Python 101 (part 8): An Exceptionally Clever Snake

By Vikram Vaswani

Table of Contents

Not Your Regular Tour.....1

Anatomy Of An Exception.....2

Trying Harder.....4

Different Strokes.....7

Passing The Buck.....11

Bad Boys.....15

Raising The Bar.....17

Strong Pythons (And The Exceptions That Love Them).....20

The End Of The Affair.....21

Not Your Regular Tour

If you've been following along since the beginning, I think you'll agree with me that it's certainly been an interesting eight weeks.

Together, we've visited all the standard attractions − we've thumbed through the rulebook for strings, numbers, lists, dictionaries and those oddly−named tuples; mucked around with system files and directories; packaged code into functions and functions into modules; and dissected the Python interpreter and some of Python's innumerable built−in functions − and indulged in some decidedly non−standard activities − learning the names of exotic Italian dishes; stomping on all kinds of creepy−crawlies; attempting to con unwitting customers into purchasing air in a bottle; and hooking up with old flames, new superheroes and the complete cast of "Star Wars".

In this concluding article, I'm going to be demonstrating Python's error−handling routines, and showing you how to wrap your code in them to avoid violent − and potentially embarrassing − flame−outs. I'll be explaining the different types of errors you might encounter and the Python constructs available to manage them, with examples that demonstrate just how powerful this capability really is.

Talk about getting your money's worth!

Anatomy Of An Exception

No developer, no matter how good (s)he is, writes bug−free code all the time. Which is why most programming languages − including Python − come with built−in capabilities to catch errors and take remedial action. This action could be something as simple as displaying an error message, or as complex as heating your computer's innards until they burst into flame (just kidding!)

Normally, when a Python program encounters an error, be it syntactical or logical, it exits the program at that stage itself with a message indicating the cause of the error. Now, while this behaviour is acceptable during the development phase, it cannot continue once a Python program has been released to actual users. In these "live" situations, it is unprofessional to display cryptic error messages (which are usually incomprehensible to non−technical users); rather, it is more professional to intercept these errors and either resolve them (if resolution is possible), or notify the user with a clear error message (if not).

The term "exceptions" refers to those errors which can be tracked and controlled. For example, if a function attempts an unsupported operation on a built−in Python object (say, modifying an immutable object), Python will generate a "TypeError" exception, together with a stack trace or detailed explanation of the problem. Exceptions like these can be caught by the application, and appropriately diverted to an exception−handling routine.

An example might make this clearer. Consider the following Python program,

```
#!/usr/bin/python



# set up tuple
dessert = ('apple pie', 'chocolate fudge cake', 'icecream')



# change tuple value
dessert[1] = 'chocolate brownies'
```

which generates a TypeError because I'm trying to modify a tuple.

```
Traceback (innermost last):
File "dessert.py", line 5, in ?
dessert[1] = 'chocolate brownies'
TypeError: object doesn't support item assignment
```

Now, let's add some code to trap and handle the exception gracefully.

```
#!/usr/bin/python



try:
# try running this code
dessert = ('apple pie', 'chocolate fudge cake', 'icecream')
dessert[1] = 'chocolate brownies'
except TypeError:
# if error, do this...
print "Whoops! Something bad just happened. Terminating
script."
```

And this time, when the script is executed, it will not generate a TypeError − instead, it will output

```
Whoops! Something bad just happened. Terminating script.
```

This is a very basic example of how Python exceptions can be trapped, and appropriate action triggered. As you will see, the ability to handle errors in a transparent manner throws open some pretty powerful possibilities...

Trying Harder

Python offers two flavours of exception handler − the first is the "try−except" construct you just saw, while the second is the "try−finally" construct.

The "try−except" construct − actually the "try−except−else" construct − allows developers to trap different types of errors and execute appropriate exception−handling code depending on the exception type. It's a lot like the "if−elif−else" conditional construct, allowing for the execution of different code blocks depending on the type of error generated.

The structure of a "try−except−else" block looks like this:

```
try:
execute this block
except err1:
execute this block if exception "err1" is generated
except err2:
execute this block if exception "err2" is generated



... and so on ...



else:
execute this block
```

When Python encounters code wrapped within a "try−except−else" block, it first attempts to execute the code within the "try" block. If this code is processed without any exceptions being generated, Python checks to see if the optional "else" block is present. If it is, the code within it is executed.

