Python Object-Oriented Program with Libraries

Unit 2: I/O, File System and Exceptions

CHAPTER 1: EXCEPTIONS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Topics

1. Overview
2. Exception Handling
3. Exception Classes
4. Custom-Design Exceptions
5. Advanced Topics: finally, Re-throwing, and Chained Exception Handling

Overview for Exception Handling

LECTURE 1

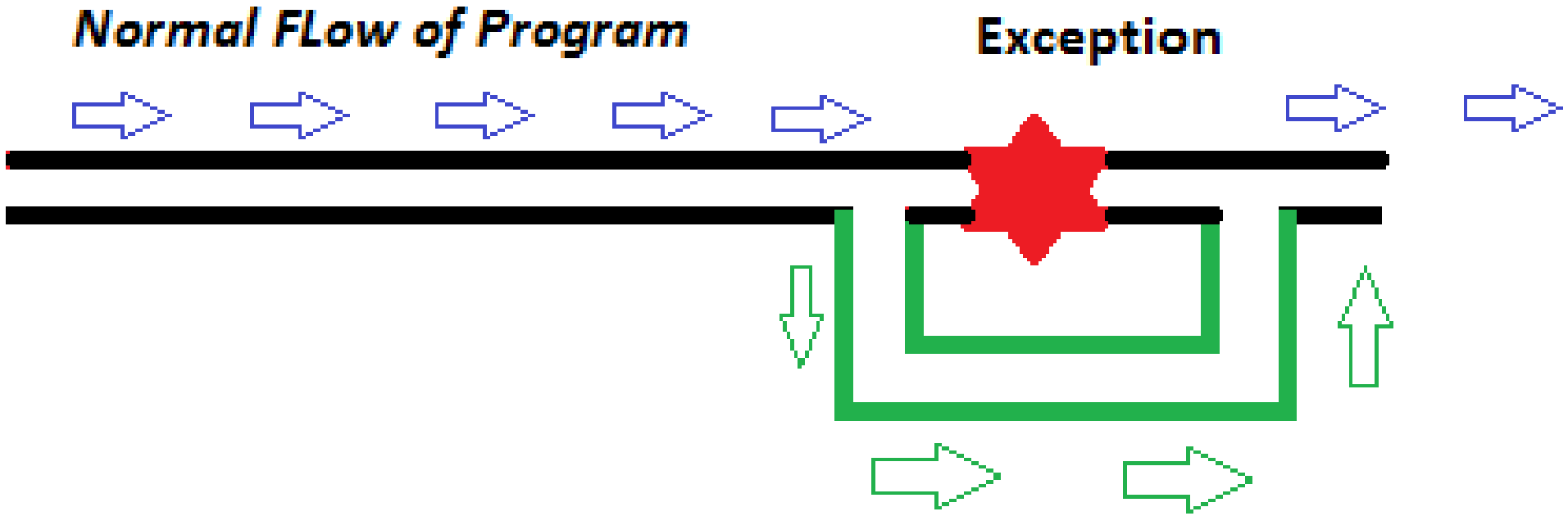
```
# oops.py
#
# ICS 32 Fall 2017
# Code Example
#
# This module contains a Python program that always crashes.  Run this
# module and take a look at its output (a traceback).  A valuable skill
# as a Python programmer is the ability to read a traceback and use it
# to help diagnose the cause of unexpected crashes.

def f():
    x = 3
    g(x)

def g(n):
    print(len(n))

if __name__ == '__main__':
    f()
```

```
C:\Python\Python36\python.exe "C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/PyDev/U2 IO System/Chapter1/Exception/oops.py"
Traceback (most recent call last):
  File "C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/PyDev/U2 IO System/Chapter1/Exception/oops.py", line 22, in <module>
    f()
  File "C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/PyDev/U2 IO System/Chapter1/Exception/oops.py", line 14, in f
    g(x)
  File "C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/PyDev/U2 IO System/Chapter1/Exception/oops.py", line 18, in g
    print(len(n))
TypeError: object of type 'int' has no len()
```

oops.py

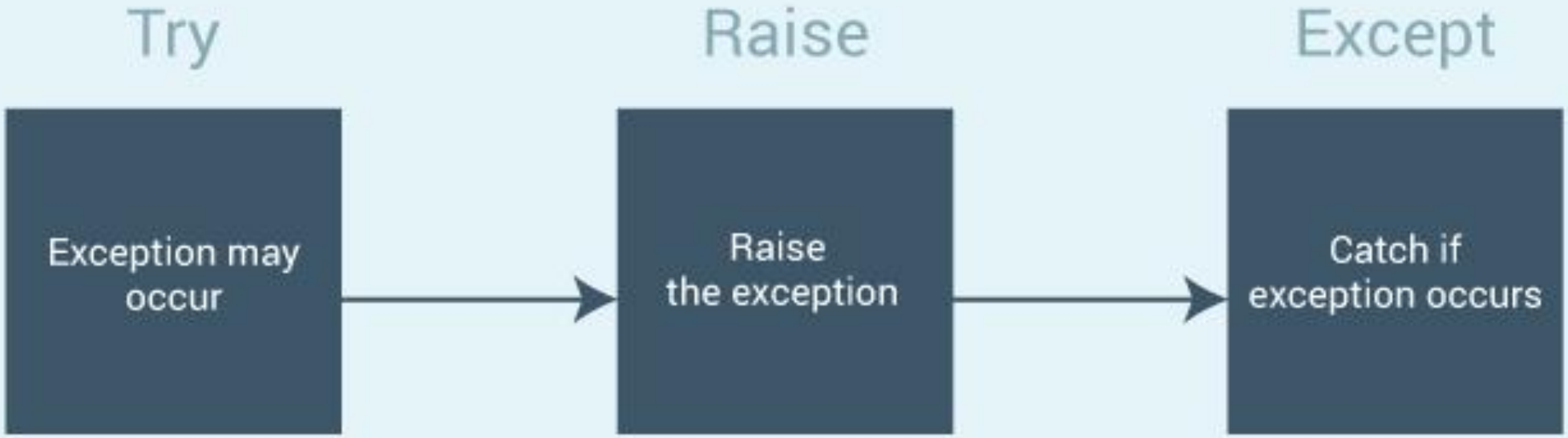
When you see a traceback like this, it's important to actually pay attention to what it says. Reading a traceback from the bottom up provides a lot of useful information, even if you don't quite understand the error message at first, because tracebacks don't only tell us *what* the error was, but also *where* the error occurred.

- The type of the exception — exceptions are objects in Python, just like everything else — is a **TypeError**.
- A more descriptive account of the problem is **object of type 'int' has no len()**. That's a hint that we were trying to get the length of an integer, but that integers have no length.
- The exception was actually raised by code in the function **g()**. (The traceback even shows us the code that's on the statement **print(len(n))**.)
- The function **g()** had been called by the function **f()**.
- The function **f()** had been called by the "main" **if** statement.

