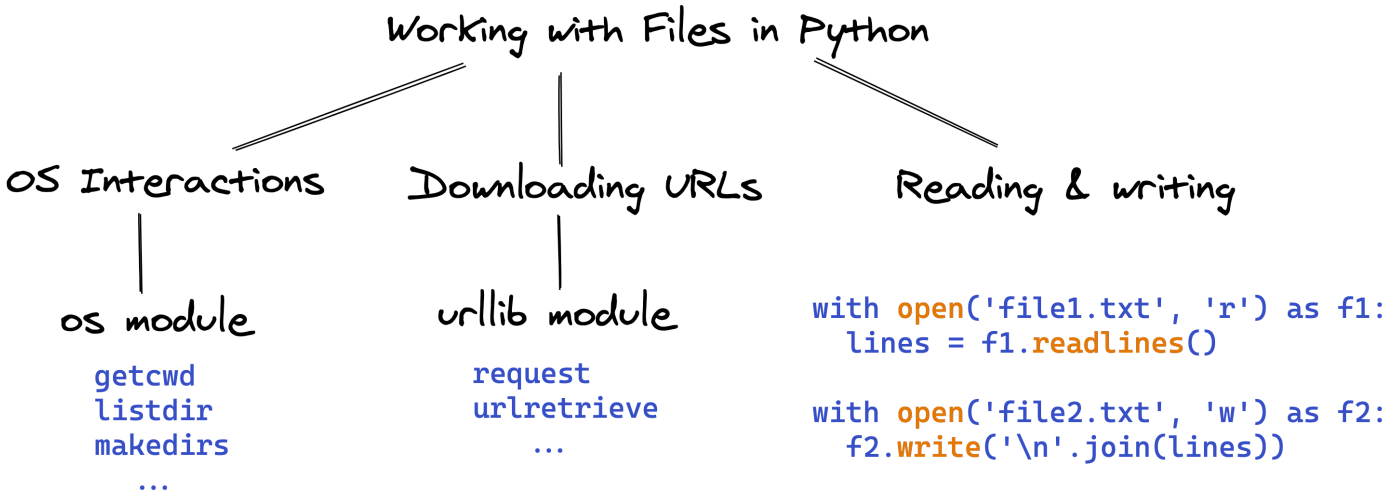
Reading from and Writing to Files using Python



This tutorial series is a beginner-friendly introduction to programming and data analysis using the Python programming language. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself. Check out the full series here:

1. First Steps with Python and Jupyter
2. A Quick Tour of Variables and Data Types
3. Branching using Conditional Statements and Loops
4. Writing Reusable Code Using Functions
5. Reading from and Writing to Files
6. Numerical Computing with Python and Numpy
7. Analyzing Tabular Data using Pandas
8. Data Visualization using Matplotlib & Seaborn
9. Exploratory Data Analysis - A Case Study

This tutorial covers the following topics:

- Interacting with the filesystem using the os module
- Downloading files from the internet using the `urllib` module
- Reading and processing data from text files
- Parsing data from CSV files into dictionaries & lists
- Writing formatted data back to text files

How to run the code

This tutorial is an executable Jupyter notebook hosted on Jovian. You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on Google Colab or Kaggle to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up Python, download the notebook and install the required libraries. We recommend using the Conda distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

> **Jupyter Notebooks**: This tutorial is a Jupyter notebook - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Interacting with the OS and filesystem

The `os` module in Python provides many functions for interacting with the OS and the filesystem. Let's import it and try out some examples.

```
import os
```

We can check the present working directory using the `os.getcwd` function.

```
os.getcwd()
```

```
'/Users/aakashns/workspace/zerotoanalyst/python-os-and-filesystem'
```

