Files_in_Python

April 18, 2025

1 Files in Python

Beija Richardson 4/18/2025

[2]:
```
import pandas as pd
```

1.1 GUI- Graphic User interfaces

Handy if a non-programmer needs to use your code, or if you want a file browser or something similar.

The long term Python standy by is TKinter- The python interface to a general purpose packag ecalled the Tcl/Tk GUI toolkit

https://docs.python.org/3/library/tkinter.html

1.2 Simple GUI constructor

A more recent package for making simple GUIs is the easygui package

conda install -c conda-forge easygui

http://easygui.sourceforge.net/

1.3 Build a file browser using easyqui

This will act like the R file_choose() function

[9]:
```
!pip install easygui
```

```
Requirement already satisfied: easygui in c:\users\luke\anaconda3\lib\site-
packages (0.98.3)
```

[ ]:
```
# A file selector in easygui

import easygui as g

title = 'dr_jeckyl-1.csv'
filename = g.fileopenbox( title )

print(filename)
```

```
#note that figuring out where the fileopenbox is located can be a bit tough
```

[ ]:
```
# a directory selector in easygui

dir_name=g.diropenbox()
print(dir_name)
```

[ ]:
```
# Text window in easygui
my_string="once in while \n things happen.  Or not."
my_title="You know how it goes"
g.textbox(my_string,my_title)
```

[ ]:
```

```

[ ]:
```

```

2 Files and Pandas

We can read and write files easily using Pandas, there are a bunch of utilities to do this

This is the most useful approach when the data in question is readily imported into a Pandas dataframe, ie when the data is in flat tabular form.

The easiest format to work with is a csv (comma separated values)

To open the file we always need the filename.

If the filename is not in the current working directory, then we need a full path name to open the file.

We will

create a pandas dataframe save it to a csv file in the current directory load it again

[ ]:
```
temp={'names':["Bob","Nancy","Chris","Pat", "Morgan"], 'ages':[10,11,9,10,10]}

my_df=pd.DataFrame(temp)

my_df
```

[ ]:
```
#Save this dataframe to the current working directory

# its not a bad ideas to create a variable to save the file name

outfile="name_example.csv"

# then call the method to_csv with the outfile.  This command also indicates␣
 ↪that we do not want an row index written to the file

my_df.to_csv(outfile, index=False)
```

[ ]:
```
#reading a CSV file,   We have seen this many times

# again a variable to hold the file name we want to load
# we will use a file browser from easygui

infile = g.fileopenbox( title )

check_df=pd.read_csv(infile)

check_df
```

3 Details on read_csv

There are some things to be aware of with read_csv

First, it actually loads the file as a text file, and then analyzes the contents of each row to try to infer what the data type is for each column

If your data file is truly immense, this will be a slow process, and it can take up a lot of memory to load all the data as text (since integers or floats usually take less space to store).

We can specify the data types of the columns and specify this in the read_csv function

For an unknown file, we might read the first 5 lines or so to see the variables, then load it again, supplying a list of variable types

See the pandas read_csv manual at

https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

[ ]:
```
infile="name_example.csv"

check_df=pd.read_csv(infile,nrows=5)

check_df
```

[ ]:
```
# what datatypes did pandas infer?

check_df.dtypes
```

[ ]:
```
# I can now create a dictionary that specifies the data types of each column
#  I'm okay with the data types here, but we could specify other data types,␣
 ↪like 16 bit integers to save space
# relative to the default 32 bit - that alone cuts the storage space of these␣
 ↪columns in half.


my_dtypes={"names":str,"ages":int}

infile="name_example.csv"
```

```
check_df=pd.read_csv(infile,nrows=5,dtype=my_dtypes)

check_df
```

[ ]:
```
check_df.dtypes
```

In such a small example, this type of detail hardly matters.

But if you are working with data sets that are many gigs in size, this will matter greatly

4 Pandas and other file formats

I typically just convert other common formats into a .csv file, it's the simplest approach, but Pandas can load other file formats

There are tools to read Excel files, or other formats, and to read and write to the operating system clipboard, which can be handy.

See

https://pandas.pydata.org/docs/reference/io.html

5 Working with text files

It is also possible to work with unformatted text files.

