15-110 Principles of Computing – F19

Lecture 24:
Files I/O 3, Handling Exceptions

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Moving the reading head: `seek()` and `tell()` methods

- Get the current position (in bytes) in the file from the beginning (position 0):

  `position = file_handle.tell()`

```
f = open('data.txt', 'r')
data = f.read(11)
pos = f.tell()          # pos has value 11
data = f.read(19)
pos = f.tell()          # pos has now value 30
```

- Go to the given position (in bytes) in the file from the beginning (position 0):

  `position = file_handle.seek(pos, <from_where>)`

```
f = open('data.txt', 'r')
pos = f.seek(30)          # pos has value 30
data = f.read(10)
pos = f.tell()          # pos has now value 40
```

Only for binary files:

- `from_where`'s default value is 0, meaning from the beginning
- `from_where` = 1 means relative to current positions
- `from_where` = 2 means relative to end

Read a file record-by-record using a for loop

- A text file is organized in **records / lines** separated by newlines: it is possible to iterate over all records/lines, that are string types
- Individual **fields** inside a record need to be extracted based on the knowledge of the structure of the record

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```

```
f = open('data.txt', 'r')
for record in f:
   print(record, end = '')    # the end option removes the default newline from the record string
```

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```

```
f = open('data.txt', 'r')
for record in f:
    print(record)
```

```
This line is 26 characters

Line 1: 0.1 5.4 2 4 20 .03

Line 2: 1 3. 2.0 43 12

Line 3: 1 2

This is the last record
```

Read individual records / lines in a file: `readline()` method

- **Individual records / lines** can be read by invoking the `readline()` method
- The function returns the read string (that includes a newline \n)
- If the function is called at the EOF it returns an empty string (' ')

```
f = open('data.txt', 'r')
bytes_so_far = 0
record = f.readline() # 'This line is 26 characters\n'
bytes_so_far += len(record)
record = f.readline() # 'Line 1: 0.1 5.4 2 4 20 .03\n'
bytes_so_far += len(record)
record = f.readline() # 'Line 2: 1 3. 2.0 43 12 \n'
bytes_so_far += len(record)
f.seek(bytes_so_far + 5)
record = f.readline() # ': 1 2\n'
record = f.readline() # 'This is the last record\n'
```

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```

Read all remaining records / lines in a file: `readlines()` method

- **All records / lines** in the file from the current position can be read using the `readlines()` method
- The method **returns a list of strings**, where each string is a consecutive line/ record of the file

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```

```
f = open('data.txt', 'r')
record = f.readline() # 'This line is 26 characters\n'
record = f.readline() # 'Line 1: 0.1 5.4 2 4 20 .03\n'
record = f.readlines()
```

`record` is the list: [`'Line 2: 1 3. 2.0 43 12\n',`
`'Line 3: 1 2\n',`
`'This is the last record'`]

Write in a file: `write( )` method

- To write something into a file, the open function must be invoked with one of the mode flags: `w, a, x, r+, w+, a+`
- Opening with `w` *erases* file's content if file exists, writing starts at the (new) beginning
- Opening with `a` lets writing start at the end of the file (appending)
- Opening with `r+` lets writing start at the beginning of the file (overwriting)
- If a file doesn't exist, `w, a, x` will create it
- The `+` versions allow both writing and reading

➢ **Write** data to a file open with a writing flag:

```
written_bytes = file_handle.write(string_to_write)
```

```
f = open('data.txt', 'a')
nbytes = f.write('New line: 0 3 5.5')
nbytes = f.write('Another new line: 1 2 3')
```

Write in a text file: newline characters and mode flags

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```

```python
f = open('data.txt', 'a')
nbytes = f.write('New line: 0 3 5.5')
nbytes = f.write('Another new line: 1 2 3')
```

17 bytes

23 bytes

There was no \n (newline) at the end of the last record

There is no \n (newline) at the end of the first newly appended record

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last recordNew line: 0 3 5.5Another new line: 1 2 3
```

Write in a text file: newline characters and mode flags

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```

```
f = open('data.txt', 'w')
nbytes = f.write('New line: 0 3 5.5\n')
nbytes = f.write('Another new line: 1 2 3\n')
```

18 bytes

24 bytes

The w mode flag causes the *erase* of the existing file

There is a \n (newline) at the end of the two newly appended records

```
New line: 0 3 5.5
Another new line: 1 2 3
```

Write in a text file: newline characters and mode flags

```
f = open('data.txt', 'x')
```

`data.txt` exists in the file system, the open returns with an error!

