Data 301 Data Analytics
Reading and Writing Files in Python

Dr. Irene Vrbik

University of British Columbia Okanagan
irene.vrbik@ubc.ca

Python File Input/Output

Many data processing tasks require reading and writing to files.

`open()` is a built-in function for creating, writing and reading files.

This function takes two parameters; *filename*, and *mode*[1].

- Open a file for *reading* (default):
  ```
  infile = open("input.txt", mode = "r")
  ```
- Open a file for *writing*:
  ```
  myfile = open("output.txt", mode = "w")
  ```

Python will look for the file in the current directory, but you can specify an path if it is located elsewhere.

[1] there are other optional paramters, see `help(open)`

Reading from a text file (as one string)

- Notice that even if you are reading in a text file, the object `myfile` will *not* be a string.

```
>>> infile = open("input.txt", "r")
>>> class(infile)
<class '_io.TextIOWrapper'>
```

- The `open` function returns a file object.
- In the above, I saved my file object to a variable called `infile`. Common naming conventions for file input/file output objects are `fin` and `fout`, resp.
- N.B. If the file cannot be opened, an `OSError` is raised.

Reading from a text file (as one string)

- One way to read a text file in Python is using the `<filename>.read()` method:
- By default `read()` will return the entire text document.

```
>>> this = infile.read()
>>> print(this)
'1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n'
```

- Notice that all characters (including line breaks `\n`) are returned.
- The out outputs of the `read()` method is a string

```
>>> type(this)
<class 'str'>
```

Reading from a text file by parts

- Alternatively we can specify the the number characters you want to return as an argument (notice how newline character `\n` is treated as a character:

```
>>> infile = open("input.txt", "r")
>>> infile.read(4)
'1\n2\n'
```

- `read` will keep our place in the document so that if we call it again it will read the *next* 4 characters.

```
>>> infile.read(4)
'3\n4\n'
```

- Notice how when we call these strings in the `print()` function `\n` is converted to new lines:

```
>>> print(infile.read(4))
5
6
```

Reading from a text file by parts

We can repeatedly call this function until there are no more characters to be read.

```
# this reaches the end of the document
>>> print(infile.read(9))
7
8
9
10

# there are no more characters to be read
>>> infile.read(1)
''
```

Reading from a text file by line

- Alternatively we can read the contents line by line:

```
>>> infile = open("input.txt", "r")
>>> infile.readline()
'1\n'
>>> infile.readline()
'2\n'
```

- As before, it will keep track of where we are in the file until there are no more lines to be read:

```
# ...
>>> infile.readline()
'9\n'
>>> infile.readline()
'10\n'
>>> infile.readline()
''
```

Reading from a text file (as one string)

It is good practice to close a file once you no longer need it to free up resources. Once the file is closed, any further attempts to use the file object (`infile`) will fail.

```
>>> fin = open("input.txt", "r")
>>> print(fin.read(5))
1
2
3

>>> fin.close()  # close the file
>>> fin.read(5)  # can no longer read from file
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Reading from a text file (as one string)

We have the option to save the entire file as a string object to a variable. If we do that, we can access the information regardless of whether or not the file object has been closed:

```
fin = open("input.txt", "r")
val = fin.read()  # read file as one string
fin.close()       # close the file
print(val)        # the entire contents of the file
print(val[0:4])   # first 4 characters
print(val[4:8])   # next 4 characters
print(fin.closed) # returns True
```

You can check whether the file is close using `<filename>.closed` which returns `True` or `False`.

Reading from a text file into a list

- We could read *every* line using the `readlines()` method (as opposed to `readline()` (singular which reads a single line).
- Notice that `readlines()` returns a Python list which includes the end of line characters.

```
>>> fin = open("input.txt", "r")
>>> alllines = fin.readlines()
>>> print(alllines)
['1\n', '2\n', '3\n', '4\n', '5\n', '6\n', '7\n', '8\n',
'9\n', '10\n']
```

Iteration with files

As with any Python list, we can iterate through its contents element by element using a `for` loop.

For this particular scenario, this corresponds to going through the `input.txt` file line by line.