To get the list of files in a directory, use `os.listdir`. You pass an absolute or relative path of a directory as the argument to the function.

```
help(os.listdir)
```

```
Help on built-in function listdir in module posix:

listdir(path=None)
    Return a list containing the names of the files in the directory.

    path can be specified as either str, bytes, or a path-like object.  If path is
bytes,
      the filenames returned will also be bytes; in all other circumstances
      the filenames returned will be str.
    If path is None, uses the path='.'.
    On some platforms, path may also be specified as an open file descriptor;\
      the file descriptor must refer to a directory.
```

```
    If this functionality is unavailable, using it raises NotImplementedError.

    The list is in arbitrary order.  It does not include the special
    entries '.' and '..' even if they are present in the directory.
```

```
os.listdir('.') # relative path
```

```
['.jovianrc', '.ipynb_checkpoints', 'python-os-and-filesystem.ipynb']
```

```
os.listdir('/usr') # absolute path
```

```
['bin',
 'standalone',
 'libexec',
 'sbin',
 'local',
 'lib',
 'X11',
 'X11R6',
 'share']
```

Command to create new directory

You can create a new directory using `os.makedirs`. Let's create a new directory called `data`, where we'll later download some files.

```
os.makedirs('./data', exist_ok=True)
```

Can you figure out what the argument `exist_ok` does? Try using the `help` function or read the documentation.

exist ok true means no error will be thrown if directory already exists, the directory will be unaltered

Let's verify that the directory was created and is currently empty.

```
'data' in os.listdir('.')
```

```
True
```

```
os.listdir('./data')
```

```
[]
```

Let us download some files into the `data` directory using the `urllib` module.

```
url1 = 'https://gist.githubusercontent.com/aakashns/257f6e6c8719c17d0e498ea287d1a386/ra
url2 = 'https://gist.githubusercontent.com/aakashns/257f6e6c8719c17d0e498ea287d1a386/ra
url3 = 'https://gist.githubusercontent.com/aakashns/257f6e6c8719c17d0e498ea287d1a386/ra
```

```
from urllib.request import urlretrieve
```

```
urlretrieve(url1, './data/loans1.txt')
```

// downloading files to data directory

```
('./data/loans1.txt', <http.client.HTTPMessage at 0x7fe15cdcc898>)
```

```
urlretrieve(url2, './data/loans2.txt')
```

```
('./data/loans2.txt', <http.client.HTTPMessage at 0x7fe1573a5390>)
```

```
urlretrieve(url3, './data/loans3.txt')
```

```
('./data/loans3.txt', <http.client.HTTPMessage at 0x7fe15cdb8748>)
```

Let's verify that the files were downloaded.

```
os.listdir('./data')
```

```
['loans2.txt', 'loans3.txt', 'loans1.txt']
```

You can also use the `requests` library to dowload URLs, although you'll need to write some additional code to save the contents of the page to a file.

Reading from a file

To read the contents of a file, we first need to open the file using the built-in `open` function. The `open` function returns a file object and provides several methods for interacting with the file's contents.

```
file1 = open('./data/loans1.txt', mode='r')
```

The `open` function also accepts a `mode` argument to specifies how we can interact with the file. The following options are supported:

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

// binary mode is for files that can't be read as text eg: images

To view the contents of the file, we can use the `read` method of the file object.

```
file1_contents = file1.read()
```

```
print(file1_contents)
```

```
amount,duration,rate,down_payment
100000,36,0.08,20000
200000,12,0.1,
628400,120,0.12,100000
4637400,240,0.06,
42900,90,0.07,8900
916000,16,0.13,
45230,48,0.08,4300
991360,99,0.08,
423000,27,0.09,47200
```

The file contains information about loans. It is a set of comma-separated values (CSV).

> **CSVs**: A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. A CSV file typically stores tabular data (numbers and text) in plain text, in which case each line will have the same number of fields. (Wikipedia)

The first line of the file is the header, indicating what each of the numbers on the remaining lines represents. Each of the remaining lines provides information about a loan. Thus, the second line `100000,36,0.08,20000` represents a loan with:

- an *amount* of $`100000`,
- *duration* of 36 months,
- *rate of interest* of 8% per annum, and
- a down payment of $`20000`

The CSV is a standard file format used for sharing data for analysis and visualization. Over the course of this tutorial, we will read the data from these CSV files, process it, and write the results back to files. Before we continue, let's close the file using the `close` method (otherwise, Python will continue to hold the entire file in the RAM)

```
file1.close()
```

