Exceptions in Python

Pai H. Chou
National Tsing Hua University

Outline

- Types of errors
  - Syntax errors, type errors, I/O errors
- Handling exceptions
  - **`try`**-**`except`**
  - **`else`**, **`finally`**
- Raising exceptions
  - **`raise`** an exception
  - **`assert`**

Errors in Python

- Syntax error
  - grammatically correct, illegal identifiers, unmatched quotes or parentheses, missing semicolon, illegal operators...
- Logical errors
  - divide by 0, using undefined variables, index out of bound, ...
  - modifying an immutable object, dictionary key not found...
- System and package error
  - File not found, permission denied, terminating a program...
- Many other kinds of errors...

Exceptions

- Exception = mechanism for program to report error
  - callee <u>raises</u> (or "throws") an exception
  - caller <u>handles</u> (or "catches") an exception
- Handling exception => program can continue
  - handler can set program to known state
- if exception is not handled,
  - exception propagates to its caller to handle
  - if no caller handles it => program crashes

Example form of exception: traceback in interactive mode

- happens in interactive mode

```
>>> z   # assume the name z has not been defined
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

  - the Python shell is trying to lookup the name z but could not find it => hence NameError

- why exception?
  - give user a chance to fix problem and keep running
  - if not handled, then the program "crashes"

Example exception related to numbers

- ZeroDivisionError

```
>>> z = 10 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

- OverflowError (floating point)

```
>>> 10000000000.0**10000000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Result too large')
```

TypeError

- when given wrong wrong type of data
- e.g., indexing expects int but gets non-int

```
>>> L = [1, 2, 3]
>>> L['xyz']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not str
```

- e.g., attempt to add int and another type

```
>>> 1 + 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

ValueError

- type may be correct but value is invalid
- Example: conversion from str to int

```
>>> int('23')
23
>>> int('xyz')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'xyz'
```

  - e.g., when prompting user to input an int, they type in some non-int string value

Common Exceptions

error	**meaning**
`NameError`	trying to access an undefined variable or name
`ZeroDivisionError`	trying to divide by 0
`SyntaxError`	grammatically incorrect code
`IndexError`	`L[i]` when `i` < -*N* or `i` >= *N* where N = `len(L)`
`KeyError`	`dict[`*k*`]` when dict doesn't contain key *k*
`OSError` (was `IOError`)	system access problems, including files (not found, no permission)
`AttributeError`	`obj.attr` when `obj` does not have attribute `attr`

try-except for handling exceptions

- Catch-all

```
try:
   statementSuite
except:
   excHandler
```

- Catch specific

```
try:
   statementSuite
except E1:
   excHandler1
except E2:
   excHandler2
```

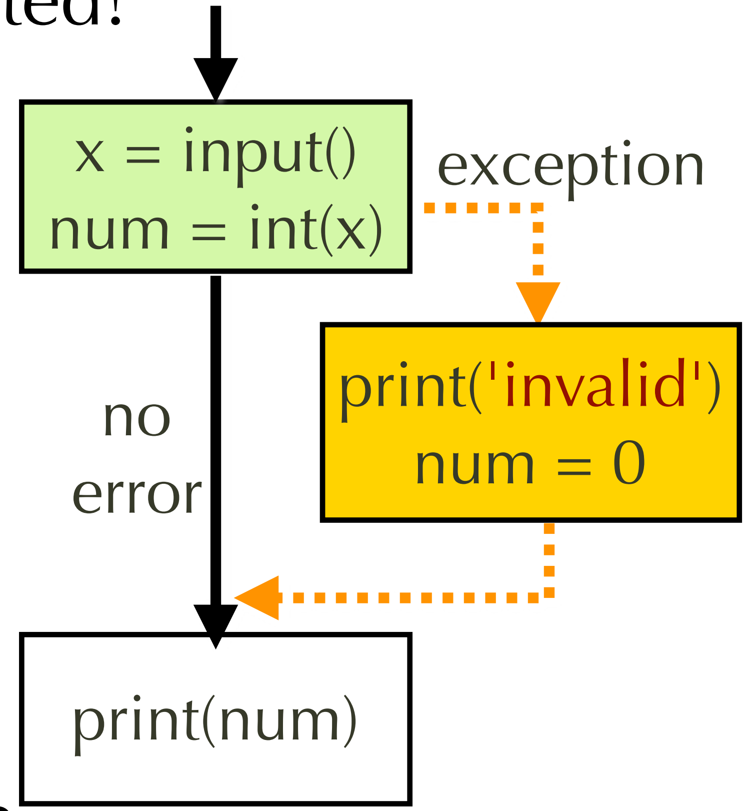
```
try:
    fh = open('myfile')
except:    # catches all errors
    print('cannot open file')
    sys.exit(1)
s = fh.read()
fh.close()
```