```
f = open('new_data.txt', 'x')
nbytes = f.write('New line: 0 3 5.5\n')
nbytes = f.write('Another new line: 1 2 3\n')
```

```
New line: 0 3 5.5
Another new line: 1 2 3
```

`new_data.txt` is now a new file in the file system

Inspecting the access mode of a file: `readable()`, `writable()`

- Check weather a file is open with **read mode flag** or not:

```
read_mode = file_handle.readable()
```

`True` is returned when file is readable, `False` otherwise

- Check weather a file is open with **write mode flag** or not:

```
write_mode = file_handle.writable()
```

`True` is returned when file is writable, `False` otherwise

```
f = open('numbers.txt', 'r')
if f.readable():
    data = f.readlines()
    print(data)
if f.writable():
    f.write('Add another record')
```

Closing a file after use: `close()` method

➢ **Close** a file when no further operations are needed / allowed:

```
file_handle.close()
```

- Closing a file frees up used file resources (and let the file accessible for deleting/renaming by the OS)
- If a close() isn't explicitly called, python's garbage collector does eventually the job of closing the file
- Explicitly closing the file prevents the program to perform any (unwanted) further operations on file
- close() returns `None`

```
f = open('numbers.txt', 'w')
f.write('This file contains important data\n')
f.write('0 1 2 3 4\n')
f.write('4 3 2 1\n')
f.close()
```

Problems with file operations that can result into errors

- Trying to open (with a read flag) a non-existing file results into an error (we can check this first with `os` methods)
- Trying to perform operations in a file for which we don't have the right permissions result into an error (again, we can avoid this by using the os methods)
- Trying to perform a read / write operation on an already closed file results into an error (no way to overcome this with `os` methods)
- How do we deal "flexibly" with these situations that could generate errors?
- A more general question:

Can we **try out** operations that could generate an error
without having the program being aborted whenever the error is actually generated?

Dealing with errors: `try-except-else-finally` construct

- When an **error** occurs during the program, Python generates an **exception:** it generates an error type that identifies the exception and then **stops** the execution
- Exceptions can be handled using the **try statement** to avoid that the program does actually stop when an error occurs during the execution

- `try-except-else-finally` blocks:
  - ✓ The **try** block let executing a block of code that can potentially generate an exception
  - ✓ The **except** block let handling the error, if generated by the try block (i.e., what to do when an error occurs)
  - ✓ The **else** block let specifying a block of code that is executed if the try block *didn't generate any exception*
  - ✓ The **finally** block let executing the code, regardless of the result of the try- and except blocks.

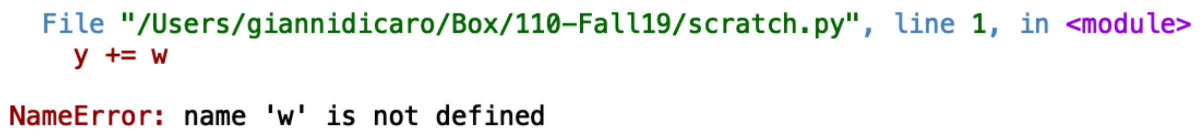
Optional

