Introduction

The programs that we have been creating so far have helped us learn the basics of Python and how to organize our code. Now it's time to learn how to work with files in Python and how to save data so that we can reuse it later.

Learning to work with files in Python is a major step in your programming experience as it means you can import data into your programs and export data.

Reading from a file

When you start to write Python in the real world and begin to work with other developers you will notice that an incredible amount of data is held in text files. Reading from a file is an essential Python skill. We're going to learn how to read the entire contents of a file and how to work through the contents of a file one line at time.

Let's get started. The first thing we need is a simple text file which I have prepared with four names in it. now, let's write a program to open, read and print the contents of this file:

```
In [ ]: with open('names.txt') as file_object:
            contents = file_object.read()
        print(contents)
```

Most of what we have just written will be new to you so let's step through it. At line 1 we are using the *open()* function. If we want to do any work with a file we first need to open the file to access it. Makes sense! The *open()* function needs one argument, the name of the file that we want to open. Again, makes sense! Python will look for this file in the directory where the program that's being executed is saved. In this example I have saved the file in the same directory with my Jupyter Notebooks so thats where it looks. The *open()* function returns an object representing the file. Python then assigns this object to *file_object* so that we can work with it.

We have opened the file but we haven't closed it. The keyword *with* used in line 1 closes the file for us once our program no longer needs access to it. We could close the file by calling *close()*, but this can sometimes lead to problems, for example, if a bug in our code prevents the *close()* method from being executed, the file may never close. This is not a big issue for us right now but as your programs get bigger, failing to correctly close them will lead to data corruption and loss. There is also the risk that if we call *close()* too early in our program we will end up trying to perform tasks on a closed file which leads to more errors. By using the keyword *with* Python will figure out when to close the file for us when the *with* block finishes execution.

Now that we have a file object that represents *names.txt* we can use the *read()* method in line 2. This method will read the entire contents of the file and store it as one long string in the variable *contents*. At line 3, we print the value of *contents* and we see that the entire file is returned.

If you are using a different code editor from Jupyter you may notice that an additional blank line has been added after the last name of the output. This is happening in Jupyter to, it's just harder to see it. The reason this is happening is because the *read()* method returns an empty string when it reaches the end of the file. This empty string shows up as blank line. If you want to remove the blank line you can use *rstrip()* in the *print* call.

```
print(contents.rstrip())
```

File paths

Oh god, I hate file paths, they can be so frustrating some times. But they are a necessary evil. In the previous code example we passed in the filename *names.txt* to the *open()* function. Python looked in the directory where the current program file, which is our Jupyter Notebook in this case, is stored. So the first place that Python looks for a file to open is in the home location of the executing program.

There will be times when you want to open a file that is not in your program's home directory. For example, you might keep your program files in a folder called *my_programs* and keep any files that you need to open in a folder called *my_text_files* that sits inside the *my_programs* folder. Even though *my_text_files* sits inside *my_programs* passing the *open()* method the name of a file in *my_text_files* won't work because Python will only look in *my_programs* and no where else, it's lazy like that. It won't get up off the couch and go look in *my_text_files*, you have to make it. To get Python to open files from a directory other than the one where your program file is stored you need to provide a *file path* which tells Python to stop being so lazy, get up of off the couch and go look in a specific location.

Because the folder *my_text_files* is inside *my_programs* we can use a relative file path to open a file from *my_text_files*. A relative file path tells Python to look for a given location relative to the directory where the currently running program is stored. For example:

```
with open('my_text_files/filename.txt') as file object:
```

This line of code tells Python to look for the needed .txt file in the folder *my_text_files* and assumes that *my_text_files* is located inside *my_programs* which it is.

- For windows users, the Windows system uses a backslash (\) instead of a forward slash(/) when displaying file paths, but you can still use a forward slash in your code and it will work as expected.

We can also use what are called *absolute file paths*, because these are usually longer than relative file paths it's helpful to assign them to a variable and then pass that variable to *open()*:

```
file_path =
'/home/tonystaunton/projects/python_projects/text_files/filename.tx
with open(file_path) as file_object:
```

Absolute file paths allow you to read files from any location on your computer. As our programs our small at the moment I would recommend using relative file paths for the moment and keep your files all within the one project or program folder.

```
When using absolute file paths you can't use backslashes
because in Python when you place a backslash inside single
quotes Python interprets this as an escape character. If you
want to use backslashes then you need to use two back
slashes, for example,
'\\home\\tonystaunton\\projects\\python_projects\\text_files\\filen
```

Reading a file line by line

OK, so if you are not too traumatized from the file paths conversation let's continue and learn how to read a file line by line. Why would you need to read a file line by line? You may be looking for certain information in a file, or you may want to modify the text in a file in some way. You can use a *for* loop on the file object to examine each line from a file one at a time.