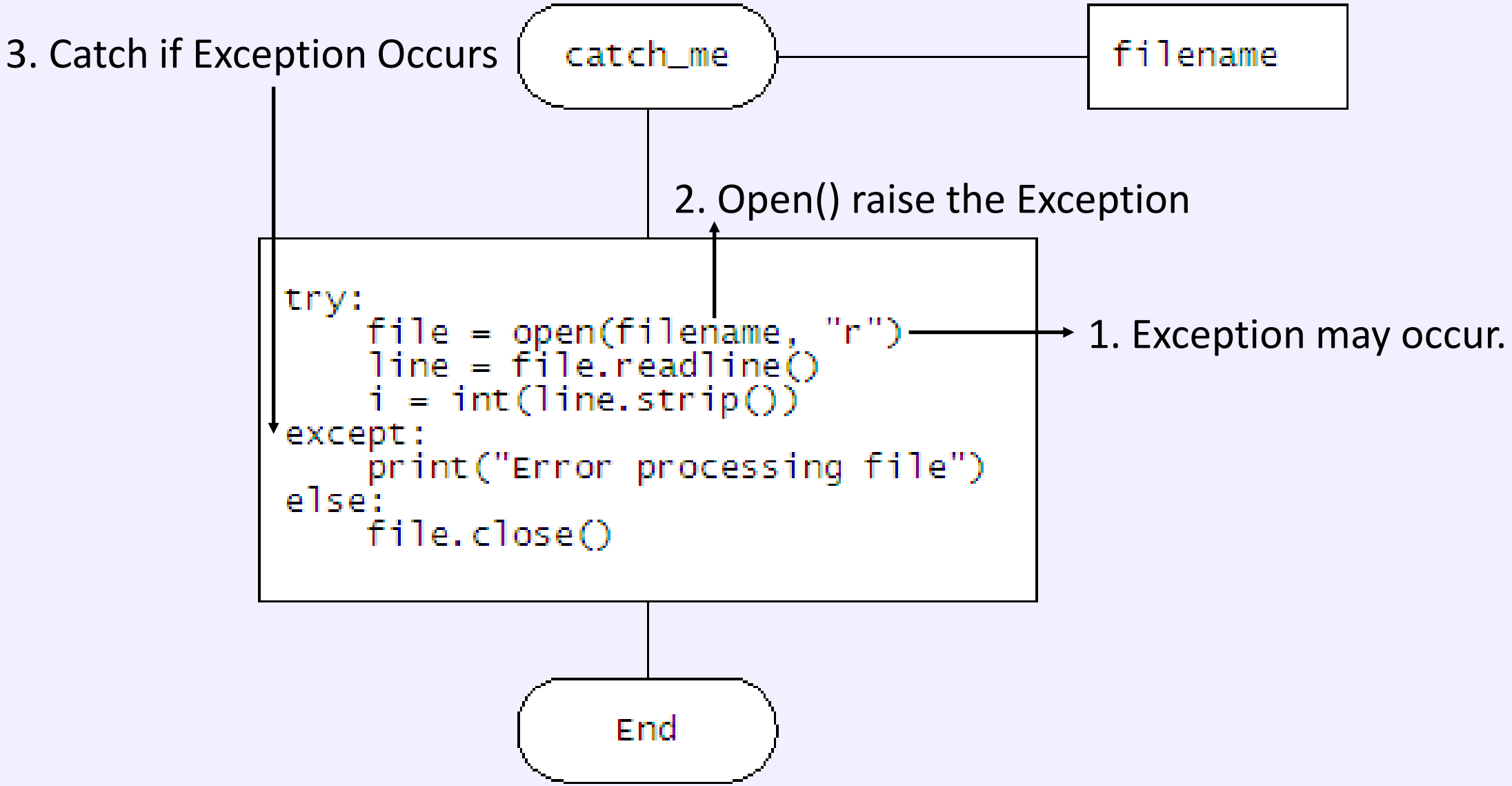


EXCEPTION HANDLING

Alternate way to continue flow of program



Exception handling in Python


3. Catch if Exception Occurs
catch_me
filename
2. Open() raise the Exception
try:
file = open(filename, "r")
line = file.readline()
i = int(line.strip())
except:
print("Error processing file")
else:
file.close()
1. Exception may occur.
End

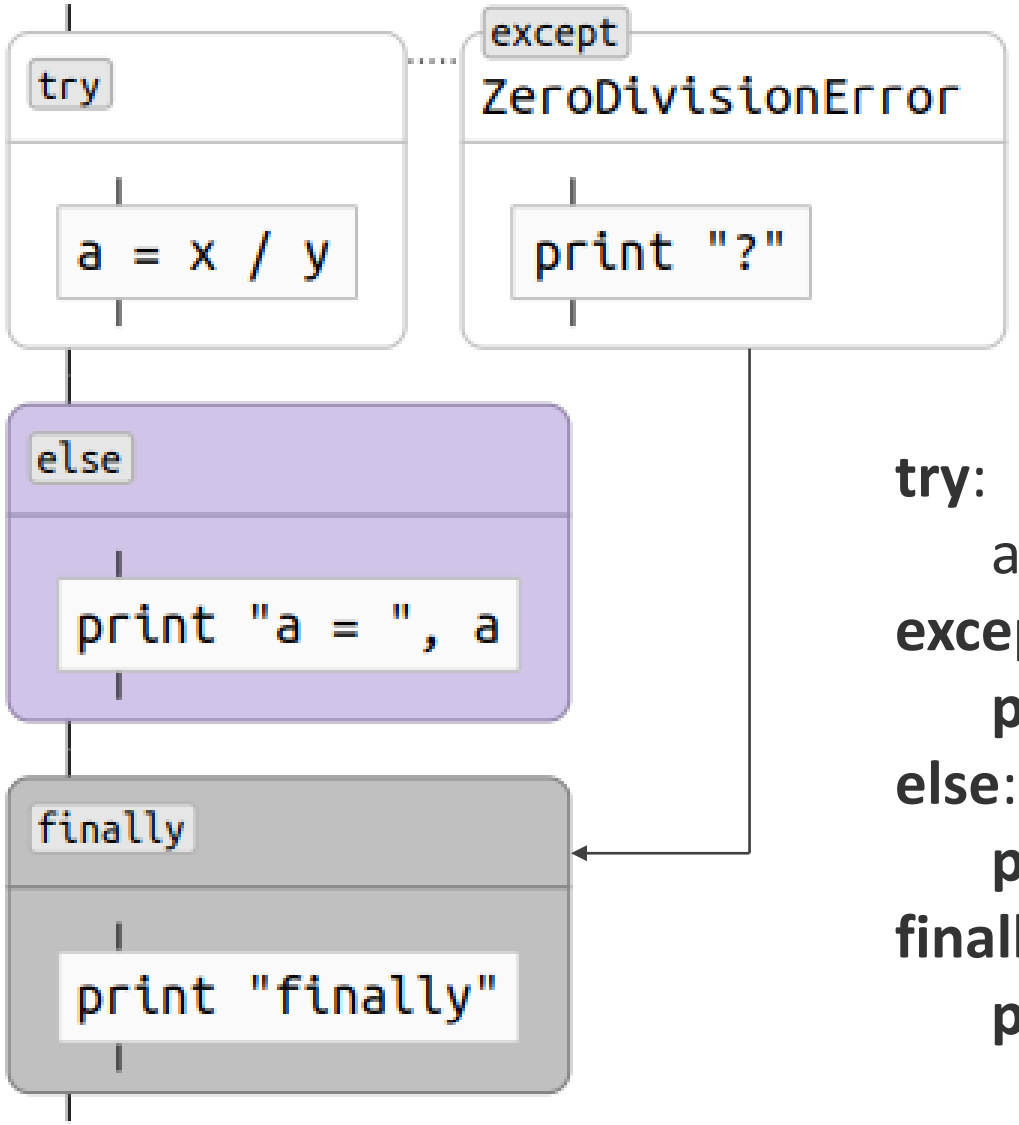
Understanding the difference between success and failure

