FACULTEIT ECONOMIE
EN BEDRIJFSKUNDE

CHAPTER 13
FILES AND EXCEPTION HANDLING

UNIVERSITEIT
GENT

MOTIVATIONS

- Data stored in the program are temporary; they are lost when the program terminates. To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage. The file can be transported and can be read later by other programs. There are two types of files: text and binary. Text files are essentially strings on disk. This chapter introduces how to read/write data from/to a text file.

- When a program runs into a runtime error, the program terminates abnormally. How can you handle the runtime error so that the program can continue to run or terminate gracefully? This is the subject we will introduce in this chapter.

OBJECTIVES

- To open a file, read/write data from/to a file (§13.2)
- To use file dialogs for opening and saving data (§13.3).
- To develop applications with files (§13.4)
- To read data from a Web resource (§13.5).
- To handle exceptions using the try/except/finally clauses (§13.6)
- To raise exceptions using the raise statements (§13.7)
- To become familiar with Python's built-in exception classes (§13.8)
- To access exception object in the handler (§13.8)
- To define custom exception classes (§13.9)
- To perform binary IO using the pickle module (§13.10)

OPEN A FILE

How do you write data to a file and read the data back from a file? You need to create a file object that is associated with a physical file. This is called *opening a file*. The syntax for opening a file is as follows:

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

Mode	Description
'r'	Open a file for reading only.
'w'	Open a file for writing only.
'a'	Open a file for appending data. Data are written to the end of the file.
'rb'	Open a file for reading binary data.
'wb'	Open a file for writing binary data.

WRITE TO A FILE

```
outfile = open("test.txt", "w")
outfile.write("Welcome to Python")
```

file	
read([number: int]): str	Returns the specified number of characters from the file. If the argument is omitted, the entire remaining contents are read.
readline(): str	Returns the next line of file as a string.
readlines(): list	Returns a list of the remaining lines in the file.
write(s: str): None	Writes the string to the file.
close(): None	Closes the file.

WriteDemo

UNIVERSITEIT GENT

TESTING FILE EXISTENCE

```
import os.path
if os.path.isfile("Presidents.txt"):
    print("Presidents.txt exists")
```

READ FROM A FILE

After a file is opened for reading data, you can use the read method to read a specified number of characters or all characters, the readline() method to read the next line, and the readlines() method to read all lines into a list.

ReadDemo

APPEND DATA TO A FILE

You can use the 'a' mode to open a file for appending data to an existing file.

AppendDemo

WRITING/READING NUMERIC DATA

To write numbers, convert them into strings, and then use the write method to write them to a file. In order to read the numbers back correctly, you should separate the numbers with a whitespace character such as ' ', '\n'.

WriteReadNumbers

PROBLEM: COUNTING EACH LETTER IN A FILE

The problem is to write a program that prompts the user to enter a file and counts the number of occurrences of each letter in the file regardless of case.

CountEachLetter

RETRIEVING DATA FROM THE WEB

Using Python, you can write simple code to read data from a Website. All you need to do is to open a URL link using the urlopen function as follows:

```
infile = urllib.request.urlopen('http://www.yahoo.com')
```

```
import urllib.request
infile = urllib.request.urlopen('http://www.yahoo.com/index.html')
print(infile.read().decode())
```

EXCEPTION HANDLING

When you run the program in Listing 11.3 or Listing 11.4, what happens if the user enters a file or an URL that does not exist? The program would be aborted and raises an error. For example, if you run Listing 11.3 with an incorrect input, the program reports an IO error as shown below:

```
c:\pybook\python CountEachLetter.py
Enter a filename: newinput.txt
Traceback (most recent call last):
  File "CountEachLetter.py", line 23, in <module>
main()
  File "CountEachLetter.py", line 4, in main
Infile = open(filename, "r"> # Open the file
IOError: [Errno 2] No such file or directory: 'newinput.txt'
```

THE TRY ... EXCEPT CLAUSE

```
try:
    <body>
except <ExceptionType>:
     <handler>
```

THE TRY ... EXCEPT CLAUSE

```
try:
<body>
except <ExceptionType1>:
    <handler1>
...
except <ExceptionTypeN>:
    <handlerN>
except:
    <handlerExcept>
else:
   <process_else>
finally:
   <process_finally>
```

TestException

UNIVERSITEIT
GENT

RAISING EXCEPTIONS

You learned how to write the code to handle exceptions in the preceding section. Where does an exception come from? How is an exception created? Exceptions are objects and objects are created from classes. An exception is raised from a function. When a function detects an error, it can create an object of an appropriate exception class and raise the object, using the following syntax:

- raise ExceptionClass("Something is wrong")

UNIVERSITEIT GENT

RaiseException

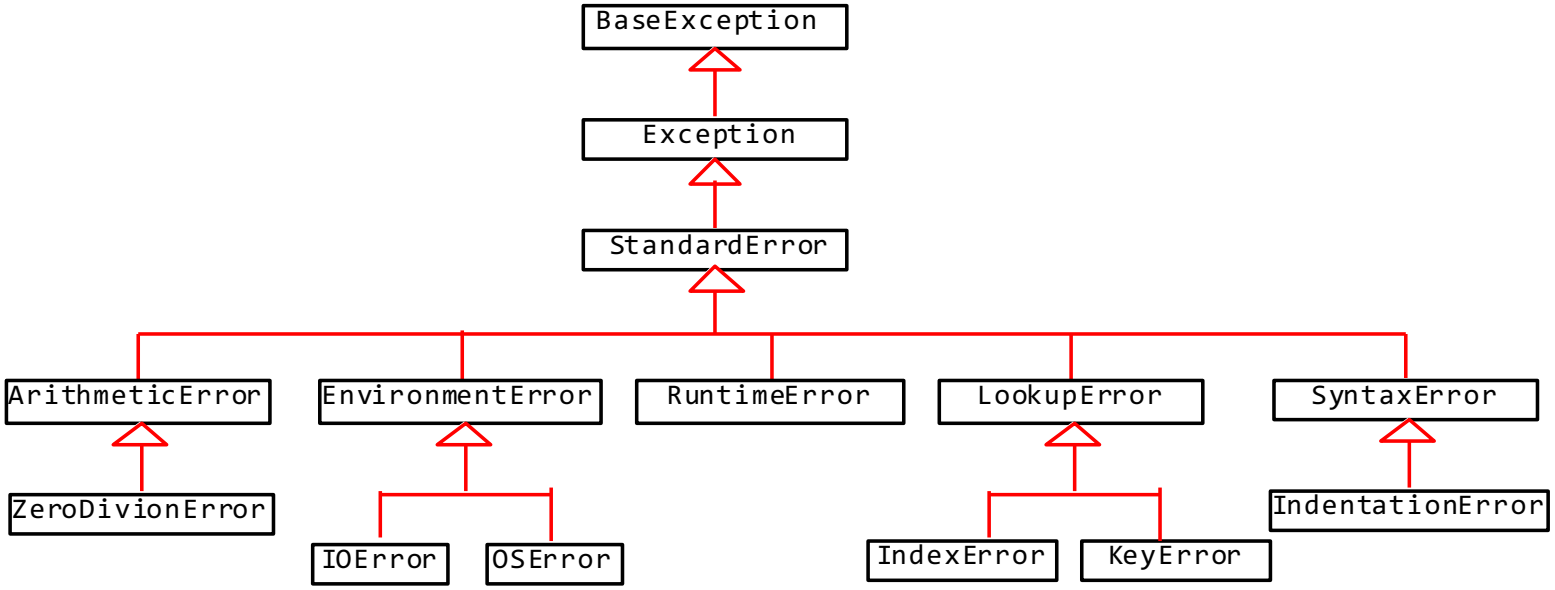
PROCESSING EXCEPTIONS USING EXCEPTION OBJECTS

– You can access the exception object in the except clause.

```
try
    <body>
except ExceptionType as ex:
    <handler>
```

ProcessExceptionObject

DEFINING CUSTOM EXCEPTION CLASSES



TestCircleWithCustomException

InvalidRadiusException

UNIVERSITEIT GENT

CHAPTER 14
TUPLES, SETS, AND DICTIONARIES

MOTIVATIONS

The No Fly List is a list, created and maintained by the United States government's Terrorist Screening Center, of people who are not permitted to board a commercial aircraft for travel in or out of the United States. Suppose we need to write a program that checks whether a person is in the No Fly List. You can use a Python list to store the persons in the No Fly List. However, a more efficient data structure for this application is a set.

OBJECTIVES

- To use tuples as immutable lists (§14.2).
- To use sets for storing and fast accessing non-duplicate elements (§14.3).
- To understand the performance differences between sets and lists (§14.4).
- To store key/value pairs in a dictionary and access value using the key (§14.5).
- To use dictionaries to develop applications (§14.6).

TUPLES

Tuples are like lists except they are immutable. Once they are created, their contents cannot be changed.

