Introduction to

python

for scientific computing

- Lecture 6

Recap from Lecture 5

New type of string: f-strings

```
print("There are " + str(len(genres_list)) + " unique genres")
```

can be written as:

```
print(f"There are {len(genres_list)} unique genres")
```

- Remember to put the `f` before the first `"` sign
- Anything in the brackets will be automatically replaced inside the string

New data type: `set`

- A set contains an unordered collection of unique and immutable objects

Syntax:

For empty set:

```
setName = set()
```

For populated sets:

```
setName = {1,2,3,4,5}
```

Common operations on `sets`

```
set.add(a)
len(set)
a in set
```

In [74]:

```
x = set()
x.add(100)
x.add(25)
x.add(3)
x.add('3.0')
x.add(3)
x
#for i in x:
#    print(type(i))
#print(x)
#x.append(4)
##mySet = {2,5,1,3}
#mySet.add(5)
#mySet.add(4)
#print(mySet)
```

Out[74]:

```
{100, 25, 3, '3.0'}
```

New data type: `dictionary`

- A dictionary is a mapping of unique keys to values
- Dictionaries are mutable

Syntax:

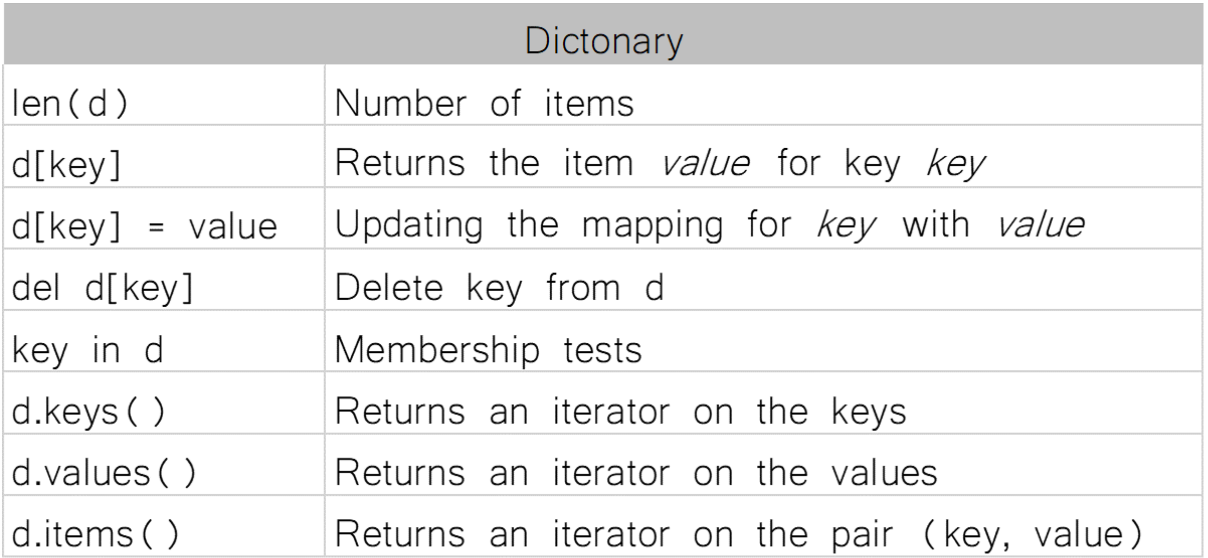
`a = {}` (create empty dictionary)

`d = {'key1':1, 'key2':2, 'key3':3}`

```
In [82]: myDict = {'drama': 4, 'thriller': 2, 'romance': 5}
         myDict
```

```
Out[82]: {'drama': 4, 'thriller': 2, 'romance': 5}
```

Operations on Dictionaries



Dictonary	
len(d)	Number of items
d[key]	Returns the item *value* for key *key*
d[key] = value	Updating the mapping for *key* with *value*
del d[key]	Delete key from d
key in d	Membership tests
d.keys( )	Returns an iterator on the keys
d.values( )	Returns an iterator on the values
d.items( )	Returns an iterator on the pair (key, value)

