Heuristics

Lab Section 2

For students with Programming background

- Do these exercises and send me the code/output screenshots:
- filein.py
- fileinmapy.py
- fileinmapy2.py

Today’s Lab

- We will explore these exercises and concepts:
- zoo.py
- Dictionaries
- simpsons.py
- List comprehension
- Functions
- Map
- parsingInput.py
- Filing (file input and output)
- filein.py
- fileinmapy.py
- fileinmapy2.py

zoo.py

- Given the list of animals, take the animal name as input from the user and then delete the name from the list if it is present, if the given name is not present in the list, inform the user that the animal name is not present in the list

- From the remaining list, print these:
  - Length of the list
  - the first element/item of the array/list
  - the second through fourth elements/items of the array/list
  - the second to last element/item in the list

- Check out the (zoo.py) script.

height.py

- Get the user, gender and height information, and tell them if they are taller/shorter/equal to the average.
- The respective average heights for males and females are:
- Males: 175 cm, Females: 165 cm
- How many inputs do we need?
- You can only use strings.
- Checkout the (height.py) script.

Dictionaries

- A dictionary is a collection of key-value pairs
- Dictionaries are defined using {}
- Dictionaries are mutable

- Each element in the dictionary consists of a key and a value (key-value pair)
  - grades = {key: value, key: value, key: value}

- The keys and values in an element can be any type
  - keys are unique and must be an immutable data type:
    - int, float, string
  - values can be immutable or mutable data types
    - int, float, string
    - list, dict

Creating a Dictionary

- Dictionaries are created using curly brackets {}

```
grades = {   keys       values
            ‘Jon’:     9.5,
            ‘Robb’:    10,
            ‘Arya’:    10
           }
```

```
print(grades)      {‘Jon’: 9.5, ’Robb’: 10, ’Arya’: 10}
```

Creating a Dictionary

Dictionaries are mutable!

Dynamically creating a dictionary during runtime:

```
grades = {}    # create empty dictionary
grades[‘Jon’] = 9.5
grades[‘Robb’] = 10
grades[‘Arya’] = 10

print(grades)     {‘Jon’: 9.5, ’Robb’: 10, ’Arya’: 10}
```

Retrieving a Value

- Dictionaries are different from Lists:
  - You can not use numeric indices to access a value by its specific position
  - Instead, you need to use the key to retrieve a value
- dictionary_name[key] will retrieve the value associated with that key

value = dictionary_name[key]

Example: Lists and Dicts

```
student_list = [’Jon’, ’Robb’, ’Arya’]
student_dictionary = {’Jon’:10, ’Robb’:5, ’Arya’:9}

print(student_list)                      [’Jon’, ’Robb’, ’Arya’]
print(student_dictionary)                {’Arya’: 9, ’Robb’: 5, ’Jon’: 10}

print(student_list[0])                   ’Jon’
print(student_dictionary[0])             KeyError: 0
print(student_dictionary[’Jon’])         10
print(student_dictionary[’Cercei’])      KeyError: ‘Cercei’
```

How to Prevent Key Errors?

- Remember the in operator? It also works for dictionaries

```
if ‘Cercei’ in student_dictionary:
    print(student_dictionary[‘Cercei’])

if ‘Cercei’ not in student_dictionary:
    print(“Cercei is not found”)
```

Adding or Updating a key-value pair

- Adding a new key-value pair to a dictionary:
  - dictionary_name[key] = value

- Updating/overwriting a value in a dictionary:
  - dictionary_name[key] = new_value

Useful dictionary statements

- Deleting an key-value pair from a dictionary:
  - `del dictionary_name[key]`
- Number of key-value pairs in a dictionary:
  - `len(dictionary_name)`

Dictionary exercise

- Write a Python program that create/prints a dictionary where the keys are numbers between 1 and 15 and the values are the square of the keys.

- Expected output: print(my_dict)

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8:
64, 9: 81, 10: 100, 11: 121, 12: 144, 13:
169, 14: 196, 15: 225}
```

See: dictionary.py

simpsons.py

- Create a dictionary consisting of the main characters of the Simpsons with keys as character names and values as their relation or role they play in the family, for example, Homer is a father.
- Print the dictionary.
- If you have not added, “Santa”, add in the dictionary with the relation of pet/Dog.
- Print the Dictionary again.
- Add the information for Bart that he is not a good son, while also keeping the previous value/information about Bart.
- Ask the user to input any character name, if the name is correct, print whatever information you have on the character.

List comprehension

- Python's List Comprehension is a quick and compact syntax to generate a list from a string or any other list.

- Creating a new list by performing an operation on each item in the existing list is a very straightforward way.

- Comprehension of the list is considerably faster than using the for loop to process a list.

List comprehension - Syntax

- A list comprehension generally consist of these parts :
- Output expression, input sequence, a variable representing a member of the input sequence, and an optional predicate part.
- We could also have nested loops by having multiple input sequences and variables representing a member of those input sequences, but that is rarely used.

- For example :
  - list = [x ** 2 for x in range (1, 11) if x % 2 == 1]

  here, x ** 2 is the output expression,
  range (1, 11) is in the put sequence,
  x is variable and if x % 2 == 1 is predicate part.

Example: Separate Letters from String