If the contents of a list in your application do not change, you should use a tuple to prevent data from being modified accidentally. Furthermore, tuples are more efficient than lists

CREATING TUPLES

```
t1 = () # Create an empty tuple

t2 = (1, 3, 5) # Create a set with three elements

# Create a tuple from a list
t3 = tuple([2 * x for x in range(1, 5)])

# Create a tuple from a string
t4 = tuple("abac") # t4 is ['a', 'b', 'a', 'c']
```

TupleDemo

SETS

Sets are like lists to store a collection of items. Unlike lists, the elements in a set are unique and are not placed in any particular ordered. If your application does not care about the order of the elements, using a set to store elements is more efficient than using lists. The syntax for sets is braces {}.

CREATING SETS

```
s1 = set() # Create an empty set

s2 = {1, 3, 5} # Create a set with three elements

s3 = set([1, 3, 5]) # Create a set from a tuple

# Create a set from a list
s4 = set([x * 2 for x in range(1, 10)])

# Create a set from a string
s5 = set("abac") # s5 is {'a', 'b', 'c'}
```

MANIPULATING AND ACCESSING SETS

```
>>> s1 = {1, 2, 4}
>>> s1.add(6)
>>> s1
{1, 2, 4, 6}
>>> len(s1)
4
>>> max(s1)
6
>>> min(s1)
1
>>> sum(s1)
13
>>> 3 in s1
False
>>> s1.remove(4)
>>> s1
{1, 2, 6}
>>>
```

UNIVERSITEIT GENT

SUBSET AND SUPERSET

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 5, 2, 6}
>>> s1.issubset(s2) # s1 is a subset of s2
True
>>>
```

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 5, 2, 6}
>>> s2.issuperset(s1) # s2 is a superset of s1
True
>>>
```

EQUALITY TEST

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 2}
>>> s1 == s2
True
>>> s1 != s2
False
>>>
```

UNIVERSITEIT
GENT

COMPARISON OPERATORS

Note that it makes no sense to compare the sets using the conventional comparison operators (>, >=, <=, <), because the elements in a set are not ordered. However, these operators have special meaning when used for sets.

s1 > s2 returns true is s1 is a proper superset of s2.

s1 >= s2 returns true is s1 is a superset of s2.

s1 < s2 returns true is s1 is a proper subset of s2.

s1 <= s2 returns true is s1 is a subset of s2.

SET OPERATIONS (UNION, |)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.union(s2)
{1, 2, 3, 4, 5}
>>>
>>> s1 | s2
{1, 2, 3, 4, 5}
>>>
```

UNIVERSITEIT
GENT

SET OPERATIONS (INTERSECTION, &)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.intersection(s2)
{1}
>>>
>>> s1 & s2
{1}
>>>
```

UNIVERSITEIT
GENT

SET OPERATIONS (DIFFERENCE, -)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.difference(s2)
{2, 4}
>>>
>>> s1 - s2
{2, 4}
>>>
```