```
try:
    fh = open('myfile')
except OSError:
    print('cannot open file')
    sys.exit(1)
s = fh.read()
fh.close()
```

Control flow of exception

- try-except suite
  - if no error, suite runs normally and skips except part
  - if error, jumps from error spot to except suite
    => try-except suite may be incompletely executed!

```python
try:
    x = input('enter a number:')
    num = int(x)
except:
    print('invalid number')
    num = 0
print('number = ', num)
```

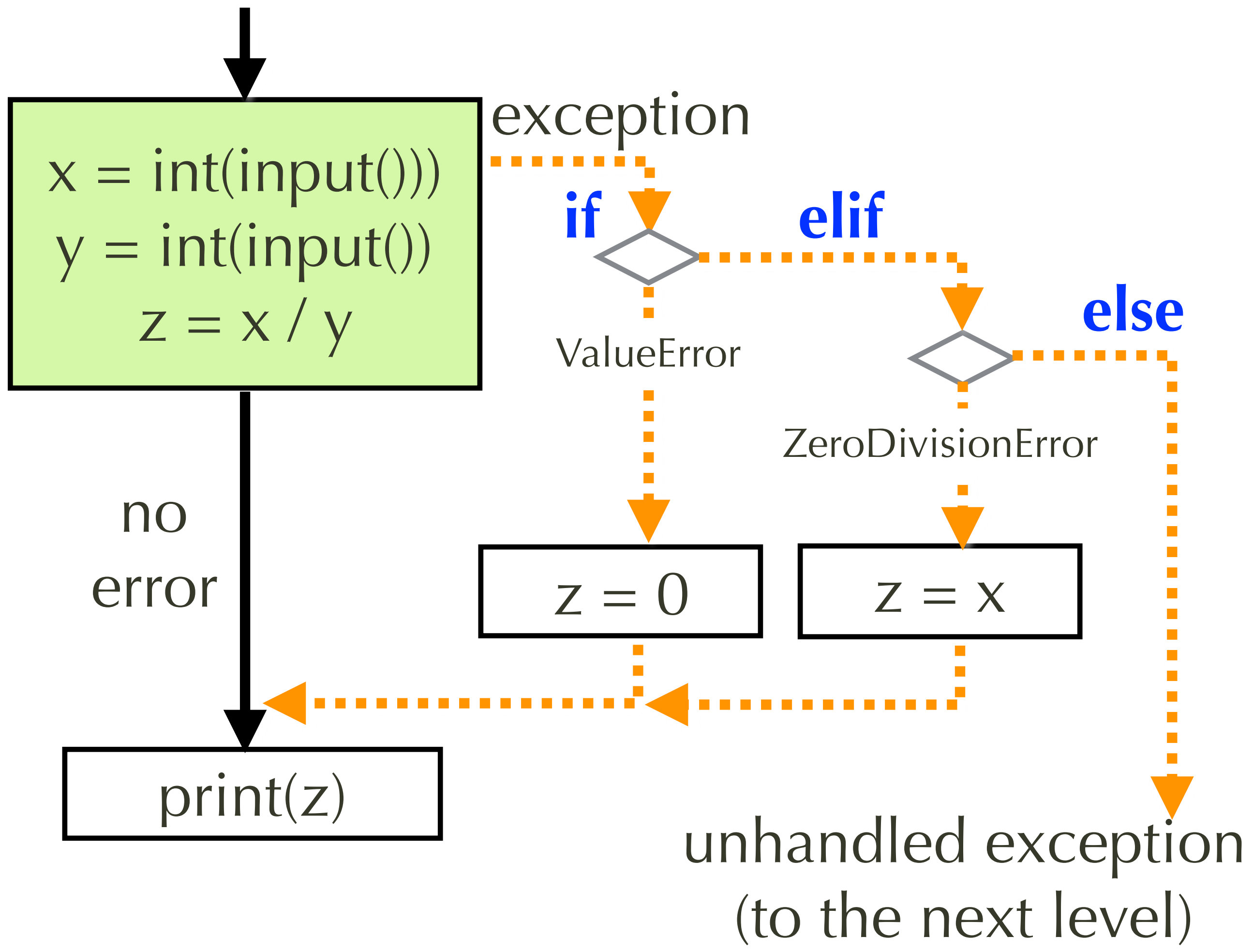


