```
%matplotlib inline
import matplotlib
import seaborn as sns
matplotlib.rcParams['savefig.dpi'] = 144
```

```
import expectexception
```

Importing and Exporting Data

So far we've only dealt with data that we have created within Python. Generating random data is helpful for testing out ideas, but we want to work with real data. Most often that data will be stored in a file, either locally on the computer or online. In this notebook we'll learn how to read and write data to files.

Python file handles (open)

In Python we interact with files on disk using the commands `open` and `close`. We've included a file in the `data` folder called `sample.txt`. Let's open it and read its contents

```
f = open('./data/sample.txt', 'r')
data = f.read()
f.close()

print(data)
print(f)
```

```
Hello!
Congratulations!
You've read in data from a file.
<_io.TextIOWrapper name='./data/sample.txt' mode='r' encoding='UTF-8'>
```

Notice that we `open` the file and assign it to `f`, `read` the data from `f`, and then close `f`. What is `f`? It's called a **file handle**. It's an object that connects Python to the file we `open`. We `read` the data using this connection, and then once we're done with `close` the connection. It's a good habit to `close` a file handle once we're done with it, so usually we will do it automatically using Python's `with` keyword.

```
# f is automatically closed
# at the end of the body of the with statement
with open('./data/sample.txt', 'r') as f:
    print(f.read())

print(f)
```

```
Hello!
Congratulations!
You've read in data from a file.
<_io.TextIOWrapper name='./data/sample.txt' mode='r' encoding='UTF-8'>
```

We can also read individual lines of a file.

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

```
Hello!
```

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

```
['Hello!\n', 'Congratulations!\n', "You've read in data from a file."]
```

Writing data to files is very similar. The main difference is when we open the file, we will use the `'w'` flag instead of `'r'`.

```
with open('./data/my_data.txt', 'w') as f:
    f.write('This is a new file.')
    f.write('I am practicing writing data to disk.')

with open('./data/my_data.txt', 'r') as f:
    my_data = f.read()

print(my_data)
```

```
This is a new file.I am practicing writing data to disk.
```

No matter how often I execute the above cell, the same output gets printed. Opening the file with the `'w'` flag will overwrite the contents of the file. If we want to add to what is already in the file, we have to open the file with the `'a'` flag (`'a'` stands for *append*).

```
with open('./data/my_data.txt', 'a') as f:
    f.write('\nAdding a new line to the file.')

with open('./data/my_data.txt', 'r') as f:
    my_data = f.read()

print(my_data)
```

```
This is a new file.I am practicing writing data to disk.
Adding a new line to the file.
```

We always need to be careful when writing to disk, because we could overwrite or alter data by accident. It is also easy to encounter errors when working with files, because we might not know ahead of time if the file we're trying to access exists, or we might mix up the `'r'`, `'w'`, and `'a'` flags.

```
%%expect_exception IOError

# if a file doesn't exist
# we can't open it for reading
# (but we can open it for writing)

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

```
---------------------------------------------------------------------------

ExceptionExpected                         Traceback (most recent call last)

<ipython-input-11-a2eda2783467> in <module>()
----> 1 get_ipython().run_cell_magic('expect_exception', 'IOError', "\n# if a file doesn't exist\n# we
can't open it for reading\n# (but we can open it for writing)\n\nwith open('./data/fail.txt', 'r') as
f:\n    f.read()")


/opt/conda/lib/python3.6/site-packages/IPython/core/interactiveshell.py in run_cell_magic(self,
magic_name, line, cell)
   2165             magic_arg_s = self.var_expand(line, stack_depth)
   2166             with self.builtin_trap:
-> 2167                 result = fn(magic_arg_s, cell)
   2168             return result
   2169


<decorator-gen-125> in expect_exception(self, line, cell)


/opt/conda/lib/python3.6/site-packages/IPython/core/magic.py in <lambda>(f, *a, **k)
```

```
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188 
    189         if callable(arg):


/opt/conda/lib/python3.6/site-packages/expectexception/excpectexceptionmagic.py in expect_exception(self,
line, cell)
     25     @cell_magic
     26     def expect_exception(self, line, cell):
---> 27         self.run_cell(line, cell, True)
     28 
     29     @cell_magic


