

Python Programming

Lesson 2 – Debugging

Lesson 2 - Outline

- More About File Handling
- Debugging

More About File Handling

Retrieve Filename and Directory

Retrieve Filename and Path

- **Retrieve Filename**

- **Syntax**

`os.path.basename(path)`

- **Example 1**

`os.path.basename("C:/Python/test_write.txt")`

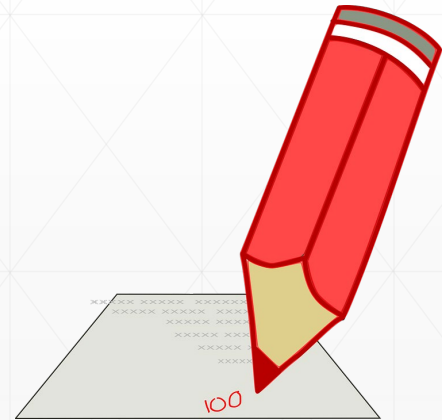
- **Explanation**

Return the basename of the file
(i.e. usually return the file name from the given path)

Retrieve Filename and Path

- Retrieve Filename

Let's try the exercise together!



Retrieve Filename and Directory

- **Retrieve Directory**

- **Syntax**

`os.path.dirname(path)`

- **Example 2**

`os.path.dirname("C:/Python/test_write.txt")`

- **Explanation**

Return the directory (*includes the path*) from the given path

Get File Size

- **Get file size**

- **Syntax**

`os.path.getsize(path)`

- **Example 3**

`os.path.getsize("C:/Python/test_write.txt")`

- **Explanation**

Return the size, in bytes, of path.

(Raise `os.error` if the file does not exist or is inaccessible)

Error handling will be covered at later session

How to Use Relative Path

- **Use Relative Path**

- **Syntax**

```
dirname = os.path.dirname(__file__)
```

```
filename = dirname + "\\path\\to\\join\\filename"
```

- **Example 4**

```
dirname = os.path.dirname(__file__)
```

```
filename = dirname + r"\\test\\test.txt")
```

- **Explanation**

file is a variable that contains the file that being executed (includes the current path)

How to Use Relative Path

- Use Relative Path (Advance)
- Syntax

`os.path.join(path, *paths)`

Try to explore the behavior of `os.path.join()` yourself

List all the files in a directory

- List all the files in a directory

- Syntax

```
import glob
for name in glob.glob("path or filename"):
    print(name)
```

- Example 5

```
import glob
for name in glob.glob("C:/Python/*"):
    print(name)
```

- Explanation

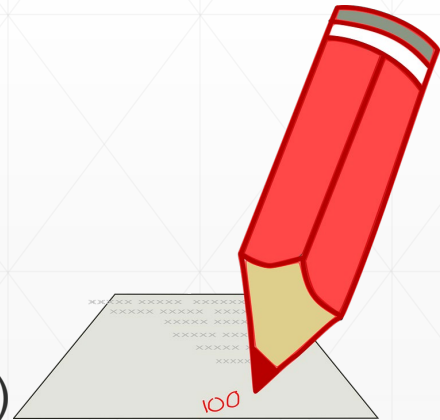
Glob module can retrieve files/pathnames matching a specified pattern and recursively

List all the files in a directory

- **More about glob**

Let's try the following cases together

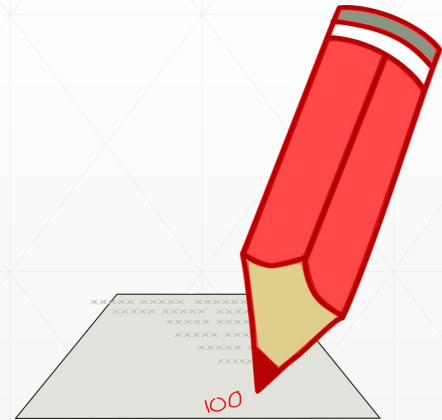
- **Case 1:**
Give the full path for a file name
- **Case 2:**
Use "*" wildcard
- **Case 3:**
Use "?" wildcard
- **Case 4:**
Use "**" implies recursive
e.g. `glob.glob("C:/**/test.txt", recursive = True)`



List all the files in a directory

- List all the files in a directory

Let's try the exercise together!



