This chapter is taken from the book A Primer on Scientific Programming with Python (http://www.springer.com/gp/book/9783662498866) by H. P. Langtangen, 5th edition, Springer, 2016.

Handling errors

Suppose we forget to provide a command-line argument to the `c2f_cml.py` program from the section Providing input on the command line (._input-readable002.html#sec:input:cml):

```
c2f_cml.py
Traceback (most recent call last):
  File "c2f_cml.py", line 2, in ?
    C = float(sys.argv[1])
IndexError: list index out of range
```

Python aborts the program and shows an error message containing the line where the error occurred, the type of the error (`IndexError`), and a quick explanation of what the error is. From this information we deduce that the index `1` is out of range. Because there are no command-line arguments in this case, `sys.argv` has only one element, namely the program name. The only valid index is then `0`.

For an experienced Python programmer this error message will normally be clear enough to indicate what is wrong. For others it would be very helpful if wrong usage could be detected by our program and a description of correct operation could be printed. The question is how to detect the error inside the program.

The problem in our sample execution is that `sys.argv` does not contain two elements (the program name, as always, plus one command-line argument). We can therefore test on the length of `sys.argv` to detect wrong usage: if `len(sys.argv)` is less than 2, the user failed to provide information on the `C` value. The new version of the program, `c2f_cml_if.py` , starts with this `if` test:

```
if len(sys.argv) < 2:
    print 'You failed to provide Celsius degrees as input '\
          'on the command line!'
    sys.exit(1)  # abort because of error
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

We use the `sys.exit` function to abort the program. Any argument different from zero signifies that the program was aborted due to an error, but the precise value of the argument does not matter so here we simply choose it to be `1` . If no errors are found, but we still want to abort the program, `sys.exit(0)` is used.

A more modern and flexible way of handling potential errors in a program is to *try* to execute some statements, and if something goes wrong, the program can detect this and jump to a set of statements that handle the erroneous situation as desired. The relevant program construction reads

```
try:
    <statements>
except:
    <statements>
```

If something goes wrong when executing the statements in the `try` block, Python raises what is known as an *exception*. The execution jumps directly to the `except` block whose statements can provide a remedy for the error. The next section explains the `try-except` construction in more detail through examples.

Exception handling

To clarify the idea of exception handling, let us use a `try-except` block to handle the potential problem arising when our Celsius-Fahrenheit conversion program lacks a command-line argument:

```
import sys
try:
    C = float(sys.argv[1])
except:
    print 'You failed to provide Celsius degrees as input '\
          'on the command line!'
    sys.exit(1)  # abort
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

The program is stored in the file `c2f_cml_except1.py` (http://tinyurl.com/pwyasaa/input/c2f_cml_except1.py). If the command-line argument is missing, the indexing `sys.argv[1]`, which has an invalid index `1`, *raises an exception*. This means that the program jumps directly to the `except` block, implying that `float` is not called, and `C` is not initialized with a value. In the except block, the programmer can retrieve information about the exception and perform statements to recover from the error. In our example, we know what the error can be, and therefore we just print a message and abort the program.

Suppose the user provides a command-line argument. Now, the `try` block is executed successfully, and the program neglects the `except` block and continues with the Fahrenheit conversion. We can try out the last program in two cases:

```
c2f_cml_except1.py
You failed to provide Celsius degrees as input on the command line!

c2f_cml_except1.py 21
21C is 69.8F
```

In the first case, the illegal index in `sys.argv[1]` causes an exception to be raised, and we perform the steps in the `except` block. In the second case, the `try` block executes successfully, so we jump over the `except` block and continue with the computations and the printout of results.

For a user of the program, it does not matter if the programmer applies an `if` test or exception handling to recover from a missing command-line argument. Nevertheless, exception handling is considered a better programming solution because it allows more advanced ways to abort or continue the execution. Therefore, we adopt exception handling as our standard way of dealing with errors in the rest of this document.

Testing for a specific exception

Consider the assignment

```
C = float(sys.argv[1])
```

There are two typical errors associated with this statement: i) `sys.argv[1]` is illegal indexing because no command-line arguments are provided, and ii) the content in the string `sys.argv[1]` is not a pure number that can be converted to a `float` object. Python detects both these errors and raises an `IndexError` exception in the first case and a `ValueError` in the second. In the program above, we jump to the `except` block and issue the same message regardless of what went wrong in the `try` block. For example, when we indeed provide a command-line argument, but write it on an illegal form (`21C`), the program jumps to the `except` block and prints a misleading message:

```
c2f_cml_except1.py 21C
You failed to provide Celsius degrees as input on the command line!
```

The solution to this problem is to branch into different `except` blocks depending on what type of exception that was raised in the `try` block (program `c2f_cml_except2.py` (http://tinyurl.com/pwyasaa/input/c2f_cml_except2.py)):

```
import sys
try:
    C = float(sys.argv[1])
except IndexError:
    print 'Celsius degrees must be supplied on the command line'
    sys.exit(1)  # abort execution
except ValueError:
    print 'Celsius degrees must be a pure number, '\
          'not "%s"' % sys.argv[1]
    sys.exit(1)

F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

Now, if we fail to provide a command-line argument, an `IndexError` occurs and we tell the user to write the `C` value on the command line. On the other hand, if the `float` conversion fails, because the command-line argument has wrong syntax, a `ValueError` exception is raised and we branch into the second `except` block and explain that the form of the given number is wrong:

```
c2f_cml_except1.py 21C
Celsius degrees must be a pure number, not "21C"
```

Examples on exception types

List indices out of range lead to `IndexError` exceptions:

