Session Overview – Day 3

- Data Structures
  - Strings
  - Tuple

Data Structure in Python

- Dynamic way of organizing data in memory
- Some of the Data Structures available in Python are:
  - Strings
  - List
  - Tuples
  - Sets
  - Dictionaries

Topics Overview – Day 4

- Lists
- Set
- Dictionary

Lists

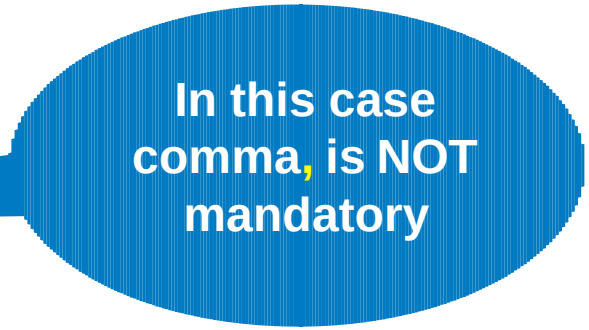
- An ordered group of sequences enclosed inside square brackets and separated by symbol ,
- Lists are **mutable.**

Syntax:

list1 = [ ] #Creation of empty List

list2 = [Sequence1,]

list3 = [Sequence1, Sequence2]



Examples:

```
language = ['Python']
languages = ['Python','C','C++','Java','Perl']
```

List Notation and Examples

List Example	Description
[ ]	An empty list
[1, 3, 7, 8, 9, 9]	A list of integers
[7575, “Shyam”, 25067.56]	A list of mixed data types
[“Bangalore”, “Bhubaneshwar”, “Chandigarh”, “Chennai”, “Hyderabad”, “Mangalore”, “Mysore”, “Pune”, “Trivandrum”]	A list of Strings
[[7575, “John”, 25067.56], [7531, “Joe”, 56023.2], [7821, “Jill”, 43565.23]]	A nested list
[“India”, [“Karnataka”, [“Mysore”, [GEC1, GEC2]]]]	A deeply nested list

Basic List Operations

Python Expression	Result	Operation
len([4, 5, 6])	3	Length
[1, 3, 7] + [8, 9, 9]	[1, 3, 7, 8, 9, 9]	Concatenation
['Hello'] * 4	['Hello' , 'Hello' , 'Hello' , 'Hello']	Repetition
7 in [1, 3, 7]	True	Membership
for n in [1, 3, 7] : print(n)	1 3 7	Iteration
n = [1, 3, 7] print(n[2])	7	Indexing: Offset starts at 0
n = [1, 3, 7] print(n[-2])	3	Negative slicing: Count from right
n = [1, 3, 7] print(n[1:])	[3, 7]	Slicing (from position 1 to end)

Basic List Operations

S.No.	Methods with Description
1	list.append(obj) Appends object obj to list
2	list.count(obj) Returns count of how many times obj occurs in list
3	list.extend(seq) Appends the contents of seq to list
4	list.index(obj) Returns the lowest index in list that obj appears
5	list.insert(index, obj) Inserts object obj into list at offset index

Basic List Operations

6	**list.pop()** Removes and returns last object or obj from list
7	list.remove(obj) Removes object obj from list
8	list.reverse() Reverses objects of list in place
9	list.sort() Sorts objects of list

Sets

- An un-ordered collection of unique elements
- Are lists with no index value and no duplicate entries
- Can be used to identify unique words used in a paragraph

Syntax:

```
set1  =  { }          #Creation of empty set

 set2  =  {“John”}          #Set with an element
```

Example:

```
s1 = set("my name is John and John is my name".split())

s1 = {'is', 'and', 'my', 'name', 'John'}
```