```
• chars=[]
• for ch in 'GKTCS INNOVATIONS':
•     chars.append(ch)
• print(chars)


• chars = [ch for ch in 'GKTCS INNOVATIONS']
• print(chars)
```

Functions

- Functions are used to structure your code, improve code readability and make code reusable
- We have used functions already:
  - Built-in functions: like print(), len(), type(), etc
  - Module functions like random.randint(num1, num2), time.sleep(sec), etc
  - Methods are also functions
- We have seen that a all functions
  - have a name
  - have a set of arguments defined inside “( )”
  - can return a value (if not, None is returned)

Functions Types

- Two types of functions exist:
  - Void functions
  - Fruitful functions

- Void functions do not return a value
- Fruitful functions return a value

Functions Definition

- Syntax:

```
def function_name(arguments):
    # indented code block
    return value # return statement is optional
```

- Arguments:
  - A function can have as many arguments as desired, separated by commas
- The indented code block is executed when the function is called
- The function exits if the end of the indented code block is reached
  - or if a return statement is reached, if present

Function Definition

```
def print_hello_world():
    print(“Hello”, end=“ “)
    print(“World!”)
```

Function without arguments

Function body

Function Definition

```
def measure_distance(x1, y1, x2, y2):

    ‘’’This function is used to measure the
    distance between two points (x1,y1) and
    (x2,y2)
    Example: measureDistance(0,0,5,5)

    Precondition: all input arguments must
    be either an int or a float’’’