/opt/conda/lib/python3.6/site-packages/expectexception/excpectexceptionmagic.py in run_cell(self, line,
cell, exception_required)
     47             result = self.shell.run_cell(cell)
     48             if exception_required and not result.error_in_exec:
---> 49                 raise ExceptionExpected("This cell did not raise the expected %s." % line)
     50         finally:
     51             self.shell.CustomTB = old_CustomTB


ExceptionExpected: This cell did not raise the expected IOError.
```

```
%%expect_exception IOError

# we can't read a file open for writing

with open('./data/fail.txt', 'w') as f:
    f.read()
```

```
□[0;31m---------------------------------------------------------------------------□[0m
□[0;31mUnsupportedOperation□[0m                     Traceback (most recent call last)
□[0;32m<ipython-input-12-2396ab194fe8>□[0m in □[0;36m<module>□[0;34m()□[0m
□[1;32m      3□[0m □[0;34m□[0m□[0m
□[1;32m      4□[0m □[0;32mwith□[0m □[0mopen□[0m□[0;34m(□[0m□[0;34m'./data/fail.txt'□[0m□[0;34m,□[0m
□[0;34m'w'□[0m□[0;34m)□[0m □[0;32mas□[0m □[0mf□[0m□[0;34m:□[0m□[0;34m□[0m□[0m
□[0;32m----> 5□[0;31m     □[0mf□[0m□[0;34m.□[0m□[0mread□[0m□[0;34m(□[0m□[0;34m)□[0m□[0;34m□[0m□[0m
□[0m
□[0;31mUnsupportedOperation□[0m: not readable
```

```
%%expect_exception IOError

# and we can't write to a file open for reading

with open('./data/sample.txt', 'r') as f:
    f.write('This will fail')
```

```
□[0;31m---------------------------------------------------------------------------□[0m
□[0;31mUnsupportedOperation□[0m                     Traceback (most recent call last)
□[0;32m<ipython-input-13-415e83617ea3>□[0m in □[0;36m<module>□[0;34m()□[0m
□[1;32m      3□[0m □[0;34m□[0m□[0m
□[1;32m      4□[0m □[0;32mwith□[0m □[0mopen□[0m□[0;34m(□[0m□[0;34m'./data/sample.txt'□[0m□[0;34m,□[0m
□[0;34m'r'□[0m□[0;34m)□[0m □[0;32mas□[0m □[0mf□[0m□[0;34m:□[0m□[0;34m□[0m□[0m
□[0;32m----> 5□[0;31m     □[0mf□[0m□[0;34m.□[0m□[0mwrite□[0m□[0;34m(□[0m□[0;34m'This will
fail'□[0m□[0;34m)□[0m□[0;34m□[0m□[0m
□[0m
□[0;31mUnsupportedOperation□[0m: not writable
```

Can we prevent some of these errors? How do we find out what files are on disk?

`os` module

Python has a module for navigating the computer's file system called `os`. There are many useful tools in the `os` module, but there are two functions that are most useful for finding files.

```
import os

# list the contents of the current directory
# ('.' refers to the current directory)
os.listdir('.')
```

```
['.done',
 'DS_IO.ipynb',
 'DS_Basic_DS_Modules.ipynb',
 'data',
 'DS_Data_Munging.ipynb',
 'DS_Intro_Statistics.ipynb',
 'miniprojects',
 'DS_Pandas.ipynb',
 'DS_SQL.ipynb',
 '.ipynb_checkpoints',
 'DS_Classes_and_ORM.ipynb']
```

The command `listdir` is the simpler of the two functions we'll cover. It simply lists the contents of the directory path we specify. When we pass `'.'` as the argument, `listdir` will look in the current directory. It lists all the Jupyter notebooks we're using for the course, as well as the `data` subdirectory. We could find out what's in the `data` subdirectory by looking in `'./data'`.

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