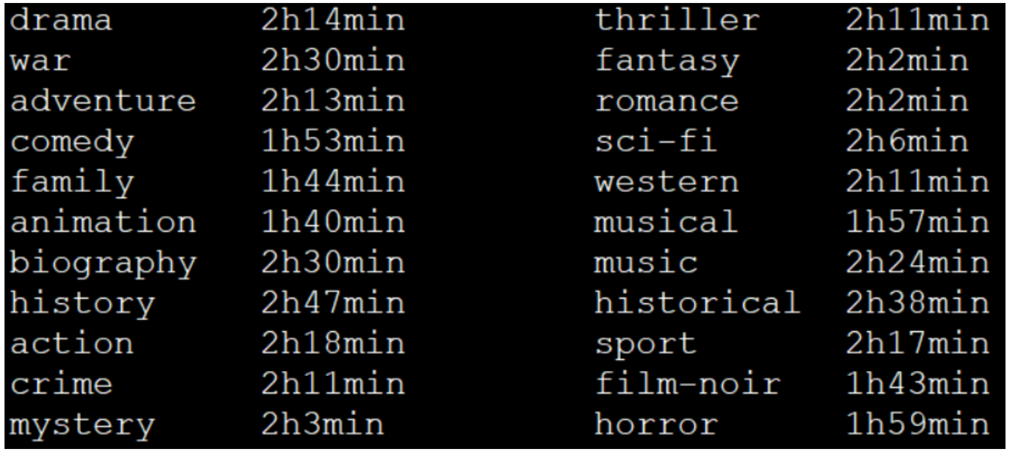
In [15]:

```
myDict = {'drama': 4,
          'thriller': 2,
          'romance': 5}

myDict["thriller"] += 1
some_variable = "drama"
myDict[some_variable]
#len(myDict)
#myDict['drama']
#myDict['horror'] = 2
#del myDict['horror']
#"adventure" in myDict
#myDict.keys()
#myDict.items()
#myDict.values()
```

```
Out[15]: 4
```

Answer

What is the average length of the movies (hours and minutes) in each genre?

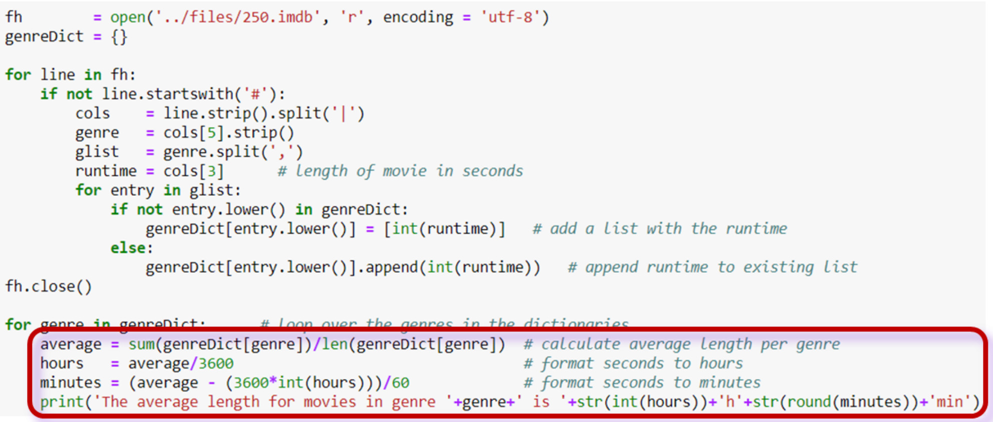


```
drama        2h14min          thriller    2h11min
war          2h30min          fantasy     2h2min
adventure    2h13min          romance     2h2min
comedy       1h53min          sci-fi      2h6min
family       1h44min          western     2h11min
animation    1h40min          musical     1h57min
biography    2h30min          music       2h24min
history      2h47min          historical  2h38min
action       2h18min          sport       2h17min
crime        2h11min          film-noir   1h43min
mystery      2h3min           horror      1h59min
```

Tip!
Here you have to loop twice

```python
fh        = open('../downloads/250.imdb', 'r', encoding = 'utf-8')
genreDict = {}

for line in fh:
    if not line.startswith('#'):
        cols    = line.strip().split('|')
        genre   = cols[5].strip()
        glist   = genre.split(',')
        runtime = cols[3]      # length of movie in seconds
        for entry in glist:
            if not entry.lower() in genreDict:
                genreDict[entry.lower()] = [int(runtime)]   # add a list with the runtime
            else:
                genreDict[entry.lower()].append(int(runtime))   # append runtime to existing list
fh.close()
for genre in genreDict:      # loop over the genres in the dictionaries
    average = sum(genreDict[genre])/len(genreDict[genre])  # calculate average length per genre
    hours   = int(average/3600)                                 # format seconds to hours
    minutes = (average - (3600*hours))/60             # format seconds to minutes
    print('The average length for movies in genre '+genre\
          +' is '+str(hours)+'h'+str(round(minutes))+'min')
```