- Operations like intersection, difference, union, etc can be performed on sets

Operations on Sets

Operation	Equivalent	Operation
len (s)		Length of set ‘s’
x in s		Membership of ‘x’ in ‘s’
x not in s		Membership of ‘x’ not in ‘s’
s.issubset(t)	**s <= t**	Check whether ‘s’ is subset ‘t’
s.issuperset(t)	**s >= t**	Check whether ‘t’ is superset of ‘s’
s.union(t)	**s | t**	Union of sets ‘s’ and ‘t’
s.intersection(t)	**s & t**	Intersection of sets ‘s’ and ‘t’
s.difference(t)	**s - t**	Returns elements in ‘s’ but not in ‘t’
s.symmetric_difference(t)	**s ^ t**	Returns elements in either ‘s’ or ‘t’ but not both

Dictionary

- A list of elements with key and value pairs(separated by symbol :) inside curly braces.
- Keys are used instead of indexes
- Keys are used to access elements in dictionary and keys can be of type – strings, number, list, etc
- Dictionaries are mutable, i.e it is possible to add, modify and delete key-value pairs

Syntax:

```
phonebook  =  {}            #Creation of empty Dictionary
phonebook={“John”:938477565} #Dictionary with one key-value pair
phonebook={“John”:938477565, “Jill”:938547565} #2 key-value pairs
```

Dictionary

1	**dict.clear()** **Removes all elements of dictionary dict**
2	dict.copy() Returns a shallow copy of dictionary dict
3	dict.get(key, default=None) For key key, returns value or default if key not in dictionary
4	dict.items() Returns a list of dict's (key, value) tuple pairs
5	dict.keys() Returns list of dictionary dict's keys
6	dict.values() Returns list of dictionary dict's values

Mutable v/s Immutable Data Types

Mutable Data Type	Immutable Data Type
Sequences can be modified after creation	Sequences cannot be modified after creation
Ex: Lists, Sets, Dictionary	Ex: Strings, Tuples
Operations like add, delete and update can be performed	Operations like add, delete and update cannot be performed

Strings

- Accepts 3 types of quotes to assign a string to a variable.
  - single ('), double (") and triple (''' or """)
  - String starts and ends with same type of quote
  - Triple quotes are used to span string across multiple lines.
- Index starts from zero.
- Can be accessed using negative indices. Last character will start with -1 and traverses from right to left.

Syntax:

```
word = ‘Programming’

sentence = “Object Oriented Programming."

paragraph = """ Python is a Object Oriented Programming Language.
It is a Beginner’s language."""
```

String Operators and Functions

- **Concatenation**
  - Strings can be concatenated with '+' operator
    - "Hello" + "World" will result in HelloWorld
- **Repetition**
  - Repeated concatenation of string can be done using asterisk operator "*"
    - "Hello" * 3 will result in *HelloHelloHello*
- **Indexing**
  - "Python"[0] will result in "P"
- **Slicing**
  - Substrings are created using two indices in a square bracket separated by a ':'
    - "Python"[2:4] will result in "th"
- **Size**
  - prints length of string
    - len("Python") will result in 6

String Methods

str1="welcome to python programming"

```
str1.upper()
str1.lower()
str1.count()
str1.find(“p”)
str1.replace(“r”,”s”)
len(str1)
str1.index("o")
```

String Methods

str1="welcome to python programming"

```
str1.count("a")
str1[3:7]
str1[3:7:2]
str1[::-1]
str1.startswith("Hello")
str1.endswith(“programming")
str1.split(" ")
```

Tuples

- An ordered group of sequences separated by , symbol and enclosed inside the parenthesis
- Tuples are **immutable.**

Syntax:

```
tuple1 = () #Creation of empty tuple
tuple2 = (Sequence1,)
tuple3 = (Sequence1, Sequence2)
```

, symbol is mandatory without which it becomes just a string assignment operation

Examples:

```
customer = (“John”,)
customers = (‘John’, ’Joe’, ’Jack’, ’Jill’, ’Harry’)
```

Tuple Operations

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );

print (tup1[0])
print (tup2[1:5])
```

```
physics
[2, 3, 4, 5]
```

Tuple Operations

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

tup1[0] = 100;

tup3 = tup1 + tup2;
print (tup3)

(12, 34.56, 'abc', 'xyz')
```

Tuple Operations

```
tup = ('physics', 'chemistry', 1997, 2000);

print (tup)

del tup;
print ("After deleting tup : “)
print (tup)
```

Tuples Operations

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Basic Tuples Operations

L = ('spam', 'Spam', 'SPAM!')

