Exceptions and File Operations

Exceptions

- Unusual conditions that cause a program to fail to operate
- Examples:
  - Improper input data –
  - strings instead of numbers, for example
  - Non-working hardware –
  - a network drive that goes offline, or a missing USB drive, etc.
  - Unexpected operating conditions –
  - *e.g.*, a calculator program that tries to divide by 0, or trying to open a file that doesn't exist

Default Behavior

- Python "raises an exception" that describes what went wrong
- Program stops running

- Try:
  - x = 5 / 0
  - x = novar / 5
  - x = int(input('Enter a string? '))
  - h = open('badfile.txt', 'r')

Basic File-Reading Code

```
name = input('file name? ')

nlines, nwords, nchars = 0, 0, 0
with open(name, 'r') as h:
    for line in h.readlines():
        nlines += 1
        nchars += len(line)
        nwords += len( line.split() )
print(name, ':', nlines, nwords, nchars)
```

- What happens if the file doesn't exist?

Dealing with exceptions

- Python's `try` – `except` catches exceptions and allows special processing
- similar to an `if` – `else`
  - ...but `if` – `else` can't work with exceptions
  -
- Put statements that might cause an exception inside a "try" block
- Provide an "except" block that takes over if an exception occurs
  - Usually, do something that "fixes" the problem so the program can keep running

Catching an Exception

```
while True:
    name = input('file name? ')
    try:
        h = open(name, 'r')
        break
    except:
        print('Bad file name', name)

nlines, nwords, nchars = 0, 0, 0
for line in h.readlines:
    nwords += len(line.split() )
h.close()
print(name, ':', nwords)
```

try - except

- Pattern:

```
try:
   # Statements that might fail.
   # Keep this block as small as possible…
   # ….to avoid confusion
except:
   # Statements to repair the failure.
   # Do whatever is necessary here…
   # …but nothing more.
```

- Exceptions should occur *rarely* – if they happen often, consider using an if – else block instead

Start with this script:

```
#!/usr/bin/env python3
# This will crash and burn

x = None # create a variable x

while type(x) != float:
    s = input('Enter non-number? ')
    x = 2**(1/float(s))
print(x)
```

Try it out:

```
for i in range(5):

    s = input('? ')
    try:
        x = 2**(1/float(s))
    except:
        x = "Can't divide by zero."
    print( x )
```

Extend the previous script:

```
#!/usr/bin/env python3
# Get a number from the keyboard

x = None # create a variable x

while type(x) != float:
    s = input('? ')
    try:
        x = 2**(1/float(s))
    except:
        print('Try again.')
print(x)
```

Another approach to the previous script:

```
#!/usr/bin/env python3
# Get a number from the keyboard

while True:
    s = input('? ')
    try:
        x = 2**(1/float(s))
        break
    except:
        print('Try again.')
print(x)
```

You Can Catch Many Exceptions

- Multiple kinds of exceptions are possible
- Catch any exception into a variable
- Also possible to catch only one kind of exception but let others fail out
- Some kinds of exceptions:
  - NameError
  - ValueError
  - ZeroDivisionError
  - OverflowError
  - TypeError
  - UnicodeDecodeError

Try it out (catch all exceptions):

```
x = None # create a variable x

while type(x) != float or x == 0:
    s = input('? ')
    try:
        x = 2**(1/float(s))
    except Exception as e:
        print(e, 'Try again.')
print( x )
print('Done')
```

Try it out (catch a specific exception):

```
x = None # create a variable x

while type(x) != float or x == 0:
    s = input('? ')
    try:
        x = 2**(1/float(s))
    except ValueError as e:
        print(e, 'Try again.')
print( x )
print('Done')
```

Catching Multiple Exceptions

- A "try" clause can be followed by more than one "except" clause, to handle different kinds of exceptions
- Pattern:

```
try:
   # Statements that might fail.
except ValueError as e:
   # Statements to repair a Value failure.
except IOError as e:
   # Statements to repair an IO failure.
except Exception as e:
   # Statements to handle other exceptions.
```

Un-fixable Exceptions

- Some exceptions require handling, but should also terminate the program
- The "raise" statement allows the program to raise (or re-raise) an exception
- Pattern:

```
try:
   # Statements that might fail.
except ValueError as e:
   # Statements to repair a Value failure.
except Exception as e:
   # Statements to handle other exceptions.
   raise e    # re-raise exception, kill the program
```

Try it out (catch multiple exceptions):

```
while True:
    s = input('? ')
    try:
        y = 2**(1/float(s))
        break
    except ValueError as e:          # e.g., 'five'
        print(e, 'Try again.')
    except ZeroDivisionError as e:   # enter 0
        print('Can't use zero.')
    except Exception as e:           # enter  0.00001
        print('Oops')
        raise e

print( '{:.2f}'.format(y) )
```

Nested try-except Blocks

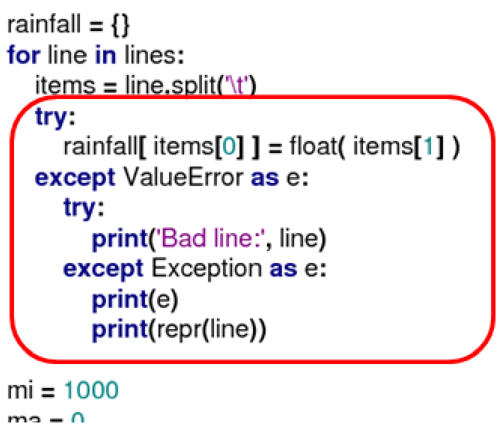
- Handling an exception can raise another (different) exception
- Exception-handling clause can include another try-except block to deal with new exceptions
- Pattern:

```
try:
   # Statements that might fail.
except ValueError as e:
   try:
      # Statements to repair a Value failure.
   except Exception as e:
      # Statements to handle secondary exceptions.
```

