Unit 10: Error handling

Richard Foltyn

University of Glasgow

September 10, 2022

Contents

1 Error handling

In this unit we will briefly look at error handling in Python. The Python approach to error handling is "to ask for forgiveness rather than for permission." This means that when writing Python code, we frequently don't check whether some data satisfies certain requirements, but we instead attempt to clean up once something does not work as expected.

1.1 Exceptions

If something goes wrong in a function, we in principle have two options to communicate the error to the caller:

1. We can return some special value (a status code or error flag) that signals when something fails.

   This approach is quite inelegant, since error codes can overlap with the actual result a function would return in the absence of error. For this reasons, functions need to implement two different return values and reserve one for the status code.

   In Python, this could look like this:

   ```
   def func(x):
       # process x
       # Two return values: actual result and error flag
       return result, flag
   ```

2. We can use so-called exceptions for error handling. This is the approach taken by almost all modern languages such as Java, C++ and also Python (see here for the official documentation on error and exception handling).

   Exceptions provide means to communicate errors that are completely independent of regular return values. Furthermore, exceptions propagate along the entire call stack: If we call `func1()`, which in turn calls `func2()`, and an error occurs in `func2()`, there is no need to handle this error in `func1()`: the exception will automatically be propagated to the caller of `func1()`.

1.1.1 Common exceptions

We have already encountered numerous exceptions throughout this course, but so far we did not know how to handle them other than fixing the code that produced the exception.

There are numerous exceptions in Python, see here for a list of built-in ones. We provide a few examples of exceptions that you are most likely to encounter below.

Examples:

Trying to access an element in a collection outside of the permissible ranger produces an `IndexError`.

[1]:
```
# access to out-of-bounds index in a collection
items = 1, 2, 3
items[5]
```

```
IndexError: tuple index out of range
```

Retrieving a non-existent key in a dictionary raises another type of exception, a `KeyError`.

[2]:
```
# Access non-existant dictionary key
dct = {'language': 'Python', 'version': 3.8}
dct['course']
```

```
KeyError: 'course'
```

Mistakenly trying to access a non-existent attribute will trigger an `AttributeError`:

[3]:
```
value = 1.0
value.shape
```

```
AttributeError: 'float' object has no attribute 'shape'
```

When we try to apply an operation to data that does not support that particular operation, we get a `TypeError`:

[4]:
```
items = 1, 2, 3
items + 1
```

```
TypeError: can only concatenate tuple (not "int") to tuple
```

Division by zero also triggers an exception of type `ZeroDivisionError`:

[5]:
```
1/0
```

```
ZeroDivisionError: division by zero
```

Attempting to import a module or symbol from within a module that does not exist raises an `ImportError`:

[6]:
```
from numpy import function_that_does_not_exist
```

```
ImportError: cannot import name 'function_that_does_not_exist' from 'numpy' (/hom /
 ↪richard/.conda/envs/py3-default/lib/python3.10/site-packages/numpy/__init__.py)
```

Performing an operation on arrays of non-conforming shape produces a `ValueError`:

[7]:
```
import numpy as np

a = np.arange(3)
b = np.arange(2)
a + b
```

```
ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

Trying to open a non-existing file will raise an `FileNotFoundError`.

[8]:
```
open('file_does_not_exists.txt', 'rt')
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'file_does_not_exists.txt
```

In other cases, for example when using NumPy's `loadtxt`, trying to open a non-existing file will instead raise a `OSError`.

[9]:
```
import numpy as np

data = np.loadtxt('path/to/nonexisting/file.txt')
```

```
OSError: path/to/nonexisting/file.txt not found.
```

1.2 Handling errors

As you just saw, there are numerous types of exceptions raised by Python libraries we use every day. We can handle these in two ways:

1. Avoid errors before they arise.
2. Catch exceptions once they arise in special exception-handling blocks.

1.2.1 Avoiding errors

We could have avoided almost all of the above exception if we had surrounded them with `if` statements and checked whether an operation could actually be performed.

This, however, is usually not the way we write Python code, unless we are implementing library functions that need to work in situations over which we have little control. We certainly don't want to clutter "regular" code with `if` statements everywhere. There are other ways to avoid errors.

Examples:

Returning to the dictionary example, we could write something like this:

[10]:
```
# Access non-existant dictionary key
dct = {'language': 'Python', 'version': 3.10}
if 'course' in dct:
    print(dct['course'])
```

However, if we have a default value that should be used whenever a key is not present, we can more elegantly use the `get()` method which accepts a default value. No `if`'s needed:

[11]:
```
# access non-existing key
dct.get('course', 'Default programming language')
```

[11]:
```
'Default programming language'
```

If a key does exist, the default will of course be ignored:

[12]:
```
# access existing key
dct.get('language', 'Default programming language')
```

[12]:
```
'Python'
```

Another easily avoidable exception is the `IndexError`. There is hardly ever a reason to attempt retrieving elements at arbitrary indices. Usually, we first check the size of a collection:

[13]:
```
items = 1, 2, 3