In [ ]:
```python
filename = 'names.txt'

with open(filename) as file_object:
    for line in file_object:
        print(line)
```

In this code example we assign the name of the file we're working with to the variable *filename*. This is a best practice when working with files in Python. Why? The variable *filename* does not represent that actual file, only a string telling Python where to find the file. We could easily change this to 'company_name.txt' or anything else and our program would still perform the same tasks on this file.

After we call *open()* an object representing the file and its contents is assigned to the variable *file_object*. Also, at line 3 we are using the *with* syntax to let Python open and close the file correctly. To examine the files contents we work through each line in the file by looping over the file object at line 4.

As you can see from the output each name is separated by a blank line which is not in our file. These new lines appear because Python adds a newline character to the end of each line in the text file. Don't forget that the *print* function also adds its own newline each time we call it, so we end up with two new line characters at the end of each line, one from the file and one from *print()*. We can use *rstrip()* on each line in the *print()* call to remove these extra blank lines:

In [ ]:
```python
filename = 'names.txt'

with open(filename) as file_object:
    for line in file_object:
        print(line.rstrip())
```

Making a list of lines from a file

One disadvantage of using *with* is that the file object returned by *open()* is only available inside the *with* block that contains it. If you would like to access a files contents outside the *with* block you can store the file's lines in a list inside the block and then work with that list.

Let's take a look at an example were we store the lines of *names.txt* in a list inside the *with* block and then print the lines outside the block...magic!

In [ ]:
```
filename = 'names.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line.title().rstrip())
```

In this code example we use the *readlines()* method at line 4 to take each line from the file and store it in a list. This list is then assigned to the variable *lines* which we can continue to work with after the *with* block ends. At line 6 we have a simple *for* loop which we use to print each line from *lines*.

Working with a files contents

After a file has been read into memory you can perform any action you want on the data. Let's take a look:

In [ ]:
```
filename = 'names.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

name_string = ''
for line in lines:
    name_string += line.rstrip()

print(name_string)
print(len(name_string))
```

In this code example we start by opening the file and storing each line of names in a list. At line 6 we create a variable called *name_string* which will hold all of the names in our file. At line 7 we create a loop that adds each name to *name_string* and removes the newline character from each line. Then at line 3 we print this string of names and also show the length of the string.

Large files

Our programs so far have worked with a small text file containing only four name. Our code would work just as well with a file that is much larger in size. Let's take a look. The book Wuthering Heights is free to download as a text file from http://www.gutenberg.org/files/768/768-0.txt. Go ahead and download now or take it from the resources section of this lesson. Make sure to save it in the same directory as your current python programs.

In [ ]:
```
filename = 'wh.txt'

with open(filename) as file_object:
    lines = file_object.readlines()
book_string = ''
for line in lines:
    book_string += line.strip()

print(f"{book_string[:1000]}...")
print(len(book_string))
```

The output shows the first 1000 characters of the file along with how many characters are in the entire file.

Is your name in Wuthering Heights?

Let's write a program to scan the book Wuthering Heights and if a particular name is contained within. We can do this by expressing each name as a string and seeing if that string appears anywhere in the book:

In [ ]:
```
filename = 'wh.txt'

with open(filename) as file_object:
    lines = file_object.readlines()
book_string = ''
for line in lines:
    book_string += line.strip()

name = input("What is your name? ")
if name in book_string:
    print("Your name appears in wuthering heights!")
else:
    print("Sorry!")
```

In this code example we ask the user to input their name at line 9 we then assign this input to the variable *name*. At line 10 we check if the string is in Wuthering Heights.

As you can see the speed of return is very fast.

Exercises

- creata a blank txt file and add some lines of daily to-dos in it, for example 'buy milk', 'clean the car'. Save the file task_list.txt in the same directory as your current programs. Write a program that reads your task list and prints it out.
- Print the contents of your tasks list by reading the entire file, by looping over the file object and by storing the lines in a list and then working with them outside the

block.
- Add some intro text to your output telling a who what is on their list for today

Writing to a file

We've learned how to read data from a file, now let's learn how to write data to a file. Writing data to a file means that a programs output will still be available after you close the code editor or terminal containing your program. As a result, you can review a programs output after the program has finished running, and you can share the output with others.