In Python, when a **function** is called, it is being asked to do a job. Broadly speaking, just like in the case of sending your friend for coffee, there are two possible outcomes, even assuming the function has no bugs:

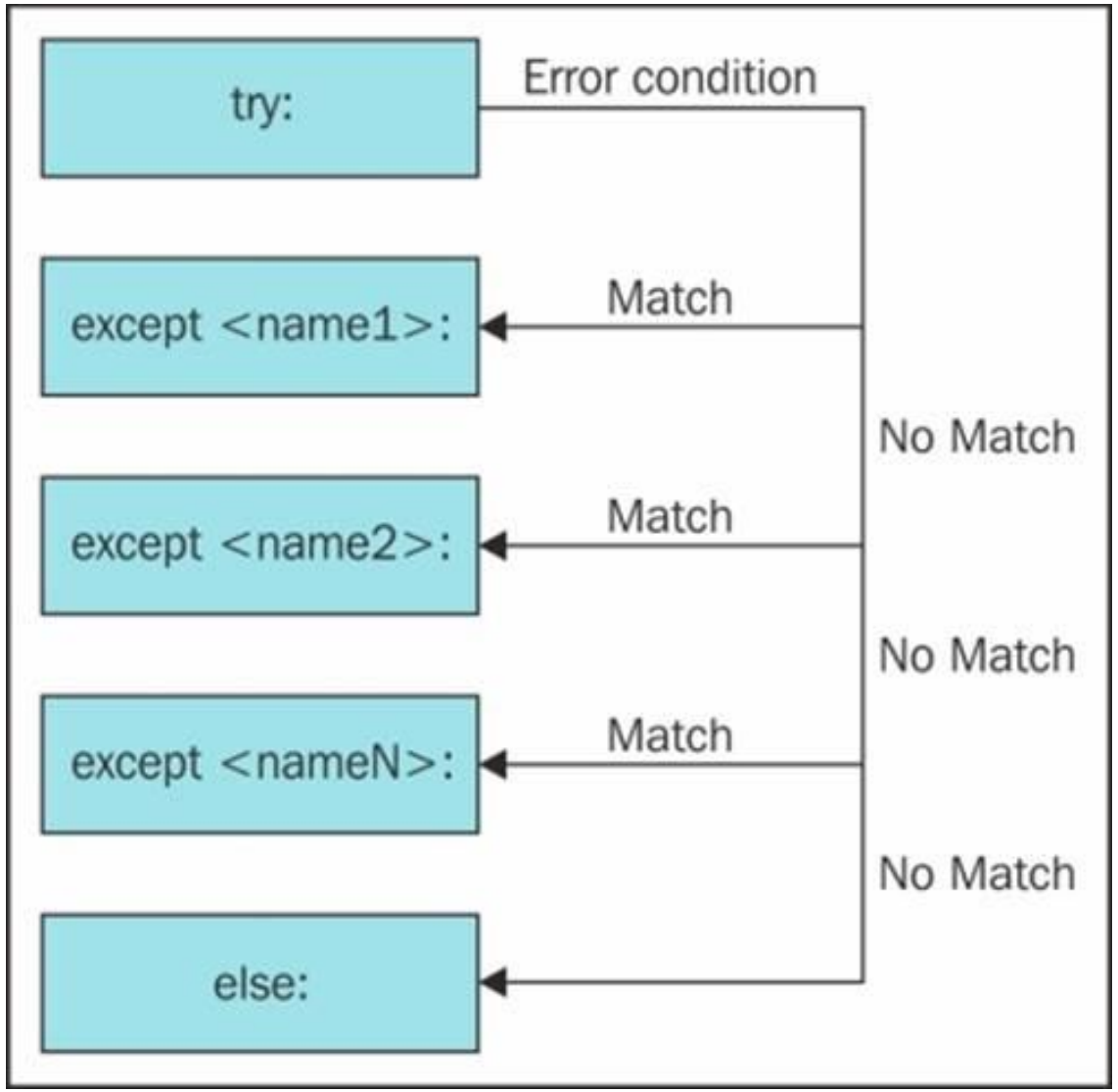
- The function will **complete** its job successfully and return an object of a type you expect.
- The function will **fail** to complete its job. Functions fail differently than they succeed in Python; rather than just returning an object that indicates failure, they don't return an object at all, but instead **raise an exception**.

Python:

- Try
- Except
- Else
- Finally




```
try:
    a = x / y
except ZeroDivisionError:
    print("?")
else:
    print("a = ", a)
finally:
    print("finally“)
```


try:
Error condition
except <name1>:
Match
No Match
except <name2>:
Match
No Match
except <nameN>:
Match
No Match
else:

Catching an Exception

try:

statements that will be attempted once

if any exception is raised, control leaves the "try" clause immediately

except:

statements that will execute after any statement in the "try" clause raises an exception

else:

statements that will execute after leaving the "try", but only if no exception was raised

finally:

statements that will always execute after leaving the "try", whether an exception was raised or not

note that these statements will happen after any in the "except" or "else" that also need to execute

Catching an Exception

There are a few combinations of these clauses that are legal; other combinations are illegal because they are nonsensical. (Think about why.) In both cases, the clauses must be listed in the order below:

- A **try** and a **finally** and nothing else
- A **try**, at least one **except**, (optionally) an **else**, and (optionally) a **finally**

Demo Program:

firstexception.py

Go PyCharm!!!

```
x = int(input("Enter the nominator:  "))
y = int(input("Enter the denominator:"))

try:
    a = x / y
except ZeroDivisionError:
    print("?")
else:
    print("a = ", a)
finally:
    print("finally")
```

```
Enter the nominator:  4
Enter the denominator:0
?
finally

Enter the nominator:  3
Enter the denominator:3
a =  1.0
finally
```

Exception Handling

LECTURE 2

Exception Handling

- An exception is an **error** that happens during the execution of a program.
- Exceptions are known to non-programmers as instances that do not conform to a general rule.
- The name "exception" in computer science has this meaning as well: It implies that the problem (the exception) doesn't occur frequently, i.e. the exception is the "exception to the rule".
- Exception handling is a construct in some programming languages to handle or deal with errors automatically.

