Exceptions

Python 3

home (../handouts.html)

Introduction: Exceptions

Exception handling is a flow control mechanism -- rerouting program flow when an error occurs.

When a program encounters an error it is referred to as an *exception*.

Errors are endemic in any programming language, but we can broadly classify errors into two categories:

- *unanticipated errors* (caused by programmer error or oversight)
- *anticipatable errors* (caused by incorrect user input, environmental errors such as permissions or missing files, networking or process errors such as database failures, etc.) *Trapping exceptions* means deciding what to do when an anticipatable error occurs. When we trap an error using a *try/except* block, we have the opportunity to have our program respond to the error by executing a block of code. In this way, exception handling is another flow control mechanism: **if** this error occurs, **do** something about it.

Objectives for the Unit: Exceptions

- identify exception types (IndexError, KeyError, IOError, WindowsError, etc.)
- use *try:* blocks to identify code where an exception is anticipatable
- use *except:* blocks to specify the anticipatable exception and to provide code to be run in the event of an exception
- trap multiple exception types anticipatable from a **try:** block, and chain **except:** blocks to execute different code blocks depending on which exception type was raised.

Summary: Exceptions signify an error condition

Exceptions are *raised* when an error condition is encountered; this condition can be handled.

In each of these *anticipateable* errors below, the user can easily enter a value that is invalid and would cause an error if not handled. We are not handling these errors, but we should:

ValueError: when the wrong value is used with a function or statement

```
uin = input('please enter an integer:  ')

intval = int(uin)               # user enters 'hello'

print('{} doubled is {}'.format(uin, intval*2))
```

KeyError: when a dictionary key cannot be found. Here we ask the user for a key in the dict, but they could easily enter the wrong key:

```
mydict = {'1972': 3.08, '1973': 1.01, '1974': -1.09}

uin = raw_input('please enter a year: ')    # user enters 2116

print 'mktrf for {} is {}'.format(uin, mydict[uin])


Traceback (most recent call last):
   File "getavg.py", line 4, in
KeyError: '2116'
```

IndexError: when a list index can't be found. Here we ask the user to enter an argument at the command line, but they could easily skip entering the argument:

```
import sys

user_input = sys.argv[1]              # user enters no arg at command line


Traceback (most recent call last):
   File "getarg.py", line 3, in
IndexError:  1
```

OSError: when a file or directory can't be found. Here we ask the user to enter a filename

```
import os

user_file = raw_input('please enter a filename:  ')    # user enters a file that doesn't exist

file_size = os.listdir(os.path.getsize(user_file))


Traceback (most recent call last):
    File "getsize.py", line 5, in
OSError:  No such file or directory:  'mispeld.txt'
```

In each of these situations we are working with a process that may make an error (in this case, the 'process' is the user). We can then say that these errors are *anticipatable* and thus can be handled by our script.

Summary statements: *try* block and *except* block

The **try:** block contains statements from which a potential error condition is anticipated; the **except:** block identifies the anticipated exception and contains statements to be excecuted if the exception occurs.

How to avoid an anticipatable exception?

- wrap the lines where the error is anticipated in a **try:** block
- define statements to be executed if the error occurs

```
try:
    firstarg = sys.argv[1]
    secondarg = sys.argv[2]
except IndexError:
    exit('error:  two args required')
```

This code anticipates that the user may not pass arguments to the script. If two arguments are not passed, then **sys.argv[1]** or **sys.argv[2]** will fail with an **IndexError** exception.

Summary technique: trapping multiple exceptions

Multiple exceptions can be trapped using a **tuple** of exception types.

```
try:
    firstarg = sys.argv[1]
    secondarg = sys.argv[2]

    firstint = int(firstarg)
    secondint = int(secondarg)

except (IndexError, ValueError):
    exit('error:  two int args required')
```

In this case, whether an **IndexError** or a **ValueError** exception is raised, the **except:** block will be executed.

Summary technique: chaining except: blocks

The same **try:** block can be followed by multiple **except:** blocks.

```
try:
    firstint = int(sys.argv[1])
    secondint = int(sys.argv[2])
except IndexError:
    exit('error:  two args required')
except ValueError:
    exit('error:  args must be ints')
```

The exception raised will be matched against each type, and the first one found will excecute its block.