# Assume idx was passed as an argument to a function
# so we need to handle unforeseen cases
idx = 1000

# Enforce valid upper bound in case the index is
# out of bounds.
items[min(idx, len(items) - 1)]
```

[13]:
```
3
```

When operating on NumPy arrays, we frequently have to retrieve their dimensions first, so there is no risk of accessing an invalid position:

[14]:
```
import numpy as np

mat = np.arange(6).reshape(2, 3)

# Retrieve array dimensions
nrow, ncol = mat.shape

# Loop makes sure to never step out of bounds
for i in range(nrow):
    for j in range(ncol):
        print(mat[i, j])
```

```
0
1
2
3
4
5
```

There are also many helper routines that allow for "robust" programming. Imagine we want a function that returns the element at position `[0, 0]`:

[15]:
```
def get_elem(x):
    return x[0,0]
```

Calling this on a matrix works as intended:

[16]:
```
get_elem(np.ones((2,2)))
```

[16]:
```
1.0
```

But what if we pass a nested list or tuple?

[17]:
```
get_elem([[1,2], [3,4]])
```

```
TypeError: list indices must be integers or slices, not tuple
```

With very little effort, we can make this function more robust by using `np.atleast_2d()` which ensures that its result is at least a 2-dimensional NumPy array (it returns higher-dimensional arrays unmodified):

[18]:
```
import numpy as np

def get_elem(x):
    x = np.atleast_2d(x)
    return x[0,0]
```

[19]:
```
get_elem([[1,2], [3,4]])        # Now works on nested lists
```

[19]:
```
1
```

This function suddenly becomes much more flexible, maybe too flexible since it works on all sorts of arguments:

[20]:
```
get_elem([1, 2])        # simple list
get_elem(1.0)           # scalar
```

[20]:
```
1.0
```

NumPy also implements `np.atleast_1d()` and `np.atleast_3d()` which serve the same purpose, but return 1-dimensional or 3-dimensional arrays instead.

1.2.2 Raising exceptions

There are situations when we explicitly want to ensure that some condition is met, instead of letting the code fail somewhere down the line. This is particularly important when we write library functions that might be called from many different contexts or by many different users. Raising an exception with a clear error message is beneficial in such situations.

To illustrate the benefit of clear error messages, consider the following (highly artificial) example:

[21]:
```
def get_row(mat, i):
    # restrict to valid row indices
    irow = min(mat.shape[0] - 1, max(0, i))

    # return row
    row = mat[irow]
    return row
```

We define the function `get_row` that returns the `i`-th row of a matrix. The function ensures that the row index is within the admissible range for the given array.

Let's call this function as follows:

[22]:
```
import numpy as np
mat = np.arange(6).reshape((3, 2))
get_row(mat, 1.0)
```

```
IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis (`None`)
↪and integer or boolean arrays are valid indices
```

This raises an `IndexError`, notifying the user that the statement `row = mat[irow]` was problematic. However, the caller does not know what `irow` is since this is not the name of the original argument. In

the worst case, the user would have to inspect the implementation of `get_row()` to figure out what is wrong.

How can we rectify this situation? We cannot prevent someone from calling this function with an inadmissible value, but we can raise an exception once such a value is encountered.

We raise exceptions using the `raise` statement which is followed by an exception:

[23]:
```
def get_row(mat, i):
    # Check whether i is an integer
    if not isinstance(i, int):
        msg = f'Integer argument required, received {i}'
        raise ValueError(msg)
    # restrict to valid row indices
    irow = min(mat.shape[0] - 1, max(0, i))