- except suite may fix problem and allow the program to continue execution

Distinguish between different types of exceptions

- Several kinds of things can go wrong
  - multiple except clauses, each for its own type
  - at most one clause will be executed for an error

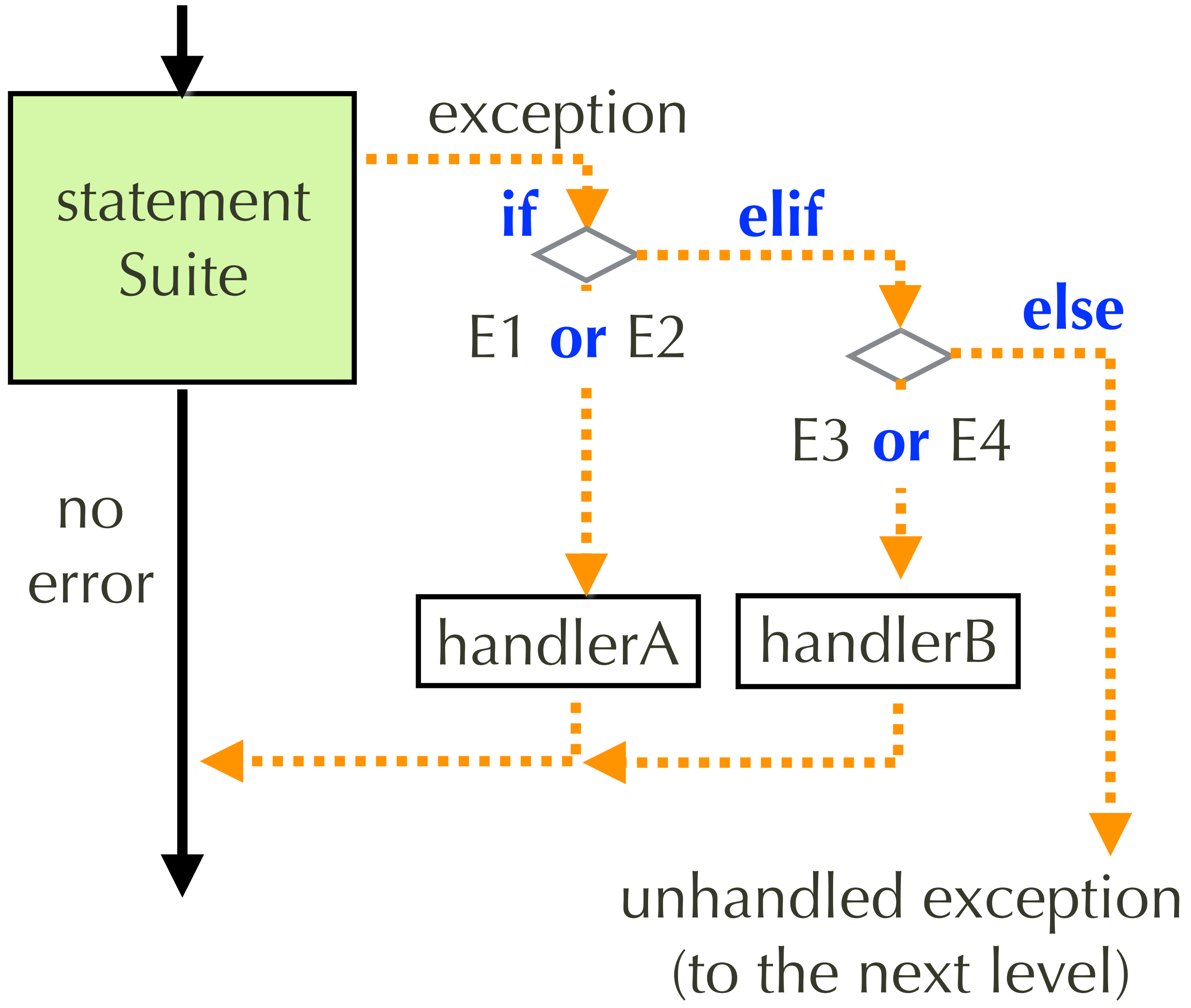
```
try:
    x = int(input('enter num1:'))
    y = int(input('enter num2:'))
    z = x / y
except ValueError:
    z = 0
except ZeroDivisionError:
    z = x
print(z)
```



Sharing handlers between multiple types of exceptions

- Catch specific