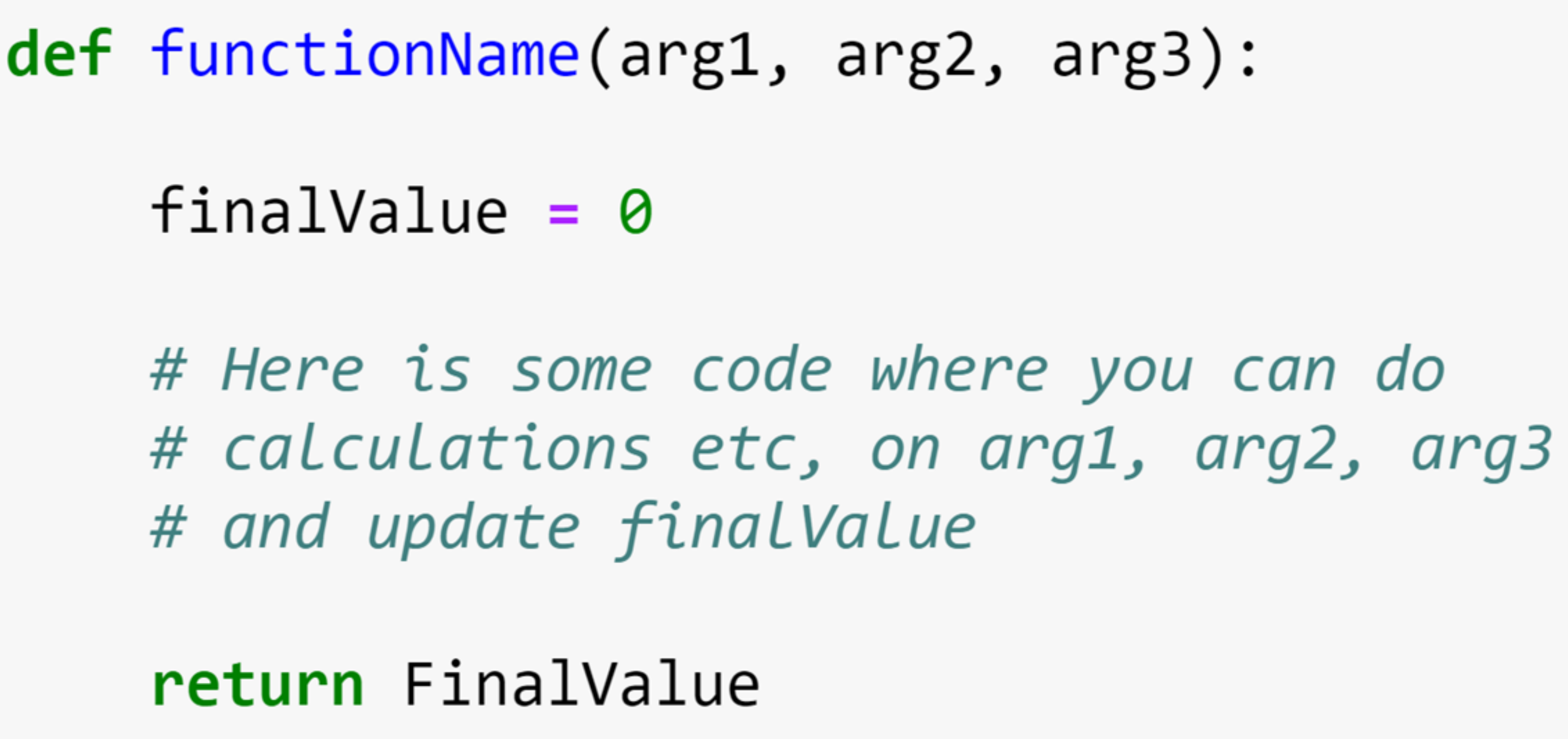
New topic: Functions



```
fh        = open('../files/250.imdb', 'r', encoding = 'utf-8')
genreDict = {}

for line in fh:
    if not line.startswith('#'):
        cols    = line.strip().split('|')
        genre   = cols[5].strip()
        glist   = genre.split(',')
        runtime = cols[3]      # length of movie in seconds
        for entry in glist:
            if not entry.lower() in genreDict:
                genreDict[entry.lower()] = [int(runtime)]   # add a list with the runtime
            else:
                genreDict[entry.lower()].append(int(runtime))   # append runtime to existing list
fh.close()

for genre in genreDict:      # loop over the genres in the dictionaries
    average = sum(genreDict[genre])/len(genreDict[genre])  # calculate average length per genre
    hours   = average/3600                                 # format seconds to hours
    minutes = (average - (3600*int(hours)))/60             # format seconds to minutes
    print('The average length for movies in genre '+genre+' is '+str(int(hours))+'h'+str(round(minutes))+'min')
```