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in Tuple Functions

- ✓cmp(tuple1, tuple2) - Compares elements of both tuples.
- ✓len(tuple) - Gives the total length of the tuple.
- ✓max(tuple) - Returns item from the tuple with max value.
- ✓min(tuple) - Returns item from the tuple with min value.

Topics Overview – Day 5

- Day 5
  - Functions
  - Different types of arguments
  - Recursion

Functions

- Are blocks of organized, reusable code used to perform single or related set of actions
- Provide better modularity and high degree of reusability
- Python supports:
  - Built-in functions like print() and
  - User - defined functions

Functions (Cont…)

- **Defining a function:**
  - Function blocks starts with a keyword '**def'** followed by **function_name,** parenthesis **(())** **and a colon :**
  - Arguments are placed inside these parenthesis
  - Function block can have optional statement/comment for documentation as its first line
  - Every line inside code block **is indented**
  - **return [expression] statement exits the function** by returning an expression to the caller function.
  - return statement with no expression is same as return None.

Syntax:

```
def function_name( parameters ):
        "—optional: Any print statement for
        documentation"  function_suite
        return [expression]
```

  - Parameters exhibit positional behavior, hence should be passed in the same order as in function definition

Functions (Cont…)

- **Calling a Function**
  - Defining a function gives it a name, specifies function parameters and structures the blocks of code.
  - Functions are invoked by a function call statement/code which may be part of another function

- ***Example:***

```
# Defining function print_str(str1)
def print_str(str1):
    print("This function prints string passed as an argument")
    print(str1)
    return

# Calling user-defined function
print_str("Calling the user defined function print_str(str1)")
```

Observe the usage of optional print statement for documentation

Function Call

- ***Output:***

```
This function prints string passed as an argument
Calling the user defined function print_str(str1)
```

Functions (Cont…)

- **Pass arguments to functions:**
  - Arguments are passed by reference in Python
  - Any change made to parameter passed by reference in the called function will reflect in the calling function based on whether **data type of argument passed** is **mutable** or **immutable**
  - In Python
    - **Mutable Data types** include Lists, Sets, Dictionary
    - **Immutable Data types** include Number, Strings, Tuples

Functions (Cont…)

- **Pass arguments to functions: Immutable Data Type - Number**

Example:

```
#Function Definition
def change(cust_id):
    cust_id += 1
    print("Customer Id in function definition: ", cust_id)
    return

# Function Invocation with arguments of immutable data  type
cust_id = 100
print("Customer Id before function invocation: ", cust_id)
change(cust_id)
print("Customer Id after function invocation: ", cust_id)
```

Output:

```
Customer Id before function invocation: 100
Customer Id in function definition: 101
Customer Id after function invocation:  100
```

Functions (Cont…)

- **Pass arguments to functions: Mutable Data Type - List**

Example:

```
#Function Definition
def change(list_cust_id):
    #Assign new values inside the function
    list_cust_id.append([10, 20, 30])
    print("Customer Id in function definition: ", list_cust_id)
    return

# Function Invocation with arguments of immutable data type
list_cust_id = [100, 101, 102]
print("List of Customer Id before function invocation: ", list_cust_id)
change(list_cust_id)
print("Customer Id after function invocation: ", list_cust_id)
```

Output:

```
List of Customer Id before function invocation:  [100, 101, 102]
List of Customer Id in function definition:  [100, 101, 102, [10, 20, 30]]
List of Customer Id after function invocation:  [100, 101, 102, [10, 20, 30]]
```

Functions (Cont…)

- Different types of formal arguments:
  - Required arguments
  - Keyword arguments
  - Default arguments
  - Variable – length arguments

Functions (Cont…)

- **Required arguments**
  - Arguments follow positional order
  - No. of arguments and the order of arguments in the function call should be exactly same as that in function definition

Example:

```
# Function Definition
def print_str(str1):
    print("This function prints the string passed as an argument")
    print(str1)
    return

# Function Invocation without required arguments
print_str()
```

