② create a tuple for new image size

③ resize to specified bounds

④ apply emboss filter

⑤ save as new file

⑥ apply blur filter

Chapter 2 Exercises

NOTE	In all cases, save with a new name – don't overwrite existing files.

Exercise 2-1 (salad_half.py)

Open the image salad.jpeg in the DATA folder. Save it as exactly half its original size.

Exercise 2-2 (salad_flip.py)

Open salad.jpeg, flip it horizontally, and save.

Exercise 2-3 (salad_thumb.py)

Open salad.jpeg, make a 50x50 thumbnail, and save.

Index