File IO and Exceptions

1DV501 - Introduction to Programming

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

The slides are available in Moodle

September 28, 2020

Remember to ...

The Python Test

- Sign up for the first Python Test on Friday, October 23.
- Read information posted (as News) in Moodle.
- Registration is mandatory. **Deadline October 16.**
- Registration is now open in Moodle (Scroll down a bit to find it.)

Select courses for the spring semester

- All students must apply for spring semester courses
  (Also students in programs where all courses are mandatory.)
- Application period: October 1 to October 15
- Program specific details should have been mailed to you, it is also available in Program Moodle room.

Assignment 3

- Assignment 3 will be published later on today (September 28)
- Assignment 3 should be submitted using Gitlab. **This holds for all students** $\Rightarrow$ campus, distance, as well as others "Others".
- Instructions for how to use Gitlab will be published in Moodle. Post questions in Slack if you have problems.

Today ...

- Working with Files and Directories
- File input/output (IO)
  - Working text files
  - Working with data files
- Errors and Exceptions

Reading instructions: 9.3, 12.1-12.6, 12.8

The os module

```
import os                                   # Operating system module

path = os.getcwd()                          # Get current working directory
print("Current dir:", path)

subdir = os.chdir('jlnmsi_assign1')         # Move to subdir jlnmsi_assign1
print("Moved to dir:", os.getcwd())
```

Output (fully qualified path names):

```
Current dir: /Users/jlnmsi/software/python_courses/1dv501
Moved to dir: /Users/jlnmsi/software/python_courses/1dv501/jlnmsi_assign1
```

- The os module gives support for queries related to files and directories
- `os.getcwd()` ⇒ name of current working directory
  - The virtual machine's start directory in this execution
  - Not same as folder containing this program code
  - Topmost directory inside Visual Studio Code
- `os.chdir('jlnmsi_assign1')` ⇒ change to child directory
- `os.chdir('..')` ⇒ change to parent directory

Files and directories using module `os`

```
import os                              # Operating system module

path = os.getcwd()                     # Get current working directory
print("Current dir:", path)            # ... /1DV501

lst = os.listdir(path)  # List files and directories in path directory
for s in lst:
    print(s)                # .DS_Store, jlnmsi_assign2, jlnmsi_assign1, .vscode

subdir = os.chdir('jlnmsi_assign1')     # Move to subdir jlnmsi_assign1
print("\nMoved to dir:", os.getcwd())   # ... /1DV501/jlnmsi_assign1
lst = os.listdir(subdir)                # List files and folders in subdir
for s in lst:
    if s.endswith(".py"):       # Print files ending with ".py"
        print(s)                # time.py, tax.py, shortname.py, quote.py, intere
```

- `os.listdir(path)` ⇒ List content (as strings) of directory `path`
- Directory content ⇒ files and directories
- Hidden entities (e.g. `.vscode`) have names starting with a `.`

Files and directories using `os.scandir(...)`

Previously: Using `os.listdir(...)`, Now: Using `os.scandir(...)`

```
import os

def is_hidden(entry):
    return entry.name.startswith(".")

def print_entries(list_of_entries):
    for entry in list_of_entries:
        if entry.is_file() and not is_hidden(entry):
            print("File: ", entry.name, type(entry) )
        elif entry.is_dir() and not is_hidden(entry):
            print("Dir: ", entry.name, entry.path)

path = os.getcwd()
entries = os.scandir(path)  # List of entries of type DirEntry
print_entries(entries)      # See output on next slide
print()

subdir = os.chdir('jlnmsi_assign1')
entries = os.scandir(subdir)  # List of entries of type DirEntry
print_entries(entries)        # See output on next slide
```

Output from previous slide

```
/Users/jlnmsi/software/python_courses/1dv501
Dir:  temp /Users/jlnmsi/software/python_courses/1dv501/temp
Dir:  jlnmsi_assign2 /Users/ ... /python_courses/1dv501/jlnmsi_assign2
Dir:  jlnmsi_assign1 /Users/ ... /python_courses/1dv501/jlnmsi_assign1
File:  jlnmsi_assign2.zip <class 'posix.DirEntry'>

File:  time.py <class 'posix.DirEntry'>
File:  tax.py <class 'posix.DirEntry'>
File:  shortname.py <class 'posix.DirEntry'>
File:  quote.py <class 'posix.DirEntry'>
File:  interest.py <class 'posix.DirEntry'>
File:  oddpositive.py <class 'posix.DirEntry'>
File:  sumofthree.py <class 'posix.DirEntry'>
File:  print.py <class 'posix.DirEntry'>
File:  randomsum.py <class 'posix.DirEntry'>
File:  largest.py <class 'posix.DirEntry'>
File:  squarecolor.py <class 'posix.DirEntry'>
File:  area.py <class 'posix.DirEntry'>
File:  fahrenheit.py <class 'posix.DirEntry'>
File:  change.py <class 'posix.DirEntry'>
```