Once a file is closed, you can no longer read from it.

```
file1.read()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-2fcf3b5e742f> in <module>
----> 1 file1.read()

ValueError: I/O operation on closed file.
```

Closing files automatically using `with`

To close a file automatically after you've processed it, you can open it using the `with` statement.

```
with open('./data/loans2.txt') as file2:
    file2_contents = file2.read()
    print(file2_contents)
```

```
amount,duration,rate,down_payment
828400,120,0.11,100000
4633400,240,0.06,
42900,90,0.08,8900
983000,16,0.14,
15230,48,0.07,4300
```

Once the statements within the `with` block are executed, the `.close` method on `file2` is automatically invoked. Let's verify this by trying to read from the file object again.

```
file2.read()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-4a9205324152> in <module>
----> 1 file2.read()

ValueError: I/O operation on closed file.
```

Reading a file line by line

File objects provide a `readlines` method to read a file line-by-line.

```
with open('./data/loans3.txt', 'r') as file3:
    file3_lines = file3.readlines()
```

```
file3_lines
```

```
['amount,duration,rate,down_payment\n',
 '45230,48,0.07,4300\n',
 '883000,16,0.14,\n',
 '100000,12,0.1,\n',
 '728400,120,0.12,100000\n',
 '3637400,240,0.06,\n',
 '82900,90,0.07,8900\n',
 '316000,16,0.13,\n',
 '15230,48,0.08,4300\n',
 '991360,99,0.08,\n',
 '323000,27,0.09,4720010000,36,0.08,20000\n',
 '528400,120,0.11,100000\n',
 '8633400,240,0.06,\n',
 '12900,90,0.08,8900']
```

this newline coming at end could be stripped using strip function, strip function removes extra space at start or end or newline character (\n) eg:- file3_lines[0].strip()

Processing data from files

Before performing any operations on the data stored in a file, we need to convert the file's contents from one large string into Python data types. For the file `loans1.txt` containing information about loans in a CSV format, we can do the following:

- Read the file line by line
- Parse the first line to get a list of the column names or headers
- Split each remaining line and convert each value into a float
- Create a dictionary for each loan using the headers as keys
- Create a list of dictionaries to keep track of all the loans

Since we will perform the same operations for multiple files, it would be useful to define a function `read_csv`. We'll also define some helper functions to build up the functionality step by step.

Let's start by defining a function `parse_header` that takes a line as input and returns a list of column headers.

```
def parse_headers(header_line):
    return header_line.strip().split(',')
```

The `strip` method removes any extra spaces and the newline character `\n`. The `split` method breaks a string into a list using the given separator (`,` in this case).

```
file3_lines[0]
```

```
'amount,duration,rate,down_payment\n'
```

```
headers = parse_headers(file3_lines[0])
```

```
headers
```

```
['amount', 'duration', 'rate', 'down_payment']
```

Next, let's define a function `parse_values` that takes a line containing some data and returns a list of floating-point numbers.

```
def parse_values(data_line):
    values = []
    for item in data_line.strip().split(','):
        values.append(float(item))
    return values
```

```
file3_lines[1]
```

```
'45230,48,0.07,4300\n'
```

```
parse_values(file3_lines[1])
```

```
[45230.0, 48.0, 0.07, 4300.0]
```

The values were parsed and converted to floating point numbers, as expected. Let's try it for another line from the file, which does not contain a value for the down payment.

```
file3_lines[2]
```

```
'883000,16,0.14,\n'
```

```
parse_values(file3_lines[2])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-32-b170e8564609> in <module>
----> 1 parse_values(file3_lines[2])

<ipython-input-28-8be4024619c8> in parse_values(data_line)
      2     values = []
      3     for item in data_line.strip().split(','):
----> 4         values.append(float(item))
      5     return values

ValueError: could not convert string to float:
```

The code above leads to a `ValueError` because the empty string `''` cannot be converted to a float. We can enhance the `parse_values` function to handle this *edge case*. We will also handle the case where the value is not a float.

```
def parse_values(data_line):
    values = []
    for item in data_line.strip().split(','):
        if item == '':
            values.append(0.0)
        else:
            try:
                values.append(float(item))
            except ValueError:
                values.append(item)
    return values
```

```
file3_lines[2]
```

```
'883000,16,0.14,\n'
```

```
parse_values(file3_lines[2])
```

```
[883000.0, 16.0, 0.14, 0.0]
```