```
['customers.csv',
 'factbook.csv',
 'bad_csv.csv',
 'eog_djvu.txt',
 'my_data.txt',
 'products.csv',
 'short_text.txt',
 'eog_djvu_scrambled.txt.gz',
 'pd_write.csv',
 'eog_djvu.txt.gz',
 'sample.txt',
 'yelp.json.gz',
 'csv_sample.txt',
 'library.json',
 'PEP_2016_PEPANNRES.csv',
 'fail.txt',
 'orders.csv',
 'short_text.txt.gz',
 'eog_djvu.txt.1',
 'example_df.json']
```

What if we wanted to find all the files and subdirectories below a directory somewhere on our computer? With `listdir` we only see the files and subdirectories under the particular directory we're looking in. We cannot use `listdir` to automatically search through subdirectories. For this we need to use `walk`, which "walks" through all the subdirectories below our chosen directory. We won't cover `walk` in this course, but it's one of the very useful tools (along with the `os.path` sub-module) for working with files in Python, particularly if you are working with many different data files at once.

CSV files

One of the simplest and most common formats for saving data is as comma-separated values (CSV).

```
with open('./data/csv_sample.txt', 'r') as f:
    csv = f.read()

print(csv)
```

```
index,name,age
0,Dylan,28
1,Terrence,54
2,Mya,31
```

This format is often used to represent tables of data. Usually a CSV will have rows (separated by newline characters, `'\n'`) and columns (separated by commas). Otherwise they are no different from any other text file. We can use the special formatting of a CSV to create a list of lists representing the table.

```
list_table = []
with open('./data/csv_sample.txt', 'r') as f:
    for line in f.readlines():
        list_table.append(line.strip().split(','))

list_table
```

```
[['index', 'name', 'age'],
 ['0', 'Dylan', '28'],
 ['1', 'Terrence', '54'],
 ['2', 'Mya', '31']]
```

However, we can work with tabular data much more easily in a Pandas DataFrame. Pandas provides a `read_csv` method to read the data directly into a DataFrame.

```
import pandas as pd

df = pd.read_csv('./data/csv_sample.txt', index_col=0)
df
```

```
.dataframe thead th {
    text-align: left;
}

.dataframe tbody tr th {
    vertical-align: top;
}
```

	name	age
index		
0	Dylan	28
1	Terrence	54
2	Mya	31

The `read_csv` method is very flexible to deal with the formatting of different data sets. Some data sets will include column headers while others may not. Some data sets will include an index while others may not. Some data sets may have values separated by tabs, semicolons, or other characters instead of commas. There are options in the `read_csv` method for dealing with all of these. You can read about them in the Pandas documentation on `read_csv`. We'll also discuss it further in the Pandas notebook.

```
# an example of downloading
# and importing real data using `read_csv`

if 'factbook.csv' not in os.listdir('./data/'):
    !wget -P ./data/ https://perso.telecom-paristech.fr/eagan/class/igr204/data/factbook.csv

countries = pd.read_csv('./data/factbook.csv', delimiter=';', skiprows=[1])
countries.head()
```

```
.dataframe thead th {
    text-align: left;
}

.dataframe tbody tr th {
    vertical-align: top;
}
```

	Country	Area(sq km)	Birth rate(births/1000 population)	Current account balance	Death rate(deaths/1000 population)	Debt - external	Electricity - consumption(kWh)	Electricity - production(kWh)
0	Afghanistan	647500	47.02	NaN	20.75	8.000000e+09	6.522000e+08	5.400000e+08
1	Akrotiri	123	NaN	NaN	NaN	NaN	NaN	NaN
2	Albania	28748	15.08	-5.040000e+08	5.12	1.410000e+09	6.760000e+09	5.680000e+09
3	Algeria	2381740	17.13	1.190000e+10	4.60	2.190000e+10	2.361000e+10	2.576000e+10
4	American Samoa	199	23.13	NaN	3.33	NaN	1.209000e+08	1.300000e+08

5 rows × 45 columns

```
# we can also use pandas to write CSV
# using the DataFrame's to_csv method

pd.DataFrame({'a': [0, 3, 10], 'b': [True, True, False]}).to_csv('./data/pd_write.csv')

pd.read_csv('./data/pd_write.csv', index_col=0)
```

```
.dataframe thead th {
    text-align: left;
}

.dataframe tbody tr th {
    vertical-align: top;
}
```

	a	b
0	0	True
1	3	True
2	10	False

Sometimes a CSV won't be perfect. For example, maybe different rows have different numbers of commas. This makes it difficult to interpret the contents of the file as a table.