    distance = ((x2-x1)**2+(y2-y1)**2)**0.5

    return distance
```

← function with arguments

Specification (docstring)

Function body

Optional return statement

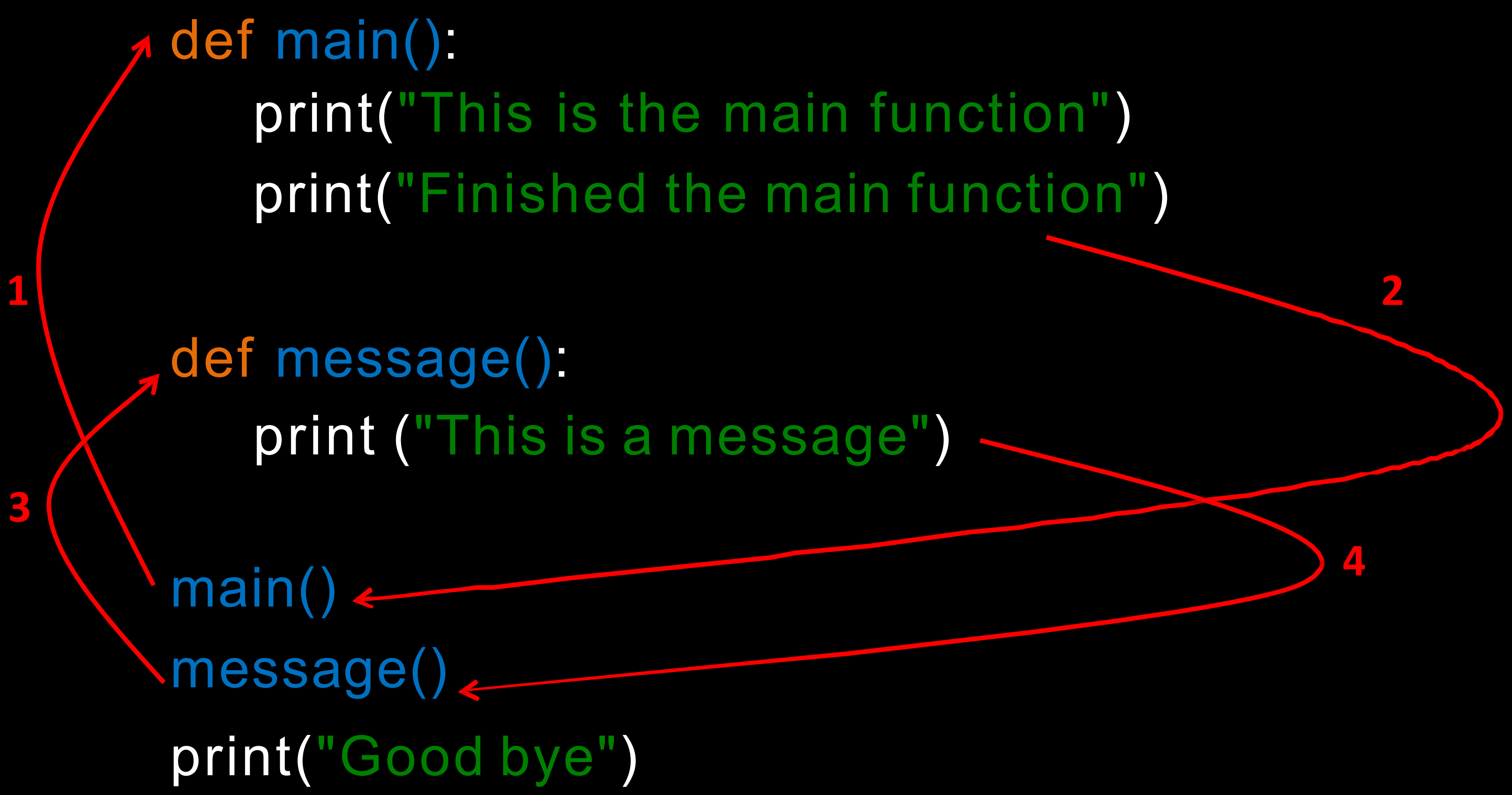
Functions Definition

- Function definition specifies what a function does:
  - It **does not** cause the function to execute
- If you want the function to execute you need to **call** the function:
  - You can call the function by writing the function name followed by “**( )**”

Functions Flow

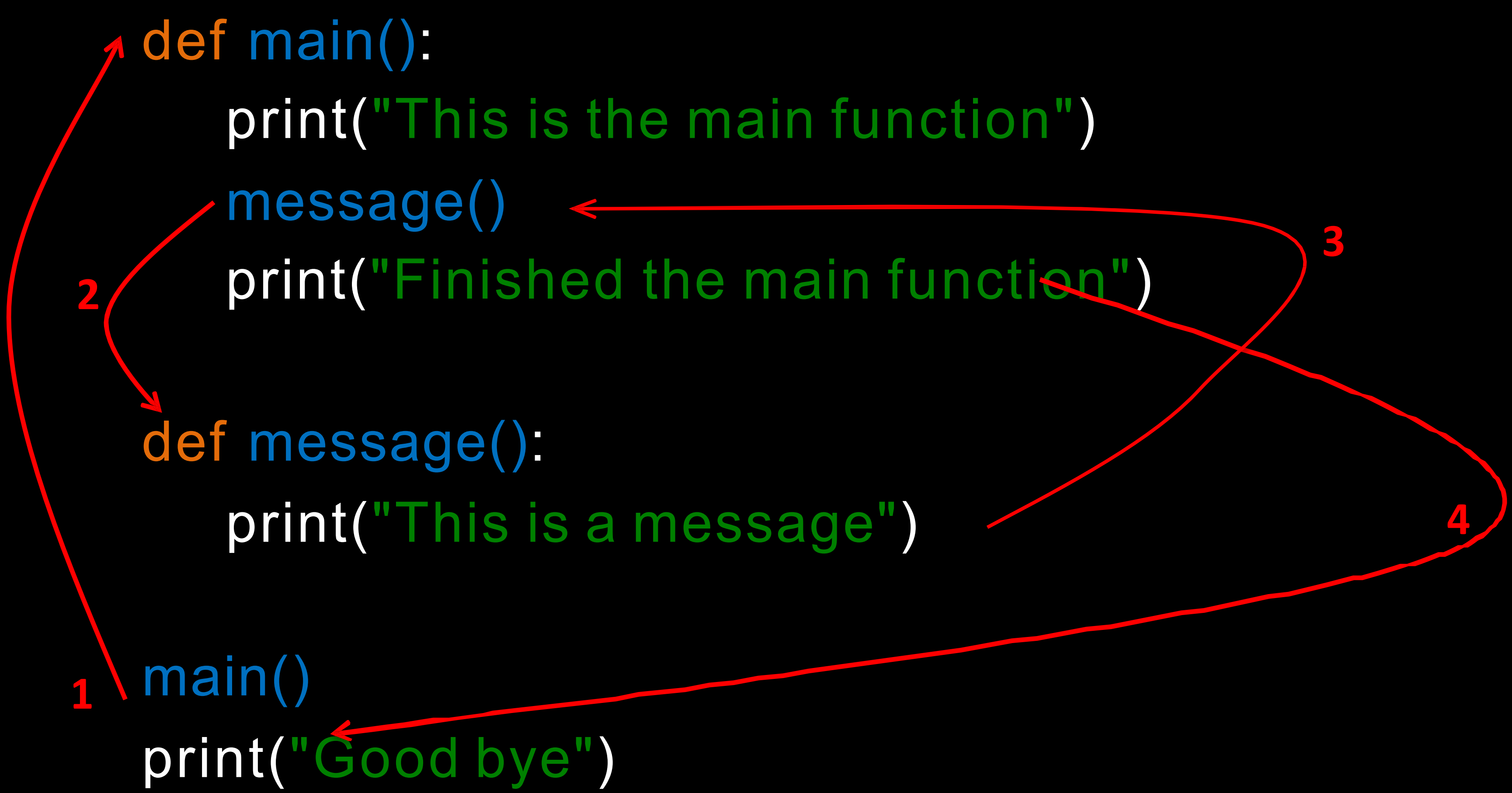
- When you write a program with functions, always add the function definition to the beginning of your program:
  - You cannot use functions before they are defined
- When you call a function, the Python interpreter
  - jumps to the function definition
  - executes the function body
  - jumps back to the line that called the function

Functions Flow