    # return row
    row = mat[irow]
    return row
```

To check whether `i` is of integer type, we use the `isinstance()` function.

The convention is to raise a `ValueError` when a function argument does not satisfy some requirement. We can optionally pass an error message, as in the example above. There is no need or possibility to add an explicit `return` statement: as soon as an exception is raised, any remaining code is skipped. We will examine the details below.

[24]:
```
get_row(mat, 1)      # Call with integer argument; works as intended.
```

[24]:
```
array([2, 3])
```

[25]:
```
get_row(mat, 1.0)    # Call with float argument; raises exception
```

```
ValueError: Integer argument required, received 1.0
```

As you see, an exception is raised and a clear error message is returned to the caller.

1.2.3 Catching exceptions

If we are unable or unwilling to take measures to avoid an error, we have to deal with the resulting exception, should one occur. If we fail to do so, the entire program will be terminated.

We handle exceptions using the `try` statement (we sometimes say we "catch" exceptions, which is the keyword used in some other programming languages):

- The code that potentially raises an exception is placed in the `try` clause.
- If an error occurs, control is immediately passed on to the `except` clause and any remaining statements in the `try` clause are skipped.
- The `except` clause takes care of handling the exception, should one occur. If no exception is raised, the `except` clause is never executed.

Examples:

Say we need to process an integer value but are unsure about the data type of the input; calling `int()` might therefore work, or it might not:

[26]:
```
x = 1.2345
int(x)          # Works, float is truncated to integer
```

[26]:
```
1
```

[27]:
```
x = 'abc'
int(x)          # Does not work
```

```
ValueError: invalid literal for int() with base 10: 'abc'
```

Calling `int()` with a string such as `'abc'` which cannot be interpreted as an integer will raise a `ValueError`. We could handle such a situation as follows:

[28]:
```
x = 'abc'

try:
    i = int(x)
    print('Conversion to integer works!')
except ValueError:
    print(f'{x} cannot be converted to an integer')
```

```
abc cannot be converted to an integer
```

We see that the execution of the `try` clause terminates as soon as the exception is raised, so the `print()` function is never called. Instead, execution is passed on to the `except` clause which matches the exception type.

We can have multiple `except` clauses covering all sorts of exceptions:

[29]:
```
def func(x):
    try:
        i = int(x)
        print('Conversion to integer works!')
        # Return some value
        return 10/i
    except ValueError:
        print(f'{x} cannot be converted to an integer')
    except ZeroDivisionError:
        print('Division by zero')
    except:
        print('Other exception type occured')
```

[30]:
```
func('abc')          # ValueError: cannot convert integer
```

```
abc cannot be converted to an integer
```

[31]:
```
func(0)              # ZeroDivisionError
```

```
Conversion to integer works!
Division by zero
```

An `except` clause without an exception type catches any exceptions which do not match any preceding `except` clause. For example, this code raises a `TypeError` which is not specifically handled:

[32]:
```
func([1, 2, 3])      # TypeError, caught by default clause
```

```
Other exception type occured
```

If there is no default `except` clause and an unhandled exception occurs, it will be propagated back to the caller as if no error handling was present at all:

[33]:
```
# Define func to only handle ValueError
def func(x):
    try:
        i = int(x)
        print('Conversion to integer works!')
        # Return some value
```

```
        return 10/i
    except ValueError:
        print(f'{x} cannot be converted to an integer')
```

[34]:
```
func(0)          # Raises ZeroDivisionError, which is passed to caller
```

```
Conversion to integer works!
```

```
ZeroDivisionError: division by zero
```

This even works across multiple levels of the call stack:

[35]:
```
# inner function converts to integer
def inner(x):
    i = int(x)
    return i

# outer function divides by integer value
def outer(x):
    i = inner(x)
    return 10 /i
```

[36]:
```
outer('abc')             # ValueError raised in inner()
```

```
ValueError: invalid literal for int() with base 10: 'abc'
```

Here we call `outer()`, which in turn calls `inner()`, passing on its argument. Conversion to an integer fails in `inner()`, but since `outer()` does not handle this exception, it is automatically passed on the the original call site.

1.3 Exercises

1.3.1 Exercise 1: Sign function

Revisit the sign function you implemented in Unit 4, Exercise 1. To refresh your memory, the suggested solution looks as follows:

[37]:
```
import numpy as np

def sign(x):
    if x < 0.0:
        return -1.0
    elif x == 0.0:
        return 0.0
    elif x > 0.0:
        return 1.0
    else:
        # Argument is not a proper numerical value, return NaN
        # (NaN = Not a Number)
        return np.nan
```

This implementation is not very robust, as it returns all sorts of exceptions when passed non-numeric arguments:

[38]:
```
sign('abc')              # pass in string
```

