\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CSC121 PYTHON Programming**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lesson 11 Dictionaries, Sets and Exception Handling

# **Objectives**

In this lesson, students will learn:

- How to create and use dictionaries

- How to create and use sets

- How to write code to handle exceptions

- How to apply exception handling in Python programs

# **11.1 Dictionaries**

We learned lists earlier in this course. A list is an indexed linear data structure type. Elements are ordered and accessed by an index number. In this lesson, we are learning a new data structure type in Python. **Dictionary** is an associative data structure in which items are unordered and accessed by an associated key value. Let’s look at an example.

Suppose we have calculated the daily average temperature for each day of the week. We want to store these seven average temperatures for later use in the program. If we choose to store them in a list, we can write this:

daily\_temps = [68.8, 70.2, 67.2, 71.8, 73.2, 75.6, 74.0]

If we want to access the temperature of a specific day, we can use an index number to access it. For example, suppose we want to display the temperature of Wednesday, we can write:

print(daily\_temps[3])

We use the index 3 because Wednesday’s temperature is the fourth element in the list. Therefore, its index is 3. This index system is okay as long as we are careful not to use the wrong index number to access the wrong element. To reduce the chance of accessing the wrong element, we can use a dictionary instead of a list to store the temperatures. A dictionary does not use index to access data. Instead, we create a key to associate with each value when we add an item to a dictionary. We use the key to access the associated value. For instance, we can use strings such as ‘sun’, ‘mon’, ‘tue’ as keys in a dictionary that stores the daily average temperatures. The following statement creates this dictionary:

daily\_temps = {'sun': 68.8, 'mon': 70.2, 'tue': 67.2, 'wed': 71.8, 'thur': 73.2, 'fri': 75.6, 'sat': 74.0}

Each item in a dictionary includes two parts: a key and a value. The key is separated from its value by a colon (:), the items are separated by commas, and the whole dictionary is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

We use the key to access the value. To display the temperature on Wednesday, we write:

print(daily\_temps['wed'])

The following program creates the temperature dictionary and displays the temperatures on Wednesday and Saturday:

daily\_temps = {**'sun'**: 68.8, **'mon'**: 70.2, **'tue'**: 67.2, **'wed'**: 71.8, **'thur'**: 73.2, **'fri'**: 75.6, **'sat'**: 74.0}  
  
print(**"Temperature on Wednesday:"**, daily\_temps[**'wed'**])  
print(**"Temperature on Saturday:"**, daily\_temps[**'sat'**])

Output of the program:

Temperature on Wednesday: 71.8

Temperature on Saturday: 74.0

Suppose we are storing the names of a team of basketball players in a dictionary. We may use jersey numbers as keys:

players = {17: 'David Fox', 25: 'Jerry Rice', 8: 'George Simmons', 51: 'Larry King', 40: 'Tyler Park'}

The following program creates a dictionary for the basketball players. The program also displays the names of the players and uses the len function to find the number of items in the dictionary:

players = {17: **'David Fox'**, 25: **'Jerry Rice'**, 8: **'George Simmons'**, 51: **'Larry King'**, 40: **'Tyler Park'**}  
  
print(players[17])  
print(players[25])  
print(players[8])  
print(players[51])  
print(players[40])  
print(**"Number of players:"**, len(players))

Output of the program:

David Fox

Jerry Rice

George Simmons

Larry King

Tyler Park

Number of players: 5

The next example creates a dictionary of radio stations:

stations = {89.7:**'WCPE'**, 91.5:**'WUNC'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
print(**"89.7 FM:"**, stations[89.7])  
print(**"91.5 FM:"**, stations[91.5])  
print(**"92.3 FM:"**, stations[92.3])  
print(**"92.5 FM:"**, stations[92.5])  
print(**"Number of stations:"**, len(stations))

Output of the program:

89.7 FM: WCPE

91.5 FM: WUNC

92.3 FM: WKRR

92.5 FM: WYFL

Number of stations: 4

## **Creating a Dictionary**

In addition to using the syntax we saw earlier, we can also use the dict function to create a dictionary. If no argument is passed to the dict function, an empty dictionary is created. Example:

stations = dict()  
print(**"Number of stations:"**, len(stations))

