Python 3: Handling errors

Bruce Beckles mbb10@cam.ac.uk

Bob Dowling rjd4@cam.ac.uk

29 October 2012

Prerequisites

This self-paced course assumes that you have a knowledge of Python 3 equivalent to having completed one or other of

- Python 3: Introduction for Absolute Beginners, or
- Python 3: Introduction for Those with Programming Experience

Some experience beyond these courses is always useful but no other course is assumed.

The course also assumes that you know how to use a Unix text editor (gedit, emacs, vi, …).

Facilities for this session

The computers in this room have been prepared for these self-paced courses. They are already logged in with course IDs and have home directories specially prepared. Please do not log in under any other ID.

At the end of this session the home directories will be cleared. Any files you leave in them will be deleted. Please copy any files you want to keep.

The home directories contain a number of subdirectories one for each topic.

For this topic please enter directory `exceptions`. All work will be completed there:

```
$ cd exceptions
$ pwd
/home/x250/exceptions
$
```

These courses are held in a room with two demonstrators. If you get stuck or confused, or if you just have a question raised by something you read, please ask!

These handouts and the prepared folders to go with them can be downloaded from
`www.ucs.cam.ac.uk/docs/course-notes/unix-courses/pythontopics`

You are welcome to annotate and keep this handout.

A good starting point in the formal Python 3 documentation for the topics covered here can be found online at
`docs.python.org/release/3.2.3/reference/executionmodel.html#exceptions`

Table of Contents

Notation

Warnings

Warnings are marked like this. These sections are used to highlight common mistakes or misconceptions.

Exercises

Exercise 0

Exercises are marked like this. You are expected to complete all exercises. Some of them do depend on previous exercises being successfully completed.
An indication is given as to how long we expect the exercise to take. Do not panic if you take longer than this. If you are stuck, ask a demonstrator.
Exercises marked with an asterisk (*) are optional and you should only do these if you have time.

Input and output

Material appearing in a terminal is presented like this:

```
$ more lorem.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
--More--(44%)
```

The material you type is presented like this: **`ls`**. (Bold face, typewriter font.)

The material the computer responds with is presented like this: "`Lorem ipsum`". (Typewriter font again but in a normal face.)

Keys on the keyboard

Keys on the keyboard will be shown as the symbol on the keyboard surrounded by square brackets, so the "A key" will be written "[A]". Note that the return key (pressed at the end of every line of commands) is written "[↵]", the shift key as "[⇧]", and the tab key as "[↹]". Pressing more than one key at the same time (such as pressing the shift key down while pressing the A key) will be written as "[⇧]+[A]". Note that pressing [A] generates the lower case letter "a". To get the upper case letter "A" you need to press [⇧]+[A].

Content of files

The content[1] of files (with a comment) will be shown like this:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur.                                  This is a comment about the line.
```

1 The example text here is the famous "lorem ipsum" dummy text used in the printing and typesetting industry. It dates back to the 1500s. See http://www.lipsum.com/ for more information.

What's in this course

Thus far when our Python scripts have crashed we've assumed there is something wrong with the script and tried to fix it. This is a sensible approach to take, but it doesn't work all the time. Sometimes external factors can cause our script to crash. Imagine our script is writing to disk and runs out of space. Even if we check to make sure there's enough space before writing the file, we might still run out of space, because some other program or background process might write something to the disk after we've checked the amount of disk space. So how can we write robust scripts that can deal with failures caused by "unexpected" changes in external circumstances?

When something like that happens, Python regards it as an error and stops our script. So if we want to be able to deal with these situations we need to be able to take control from Python when an error occurs and take some remedial action ourselves. This is called "**error handling**".

This topic introduces handling errors in Python but does not cover every last detail.

1. How does Python deal with errors?
2. Some useful miscellany
    2.1. Representing "nothing": `None`
    2.2. Doing "nothing": `pass`
    2.3. Controlling loops: `break`
    2.4. Controlling loops: `continue`
3. What are the different types of exception?
4. Generating exceptions: `raise`
5. Exception handling: `try`...`except`
6. Getting information about the exception
7. What else can go wrong with our function?
8. Handling multiple exceptions
    8.1. Another way to get at the exception: `exc_info()`

The most relevant online Python documentation for this topic is at:
`http://docs.python.org/py3k/reference/executionmodel.html#exceptions`

You can explore Python's help on this topic by using the `help()` function:

```
>>> help('EXCEPTIONS')
```

How does Python deal with errors?

Errors in Python are called "**exceptions**". When something goes wrong, Python will "**raise an exception**", i.e. generate an error. It is up to your script to "**handle**" (deal with) that error. The code that deals with such errors is called an "**exception handler**". If your script does not have an exception handler for the exception that occurred, then Python's default exception handler is used, which normally halts your script and prints out some information about what has gone wrong (as we have seen on various occasions in the Python introductory courses when our scripts haven't worked). For example:

```
$ python3 wrong1.py
Traceback (most recent call last):
  File "wrong1.py", line 3, in <module>
    print(hello)
NameError: name 'hello' is not defined
```

Python's default exception handler first prints out the "**traceback**", which tells us how we got to this sorry state. Let's examine this in some detail before moving on.

A traceback is the error's history: how we got to be here making this mistake, if you like. The error itself will come at the end, preceded with how we got to be there. This is why it says the traceback shows the "most recent call last". (If our script has jumped through several hoops to get there we will see a list of the hoops.) In the above example the error occurs as soon as we try to do something, so the traceback is pretty trivial.

This particular traceback consists of two lines. In more complex examples there would be more than two but they would still have the general structure of a "file" line followed by a "what happened" line. The file line tells us what file we were running (`wrong1.py`) and the line where the error occurred.

(If we were typing this into an interactive Python session, then the file line would says that the error occurred in a file called "`<stdin>`". What this actually means is that the error occurred on some Python being fed to the Python interpreter from "**standard input**". Standard input means our keyboard: we typed the erroneous line and so the error came from us. In an interactive session, each line at the "`>>>`" prompt is processed before the prompt comes back and counts as "line 1", so if we tried this interactively it would tell us the error was on line 1.)

The "`<module>`" refers to what function (and module) we were in when the error occurred. This command wasn't in a function (or a module), so we get "`<module>`" as the function name (indicating we weren't in a function, or a module, for that matter).

After displaying the traceback, Python tells us what actually went wrong: first it tells us the type of error that occurred (a `NameError` in this case), and then it gives us whatever information it has about the particular circumstances of this occurrence of the error. In this case it tells us that there is no such variable as `hello` ("`name 'hello' is not defined`"). Sometimes, as here, this information is just a text message, but sometimes it may contain other information, such as an error number, or a filename or some other value the code that went wrong didn't like.

Some useful miscellany

So how do we write exception handlers of our own? Before we can do this, there are a few quick miscellaneous concepts we should cover. (Feel free to skip over any of these concepts with which you are already familiar.)

Representing "nothing": `None`

The value `None` is a Python special value (Python calls it a "**null object**") designed for situations when you need a value, but that value should not represent anything. For instance, it is often used as a "placeholder" for variables that will be attached to a value later in the script. It has its own separate type (`NoneType`). For the purpose of tests, it is equivalent to `False` (i.e. its "truth value" is `False`).

Many Python functions use `None` to mean that there are no appropriate value(s) for whatever they were asked to do. We will see some examples of how it can be used later in this topic.

It is also the value that is returned by a function that doesn't explicitly `return` anything else. So if your function does not explicitly `return` anything, then it actually returns `None`.

Try the following in an interactive Python 3 session:

```
>>> None
>>>
>>> type(None)
<class 'NoneType'>
>>> bool(None)
False
```

Doing "nothing": `pass`

The `pass` statement is a command that, when executed, does nothing (this is known as a "**null operation**" or a "**no-op**"). Unlike a comment, which is entirely ignored by Python (as far as Python is concerned, comments don't exist, save for the purpose of working out the line number of a line of code in your program), Python treats the `pass` statement as a real command, just one that does nothing.

This may seem like a useless command for Python to have, but it is quite useful as a "placeholder" for some code you haven't gotten around to writing yet, for instance, a function which you've defined but whose code you have not yet written. Function definitions must contain at least one line of code (and remember comments don't count!). So if you are in the process of writing a program and want to have some function definition lines in your code to remind you you need to eventually write those functions, you can use the `pass` statement as the function body until you actually get around to writing the function:

```
def integrate(function, upper, lower, tolerance):
    pass
```

`pass` can also be extremely useful when handling errors. Python always quits your program when it encounters an error. So if there are certain circumstances in which an error should be ignored, you could write your own exception handler that would use the `pass` statement to do nothing under such circumstances, thus allowing your program to ignore the error and continue.

Try the following in an interactive Python 3 session:

```
>>> pass
>>>
```

Controlling loops: `break`

The `break` statement causes Python to stop executing whatever loop it might be executing and jump to the first statement immediately after that loop. If you have nested loops (i.e. loops within loops, e.g. a `for` loop within a `while` loop), the `break` statement will terminate the loop that in which it occurs, transferring control to the next statement in any containing loop. Examine the code snippets below to see examples of how the `break` statement might be used:

```
while x % 2 == 0:
    print(x, 'still even')
    x = x // 2
    break

for prime in primes:
    print(prime)
    if prime > 7:
        break
```

The `break` statement is particularly useful if something goes wrong during your loop and you need to stop executing the loop, as might be the case if an exception occurred while your loop was running.

Controlling loops: `continue`

The `continue` statement causes Python to stop executing the current iteration of whatever loop it might be executing and start on the next iteration of that loop. If you have nested loops (i.e. loops within loops, e.g. a `for` loop within a `while` loop), the `continue` statement will start the next iteration of the loop that in which it occurs. Examine the code snippets below to see examples of how the `continue` statement might be used:

```
while x % 2 == 0:
    if x == 4:
       continue
    x = x // 2

for prime in primes:
    print(prime)
    continue
    print('This line is never executed.')
```

What are the different types of exception?

So, having gotten that useful miscellany out of the way, you may be wondering about the different types of exceptions Python has. How many are there? Under what circumstances might they occur? How can we find out about them?

Python has a variety of different types of exception that are part of the language. These are called "**built-in exceptions**" and are listed in the Python documentation:
`http://docs.python.org/py3k/library/exceptions.html`

Sadly this documentation is not as detailed as one might wish, but it provides a general idea of the circumstances under which each particular exception might occur. A close reading of the documentation sometimes also reveals what information is provided with the exception. It is often more useful to write some Python code that will naturally cause the exception to occur and inspect the output produced by Python's default exception handler.

Modules can also define their own types of exception, and these should be described in the modules documentation. It is also possible to define your own types of exception (called "**user-defined exceptions**"), but that is far beyond the scope of this course; see the Python Tutorial for further information:
`http://docs.python.org/py3k/tutorial/errors.html#tut-userexceptions`

In these notes we'll mainly be concerned with Python's `ValueError` and `IOError` exceptions. There is nothing particularly special about these exceptions; they just happen to be convenient for our purposes.

A `ValueError` occurs when an operation or function receives an argument that has the right type but an inappropriate value, and there are no other exceptions that more precisely define the error. The information provided with this exception is typically a single string containing a description of the error.

An `IOError` occurs when an "**input/output (I/O) operation**", such as operations involving files or disks (e.g. opening a file, reading a file, writing to a file, closing a file, etc.), fails. Typically 2 or 3 pieces of information will be provided with this exception: an error number (which is an integer), an error message describing the error (a string), and, in some cases, a filename (as a string). Typically the error number is actually provided by the operating system to explain what went wrong, and is not terribly useful unless you are familiar with the low-level functions that perform your operating system's I/O operations. The error message is more helpful, and we will make use of it in our examples later on. The filename, if provided, will be the name of the file that the system was trying to access when the error occurred.

Generating exceptions: `raise`

Before we look at how we can handle exceptions, it will be useful to know how we can

You can cause an exception by using the `raise` statement (this is called "raising an exception"). This can be useful for testing your exception handler, and also if you want your functions to cause certain exceptions to happen under circumstances; for instance, you might want your function to raise a `ValueError` exception if it was given an invalid value as input. You use `raise` like this:

```
raise ExceptionType(exceptioninformation)
```

where *`ExceptionType`* is the type of exception you want to cause (e.g. a `NameError` or `ValueError`), and *`exceptioninformation`* is the information about the exception that you want to provide (e.g. a string saying the supplied variable doesn't exist, as we saw earlier when we ran `wrong1.py`). The information you provide is typically a string explaining what went wrong, but exactly what it is depends on the type of exception. Some exception take several pieces of information, in which case they would be supplied as separate arguments separated by commas (see the example on the next page).

Try the following in an interactive Python 3 session:

Make sure that you are using **Python 3** and ***not*** Python 2. On many systems Python 2 is the default, so if you just type "`python`" at a Unix prompt, you will probably get Python 2. To make sure you are using Python 3 type "**`python3`**" at the Unix prompt:

```
$ python3
```

```
>>> raise ValueError('Invalid value provided')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Invalid value provided
>>>
>>> raise IOError(2,'File not found')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] File not found
>>> raise IOError(2,'File not found','rubbish.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] File not found: 'rubbish.txt'
```

Exception handling: `try...except`

Exception handling in Python is done using the `try…except` construct. Essentially, whenever we think that some commands may fail, we create an exception handler for the errors we expect using the `try…except` construct. If we don't specify a specific exception (error) to handle, then our exception handler is used for any errors that occur while executing those commands. If our exception handler only handles certain exceptions and our script produces an exception that our handler wasn't written to deal with, Python's default exception handler will handle that error for us.

The `try…except` construct works like this:

```
try:
    Python commands

except:
    Exception handler
```

Python attempts to run the commands in the `try` block (the indented block of commands that immediately follows the "`try:`") and, if an exception occurs it stops running the `try` block and immediately executes the `except` block (the indented block of commands that immediately follows the "`except:`"). If no exceptions occur then the commands in the `except` block are ignored. The commands in the `except` block are the *exception handler* for the `try` block.

Often we only want our exception handler to handle a specific type of error and let something else (often Python's default exception handler) handle any other types of error. We can do this by declaring that the except block only handles a particular type of exception like this:

```
try:
    Python commands

except ExceptionType:
    Exception handler
```

…where *ExceptionType* is the type of exception that we want our exception handler to deal with.

This will become clearer as we work through an example.

We'll start off with a simple exception handler (that only handles `IOError`'s) and gradually make our exception handling more sophisticated.

First, consider the function below (which is in the `utils.py` file):

```
def file2dict(filename):

    dictionary = {}

    data = open(filename, 'r')

    for line in data:
        [ key, value ] = line.split()
        dictionary[key] = value

    data.close()

    return dictionary
```

utils.py

This function reads opens a file of key/value pairs, reads the file a line at a time and creates a dictionary from each key/value pair (which it treats as strings). It then closes the file and returns the dictionary. (If you do not understand any of the Python in this function, please ask one of the demonstrators now.)

One of the problems with this function is that if there are any problems with the file it is trying to read it falls over in a heap. So let's start by getting the function to handle any `IOError`'s that might occur when it is running.

The first thing to do is to identify which parts of the function involve file I/O operations. Clearly opening the file (with `open()`) is a file I/O operation. Reading each line from the file using our `for` loop is also a file I/O operation, so we need to include that as well. Finally closing the file (with the `close()` method) is also a file I/O operation. So all those bits of Python will go in our try block.

For our exception handler, we will declare that it only handles IOError's and make it print a message saying something went wrong, close the file, and then quit. Edit the `file2dict()` function to make the changes shown in **bold** below (and make sure you save the file when you've finished editing it):

```
def file2dict(filename):
    import sys
    dictionary = {}
    try:
        data = open(filename, 'r')
        for line in data:
            [ key, value ] = line.split()
            dictionary[key] = value
        data.close()
    except IOError:
        print('Problem with file', filename)
```