Check word count in a file

- **Check Word Count in a file**

- **Syntax**

```
file = open(filename, mode)
data = file.read()
words = data.split()
```

- **Example 6**

```
file = open("C:\Python\test.txt", "r")
data = file.read()
words = data.split()
print("How many words in text file :", len(words))
```

- **Explanation**

The **split()** method splits a string into a list. You can specify the separator, default separator is any whitespace.

Debugging

Debugging

- What is Error and Exception
- Introduce **Try – Except – Else – Finally** block
- Introduce **Raise** usage
- Debugging Techniques

Error and Exception

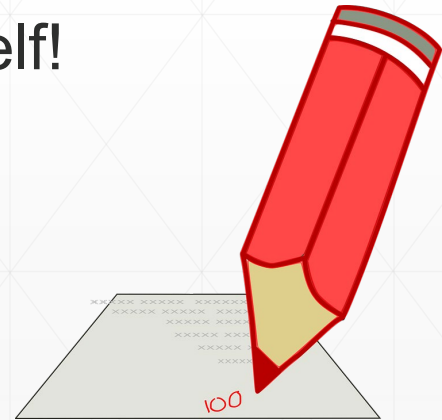
Error and Exception

- In Python, under **Errors** cannot be handled while **Exceptions** can be handled during the runtime
- Common Error Types:
 1. Syntax Error
 2. Indentation Error
 3. Out of Memory Error
 4. Recursion Error

Error and Exception

- **Syntax Error**
- Mistakes, for example, missing an operator, a quotation mark or misspelling a keyword

Try to produce a syntax error yourself!



Error and Exception

- **Indentation Error**

- Actually indentation error is one of the syntax error caused when the indentation is not appropriate, for example, while using if statement / for loop.

Try to produce an indentation error yourself!



Error and Exception

- **Out of Memory Error**
- Occur when having large objects in memory and not enough memory (e.g. in RAM)

Error and Exception

- **Recursion Error**
- Invoke when too many methods, one inside another is executed (one with an infinite recursion)
- **Recursion Error - Example 7**

```
def recursion():  
    return recursion()  
  
recursion()
```

Error and Exception

- Python has a list of built-in exceptions (Runtime errors):
- Common Exception Types:

Exception	Description
AttributeError	Raised on the attribute assignment or reference fails.
EOFError	Raised when the input() function hits the end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raised when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the 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 the interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in the local or global scope.
NotImplementedError	Raised by abstract methods.
OverflowError	Raised when the 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.
StopIteration	Raised by the next() function to indicate that there is no further item to be returned by the iterator.
SystemExit	Raised by the sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of an 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 translation.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of a division or module operation is zero.

Error and Exception

- **Zero Division Error**
- Occur when the divisor is zero
- **Zero Division Error - Example 8**

```
result = 100 / 0  
print(result)
```


Error and Exception

- **Overflow Error**

- Occur when the result of an arithmetic operation is out of range

- **Overflow Error - Example 9**

```
import math  
print(math.exp(999))
```

Error and Exception

- **Import Error**

- Import a module that does not exist or unable to load in its standard path. Also can be caused by making a typo in the module's name

- **Import Error - Example 10**

```
import abcdef
```

Error and Exception

- **Index Error and Key Error**

- Occur when an index or key used on a mapping of a list or dictionary is invalid or not found

- **Index Error in List - Example 11**

```
a = ['a', 'b', 'c']  
print (a[3])
```

- **Key Error in Dictionary - Example 12**

```
a = {1:'a', 2:'b', 3:'c'}  
print (a[4])
```

Error and Exception

- **Name Error**
- Occur when a local or global name is not found (defined)
- **Name Error - Example 13**

```
print(result)
```

Error and Exception

- **Type Error**
- Occur when unrelated types of operands or objects are combined
- **Type Error - Example 14**

```
a = 10  
b = "string"  
c = a + b
```

Error and Exception

- **Value Error**
- Occur when the built-in operation or a function receives an argument is invalid value
- **Value Error - Example 15**

```
print(float("string"))
```