Output of the program:

Number of stations: 0

A dictionary is created if a list of the following key-value pairs is passed to the dict function:

[[key1, value1], [key2, value2], [key3, value3], …]

Example:

station\_list = [[89.7,**'WCPE'**], [91.5,**'WUNC'**], [92.3,**'WKRR'**], [92.5,**'WYFL'**]]  
stations = dict(station\_list)  
  
print(**"89.7 FM:"**, stations[89.7])  
print(**"91.5 FM:"**, stations[91.5])  
print(**"92.3 FM:"**, stations[92.3])  
print(**"92.5 FM:"**, stations[92.5])  
print(**"Number of stations:"**, len(stations))

Output of the program:

89.7 FM: WCPE

91.5 FM: WUNC

92.3 FM: WKRR

92.5 FM: WYFL

Number of stations: 4

## **Adding or Modifying an Item**

You can update a dictionary by adding a new entry or modifying an existing entry as shown below in the example:

stations = {89.7:**'WCPE'**, 91.5:**'UNCW'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
stations[91.5] = **'WUNC'** *# update existing item*stations[88.9] = **'WSHA'** *# add new item*print(**"88.9 FM:"**, stations[88.9])  
print(**"89.7 FM:"**, stations[89.7])  
print(**"91.5 FM:"**, stations[91.5])  
print(**"92.3 FM:"**, stations[92.3])  
print(**"92.5 FM:"**, stations[92.5])  
print(**"Number of stations:"**, len(stations))

Output of the program:

88.9 FM: WSHA

89.7 FM: WCPE

91.5 FM: WUNC

92.3 FM: WKRR

92.5 FM: WYFL

Number of stations: 5

## **Deleting an Item**

You can either remove individual dictionary items or clear the entire contents of a dictionary. Example:

stations = {89.7:**'WCPE'**, 91.5:**'UNCW'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
**del** stations[91.5] *# delete dictionary item*print(**"Number of stations after del:"**, len(stations))  
  
stations.clear() *# delete all items*print(**"Number of stations after clear:"**, len(stations))

Output of the program:

Number of stations after del: 3

Number of stations after clear: 0

## **Testing Membership**

You can use the in operator to test whether a key exists in a dictionary. Example:

stations = {89.7:**'WCPE'**, 91.5:**'UNCW'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
target = float(input(**"Enter a station number: "**))  
  
**if** target **in** stations:  
 print(**"Station found in dictionary"**)  
**else**:  
 print(**"Station not found in dictionary"**)

Sample test run:

Enter a station number: 92.5

Station found in dictionary

## **Creating Lists from a Dictionary**

Sometimes we want extract the keys or values from a dictionary and store them in lists. The dictionary methods keys returns a sequence of the dictionary’s keys. Similarly, the values method returns a sequence of the dictionary’s values, while the items method returns a sequence of (key, value) tuples. These sequences can then be converted to lists using the list function. The following is an example:

stations = {89.7:**'WCPE'**, 91.5:**'UNCW'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
stations\_nums = list(stations.keys())  
print(**"Station Keys:"**)  
**for** key **in** stations\_nums:  
 print(key)  
  
stations\_names = list(stations.values())  
print(**"Station Names:"**)  
**for** name **in** stations\_names:  
 print(name)  
  
stations\_tuples = list(stations.items())  
print(**"Key-Name Tuples:"**)  
**for** tup **in** stations\_tuples:  
 print(tup)

Output of the program:

stati Station Keys:

92.5

89.7

91.5

92.3

Station Names:

WYFL

WCPE

UNCW

WKRR

Key-Name Tuples:

(92.5, 'WYFL')