```
        print('Aborting')
        data.close()
        sys.exit(1)
    return dictionary
```

utils.py

Make sure you add the extra indentation to the lines in the `try` block.

Recall that the `exit()` function that lives in the `sys` module is the proper function to use to quit a script. Since this function lives in the `sys` module, we need to put "`import sys`" at the start of the `file2dict()` function. If you give this function an integer as input, then that integer will be the "**exit status**"[2] of the program. If you don't supply an integer, then the function behaves as though it had been called with the integer 0 (i.e. the exit status will be 0). By convention, the exit status of a program or script should be 0 if it completed successfully, and non-zero if it didn't. If we get to the `except` block then there's been a problem, so we should exit with a non-zero exit status.

Exercise 1

Make the changes to the `file2dict()` function shown in bold above. **Remember** to add the extra indentation to the lines in the `try` block. **Make sure** you save the `utils.py` file once you have made these changes.

Now try the following in an interactive Python 3 session:

```
>>> import utils
>>> mydict = utils.file2dict('output')
Problem with file output
Aborting
Traceback (most recent call last):
  File "utils.py", line 7, in file2dict
    data = open(filename, 'r')
IOError: [Errno 2] No such file or directory: 'output'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "utils.py", line 15, in file2dict
    data.close()
UnboundLocalError: local variable 'data' referenced before assignment
>>>
```

Well, that didn't work quite as expected.

2 If you are unfamiliar with the concept of the exit status of a program (also called exit code, return code, return status, error code, error status, errorlevel or error level), the following Wikipedia article gives a bit more detail:
`http://en.wikipedia.org/wiki/Exit_status`

Observe that there was an error in our `except` block, and so Python's default exception handler took over to handle it (well, an `except` block can hardly be expected to handle an error within itself).

The problem here is that, since the file doesn't exist, the variable `data` never gets assigned a value (because `open()` fails), and so it doesn't exist when we reference it to `close()` the file in our `except` block. The easiest way of fixing this problem is to assign `data` a value before we enter the `try` block. What value should we use? Well, it doesn't really matter so long as it is not another file object. We'll use the value `None` that we met earlier, as it is a Python special value designed for such purposes. (Many Python functions use it to mean that there are no appropriate values for whatever they were asked to do.)

So let's fix this error and try again. Make the changes shown in **bold** below to the `file2dict()` function (and remember to save the file when you've finished editing it):

```
def file2dict(filename):
    import sys
    dictionary = {}
    data = None
    try:
        data = open(filename, 'r')
        for line in data:
            [ key, value ] = line.split()
            dictionary[key] = value
        data.close()
    except IOError:
        print('Problem with file', filename)
        print('Aborting')
        if type(data) != type(None):
            data.close()
        sys.exit(1)
    return dictionary
```

utils.py

Make sure you add the extra indentation to the `data.close()` line in the `except` block.

Exercise 2

Make the changes to the `file2dict()` function shown in bold above. **Remember** to add the extra indentation to the `data.close()` line in the `except` block. **Make sure** you save the `utils.py` file once you have made these changes.

Now try the following in an interactive Python 3 session:

```
>>> import utils
>>> mydict = utils.file2dict('output')
Problem with file output
Aborting
```

```
$
```

Yay, it works: our first exception handler!

Getting information about the exception

Now, our error message is a little vague (“Problem with file”) and it would be nice if we could be a bit more specific. Recall that Python’s default exception handler not only tells you the type of error but gives you some detail in the form of an “error message”. How can we get access to this?

As well as specifying what type of exception we want to handle, we can also attach a variable of our choice to the exception, which will enable us to retrieve information from about the exception. We do this by specifying “ `as` ” followed by the name of our chosen variable after the type of exception we want to handle. So a more complete syntax for the `try`…`except` construct is:

```
try:
    Python commands


except ExceptionType as errorvariable:
    Exception handler
```

…where *`ExceptionType`* is the type of exception that we want our exception handler to deal with and *`errorvariable`* is the optional variable that allows us to refer to the exception in our exception handler.

There may actually be several pieces of information supplied about the error: the error number, the error message, etc. which are contained in different attributes of the exception. For `IOError`’s, the most useful piece of information, a string describing the errror, is held in the `strerror` attribute of the exception. We access this by using the *`errorvariable`* we’ve defined: *`errorvariable`*`.strerror`. Also, if we use the `str()` function on the *`errorvariable`*, this usually gives a string that contains a combination of all the separate pieces of information about the error.

Modify the `file2dict()` function as show below (changes in **bold**). Make sure you save the file after you have finished editing it.