```
TypeError: '<' not supported between instances of 'str' and 'float'
```

[39]:
```
sign(np.array([1, 2, 3]))           # Pass in NumPy array
```

```
ValueError: The truth value of an array with more than one element is ambiguous.␣
↪Use a.any() or a.all()
```

Modify the `sign()` function such that it only accepts built-in numerical Python types (integers, floats) and raises a `ValueError` in all other cases

1.3.2 Exercise 2: Factorials

Consider the `factorial()` function you wrote in Unit 4, Exercise 4:

[40]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        # Use recursion to compute factorial
        return n * factorial(n-1)
```

This implementation is also not very robust to nonsensical arguments, for example:

[41]:
```
factorial(1.123)
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

Modify this function such that it only accepts *numerical* arguments that are either integers, or can be interpreted as integers without loss of data, such as a float `1.0` or a scalar array `np.array(1.0)`.

The function should raise a `ValueError` for all other inputs.

1.3.3 Exercise 3: Bisection

Recall the `bisect()` function from Unit 4, Exercise 5:

[42]:
```
def bisect(f, a, b, tol=1.0e-6, xtol=1.0e-6, maxiter=100):

    for iteration in range(maxiter):
        # Compute candidate value as midpoint between a and b
        mid = (a + b) / 2.0
        if abs(b-a) < xtol:
            # Remaining interval is too small
            break

        fmid = f(mid)

        if abs(fmid) < tol:
            # function value is close enough to zero
            break

        print(f'Iteration {iteration}: f(mid) = {fmid:.4e}')
        if fmid*f(b) > 0.0:
            # f(mid) and f(b) have the same sign, update upper bound b
            print(f'  Updating upper bound to {mid:.8f}')
```

```
            b = mid
        else:
            # f(mid) and f(a) have the same sign, or at least one of
            # them is zero.
            print(f'  Updating lower bound to {mid:.8f}')
            a = mid

    return mid
```

This function accepts quite a few arguments, but we never implemented any input validation. Add the following input checks at the top of the function and raise a `ValueError` if any of them is violated:

1. Check that `f(a)` and `f(b)` are of opposite sign, a precondition for the bisection algorithm to work.
2. Check that `tol` and `xtol` are positive and can be interpreted as floating-point numbers.
3. Check that `maxiter` is positive and can be interpreted as an integer.

1.4 Solutions

1.4.1 Solution for exercise 1

We can use the built-in `float()` function to determine whether something can be represented as a floating-point number.

We use only the default `except` clause without any type specification as the code in the `try` clause raises several types of exceptions, depending on the input argument.

[43]:
```
import numpy as np

def sign(x):
    try:
        # Convert to float, which is more generic than int
        x = float(x)
    except:
        # The above statement raises at least two types
        # of exceptions: ValueError and TypeError
        raise ValueError('Numerical argument required!')

    if x < 0.0:
        return -1.0
    elif x == 0.0:
        return 0.0
    elif x > 0.0:
        return 1.0
    else:
        # Argument is not a proper numerical value, return NaN
        # (NaN = Not a Number)
        return np.nan
```

[44]:
```
sign(123)           # integer argument
```

[44]:
```
1.0
```

[45]:
```
sign('abc')         # string argument
```

```
ValueError: could not convert string to float: 'abc'


During handling of the above exception, another exception occurred:
```

```
ValueError: Numerical argument required!
```

[46]:
```
sign(np.array([1, 2, 3]))        # NumPy array argument
```

```
TypeError: only size-1 arrays can be converted to Python scalars


During handling of the above exception, another exception occurred:

ValueError: Numerical argument required!
```

1.4.2 Solution for exercise 2

One possible solution looks as follows:

[47]:
```
def factorial(n):
    try:
        i = int(n)
        assert i == n
    except:
        raise ValueError(f'Not an integer argument: {n}')

    if i == 0:
        return 1
    else:
        # Use recursion to compute factorial
        return i * factorial(i-1)
```

We perform input validation in two steps:

1. We use the `int()` function to convert the input to an integer. This will eliminate some inadmissible arguments such as `'abc'` or `[1, 2, 3]` but will accept others such as `'1'` or `1.1`. We want to eliminate these as well, since `'1'` is not numeric and `1.1` cannot be represented as an integer without loss of data.
2. We achieve this with the `assert` statement where we check whether `i == n`: this will only be true if `n` is numerical and does not have a fractional part.

   The `assert` statement will raise an `AssertionError` whenever a condition is not `True`, which will also be handled by the `except` clause.

[48]:
```
factorial(1)             # integer argument
```

[48]:
```
1
```

[49]:
```
factorial(1.0)           # not an integer argument, but can be
                         # represented as an integer.
```

[49]:
```
1
```

[50]:
```
factorial(1.1)           # Floating-point argument that
                         # cannot be represented as an integer
```