```
try:
    statementSuite
except (E1, E2):
    handlerA
except (E3, E4):
    handlerB
```



Multiple exceptions in a try clause

- For example, if you want `OverflowError` and `ZeroDivisionError` to be handled by the same code, do

```
try:
    quotient = M / X
except (OverflowError, ZeroDivisionError):
    quotient = 0.0
...
```

- instead of listing them multiple times

```
try:
    quotient = M / X
except OverflowError:
    quotient = 0.0
except ZeroDivisionError:
    quotient = 0.0
```

Exceptions with Arguments

-
  ```
  try:
      statementSuite
  except E1 as e:
      excHandler1
  ```

- *E1* is the exception type
- e is the exception argument, which carries associated information
  - e.g., error message

Exception arguments

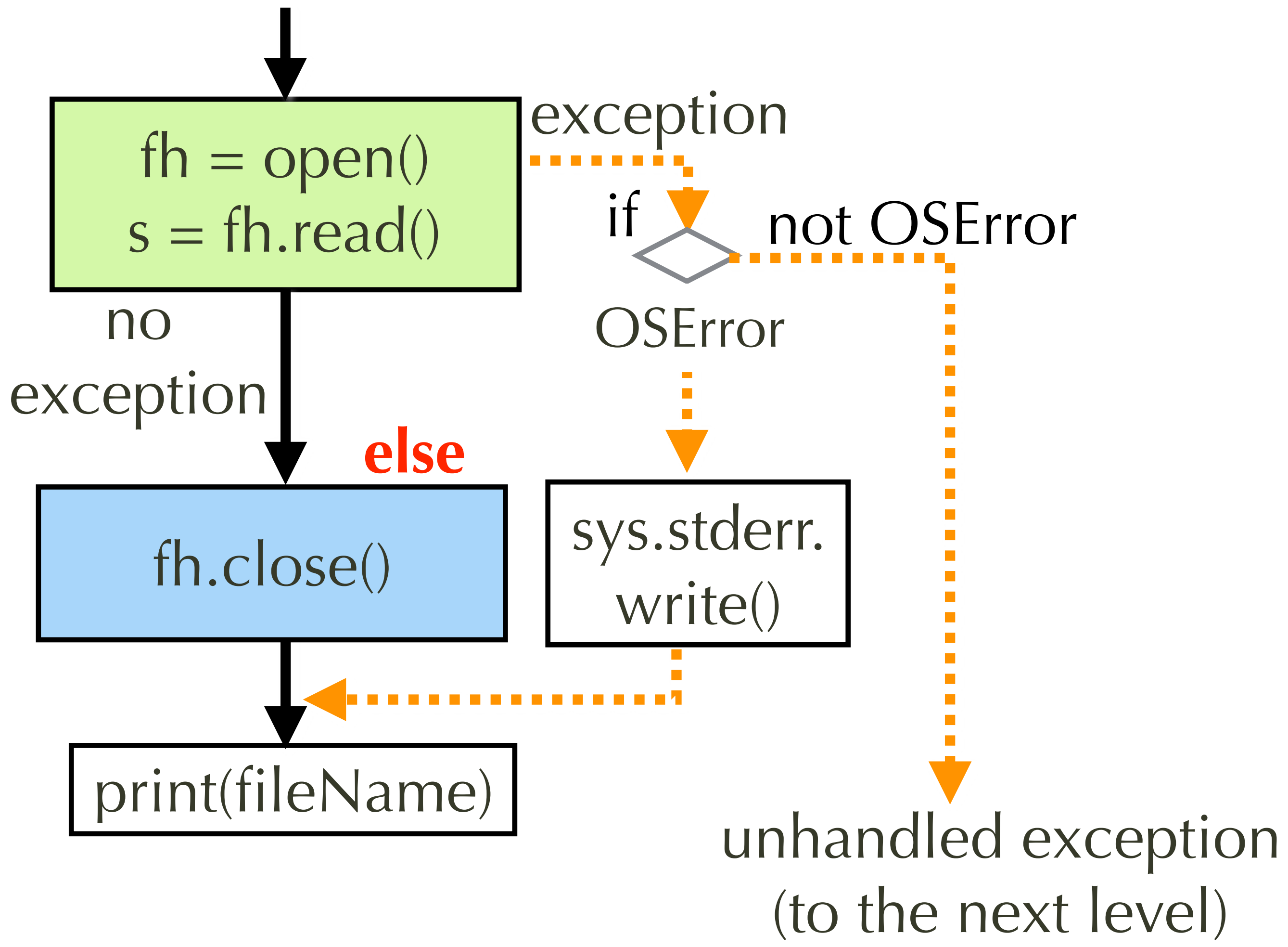
- Additional info associated with exception
- Example: OSError could be several things
  - reading: file not found
  - writing: permission denied
  - both are OSError, but how to distinguish them?
- The argument contains more information

```
>>> try:
...     fh = open('myfile')
... except OSError as err:
...     print('error: %s' % str(err))
...
error: [Errno 2] No such file or directory: 'myfile'
```

else clause in try-except

- executed on "normal" path (i.e., no error)
  - analogous to **else** in **while** or **for** loops

```python
import sys
fileName = 'myfile'
try:
    fh = open(fileName)
    s = fh.read()
except OSError as err:
    sys.stderr.write(str(err))
else: # normal completion of try
    fh.close()
print(fileName)
```



nested try-except statements

- a `try-except` statement inside another `try-except`:

outer try