Exception Handling

Catch-block (except) can be considered as a exception object receiving function.

- Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the **exception handler (a function receiving exception object)**.
- Depending on the kind of error ("division by zero", "file open error" and so on) which had occurred, the error handler can "fix" the problem and the program can be continued afterwards with the previously saved data.

Exceptions

Exceptions are events that can modify the flow or control through a program.

They are automatically triggered on errors.

try/except : catch and recover from raised by you or Python exceptions

try/finally: perform cleanup actions whether exceptions occur or not

raise: trigger an exception manually in your code

assert: conditionally trigger an exception in your code

Exception Roles

Error handling

- Wherever Python detects an error it raises exceptions
- Default behavior: stops program.
- Otherwise, code try to catch and recover from the exception (try handler)

Event notification

- Can signal a valid condition (for example, in search)

Special-case handling

- Handles unusual situations

Termination actions

- Guarantees the required closing-time operators (try/finally)

Unusual control-flows

- A sort of high-level “goto”

Being careful about what kinds of exceptions you catch

- Exceptions are Python objects; like all objects, they have a type.
- An exception's type classifies what kind of failure occurred.
- When you see a traceback, the exception's type is included in what's displayed, which helps you to understand what went wrong. For example, the last line of the traceback in the example above said this:

  TypeError: object of type 'int' has no len()

- In this case, the type of exception that was raised was one called TypeError, a type built into Python that represents a problem revolving around incompatibility of types with the operations you're trying to perform on them
- There are other types of exceptions that are built into Python that you might have seen before, as well, such as **ValueError**, **NameError**, and **IndexError**; these represent other ways that functions can fail.

Catching only one type of exception

An except clause can specify a type of exception by simply listing its name after the word except.

```
def read_number_and_print_square() -> None:
    try:
        number = int(input())
        print('The square of the number you entered is {}'.format(number * number))
    except ValueError:
        print('That is not a number')
```

In this example, we're catching only the **one** type of exception that we reasonably expect might go wrong. The call to int() may fail if the user's input is something that can't be converted into an integer; if that's the case, it will raise a **ValueError**. So, here, we've handled just the **ValueError.** If we had misspelled the name of a variable or made any other minor mistake in writing this function, it would have manifested itself in a different kind of exception, one that this function does not know how to handle.

Example

Raise Built-in Exception Type

```
def division(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print("division by zero")

division(3, 0)
print(division(5, 2))
division(5, 0)
```

exception01.py

Output:

```
division by zero
2.5
division by zero
```

Raise String Exception Type

#Note: except: is the way to catch string exception in python 3.x

```
def div(x, y):
    try:
        if y==0: raise "zero"
        return x/y
    except: print("ZERO")

print(div(3, 4))
div(3, 0)
```

exception02.py

Output:

```
C:\Python\Python36\python.exe
0.75
ZERO
```

try/except/else

```
try:
    <block of statements>              #main code to run
except <name1>:                        #handler for exception
    <block of statements>
except <name2>,<data>:                 #handler for exception with outgoing parameter from raise function
    <block of statements>
except (<name3>,<name4>):              #handler for exception
    <block of statements>
except:                                #handler for all exception (strong exception)
     <block of statements>
else:                                  # optional, runs if no exception occurs
    <block of statements>
```

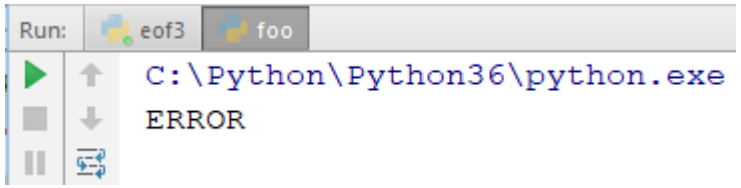
Universal except

except clauses with no type listed can handle **any** kind of exception, though these are somewhat dangerous in practice, because they'll handle every kind of problem the same way, meaning even a program bug will be handled the same way as the kinds of problems you expect.

```
def foo():
    return 14

def bar():
    b1 = 3 * foo()
    return bi

def example():
    try:
        answer = bar()
        print('The answer is {}'.format(answer))
    except:
        print('ERROR')
```

misspelled

foo.py



Example

```
>>>try:
      action()
    except NameError(): …
    except IndexError():  …
    except KeyError(): …
    except (AttributeError,TypeError,SyntaxError):…
    else: ….
```

- General catch-all clause: add empty except.
- It may catch unrelated to your code system exceptions.
- It may catch exceptions meant for other handler (system exit)

try/else

else is used to verify if no exception occurred in **try**.

- You can always eliminate *else* by moving its logic at the end of the *try* block.
- However, if “else statement” triggers exceptions, it would be misclassified as exception in try block.

try/finally

- In **try/finally**, *finally* block is always run whether an exception occurs or not

try:

    <block of statements>

finally:

    <block of statements>

- Ensure some actions to be done in any case
- It can not be used in the *try* with *except* and *else*.

Examples (finally): final message

exception03.py

```
try:
    print(3/0)
finally: print("finish")
```

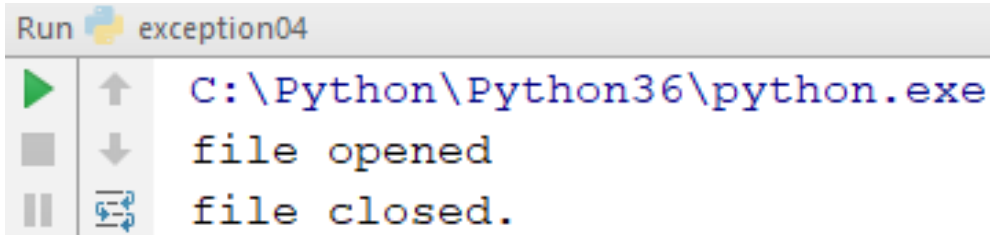
```
finish
Traceback (most recent call last):
  File "C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/PyDev/U2 IO System/Chapter1/Exception/exception03.py", line 2, in <module>
    print(3/0)
ZeroDivisionError: division by zero
```

Uncatched, unhandled exception.