```
# the 3rd line only has 2 "columns"

!cat ./data/bad_csv.csv
```

```
index,name,age
0,Dylan,27
1,54
2,Mya,31
```

```
# what happens if we try to read this
# into a DataFrame using read_csv?

pd.read_csv('./data/bad_csv.csv', index_col = 0)
```

```
.dataframe thead th {
    text-align: left;
}

.dataframe tbody tr th {
    vertical-align: top;
}
```

	name	age
index		
0	Dylan	27.0
1	54	NaN
2	Mya	31.0

Pandas' `read_csv` method will do its best to construct a table out of a poorly formatted CSV, but it may make mistakes. For example, 54 was interpreted as a name instead of as an age, because there were only 2 columns in that line of the file. Data sets will often contain mistakes like bad formatting, missing data, or typos.

Question: How could we fix the badly formatted CSV so it would work with `read_csv`?

JSON

JSON stands for JavaScript Object Notation. JavaScript is a common language for creating web applications, and JSON files are used to collect and transmit information between JavaScript applications. As a result, a lot of data on the internet exists in the JSON file format. For example, Twitter and Google Maps use JSON.

A JSON file is essentially a data structure built out of nested dictionaries and lists. Let's make our own example and then we'll examine an example downloaded from the internet.

```
book1 = {'title': 'The Prophet',
         'author': 'Khalil Gibran',
         'genre': 'poetry',
         'tags': ['religion', 'spirituality', 'philosophy', 'Lebanon', 'Arabic', 'Middle East'],
         'book_id': '811.19',
         'copies': [{'edition_year': 1996,
                     'checkouts': 486,
                     'borrowed': False},
                    {'edition_year': 1996,
                     'checkouts': 443,
                     'borrowed': False}]
         }

book2 = {'title': 'The Little Prince',
         'author': 'Antoine de Saint-Exupery',
         'genre': 'children',
         'tags': ['fantasy', 'France', 'philosophy', 'illustrated', 'fable'],
         'id': '843.912',
         'copies': [{'edition_year': 1983,
                     'checkouts': 634,
                     'borrowed': True,
                     'due_date': '2017/02/02'},
                    {'edition_year': 2015,
                     'checkouts': 41,
                     'borrowed': False}]
         }

library = [book1, book2]
library
```

```
[{'title': 'The Prophet',
  'author': 'Khalil Gibran',
  'genre': 'poetry',
  'tags': ['religion',
   'spirituality',
   'philosophy',
   'Lebanon',
   'Arabic',
   'Middle East'],
  'book_id': '811.19',
  'copies': [{'edition_year': 1996, 'checkouts': 486, 'borrowed': False},
   {'edition_year': 1996, 'checkouts': 443, 'borrowed': False}]},
 {'title': 'The Little Prince',
  'author': 'Antoine de Saint-Exupery',
  'genre': 'children',
  'tags': ['fantasy', 'France', 'philosophy', 'illustrated', 'fable'],
  'id': '843.912',
  'copies': [{'edition_year': 1983,
    'checkouts': 634,
    'borrowed': True,
```