Next, let's define a function `create_item_dict` that takes a list of values and a list of headers as inputs and returns a dictionary with the values associated with their respective headers as keys.

```
def create_item_dict(values, headers):
    result = {}
    for value, header in zip(values, headers):
        result[header] = value
    return result
```

Can you figure out what the Python built-in function `zip` does? Try out an example, or read the documentation.

```
for item in zip([1,2,3], ['a', 'b', 'c']):
    print(item)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
```

Let's try out `create_item_dict` with a couple of examples.

```
file3_lines[1]
```

```
'45230,48,0.07,4300\n'
```

```
values1 = parse_values(file3_lines[1])
create_item_dict(values1, headers)
```

```
{'amount': 45230.0, 'duration': 48.0, 'rate': 0.07, 'down_payment': 4300.0}
```

```
file3_lines[2]
```

```
'883000,16,0.14,\n'
```

```
values2 = parse_values(file3_lines[2])
create_item_dict(values2, headers)
```

```
{'amount': 883000.0, 'duration': 16.0, 'rate': 0.14, 'down_payment': 0.0}
```

As expected, the values & header are combined to create a dictionary with the appropriate key-value pairs.

We are now ready to put it all together and define the `read_csv` function.

```
def read_csv(path):
    result = []
    # Open the file in read mode
    with open(path, 'r') as f:
        # Get a list of lines
        lines = f.readlines()
        # Parse the header
        headers = parse_headers(lines[0])
        # Loop over the remaining lines
        for data_line in lines[1:]:
            # Parse the values
```

```
            values = parse_values(data_line)
            # Create a dictionary using values & headers
            item_dict = create_item_dict(values, headers)
            # Add the dictionary to the result
            result.append(item_dict)
    return result
```

Let's try it out!

```
with open('./data/loans2.txt') as file2:
    print(file2.read())
```

```
amount,duration,rate,down_payment
828400,120,0.11,100000
4633400,240,0.06,
42900,90,0.08,8900
983000,16,0.14,
15230,48,0.07,4300
```

```
read_csv('./data/loans2.txt')
```

```
[{'amount': 828400.0,
  'duration': 120.0,
  'rate': 0.11,
  'down_payment': 100000.0},
 {'amount': 4633400.0, 'duration': 240.0, 'rate': 0.06, 'down_payment': 0.0},
 {'amount': 42900.0, 'duration': 90.0, 'rate': 0.08, 'down_payment': 8900.0},
 {'amount': 983000.0, 'duration': 16.0, 'rate': 0.14, 'down_payment': 0.0},
 {'amount': 15230.0, 'duration': 48.0, 'rate': 0.07, 'down_payment': 4300.0}]
```

The file is read and converted to a list of dictionaries, as expected. The `read_csv` file is generic enough that it can parse any file in the CSV format, with any number of rows or columns. Here's the full code for `read_csv` along with the helper functions:

```
def parse_headers(header_line):
    return header_line.strip().split(',')

def parse_values(data_line):
    values = []
    for item in data_line.strip().split(','):
        if item == '':
            values.append(0.0)
        else:
            try:
                values.append(float(item))
            except ValueError:
                values.append(item)
    return values
```

```
def create_item_dict(values, headers):
    result = {}
    for value, header in zip(values, headers):
        result[header] = value
    return result

def read_csv(path):
    result = []
    # Open the file in read mode
    with open(path, 'r') as f:
        # Get a list of lines
        lines = f.readlines()
        # Parse the header
        headers = parse_headers(lines[0])
        # Loop over the remaining lines
        for data_line in lines[1:]:
            # Parse the values
            values = parse_values(data_line)
            # Create a dictionary using values & headers
            item_dict = create_item_dict(values, headers)
            # Add the dictionary to the result
            result.append(item_dict)
    return result
```

creating an empty dictionary

Try to create small, generic, and reusable functions whenever possible. They will likely be useful beyond just the problem at hand and save you significant effort in the future.