Examples (finally): close file

exception04.py

```
try:
    f= open("data.txt", "r")
    print("file opened")
finally:
    f.close()
    print("file closed.")
```



raise

raise triggers exceptions explicitly (throw in Java)

raise <name>
raise <name>,<data> # provide data to handler
raise #re-raise last exception

```
>>>try:
      raise 'zero', (3,0)
   except 'zero': print("zero argument")
   except 'zero', data: print(data)
```

- Last form may be useful if you want to propagate caught exception to another handler.
- Exception name: **built-in name**, **string**, **user-defined class**

Example (Rethrow of Exception - Propagation)

exception05.py

```
try:
    raise KeyboardInterrupt
finally:
    print("Propagate")
    raise
```

```
C:\Python\Python36\python.exe "C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/PyDev/U2 IO System/Chapter1/Exception/exception05.py"
Propagate
Traceback (most recent call last):
  File "C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/PyDev/U2 IO System/Chapter1/Exception/exception05.py", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

assert

assert is a **conditional** *raise*

assert <test>, <data>

assert <test>

exception06.py

```
def f(x,y):
    assert x > 0, "x must be positve"
    assert y < 0, "y must be negative"
    return y**x

print(f(4, -3))
f(-3, -4)
```

```
81
Traceback (most recent call last):
  File "C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/PyDev/U2 IO System/Chapter1/Exception/exception06.py", line 7, in <module>
    f(-3, -4)
  File "C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/PyDev/U2 IO System/Chapter1/Exception/exception06.py", line 2, in f
    assert x > 0, "x must be positve"
AssertionError: x must be positve
```

If <test> evaluates to false, Python raises **AssertionError** with the <data> as the exception's extra data.

Exception Objects

String-based exceptions are any string object

```
myException = "String"
try:
    raise(myException)
except:
    print("caught")
```

exception07.py

```
Run exception07
C:\Python\Python36\python.exe
caught
```

Note:
raise(myException) # need to use parentheses for python 3
to catch string exception, don't use "expect myException"

Class-based exceptions are identified with classes. They also identify categories of exceptions.

- String exceptions are matched by object identity: *is*
- Class exceptions are matched by superclass identity: *except* catches instances of the mentioned class and instances of all its subclasses lower in the class tree.

Class Exception Example

Any Exception class must Inherit **BaseException**

```
class General(BaseException):
    pass
class Specific1(General):
    pass
class Specific2(General):
    pass
def raiser0():
    X = General()
    raise(X)
def raiser1():
    raise(Specific1)
def raiser2():
    X = Specific2()
    raise(X)
for func in (raiser0, raiser1, raiser2):
    try: func()
    except General:
        import sys
        print('caught:', sys.exc_info())
    except:
        print("All other types")
```

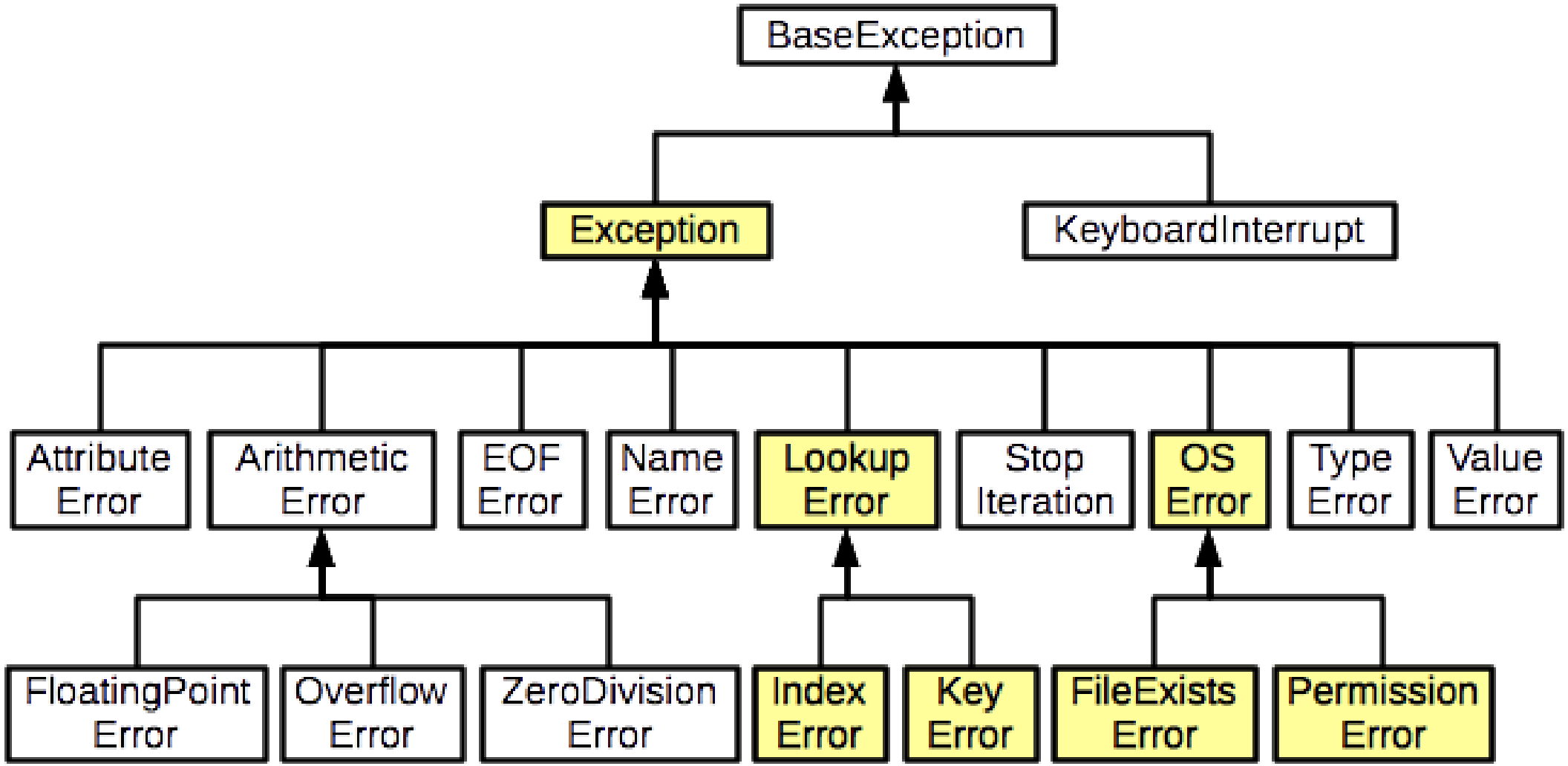
exception08.py