```
     'due_date': '2017/02/02'},
    {'edition_year': 2015, 'checkouts': 41, 'borrowed': False}]}]
```

We have two books in our `library`. Both books have some common properties: a title, an author, an id, and tags. Each book can have several tags, so we store that data as a list. Additionally, there can be multiple copies of each book, and each copy also has some unique information like the year it was printed and how many times it's been checked out. Notice that if a book is checked out, it also has a due date. It's convenient to store the information about the multiple copies as a list of dictionaries within the dictionary about the book, because every copy shares the same title, author, etc.

This structure is typical of JSON files. It has the advantage of reducing redundancy of data. We only store the author and title once, even though there are multiple copies of the book. Also, we don't store a due date for copies that aren't checked out.

If we were to put this data in a table, we would have to duplicate a lot of information. Also, since only one copy in our library is checked out, we also have a column with a lot of missing data.

|index|title|author|id|genre|tags|edition_year|checkouts|borrowed|due_date| |:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:| |0|The Prophet|Khalil Gibran|811.19|poetry|religion, spirituality, philosophy, Lebanon, Arabic, Middle East|1996|486|False|Null| |1|The Prophet|Khalil Gibran|811.19|poetry|religion, spirituality, philosophy, Lebanon, Arabic, Middle East|1996|443|False|Null| |2|The Little Prince|Antoine de Saint-Exupery|843.912|children|fantasy, France, philosophy, illustrated, fable|1983|634|True|2017/02/02| |3|The Little Prince|Antoine de Saint-Exupery|843.912|children|fantasy, France, philosophy, illustrated, fable|2015|41|False|Null|

This is very wasteful. Since JSON files are meant to be shared quickly over the internet, it is important that they are small to reduce the amount of resources needed to store and transmit them.

We can write our `library` to disk using the `json` module.

```
import json

with open('./data/library.json', 'w') as f:
    json.dump(library, f, indent=2)
```

```
!cat ./data/library.json
```

```
[
  {
    "title": "The Prophet",
    "author": "Khalil Gibran",
    "genre": "poetry",
    "tags": [
      "religion",
      "spirituality",
      "philosophy",
      "Lebanon",
      "Arabic",
      "Middle East"
    ],
    "book_id": "811.19",
    "copies": [
      {
        "edition_year": 1996,
        "checkouts": 486,
        "borrowed": false
      },
      {
        "edition_year": 1996,
        "checkouts": 443,
        "borrowed": false
      }
    ]
  },
  {
    "title": "The Little Prince",
    "author": "Antoine de Saint-Exupery",
    "genre": "children",
    "tags": [
      "fantasy",
      "France",
      "philosophy",
      "illustrated",
      "fable"
    ],
```

```
    "id": "843.912",
    "copies": [
      {
        "edition_year": 1983,
        "checkouts": 634,
        "borrowed": true,
        "due_date": "2017/02/02"
      },
      {
        "edition_year": 2015,
        "checkouts": 41,
        "borrowed": false
      }
    ]
  }
]
```

```
with open('./data/library.json', 'r') as f:
    reloaded_library = json.load(f)

reloaded_library
```

```
[{'title': 'The Prophet',
  'author': 'Khalil Gibran',
  'genre': 'poetry',
  'tags': ['religion',
   'spirituality',
   'philosophy',
   'Lebanon',
   'Arabic',
   'Middle East'],
  'book_id': '811.19',
  'copies': [{'edition_year': 1996, 'checkouts': 486, 'borrowed': False},
   {'edition_year': 1996, 'checkouts': 443, 'borrowed': False}]},
 {'title': 'The Little Prince',
  'author': 'Antoine de Saint-Exupery',
  'genre': 'children',
  'tags': ['fantasy', 'France', 'philosophy', 'illustrated', 'fable'],
  'id': '843.912',
  'copies': [{'edition_year': 1983,
    'checkouts': 634,
    'borrowed': True,
    'due_date': '2017/02/02'},
   {'edition_year': 2015, 'checkouts': 41, 'borrowed': False}]}]
```

```
# note that if we loaded it in without JSON
# the file would be interpreted as plain text

with open('./data/library.json', 'r') as f:
    library_string = f.read()

# this isn't what we want
library_string
```

```
'[\n  {\n    "title": "The Prophet",\n    "author": "Khalil Gibran",\n    "genre": "poetry",\n    "tags":
[\n      "religion",\n      "spirituality",\n      "philosophy",\n      "Lebanon",\n      "Arabic",\n
"Middle East"\n    ],\n    "book_id": "811.19",\n    "copies": [\n      {\n        "edition_year":
1996,\n        "checkouts": 486,\n        "borrowed": false\n      },\n      {\n        "edition_year":
1996,\n        "checkouts": 443,\n        "borrowed": false\n      }\n    ]\n  },\n  {\n    "title": "The
Little Prince",\n    "author": "Antoine de Saint-Exupery",\n    "genre": "children",\n    "tags": [\n
"fantasy",\n      "France",\n      "philosophy",\n      "illustrated",\n      "fable"\n    ],\n    "id":
"843.912",\n    "copies": [\n      {\n        "edition_year": 1983,\n        "checkouts": 634,\n
"borrowed": true,\n        "due_date": "2017/02/02"\n      },\n      {\n        "edition_year": 2015,\n
"checkouts": 41,\n        "borrowed": false\n      }\n    ]\n  }\n]'
```