```
>>> for lines in alllines:
...     print(lines)
...
1


2


...


10
```

Iteration with files

You may have noticed extra whitespace between lines.

This is because the `print` function has a newline character printed by default (eg, we are actually printing `'1\n\n'` for the first line).

To remedy this we could use the `strip` method to remove the newline characters before printing.

strip() removes any characters specified as arguments from both left and right of a string. If no argument is specified, then all whitespace from starting from the left (resp. right) is removed until we reach the first non-match.

```
>>> 'aacabbaaccbbcacc'.strip('ac') # removed/not removed
'bbaaccbb'
```

Iteration with files

Hence we can iterate through this file line by line using:

```
>>> for lines in alllines:
...     print(lines.strip('\n'))
...
1
2
...
10
```

While this worked well for this small file, if our text file was very long, creating this list would consume a lot of memory.

To get around this, we can loop over the file line by line, and only read the lines of text that your program needs. . .

Iteration with files

- The *file objects* themselves are actually iteratable (that is we can iterate through them in a `for` loop).
- Hence we traverse through the text file line by line and do something with it, eg, print the contents to the screen.
- Remember that the end of line characters are part of the string on each line.
- We may choose to remove them in combination with print so that we don't produce all that unnecessary white space.
- As before, we mustn't forget to close the file once we are done with it.

Iteration with files

By using a loop we can iterate through the whole file line by line:

```
infile = open("input.txt", "r")
for x in infile:
  print(x)
```

If we only wanted to read this file until we say a 4, say, we could exit the for loop using break:

check for equality

```
fin = open("input.txt", "r")
for x in fin:
    line = x.strip('\n')
    print(line)
    if (int(line) == 4):
        break
fin.close()
```

check for substring

```
fin = open("input.txt", "r")
for x in fin:
    print(x.strip('\n'))
    if '4' in x:
        break
fin.close()
```

Reading text from a file line by line

It is worth mentioning that we can also cycle through the lines of text document using a `while` loop:

Using a for loop

```
infile = open("input.txt", "r")
for line in infile:
    print(line.strip('\n'))
infile.close()
```

Using a while loop

```
infile = open("input.txt", "r")
line = infile.readline()
while line != "":
    line = infile.readline()
    print(line.strip('\n'))
infile.close()
```

Reading text from a file line by line

To avoid writing programs which forget to close the file, we could also use `with`[2]. The `with` statement will automatically close the file after the suite is exited. Hence we never have to write `infile.close()`.

```
# The following will auto-close file
with open("input.txt", "r") as infile:
     for line in infile:
          print(line.strip('\n'))
print(infile.close()) # returns True
```

Tip:

The `with` option is considered best practice as it automatically closes the file for us.

[2]that link may make more sense after we cover `try-except` statements

Writing to a Text File

Selecting the write (`'w'`) mode will allow us to *write* text to a file

```
outfile = open("output.txt", "w")
# writes the numbers 1 through 10 on new lines
for n in range(1,11):
    outfile.write(str(n) + "\n")
# not written to final until we run the following:
outfile.close()
```

Warnings

1. Python will try to overwrite the file `output.txt` if it exists, otherwise, the file will be created.
2. The contents are not written to file until we close it.
3. Numbers need to be converted to strings before writing.

Writing to a Text File

The second warning outlined on the previous slide is yet another reason that the `with` method is generally preferred.

```
with open('output.txt', 'w') as fout:
    for n in range(1,11):
        fout.write(str(n) + "\n")
```

Notice how this method makes it impossible for us to forget to close a file.

Writing to a Text File

This will overwrite the file if it already exists, otherwise, it, creates a new file for writing. The following will overwrite the contents of `output.txt`:

```
with open("output.txt", "w") as f:
    f.write("Test")
```

To create an empty file, we can use the `pass` command

```
# creates an empty file
with open("test2.txt", "w") as f:
    pass
```

Another alternative for creating an empty file is:

```
open(filename, 'w').close()
```

Writing to a Text File

Like the `read` functions, `write` will remember its place within the document and will pick up where it left off:

```
with open("test.txt", "w") as fout:
    fout.write("Test")
    fout.write("Test again")
```

Test.txt file will contain:

TestTest again

To include line breaks, we need to include the newline character `\n`.