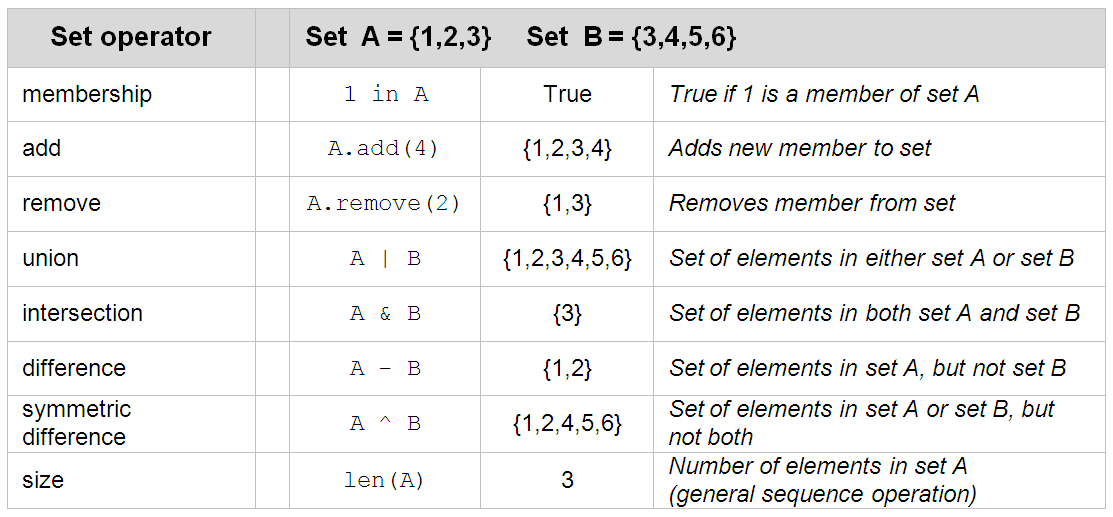
(89.7, 'WCPE')

(91.5, 'UNCW')

(92.3, 'WKRR')

# **11.2 Set Data Type**

A set is a mutable data type with non-duplicate, unordered values, providing the usual mathematical set operations as shown in Figure 9-10 of the textbook.



To create a set, enclose the elements with a pair of curly braces. Example:

fruit = {'apple', 'banana', 'pear', 'peach'}

A set named fruit is created with four elements.

Alternatively, you can use the set function to create a set from a list. The following example creates the fruit set with the set function, displays the elements and uses the len function to find the number of elements in the set:

fruit\_list = [**'apple'**, **'banana'**, **'pear'**, **'peach'**]  
fruit = set(fruit\_list)  
print(**"Number of elements:"**, len(fruit))  
print(**"Elements in the set:"**, fruit)

Output of the program:

Number of elements: 4

Elements in the set: {'peach', 'pear', 'banana', 'apple'}

To create an empty set, call the set function without passing an argument to it. Example:

fruit = set()  
print(**"Number of elements:"**, len(fruit))

Output of the program:

Number of elements: 0

We can use the add method of a set to add an element, and use the remove method to remove an element. We can also use the in operator to test whether a target value is in the set or not. Example:

fruit = set()  
fruit.add(**'apple'**)  
fruit.add(**'banana'**)  
fruit.add(**'pear'**)  
fruit.add(**'peach'**)  
print(**"Elements in the set:"**, fruit)  
fruit.remove(**'pear'**)  
print(**"Elements in the set after remove:"**, fruit)  
  
target = input(**"Enter a search target: "**)  
**if** target **in** fruit:  
 print(**"target found in set"**)  
**else**:  
 print(**"target not found in set"**)

Sample test run:

Elements in the set: {'apple', 'peach', 'banana', 'pear'}

Elements in the set after remove: {'apple', 'peach', 'banana'}

Enter a search target: banana

target found in set

There are four Python set operations implementing four mathematical set operations: union, intersection, difference and symmetric difference. The union of two sets A and B, denoted as A | B in Python, includes all elements that are either in A or B. The interaction of two sets, denoted as A & B, includes only elements that are in both sets. The difference between A and B, denoted as A - B, includes elements in set A but not in set B. Notice that B - A is not the same as A - B, because B - A gives us the elements in B that are not in A. Finally, the symmetric difference of A and B, denoted as A ^ B, gives us the elements in set A or set B, but not both. Example:

set\_A = {**'apple'**, **'banana'**, **'pear'**, **'peach'**}  
set\_B = {**'orange'**, **'banana'**, **'grape'**}  
  
print(**"A union B:"**, set\_A | set\_B)  
print(**"A intersect B:"**, set\_A & set\_B)  
print(**"A - B:"**, set\_A - set\_B)  
print(**"B - A:"**, set\_B - set\_A)  
print(**"Symmetric difference between A and B:"**, set\_A ^ set\_B)

Output of the program:

A union B: {'grape', 'pear', 'peach', 'banana', 'apple', 'orange'}

A intersect B: {'banana'}

A - B: {'pear', 'peach', 'apple'}

B - A: {'grape', 'orange'}

Symmetric difference between A and B: {'pear', 'apple', 'orange', 'grape', 'peach'}

# **11.3 Dictionary Comprehensions and** **Set Comprehensions**

Python also supports **dictionary comprehensions**, which allow you to create dictionaries at runtime using a similar concise syntax, as with list comprehensions.

A dictionary comprehension takes the form

{key: value for (key, value) in iterable if condition}

A typical example is taking two lists and creating a dictionary where the item at each position in the first list becomes a key and the item at the corresponding position in the second list becomes the value.

The following code creates a dictionary which stores the cost of fruit at a store using a dictionary expression:

prod = ['apples', 'oranges', 'squash']

price = [1.25, 0.99, 0.89]

fruit\_cost = {key : value for (key, value) in zip (prod, price)}

print (fruit\_cost)

**Output:**

**{'apples': 1.25, 'oranges': 0.99, 'squash': 0.89}**

The following code creates a dictionary which uses integers as keys and their cubes as values. The first code snippet creates the dictionary using a for loop and the second code snippet creates the same dictionary using a dictionary comprehension:

cubes = dict()

for x in range(5):

cubes[x] = x\*\*3

print (cubes)

**Output:**

**{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}**

cubes = {x: x\*\*3 for x in range(5)}

print (cubes)

**Output:**

**{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}**

The following dictionary comprehension uses two for loops.

dict\_a = {i:j for i in range(5, 10) for j in range(i)}

print (dict\_a)

Output:

**{5: 4, 6: 5, 7: 6, 8: 7, 9: 8}**

The inner for loop with ‘j’ as the loop variable creates the follow set of dictionary items on its initial traversal:

**{5: 0}, {5: 1}, {5: 2}, {5: 3}, {5: 4}**

Only the last one is retained, as consecutive items override the previous, because key values must be unique.

This last example of a dictionary comprehension uses an if statement filter to create a dictionary of only the even squares:

squ\_dict = {x: x\*\*2 for x in range(13) if x % 2 == 0}

print (squ\_dict)

**Output:**

**{0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100, 12: 144}**

The syntax for **set comprehensions** is almost identical to that of list comprehensions, but it uses curly braces instead of square brackets. The result of a set comprehension is the same as passing the output of the equivalent list comprehension to the set function.

squ\_set = {x\*\*2 for x in range(13) if x % 2 == 0}

print (squ\_ set)

**Output:**

**{0, 4, 16, 36, 64, 100, 144}**

# **11.4 Exception Handling**

An exception is an unexpected event that happens when the program is running. For instance, suppose we have written the following program to give a 5% raise to an employee.

salary = float(input(**"Enter annual salary: "**))  
salary = salary + salary \* 0.05  
print(**"New salary:"**, salary)

In the first line the program asks the user to enter his annual salary. The program expects the use to enter a floating point number. If the user enters something that cannot be read as a floating point number, an exception happens:

Enter annual salary: 21,500.0

The input string “21,500.0” cannot be converted to a floating number because Python does not allow commas in numerical values, either in integers or floating point numbers. Since a non-floating-point value is entered unexpectedly, the program cannot continue. An error message is shown and the program terminates prematurely.