Files and directories - continued

The `os.listdir(...)` approach

- `os.listdir(path)` ⇒ all file and directory names in directory `path`
- Problem: all names are given as strings
  ⇒ hard to know if it is a file or a directory
- Suitable approach when you quickly wants to find the content of a given directory

The `os.scandir(...)` approach

- `os.scandir(path)` ⇒ all files and directories in `path` as `DirEntry` objects
- Each `DirEntry` object `entry` comes with two attributes:
  - `entry.name` ⇒ short local name of file or directory
  - `entry.path` ⇒ fully qualified name of file or directory

  and two methods
  - `entry.is_file()` ⇒ True if `entry` is a file
  - `entry.is_dir()` ⇒ True if `entry` is a directory
- Suitable approach for more complex problems like:
  - List all python files in a given directory
  - Find all sub-directories (transitively) of a given directory

Count subdirectories

```
import os
# Recursive function to count subdirectories to path
def count_dirs(path):
    # print(path)
    no_dir = 1
    entries = os.scandir(path)
    for entry in entries:
        if entry.is_dir():
            no_dir += count_dirs(entry.path)  # Recursive call
    return no_dir

# Program starts
path = "/Users/jlnmsi/Documents/Teaching"
n_dirs = count_dirs(path)

print(f"Dir {path} contains {n_dirs} subdirectories")
# Output: Dir /Users/jlnmsi/Documents/Teaching contains 3620 subdirectories
```

- `count_dirs(path)` is a recursive function that visits all subdirectories
- Visits all subdirectories transitively $\Rightarrow$ subdirs to subdirs to subdirs ...
- Difficult to handle without recursion

Reading text from file

```
import os

path = os.getcwd()
path += "/temp/mamma_mia.txt"
print("Reading from ",path)

file =  open(path,"r")
line_count = 0
for line in file:
    line_count += 1
    print(line)
file.close()
print("Line count: ",line_count)
```

File mamma_mia.txt

```
Mamma mia, here I go again
My my, how can I resist you?
Mamma mia, does it show again
My my, just how much I've missed you?
```

Program output:

```
Reading from ... /1dv501/temp/mamma_mia.txt
Mamma mia, here I go again

My my, how can I resist you?

Mamma mia, does it show again

My my, just how much I've missed you?

Line count:  5
```

- `file =  open(path,"r")` ⇒ open file path for reading (`"r"`)
- `file` is here an object representing a connection to a file
- `for line in file:` ⇒ read from file line by line

Ugly printout since `line` includes a `"\n"` and `mamma_mix.txt` ends with an empty line.

Improved file reading

```
path = ...

file =  open(path,"r")
for line in file:
    print(line.strip())
file.close()
```

Ugly print problem solved by using `print(line.strip())`
$\Rightarrow$ remove trailing `"\n"`

```
path = ...

file =  open(path,"r")
full_text = ""
for line in file:
    full_text += line
file.close()
print(full_text)
```

We first store entire text in a string (including linebreaks)

Reading text is easy, just remember: a) We read the text line by line, b) Lines also includes a final `"\n"`, and c) Empty lines are also included.

It is important to close the file connections (`file.close()`) once reading/writing is done. A non-closed connection might cause problems later on when you try to access a file.

Writing text to a file

```
path = ...
full_text = ...

file = open(path,"w")
file.write(full_text)
file.close()
```

Write entire text to file.
Result: Text in file has same formatting as `full_text`.

```
path = ...
lines = ["do\n","re\n","mi\n","fa\n","so\n","la\n"]

file = open(path,"w")
file.writelines(lines)
file.close()
```

We write text line by line to file.
Result: do,re,mi,fa,so,la as six separate lines.

Writing text is also easy, just remember to handle the line breaks.