Writing to an empty file

To write text to a file we need to call *open()* this time with a second argument telling Python that will be writing to the file. Let's take a look:

In [1]:
```
filename = 'share.txt'

with open(filename, 'w') as file_object:
    file_object.write("Tony was here!")
```

Magic!

In this code example the call to *open()* at line 3 takes two arguments. The first argument is what we have seen previously, it is the name of the file that we want to open. The second argument *w*, informs Python that we want to open the file in *write mode*. You can open a file in:

- read mode, 'r'
- write mode, 'w'
- append mode, 'a'
- read and write mode, 'r+'

If you leave out or omit the mode argument Python will open the file in read-only mode by default.

Also, the *open()* function automatically creates the file you're writing to if it doesn't already exist. If a file does already exist and you open it in *'w'* mode, Python will erase the contents of the file before returning the file object.

At line 4 of the code example above we use the *write()* method on the file object to write a string to an empty file. Let's open the file to see the output.

Writing multiple lines

The *write()* function doesn't add any newlines to the text we write. So if we write more than one line without including newline characters the text will just continue on with no spacing.

In [2]:
```python
filename = 'share.txt'

with open(filename, 'w') as file_object:
    file_object.write("Tony was here!")
    file_object.write("So were you!")
```

To separate the newlines we need to include the newline character, '\n'.

In [3]:
```python
filename = 'share.txt'

with open(filename, 'w') as file_object:
    file_object.write("Tony was here!\n")
    file_object.write("so were you!")
```

You can also use space, tabs and blanks lines to achieve the desired output.

Appending to a file

The append mode allows you to add content to a file without overwriting the file contents every time. When you open a file in append mode Python doesn't erase the contents of the file before returning the file object. Any lines you write to the file will be added at the end of the file. Also, if the file does not yet already exist Python will create the file for you.

In [4]:
```python
filename = "append_mode.txt"

with open(filename, 'a') as file_object:
    file_object.write("Tony was here on Monday.\n")
    file_object.write("Tony was also here on Tuesday.\n")
```

In this code example at line 3 we use the 'a' argument to open a file for appending. At line 4 we write two new lines.

Exercises

- Write a program that asks users for their name. Add the response to a file called log.txt
- Update the above code to open log.txt in append mode so that every time a user enters their name it is added to the file
- Update your program to prompt the user for the date. Add this response to your log.txt file

Exceptions

As we have been learning Python we have seen plenty of errors, some by design and others by eh...design and not typos. Python uses special objects called exceptions to manage errors that come about during a program's execution. When an error occurs that causes Python to stop it creates an exception object. We can write code that handles these exceptions and our programs will continue to run. If we don't handle exceptions our programs will stop and show a traceback, which includes a report of the exception that was raised.

In Python, exceptions are handled with *try-except* blocks. These blocks ask Python to do something but they also tell Python what to do if an exception is raised. When we use *try-except* blocks, our programs will continue running even when things go wrong. Instead of tracebacks, which can be confusing to users, they will see helpful error messages that we write.

Handling the ZeroDivisionError Exception

Did you know that is it impossible to divide a number by zero? To double check let's ask Python:

In [5]:
```
print(5/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Input In [5], in <cell line: 1>()
----> 1 print(5/0)

ZeroDivisionError: division by zero
```

The output returned is a traceback error, and as you can see from the first line it's called a *ZeroDivisionError*. This is also what is known as an exception object. Python creates this kind of object in response to a situation where it can't do what we have asked it to do. When this happens, Python stops the program and tells us the kind of exception that was raised. We can use this information to change the way our programs behave. We'll tell Python what to do when this kind of exception occurs again.

Using try-except blocks

When you think that there is the possibility that an error may occur, you can write a *try-except* block to handle the exception that may araise. Let's take a look at a *try-except* block that will handle the ZeroDivisionError:

In [6]:
```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide a number by zero!")
```

```
You can't divide a number by zero!
```

Now when we run our program we get a helpful error message. As you can see at line 1 we have added a *try* block and inside it we have placed the code *print(5/0)*. If the code inside the *try()* block results in an error, Python then looks for an *except* block whose error matches the one that was raised and runs the code in the block. Our code has raised a ZeroDivisionError at line 2 inside a *try* block so Python looks for a corresponding *except* block to tell it how to handle the error. Python runs the code in the *except* block and our user sees a helpful error message.