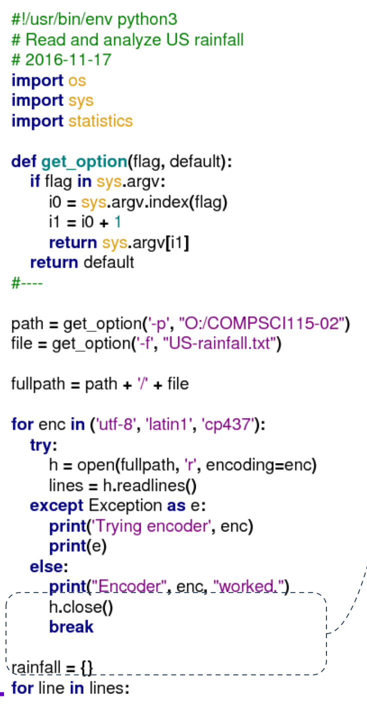
Example: Non-ASCII Rainfall File

- Script on next slide processes a file that contains non-ASCII characters

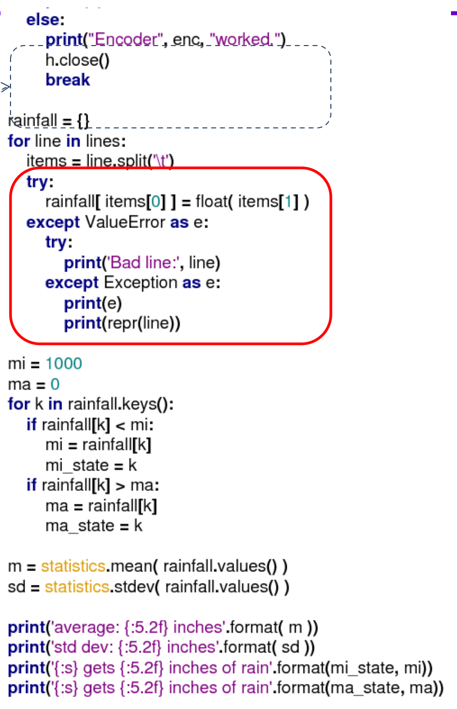
- After opening file, this try-except block
- attempts to convert float-type values
- If that fails, the except clause tries to display the offending line
- If *that* fails, a more robust display is shown



```
rainfall = {}
for line in lines:
    items = line.split('\t')
    try:
        rainfall[ items[0] ] = float( items[1] )
    except ValueError as e:
        try:
            print('Bad line:', line)
        except Exception as e:
            print(e)
            print(repr(line))

mi = 1000
ma = 0
```



```
#!/usr/bin/env python3
# Read and analyze US rainfall
# 2016-11-17
import os
import sys
import statistics

def get_option(flag, default):
    if flag in sys.argv:
        i0 = sys.argv.index(flag)
        i1 = i0 + 1
        return sys.argv[i1]
    return default
#----

path = get_option('-p', "O:/COMPSCI115-02")
file = get_option('-f', "US-rainfall.txt")

fullpath = path + '/' + file

for enc in ('utf-8', 'latin1', 'cp437'):
    try:
        h = open(fullpath, 'r', encoding=enc)
        lines = h.readlines()
    except Exception as e:
        print('Trying encoder', enc)
        print(e)
    else:
        print("Encoder", enc, "worked.")
        h.close()
        break

rainfall = {}
for line in lines:
```



```
    else:
        print("Encoder", enc, "worked.")
        h.close()
        break

rainfall = {}
for line in lines:
    items = line.split('\t')
    try:
        rainfall[ items[0] ] = float( items[1] )
    except ValueError as e:
        try:
            print('Bad line:', line)
        except Exception as e:
            print(e)
            print(repr(line))

mi = 1000
ma = 0
for k in rainfall.keys():
    if rainfall[k] < mi:
        mi = rainfall[k]
        mi_state = k
    if rainfall[k] > ma:
        ma = rainfall[k]
        ma_state = k

m = statistics.mean( rainfall.values() )
sd = statistics.stdev( rainfall.values() )

print('average: {:5.2f} inches'.format( m ))
print('std dev: {:5.2f} inches'.format( sd ))
print('{:s} gets {:5.2f} inches of rain'.format(mi_state, mi))
print('{:s} gets {:5.2f} inches of rain'.format(ma_state, ma))
```

The "else" clause

- Optional part of a "try - except" block
- Is placed after all "except" clauses
- Only executed if the "try" clause does *not* raise any exception
- Use the "else" clause for statements that should only be executed if all goes well
  - Example:
  - attempt to open() a file; only process and close it if the open() was successful.

A specific exception, and an "else" clause:

```
import sys
for arg in sys.argv:
    try:
        f = open(arg, 'r')
    except IOError as e:
        print('Cannot open', arg)
        print(e)   # …why not?
    else:
        print('{}: {} lines'.format( \
            arg, len(f.readlines()) ))
        f.close()
print('Done')
```

Finally... the "finally" clause

- Some actions require cleanup even if an exception occurs
- Example: open and read or write a file
- - the file should be closed after use, no matter what happens
- Pattern:

```
try:
   # Statements that might fail.
finally:
   # Statements that finish up
   # regardless of success or failure.
```

The "finally" clause in action:

```
arg = input('File? ')
f = open(arg, 'r')

numbers = []
try:
    for l in f.readlines():
        numbers.append( float(l) )
finally:
    f.close()
    print(len(numbers))
    print(sum(numbers)/len(numbers))
```