```
def file2dict(filename):
    import sys
    dictionary = {}
    data = None
    try:
        data = open(filename, 'r')
        for line in data:
            [ key, value ] = line.split()
            dictionary[key] = value
        data.close()
    except IOError as error:
        print('Problem with file', filename+':', error.strerror)
        print('Aborting')
        if type(data) != type(None):
            data.close()
        sys.exit(1)
```

```
    return dictionary
```

utils.py

Exercise 3

Make the changes to the `file2dict()` function shown in bold above. **Make sure** you save the `utils.py` file once you have made these changes.

Now try the following in an interactive Python 3 session:

```
>>> import utils
>>> mydict = utils.file2dict('output')
Problem with file output: No such file or directory
Aborting
$
```

Now we have a much more informative exception handler.

What else can go wrong with our function?

There's another problem we might encounter with our function. If the file is in the wrong format (e.g. either more or fewer than two values per line, or if it uses something other than spaces to separate the values) then our function will fail. What error will we get in such situations?

Let's examine our function again. The variable `line` gets set to the next line of data in the file we are reading. We then use `split()` on this variable, which returns a list of words in the line we've just read, i.e. a list of collections of characters separated by whitespace (spaces, tabs, etc). We then "unpack" this list into two variables, `key` and `value`. If the list does not contain exactly two items (i.e. if `line` does not contain exactly two "words"), then Python will complain.

We can simulate such errors by setting `line` to a string that contains more than two words, and one that contains fewer than two words, and then using `split()` and trying to unpack the resulting list, exactly as we do in our function. If we do this, we see that the exception that Python raises (in both error cases) is a `ValueError` exception:

```
>>> line = "Too many values"
>>> [ key, value ] = line.split()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
>>> line = "notenough!"
>>> [ key, value ] = line.split()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 1 value to unpack
```

So how can we modify our function to handle this exception?

We first identify the part of the function where the exception we intend to handle (the `ValueError` exception) occurs.

Then we surround this part of the function with a `try…except` construct.

We have to decide what we want to do on failure. The sensible thing is to tell the user that there is

something wrong with the file. We'll use `print()` for that.

We have two options at this point: we can either halt the Python program using the `exit()` function from the `sys` module, as we have done before, or we can carry on.

Arguably, a file that is in the wrong format is not so serious an error that we should automatically force the program to quit. However, we don't want to return a garbled or incomplete dictionary either. The sensible thing would be to return something that couldn't possibly be a valid dictionary (the special `None` value, for instance). The program or user who called our function can then look at what they got back and, if it is not a dictionary, then they can decide whether to quit or do something else (for example, they could try reading from a different file).

We then need to stop processing the file (well, we could continue, but there would be no point reading any more of the file as at this point we've decided to return `None` anyway). In this function we read and process data from the file in a `for` loop, so we need a way of telling Python to quit the `for` loop, which we covered earlier.

So we should have everything we need to do this, which leads us to the following exercise:

Exercise 4

Modify the `file2dict()` function to handle the `ValueError` exception as described above. Test that your exception handler works by trying the function on the `bad-format1.txt` and `bad-format2.txt` files.

When writing exception handlers, a common mistake is to put more parts of the program into our `try` block than is necessary (or wise). In particular, there is a strong temptation to just put *everything* into the `try` block. This is a **very bad** idea. You should only put lines in your `try` block if you have thought carefully about whether or not you (rather than Python) want to handle any errors that occur running those lines. Your `try` blocks should contain *only* exactly as many lines as they need so that you can handle the particular error(s) you want to deal with *where you expect them to occur*. Leave the rest of the exception handling to Python.