```
# Pandas can also read_json
# notice how it constructs the table
```

```
# does it represent the data well?

pd.read_json('./data/library.json')
```

```
.dataframe thead th {
    text-align: left;
}

.dataframe tbody tr th {
    vertical-align: top;
}
```

	author	book_id	copies	genre	id	tags	title
0	Khalil Gibran	811.19	[{'edition_year': 1996, 'checkouts': 486, 'bor...	poetry	NaN	[religion, spirituality, philosophy, Lebanon, ...	The Prophet
1	Antoine de Saint-Exupery	NaN	[{'edition_year': 1983, 'checkouts': 634, 'bor...	children	843.912	[fantasy, France, philosophy, illustrated, fable]	The Little Prince

```
# and to_json
df.to_json('./data/example_df.json')

!head ./data/example_df.json
```

```
{"name":{"0":"Dylan","1":"Terrence","2":"Mya"},"age":{"0":28,"1":54,"2":31}}
```

We can download JSON files many ways. Sometimes we will download it manually, but we can also use `wget` like we did for the CSV example. Often we'll connect to a website's API which will respond using JSON.

Panda's `read_json` method is capable of connecting directly to a URL (whether it's the address of a JSON file or an API connection) and reading the JSON without saving the file to our computer.

```
pd.read_json('https://api.github.com/repos/pydata/pandas/issues?per_page=5')
```

```
.dataframe thead th {
    text-align: left;
}

.dataframe tbody tr th {
    vertical-align: top;
}
```

	assignee	assignees	author_association	body	closed_at	comments	comments_u
0	NaN	[]	MEMBER	- [x] closes #27011\r\n- [x] tests added / pas...	NaT	0	https://api.github.com/repos/panda dev/pandas...
1	NaN	[]	MEMBER	Continuing with the theme of avoiding runtime ...	NaT	0	https://api.github.com/repos/panda dev/pandas...
2	NaN	[]	NONE	I'm sure this has been asked before but I can'...	NaT	2	https://api.github.com/repos/panda dev/pandas...
3	NaN	[]	NONE	#### Code Sample\r\n\r\n```python\r\nimport pa...	NaT	1	https://api.github.com/repos/panda dev/pandas...

	assignee	assignees	author_association	body	closed_at	comments	comments_u
4	NaN	[]	NONE	closes #27534\r\n	NaT	3	https://api.github.com/repos/panda dev/pandas...

5 rows × 24 columns

Compressed files (Gzip)

Another way we save storage and network resources is by using **compression**. Many times data sets will contain patterns that can be used to reduce the amount of space needed to store the information.

A simple example is the following list of numbers: 10, 10, 10, 2, 3, 3, 3, 3, 3, 50, 50, 1, 1, 50, 10, 10, 10, 10

Rather than writing out the full list of numbers (18 integers), we can represent the same information with only 14 numbers: (3, 10), (1, 2), (5, 3), (2, 50), (2, 1), (1, 50), (4, 10)

Here the first number in each pair is the number of repetitions, and the second number in the pair is the actual value. We've successfully reduced the amount of numbers we need to represent the same data. Most forms of compression use a similar idea, although actual implementations are usually more complex.

In the world of data science, the most common compression is Gzip (which uses the deflate algorithm). Gzip files end with the extension `.gz`.