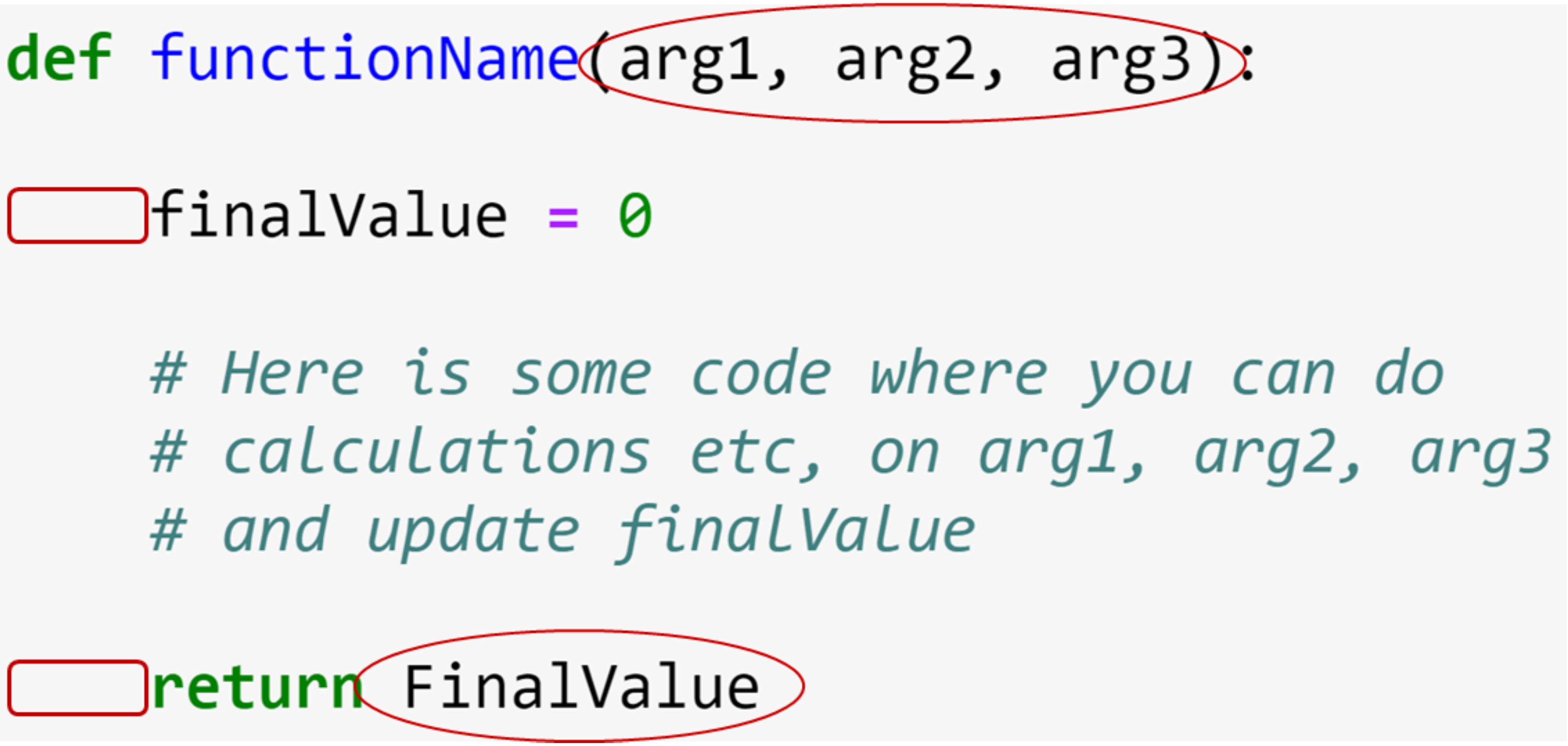
If you will do something many times, you can export it into a function. This will make your code look better, and avoid problems with copy-paste (repeated identical blocks of code)