```
with open("test.txt", "w") as f:
    f.write("Test")
    f.write("\n Test again")
```

Test.txt file will contain:

Test

Test again

Writing to a Text File

Once we close the file however, we need to select the *append* mode (`'a'`) in order to add text to the end of the document.

```
outfile = open("output.txt", "a")
for n in range(11,20):
    outfile.write(str(n) + "\n")
outfile.close()
```

- As in the `w` (write) mode, the contents are not written to file until we close the file.
- Note that, if the file does not exist, it creates a new file for writing.

Using `split` for CSV files

A common type of file you may want to read into your program is a comma separated value (CSV) file.

We can read csv files by iterating over the file object and using `strip` and `split`:

```
with open("data.csv", "r") as infile:
    for line in infile:
        line = line.strip(" \n")
        fields = line.split(",")
        for i in range(0,len(fields)):
            fields[i] = fields[i].strip()
        print(fields)
```

Using `split` for CSV files

In the code from the previous slide:

- `line` is a string (with the end of line character `\n` removed)
- `fields` is a list containing the individual cell values for the corresponding row.

For the last row in our CSV file:

```
>>> print(type(line))
<class 'str'>
>>> print(line)
1.7702,1.1211,-0.6032,-0.6982,0.4066
>>> print(type(fields))
<class 'list'>
>>> print(fields)
['1.7702', '1.1211', '-0.6032', '-0.6982', '0.4066']
```

Using modules: csv for CSV files

Alternatively, you can use the csv module to read csv files:

By importing the module named csv, we can now call the csv.reader function (see more **here**).

A useful module/function in case you forget the name of your file is the os.listdir() to list all files in a directory.

```
import os
print(os.listdir("."))
```

We can also use the module pprint (for pretty print) to make this output a little neater:

```
from pprint import pprint
pprint(os.listdir("."))
```

Using modules: csv for CSV files

```
import csv
with open("data.csv", "r") as infile:
        csvfile = csv.reader(infile)
        for row in csvfile:
                print(row)
```

- csvfile is a *reader object* we can iterate over in a for loop.
- Each iteration corresponds to a line from data.csv.
- Each row is a Python list of *string* elements containing the data found by removing the delimiters.

```
>>> print(type(csvfile))
<class '_csv.reader'>
>>> print(type(row))
<class 'list'>
>>> print(row)
['1.7702', '1.1211', '-0.6032', '-0.6982', '0.4066']
```

Using modules: csv for CSV files

Remember that each element in this list is currently being treated as a string. Before we do any calculations on this numeric values, we need to convert them using `float`.