```
caught: (<class '__main__.General'>, General(), <traceback object at 0x000001452F6785C8>)
caught: (<class '__main__.Specific1'>, Specific1(), <traceback object at 0x000001452F678648>)
caught: (<class '__main__.Specific2'>, Specific2(), <traceback object at 0x000001452F678608>)
```

exception08.py

Exception Classes

LECTURE 3


BaseException
Exception
KeyboardInterrupt
Attribute Error
Arithmetic Error
EOF Error
Name Error
Lookup Error
Stop Iteration
OS Error
Type Error
Value Error
FloatingPoint Error
Overflow Error
ZeroDivision Error
Index Error
Key Error
FileExists Error
Permission Error

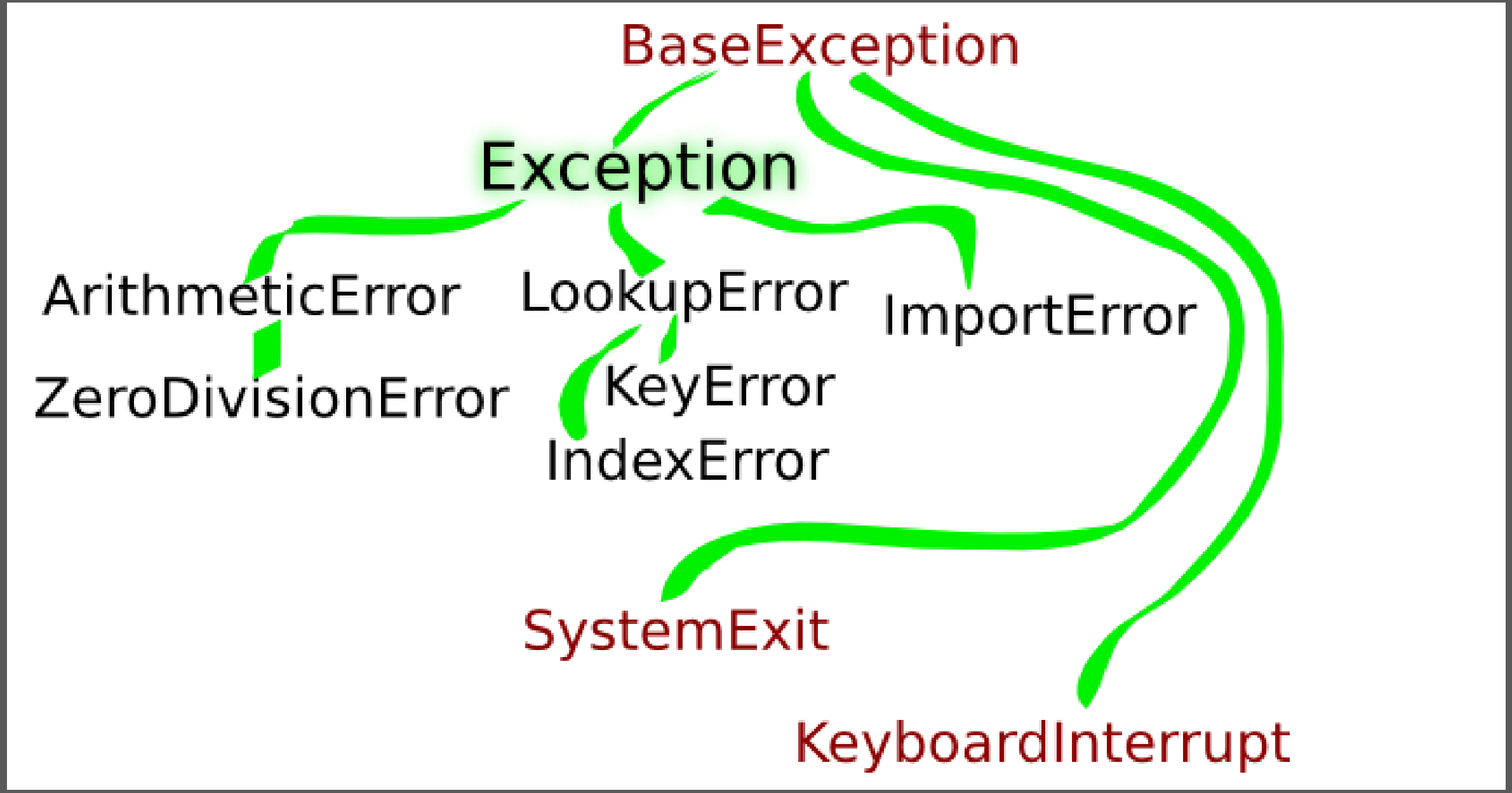
Some exceptions

- The full list of exceptions is regrettably big.
- The exceptions also have a hierarchy. Some are special cases of general error, e.g. IndexError and KeyError are special cases of LookupError.

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

Some of the more important are:

IndexError	sequence subscript out of range
KeyError	dictionary key not found
ZeroDivisionError	division by 0
ValueError	value is inappropiate for the built-in function
TypeError	function or operation using the wrong type
IOError	input/output error, file handling
EOFError	end-of-file error


BaseException
Exception
ArithmeticError
LookupError
ImportError
ZeroDivisionError
KeyError
IndexError
SystemExit
KeyboardInterrupt

Error Type ValueError:

Demo Program: valueerror.py

valueerror.py

```
while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
        print("No valid integer! Please try again ...")
print("Great, you successfully entered an integer!")
```

Output:

```
Please enter an integer: IOMega
No valid integer! Please try again ...
Please enter an integer: 3
Great, you successfully entered an integer!
```

Error ZeroDivisionError:

Demo Program: zerodivision.py

zerodivision.py

```
print("Three")
value = 10 / 2
print("Two")
value = 10 / 1
print("One")
d = 0

try:
    # This division has problem, divided by 0.
    # An error has occurred here (ZeroDivisionError).
    value = 10 / d
    print("value = ", value)

except ZeroDivisionError as e:
    print("Error: ", str(e))
    print("Ignore to continue ...")

print("Let's go!")
```

Output:

```
Three
Two
One
Error:  division by zero
Ignore to continue ...
Let's go!
```

Multiple Exception Classes

- A try statement may have more than one except clause for different exceptions. But at most one except clause will be executed.
- Our next example shows a try clause, in which we open a file for reading, read a line from this file and convert this line into an integer. There are at least two possible exceptions:
  - **IOError**
  - **ValueError**

Error Type IOError and ValueError:

Demo Program: iovalue.py

iovalue.py

```
import sys

try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    errno, strerror = e.args
    print("I/O error({0}): {1}".format(errno,strerror))
    # e can be printed directly without using .args:
    # print(e)
except ValueError:
    print("No valid integer in line.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Output:

```
C:\Python\Python36\python.exe "C:/Eric_Chou
I/O error(2): No such file or directory
```

Note:

IOError: file not found

ValueError: data format mismatch

IOError as e:

e is the exception object which you can obtain exception related information.

strip() is the same function as .trim() in Java

Multiple Exception types are caught in this example.

Combined Multiple Exception Detection

An except clause may name more than one exception in a tuple of error names, as we see in the example below:

```
try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except (IOError, ValueError):
    print("An I/O error or a ValueError occurred")
except:
    print("An unexpected error occurred")
    raise
```

Output:

```
C:\Python\Python36\python.exe
An I/O error or a ValueError occurred
```

iovalue2.py

Exceptions Happen in Functions

Demo Program: functionException.py

We want to demonstrate now, what happens, if we call a function within a try block and if an exception occurs inside the function call:

functionException.py

```
def f():
    x = int("four")

try:
    f()
except ValueError as e:
    print("got it :-) ", e)

print("Let's get on")
```

Output:

```
got it :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

Exceptions Happen in Functions
Demo Program: functionException3.py

Finding the exception handling locally and globally.

functionException3.py

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)
        raise

try:
    f()
except ValueError as e:
    print("got it :-) ", e)

print("Let's get on")
```

Exception handled by local handler in f()

And, also by global handler outside of f()

Output:

```
got it in the function :-)  invalid literal for int() with base 10: 'four'
got it :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

Exceptions Happen in Functions
Demo Program: functionException4.py

Finding the exception handling only globally (by propagation).

functionException4.py

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        raise
try:
    f()
except ValueError as e:
    print("got it :-) ", e)
print("Let's get on")
```

