UNIT V FILES, MODULES, PACKAGES

Files and exception: text files, reading and writing files, format operator; command line arguments, errors and exceptions, handling exceptions, modules, packages; Illustrative programs: word count, copy file.

FILES

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

Opening a file

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python33/README.txt") # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file. On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

Python File Modes

Mode	Description
'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and w

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')  # write in text mode
```

```
f = open("img.bmp",'r+b') # read and write in binary mode
```

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt",mode = 'r',encoding = 'utf-8')
```

Closing a File

When we are done with operations to the file, we need to properly close it.

Closing a file will free up the resources that were tied with the file and is done using the close() method.

Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

```
f = open("test.txt",encoding = 'utf-8')
# perform file
operations f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file. A safer way is to use a try...finally block.

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This way, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.

The best way to do this is using the with statement. This ensures that the file is closed when the block inside with is exited.

We don't need to explicitly call the close() method. It is done internally.

```
with open("test.txt",encoding = 'utf-8') as f:
   # perform file operations
```

Reading and writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM.

To write a file, you have to open it with mode 'w' as a second parameter:

```
>>>  fout = open('output.txt', 'w')
>>>  print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn’t exist, a new one is created.

The write method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

Again, the file object keeps track of where it is, so if you call write again, it adds the new data to the end.

```
>>>  line2 = "the emblem of our land.\n"
>>>  fout.write(line2)
```

When you are done writing, you have to close the file.

```
>>> fout.close()
```

Format operator

The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with str:

```
>>>  x = 52
>>>  fout.write(str(x))
```

An alternative is to use the **format operator**, %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence '%d' means that the second operand should be formatted as an integer (d stands for “decimal”):

```
>>>  camels = 42
>>>  '%d' % camels
'42'
```

The result is the string '42', which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>>  camels = 42
>>>  'I have spotted %d camels.' %
camels 'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses '%d' to format an integer, '%g' to format a floating-point number and '%s' to format a string:

```
>>>  'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

Filenames and paths

Files are organized into **directories** (also called “folders”). Every running program has a “current directory,” which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The os module provides functions for working with files and directories (“os” stands for “operating system”). os.getcwd returns the name of the current directory:

```
>>>  import os
>>>  cwd = os.getcwd()
>>>  print cwd
/home/dinsdale
```

cwd stands for “current working directory.” The result in this example is /home/dinsdale, which is the home directory of a user named dinsdale.

A string like cwd that identifies a file is called a **path**. A **relative path** starts from the current directory; an **absolute path** starts from the topmost directory in the file system.

The paths we have seen so far are simple filenames, so they are relative to the current directory. To find the absolute path to a file, you can use os.path.abspath:

```
>>>  os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.path.exists checks whether a file or directory exists:

```
> > > os.path.exists('memo.txt')
True
```

If it exists, os.path.isdir checks whether it’s a directory:

```
>>>  os.path.isdir('memo.txt')
False
>>>  os.path.isdir('music')
True
```

Similarly, os.path.isfile checks whether it’s a file.

os.listdir returns a list of the files (and other directories) in the given directory:

```
>>>  os.listdir(cwd) ['music',
'photos', 'memo.txt']
```

To demonstrate these functions, the following example “walks” through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dirname):
for name in os.listdir(dirname):
path = os.path.join(dirname, name)
if os.path.isfile(path):
print path
else:
walk(path)
```

os.path.join takes a directory and a file name and joins them into a complete path.

EXCEPTION

Python (interpreter) raises exceptions when it encounters errors. Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

```
>>> if a < 3
  File "<interactive input>", line 1
    if a < 3
           ^
SyntaxError: invalid syntax
```

Errors can also occur at runtime and these are called exceptions. They occur, for example, when a file we try to open does not exist (FileNotFoundError), dividing a number by zero (ZeroDivisionError), module we try to import is not found (ImportError) etc.

Whenever these type of runtime error occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

```
>>> 1 / 0
Traceback (most recent call last):
 File "<string>", line 301, in runcode
 File "<interactive input>", line 1, in <module>
ZeroDivisionError: division by zero

>>> open("imaginary.txt")
Traceback (most recent call last):
 File "<string>", line 301, in runcode
 File "<interactive input>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'
```

Python Built-in Exceptions

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur. We can view all the built-in exceptions using the local() built-in functions as follows.

```
>>> locals()['__builtins__']
```

This will return us a dictionary of built-in exceptions, functions and attributes.

Some of the common built-in exceptions in Python programming along with the error that cause then are tabulated below.

Python Built-in Exceptions

Exception	Cause of Error
AssertionError	Raised when assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() functions hits end-of-file condition.

FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets argument of correct type but improper value.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.

We can handle these built-in and user-defined exceptions in Python using try, except and finally statements.

Python Exception Handling

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.

When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.

For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A.

If never handled, an error message is spit out and our program come to a sudden, unexpected halt.

Catching Exceptions in Python

In Python, exceptions can be handled using a try statement.

A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.

It is up to us, what operations we perform once we have caught the exception. Here is a simple example.