Output:

```
TypeError: print_str() missing 1 required positional argument: 'str1'
```

Functions (Cont…)

- **Keyword arguments**
  - when used in function call, the calling function identifies the argument by parameter name
  - Allows you to skip arguments or place them out of order
  - Python Interpreter uses the keyword provided to match the values with parameters

Example:

```
# Function Definition
def customer_details (cust_id, cust_name):
    print("This function prints Customer details")
    print("Customer Id: ",cust_id)
    print("Customer Name: ",cust_name)
    return

# Function Invocation with Keyword arguments
customer_details(cust_name = "John", cust_id = 101)
```

Output:

```
This function prints Customer details
Customer Id:  101
Customer Name:  John
```

Observe the c
positional o
argume

Functions (Cont…)

- **Default Arguments:**
  - Assumes a default value if the value is not specified for that argument in the function call

Example:

```
# Function Definition
def customer_details (cust_name, cust_age = 30):
    print("This function prints Customer details")
    print("Customer Name: ",cust_name)
    print("Customer Age: ",cust_age)
    return

# Function Invocation with Default arguments
customer_details(cust_age = 25, cust_name = "John")
customer_details(cust_name = "John")
```

Output:

```
This function prints Customer details
Customer Name:  John
Customer Age:  25
This function prints Customer details
Customer Name:  John
Customer Age:  30
```

Observe the usa
default value for cu
argument

Functions (Cont…)

- **Variable-length arguments**
  - Used to execute functions with more arguments than specified during function definition
  - unlike required and default arguments, variable arguments are not named while defining a function

Syntax:

```
def functionname([formal_args,] *var_args_tuple ):  "—
   optional: Any print statement for documentation"
   function_suite
   return [expression]
```

  - An asterisk '*' is placed before variable name to hold all non-keyword variable arguments
  - ***var_args_tuple** is empty if no additional arguments are specified during function call

Functions (Cont…)

- **Variable-length arguments**

Example:

```
# Function Definition
def customer_details (cust_name, *var_tuple):
    print("This function prints Customer Names")
    print("Customer Name: ",cust_name)
    for var in var_tuple:
        print(var)
    return

# Function Invocation with Variable length arguments
customer_details("John", "Joy", "Jim", "Harry")
customer_details("Mary")
```

Output:

```
This function prints Customer Names
Customer Name:  John
Joy
Jim
Harry
This function prints Customer Names
Customer Name:  Mary
```

Functions (Cont…)

- Scope of variables
  - Determines accessibility of a variable at various portions of the program
- Different types of variables
  - **Local variables**
    - Variables defined inside the function have local scope
    - Can be accessed only inside the function in which it is defined
  - **Global variables**
    - Variables defined outside the function have global scope
    - Variables can be accessed throughout the program by all other functions as well

Example:

```
total = 0

# Function Definition
def add( arg1, arg2 ):
    # Add both the parameters and return total
    total = arg1 + arg2; # total is local variable
    print ("Value of Total(Local Variable): ", total)
    return total;

# Function Invocation
add( 25, 12 );
print("Value of Total(Global Variable): ", total)
```

Output:

```
Value of Total(Local Variable): 37
Value of Total(Global Variable):  0
```

Usage of keyword ‘Global’

- Used to access the variable outside the function

Example:

```
total = 0

# Function Definition
def add( arg1, arg2 ):
    # Add both the parameters and return total
    global total
    total = arg1 + arg2; # Here total is made global variable
    print ("Value of Total(inside the function): ", total)
    return total;

# Function Invocation
add( 25, 12 );
print("Value of Total(outside the function): ", total)
```

Observe the usage of keyword ‘global’

Output:

```
Value of Total(inside the function): 37
Value of Total(outside the function):  37
```

Variable total is accessible outside the function

Recursion

- A method invoking itself is referred to as Recursion
- Typically, when a program employs recursion the function invokes itself with a smaller argument
- Computing factorial(5) involves computing factorial(4), computing factorial(4) involves computing factorial(3) and so on
- Often results in compact representation of certain types of logic and is used as substitute for iteration