```
def main():
    print("This is the main function")
    print("Finished the main function")

def message():
    print ("This is a message")

main()
message()
print("Good bye")
```

Functions Flow



```
def main():
    print("This is the main function")
    message()
    print("Finished the main function")

def message():
    print("This is a message")

main()
print("Good bye")
```

Function Arguments

- Usually you want to send data into your function

  - So that the function can use them to do something
  - For example, if you want a function to check if a number is odd or even, you need to pass the number to the function

- Arguments are used to pass data to functions
  - You can define how many arguments the function takes
  - You also define the order of the arguments

Function Argument

```
def is_num_even_or_odd(number):
    if number%2 == 0:
        print("Number is even")
    else:
        print("Number is odd")

is_num_even_or_odd(3)     Number is odd
is_num_even_or_odd(40)    Number is even
```

Functions and Multiple Arguments

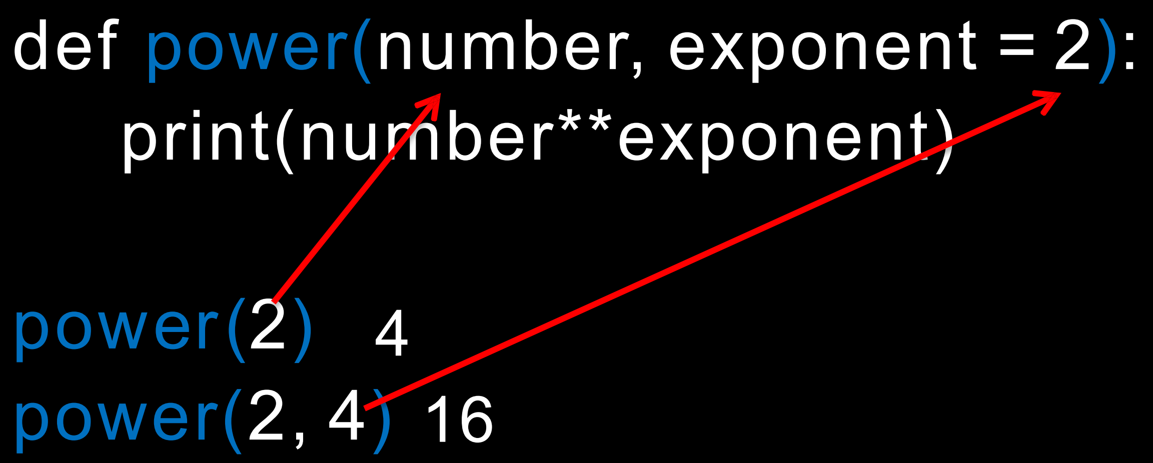
```
def power(number, exponent):
    print(number**exponent)


power(2, 4)  16
power(exponent=4, number=2)
```

Notice here the order of the arguments have changed by using explicitly the keyword argument

Default arguments

- Python allows function arguments to have default values
  - If the function is called without the argument, the argument gets its default value

- Example:



```
def power(number, exponent = 2):
     print(number**exponent)


power(2)   4
power(2, 4) 16
```

Functions and Local Variables

- If an immutable variable is assigned a value anywhere within the function's body, it is a local function variable
- A local function variable cannot be accessed outside the function they are defined in
- Consequently, you can have the same variable name in different functions

Local Variable Examples

```
def calc_square(value):
    number = value**2


calc_square(5)
print(number)
```

```
NameError: name
‘number’ is not defined
```

```
def calc_square(value):
    number = value**2


number = 5
calc_square(number)
print(number)
```

```
5
```

Variables Outside Functions

- Immutable variables that are defined outside functions
  - can be accessed inside a function
  - can not be changed inside a function

- If you use an assignment statement inside the function with the same variable name you are creating a local variable

Variables Outside Functions

```
def calc_square():
    print(number**2)  25

number = 5
calc_square()
print(number)    5
```

```
def calc_square():
    number = 10
    print(number**2)  100

number = 5
calc_square()
print(number)    5
```

Fruitful Functions

- There is a more elegant way:
  - Fruitful functions
- Fruitful functions are functions that return a value back to the caller using the return keyword
- Note: Functions can have multiple return statements, but the function exits after one is reached!

Local Variable vs. Fruitful Function

```
def calc_square(value):
    number = value**2
    return number

number = 5
result = calc_square(number)
print(result)        25
print(number)        5
```

Local Variable vs. Fruitful Function