Traceback (most recent call last):

File "C:/Users/Labuser/Desktop/Leung/p.py", line 1, in <module>

salary = float(input("Enter annual salary: "))

ValueError: could not convert string to float: '21,500.0'

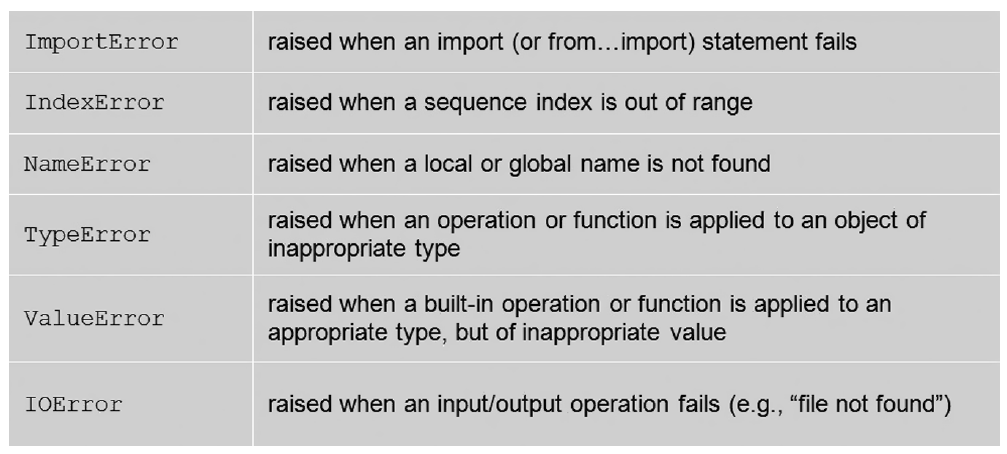
Process finished with exit code 1

This type of exception is called ValueError exception. The message “Process finished with exit code 1” means the program terminated with unhandled exception. If a program ends normally, the exit coed is 0.

Various error messages can occur when executing Python programs. So far we have let Python handle these errors by reporting them on the screen and quitting the program immediately. In fact, we can write code in the program to catch and handle exceptions. The program can either correct the error and continue execution, or terminate gracefully.

## **Standard Exceptions**

Python contains a predefined set of exceptions referred to as standard exceptions. These standard exceptions cover the most common types of “unexpected” situations. Figure 8-8 of the textbook shows some of the standard exceptions.



The standard exceptions are defined within the exceptions module of the Python Standard Library, which is automatically imported into Python programs. We have seen an example of ValueError exception earlier. The following are examples of some other types.

First, let’s look at an example of IndexError.

my\_list = [1, 2, 3]  
print(my\_list[6])

An IndexError exception is raised when we run the program above:

Traceback (most recent call last):

File "C:/Users/Labuser/Desktop/Leung/p.py", line 2, in <module>

print(my\_list[6])

IndexError: list index out of range

Next, let’s look at an example of TypeError exception.

x = 2 + **'3'**print(x)

A TypeError exception is raised when we run the program above:

Traceback (most recent call last):

File "C:/Users/Labuser/Desktop/Leung/p.py", line 1, in <module>

x = 2 + '3'

TypeError: unsupported operand type(s) for +: 'int' and 'str'

The following is an example of NameError exception.

gpa = 3.78  
print(GPA)

A NameError exception is raised when we run the program above:

Traceback (most recent call last):

