

Exception Handling

You will learn how to design programs that can recover from runtime errors.

James Tam

Error Handling With Exceptions

- Exceptions are used to deal with extraordinary errors ('exceptional ones').
- Typically these are fatal runtime errors ("crashes" program)
- A non-exhaustive list of examples:
 - trying to open a non-existent file,
 - invalid conversions between types of data (string to numeric)
 - exceeding the bounds of a composite type such as a list
 - division by zero.
- Current process from what you've been taught:
 - An exception occurs e.g. trying to open non-existent file
 - If no code is written to 'handle' the exception then a runtime error crashes the program.
 - You will learn how to write code to 'recover' from the exception (no run-time error).

James Tam

Basic Structure Of **Handling Exceptions**

Try:

Attempt something where exception error may happen

Except:

React to the error

Else: # Not always needed

What to do if no error is encountered

Finally: # Not always needed

Actions that must always be performed

James Tam

Simple Implementation Of **Recovering From** **File I/O Exceptions**

- **Name of the full online example:**

1basic_exception_handling_files.py

```
INPUT_FILENAME = "input.txt"
try:
    aFile = open("input.txt", "r")
    print(aFile.readline())
    aFile.close()
except IOError:
    print("Problem encountered during the interaction with %s"
          %(INPUT_FILENAME))
```

James Tam

Try-Except Vs. Previous (Branching): Non-Existent File

Only when file exists at
this point in the program
is this false

```
if(os.path.exists(FILE_PATH+FILE_NAME)):
    print("File %s exists in the location %s"
          %(FILE_NAME,FILE_PATH))
```

T

#The code below is added later in the program

Can still handle run time error case
where the file suddenly vanished at this
point but also it can also react to other
file IO errors

```
try:
    aFile = open("input.txt","r")
    print(aFile.readline())
    aFile.close()
except IOError:
    print("Problem encountered during the interaction with %s"
          %(INPUT_FILENAME))
```

James Tam

Better Implementation Of Reacting To File I/O Exceptions

- This second approach is more complex not only because it reads multiple lines but because it handles the exception better.
 - It allows the user to **continue running the program** rather than having to run it all over again if an exception is raised
 - Useful if something as minor as a typographical error is made while specifying the name of the input file.
 - **A flag is employed** to repeatedly attempt the file input process if an **exception is raised (occurs):**
 - **The flag is disabled** if no exception occurred during file input.
- Set flag to enabled
Do while(flag is set to enabled)

```
try
    Open file
    read from the file line by line
    close file
exception: reaction to exception to prevent a runtime error
else:
    no exception: flag is disabled
End-while
```

James Tam

Better Implementation Of Reacting To File I/O Exceptions (2)

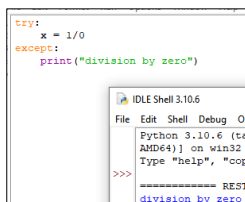
- Name of the full online example:
2better_example_of_exception_recovery.py

```
fileOK = False
while(fileOK == False):
    try:
        filename = input("Name of file to open: ")
        aFile = open(filename,"r")
        for line in aFile:
            print(line,end="")
    except IOError:
        print("Problem encountered during the interaction with
        %s" %(filename))
    else:
        aFile.close()
        fileOK = True
print()
print("Done reading from file")
```

James Tam

Some Types Of Exceptions That Can Be Caught

- IOError: occur during file input or output.
- ValueError: can occur when the value of a memory location is invalid e.g. during a conversion between types.
- ZeroDivisionError: occurs during division when the denominator is zero.
- IndexError: occurs when the index is invalid e.g. index is outside the list bounds.
- Except without specifying the type of the exception catches any type of run time exception.



James Tam

Catching Multiple Types Of Exceptions

- Name of the full online example:

3catching_multiple_exceptions.py

- The **order in which the exceptions occur** determines which exception is caught.
- The order IS NOT determined by the **order in which the exceptions are listed**.

```
aList = [1,2]
try:
    i = 2
    print(aList[i])
    quotient = 1/0
except IndexError:
    print("Outside list bounds")
except ZeroDivisionError:
    print("Cannot divide by
zero")
```

```
aList = [1,2]
try:
    i = 2
    quotient = 1/0
    print(aList[i])
except IndexError:
    print("Outside list bounds")
except ZeroDivisionError:
    print("Cannot divide by
zero")
```

James Tam

"Naming" Exceptions

- Similar to importing 'numpy' with a better name 'np' 231
import numpy as np
- A programmer can apparently provide a custom name for exceptions.
 - By apparently giving it this custom name one can display more detailed information behind the error.
 - JT: what's actually happening is that the exception itself is not being renamed. Instead you are **creating a new variable** which is an instance of the exception.
 - More information can be derived through approaches such as passing this variable to the print() function.
 - **Format:**
except <Python name for the exception> as <variable name>:
 - **Example:**
except ZeroDivisionError as **instanceOfDivisionByZero:**

James Tam

Contrast: Naming Vs. Not-Naming Exceptions

- Name of the full online example:

4naming_exceptions.py

```
try:
    quotient = 1/0
except ZeroDivisionError:
    #Type name only
    print(ZeroDivisionError)
```