```
import csv
# only print the rows that start with a number> 1
with open("data.csv", "r") as infile:
        csvfile = csv.reader(infile)
        for row in csvfile:
                if float(row[0]) > 1:
                        print(row)
```

Example

How many of the following statements are TRUE?

1. A Python file is automatically closed for you.
2. If you use the `with` syntax, Python will close the file for you.
3. To read from a file, use `w` when opening a file.
4. The `read()` method will read the entire file into a string.
5. You can use a for loop to iterate through all lines in a file.

A) 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4

Answer:

How many of the following statements are TRUE?

1. A Python file is automatically closed for you.
2. If you use the `with` syntax, Python will close the file for you.
3. To read from a file, use `w` when opening a file.
4. The `read()` method will read the entire file into a string.
5. You can use a for loop to iterate through all lines in a file.

A) 0 B) 1 C) 2 D) 3 E) 4

Answer:

How many of the following statements are TRUE?

1. A Python file is automatically closed for you.
2. If you use the `with` syntax, Python will close the file for you.
3. To read from a file, use `w` when opening a file.
4. The `read()` method will read the entire file into a string.
5. You can use a for loop to iterate through all lines in a file.

A) 0 B) 1 C) 2 D) 3 E) 4

Answer:

How many of the following statements are TRUE?

1. A Python file is automatically closed for you.
2. If you use the `with` syntax, Python will close the file for you.
3. To read from a file, use `w` when opening a file.
4. The `read()` method will read the entire file into a string.
5. You can use a for loop to iterate through all lines in a file.

A) 0 B) 1 C) 2 D) 3 E) 4

Answer:

How many of the following statements are TRUE?

1. A Python file is automatically closed for you.
2. If you use the `with` syntax, Python will close the file for you.
3. To read from a file, use `w` when opening a file.
4. The `read()` method will read the entire file into a string.
5. You can use a for loop to iterate through all lines in a file.

A) 0 B) 1 C) 2 D) 3 E) 4

Answer:

How many of the following statements are TRUE?

1. A Python file is automatically closed for you.
2. If you use the `with` syntax, Python will close the file for you.
3. To read from a file, use `w` when opening a file.
4. The `read()` method will read the entire file into a string.
5. You can use a for loop to iterate through all lines in a file.

A) 0 B) 1 C) 2 D) *3* E) 4

Try it

Example

1. Write a Python program that writes to the file test.txt the numbers from 20 to 10 on its own line in descending order.
2. Write a Python program that reads your newly created test.txt file line by line and only prints out the value if it is even.
3. Print out the contents of the census file provinces.csv available on Canvas (You may use the `csv` module).
4. Try to print out only the provinces with population>1 million people in 2015 from the data in 3. Hint: You will need to remove the commas from the numbers (eg. 44214 instead of 44,214) using the `replace()` function.

Handling Errors and Exceptions

An **exception** is an error situation that must be handled or the program will fail.

Exception handling is how your program deals with these errors. Examples:

ZeroDivisionError Attempting to divide by zero

IndexError An array index that is out of bounds

TypeError operation is applied to an object of an incorrect type.

NameError when an object could not be found

SyntaxError when a syntax error is encountered

See a list of all errors here.

Example taken from here

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError:  division by zero
>>> 4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError:  name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError:  Can't convert 'int' object to str implicitly
```

The `try-except` statement

The `try-except` statement will handle an exception that may occur in a block of statements:

Execution flow:

- The statements in the try block are executed.
- If no exception occurs:
  - If there is an else clause, it is executed.
  - Continue on with next statement after try.
- If an exception occurs:
  - Execute the code after the except.
- If the optional `finally` block is present, it is always executed regardless if there is an exception or not.
- Keyword `pass` is used if any block has no statements.

For more information see click here

Python Exceptions Block

The general syntax is

```
try:
    # try something that may produce an exception
except ErrorName:
    # only executed if an ErrorName exception
    # is raised above
# the following lines (else and finally clauses)
# are optional:
else:
    # only executed if no exception
finally:
    # always executed
```

N.B. A try statement may have more than one except clause, to specify handlers for different exceptions. However, at most one handler will be executed.

This could be useful in the context of reading files: If we try to read a file that does not exist, we need not have our entire program fail, and try to catch this exception in the following manner:

```
filename = 'nonexistingfile.txt'
try:
    with open(filename, 'r') as f:
        reader = csv.reader(f)
        for row in reader:
            pass #do stuff here
except FileNotFoundError:
    print("Could not read file:", filename)
```

If the nonexistingfile.txt doesn't live in our current working directory the above produces:
`Could not read file: nonexistingfile.txt.`

Python Exceptions Block

```
try:
    # try block, exit if error
    num = int(input("Enter a number:"))
    print("You entered:",num)
except ValueError:
    # only executed if exception
    print("Error: Invalid number")
else:
    # only executed if no exception
    print("Thank you for the number")
finally:
    # always executed
    print("Always do finally block")
```

If the input has not been a valid integer, we will generate (raise) a `ValueError`.