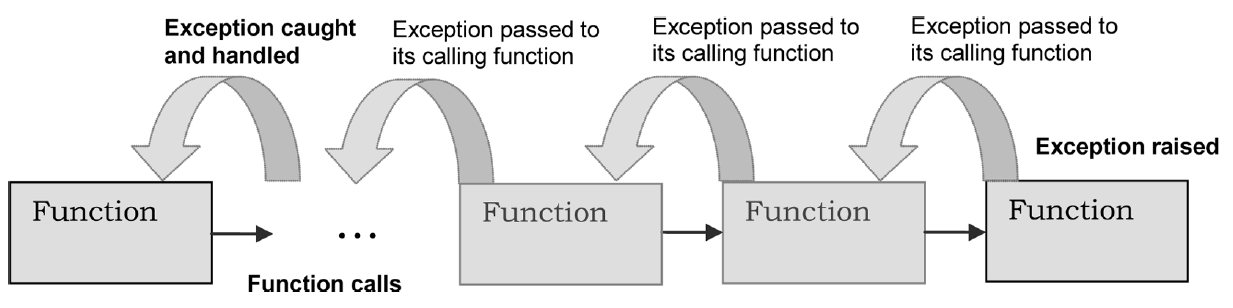
File "C:/Users/Labuser/Desktop/Leung/p.py", line 2, in <module>

print(GPA)

NameError: name 'GPA' is not defined

## **The Propagation of Raised Exceptions**

Most programs contain built-in functions and user-defined functions. Raising an exception is a way for a function to inform its client a problem has occurred that the function itself cannot handle. When an exception is raised and not handled by a function, it is automatically propagated back to the *calling code (and its* calling code, etc.) until handled. If an exception is thrown all the way back to the top level and not handled, then the program terminates and displays the details of the exception. This is shown in Figure 8-9 of the textbook.



The following is an example:

**def** main():  
 age = int(input(**"How old are you? "**))  
 get\_older(age)  
  
**def** get\_older(age):  
 age\_next\_year = ag + 1  
 print(**"Your age next year:"**, age\_next\_year)  
  
main()

The variable age is misspelled in the function get\_older. A NameError exception is raised when we run the program:

How old are you? 17

Traceback (most recent call last):

File "C:/Users/Labuser/Desktop/Leung/p.py", line 9, in <module>

main()

File "C:/Users/Labuser/Desktop/Leung/p.py", line 3, in main

get\_older(age)

File "C:/Users/Labuser/Desktop/Leung/p.py", line 6, in get\_older

age\_next\_year = ag + 1

NameError: name 'ag' is not defined

If we read the details of the error message, we will see the chain of function calls. First, the main function is called in line 9. Then in the main function, the get\_older function is called in line 3. In the get\_older function, a NameError exception is raised in line 6. When this exception is raised, the computer checks the local function (i.e., the get\_older function) to see whether there is any code to handle a NameError exception. Since no code is found in the local function, the exception is propagated up to its calling function (i.e., the main function). Since there is no code in the main function to handle a NameError exception neither, this exception is propagated up to the top level. Since no code is found up there to handle a NameError exception, the computer displays the error message and terminates the program immediately.

If an exception is caught and handled anywhere in the propagation chain before it reaches the top level, the program will not terminate with unhandled exception.

## **Catching and Handling Exceptions**

In Python programs, we can write statements to catch and handle exceptions. We can tell the computer what to do if certain type of exception happens by using try and except blocks. The following is an example.

**try**:  
 salary = float(input(**"Enter annual salary: "**))  
 salary = salary + salary \* 0.05  
 print(**"New salary:"**, salary)  
**except** ValueError:  
 print(**"Salary must be a numerical value with no commas."**)

We put statements that input, calculate and display salary inside a try block and put a customized error message inside an except block. The computer will display the customized error message if a ValueError exception occurs:

Enter annual salary: 21,500.0

Salary must be a numerical value with no commas.

Process finished with exit code 0

The program finishes with an exit code 0, which means the exception is handled within the program. That is indeed very important because in addition to allowing us to display a customized error message, exception handling allows the program to continue and end normally. This is shown in the following example.

print(**"This program calculates new salaries for 3 employees."**)  
**for** i **in** range(3):  
 **try**:  
 salary = float(input(**"Enter annual salary: "**))  
 salary = salary + salary \* 0.05  
 print(**"New salary:"**, salary)  
 **except** ValueError:  
 print(**"Salary must be a numerical value with no commas."**)

This program uses a for loop to calculate new salaries for 3 employees. We have the same try and except blocks inside the loop to catch ValueError exceptions. We know that without using exception handling, the whole program will terminate immediately if any input salary is not a valid numerical value. Let’s see what happens with exception handling:

This program calculates new salaries for 3 employees.