Recommendations

- Always look at the content of the file you are about to read to understand how it is organized
- Always open the output file when writing to a file to inspect the result

Reading and Writing text - Summary

We use `open(...)` to make a file connection

- `open(path,"r")` ⇒ open file for reading. Program will crash is file doesn't exists (or is read protected)
- `open(path,"r")` ⇒ open file for writing. The file will be created if it doesn't exist, or replaced if it does exist.
- `open(path,"a")` ⇒ open file for appending ⇒ add new text at the end of a file. The file will be created if it doesn't exist, or appended if it does exist.
- Default is `"r"` ⇒ `open(path)` means open file for reading

`file` in `file = open(...)` is a file object. File object usage:

- `for line in file:` ⇒ read one line at the time
- `full_text = file.read()` ⇒ read entire file content
- `file.write(full_text)` ⇒ write entire text
- `file.writelines(lines)` where `lines` is a list of strings ⇒ write line by line (but not adding any linebreaks)

Safe file handling with `with-as`

```
# Safe file reading
path = ...
with open(path, "r") as file:
    for line in file:
        print( line.strip() )

# Safe file writing
path = ...
with open(path, "w") as file:
    file.write("First line to add\n")
    file.write("Last line to add\n")
```

- `with` and `as` are two Python keywords
- The `with-as` statement includes file closing and was introduced to make sure that an open file is always closed (no matter what happens)
- Although a bit cryptic, it is the recommended approach to open a file.

A 10 minute break?

ZZZZZZZZZZZZZZZZZZ

Runtime Errors

```
def div(a,b):
    return a/b

def m(a,b):
    return div(a,b)

# Program starts
print( m(5,0) )
```

Execution output

```
Traceback (most recent call last):
  File "/Users/jlnmsi/software/1dv501/errors.py", line 8, in <module>
    print( m(5,0) )
  File "/Users/jlnmsi/software/1dv501/errors.py", line 5, in m
    return div(a,b)
  File "/Users/jlnmsi/software/1dv501/errors.py", line 2, in div
    return a/b
ZeroDivisionError: division by zero
```

Error message interpretation: From a call `print( m(5,0) )` in line 8, via a call `div(a,b)` at line 5, we had a `ZeroDivisionError` in line 2.

Hence, the error message not only points out where the error occurred, it also describes the executions trace leading up to the error.

A first look at error handling

```
# Returns a given element in a list
def get_element_at(lst,index):
    if 0 <= index < len(lst):
        return lst[index]
    else:
        return -99  # What else am I supposed to do?

# Program starts
a = list( range(10) )
n = get_element_at(a,15) # Index out of range
```

- The function `get_element_at(lst,index)` returns -99 when used with an index out of range. Is this really the best way to handle a detected error? Or should we let the program crash?
- In general, how do we handle errors due to an incorrect use of a function?

Exceptions - A first example

```
try:
  x = 5*y              # y is not defined
  print(" x =", x)
except NameError:
  print("An exception occurred")
```

Output: `An exception occurred`

- `try` and `except` are two Python keywords used for exception handling
- Errors occurring in the `try` block can be handled in the `except` block
- Using this approach we can avoid ugly traceback printouts (see slide 15).
- We can also decide to take some action (e.g. try again) when an error occurs.

Another exception example

```
def div(a,b):
    return a/b  # Error if b = 0

def m(a,b): return div(a,b)

# Program starts
try:
    x, y = 5,0
    div = m(x,y)
    print(f"{x} divided by {y} is {div}")
except ZeroDivisionError:
    print("Division by zero")
```

Output: `Division by zero`

- Errors occurring due to code or calls executed in the `try` block
- ... can be handled in the enclosing `except` block
- The execution jumps directly from the error (in function `div(a,b)`) to the `except` block ⇒ `print(f"{x} ...")` is not executed.

One more exception example

```
repeat = True
while repeat:
    x = int( input("Enter integer x: "))
    y = int( input("Enter integer y: "))
    try:
        result = x/y        # Error if y = 0
        print(f"{x} divided by {y} is {result}")
        repeat = False      # Terminates loop
    except ZeroDivisionError:
        print("Dividing by zero, try again ...\n")
```

- The program will keep asking the user for input as long y is zero.
- Each time zero is entered for y, the error message `Dividing by zero, try again ...` will be displayed.

Raising exceptions

```
# A function with error handling
def get_element_at(lst,p):
    if 0 <= p < len(lst):
        return lst[p]
    else:
        err_msg = f"Index {p} not in valid range [0,{len(lst)-1}]"
        raise IndexError(err_msg)  # We raise an exception

# Program starts
try:
    a = list( range(10) )
    n = get_element_at(a,15) # Index out of range
except IndexError as e:
    print("An error has occurred!")
    print(type(e)," ==> ",e)
```

Output:

```
An error has occurred!
<class 'IndexError'>  ==>  Index 15 not in valid range [0, 9]
```

Raising exceptions - A tedious example

```
def input_odd_int():
    s = input("Enter an odd integer: ")
    try:
        n = int(s)          # Fails if s not an integer
    except ValueError:
        raise ValueError("The input must be an integer!")
    if n%2 == 0:
        raise ValueError("The integer must be odd!")
    return n

# Program starts
try:
    n = input_odd_int()
    print("A valid input is", n)
except ValueError as e:
    print(type(e),"==>",e)
```

A function that raises a `ValueError` if input is a non-integer or an even number. Do **not** use this approach in assignment if not explicitly asked for.