If an exception is encountered while running the code within the "try" block, Python stops execution of the "try" block at that point and begins checking each "except" block to see if there is a handler for the exception. If a handler is found, the code within the appropriate "except" block is executed; if not, control either moves to the parent "try" block, if one exists, or to the default handler, which terminates the program and displays a stack trace.

Once the "try" block has been executed and assuming that the program has not been terminated, the lines following the "try" block are executed.

Let's illustrate this with a simple Python program, which accepts two numbers and attempts to divide them by each other.

```
#!/usr/bin/python
```

```
alpha = input("Gimme a number: ")
beta = input("Gimme another number: ")
gamma = alpha / beta
print alpha, "divided by", beta, "is", gamma
```

Now look what happens when I run this with different values.

```
Gimme a number: 10
Gimme another number: 2
10 divided by 2 is 5



Gimme a number: 10
Gimme another number: 0
Traceback (innermost last):
File "div.py", line 6, in ?
gamma = alpha / beta
ZeroDivisionError: integer division or modulo



Gimme a number: 67347328
Gimme another number: 943282646373739
Traceback (innermost last):
File "div.py", line 4, in ?
beta = input("Gimme another number: ")
OverflowError: integer literal too large
```

Let's now add a couple of exception handlers to this code, so that it knows how to gracefully handle errors like the ones above:

```
#!/usr/bin/python



try:
alpha = input("Gimme a number: ")
beta = input("Gimme another number: ")
gamma = alpha / beta
print alpha, "divided by", beta, "is", gamma
except ZeroDivisionError:
```

```
print "Cannot divide by zero!"
except OverflowError:
print "Number too large!"
else:
print "No errors encountered!"



print "-- All done --"
```

And now let's try the different test cases above again:

```
Gimme a number: 10
Gimme another number: 2
10 divided by 2 is 5
No errors encountered!
-- All done --



Gimme a number: 10
Gimme another number: 0
Cannot divide by zero!
-- All done --



Gimme a number: 823748237
Gimme another number: 234378264732647326476327
Number too large!
-- All done --
```

Depending on the type of error encountered, the appropriate exception handler is triggered and an error message displayed. The optional "else" block is executed at the end of the script *only* if no exceptions are encountered.

Once an exception has been caught and resolved, the remainder of the "try" block is ignored and Python executes the lines following the entire "try−except−else" block.

Different Strokes

You can use a single "except" statement to handle more than one error by separating the various exception names with commas and enclosing them in parentheses. Modifying the example above, we have

```
#!/usr/bin/python


try:
alpha = input("Gimme a number: ")
beta = input("Gimme another number: ")
gamma = alpha / beta
print alpha, "divided by", beta, "is", gamma
except (ZeroDivisionError, OverflowError):
print "You entered an illegal value!"
else:
print "No errors encountered!"


print "-- All done --"
```

If you take a close look at the stack trace on the previous page, you'll see that when Python encounters an exception, it prints both an exception name and a descriptive string explaining the error.

```
ZeroDivisionError: integer division or modulo


OverflowError: integer literal too large
```

This descriptive text can also be caught and used by an exception handler, if you define a variable to store it in the "except" statement. Consider the following code:

```
#!/usr/bin/python


try:
alpha = input("Gimme a number: ")
beta = input("Gimme another number: ")
gamma = alpha / beta
```

```
print alpha, "divided by", beta, "is", gamma
except ZeroDivisionError, desc:
print "Illegal value (", desc, ")"
except OverflowError, desc:
print "Illegal value (", desc, ")"
else:
print "No errors encountered!"



print "-- All done --"
```

In this case, the "desc" variable in the exception handler stores the descriptive error message generated by Python; this variable may then be used within the handler.

Here's the output:

```
Gimme a number: 10
Gimme another number: 5
10 divided by 5 is 2
No errors encountered!
-- All done --



Gimme a number: 10
Gimme another number: 0
Illegal value ( integer division or modulo )
-- All done --



Gimme a number: 89747583478534785
Illegal value ( integer literal too large )
-- All done --
```

This works even if your exception handler is handling more than one exception.

```
#!/usr/bin/python



try:
alpha = input("Gimme a number: ")
beta = input("Gimme another number: ")
```

```
gamma = alpha / beta
print alpha, "divided by", beta, "is", gamma
except (ZeroDivisionError, OverflowError), desc:
print "Illegal value (", desc, ")"
else:
print "No errors encountered!"



print "-- All done --"
```

Now, the "try" statement can only deal with exceptions that it knows about. What about the ones the developer can't predict?

```
Gimme a number: 76
Gimme another number: abc
Traceback (innermost last):
File "div.py", line 5, in ?
beta = input("Gimme another number: ")
File "gt;", line 0, in ?
NameError: abc
```