```
y = 1
try:
    x /= 10
    y += x
except:
    print("x doesn't exist")
else:
    print('x:', x)
    del x
finally:
    print('y:', y)
```

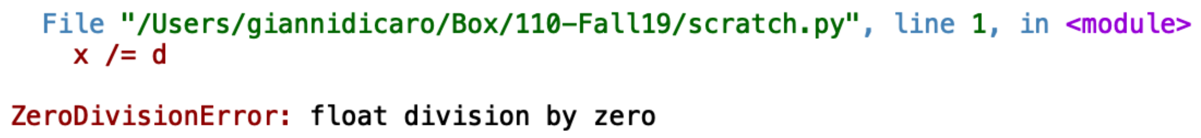
Catching multiple exceptions

✓ **Multiple, different exceptions can be caught**

```
y = 1
x = 10
d = 0.0
try:
    x /= d
    y += x
except NameError:
    print(”Variable doesn't exist")
except ZeroDivisionError:
    print("Division by zero!")
```

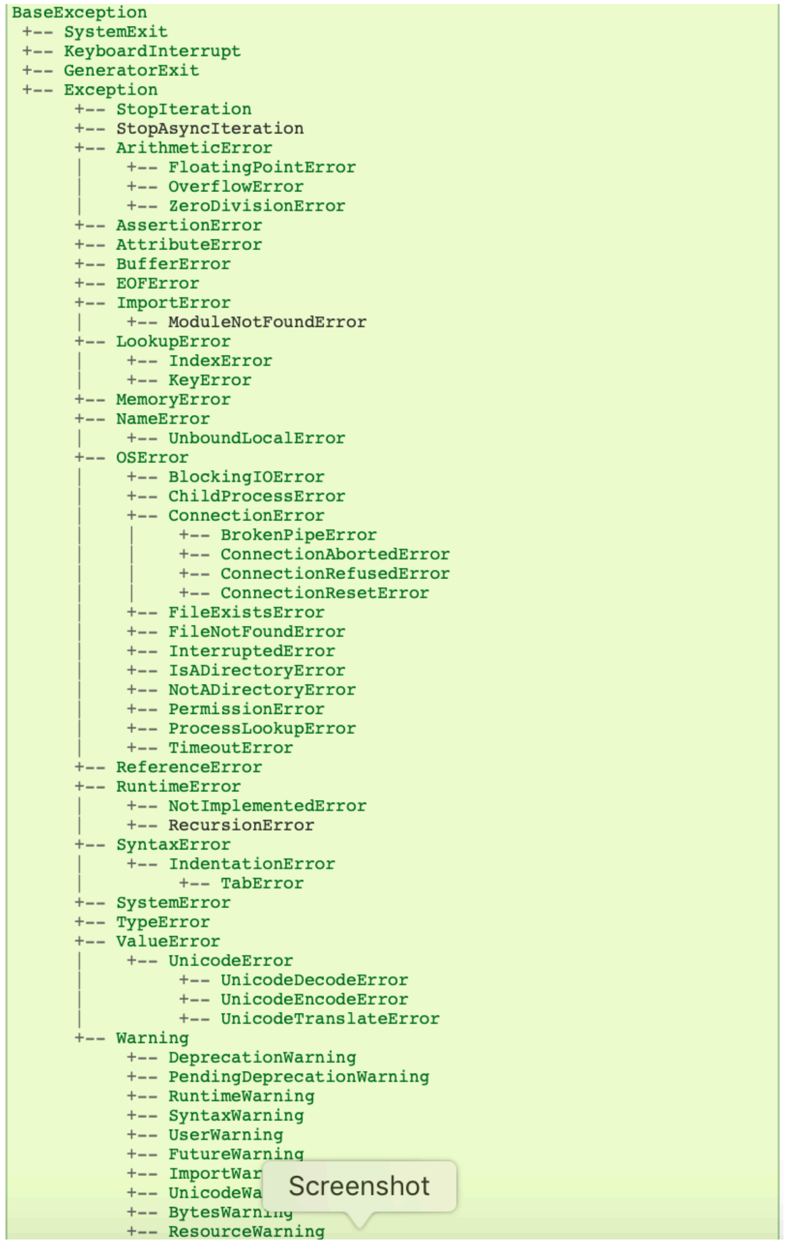