```
!wget -P ./data/ https://archive.org/stream/TheEpicofGilgamesh_201606/eog_djvu.txt
```

```
--2019-07-24 07:08:13--  https://archive.org/stream/TheEpicofGilgamesh_201606/eog_djvu.txt
Resolving archive.org (archive.org)... 207.241.224.2
Connecting to archive.org (archive.org)|207.241.224.2|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: ‘./data/eog_djvu.txt.2’

eog_djvu.txt.2          [ <=>                ] 166.90K  --.-KB/s    in 0.1s

2019-07-24 07:08:13 (1.28 MB/s) - ‘./data/eog_djvu.txt.2’ saved [170904]
```

```
import gzip

with open('./data/eog_djvu.txt', 'r') as f:
    text = f.read()

with gzip.open('./data/eog_djvu.txt.gz', 'wb') as f:
    f.write(text.encode('utf-8'))

!ls -lh ./data/eog*
```

```
-rw-rw-r-- 1 jovyan users 160K Jul 23 13:52 ./data/eog_djvu.txt
-rw-rw-r-- 1 jovyan users 160K Jul 23 13:57 ./data/eog_djvu.txt.1
-rw-r--r-- 1 jovyan users 167K Jul 24 07:08 ./data/eog_djvu.txt.2
-rw-rw-r-- 1 jovyan users  46K Jul 24 07:08 ./data/eog_djvu.txt.gz
-rw-rw-r-- 1 jovyan users 136K Jul 23 13:58 ./data/eog_djvu_scrambled.txt.gz
```

We were able to compress the text of The Epic of Gilgamesh to a third of its original size! Remember that compression depends on patterns in the data. Language has a lot of patterns, but what would happen if we scrambled all the letters in the text?

```
import numpy as np

with gzip.open('./data/eog_djvu_scrambled.txt.gz', 'wb') as f:
    f.write(np.random.permutation(list(text)))

!ls -lh ./data/eog*
```

```
-rw-rw-r-- 1 jovyan users 160K Jul 23 13:52 ./data/eog_djvu.txt
-rw-rw-r-- 1 jovyan users 160K Jul 23 13:57 ./data/eog_djvu.txt.1
-rw-r--r-- 1 jovyan users 167K Jul 24 07:08 ./data/eog_djvu.txt.2
-rw-rw-r-- 1 jovyan users  46K Jul 24 07:08 ./data/eog_djvu.txt.gz
-rw-rw-r-- 1 jovyan users 136K Jul 24 07:08 ./data/eog_djvu_scrambled.txt.gz
```

The scrambled version only compressed to two-thirds the size of the original. Compression won't perform very well on random data. Compression also doesn't work very well on data that is already small.

```
short_text = 'Hello'

with open('./data/short_text.txt', 'w') as f:
    f.write(short_text)

with gzip.open('./data/short_text.txt.gz', 'wb') as f:
    f.write(short_text.encode('utf-8'))

!ls -lh ./data/short_text*
```

```
-rw-rw-r-- 1 jovyan users  5 Jul 24 07:08 ./data/short_text.txt
-rw-rw-r-- 1 jovyan users 40 Jul 24 07:08 ./data/short_text.txt.gz
```

The compressed file is bigger than the plain text! That's because the compressed file includes a header, which takes up a small amount of extra space. Also, since the text is so short, it's not possible to use patterns to represent the text more efficiently. Therefore we usually save compression for large files.

You may have noticed that when we write Gzip files, we have been using a `'wb'` flag instead of a plain `'w'` flag. This is because Gzip is not plain text. When compressing the file we write *binary* files. The files are not readable as plain text.

```
# we have to uncompress the file
# before we can read it

!cat ./data/short_text.txt.gz
```

```
□□□□□□8]□□short_text.txt□□H□□□□□□□□□□□□□□
```

We should only use `'w'` for plain text files (which includes CSV and JSON). Using `'w'` instead of `'wb'` for Gzip files, or other files which are not plain text (e.g. images), could damage the file.

Serialization (`pickle`)

Often we will want to save our work in Python and come back to it later. However, that work might be a machine learning model or some other complex object in Python. How do we save complex Python objects? Python has a module for this purpose called `pickle`. We can use `pickle` to write a binary file that contains all the information about a Python object. Later we can load that pickle file and reconstruct the object in Python.