It's possible to use a general "except" statement to handle *any* type of exception generated by the interpreter − simply omit the exception name from the "except" statement. The following code snippet illustrates this technique:

```
#!/usr/bin/python



try:
alpha = input("Gimme a number: ")
beta = input("Gimme another number: ")
gamma = alpha / beta
print alpha, "divided by", beta, "is", gamma
except:
pass



print "-- All done --"
```

In this case, it doesn't matter what type of exception Python generates − the generic handler will catch it, ignore it and continue to process the rest of the script.

```
Gimme a number: asd
-- All done --


Gimme a number: 10
Gimme another number: 0
-- All done --


Gimme a number: 58439058349058934859
-- All done --


Gimme a number: 10
Gimme another number: 2
10 divided by 2 is 5
-- All done --
```

It should be noted, however, that this approach, although extremely simple, is *not* recommended for general use. It is poor programming practice to trap all errors, regardless of type, and ignore them; it is far better − and more professional − to anticipate the likely errors ahead of time, and use the "try−except" construct to isolate and resolve them.

Passing The Buck

If one exception handler is nested within another, Python typically uses the one closest to where the exception occurs. To illustrate this, consider the following code snippet:

```
#!/usr/bin/python


# nested_handlers.py


def popeye():
try:
olive()
except NameError:
print "Error in popeye"


def olive():
try:
print someUnnamedVar
except NameError:
print "Error in olive"


try:
popeye()
except NameError:
print "Error in main"
```

In this case, there are three exception handlers defined for the same type of exception. The outermost one is in the main program body, the next is within the popeye() function called from the main script, and the third is within the olive() function called by popeye().

When olive() runs, it generates a "NameError" exception, which is immediately handled by its own "try" block.

```
$ nested_handlers.py
Error in olive
```

However, if olive() didn't have a "try" block, or its "try" block didn't account for "NameError" exceptions,

```
#!/usr/bin/python



def popeye():
try:
olive()
except NameError:
print "Error in popeye"



def olive():
print someUnnamedVar



try:
popeye()
except NameError:
print "Error in main"
```

the exception would be passed up to the previous level, the calling popeye() function, which would generate

```
$ nested_handlers.py
Error in popeye
```

If popeye()'s exception handler didn't have the ability to handle the exception, the exception would move up even further, to the main body of the script,

```
#!/usr/bin/python



def popeye():
try:
olive()
# this wouldn't handle NameError
except IndexError:
print "Error in popeye"
```

```
def olive():
print someUnnamedVar



try:
popeye()
except NameError:
print "Error in main"
```

which would result in

```
$ nested_handlers.py
Error in main
```

And if there weren't any handlers capable of resolving the error,

```
#!/usr/bin/python



def popeye():
olive()



def olive():
print someUnnamedVar



popeye()
```

the exception would be processed by the default handlers, which would kill the script and print a stack trace.

```
Traceback (innermost last):
File "./test.py", line 9, in ?
popeye()
File "./test.py", line 4, in popeye
olive()
File "./test.py", line 7, in olive
print someUnnamedVar
```

```
NameError: someUnnamedVar
```

Bad Boys

Most of what you've just learned also applies to Python's other exception−handlng construct, the "try−finally" statement. The "try−finally" statement block differs from "try−except−else" in that it merely detects errors; it does not provide for a mechanism to resolve them. It is typically used to ensure that certain statements are always executed when an error (regardless of type) is encountered.

The "try−finally" statement block looks like this:

```
try:
execute this block
finally:
if exceptions generated, execute this block
```

If an exception is encountered when running the code within the "try" block, Python will stop execution at that point; jump to the "finally" block; execute the statements within it; and then pass the exception upwards, to the parent "try" block, if one exists, or to the default handler, which terminates the program and displays a stack trace.