Introduce Try – Except – Else – Finally block

Introduce Try – Except – Else – Finally block

- **Exception Handling**

- Important! Because it helps maintain the normal, desired flow of the program even when unexpected events occur

- **Four main components of exception handling**

1. Try
2. Except
3. Else
4. Finally

Introduce Try – Except – Else – Finally block

- **Try**
 - Run the code block in which you expect an error to occur
- **Except**
 - Define the type of exception you expect in the try block (built-in or custom)
- **Else**
 - If no exceptions occur, then this block of code will be executed
- **Finally**
 - No matter there is an exception or not, this block of code will always be executed. (Suitable to use for those object need to invoke close())

Introduce Try – Except – Else – Finally block

- Try-Except - Example 16

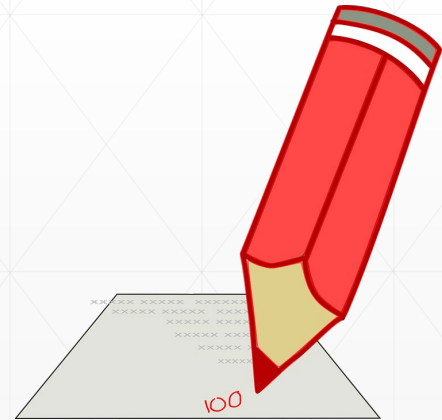
try:

```
a = 10 / 0  
print(a)
```

```
except ZeroDivisionError:  
    print ("Divided by Zero!!!")
```

Introduce Try – Except – Else – Finally block

- Try to use another exception for Try-Except syntax



Introduce

Try – Except – Else – Finally block

- Try-Except-Else - Example 17

try:

```
a = int(input("Enter an integer: "))  
b = 10 / a  
print(b)
```

except ZeroDivisionError:

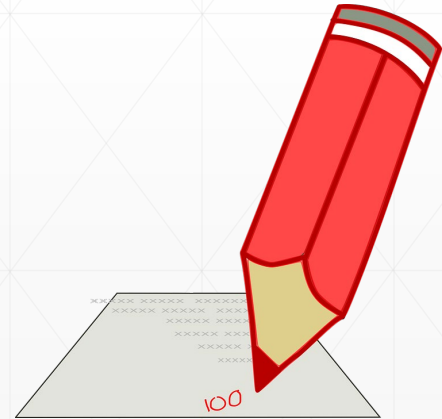
```
print ("Divided by Zero!!!")
```

else:

```
print ("No error.")
```

Introduce Try – Except – Else – Finally block

- Try to use another exception for Try-Except-Else syntax



Introduce

Try – Except – Else – Finally block

- Try-Except-Else-Finally - Example 18

try:

```
a = int(input("Enter an integer: "))  
b = 10 / a  
print(b)
```

except ZeroDivisionError:

```
print ("Divided by Zero!!!")
```

else:

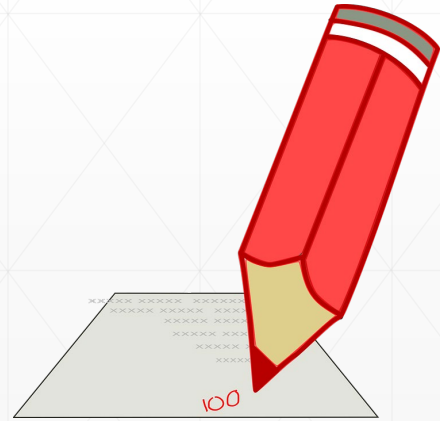
```
print ("No error.")
```

finally:

```
print ("Execute no matter exception occurs or not")
```

Introduce Try – Except – Else – Finally block

- Try to use another exception for Try-Except-Else-Finally syntax



Introduce Raise usage

Introduce Raise usage

- **Raise in Exception Handling**

- Raising exceptions allows us to distinguish between regular events and something exceptional

- **Raise - Example 19**

```
if int(input()) < 0:  
    raise Exception("Input non-negative number, please.")
```

Debugging Techniques

Debugging Techniques

1. Add some inline statement

- Debugging - Example 20

```
print("Start – copy files")  
...  
print("End – copy files")
```