Enter annual salary: 21,500.0

Salary must be a numerical value with no commas.

Enter annual salary: 30000.0

New salary: 31500.0

Enter annual salary: 40000

New salary: 42000.0

Process finished with exit code 0

The first input value is 21,500.0, which is an invalid numerical value in Python. A ValueError exception is caught and handled by the try and except blocks. Customized message is shown and most importantly, the program continues and goes into the second and third iterations of the for loop before it ends normally with an exit code 0.

Let’s look at another example. This one is about IOError exception.

In a previous lesson we learned how to open a text file for reading data. The following program opens the text file “my\_file.txt” for reading:

input\_file = open(**'my\_file.txt'**, **'r'**)  
empty\_str = **''**

line = input\_file.readline()  
**while** line != empty\_str:  
 print(line.strip(**'\n'**)) *# strip new line character* line = input\_file.readline()  
  
input\_file.close()

This program opens only the file “my\_file.txt” but not any other text file. Let’s modify the program so it will open a file chosen by the user:

file\_name = input(**"Enter file name: "**)  
input\_file = open(file\_name, **'r'**)  
empty\_str = **''**line = input\_file.readline()  
**while** line != empty\_str:  
 print(line.strip(**'\n'**))  
 line = input\_file.readline()  
  
input\_file.close()

The user is asked to enter the file name and the program will read that file line by line.

Suppose the file entered by the user does not exist. An unhandled exception is raised and the program terminates with exit code 1.

Enter file name: my\_file2.txt

Traceback (most recent call last):

File "C:/Users/mleun\_000/Desktop/untitled1/p3.py", line 2, in <module>

input\_file = open(file\_name, 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'my\_file2.txt'

Process finished with exit code 1

We can add code in the program to catch and handle this exception.

**try**:  
 file\_name = input(**"Enter file name: "**)  
 input\_file = open(file\_name, **'r'**)  
 empty\_str = **''** line = input\_file.readline()  
 **while** line != empty\_str:  
 print(line.strip(**'\n'**))  
 line = input\_file.readline()  
  
 input\_file.close()  
  
**except** IOError:  
 print(**'File Open Error.'**)

Sample test run:

Enter file name: my\_file2.txt

File Open Error.

Process finished with exit code 0

A customized error message is displayed before the program terminates normally.

We can modify the program so it will ask for a new file name until the file can be opened.

file\_name = input(**"Enter file name: "**)  
empty\_str = **''**input\_file\_opened = **False  
  
while not** input\_file\_opened:  
 **try**:  
 input\_file = open(file\_name, **'r'**)  
 input\_file\_opened = **True** line = input\_file.readline()  
 **while** line != empty\_str:  
 print(line.strip(**'\n'**))  
 line = input\_file.readline()  
  
 input\_file.close()  
  
 **except** IOError:  
 print(**'File Open Error.'**)  
 file\_name = input(**"Enter file name: "**)

This program uses a loop similar to the input validation loops we have seen in previous lessons. The loop repeats again and again until a valid input is entered. In this case, it stops only when the file can be opened. Try-and-except is used to catch IOError exception, which is raised when the file does not exist. When that happens, an error message is displayed and the user is asked to enter a different file name. The following is a sample test run:

Enter file name: my\_file2.txt

File Open Error.

Enter file name: my\_file3.txt

File Open Error.

Enter file name: my\_file4.txt

Student registered in CSC121:

Peter Holmes

David Fox

Process finished with exit code 0

Files “my\_file2.txt” and “my\_file3.txt” cannot be opened because they do not exist. The program keeps asking for a new file name until the name “my\_file4.txt” is entered. Data are then read from the file and displayed.

# **11.5 Further Readings**

Please read Section 9.1, 9.2 and 8.4 of the textbook. Section 9.1 introduces dictionaries. Section 9.2 is about sets. Section 8.4 discusses exception handling.