Function structure



```
def functionName(arg1, arg2, arg3):

    finalValue = 0

    # Here is some code where you can do
    # calculations etc, on arg1, arg2, arg3
    # and update finalValue

    return FinalValue
```

Function structure



```
def functionName(arg1, arg2, arg3):

    finalValue = 0

    # Here is some code where you can do
    # calculations etc, on arg1, arg2, arg3
    # and update finalValue

    return FinalValue
```

```python
def addFive(input_number):
    result = input_number + 5
    return result

res = addFive(4)
print(res)
9
```

```python
from datetime import datetime

def whatTimeIsIt():
    time = 'The time is: ' + str(datetime.now().time())
    return time

whatTimeIsIt()
```

Out[23]:

```
'The time is: 13:49:44.148141'
```

```
def FormatSec(genre):   # input a list of seconds, output is a string
    average   = sum(genreDict[genre])/len(genreDict[genre])
    hours     = int(average/3600) # average // 3600
    minutes   = (average - (3600*hours))/60 # (average % 3600) % 60
    return str(hours)+"h"+str(round(minutes))+"min"



fh        = open("../downloads/250.imdb", "r", encoding = "utf-8")
genreDict = {}

for line in fh:
    if not line.startswith("#"):
        cols    = line.strip().split("|")
        genre   = cols[5].strip()
        glist   = genre.split(",")
        runtime = cols[3]      # length of movie in seconds
        for entry in glist:
            if not entry.lower() in genreDict:
                genreDict[entry.lower()] = [int(runtime)]   # add a list with the runtime
            else:
                genreDict[entry.lower()].append(int(runtime))   # append runtime to existing list
fh.close()

for genre in genreDict:
    print("The average length for movies in genre "+ genre +" is "+ FormatSec(genre))
```

Why use functions?

- Cleaner code
- Better defined tasks in code
- Re-usability
- Better structure

Scope

- Variables within functions can't be seen from outside the functions
- Global variables are seen everywhere (within functions as well)

In [53]:

```
global_variable = "global string"

def some_function():
    local_variable = "local string"
    print(f"local from inside function: {local_variable}")
    print(f"global from inside function:  {global_variable}")

print(f"global from outside function: {global_variable}")
some_function() # will be printed from inside the function
print(local_variable) # can't see local variable from outside function
```

```
global from outside function: global string
local from inside function: local string
global from inside function:  global string

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [53], in <cell line: 10>()
      8 print(f"global from outside function: {global_variable}")
      9 some_function() # will be printed from inside the function
---> 10 print(local_variable)

NameError: name 'local_variable' is not defined
```