Topics covered in Day 6

- Standard Library
- Math module
- List module
- Date & Time module

Math Module

- Provides access to mathematical functions like power, logarithmic, trigonometric, hyperbolic, angular conversion, constants etc;
- Few functions are described below:

Function	Description
abs(x)	Absolute value of x: the (positive) distance between x and zero
ceil(x)	Ceiling of x: smallest integer not less than x
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y
exp(x)	Exponential of x: e^x
floor(x)	Floor of x: the largest integer not greater than x
max(x1, x2,...)	Largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	Smallest of its arguments: the value closest to negative infinity
pow(x, y)	Value of x**y
round(x [,n])	x rounded to n digits from the decimal point.
sqrt(x)	Square root of x for x > 0

String Module

- Includes built-in methods to manipulate strings. Consider the string, str = KCE

Method	Result	Description
str.count(“s”)	Returns count of occurrence of character “s” in string str	2
str.startswith(“s”)	Returns true if string str starts with character “s”	false
str.endswith(“s”)	Returns true if string str ends with character “s”	true
str.find(“s”)	Returns index position of character “s” in string str if found else -1	4
str.replace(“s”, “S”)	Replaces all occurrences of character “s” with character “S” in string str	KCE
str.isdigit()	Checks if all the characters in string str are digits and returns true or false accordingly	false
str.upper()	Converts all the characters in string str to uppercase	KCE
str.lower()	Converts all the characters in string str to lowercase	KCE

List module

- Built-in functions and methods in lists

Function	Description
cmp(list1, list2)	Compares elements of both lists
len(list)	Gives total length of list
max(list)	Returns item from the list with maximum value
min(list)	Returns item from the list with minimum value
list(seq)	Converts a tuple to list
list.append(obj)	Appends object obj to list
list.count(obj)	Returns count of how many times obj occurs in list
list.insert(index, obj)	Inserts object obj into list at offset index
obj = list.pop()	Removes the item at position -1 from list and assigns it to obj
list.remove(obj)	Removes object obj from list
list.reverse()	Reverses the order of items in list
sorted(list)	Sorts items in list

Date and Time Module

- Supplies classes for manipulating dates and times in both simple and complex ways.
- Import time module.Ex: Print(time.localtime())

Function	Description
time.clock()	Returns current time in seconds, given as a floating point number
time.gmtime()	Returns current UTC date and time (not affected by timezone)
time.localtime()	Returns time based on the current locality (is affected by timezone)
time.timezone()	Returns the number of hours difference between your tomezone and the UTC time zone (London)
time.time()	Returns the number of seconds since January 1st 1970.
time.sleep(secs)	Suspends execution of the current thread for the given number of seconds
time.daylight()	Returns 0 if you are not currently in Daylight Savings Time

Topics covered in Day 7

- File Operations
- Exception handling

File operations

- A file is a chunk of logically related data or information which can be used by computer programs.
- Python provides some basic functions to manipulate files
- ***open*** Function

  Syntax: **file object = open(file_name [, access_mode][, buffering])**

- ***close*** Function

  Syntax: **fileObject.close()**

- ***write*** Function

  Syntax: **fileObject.write(string)**

- ***read*** Function

  Syntax: **fileObject.read([count])**

open Function

- Used to open or create a file

  Syntax: **file object = open(file_name [, access_mode][, buffering])**

  - ***file_name:*** Name of the file that you want to access
  - ***access_mode:*** determines the mode in which the file is to be opened, like read, write, append, etc
  - ***buffering:*** If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file.

- Once a file is opened following list of attributes can get information related to file

Attribute	Returns (Description)
file.close	true if file is closed, false otherwise.
file.mode	access mode with which file was opened
file.name	name of the file.

open Function…

Example:

```
# Open a file
result = open("foo.txt", "w")
print("Name of the file: ", result.name)
print("Closed or not : ", result.closed)
print("Opening mode : ", result.mode)
```

w – write permission

r – only read permission

a – append to existing content

r+ - read and write

Output:

```
Name of the file: foo.txt
Closed or not : False
Opening mode :  w
```

close() Function

- flushes any unwritten information and closes the file object, after which no more writing can be done.