Output:

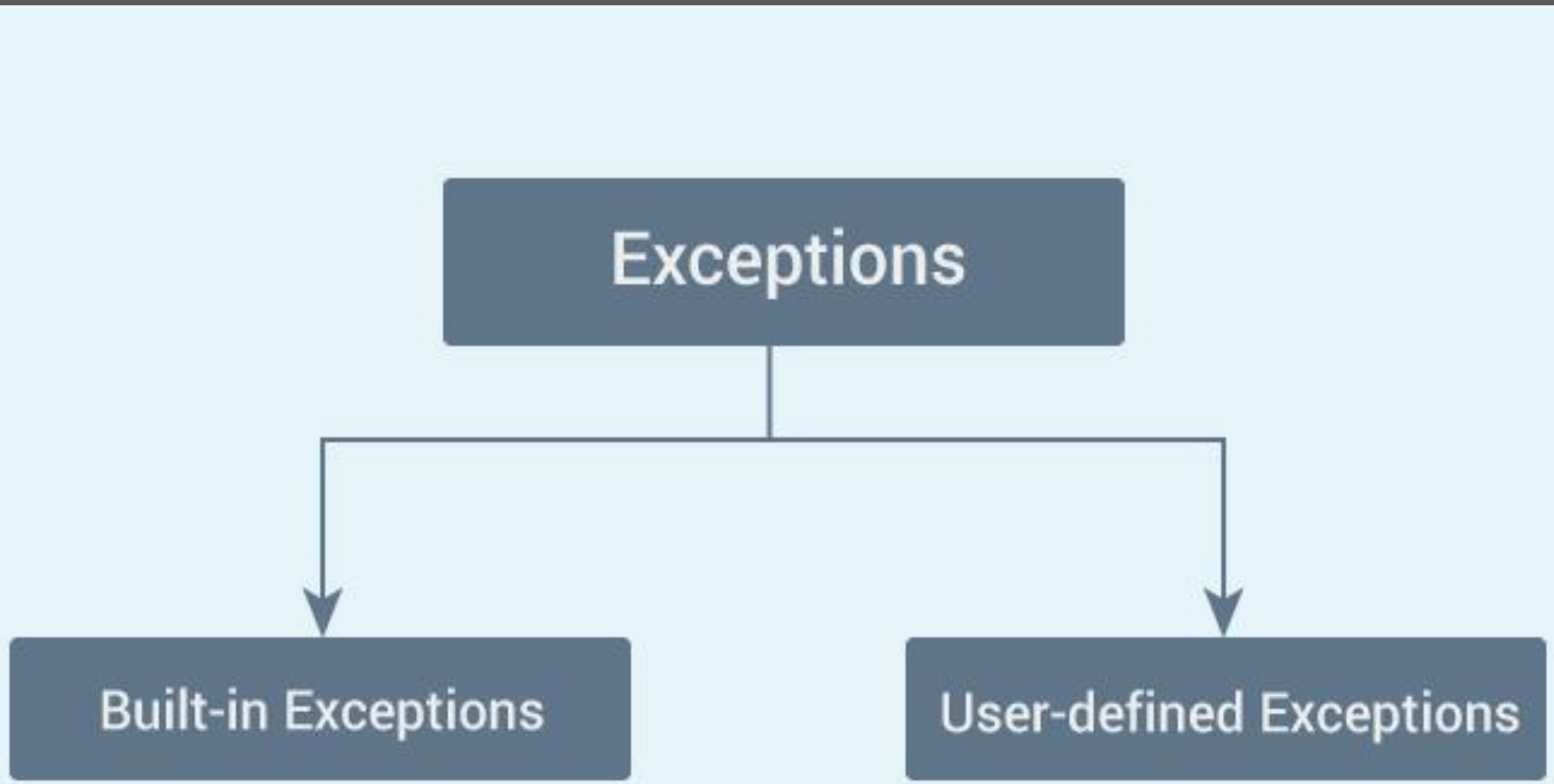
```
got it :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

About Exception Handling

1. Special exception handler for each exception type can be used.
2. Exception can be re-raised and handled by upper functional level.
3. Exception handler can be at local level, global level, multiple level or no handler.
4. If no exception handler, Python default exceptional description will show up.

Custom-Defined Exceptions

LECTURE 4


Exceptions
Built-in Exceptions
User-defined Exceptions

Built-in Exception Classes

Exception – top-level root superclass of exceptions.

StandardError – the superclass of all built-in error exceptions.

ArithmeticError – the superclass of all numeric errors.

OverflowError – a subclass that identifies a specific numeric error.

```
>>>import exceptions
>>>help(exceptions)
```

Custom-made Exceptions

Demo Program: custom1.py

It's possible to create Exceptions yourself: (Create one **SyntaxError** exceptional object)

```
raise SyntaxError("Sorry, my fault!")
```

```
C:\Python\Python36\python.exe "C:/Eric_Chou/Python Course/
Traceback (most recent call last):
  File "C:/Eric_Chou/Python Course/Python Object-Oriented
    raise SyntaxError("Sorry, my fault!")
SyntaxError: Sorry, my fault!
```

Custom-made Exceptional Object

Demo Program: custom2.py

Simple Exceptional Object:

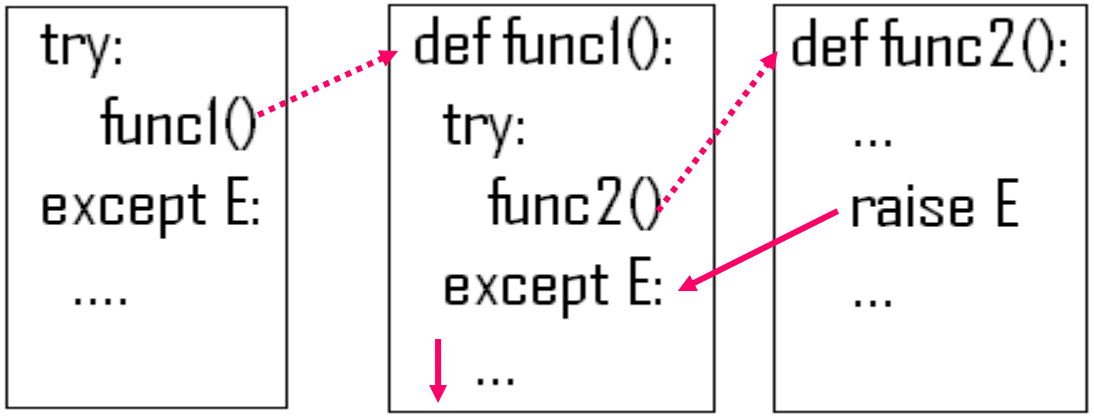
```
class MyException(Exception):
    pass