- Debugging - Example 21

```
print("This is the before value of x: ", x)  
...  
print("This is the after value of x: ", x)
```

Debugging Techniques

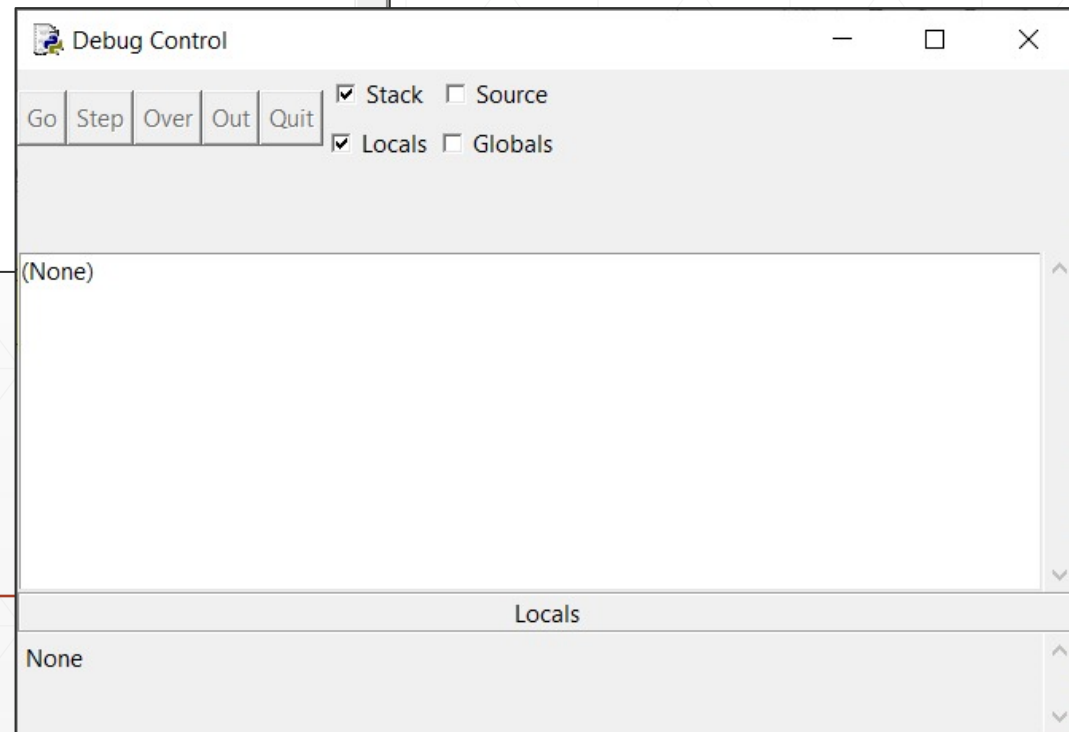
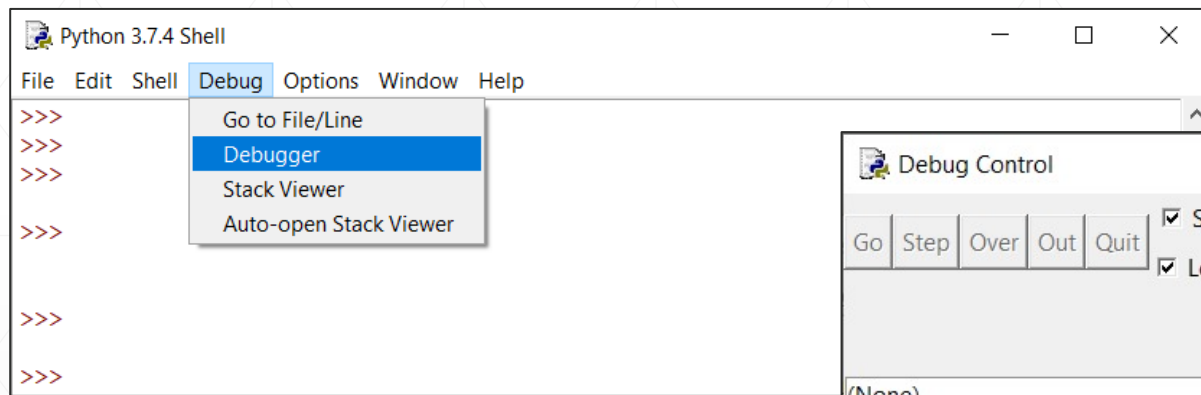
2. Using Debugger