In the previous tutorial, we defined a function to calculate the equal monthly installments for a loan. Here's what it looked like:

```
import math

def loan_emi(amount, duration, rate, down_payment=0):
    """Calculates the equal montly installment (EMI) for a loan.

    Arguments:
        amount - Total amount to be spent (loan + down payment)
        duration - Duration of the loan (in months)
        rate - Rate of interest (monthly)
        down_payment (optional) - Optional intial payment (deducted from amount)
    """
    loan_amount = amount - down_payment
    try:
        emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    except ZeroDivisionError:
        emi = loan_amount / duration
    emi = math.ceil(emi)
    return emi
```

We can use this function to calculate EMIs for all the loans in a file.

```
loans2 = read_csv('./data/loans2.txt')
```

```
loans2
```

```
[{'amount': 828400.0,
  'duration': 120.0,
  'rate': 0.11,
  'down_payment': 100000.0},
 {'amount': 4633400.0, 'duration': 240.0, 'rate': 0.06, 'down_payment': 0.0},
 {'amount': 42900.0, 'duration': 90.0, 'rate': 0.08, 'down_payment': 8900.0},
 {'amount': 983000.0, 'duration': 16.0, 'rate': 0.14, 'down_payment': 0.0},
 {'amount': 15230.0, 'duration': 48.0, 'rate': 0.07, 'down_payment': 4300.0}]
```

```
for loan in loans2:
    loan['emi'] = loan_emi(loan['amount'],
                           loan['duration'],
                           loan['rate']/12, # the CSV contains yearly rates
                           loan['down_payment'])
```

```
loans2
```

```
[{'amount': 828400.0,
  'duration': 120.0,
  'rate': 0.11,
  'down_payment': 100000.0,
  'emi': 10034},
 {'amount': 4633400.0,
  'duration': 240.0,
  'rate': 0.06,
  'down_payment': 0.0,
  'emi': 33196},
 {'amount': 42900.0,
  'duration': 90.0,
  'rate': 0.08,
  'down_payment': 8900.0,
  'emi': 504},
 {'amount': 983000.0,
  'duration': 16.0,
  'rate': 0.14,
  'down_payment': 0.0,
  'emi': 67707},
 {'amount': 15230.0,
  'duration': 48.0,
  'rate': 0.07,
  'down_payment': 4300.0,
  'emi': 262}]
```

You can see that each loan now has a new key `emi`, which provides the EMI for the loan. We can extract this logic into a function so that we can use it for other files too.

```
def compute_emis(loans):
    for loan in loans:
        loan['emi'] = loan_emi(
            loan['amount'],
            loan['duration'],
            loan['rate']/12, # the CSV contains yearly rates
            loan['down_payment'])
```

Writing to files

Now that we have performed some processing on the data, it would be good to write the results back to a CSV file. We can create/open a file in `w` mode using `open` and write to it using the `.write` method. The string `format` method will come in handy here.

```
loans2 = read_csv('./data/loans2.txt')
```

```
compute_emis(loans2)
```

```
loans2
```

```
[{'amount': 828400.0,
  'duration': 120.0,
  'rate': 0.11,
  'down_payment': 100000.0,
  'emi': 10034},
 {'amount': 4633400.0,
  'duration': 240.0,
  'rate': 0.06,
  'down_payment': 0.0,
  'emi': 33196},
 {'amount': 42900.0,
  'duration': 90.0,
  'rate': 0.08,
  'down_payment': 8900.0,
  'emi': 504},
 {'amount': 983000.0,
  'duration': 16.0,
  'rate': 0.14,
  'down_payment': 0.0,
  'emi': 67707},
 {'amount': 15230.0,
  'duration': 48.0,
  'rate': 0.07,
  'down_payment': 4300.0,
  'emi': 262}]
```

```
with open('./data/emis2.txt', 'w') as f:
    for loan in loans2:
        f.write('{},{},{},{},{}\n'.format(
            loan['amount'],
            loan['duration'],
            loan['rate'],
            loan['down_payment'],
            loan['emi']))
```

Let's verify that the file was created and written to as expected.

```
os.listdir('data')
```

```
['loans2.txt', 'loans3.txt', 'loans1.txt', 'emis2.txt']
```

```
with open('./data/emis2.txt', 'r') as f:
    print(f.read())
```

```
828400.0,120.0,0.11,100000.0,10034
4633400.0,240.0,0.06,0.0,33196
42900.0,90.0,0.08,8900.0,504
983000.0,16.0,0.14,0.0,67707
15230.0,48.0,0.07,4300.0,262
```

Great, looks like the loan details (along with the computed EMIs) were written into the file.