<class 'ZeroDivisionError'>

```
try:
    quotient = 1/0
except ZeroDivisionError as instanceOfDivisionByZero:
    #More detailed information can be displayed.
    print(instanceOfDivisionByZero)
```

division by zero

James Tam

Variables: Comparing **Type** Vs. **Instance**

- Example: **integer**
`num = 12`
- Previous example: an exception
except **ZeroDivisionError** as `instanceOfDivisionByZero`

James Tam

Including This Because It's In The Notes¹

- ...but if desired one can list **several different types** of exceptions under a single 'except':

```
try:
    <code segment that may cause error>
except (<type>, <type>, ...) as <name>:
    <action to take when an exception occurs>
except (<type>, <type>, ...) as <name>:
    <action to take when an exception occurs>

except (IOError, IndexError) as detail:
    print (detail)
```



The variable *detail* will be assigned to which ever exception is raised first.

¹ Combining exceptions like this is contrary to what is said in another part of the lecture notes created by Richard Zhao and Jonathan Hudson: "It is a good coding practice to have one `except` for each exception type."

James Tam

JT: Contrast This With The Previous Screen

Try:

```
# Where the run time exception may occur.
```

IOError:

```
print("Error during file input-output")
```

ValueError:

```
print("Invalid type during conversion")
```

ZeroDivisionError:

```
print("denominator is zero")
```

IndexError:

```
print("List index beyond list bounds")
```

James Tam

When To Use Branches (“Conditionals”) Vs. Exceptions

- When to use branches:
 - To prevent the error/exception from happening.
 - **For specific cases**
- When to use exception handling:
 - To react to the error/exception after its occurred in order to prevent a run-time error (program crash).
 - **The error cannot be easily defined** (i.e. one of many non-specific events that may happen).
- Although this website is someone's personal opinion (apparently non-profit) site I do agree with this person's reason for:
 - using branching ('conditionals') over exceptions,
 - using exceptions over branching.
 - Consequently it's listed as a resource for your reference:
 - <https://avidpython.com/python-basics/when-to-use-try-except-instead-of-if-else/>

Branches: for specific cases
e.g. Boolean in branch for non-existent file, empty file name

Exception handling: IO runtime error can occur for numerous reasons e.g. no permission to access file, file becomes damaged, file is missing, storage device is dismounted etc. etc.

James Tam

Examples: Using Branching To Prevent Errors

- All the **conditions** can be clearly defined as **Boolean expressions**.

```
list = []
denominator = 0
index = 0
size = len(list)
ageString = input("Age in years: ")
if(denominator == 0):
    print("Cannot divide by zero")
if((index < 0) or (index >= size)):
    print("Index is outside list bounds")
if(ageString.isnumeric() == False):
    print(ageString, "is not a number")
```

James Tam

Example: Using Exception Handling Over Branches

- Reasons for using exceptions for file I/O:
 - The **availability** of the file may have changed since the **branch** e.g. file is deleted, locked by another process

<pre>#Risky PATH = "C:/231F/" NAME = "input_file.txt" if(os.path.exists(PATH+NAME)): ... else: aFile = open(...)</pre>	<pre>#Better try: aFile = open(...) except IOError:</pre>
---	---

- Also the **except** can handle run time errors that may arise beyond just a **non-existent file** e.g. file exists but it cannot be opened, file can be opened but suddenly it cannot be read/written to etc.

James Tam

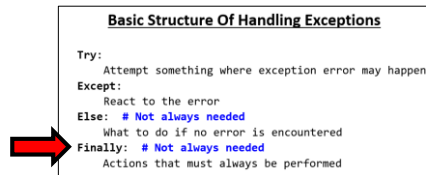
Example: Using Exception Handling Over Branches (2)

- Reasons for using exceptions for file I/O (continued)
 - An exception may occur when **opening or reading** from the file may occur for many different (some not so clear) reasons: no file with this name, file permissions don't allow access, file is corrupt, device on which the file was stored has been dismounted etc. etc.
 - The **IOError** body contains the code for all input/output errors.

```
try:
    aFile = open(file_name, "r")
    aLine = aFile.readline()
except IOError:
```

James Tam

The Last Construct: **Finally**



- Unless the program crashes this body contains instructions that are guaranteed to execute.
- It will execute whether or not an exception occurred.

James Tam

Finally: A Purely Academic Example

- **Name of the full online example:**
5try_catch_finally_learning_example_only.py
- **Finally body** always run regardless of **extraordinary by-passing** instructions (**you should never do this**).

```
def fun():
    numerator = float(input("Number to divide: "))
    denominator = float(input("Value to divide by: "))
    try:
        quotient = numerator / denominator

    except ZeroDivisionError:
        print("Cannot divide by zero")
        return() #Extremely bad style
    else:
        print("Quotient=%0.2f" %(quotient))
        return() #Extremely bad style
    finally:
        print("This message always appears")
```

James Tam

After This Section You Should Now Know

- How exceptions can be used in conjunction with file input and with invalid keyboard/console input.
- How components of the exception handling construct: try-except-else-finally each work.
- How to catch multiple types of exceptions.
- How to create an instance of an exception variable in order to access additional details about the error.

James Tam

Copyright Notification

- Unless otherwise indicated, all images in this presentation were provided courtesy of James Tam.

slide 22

James Tam