Here's an example:

```
#!/usr/bin/python



dessert = ('apple pie', 'chocolate fudge cake', 'icecream')



try:
# generate error by accessing index out of range
print dessert[10]
finally:
print "Something bad happened"
```

When this program runs, an IndexError exception will be generated and the "finally" block will execute, printing an error message. Control will then flow to the parent exception handler, which is the Python interpreter in this case; the interpreter will terminate the program and print a stack trace.

```
$ dessert.py
Something bad happened
Traceback (innermost last):
File "dessert.py", line 7, in ?
```

```
print dessert[10]
IndexError: tuple index out of range
```

Since "try−finally" blocks simply detect errors, passing the resolution buck upwards to the parent "try" block, it's possible to nest them within "try−except−else" blocks. Take a look:

```
#!/usr/bin/python



try:
dessert = ('apple pie', 'chocolate fudge cake', 'icecream')



try:
# generate error by accessing index out of range
print dessert[10]
finally:
print "Something bad happened"



except IndexError:
print "You attempted to access a non−existent element. Bad
boy!"
except NameError:
print "You attempted to access a non−existent object. What are
you
thinking?"
```

Here's what'll happen when you run it:

```
Something bad happened
You attempted to access a non−existent element. Bad boy!
```

Raising The Bar

Thus far, you've been working with Python's built−in exceptions, which can handle most logical or syntactical expressions. However, Python also allows you to get creative with exceptions, by generating your own custom exceptions if the need arises.

This is accomplished via Python's "raise" statement, which is used to raise errors which can be detected and resolved by the "try" family of exception handlers. The "raise" statement needs to be passed an exception name, and an optional descriptive string. When the exception is raised, this exception name and description will be made available to the defined exception handler.

Let's go to a quick example − the line of code

```
raise ValueError, "What on earth are you thinking?!"
```

generates the following error.

```
Traceback (innermost last):
File "./test.py", line 3, in ?
raise ValueError, "What on earth are you thinking?!"
ValueError: What on earth are you thinking?!
```

You can also name and use your own exceptions.

```
#!/usr/bin/python


import os


# define error object
error = "someError"


# function to raise error
def checkName(name):
if (name != os.environ["USER"]):
raise error, "Username mismatch!"
```

```
name = raw_input("Enter your system username: ")
checkName(name)
```

In this case, if the username entered at the prompt does not match the name stored in the environment variable $USER, Python will raise a user−defined exception named "someError", with a string of text describing the nature of the error. Take a look:

```
Enter your system username: john
Traceback (innermost last):
File "checkuser.py", line 16, in ?
checkName(name)
File "checkuser.py", line 11, in checkName
raise error, "Username mismatch!"
someError: Username mismatch!
```

Note that the exception must be assigned to an object in order for it to work correctly.

```
# define error object
error = "someError"
```

Trapping user−defined errors is exactly the same as trapping pre−defined Python errors. The following refinement of the code above illustrates this:

```
#!/usr/bin/python


import os


# define error object
error = "someError"


# function to raise error
def checkName(name):
if (name != os.environ["USER"]):
raise error, "Username mismatch!"
```

```
# try this code
try:
name = raw_input("Enter your system username: ")
checkName(name)
except error, desc:
print desc
```

Here's the output of the script above, when the wrong username is entered.

```
Enter your system username: john
Username mismatch!
```

Strong Pythons (And The Exceptions That Love Them)

A number of standard exceptions are built into Python − here's a list of the most common ones.

IOError − generated when an I/O operation fails;

ImportError − generated when a module import fails

IndexError − generated when an attempt is made to access a non−existent element index;

KeyError − generated when an attempt is made to access a non−existent dictionary key;

MemoryError − generated when an out−of−memory error occurs;

NameError − generated when an attempt is made to access a non−existent variable;

SyntaxError − generated when the interpreter finds a syntax error;

TypeError − generated when an attempt is made to run an operation on an incompatible object type;

ZeroDivisionError − generated when an attempt is made to divide by zero.

For a complete list, take a look at http://www.python.org/doc/current/lib/module−exceptions.html

The End Of The Affair

And that just about brings the curtain down on this series of tutorials. I hope you've enjoyed reading these articles as much as I've enjoyed writing them, and that you now have a better understanding of the wonder that is Python.

Should you need more information on the topics covered in this tutorial, please consider visiting the following Web sites:

The official Python Web site, at http://www.python.org/

The Python Cookbook, at http://aspn.activestate.com/ASPN/Cookbook/Python

The Vaults of Parnassus, at http://www.vex.net/parnassus/

Jython (Python+Java), at http://www.jython.org/

Python HOWTOs, at http://py−howto.sourceforge.net/

The Python FAQ, at http://www.python.org/doc/FAQ.html

The Python Quick Reference, at http://starship.python.net/quick−ref1_52.html

As for me, I'll be back soon to talk about object−oriented programming and Web development with Python − so keep an eye out for those articles. Until then, though, stay healthy...and thanks for watching!

Note: All examples in this article have been tested on Linux/i586 with Python 1.5.2. Examples are illustrative only, and are not meant for a production environment. YMMV!