```
def calc_square(value):
    number = value**2
    return number

number = 5
number = calc_square(number)
print(number)
```

```
25
```

Multiple Return Values

- What if the function should return multiple variables?

- Option 1:
  - return a, b, c
  - x, y, z = function()

- Option 2:
  - return [a, b, c]
  - values = function()

Functions as parameters

- There are cases where we want to pass an entire function as a parameter

- Python has functions as first-class citizens, so you can do this

- You simply pass the functions by name

Map- Higher functions

- A higher-order function is a function that takes another function as a parameter
- They are “higher-order” because it’s a function of a function
- Examples
  - Map
  - Reduce – will not use in this course
  - Filter – will not use in this course

Map

- `map(`**`function, iterable, ...`**`)`
- Map applies a function to each element of iterable and creates a list of the results
- You can optionally provide more iterables (sequences) as parameters to map and it will place tuples in the result list
- Map returns an iterator that can be cast to list

Map- Example

- Example:
- `def return_mod_of_5(x):`
- `    return x % 5`

- `nums = [0, 4, 7, 2, 1, 0 , 9 , 3, 5, 6, 8, 0, 3]`

- `num_new = list(map(return_mod_of_5, nums))`

- `print(num_new)`
- `#[0, 4, 2, 2, 1, 0, 4, 3, 0, 1, 3, 0, 3]`

parsingInput.py

- Get a list of numbers as a comma-separated string from the user and create the list using
- 1) simple for loop and list append method,
- 2) map, and
- 3) list comprehension.

- Check out the (parsingInput.py) script.

Files

- So far, we have only variables to store data

- Variables are stored in memory
  - Once the program is terminated, variables are cleared and their data is lost

- If we want to keep data even if the program terminates, the data has to be stored in a file

We use files every day

- We use files on a day-to-day basis to store different data in our computers
  - Image or videos
  - MS Word documents
  - Emails
  - Games
  - Homework assignments

- Most of the programs that we use store data in files

File Types

- Generally, two types of files:
  - binary: images, audio, videos, MS Word, executables, etc.
  - text: files that contain characters

- Text files store data as text using a certain coding scheme like ASCII
  - We can easily open it with any text editor

- Binary files store data as 0 and 1
  - The data stored are intended for the program and we can not open it in a text editor

File Types

- We will only focus on text files
- We will be able to open these files in any text editor and look at their content

Opening a File

- Syntax:

```
file_variable = open(filename, mode)
```

- open() function returns a file object
- filename is a string specifying the file to open
- mode specifies how to open the file
  - ‘w’ to open a file for writing
  - ‘a’ to open a file for writing and appending to it
  - ‘r’ to open a file for reading only (you can not write to the file)

‘w’ vs. ‘a’ Modes

- Both ‘w’ and ‘a’ modes open a file for writing

- ‘w’:
  - If the file exists, it will erase/overwrite its content
  - If the file does not exist, it will create it

- ‘a’:
  - If the file exists, it will keep its content and appends new data to the end of the file
  - If the file does not exist, it will create it

‘r’ Mode

- The ‘r’ mode opens the file in read-only mode
  - if the file does not exist, an error is thrown

```
input_file = open("my_file.txt", ‘a’)
input_file.close()
input_file = open("my_file.txt", ‘r’)

# read data from the file

input_file.close()
```

Workaround:
Creates a file if
it did not exist
and closes it

Don’t forget to close the file!

- When a program opens a file, you always have to close it at the end once you finished writing
- If you don’t close the file, you run the risk that your writings might not be recorded in the file
- To close the file, use the `.close()` method as:

```
file_variable.close()
```

Writing to a File

- After opening a file in write mode (‘w’ or ‘a’), use the .write() method to write to the file:

- Example:

```
output_file = open(‘my_file.txt’, ’w’)
output_file.write(“a”)
output_file.write(“b”)
output_file.write(“c”)

output_file.close()
```

my_file.txt

```
1 abc
```

Example: Writing to a File

```
output_file = open(‘my_file.txt’, ‘w’)

output_file.write(‘a\n’)
output_file.write(‘b\n’)
output_file.write(‘c\n’)

output_file.close()
```

my_file.txt

```
1 a
2 b
3 c
4
```

Reading from a File

- Everything that is read from a file is a string!
- There are three methods for reading content from a file:
  - .read():
    - reads the entire file at once
    - Use this with caution, because if the file is bigger than your memory space, you might run into problems
  - .readlines():
    - read all the lines at a single go and then return them as each line a string element in a list.
    - This function can be used for small files, as it reads the whole file content to the memory, then split it into separate lines. We can iterate over the list and strip the newline '\n' character using the strip() function.
  - .readline(): Reads a single line from the file, including the \n
    - Every subsequent call will read the next line

    Once you reach the last line it will return a ''

Example: Reading from a File

```
input_file = open(‘my_file.txt’, ‘r’)

file_content = input_file.read()
# file_content: ’a\nb\nc\n’

print(file_content)# ’a\nb\nc\n\n’
print(‘Good bye’)
input_file.close()
```

my_file.txt

```
1 a
2 b
3 c
4
```

Terminal output:

```
a
b
c

Good bye
```

The print() adds a \n

Example 2: Reading from a File

```
input_file = open(‘my_file.txt’, ‘r’)


file_content = input_file.readlines()
# file_content: [’a\n’, ‘b\n’, ‘c\n’]



print(file_content)
print(‘Good bye’)
input_file.close()
```

my_file.txt

```
a
b
c

```

Terminal output:

```
[’a\n’, ‘b\n’,
‘c\n’]


Good bye
```

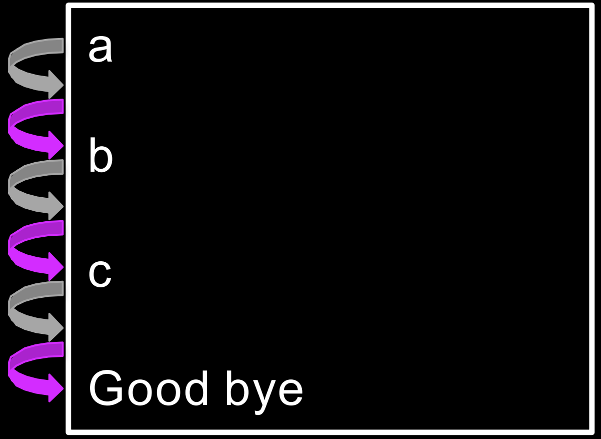
Example 3: Reading from a File

```
input_file = open(‘my_file.txt’, ‘r’)

line_1 = input_file.readline()
# line_1: ‘a\n’
line_2 = input_file.readline()
# line_2: ‘b\n’
line_3 = input_file.readline()
# line_3: ‘c\n’

input_file.close()

print(line_1) # ‘a\n\n’
print(line_2) # ‘b\n\n’
print(line_3) # ‘c\n\n’
print(‘Good bye’)
```

my_file.txt

```
1 a
2 b
3 c
4
```

Terminal output:



```
a

b

c

Good bye
```

String strip() Method

- The .strip() method returns a copy of the string with both leading and trailing characters removed, i.e. “\n”, “ “, etc
- Example:

```
string = “      Hello World!            \n”
 string = string.strip()
 print(string)          “Hello World!”
```

Example 3: Removing the \n?

```
input_file = open(‘my_file.txt’, ‘r’)

line_1 = input_file.readline().strip() # ‘a’
line_2 = input_file.readline().strip() # ‘b’
line_3 = input_file.readline().strip() # ‘c’

input_file.close()

print(line_1) # ‘a\n’
print(line_2) # ‘b\n’
print(line_3) # ‘c\n’
print(‘Good bye’)
```

my_file.txt

```
1 a
2 b
3 c
4
```

Terminal output:

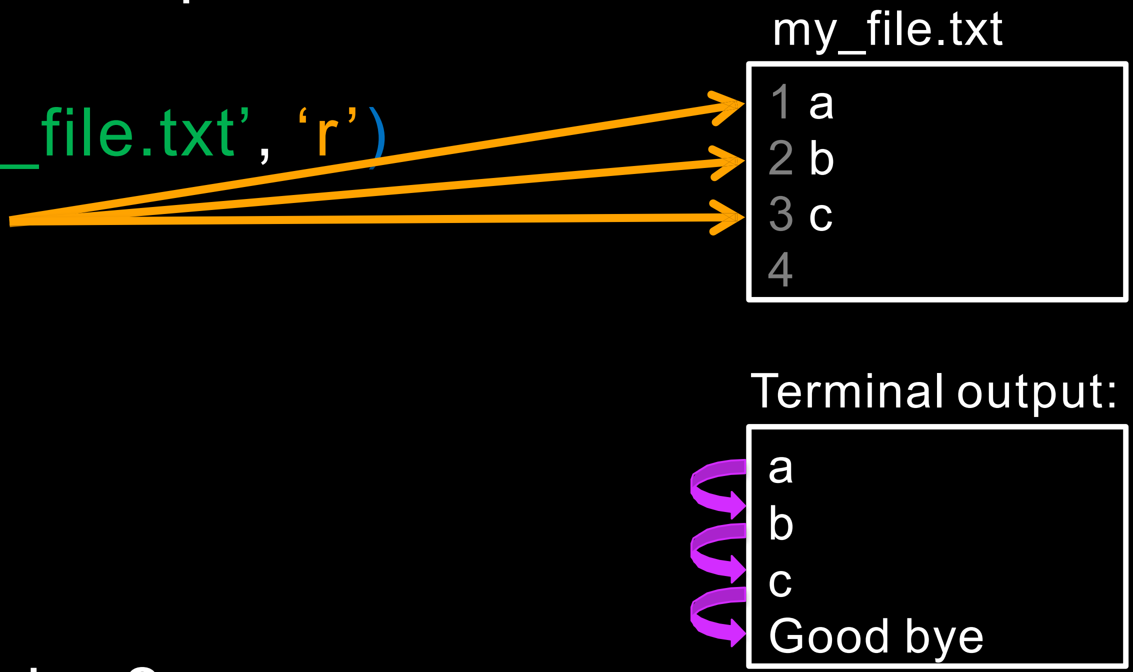
```
a
b
c
Good bye
```

Looping through Files

- Remember that for-loops can loop through sequences!?
- A file object is also a sequence!