If more code were to follow the *try-except* block then that would still work because we have told Python what to do.

```
In [7]: print(5/0)
        print("I told you!")
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Input In [7], in <cell line: 1>()
----> 1 print(5/0)
      2 print("I told you!")

ZeroDivisionError: division by zero
```

```
In [8]: try:
            print(5/0)
        except ZeroDivisionError:
            print("You can't divide a number by zero!")

        print("I told you!")
```

```
You can't divide a number by zero!
I told you!
```

Using exceptions to prevent crashes

As you can see, handling errors becomes even more important if you need your programs to keep running after an error has occurred. This is a common problem with programs that ask a user for input. What happens if a user enters the wrong kind of information, should you program just stop working? Let's take a look at an example:

```
In [12]: print("Divide 2 numbers.")
         print("Enter 'q' to quit.")

         while True:
             first_nmber = input("\nFirst number: ")
             if first_nmber == 'q':
                 break
             second_number = input("Second number: ")
             if second_number == 'q':
                 break
             answer = int(first_nmber)/int(second_number)
             print(answer)
```

```
Divide 2 numbers.
Enter 'q' to quit.

First number: 4
Second number: 2
2.0
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Input In [12], in <cell line: 5>()
      2 print("Enter 'q' to quit.")
      4 while True:
----> 5     first_nmber = input("\nFirst number: ")
      6     if first_nmber == 'q':
      7         break

File ~/opt/anaconda3/lib/python3.9/site-packages/ipykernel/kernelbase.py:107
5, in Kernel.raw_input(self, prompt)
   1071 if not self._allow_stdin:
   1072     raise StdinNotImplementedError(
   1073         "raw_input was called, but this frontend does not support in
put requests."
   1074     )
-> 1075 return self._input_request(
   1076     str(prompt),
   1077     self._parent_ident["shell"],
   1078     self.get_parent("shell"),
   1079     password=False,
   1080 )

File ~/opt/anaconda3/lib/python3.9/site-packages/ipykernel/kernelbase.py:112
0, in Kernel._input_request(self, prompt, ident, parent, password)
   1117             break
   1118 except KeyboardInterrupt:
   1119     # re-raise KeyboardInterrupt, to truncate traceback
-> 1120     raise KeyboardInterrupt("Interrupted by user") from None
   1121 except Exception:
   1122     self.log.warning("Invalid Message:", exc_info=True)

KeyboardInterrupt: Interrupted by user
```

In this code example we ask a user enter a first number and then a second number, so long as they do not enter 'q' to quit. The program divides the two numbers and displays the output. There is, obviously, no error handling in this program.

When our program crashes there is no way for a user to continue. They will need to restart the program to divide another two numbers.

The else block

We can help this program by wrapping the line that might produce an error in a *try-except* block. In the previous example the error occurred on the line that performs the division. That's where we'll add the *try-except* block.

In [13]:
```
print("Divide 2 numbers.")
print("Enter 'q' to quit.")

while True:
    first_nmber = input("\nFirst number: ")
    if first_nmber == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break
    try:
        answer = int(first_nmber)/int(second_number)
    except ZeroDivisionError:
        print("You can't divide by 0!")
    else:
        print(answer)
```

```
Divide 2 numbers.
Enter 'q' to quit.

First number: 5
Second number: 0
You can't divide by 0!

First number: 4
Second number: 2
2.0

First number: q
```

This code example includes an *else* block. Any code that depends on the *try* block executing correctly goes in the *else* block.

Just as we have done before, we ask Python to divide two numbers provided by the user in a *try* block at line 11. As you can see the *try* block only contains the line of code that might cause the error. Any code that depends on the code block succeeding is placed inside the *else* block. In this code example, if the division is successful then the *else* block will print the answer.

The *except* block at line 13 tells Python how to respond if and when a *ZeroDivisionError* occurs. If the *try* block fails because a user tries to divide a number by 0 then we print a helpful error message. In this code example the user never sees a traceback error.

Handling the FileNotFoundError Exception

As you start to work with files you will notice a common problem again and again, and that is missing files. The file you are looking for may have been moved, deleted or renamed. When these situations occur they can be handled using a *try-except* block.

Let's try to read a file that doesn't exist.