This is a way that you can open pretty much any file and extract the information from it.

With patience and a good understanding of regex, you can open pretty much any file as text (or binary data) and then process it to extract the data.

This gets important when the data is formatted to be primarily "human-readable" rather than machine-readable.

There are some common specialty formats as well

-JSON and YAML are file formats that use key:value pairs to store data in ways that are both machine readable and also easily read by humans. There are tools to read and write these formats from within python.

-html and xml- These are mark-up languages used for websites and some data storage forms. They have embedded tags that can be used to parse these formats and extract data. This is often important in webscraping

We will focus here on plain text files

6 File handles

Typically, we first obtain a file handle from the operator system, which is used to access the file

(this is somewhat similar in nature to the connection we use to connect to a database)

A file handle is obtained using the open() command.

To open a file, we must specify the file name (including the file path), we must indicate if we want to read (r), write(w), append(a), both read and write (r+). The default is text mode, but we can specify binary mode using rb, wb, ab, rb+ etc.

We may also need to specify the *encoding*, because not all text files use the same encoding of language.

[ ]:
```
infile2="dr_jeckyl.txt"

fhandle2=open(infile2,"r",encoding="utf8")
```

[ ]:
```
# read just 1000 bytes from the file

my_text=fhandle2.read(1000)

print(my_text)
```

7 Encoding

The default encoding in Windows is UTF-16, the Ubuntu version of Linux uses UTF-8, Mac OS uses UTF-8

Older window systems used Latin-1

Most of the time this doesn't really matter

For non-english language sources, many different encodings are in use to handle the diacritical marks used in other languages

8 Reading options

There are many options for file reading functions once we have the handle

I tend to just read the entire text in at once

It comes in as one gigantic string, which then can be parsed out, and split up into individual words or sentences or lines

[ ]:
```
my_text=fhandle2.read()

type(my_text)
```

[ ]:
```
len(my_text)
```

[ ]:
```
my_text[0:1000]
```

[ ]:
```
# When we are done, we should release the file handle and close the file

fhandle2.close()
```

9 Reading a file line by line

We can use the "with" operator and readline() together to read a textfile in line by line, which may be easier to work with.

The "with" operator automates the file closing as well.

This set of operations generates a list of strings

[ ]:
```
count=0;                        #running a counter to determine how many lines
my_lines=[]                     # an empty list, I'll add each line to it as we␣
 ↪load it

for line in open(infile2, encoding="utf-8"):
    my_lines.append(line)
    count=count+1

print(count-1)
print(my_lines[0])
print(my_lines[1])
print(my_lines[2])
```

10 Writing a file

At times you may find the need to write data to a file so you can load it into some other piece of software.

You may find that you need a very specific, non-standard format, so being able to write data out line by line can be what you need to do

The python format string method helps greatly

It is written as f'Just ordinary text writing {variable_name}'

Anything in the curly brackets is added as text

[ ]:
```
a= 4
b=5

print( f'Well, the first value is {a} and the second is {b}')
```

[ ]:
```
# backslash gives us a linefeed
c=-1
d="NYC"

# this is a single line of CSV values

print(f'{a},{b},{c},{d} \n')
```

[ ]:
```
# specify the file and path
# get a file handle to write in text, using utf8 encoding

outfile="write_test.txt"
fhandle3=open(outfile, "w",encoding="utf8")

fhandle3.write(f'{a},{b},{c},{d} \n')

fhandle3.close()

# if you double click on this file name in the file browser section of Jupyter␣
 ↪lab on the left, it will open and you can read the contents
```

11 More

There are many more file operations available, from here you can do some reading on all the options

The text file opening routines you have seen here I first saw when I learned the C language back in 1985. The basic file operations (read, write) in Python (and R, Matlab and many other languages) all seem to work the same way. I don't know if this method was the same in earlier languages or not, C may have lifted it from somewhere else. I do know R and Python use subroutines in C, so I suppose the use of C styel file I/O should not come as a surprise. In any case, you now know how this family of methods works, just look up the fine details in R or Python as needed.

Here is the outline again:

-get a handle, specifying the types of operations

-use the handle to read and/or write

-close the handle.