```
pickle_example = ['hello', {'a': 23, 'b': True}, (1, 2, 3), [['dogs', 'cats'], None]]
```

```
%%expect_exception TypeError

# we can't save this as text
with open('./data/pickle_example.txt', 'w') as f:
    f.write(pickle_example)
```

```
□[0;31m---------------------------------------------------------------------------□[0m
□[0;31mTypeError□[0m                                 Traceback (most recent call last)
□[0;32m<ipython-input-40-dc175613edd9>□[0m in □[0;36m<module>□[0;34m()□[0m
□[1;32m      2□[0m □[0;31m# we can't save this as text□[0m□[0;34m□[0m□[0;34m□[0m□[0m
□[1;32m      3□[0m □[0;32mwith□[0m
□[0mopen□[0m□[0;34m(□[0m□[0;34m'./data/pickle_example.txt'□[0m□[0;34m,□[0m □[0;34m'w'□[0m□[0;34m)□[0m
□[0;32mas□[0m □[0mf□[0m□[0;34m:□[0m□[0;34m□[0m□[0m
□[0;32m----> 4□[0;31m
□[0mf□[0m□[0;34m.□[0m□[0mwrite□[0m□[0;34m(□[0m□[0mpickle_example□[0m□[0;34m)□[0m□[0;34m□[0m□[0m
```

```
□[0m
□[0;31mTypeError□[0m: write() argument must be str, not list
```

```
import pickle

# we can save it as a pickle
with open('./data/pickle_example.pkl', 'wb') as f:
    pickle.dump(pickle_example, f)

with open('./data/pickle_example.pkl', 'rb') as f:
    reloaded_example = pickle.load(f)

reloaded_example
```

```
['hello', {'a': 23, 'b': True}, (1, 2, 3), [['dogs', 'cats'], None]]
```

```
# the reloaded example is the same as the original

reloaded_example == pickle_example
```

```
True
```

Pickle is an important tool for data scientists. Data processing and training machine learning models can take a long time, and it is useful to save checkpoints.

Pandas also has `to_pickle` and `read_pickle` methods.

NumPy file formats

NumPy also has methods for saving and loading data. They are straightforward to use. You may encounter these when working with certain machine learning libraries that require data be stored in NumPy arrays. NumPy arrays are also often used when working with image data.

```
sample_array = np.random.random((4, 4))
print(sample_array)
```

```
[[0.02573499 0.82494109 0.89756743 0.84206605]
 [0.70146385 0.1468585  0.45772617 0.23692087]
 [0.005141   0.22425271 0.29602516 0.64871444]
 [0.15156162 0.46722448 0.37752783 0.10490116]]
```

```
# to save as plain text
np.savetxt('./data/sample_array.txt', sample_array)
```

```
!cat ./data/sample_array.txt
```

```
2.573499304710569202e-02 8.249410915227861629e-01 8.975674256604490031e-01 8.420660467417920847e-01
7.014638530667735017e-01 1.468584962112742254e-01 4.577261675584743950e-01 2.369208677107362826e-01
5.140998044749989226e-03 2.242527110614195296e-01 2.960251573689319793e-01 6.487144382421085043e-01
1.515616208846672919e-01 4.672244790863220310e-01 3.775278308063384491e-01 1.049011560631800677e-01
```

```
print(np.loadtxt('./data/sample_array.txt'))
```

```
[[0.02573499 0.82494109 0.89756743 0.84206605]
 [0.70146385 0.1468585  0.45772617 0.23692087]
```

```
 [0.005141   0.22425271 0.29602516 0.64871444]
 [0.15156162 0.46722448 0.37752783 0.10490116]]
```

```
# to save as compressed binary
np.save('./data/sample_array.npy', sample_array)
```

```
!cat ./data/sample_array.npy
```

```


```

```
print(np.load('./data/sample_array.npy'))
```

```
[[0.02573499 0.82494109 0.89756743 0.84206605]
 [0.70146385 0.1468585  0.45772617 0.23692087]
 [0.005141   0.22425271 0.29602516 0.64871444]
 [0.15156162 0.46722448 0.37752783 0.10490116]]
```

Topics used by not discussed:

- BASH commands (!)
- `wget`
- `str.split()`
- APIs