```
input_file = open(‘my_file.txt’, ‘r’)
for line in input_file:
    print(line.strip())

input_file.close()
print(‘Good bye’)
```

Is the .readline() missing?



my_file.txt

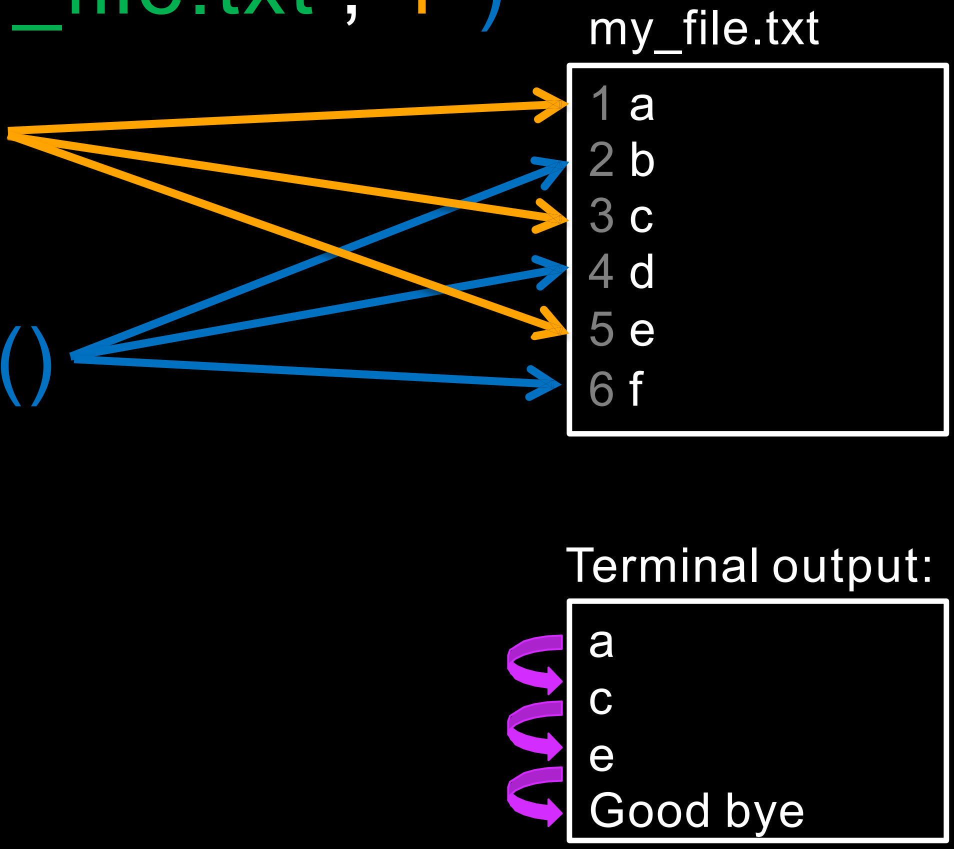
```
a
b
c

```

Terminal output:

```
a
b
c
Good bye
```

Looping through Files

```
input_file = open(‘my_file.txt’, ‘r’)
for line in input_file:
    print(line.strip())
    input_file.readline()

input_file.close()
print(‘Good bye’)
```



my_file.txt

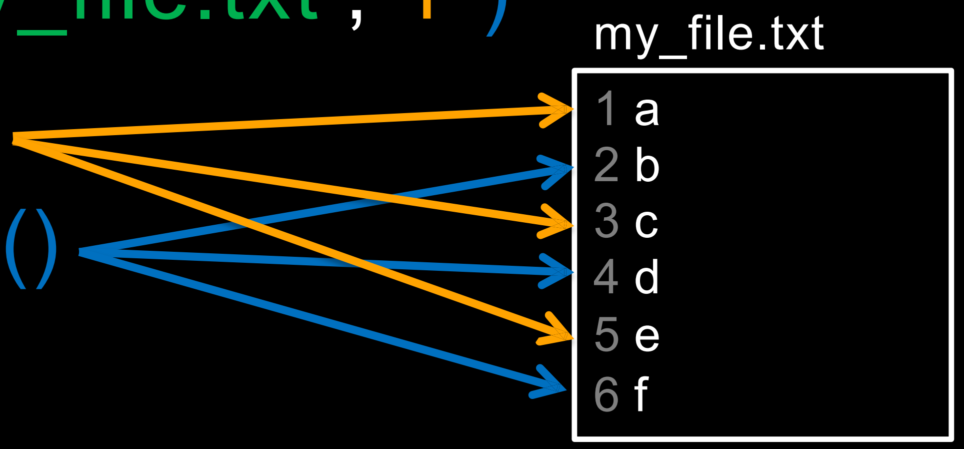
```
1 a
2 b
3 c
4 d
5 e
6 f
```

Terminal output:

```
a
c
e
Good bye
```

Looping through Files

```
input_file = open(‘my_file.txt’, ‘r’)
for line in input_file:
    input_file.readline()
    print(line.strip())

input_file.close()
print(‘Good bye’)
```



my_file.txt

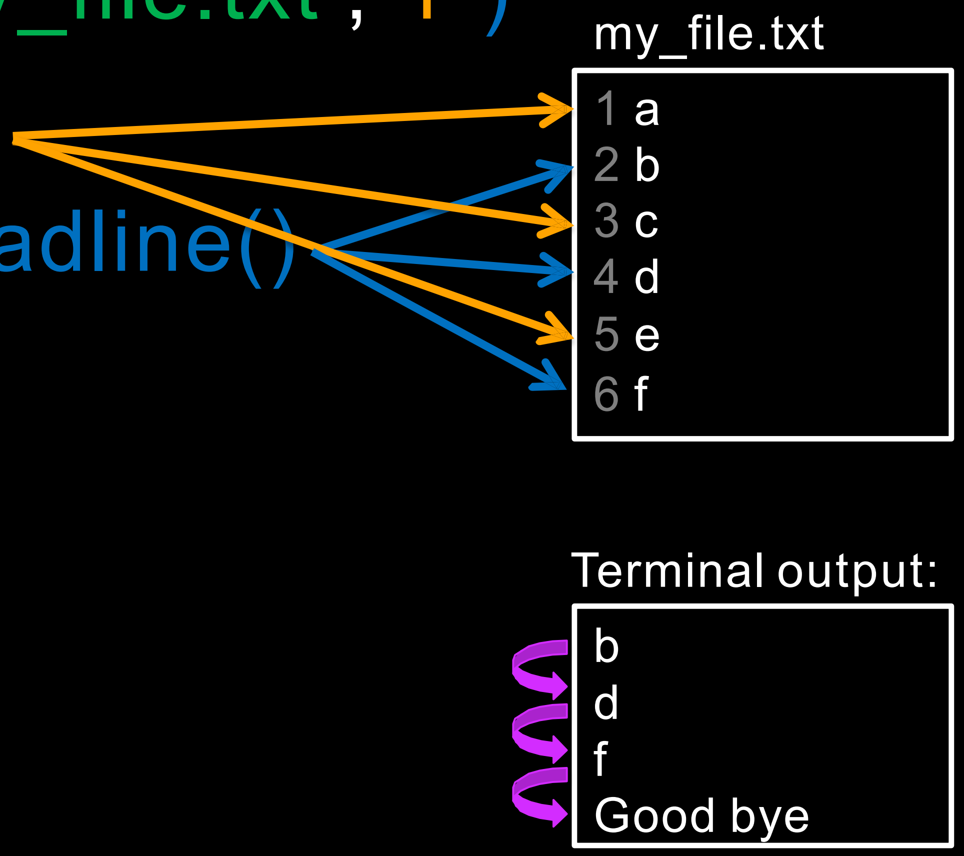
```
1 a
2 b
3 c
4 d
5 e
6 f
```

Terminal output:

```
a
c
e
Good bye
```

Looping through Files

```
input_file = open(‘my_file.txt’, ‘r’)
for line in input_file:
    line = input_file.readline()
    print(line.strip())

input_file.close()
print(‘Good bye’)
```



my_file.txt

```
1 a
2 b
3 c
4 d
5 e
6 f
```

Terminal output:

```
b
d
f
Good bye
```

filein.py

- Read the ‘preamble’ file that contains some text and perform these tasks:
  - Read lines in a list
  - Iterate over the list and :
  - Print the length of the line and the line without ending “’\n”
  - Split the line into words, print the list containing the first word, and print the first word in string form.
  - Instead of splitting on space ‘ ’, split on letter ‘e’ and do the last task of printing the list containing the first word and also print the first word in string form.
  - Expected output: for the first line, the output will be:
    - 21 We the people of the
    - ['We', 'the', 'people', 'of', 'the']
    - We
    - ['W', ' th', ' p', 'opl', ' of th', '']
    - W

fileinmap.py

- Read the ‘ourownfile’ file that contains the Invictus poem and perform these tasks:
  - Read lines in a list
  - Iterate over the list and :
  - Print the length of the line and the line without ending ‘’\n”
  - Split the line into words and
  - For each word in the line, create a new word which is a repetition of the original word separated by ‘_’, happy -> happy_happy
  - Do the above functionality using the map
  - Expected output: for the first line, the output will be:
    - 9 Invictus
    - Invictus_Invictus

fileinmap2.py

- Read the ‘ourownfile’ file that contains the Invictus poem and perform these tasks:
  - Read lines in a list
  - Iterate over the list and :
  - Print the length of the line and the line without ending “’\n”
  - Split the line into words and
  - Using words in the line, create pairs separated by ‘_’, where the first word in the pair precedes the second word in the actual sentence, ‘trying my best’ -> ‘trying_my’ , ‘my_best’
  - Do the above functionality using the map and list comprehension
  - Expected output: for the third line, the output will be:
    - 32 Out of the night that covers me
    - Out_of
    - of_the
    - the_night
    - night_that
    - that_covers
    - covers_me

End