In [14]:
```
filename = 'abc.txt'

with open(filename, encoding='utf-8') as f:
    contents = f.read()
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Input In [14], in <cell line: 3>()
      1 filename = 'abc.txt'
----> 3 with open(filename, encoding='utf-8') as f:
      4     contents = f.read()

FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'
```

This code doesn't work but we'll get to that in a minute. We have done a couple of things differently in this code example. The first is that we have used the letter *f* to represent the file object which is a common programming convention in Python. Secondly, we have added the *encoding* argument. This argument is needed when the default encoding of your system doesn't match the encoding of the files that's being read.

Python can't read from a missing file so it raises a traceback error. In this code the *open()* function is producing the error so we handle it by placing it inside a *try* block:

```
In [15]: filename = 'abc.txt'

         try:
             with open(filename, encoding='utf-8') as f:
                 contents = f.read()
         except FileNotFoundError:
             print(f"Sorry, the file {filename} can't be found.")
```

```
Sorry, the file abc.txt can't be found.
```

In this code example, the code in the *try* block produces a FileNotFoundError and so Python looks for an *except* block that matches that error. Python then runs the code found in that block and the user sees a helpful error message.

Analyzing text

In Python you can analyze text files containing entire books, and we seen a very small example of this in a previous program. Let's use Python to count how many words there are in Wuthering Heights, which we used in a previous program. We'll use the string method *split()* which can build a list of words from a string. Here's a small example of the *split()* method.

```
In [16]: title = "Wuthering Heights"
         title.split()
```

```
Out[16]: ['Wuthering', 'Heights']
```

The *split()* method does exactly what it sounds like, it splits or separates a string into parts wherever it finds a space and store all the parts of the string in a list. Now, let's use the *split()* method on the entire text of Wuthering Heights.

```
In [19]: filename = 'wh.txt'

         try:
             with open(filename, encoding='utf-8') as f:
                 contents = f.read()
         except FileNotFoundError:
             print(f"Sorry, the file {filename} can't be found.")
         else:
             # Count the words in a text file
             words = contents.split()
             num_words = len(words)
             print(f"The file {filename} has approx. {num_words} words in it.")
```

```
Sorry, the file wh.txt can't be found.
```

In this code example, at line 10 we take the contents of the string, which contains the entire text of Wuthering Heights, and use *split()* to produce a list of all the words in the book. At line 11, we use the *len()* to get the number of words in the list. This number is assigned to the variable *num_words*. At line 12 we print out the filename and how many words it contains. Youll notice that all of this code was placed within a *try-except* block.

Working with multiple files

Let's add more books to analyze. To do this lets first optimize our code by moving it to a function called *word_count()*. This will make it easier to work with several books.

In [21]:
```python
def word_count(filename):
    """Count the number of words in a text file."""

    try:
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        print(f"Sorry, the file {filename} can't be found.")
    else:
        words = contents.split()
        num_words = len(words)
        print(f"The file {filename} has approx. {num_words} in it.")

filename = 'wh.txt'
word_count(filename)
```

```
The file wh.txt has approx. 118980 in it.
```

In this code example, we have placed our code inside a function called *word_count()*. We can now write a loop to count the words in any text. We do this by storing the names of the files we want to analyze in a list, we then call *word_count()* for each file in the list.

In [23]:
```python
def word_count(filename):
    """Count the number of words in a text file."""

    try:
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        print(f"Sorry, the file {filename} can't be found.")
    else:
        words = contents.split()
        num_words = len(words)
        print(f"The file {filename} has approx. {num_words} words in it.")

filenames = ['wh.txt', 'md.txt', 'abc.txt']
for filename in filenames:
    word_count(filename)
```

```
The file wh.txt has approx. 118980 words in it.
Sorry, the file md.txt can't be found.
Sorry, the file abc.txt can't be found.
```

As you can see in this example we added a loop at line 15 to loop through the books we want to analyze. I have also added in a book that does not exist so that we can see our error handling in action. As you can see the program still runs if after an error has occurred. Also, in this code a user does not see a traceback error just a helpful text message.

Failing silently

Sometimes, in your programs you will not want to report every error that occurs to the user. In these instances you will just want the program to continue on. To make a program fail silently, we write a *try* block as normal, but we explicitly tell Python to do nothing in the *except* block. To accomplish this we use Python's *pass* statement.