```
>>> data = [1.0/i for i in range(1,10)]
>>> data[9]
...
IndexError: list index out of range
```

Some programming languages (Fortran, C, C++, and Perl are examples) allow list indices outside the legal index values, and such unnoticed errors can be hard to find. Python always stops a program when an invalid index is encountered, unless you handle the exception explicitly as a programmer.

Converting a string to `float` is unsuccessful and gives a `ValueError` if the string is not a pure integer or real number:

```
>>> C = float('21 C')
...
ValueError: invalid literal for float(): 21 C
```

Trying to use a variable that is not initialized gives a `NameError` exception:

```
>>> print a
...
NameError: name 'a' is not defined
```

Division by zero raises a `ZeroDivisionError` exception:

```
>>> 3.0/0
...
ZeroDivisionError: float division
```

Writing a Python keyword illegally or performing a Python grammar error leads to a `SyntaxError` exception:

```
>>> forr d in data:
...
    forr d in data:
         ^
SyntaxError: invalid syntax
```

What if we try to multiply a string by a number?

```
>>> 'a string'*3.14
...
TypeError: can't multiply sequence by non-int of type 'float'
```

The `TypeError` exception is raised because the object types involved in the multiplication are wrong (`str` and `float`).

Digression

It might come as a surprise, but multiplication of a string and a number is legal if the number is an integer. The multiplication means that the string should be repeated the specified number of times. The same rule also applies to lists:

```
>>> '--'*10   # ten double dashes = 20 dashes
'--------------------'
>>> n = 4
>>> [1, 2, 3]*n
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> [0]*n
[0, 0, 0, 0]
```

The latter construction is handy when we want to create a list of `n` elements and later assign specific values to each element in a `for` loop.

Raising exceptions

When an error occurs in your program, you may either print a message and use `sys.exit(1)` to abort the program, or you may raise an exception. The latter task is easy. You just write `raise E(message)`, where `E` can be a known exception type in Python and `message` is a string explaining what is wrong. Most often `E` means `ValueError` if the value of some variable is illegal, or `TypeError` if the type of a variable is wrong. You can also define your own exception types. An exception can be raised from any location in a program.

Example

In the program `c2f_cml_except2.py` from the section Exception handling we show how we can test for different exceptions and abort the program. Sometimes we see that an exception may happen, but if it happens, we want a more precise error message to help the user. This can be done by raising a new exception in an `except` block and provide the desired exception type and message.

Another application of raising exceptions with tailored error messages arises when input data are invalid. The code below illustrates how to raise exceptions in various cases.

We collect the reading of `C` and handling of errors a separate function:

```
def read_C():
    try:
        C = float(sys.argv[1])
    except IndexError:
        raise IndexError\
        ('Celsius degrees must be supplied on the command line')
    except ValueError:
        raise ValueError\
        ('Celsius degrees must be a pure number, '\
        'not "%s"' % sys.argv[1])
    # C is read correctly as a number, but can have wrong value:
    if C < -273.15:
        raise ValueError('C=%g is a non-physical value!' % C)
    return C
```

There are two ways of using the `read_C` function. The simplest is to call the function,

```
C = read_C()
```

Wrong input will now lead to a raw dump of exceptions, e.g.,

```
c2f_cml_v5.py
Traceback (most recent call last):
  File "c2f_cml4.py", line 5, in ?
    raise IndexError\
IndexError: Celsius degrees must be supplied on the command line
```

New users of this program may become uncertain when getting raw output from exceptions, because words like `Traceback`, `raise`, and `IndexError` do not make much sense unless you have some experience with Python. A more user-friendly output can be obtained by calling the `read_C` function inside a `try-except` block, check for any

exception (or better: check for `IndexError` *or* `ValueError`), and write out the exception message in a more nicely formatted form. In this way, the programmer takes complete control of how the program behaves when errors are encountered:

```
try:
    C = read_C()
except Exception as e:
    print e            # exception message
    sys.exit(1)        # terminate execution
```

`Exception` is the parent name of all exceptions, and `e` is an exception object. Nice printout of the exception message follows from a straight `print e`. Instead of `Exception` we can write `(ValueError, IndexError)` to test more specifically for two exception types we can expect from the `read_C` function:

```
try:
    C = read_C()
except (ValueError, IndexError) as e:
    print e            # exception message
    sys.exit(1)        # terminate execution
```

After the `try-except` block above, we can continue with computing `F = 9*C/5 + 32` and print out `F`. The complete program is found in the file `c2f_cml.py` (http://tinyurl.com/pwyasaa/input/c2f_cml.py). We may now test the program's behavior when the input is wrong and right:

```
c2f_cml.py
Celsius degrees must be supplied on the command line

c2f_cml.py 21C
Celsius degrees must be a pure number, not "21C"

c2f_cml.py -500
C=-500 is a non-physical value!

c2f_cml.py 21
21C is 69.8F
```

This program deals with wrong input, writes an informative message, and terminates the execution without annoying behavior.

Scattered `if` tests with `sys.exit` calls are considered a bad programming style compared to the use of nested exception handling as illustrated above. You should abort execution in the main program only, not inside functions. The reason is that the functions can be re-used in other occasions where the error can be dealt with differently. For instance, one may avoid abortion by using some suitable default data.

The programming style illustrated above is considered the best way of dealing with errors, so we suggest that you hereafter apply exceptions for handling potential errors in the programs you make, simply because this is what experienced programmers expect from your codes.

← Prev (._input-readable006.html) Next → (._input-readable008.html)