Scripting & Computer Environments

Core Python: Exceptions

IIIT-H

Errors vs Exceptions

- Errors are everyday ‘friends’ of a programmer.
- Types
  - Parsing (Syntax) errors
  - Logical errors
  - Runtime errors
- Examples?

Exceptions

Events that can modify the control flow through a program.

Why Study?

- Exceptions are inevitable and could be fatal.

  e.g. The Y2K bug, critical systems (e.g. industrial control systems, grid systems)

- Secure Programming (a.k.a Defensive Programming)
  - Not an option, especially these days.
  - Runtime errors are mostly due to external reasons.

    e.g. Poor user input, malicious input, some sort of failure

- Proper handling of them is a rewarding process.

Exceptions

- Are generated automatically on errors.
- Built-in vs User-defined
- Can be triggered and handled by our code
- Generally, a two-phase process:
  1. Detection of exception condition
     - Interpretter *raises* (throws/triggers/generates) the exception.
     - Programmer too can raise explicitly.
  2. Exception handling

     e.g. Ignore error, log error, abort program, remedial actions, etc

Example

C++, Python, Java, Eiffel, Modula-3

Exceptions (Python)

- Some standard exceptions you've probably encountered:
  - *NameError* - access uninitialized variable
  - *SyntaxError*
  - *ZeroDivisionError*
  - *KeyError* - access non-existing dictionary key
  - *IndexError* - access out-of-range index
  - *IOError* - input/output (e.g. in file read/write)
  - *TypeError* - operations with invalid type.
- On error, the *default exception handler* throws the error messages + stack trace.

The Constructs

- Exceptions can be detected by a `try` statement.
- Flavors :
    - `try...except...[else]`
    - `try...finally`

try...except

```
try:
   <statements>                     # suspicious code
except <e1>:
    <statements>                    # if <e1> was raised
except (e2, e3, ...eN):
    <statements>                    # if any of e2...eN was raised
except:
      <statements>                  # for all other exceptions
else:                               # optional else block
    <statements>
```

Example

```
try:
     f=open(‘IDoNotExist.txt’)
except IOError:
     print ‘Unable to open the file’
```

Example

```
try:
    float(‘this is test’)
    float([1,2])
except(ValueError, TypeError):
    print ‘Invalid Argument Encountered’
else:
    print ‘No exception occured!’
```

try...finally

```
try:
   <statements>
finally:
   <statements>                  # Always run this code
```

- Unlike an `except` clause, `finally` is not used to handle exception.
- The clause executes regardless of exceptions within the `try` clause.
- Useful to specify cleanup actions that must occur, regardless of exception.

  e.g. File close, server disconnects, etc

Example

```
try:
     n=float(raw_input('Enter your number:'))
     double = 2 * n

finally:
     print 'Who can stop me from executing?'

print 'Double=', double
```

Exception Arguments

- Exceptions can have arguments.
- Are values that give additional info about the error (if any), usually error string, number and location.
- Captured by supplying a variable in the `except` clause.

exception args

```
try:
   try_block
except <single or multiple exception>, argument:
   exception handler
```

- An alternative is by accessing the `exc_info()` method of the `sys` module (returns a 3-tuple info).

Raising Exceptions

- To explicitly raise exceptions, use the `raise` statement.

The raise statement

```
raise <exception_to_be_raised> [, args]
```

- If no exception supplied with the `raise` statement, the last exception (if any) in the current try block is re-raised; otherwise, `TypeError` (no exception to re-raise).

Example

```
try:
    raise NameError
except NameError:
    print 'Exception ocurred!'
    raise
```

Assertions

- Are diagnostic predicates which *must evaluate to true*.
- If false, an `AssertionError` exception is thrown.
- Think of them as conditional raise i.e. `raise-if/raise-if-not`

The assert statement

```
assert <test>
```

Example

```
>>>assert 2==‘2’
>>>assert range(2)== [1,2]

>>>def f(n):
      assert n>0                               # must be positive
      return math.sqrt(n)
```