In [24]:
```
def word_count(filename):
    """Count the number of words in a text file."""

    try:
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        pass
    else:
        words = contents.split()
        num_words = len(words)
        print(f"The file {filename} has approx. {num_words} in it.")

filenames = ['wh.txt', 'md.txt', 'abc.txt']
for filename in filenames:
    word_count(filename)
```

```
The file wh.txt has approx. 118980 in it.
```

As you can see at line 8 we have added the *pass* statement. Now when a FileNotFoundError occurs the code in the *except* block runs but nothing happens. Users see no indication that a file was not found.

As a developer you need to pick and choose when you display error messages. What do I mean? For example in the last program we failed silently but a user may need to know that a file was not found to complete their work. A good rule of thumb is to place code that requires user input inside *try-except* blocks and to provide helpful error messages. This is something that will improve as you gain more experience.

Exercises

- Download a book from gutenberg.org and write a program that opens the file and appends a piece of text to it
- Append the text "I have read this book" to your file
- Use the code from a previous example to scan the text file and see how many times a particular word occurs for example 'house' or 'dog'
- Use the *count()* method to find out how many times a word or phrase appears in a string. For example: message = "hello tony" line.count('tony')

Storing data

We've seen how to store data in Python data structures such as lists and dictionaries. At the moment when a user closes our programs the data is lost. We can give users the ability to store information using the *json* module.

The json module allows you to dump simple Python data structures into a file and load the data from the file the next time the program is run. A distinct advantage to using the *JSON* data format to persist data is that it is not specific to Python, which means you can share data you store in JSON with developers working with other programming languages. Let's take a look:

Using json.dump() and json.load()

Let's create a program to store a list of names, and another program to read the list. The first program will use *json.dump* to store, and the second program will use *json.load()*.

In [25]:
```
import json

names = ['tony', 'frank', 'mary']

filename = 'names.json'
with open(filename, 'w') as f:
    json.dump(names, f)
```

As you can see in this code example, the *json.dump()* function takes two arguments, a piece of data to store and a file object it can use to store the data.

At line we import the *json* module. At line 3 we create a list of names to work with. At line 5 we choose a filename in which to store the list of numbers. Its common to use the *.json* file extension to indicate that the file is JSON format. At line 6 we open the file in write mode which allows json to write the data to the file. Finally at line 7 we use the *json.dump()* function to store the list of names.

There is no output from this file.

In [27]:
```
import json

filename = 'names.json'
with open(filename) as f:
    names = json.load(f)

print(names)
```

```
['tony', 'frank', 'mary', 'paul']
```

BOOM! Like magic!

At line 3 we are reading in the same file that we wrote to. At line 4 when we open the file we are doing so in read mode. Remember Python opens in read mode by default. Then at line 5 we use the *json.load()* function to load the information stored in *names.json* and we assign it to the variable *names*. We then print the list.

Saving and reading user generated data

Let's now create a program to store user input:

In [28]:
```
import json

username = input("Please enter your username? ")

filename = 'username.json'
with open(filename, 'w') as f:
    json.dump(username, f)
    print(f"Thank you! Your username, {username}, has been saved")
```

```
Please enter your username? tony
Thank you! Your username, tony, has been saved
```

In this code example we ask a user to enter the user name at line 3. We then use *json.jump()* and pass it a username and a file object to store the username in. Finally we print out a message.

Now let's write a program that welcomes a user by their username:

In [29]:
```
import json

filename = 'username.json'

with open(filename) as f:
    username = json.load(f)
    print(f"Hi {username}!")
```

```
Hi tony!
```

In this example we load data back from the json file at line 6.

Next let's combine these two programs into one. As we do we'll add a *try* block that attempts to recover a username. If the file does not exist we can use the *except* block to prompt for a user name and store it in *username.json* for use next time.

In [32]:
```
import json

# Load a username if it has been saved previously
# Otherwise, prompty for the username and save it.

filename = 'username.json'

try:
    with open(filename) as f:
        username = json.load(f)
except FileNotFoundError:
    username = input("Please enter your username? ")
    with open(filename, 'w') as f:
        json.dump(username, f)
        print(f"Thank you! Your username, {username}, has been saved")
else:
    print(f"Hi, {username}!")
```

```
Hi, tony!
```

There is nothing new here, just previous code combined. At line 9 we open the file *username.json*. If the file exists, the user name is printed to the screen. If the files does not exist a FileNotFoundError will occur. Python will move to line 11, the *except* block, where a user is asked to enter a username. We then use *json.dump()* to store the username and print a success message.

Exercises

- Write a program that asks a user for their favorite color and save the response to a json file.

- Write another program to tell a user what there favorite color is.
- Combine the two programs from above