```
# import module sys to get the type of
exception import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!",sys.exc_info()[0],"occured.")
        print("Next entry.")
        print()
print("The reciprocal of",entry,"is",r)
```

Output

```
The entry is a
Oops! <class 'ValueError'> occured.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError' > occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

In this program, we loop until the user enters an integer that has a valid reciprocal. The portion that can cause exception is placed inside try block.

If no exception occurs, except block is skipped and normal flow continues. But if any exception occurs, it is caught by the except block.

Here, we print the name of the exception using ex_info() function inside sys module and ask the user to try again. We can see that the values 'a' and '1.3' causes ValueError and '0' causes ZeroDivisionError.

try...finally

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.

For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution. Here is an example of file operations to illustrate this.

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

MODULES

Any file that contains Python code can be imported as a module. For example, suppose you have a file named wc.py with the following code:

```
def linecount(filename):
count = 0
for line in open(filename):
count += 1
return count
print linecount('wc.py')
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7.

You can also import it like this:

```
>>>  import wc
7
```

Now you have a module object wc:

```
>>>  print wc
<module 'wc' from 'wc.py'>

>>>  wc.linecount('wc.py')
7
```

So that’s how you write modules in Python.

The only problem with this example is that when you import the module it executes the test code at the bottom. Normally when you import a module, it defines new functions but it doesn’t execute them.

Programs that will be imported as modules often use the following idiom: *if __name__ == '__main__':*

```
print linecount('wc.py')
```

__name__ is a built-in variable that is set when the program starts. If the program is running as a script, __name__ has the value __main__; in that case, the test code is executed. Otherwise, if the module is being imported, the test code is skipped. Eg:

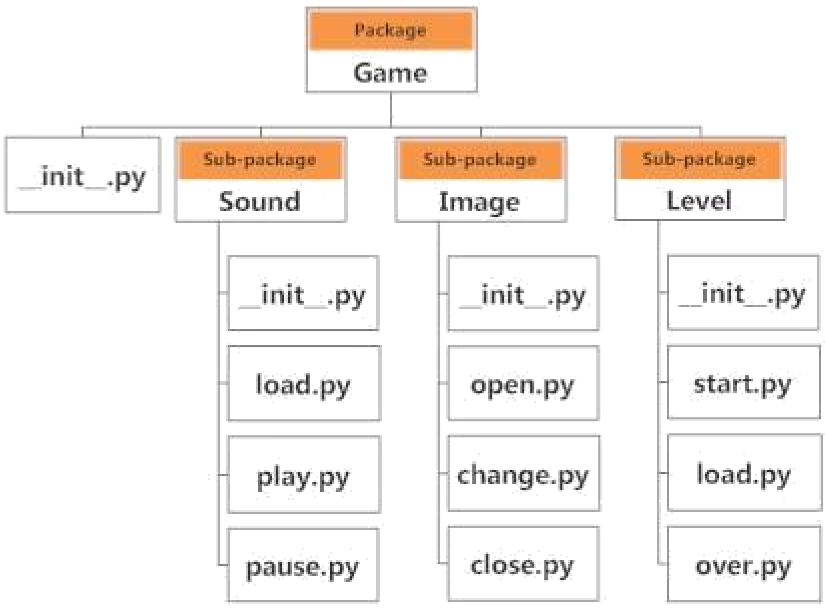
```
# import  module
import calendar

yy = 2017
mm = 8

# To ask month and year from the user
# yy = int(input("Enter year: "))
# mm = int(input("Enter month: "))

# display the calendar
print(calendar.month(yy, mm))
```

PACKAGE

A **package** is a collection of modules. A Python package can have sub-packages and modules.

A directory must contain a file named __init__.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.



Importing module from a package

We can import modules from packages using the dot (.) operator.

For example, if want to import the start module in the above example, it is done as follows. *import Game.Level.start*

Now if this module contains a function named *select_difficulty()*, we must use the full name to reference it.
Game.Level.start.select_difficulty(2)

If this construct seems lengthy, we can import the module without the package prefix as follows.
from Game.Level import start

We can now call the function simply as follows.
start.select_difficulty(2)

Yet another way of importing just the required function (or class or variable) form a module within a package would be as follows.
from Game.Level.start import select_difficulty

Now we can directly call this function.
select_difficulty(2)

Although easier, this method is not recommended. Using the full namespace avoids confusion and prevents two same identifier names from colliding.
While importing packages, Python looks in the list of directories defined in sys.path, similar as for module search path.

ILLUSTRATION PROGRAM

<u>Word Count of a file:</u>

```
import sys
fname=sys.argv[1]
n=0
with open(fname,'r') as f:
    for line in f:
        words=line.split()
        n+=len(words)
print("Number of words:",n)
```

<u>Copy file:</u>

```
f1=open(“sourcefile.txt”,”r”)
f2=open(“destinationfile.txt”,”w”)
for line in f1:
        f2.write(“\n”+line)
f1.close( )
f2.close( )
print(“Content of Source file:”)
f1=open(“sourcefile.txt”,”r”)
print(f1.read( ))
print(“Content of Copied file:”)
f2=open(“destinationfile.txt”,”r”)
print(f2.read( ))
```