raise MyException("An exception doesn't always prove the rule!")
```

```
Traceback (most recent call last):
  File "C:/Eric_Chou/Python Course/Python Object-Oriented Programming
    raise MyException("An exception doesn't always prove the rule!")
__main__.MyException: An exception doesn't always prove the rule!
```

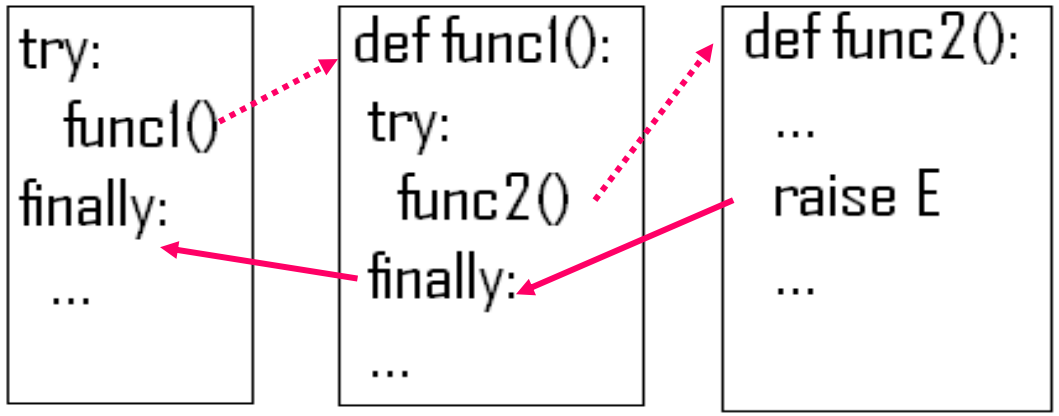
Advanced Topics I: finally, Re-throwing, and Chained Exception Handling

LECTURE 5

Nesting Exception Handlers



Once the exception is caught, it's life is over. (Unless it is raised again.)



- *finally* does not kill exception – just specify code to be run on the way out.

```
def func3():
    try:
        print("Tried Function 3")
        x = 9 / 0
    except:
        print("Handled in Except 3")
def func2():
    try:
        print("Tried Function 2")
        func3()
    except:
        print("Handled in Except 2")
def func1():
    try:
        print("Tried Function 1")
        func2()
    except:
        print("Handled in Except 1")
def main():
    try:
        print("Tried Function main()")
        func1()
    except:
        print("Handled in Except main()")
if __name__ == "__main__":
    main()
```

Output:

```
Tried Function main()
Tried Function 1
Tried Function 2
Tried Function 3
Handled in Except 3
```

nestedexcept.py

```
def func3():
    try:
        print("Tried Function 3")
        x = 9 / 0
    except:
        print("Forwarded in Except 3")
        raise
def func2():
    try:
        print("Tried Function 2")
        func3()
    except:
        print("Forwarded in Except 2")
        raise
def func1():
    try:
        print("Tried Function 1")
        func2()
    except:
        print("Handled in Except 1")
def main():
    try:
        print("Tried Function main()")
        func1()
    except:
        print("Handled in Except main()")
if __name__ == "__main__":
    main()
```

Output:

```
Tried Function main()
Tried Function 1
Tried Function 2
Tried Function 3
Forwarded in Except 3
Forwarded in Except 2
Handled in Except 1
```

nestedexcept2.py

```python
def func3():
    try:
        print("Tried Function 3")
        x = 9 / 0
    finally:
        print("In finally 3")
def func2():
    try:
        print("Tried Function 2")
        func3()
    finally:
        print("In finally 2")
def func1():
    try:
        print("Tried Function 1")
        func2()
    finally:
        print("In finally 1")
def main():
    try:
        print("Tried Function main()")
        func1()
    finally:
        print("In finally main()")
if __name__ == "__main__":
    main()
```

Output:

```
Traceback (most recent call last):
Tried Function main()
Tried Function 1
Tried Function 2
Tried Function 3
In finally 3
In finally 2
In finally 1
In finally main()
  File "C:/Eric_Chou/Python Course/Python
    main()
  File "C:/Eric_Chou/Python Course/Python
    func1()
  File "C:/Eric_Chou/Python Course/Python
    func2()
  File "C:/Eric_Chou/Python Course/Python
    func3()
  File "C:/Eric_Chou/Python Course/Python
    x = 9 / 0
ZeroDivisionError: division by zero
```

nestedfinally.py

Advanced Topics II: File Input Preprocessing

LECTURE 6

Exception Idioms

- All errors are exceptions, but not all exceptions are errors. It could be signals or warnings (warnings module)

eof.py

```
f = open("data.txt", "r")
i=0
while True:
    try:
        a = int(f.readline())
    except ValueError:
        break
    else:
        i += 1
        print("Line ", i, ": ", a)
f.close()
```

Output:

```
Line  1 :  1
Line  2 :  2
Line  3 :  3
Line  4 :  4
Line  5 :  5
Line  6 :  6
Line  7 :  7
Line  8 :  8
Line  9 :  9
Line  10 :  10
```

While-loop using ValueError to Detect EOF

eof.py

```
f = open("data.txt", "r")
i=0
while True:
    try:
        a = int(f.readline())
    except ValueError:
        break
    else:
        i += 1
        print("Line ", i, ": ", a)
f.close()
```

While-loop using readline() returned Boolean value to Detect EOF

eof2.py

```
f = open("data.txt", "r")
i=0
a = f.readline()
while a:
    i+=1
    print("Line ", i, ": ", int(a))
    a = f.readline()
f.close()
```

Using try-except and EOFError, for-loop and readlines()

eof3.py

```
f = open("data.txt", "r")
i=0
try:
    for a in f.readlines():
        i += 1
        print("Line ", i, ": ", int(a))
except EOFError:
    pass

f.close()
```

Output:

```
Line  1 :  1
Line  2 :  2
Line  3 :  3
Line  4 :  4
Line  5 :  5
Line  6 :  6
Line  7 :  7
Line  8 :  8
Line  9 :  9
Line  10 :  10
```

Notes:

- **readlines()** function is used to read all of the lines in the file f into a list.
- **a** is an item picked from the list
- **EOFError** is an Exception type. It will happen.

Input Format Checking by Try-Except Block

doWhileInput.py

```
done = False
while not done:
    try:
        num = int(input("Enter an integer (or exit): "))
        print("%d is a valid integer." % num)
    except ValueError:
        done = True
```

Output:

```
Enter an integer (or exit): 3
3 is a valid integer.
Enter an integer (or exit): 4
4 is a valid integer.
Enter an integer (or exit): 5
5 is a valid integer.
Enter an integer (or exit): exit
```

Exception Design Tips

- Operations that commonly fail are generally wrapped in *try* statements(file opens, socket calls).
- However, you may want failures of such operations to kill your program instead of being caught and ignored if the failure is a show-stopper. Failure = useful error message.
- Implement termination in try/finally to guarantee its execution.
- It is sometimes convenient to wrap the call to a large function in a single *try* statement rather than putting many try statements inside of the function.

Chapter Program

LECTURE 7

Demo Program: line_count.py

Try **usdeclar.txt** for this example.

This program is similar to our eof series program. Just take a look.

Go PyCharm!!!

Click on this red button if you want to terminate the execution.