Exceptions

- Getting started with exceptions
- Additional exception techniques

Getting started with exceptions

- Overview
- Standard exceptions in Python
- Simple exception example
- Accessing the exception object

Overview

Exceptions are a run-time mechanism for indicating exceptional conditions in Python

- If you detect an "exceptional" condition, you can throw an exception
- An exception is an object that contains relevant error info

Somewhere up the call stack, the exception is caught and dealt with

- If the exception is not caught, your application terminates

Standard exceptions in Python

There are lots of things that can go wrong in a Python app

- Therefore, there are lots of different exception classes
- Each exception class represents a different kind of problem

Here are some of the standard exception classes in Python:

- KeyboardInterrupt
- OSError
- EOFError
- ValueError
- … etc.

Simple exception example

Here’s a simple example of how to deal with exceptions in a Python app

- The try block contains code that might cause an exception
- The except block catches a particular type of exception

```python
# Keep on looping until the user enters a number.

while True:

    try:
        inp = input("What's your favourite number? ")
        num = int(inp)
        print("Thanks, your favourite number is %d" % num)
        break

    except ValueError:
        print("Eek, that's not valid a number!")
```

Accessing the exception object

In your except clause, you can specify a name for the exception object you just caught

- Allows you to use the exception object in your except block

Example

- Catch ValueError and display error message on console

```python
# Keep on looping until the user enters a number.
while True:
    try:
        inp = input("What's your favourite number? ")
        num = int(inp)
        print("Thanks, your favourite number is %d" % num)
        break

    except ValueError as err:
        print("ValueError occurred: %s" % err)
```

2. Additional Exception Techniques

- Catching multiple exception types
- The "all ok" scenario
- Unconditional "wrap-up" code
- Exception hierarchies
- Defining custom exception classes
- Raising exceptions

Catching multiple exception types (1/2)

If your try block contains complex code, then multiple different types of exception might occur

- You can define multiple except blocks, to catch each type of error
- Optionally the last except block can be a catch-all (omit the type)

Example

```python
import sys

try:
    fh = open('favNum.txt')
    str = fh.readline()
    num = int(str.strip())
    print("The number in the file is %d" % num)

except OSError as err:
    print("OSError occurred: %s" % err)

except ValueError as err:
    print("ValueError occurred: %s" % err)



except:
```

Catching multiple exception types (2/2)

If you want to perform the same processing for several types of exception:

- Group the exceptions together in a single except block
- Specify the exception types as a tuple

```python
import sys

try:
    fh = open('favNum.txt')
    str = fh.readline()
    num = int(str.strip())
    print("The number in the file is %d" % num)

except (OSError, ValueError) as err:
    print("Error occurred: %s" % err)

except:
    print("Some other error occurred")
```

The "all ok" scenario

You can add an else block at the end of try…except

- Executed only if the try block completed successfully

```
import sys

try:
    fh = open('favNum.txt')
    str = fh.readline()
    num = int(str.strip())
    print("The number in the file is %d" % num)

except OSError as err:
    print("OSError occurred: %s" % err)
…

else:
    print("All completed OK!")
    fh.close()
```

Unconditional "wrap-up" code

You can add a finally block at the end of everything

- Always executed at the end of the try…except…else construct
- Whether an exception occurred or not

```python
import sys

try:
    fh = open('favNum.txt')
    str = fh.readline()
    num = int(str.strip())
    print("The number in the file is %d" % num)

except OSError as err:
    print("OSError occurred: %s" % err)
…

else:
    print("All completed OK!")
    fh.close()

finally:
    print("That's all folks. This message will always appear!")
```

Exception hierarchies (1/2)

Python organizes exceptions into an inheritance hierarchy

- Represents specializations of general error conditions

Example

- There are several subclasses of OSError

- BaseException
  - Exception
    - OSError
      - FileNotFoundError
      - FileExistsError
      - PermissionError
      - ChildProcessError

Exception hierarchies (2/2)

When you define an except block...

- It will catch that exception type, plus any subclasses

Example:

- "Special" processing for FileNotFoundError exceptions
- "Generic" processing for any other kind of OSError exceptions

```python
import sys

try:
    fh = open('favNum.txt')
    str = fh.readline()
    num = int(str.strip())
    print("The number in the file is %d" % num)

except FileNotFoundError as err:
    print("File not found: %s" % err)

except OSError as err:
    print("More general OSError occurred: %s" % err)
```

Defining custom exception classes

You can define custom exception classes

- To represent important types of error in your application

How to do it:

- Define a class that inherits from Exception (or a subclass)
- Implement `` `__init__` `` and `` `__str__` `` methods

Example:

```python
class MyError(Exception):

    def __init__(self, value):
        self.value = value

    def __str__(self):
        return repr(self.value)
```

Raising exceptions

To raise (i.e. trigger) an exception:

- Use the raise keyword
- Specify the type of exception you want to raise
- Pass in any constructor arguemnts as appropriate

Example:

```python
try:
    raise MyError("EEK ERROR ERROR ERROR")

except MyError as err:
    print("It appears my exception occurred, the value is %s" % err.value)
```

Any questions?