```python
import sys
try:
    fh = open('myfile')
    try:
        A = int(fh.read())
        B = int(fh.read())
        quotient = A / B
    except (OverflowError, ZeroDivisionError):
        quotient = 0.0
    except ValueError:
        quotient = 1.0
    print('quotient = %f' % quotient)
except OSError as err:
    # catches uncaught inner exception
    sys.stderr.write(str(err))
```

inner try

how nested try works

- inner try catches => no problem
- uncaught inner error => outer can catch it

outer try

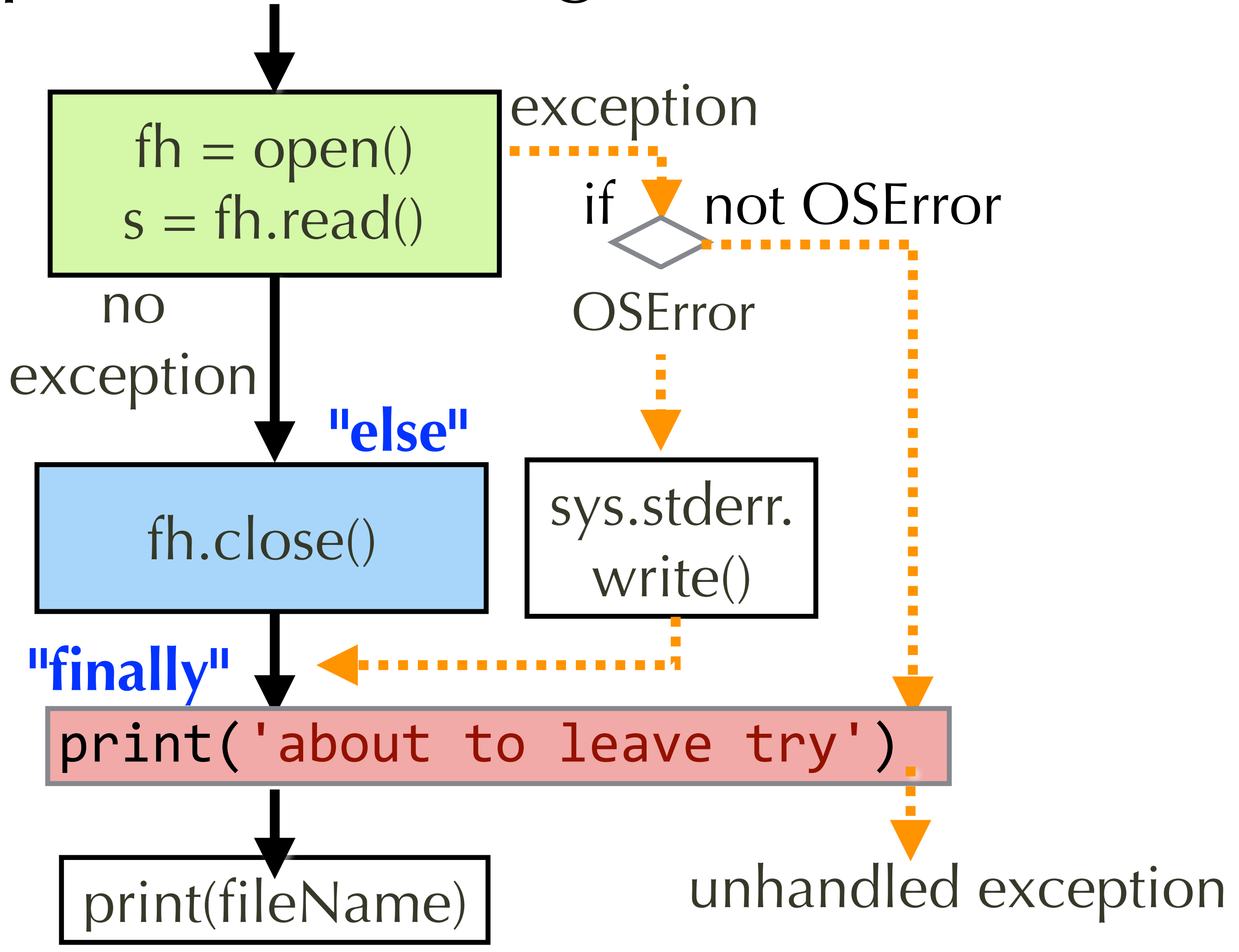
inner try