Importing functions

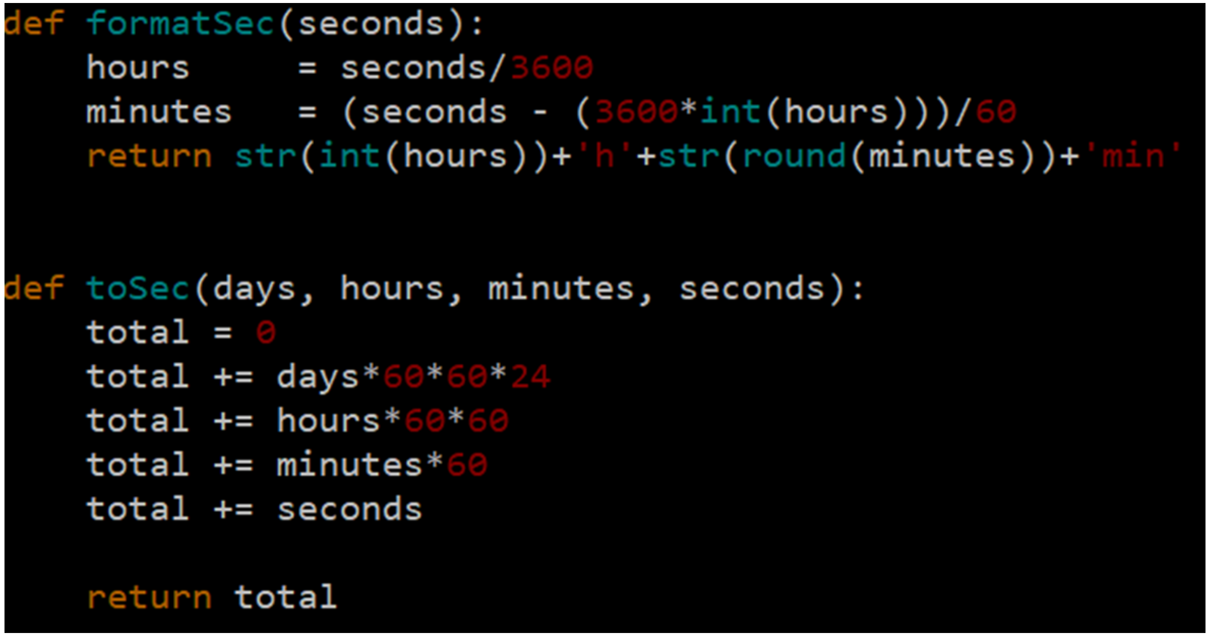
- Maybe there are functions you reuse all the time across different scripts
- Collect all your functions in another file -> import that file
- Keeps main code cleaner
- Easy to use across different code

Example:

1. Create a file called myFunctions.py, located in the same folder as your script
2. Put a function called `formatSec()` in the file
3. Start writing your code in a separate file and `import` the function

```
In [41]: from myFunctions import formatSec

         seconds = 32154

         formatSec(seconds)

Out[41]: '8h56min'
```

myFunctions.py (it’s in this same folder)



```python
def formatSec(seconds):
    hours      = seconds/3600
    minutes    = (seconds - (3600*int(hours)))/60
    return str(int(hours))+'h'+str(round(minutes))+'min'


def toSec(days, hours, minutes, seconds):
    total = 0
    total += days*60*60*24
    total += hours*60*60
    total += minutes*60
    total += seconds

    return total
```

```
from myFunctions import  formatSec, toSec

seconds = 21154
print(formatSec(seconds))
#print(myFunctions.formatSec(seconds))

days    = 0
hours   = 21
minutes = 56
seconds = 45

print(toSec(days, hours, minutes, seconds))
```

```
5h53min
79005s
```

Summary

- A function is a block of organized, reusable code that is used to perform a single, related action
- Variables within a function are local variables
- Functions can be organized in separate files and imported to the main code

New topic: `sys.argv`

- We have seen how you can write your own script
- We have seen how to read and write files in those scripts: ``` ... fh = open("../Downloads/250.imdb", "r") ... out = open("results.csv", "w") ...

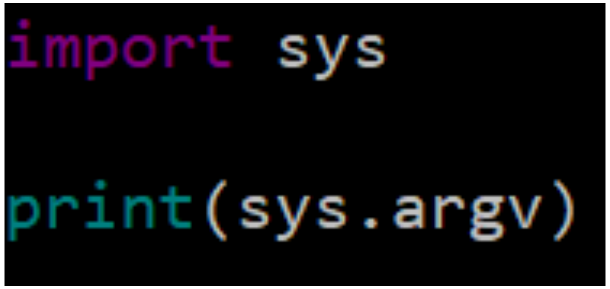
- What if we want this script to work on *any* input or output file?

New topic: `sys.argv`

- Avoid hardcoding the filename in the code
- Easier to re-use code for different input files
- Uses command-line arguments
- Input is list of strings:
    - Position 0: the program name
    - Position 1: the first argument

Example: `sys.argv`

Python script called `print_argv.py` :