```
AssertionError


During handling of the above exception, another exception occurred:
```

```
ValueError: Not an integer argument: 1.1
```

[51]:
```
factorial('1')          # String argument
```

```
AssertionError


During handling of the above exception, another exception occurred:

ValueError: Not an integer argument: 1
```

[52]:
```
factorial(np.array(10.0))       # Scalar array argument
```

[52]:
```
3628800
```

1.4.3 Solution for exercise 3

We modify the function as follows:

[53]:
```
def bisect(f, a, b, tol=1.0e-6, xtol=1.0e-6, maxiter=100):

    fa = f(a)
    fb = f(b)

    if fa*fb > 0.0:
        raise ValueError(f'Not a bracketing interval [{a}, {b}]')
    try:
        tol = float(tol)
        assert tol > 0.0
    except:
        raise ValueError('Argument tol must be a positive number!')

    try:
        xtol = float(xtol)
        assert xtol > 0.0
    except:
        raise ValueError('Argument xtol must be a positive number!')

    try:
        maxiter = int(maxiter)
        assert maxiter > 0
    except:
        raise ValueError('Argument maxiter must be a positive integer!')


    for iteration in range(maxiter):
        # Compute candidate value as midpoint between a and b
        mid = (a + b) / 2.0
        if abs(b-a) < xtol:
            # Remaining interval is too small
            break

        fmid = f(mid)

        if abs(fmid) < tol:
            # function value is close enough to zero
            break
```

```
        print(f'Iteration {iteration}: f(mid) = {fmid:.4e}')
        if fmid*f(b) > 0.0:
            # f(mid) and f(b) have the same sign, update upper bound b
            print(f'  Updating upper bound to {mid:.8f}')
            b = mid
        else:
            # f(mid) and f(a) have the same sign, or at least one of
            # them is zero.
            print(f'  Updating lower bound to {mid:.8f}')
            a = mid

    return mid
```

As in the main loop of the function, we check whether two values are non-zero and have the same sign using the condition `fa * fb > 0`, in which case we have no bracketing interval and need to raise a `ValueError`.

The remaining checks are performed using the same code as in earlier exercises.

[54]:
```
# Function call with valid argument
x0 = bisect(lambda x: x**2.0 - 4.0, -3.0, 0.0, tol=1.0e-3)
```

```
Iteration 0: f(mid) = -1.7500e+00
  Updating upper bound to -1.50000000
Iteration 1: f(mid) = 1.0625e+00
  Updating lower bound to -2.25000000
Iteration 2: f(mid) = -4.8438e-01
  Updating upper bound to -1.87500000
Iteration 3: f(mid) = 2.5391e-01
  Updating lower bound to -2.06250000
Iteration 4: f(mid) = -1.2402e-01
  Updating upper bound to -1.96875000
Iteration 5: f(mid) = 6.2744e-02
  Updating lower bound to -2.01562500
Iteration 6: f(mid) = -3.1189e-02
  Updating upper bound to -1.99218750
Iteration 7: f(mid) = 1.5640e-02
  Updating lower bound to -2.00390625
Iteration 8: f(mid) = -7.8087e-03
  Updating upper bound to -1.99804688
Iteration 9: f(mid) = 3.9072e-03
  Updating lower bound to -2.00097656
Iteration 10: f(mid) = -1.9529e-03
  Updating upper bound to -1.99951172
```

[55]:
```
# Function call with f(a) and f(b) both positive
x0 = bisect(lambda x: x**2.0 - 4.0, 10.0, 20.0)
```

```
ValueError: Not a bracketing interval [10.0, 20.0]
```

[56]:
```
# Function call with invalid tolerance criterion
x0 = bisect(lambda x: x**2.0 - 4.0, -3.0, 0.0, tol=0.0)
```

```
AssertionError


During handling of the above exception, another exception occurred:

ValueError: Argument tol must be a positive number!
```