Let's define a generic function `write_csv` which takes a list of dictionaries and writes it to a file in CSV format. We will also include the column headers in the first line.

```
def write_csv(items, path):
    # Open the file in write mode
    with open(path, 'w') as f:
        # Return if there's nothing to write
        if len(items) == 0:
            return

        # Write the headers in the first line
        headers = list(items[0].keys())
        f.write(','.join(headers) + '\n')

        # Write one item per line
        for item in items:
            values = []
            for header in headers:
                values.append(str(item.get(header, "")))
            f.write(','.join(values) + "\n")
```

Do you understand how the function works? If now, try executing each statement by line by line or a different cell to figure out how it works.

Let's try it out!

```
loans3 = read_csv('./data/loans3.txt')
```

```
compute_emis(loans3)
```

```
write_csv(loans3, './data/emis3.txt')
```

```
with open('./data/emis3.txt', 'r') as f:
    print(f.read())
```

```
amount,duration,rate,down_payment,emi
45230.0,48.0,0.07,4300.0,981
883000.0,16.0,0.14,0.0,60819
100000.0,12.0,0.1,0.0,8792
728400.0,120.0,0.12,100000.0,9016
3637400.0,240.0,0.06,0.0,26060
82900.0,90.0,0.07,8900.0,1060
316000.0,16.0,0.13,0.0,21618
15230.0,48.0,0.08,4300.0,267
991360.0,99.0,0.08,0.0,13712
323000.0,27.0,0.09,4720010000.0,-193751447
528400.0,120.0,0.11,100000.0,5902
8633400.0,240.0,0.06,0.0,61853
12900.0,90.0,0.08,8900.0,60
```

With just four lines of code, we can now read each downloaded file, calculate the EMIs, and write the results back to new files:

```
for i in range(1,4):
    loans = read_csv('./data/loans{}.txt'.format(i))
    compute_emis(loans)
    write_csv(loans, './data/emis{}.txt'.format(i))
```

```
os.listdir('./data')
```

```
['loans2.txt',
 'loans3.txt',
 'loans1.txt',
 'emis3.txt',
 'emis2.txt',
 'emis1.txt']
```

Isn't that wonderful? Once all the functions are defined, we can calculate EMIs for thousands or even millions of loans across many files in seconds with just a few lines of code. Now we're starting to see the real power of using a programming language like Python for processing data!

Using Pandas to Read and Write CSVs

There are some limitations to the `read_csv` and `write_csv` functions we've defined above:

- The `read_csv` function fails to create a proper dictionary if any of the values in the CSV files contains commas
- The `write_csv` function fails to create a proper CSV if any of the values to be written contains commas

When a value in a CSV file contains a comma (`,`), the value is generally placed within double quotes. Double quotes (`"`) in values are converted into two double quotes (`""`). Here's an example:

```
title,description
Fast & Furious,"A movie, a race, a franchise"
The Dark Knight,"Gotham, the ""Batman"", and the Joker"
Memento,A guy forgets everything every 15 minutes
```

Let's try it out.

```
movies_url = "https://gist.githubusercontent.com/aakashns/afee0a407d44bbc02321993548021
```

```
urlretrieve(movies_url, 'data/movies.csv')
```

```
('data/movies.csv', <http.client.HTTPMessage at 0x7fe15cf813c8>)
```

```
movies = read_csv('data/movies.csv')
```

```
movies
```

```
[{'title': 'Fast & Furious', 'description': '"A movie'},
 {'title': 'The Dark Knight', 'description': '"Gotham'},
 {'title': 'Memento',
  'description': 'A guy forgets everything every 15 minutes'}]
```

As you can seen above, the movie descriptions weren't parsed properly.