Python Exceptions Block

We can be more generic and catch any error using the following

```
try:
    # try something that may produce an exception
except:
    # only executed if an exception is raised above
else:
    # only executed if no exception
finally:
    # always executed
```

Example

What is the output of the following code if we enter 10?

```
try:
    num = int(input("Enter an integer:"))
    print(num)
except ValueError:
    print("Invalid")
else:
    print("Thanks")
finally:
    print("Finally")
```

A) 10

B) 10
Finally

C) Invalid

D) 10
Thanks

E) 10
Thanks
Finally

Answer:

What is the output of the following code if we enter 10?

```
try:
    num = int(input("Enter an integer"))
    print(num)
except ValueError:
    print("Invalid")
else:
    print("Thanks")
finally:
    print("Finally")
```

A) 10

B) 10
Finally

C) Invalid

D) 10
Thanks

E) *10*
Thanks
Finally

Example

What is the output of the following code if we enter "hat"?

```
try:
    num = int(input("Enter an integer"))
    print(num)
except ValueError:
    print("Invalid")
else:
    print("Thanks")
finally:
    print("Finally")
```

A) hat

B) Invalid

C) Invalid
Finally

D) hat
Thanks
Finally

E) Finally

Answer:

What is the output of the following code if we enter "hat"?

```
try:
        num = int(input("Enter num:"))
        print(num)
except ValueError:
    print("Invalid")
else:
    print("Thanks")
finally:
    print("Finally")
```

A) hat

B) Invalid

C) *Invalid*
Finally

D) hat
Thanks
Fianlly

E) Finally

Raising Errors

- Note that we can always generate an error using the `raise` statement.

```
def raiseHell():
  try:
    raise ValueError
  except ValueError:
    print("You raised Hell!")
```

Calling this function produces:

```
>>> raiseHell()
You raised Hell!
```

Try it: Python Exceptions

Example

Write a Python program that reads two numbers and converts them to integers, prints both numbers, and then divides the first number by the second number and prints the result.

- If we get an exception `ValueError` when converting to an integer, print `Invalid`.
- If we get a `ZeroDivisionError`, print `Cannot divide by 0!`

Internet Terminology

An **Internet Protocol (IP)** address is an identifier for any device – including computers, smartphones and game consoles – that connects to the Internet.

- IP version 4 (IPv4) address comprise 32 bits: 4 numbers in the range of 0 to 255. The numbers are separated by dots. Eg:

`142.255.0.1`

While IPv4 accommodates over 4 billion addresses, the number of unused IPv4 addresses will eventually run out. For that reason, Internet Protocol version 6 (IPv6) is being deployed (which can accommodate 340 billion billion billion billion, addresses)

- IP version 6 (IPv6) address have 128 bits and are represented as a series of eight 4-character hexadecimal numbers. Eg:

`2002 : CE57 : 25A2 : 0000 : 0000 : 0000 : CE57 : 25A2`

Internet Terminology

A **domain** is a related group of networked computers. A **domain name** is a text name for computer(s) that are easier to remember.

- Eg. *facebook.com* is the domain name for the IP address `31.13.80.36`
- Domain names are organized *hierarchically*. The most general part of the hierarchy is at the end of the name.
- Example: people.ok.ubc.ca: ca = Canadian domain, ubc = University of British Columbia, ok = Okanagan campus, people = name of computer/server on campus

Read more about this in this three part series part 1, part 2, part 3.

Internet terminology basics

A **uniform resource locator** (URL) is an address of an item on the Internet. A URL has three parts:

1. Protocol: `http://` Hypertext Transfer Protocol: Tells the computer how to handle the file
2. Server computer's domain name or IP address
3. Item's path and name: Tells the server which item (file, page, resource) is requested and where to find it.

Example:

`http://people.ok.ubc.ca/ivrbik/teaching.html`

http protocol server domain name location of file/resource on server

Accessing (GET) Web Sites via URL with Python

urllib is a package that collects several modules for working with URLs:

urllib.request for opening and reading URLs, more here

urllib.parse for parsing URLs; more here

```
import urllib.request
# dont forget the http://
loc="http://google.com"
site = urllib.request.urlopen(loc)
contents = site.read()
print(contents)
site.close()
```

Engine Search with Python

The URL (that we have input as a string) must be properly URL encoded.