Exceptions: Basics

- Python handles all errors and abnormal conditions using *exceptions*.
- An exception is an object that encapsulates information about an error.
- Error ⇒ program *raises* (or *throws*) an exception.
  (e.g., `raise IndexError(err_msg)`)
- ⇒ execution halts immediately
- ⇒ call stack is unwounded until an appropriate enclosing exception handler is found (e.g., `except IndexError as e:`).
- No enclosing exception handler ⇒ The virtual machine catches exception, abruptly terminates program, and prints a stack trace
- Advantages
  - Uniform handling of all abnormal conditions
  - Separation of responsibilities:
    The programmer identifies problems and raises exceptions.
    The client (or user) determines how to handle the problem
    (ignore and continue, recover, try again, exit, ...).
  - Here we talk about a *programmer* responsible for the development of a software component, and a *user* or *client* (most likely also a programmer) that uses the component.

An Unspoken Contract

Background

- The programmer can't know how a user wants to deal with an error.
- Different users and situations ⇒ different types of error handling.

An Unspoken Contract

- The programmer is responsible for identifying errors and to notify the user by raising an exception.
- The user/client decides how to handle the exception.

Example: The function `get_element_at(lst,index)`

- The programmer finds the faulty index (outside the range) ⇒

```
raise IndexError("Index out of range: " + str(index))
```

- The function user can (if he/she likes) catch and handle the error

```
try:
    a = list( range(10) )
    n = get_element_at(a,15) # Index out of range
except IndexError as e:
    print("An error has occurred!")
    print(type(e)," ==> ",e)
```

Handling multiple types of exceptions

We might have several except blocks. Each one handling a specific type of errors

```
try:
    ...
except IndexError as e:
    "Do somethong with e"
except ValueError as e:
    "Do somethong with e"
except Exception as e:
    "Do somethong with e"
finally: # Always executed
    "Save what is possible. Close database connections, networks and so on"
```

- Repeated except ⇒ the first suitable is used.
- Exception is the base class for all exceptions ⇒ handles everything
- The finally block is always executed. It is mainly used to save what possibly can be saved before the program crashes.

Built-in Errors to chose from

There are a number of built-in errors to chose from:

- IndexError is thrown when trying to access an item at an invalid index.
- ImportError is thrown when a specified function can not be found.
- TypeError is thrown when an operation or function is applied to an object of an inappropriate type.
- NameError is thrown when an object could not be found.
- ZeroDivisionError is thrown when the second operator in the division is zero.
- ValueError is thrown when a function's argument is of an inappropriate type.
- Exception handles all type of exceptions $\Rightarrow$ catches everything
- ... and many more.

Hence, when you want to raise an exception, select a suitable error type, and put together a suitable error message. The just `raise ErrorType(err_msg)`.

Always position `except Exception` last if you are catching multiple error types.

Handling IOErrors

```
import os

# Safe file reading handling IOErrors
path = os.getcwd()
path += "/temp/mamma_PIA.txt"        # File name error!
try:
    with open(path, "r") as file:
        for line in file:
            print( line.strip() )
except IOError as e:
    print(type(e),"==>",e)
    print("No such file: ",path)
```

- File IO often results in errors. For example, reading from a non-existing (or read protected) file.
- Thus, enclosing all critical file IO operations with a try-except block is a very common programming pattern.

Exceptions summary

In general

- By enclosing error prone code with a try-except block we can catch and handle errors.
- By raising exceptions we can inform a user of an error
- Basic idea: The programmer is responsible for identifying errors and to notify the user by raising an exception. The user/client decides how to handle the exception.

Are exceptions important?

- Exception handling is very important in larger (commercial) systems since we don't want our customers to experience an ugly stack trace due to an unhandled exception. A commercial system should never crash.
- It is less important for smaller projects when the user and programmer often is the same group of persons. After a crash we simply try to fix the problem and run the program again.
- Python is very liberal when it comes to exception handling. The programmer decides when and if to handle a potential exception. Other languages (like Java) is much stricter. Certain operations (like File IO) must include exception handling.