To read this CSV properly, we can use the `pandas` library.

```
!pip install pandas --upgrade --quiet
```

```
import pandas as pd
```

The `pd.read_csv` function can be used to read the CSV file into a pandas data frame: a spreadsheet-like object for analyzing and processing data. We'll learn more about data frames in a future lesson.

```
movies_dataframe = pd.read_csv('data/movies.csv')
```

```
movies_dataframe
```

	title	description
0	Fast & Furious	A movie, a race, a franchise
1	The Dark Knight	Gotham, the "Batman", and the Joker
2	Memento	A guy forgets everything every 15 minutes

A dataframe can be converted into a list of dictionaries using the `to_dict` method.

```
movies = movies_dataframe.to_dict('records')
```

```
movies
```

```
[{'title': 'Fast & Furious', 'description': 'A movie, a race, a franchise'},
 {'title': 'The Dark Knight',
  'description': 'Gotham, the "Batman", and the Joker'},
 {'title': 'Memento',
  'description': 'A guy forgets everything every 15 minutes'}]
```

If you don't pass the arguments `records`, you get a dictionary of lists instead.

```
movies_dict = movies_dataframe.to_dict()
```

```
movies_dict
```

```
{'title': {0: 'Fast & Furious', 1: 'The Dark Knight', 2: 'Memento'},
 'description': {0: 'A movie, a race, a franchise',
  1: 'Gotham, the "Batman", and the Joker',
  2: 'A guy forgets everything every 15 minutes'}}
```

Let's try using the `write_csv` function to write the data in `movies` back to a CSV file.

```
write_csv(movies, 'movies2.csv')
```

```
!head movies2.csv
```

```
title,description
Fast & Furious,A movie, a race, a franchise
The Dark Knight,Gotham, the "Batman", and the Joker
Memento,A guy forgets everything every 15 minutes
```

As you can see above, the CSV file is not formatted properly. This can be verified by attempting to read the file using `pd.read_csv`.

```
pd.read_csv('movies2.csv')
```

		title	description
Fast & Furious	**A movie**	a race	a franchise
The Dark Knight	**Gotham**	the "Batman"	and the Joker
Memento	**A guy forgets everything every 15 minutes**	NaN	NaN

To convert a list of dictionaries into a dataframe, you can use the `pd.DataFrame` constructor.

```
df2 = pd.DataFrame(movies)
```

```
df2
```

	title	description
0	Fast & Furious	A movie, a race, a franchise
1	The Dark Knight	Gotham, the "Batman", and the Joker
2	Memento	A guy forgets everything every 15 minutes

It can now be written to a CSV file using the `.to_csv` method of a dataframe.

```
df2.to_csv('movies3.csv', index=None)
```

Can you guess what the argument `index=None` does? Try removing it and observing the difference in output.

```
!head movies3.csv
```

```
title,description
Fast & Furious,"A movie, a race, a franchise"
The Dark Knight,"Gotham, the ""Batman"", and the Joker"
Memento,A guy forgets everything every 15 minutes
```

The CSV file is formatted properly. We can verify this by trying to read it back.

```
pd.read_csv('movies3.csv')
```

	title	description
0	Fast & Furious	A movie, a race, a franchise
1	The Dark Knight	Gotham, the "Batman", and the Joker
2	Memento	A guy forgets everything every 15 minutes

We're able to write and read the file properly with `pandas`.

In general, it's always a better idea to use libraries like Pandas for reading and writing CSV files.

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. Jovian offers an easy way of saving and sharing your Jupyter notebooks online.

```
# Install the library
!pip install jovian --upgrade --quiet
```

```
# Import the jovian module
import jovian
```

```
jovian.commit(project='python-os-and-filesystem')
```

```
[jovian] Attempting to save notebook..
```

The first time you run `jovian.commit`, you'll be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your Jovian profile page after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Exercise - Processing CSV files using a dictionary of lists

We defined the functions `read_csv` and `write_csv` above to convert a CSV file into a list of dictionaries and vice versa. In this exercise, you'll transform the CSV data into a dictionary of lists instead, with one list for each column in the file.