Rather than learning the URL encoding language, we can use `urllib.parse.urlencode` to do it for us.

Notes:

- Requests package is even higher-level and easy to use
- `urllib.request.urlopen` could also take a `Request` object (like a .txt document which can be `read`) as an argument.

To get a shallow understanding of what is going on under the hood, go to www.ask.com and search (or query) "data analysis".

You'll get a new URL that might look something like this:

```
https://www.ask.com/web?q=data+analysis
```

In the above the q is a parameter, with the input data+analysis

We will save this information in a python dictionary (called values on the next slide) and pass it to the urlencoder:

```
>>> values = {'q':'data analysis'}
>>> data = urllib.parse.urlencode(values)
>>> print(data)
q=data+analysis
>>> data = data.encode('utf-8')
>>> print(data)
b'q=data+analysis'
```

Engine Search with Python (POST request)

```
import urllib.parse
import urllib.request

url = 'https://www.ask.com/web'
# Build and encode data
values = {'q':'data analysis'}

data = urllib.parse.urlencode(values)
data = data.encode('utf-8')
req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response:
    page = response.read()
    print(page)
```

- Right now we are just looking at the source and the content is in a form that is not very useful to us.
- Our browser typically does the heavy lifting in order to convert that mess to something readable.
- In the next lecture we'll see some useful tools for parsing though this data to extract the useful information from websites.

Notes:

- BeautifulSoup library to make easier to extract information from HTML pages.

Example

How many of the following statements are TRUE?

1. An IPv4 address has 4 numbers between 0 and 256 inclusive.
2. A domain name is hierarchical with most specific part at the end.
3. Typically, a URL will reference more than one resource/item.
4. Python uses the file module for accessing URLs.

A) 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4

Answer:

How many of the following statements are TRUE?

1. An IPv4 address has 4 numbers between 0 and 255 inclusive.
2. A domain name is hierarchical with most specific part at the end.
3. Typically, a URL will reference more than one resource/item.
4. Python uses the file module for accessing URLs.

A) 0 B) 1 C) 2 D) 3 E) 4

Answer:

How many of the following statements are TRUE?

1. An IPv4 address has 4 numbers between 0 and 255 inclusive.
2. A domain name is hierarchical with most specific part at the end.
3. Typically, a URL will reference more than one resource/item.
4. Python uses the file module for accessing URLs.

A) 0 B) 1 C) 2 D) 3 E) 4

Answer:

How many of the following statements are TRUE?

1. An IPv4 address has 4 numbers between 0 and 255 inclusive.
2. A domain name is hierarchical with most specific part at the end.
3. Typically, a URL will reference more than one resource/item.
4. Python uses the file module for accessing URLs.

A) 0 B) 1 C) 2 D) 3 E) 4

Answer:

How many of the following statements are TRUE?

1. An IPv4 address has 4 numbers between 0 and 255 inclusive.
2. A domain name is hierarchical with most specific part at the end.
3. Typically, a URL will reference more than one resource/item.
4. Python uses the file module for accessing URLs.

A) *0* B) 1 C) 2 D) 3 E) 4

Try it

Example

1. Write a Python program that connects to any web page and prints its contents.
2. Write a Python program that connects to: http://www.sharecsv.com/dl/ab69f200ce5071b27e4af626da293d27/province_population.csv and outputs the CSV data. Modify your program to print each province and its 2015 population in descending sorted order. (See next slide for hint)

Advanced sorting

- Each element in `provinces` list is yet another list containing the province's population and name
- Calling `sort` on this list will sort according to that *first* element (i.e. population)
- For advanced sorting, we could also specify a function as a second argument in `sort`. General syntax:

  `<list>.sort(reverse=<True/False>, key= myFunc)`

- For more on this topic see w3schools and GeeksforGeeks