UNIVERSITEIT
GENT

SET OPERATIONS (SYMETRIC_DIFFERENCE, ^)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.symmetric_difference(s2)
{2, 3, 4, 5}
>>>
>>> s1 ^ s2
{2, 3, 4, 5}
>>>
```

SetListPerformanceTest

SetDemo

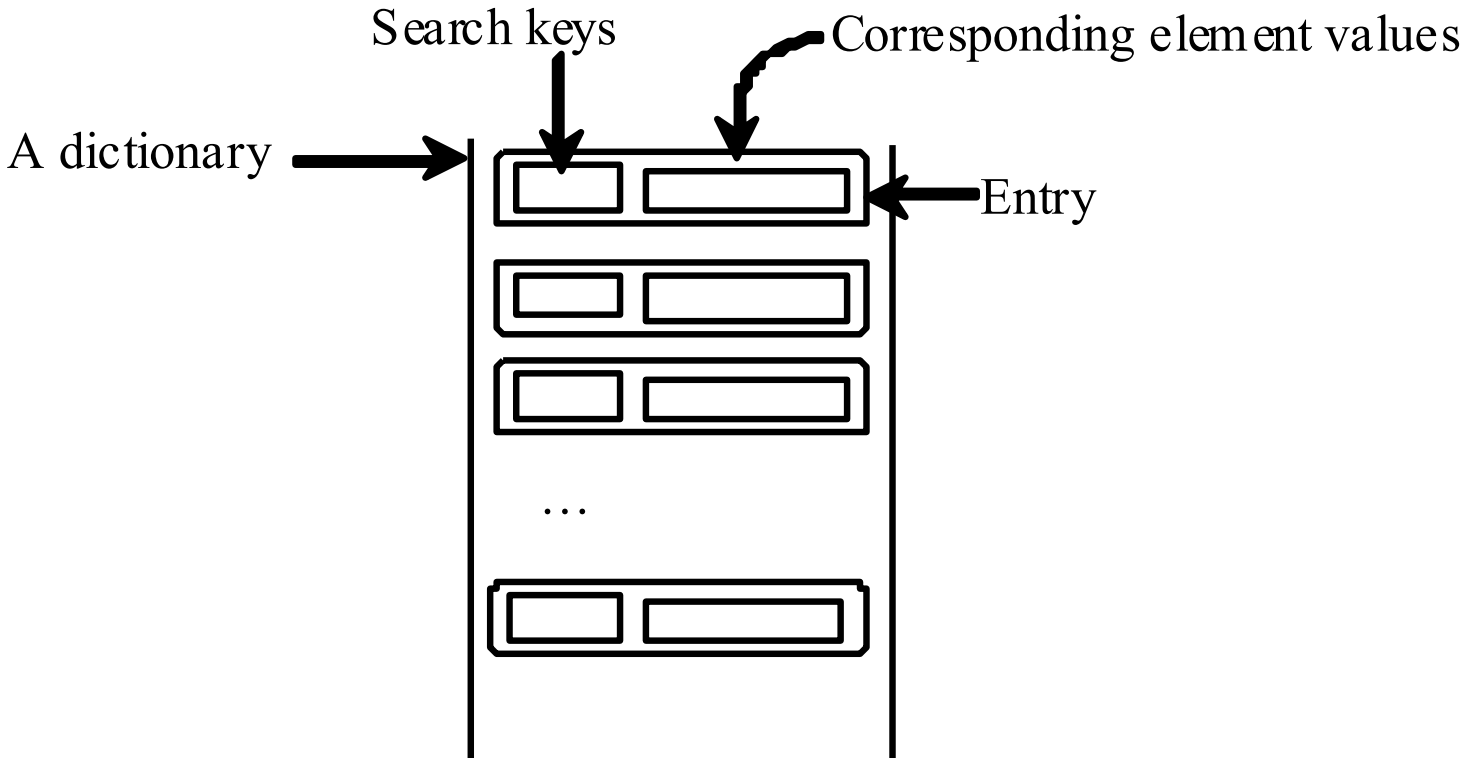
UNIVERSITEIT GENT

DICTIONARY

Why dictionary?

Suppose your program stores a million students and frequently searches for a student using the social security number. An efficient data structure for this task is the dictionary. A dictionary is a collection that stores the elements along with the keys. The keys are like an indexer.

KEY/VALUE PAIRS



UNIVERSITEIT
GENT

CREATING A DICTIONARY

```
dictionary = {} # Create an empty dictionary
dictionary = {"john":40, "peter":45} # Create a dictionary
```

ADDING/MODIFYING ENTRIES

- To add an entry to a dictionary, use
  dictionary[key] = value

- For example,
  dictionary["susan"] = 50

DELETING ENTRIES

- To delete an entry from a dictionary, use
  del dictionary[key]

- For example,
  del dictionary["susan"]

LOOPING ENTRIES

```
for key in dictionary:
    print(key + ":" + str(dictionary[key]))
```

THE LEN AND IN OPERATORS

len(dictionary) returns the number of the elements in the dictionary.

```
>>> dictionary = {"john":40, "peter":45}
>>> "john" in dictionary
True
>>> "johnson" in dictionary
False
```

THE DICTIONARY METHODS

dict	
keys(): tuple	Returns a sequence of keys.
values(): tuple	Returns a sequence of values.
items(): tuple	Returns a sequence of tuples (key, value).
clear(): void	Deletes all entries.
get(key): value	Returns the value for the key.
pop(key): value	Removes the entry for the key and returns its value.
popitem(): tuple	Returns a randomly-selected key/value pair as a tuple and removes the selected entry.

CASE STUDIES: OCCURRENCES OF WORDS

This case study writes a program that counts the occurrences of words in a text file and displays the words and their occurrences in alphabetical order of words. The program uses a dictionary to store an entry consisting of a word and its count. For each word, check whether it is already a key in the dictionary. If not, add to the dictionary an entry with the word as the key and value 1. Otherwise, increase the value for the word (key) by 1 in the dictionary.

CountOccurrenceOfWords