For example, consider the following CSV file:

```
amount,duration,rate,down_payment
828400,120,0.11,100000
4633400,240,0.06,
42900,90,0.08,8900
983000,16,0.14,
15230,48,0.07,4300
```

We'll convert it into the following dictionary of lists:

```
{
  amount: [828400, 4633400, 42900, 983000, 15230],
  duration: []120, 240, 90, 16, 48],
  rate: [0.11, 0.06, 0.08, 0.14, 0.07],
  down_payment: [100000, 0, 8900, 0, 4300]
}
```

Complete the following tasks using the empty cells below:

1. Download three CSV files to the folder `data2` using the URLs listed in the code cell below, and verify the downloaded files.

2. Define a function `read_csv_columnar` that reads a CSV file and returns a dictionary of lists in the format shown above.
3. Define a function `compute_emis` that adds another key `emi` into the dictionary with a list of EMIs computed for each row of data.
4. Define a function `write_csv_columnar` that writes the data from the dictionary of lists into a correctly formatted CSV file.
5. Process all three downloaded files and write the results by creating new files in the directory `data2`.

Define helper functions wherever required.

```
url1 = 'https://gist.githubusercontent.com/aakashns/257f6e6c8719c17d0e498ea287d1a386/ra
url2 = 'https://gist.githubusercontent.com/aakashns/257f6e6c8719c17d0e498ea287d1a386/ra
url3 = 'https://gist.githubusercontent.com/aakashns/257f6e6c8719c17d0e498ea287d1a386/ra
```

```

```

```

```

```

```

```

```

Finally, let's save a snapshot of our work using `jovian.commit`.

```
jovian.commit(project='python-os-and-filesystem')
```

Summary and Further Reading

With this, we complete our discussion of reading from and writing to files in Python. We've covered the following topics in this tutorial:

- Interacting with the file system using the os module
- Downloading files from URLs using the `urllib` module
- Opening files using the open built-in function
- Reading the contents of a file using `.read`
- Closing a file automatically using `with`
- Reading a file line by line using `readlines`
- Processing data from a CSV file by defining functions
- Using helper functions to build more complex functions
- Writing data to a file using `.write`

This tutorial on working with files in Python is by no means exhaustive. Following are some more resources you should check out:

- Python Tutorial at W3Schools: https://www.w3schools.com/python/
- Practical Python Programming: https://dabeaz-course.github.io/practical-python/Notes/Contents.html
- Python official documentation: https://docs.python.org/3/tutorial/index.html

You are ready to move on to the next tutorial: Numerical Computing with Python and Numpy.

Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is the purpose of the os module in Python?
2. How do you identify the current working directory in a Jupyter notebook?
3. How do you retrieve the list of files within a directory using Python?
4. How do you create a directory using Python?
5. How do you check whether a file or directory exists on the filesystem? Hint: `os.path.exists`.
6. Where can you find the full list of functions contained in the os module?
7. Give examples of 5 useful functions from the os and `os.path` modules.
8. How do you download a file from a URL using Python?
9. How do you open a file using Python? Give an example?
10. What are the different modes for opening a file in Python?
11. Can you open a file in multiple modes? Illustrate with an example.
12. What is the file object? How is it useful?
13. How do you read the contents of a file into a string?
14. What is a CSV file? Give an example.
15. How do you close an open file?
16. Why is it essential to close a file after processing it?
17. How do you ensure that files are closed automatically after processing? Give an example.
18. How is the `with` statement useful for working with files?
19. What happens if you try to read from a closed file?
20. How do you read the contents of a file line by line?
21. Write a function to convert the contents of a CSV file into a list of dictionaries (one dictionary for each row of the file).
22. Write a function to convert the contents of a CSV file into a dictionary of lists (one dictionary for each column of the file).
23. How do you write to a file using Python?
24. How is the string `.format` method for writing data to a file in CSV format?
25. Write a function to write data from a list of dictionaries into a CSV file.
26. Write a function to write data from a dictionary of lists into a CSV file.

27. Where can you learn about the methods supported by the file object in Python?

28. How can you read from and write to CSV files using Pandas?