- IDLE has a build-in debugger which is very useful for stepping through a program and watching the variables change values
 - Add breakpoint to an appropriate position is important
 - Five Buttons in IDLE Debugger
 1. **Go** - Executes the rest of the code as normal, or until it reaches a break point
 2. **Step** - Step one instruction. If the line is a function call, the debugger will step into the function
 3. **Over** - Step one instruction. If the line is a function call, the debugger won't step into the function, but instead step over the call
 4. **Out** - Keeps stepping over lines of code until the debugger leaves the function it was in when Out was clicked. This steps out of the function
 5. **Quit** - Immediately terminates the program
-

Debugging Techniques

2. Using Debugger

- Step 1 – In Python Shell, click Debug > Debugger, the Debug Control will prompt out.

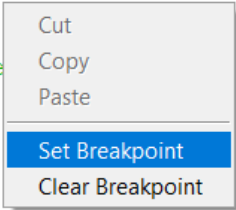


Debugging Techniques

2. Using Debugger

- Step 2 – In Python Program (i.e. the *.py file, right click on the line and choose “Set Breakpoint”. Yellow highlight indicated the lines with breakpoint.

```
a = 1  
b = 2  
c = a + b  
print(c)  
print("Finished")
```



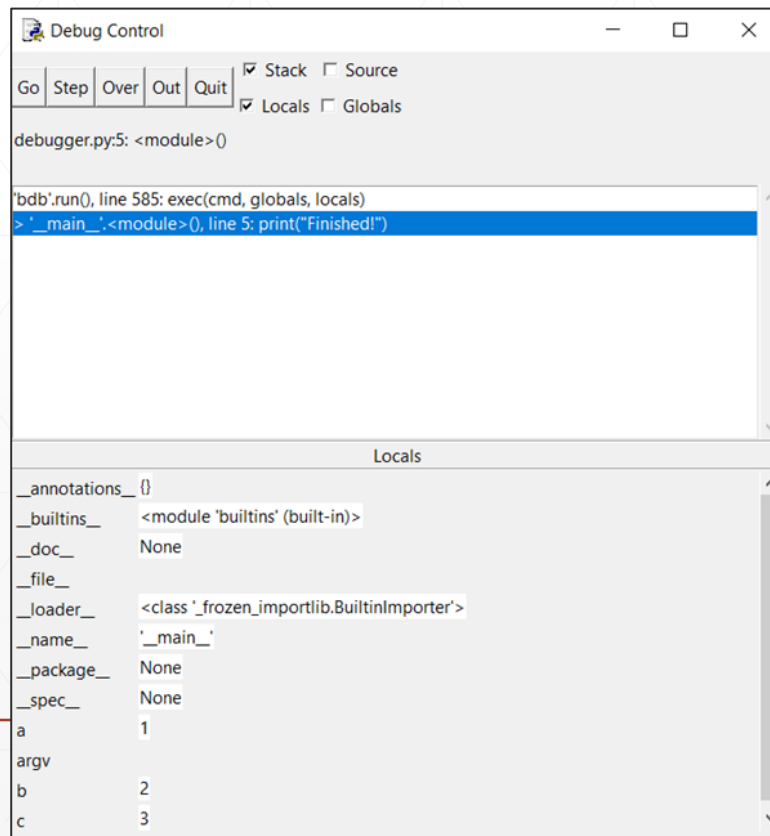
Cut
Copy
Paste
Set Breakpoint
Clear Breakpoint

```
a = 1  
b = 2  
c = a + b  
print(c)  
print("Finished!")
```

Debugging Techniques

2. Using Debugger

- Step 3 – In Debug Control, view the variables and try “Go”, “Step”, “Over”, “Out” and “Quit” together



Thank you