```
  File "/Users/giannidicaro/Box/110-Fall19/scratch.py", line 1, in <module>
    y += w

NameError: name 'w' is not defined
```



```
  File "/Users/giannidicaro/Box/110-Fall19/scratch.py", line 1, in <module>
    x /= d

ZeroDivisionError: float division by zero
```

List of python's exceptions



```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWar
           +-- UnicodeWa
           +-- BytesWarning
           +-- ResourceWarning
```

Exceptions list:

https://docs.python.org/3/library/exceptions.html

Catching multiple exceptions

✓ **Multiple, different exceptions can be caught, but only a few may need to be explicitly named**

```
y = 1
x = 10
d = 0.0
try:
    x /= d
    y += x
except ZeroDivisionError:
    print("Division by zero!")
except:
    print("Something went wrong!")
```

Try/Catch with files

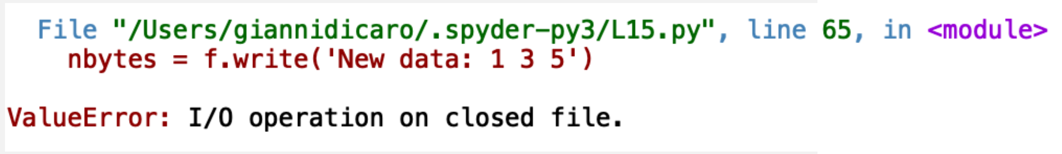
- ✓ **Try to open a file for writing, otherwise open a different file if it fails, and at the end always issue a close()**

```
try:
    f = open('sales.dat', 'r')
except FileNotFoundError:
    print('File sales.dat does not exist in the current folder')
    print("I will open another file, that it's for sure in the system")
    f = open('all_sales.dat', 'r')
except:
    print('File sales.dat does exist but it is not readable')
else:
    print("Add data to the file")
    f.write ('New sale: 8000')
finally:
    print("Files must be closed, no error will be thrown if open failed")
    f.close()
```

Try/Catch with files

✓ **Try to write on a file, reopen it if it was previously closed**

```
try:
    nbytes = f.write('New data: 1 3 5')
except ValueError:
    print('File was previously closed! To write, I will reopen it')
    f = open('sales.dat', 'a+')
    nbytes = f.write('New data: 1 3 5')
except:
    print('File sales.dat does exist but it is not readable')
finally:
    print("Let’s close the file anyway")
    f.close()
```



```
  File "/Users/giannidicaro/.spyder-py3/L15.py", line 65, in <module>
    nbytes = f.write('New data: 1 3 5')

ValueError: I/O operation on closed file.
```

Generating custom exceptions: `assert`

- **Raise an error if an expression is evaluated False**: `assert Expression<, argument>`
- The argument is optional, in its absence no custom message is generated
- ✓ **Sanity-check:** if something is wrong generates an AssertionException error with a user-defined argument
- ✓ The error can be dealt with `try-except`, otherwise it will just abort the program

```
def KelvinToCelsius(temperature):
    assert (temperature >= 0),"Negative Kelvin!"
    return (temperature-273.15)

print (KelvinToCelsius(273))
print (KelvinToCelsius(-5))
```

```
Celsius:   -0.150
Traceback (most recent call last):

  File "<ipython-input-182-d1dbd701d3ba>", line 7, in <module>
    print ("Celsius: {:8.3f} ".format(KelvinToCelsius(-5)))

  File "<ipython-input-182-d1dbd701d3ba>", line 2, in KelvinToCelsius
    assert (temperature >= 0),"Negative Kelvin! "

AssertionError: Negative Kelvin!
```

```
try:
    print ("Celsius: {:8.3f} ".format(KelvinToCelsius(273)))
    print ("Celsius: {:8.3f} ".format(KelvinToCelsius(-5)))
except AssertionError:
    print ("Provide a Kelvin temperature >= 0")
```

```
Celsius:   -0.150
Provide a Kelvin temperature >= 0
```