Syntax: **fileObject.close()**

Example:

```
# Open a file
result = open("foo.txt", "w")
print("Name of the file: ", result.name)

# Close opened file
result.close()
```

Output:

```
Name of the file:  foo.txt
```

write Function

- Writes any string to an open file.
- Strings can have binary data or text data.
- Does not add a newline character ('\n') to the end of the string

Syntax: **fileObject.write(string)**

Example:

```
# Open a file
result = open("foo.txt", "w")
result.write( "Python is a great language.\nYeah its great!!\n");

# Close opened file
result.close()
```

Output:

```
Contents of file foo.txt:
Python is a great language.
Yeah its great!!
```

read Function

- Reads a string from an open file.

Syntax: **fileObject.read([count])**

- Parameter ‘count’ is the number of bytes to be read from the opened file.
- It starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

Example:

```
# Open a file
result = open("foo.txt", "r+")
sentence = result.read(10);
print ("Read String is : ", sentence)

# Close opened file
result.close()
```

Output:

```
Read String is :  Python is
```

Errors and Exception Handling

Errors and Exception handling

- Used to handle any unexpected error in Python programs
- Few Standard Exceptions:

Exception Name	Description
Exception	Base class for all exceptions
Arithmetic Error	Base class for all errors that occur for numeric calculation
Floating Point Error	Raised when a floating point calculation fails.
Zero Division Error	Raised when division or modulo by zero takes place for all numeric types.
IO Error	Raised when an input / output operation fails, such as print() or open() functions when trying to open a file that does not exist.
Syntax error	Raised when there is a error on Python syntax
Indentation error	Raised when indentation is not specified properly
Value Error	Raised when built-in-function for a data type has a valid type of arguments, but the arguments have invalid values specified
Runtime Error	Raised when a generated error does not fall into any category

Handling an Exception

- Exception is an event, which occurs during the execution of program and disrupts the normal flow of program's instructions.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a ***try:*** block.
- After the ***try:*** block, include an ***except:*** statement, followed by a block of code which handles the problem as elegantly as possible.
- Different ways of Exception Handling in Python are:
  - ***try….except…else***
  - ***try…except***
  - ***try…finally***

Handling an Exception…

- ***try…except…else***
  - A single try statement can have multiple except statements
  - Useful when we have a try block that may throw different types of exceptio
  - Code in else-block executes if the code in the try: block does not raise an ception

Try to open the same file when you do not have write permission, it raises an exception

Syntax:

```
try:
   You do your operations here;
   ......................
except ExceptionA:
   If there is ExceptionA, then execute this
block.
except ExceptionB:
   If there is ExceptionB, then execute this
block.
   ......................
else:
   If there is no exception then execute
this block
```

Example:

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file")
except IOError:
    print ("Error: can\'t find file or read data")
else:
    print ("Written content to file successfully")
    fh.close()
```

Output:

```
Written content to file successfully
```

Handling an Exception…

- ***try…except..***
  - Catches all exceptions that occur
  - It is not considered as good programming practice though it catches all exceptions as it does help the programmer in identifying the root cause of the problem that may occur.

Syntax:

```
try:
You do your operations here;
   ......................
except ExceptionA:
   If there is ExceptionA, then execute this
block.
except ExceptionB:
   If there is ExceptionB, then execute this
block.
   ......................
```

Try to write to the file when you do not have write permission, it raises an exception

Example:

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception
handling!!")
except IOError:
    print ("Error: can\'t find file or write data")
```

Output:

```
Error: can't find file or write data
```

Handling an Exception…

- ***try…finally..***
  - finally block is a place to put any code that must execute irrespective of try-block raised an exception or not.
  - else block can be used with finally block

Syntax:

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be
skipped.
finally:
   This would always be executed.
   ......................
```

Try to write to the file when you do not have write permission

Example:

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception
handling!!")
finally:
    print("Error: can\'t find file or write data")
```

Output:

```
Error: can't find file or write data
```