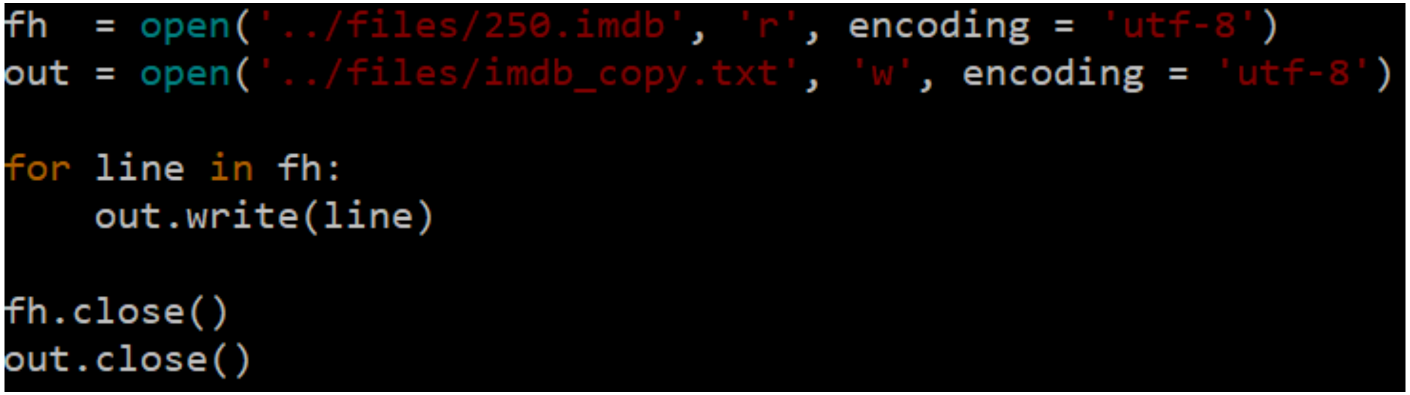
```
import sys

print(sys.argv)
```

Running the script with command line arguments as input:

```
In [47]: !python print_argv.py 250.imdb output_file
```

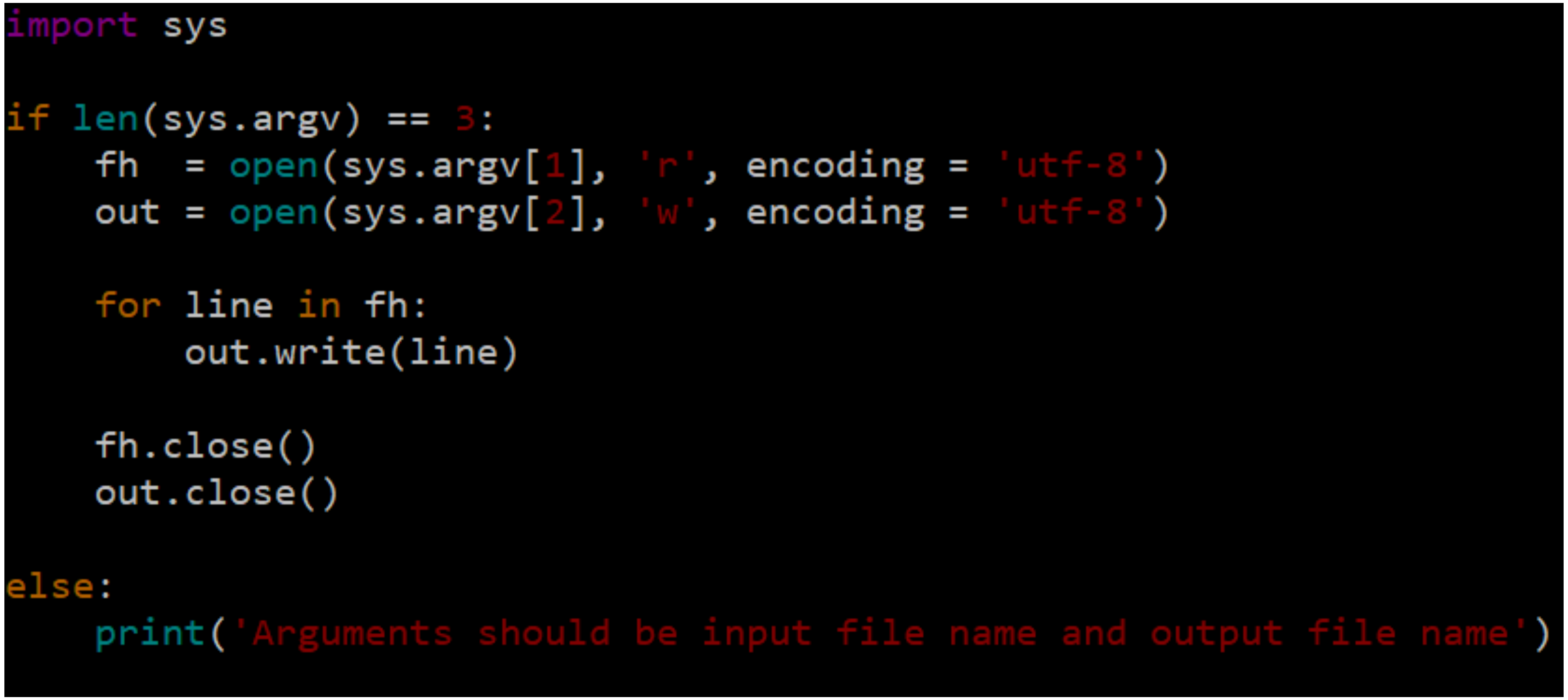
```
['print_argv.py', '250.imdb', 'output_file']
```