```
import sys
try:
    fh = open('myfile')
    try:
        A = int(fh.read())
        B = int(fh.read())
        quotient = A / B
    except (OverflowError, ZeroDivisionError):
        quotient = 0.0
    except ValueError:
        quotient = 1.0
    print('quotient = %f' % quotient)
except OSError as err:
    # catches uncaught inner exception
    sys.stderr.write(str(err))
```

finally clause in try-except

- executed on "all" paths
  - Why? try-suite may be incompletely executed
  - last chance to clean up before exiting

```
import sys
fileName = 'myfile'
try:
    fh = open(fileName)
    s = fh.read()
except OSError as err:
    sys.stderr.write(str(err))
else: # normal completion of try
    fh.close()
finally:
    print('about to leave try')
print(fileName)
```



try-finally

- finally clause executed regardless of exception or normal, including uncaught exception (to be propagated outward)

```
try:
    S1
except OSError:
    S2
finally:
    S3
```

not the same!

```
try:
    S1
except OSError:
    S2
S3
```

S3 is executed in 3 ways

- Normal completion of S1, then S3
- Incomplete S1 due to OSError, S2, then S3
- Incomplete S1 that is not OSError, execute S3, then go to higher level

S3 is executed in 2 ways

- Normal completion of S1, then S3
- Incomplete S1 due to OSError, S2, then S3

But if another exception that is not OSError, S1 not executed! Go to higher level

exception class hierarchy

- Exceptions can be categorized
- Example: ArithmeticError (the "superclass")
  - FloatingPointError (a specific kind of ArithmeticError)
  - ZeroDivisionError (another specific arith. err.)
  - OverflowError (another specific arith. err.)
- ArithmeticError (more general) covers the other more specific errors!

exception class hierarchy

- ArithmeticError
  - FloatingPointError
  - ZeroDivisionError
  - OverflowError

```
try:
    ....
except (FloatingPointError, ZeroDivisionError, OverflowError):
```

- is the same as

```
try:
    ....
except ArithmeticError:
```

Redundant clauses => tested in serial (if-elif-else) order!!

```
try:
    S1
except FloatingPointError:
    S2
except ArithmeticError:
    S3
```

- FloatingPointError => S2, **not** S3
- ZeroDivisionError or OverflowError => S3, not S2

```
try:
    S1
except ArithmeticError:
    S2
except FloatingPointError:
    S3
```

- All arithmeticError => handled by S2
- S3 is **never** executed, because it is already covered by ArithmeticError!!!!

Raising your own exceptions

- Why?
  - enforce parameter type, value, key into dictionary,
  - enforce syntax of command, application-specific condition
- Syntax
  - `raise` *ExceptionType*(*'arguments'*) # with arguments
  - `raise` *ExceptionType* # without arguments
  - `raise` # re-raise the exception for outer level to handle
  - `assert` *condition*
  - `assert` *condition*, *expression*

Example: rock-paper-scissors game

- Ask the user for rock, paper, scissors by typing 'r', 'p', or 's',
  - type 'q' to quit
  - Raise exception if the input is not one of them

```
rps = input('Rock, Paper, Scissors, or Quit? [rpsq] ')
if rps == 'q':
    sys.exit(0)
if rps in {'r', 'p', 's'}:
    # play the game...
else:
    raise ValueError(f'invalid input: {rps}')
```

Assertions

- A special case of raising exceptions
  - Syntax 1: **`assert`** *condition*

    meaning: **if not** *condition*: **raise** AssertionError
  - Syntax 2: **`assert`** *condition*, *expression*

    if not *condition*: **raise** AssertionError(*expression*)

```python
try:
    fh = open('myfile')
    s = fh.read()
    assert s != ''
except AssertionError:
    print('file empty!')
```

```python
try:
    fh = open('myfile')
    s = fh.read()
    assert s != '', 'empty file'
except AssertionError as e:
    print(str(e))
```

Use of Assertions

- Good practice to include assertions in your own code
  - making sure important assumptions hold before you execute some code
- You don't have to try-catch AssertionError
  - The main purpose is to help locate your bug!
  - Rely on runtime system to report assertion error