Example: copying 250.imdb to another file



```
fh  = open('../files/250.imdb', 'r', encoding = 'utf-8')
out = open('../files/imdb_copy.txt', 'w', encoding = 'utf-8')

for line in fh:
    out.write(line)

fh.close()
out.close()
```

Becomes: copying any file to any other file



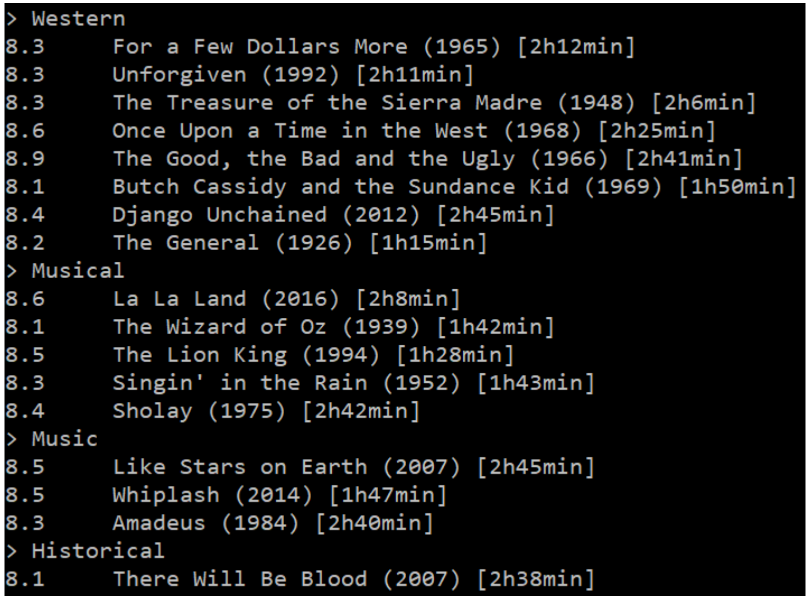
```
import sys

if len(sys.argv) == 3:
    fh  = open(sys.argv[1], 'r', encoding = 'utf-8')
    out = open(sys.argv[2], 'w', encoding = 'utf-8')

    for line in fh:
        out.write(line)

    fh.close()
    out.close()

else:
    print('Arguments should be input file name and output file name')
```

`argv[1]` -> first file

`argv[2]` -> second file

(yet another) IMDb exercise

Re-structure and write the output to a new file as below



```
> Western
8.3     For a Few Dollars More (1965) [2h12min]
8.3     Unforgiven (1992) [2h11min]
8.3     The Treasure of the Sierra Madre (1948) [2h6min]
8.6     Once Upon a Time in the West (1968) [2h25min]
8.9     The Good, the Bad and the Ugly (1966) [2h41min]
8.1     Butch Cassidy and the Sundance Kid (1969) [1h50min]
8.4     Django Unchained (2012) [2h45min]
8.2     The General (1926) [1h15min]
> Musical
8.6     La La Land (2016) [2h8min]
8.1     The Wizard of Oz (1939) [1h42min]
8.5     The Lion King (1994) [1h28min]
8.3     Singin' in the Rain (1952) [1h43min]
8.4     Sholay (1975) [2h42min]
> Music
8.5     Like Stars on Earth (2007) [2h45min]
8.5     Whiplash (2014) [1h47min]
8.3     Amadeus (1984) [2h40min]
> Historical
8.1     There Will Be Blood (2007) [2h38min]
```

Note:

- Use a text editor, not notebooks for this
- Use functions as much as possible
- Use `sys.argv` for input/output

Some tips:

- Use f-strings: f"{rating} ..."
- Use dictionaries