09 Error handling and testing

September 13, 2020

1 Introduction

It is bad when a program crashes without an error message. It is even worse when a program used in engineering, science or finance computes an erroneous result. A crash without an informative error message causes frustration, and it costs time because we get no information on what went wrong, and this makes it hard to fix. An erroneous result used in an engineering design could risk life. With good software engineering we can mitigate these risks.

1.1 Bugs and testing

Programs have bugs and users make mistakes. It is important to minimise the number of bugs, and where possible detect user mistakes. We do this through careful engineering of a program, error checking and extensive (automated) testing. Testing is critical in engineering programs to build confidence that a program computes the 'right' thing. Good software engineering and thorough testing would have prevented both the Patriot missile and Ariane 5 failures that were discussed in Activity 03.

Testing is not a one-off exercise. As you develop programs, it is good practice to add tests as you go. It is also good to test very small units of a program so that if a test fails you can quickly narrow down the possible causes. Modern software development uses *continuous integration*, in which a suite of tests is run every time a change is made to a code. This helps catch issues early. It also helps tremendously when more than one person is working on a program. You may add a feature today and test it, but tomorrow your colleague might make a seemingly unrelated change that breaks your feature. Continuous testing helps in this case.

1.2 Objectives

- Introduction to raising exceptions
- Use exception handling
- Creation of tests

2 Errors and exceptions

There are two types of program errors: *syntax errors* and *exceptions.*

2.1 Syntax errors

Syntax errors are when the syntax of a program does not conform to the rules of the language. These are (generally) easy to detect because the interpreter/compiler will print an error. You will have seen plenty of syntax error messages by now!

2.2 Exceptions

Exceptions are when something unexpected or anomalous occurs during the execution of a program. The syntax is correct, but something goes wrong while a program is running. Simple examples, where a well-engineered program will *raise an exception*, include:

- Attempting to divide by zero;
- Receiving negative data when only positive data is permitted, e.g. taking the log of a negative number, or a negative integer for the number students in a class;
- Unexpected integer overflows; and
- Attempting to compute the dot product between two vectors of different lengths.

These are all cases that can be tested in a program. We should check for invalid data and attempts to use our programs beyond the limits of their designs, and in these cases raise an exception with an informative message.

2.2.1 USS Yorktown Smart Ship

USS Yorktown was used for the US Navy Smart Ship program. A computer system was fitted to operate a control centre from the bridge. In 1997, a crew member entered data into the system that led to an attempted division by zero. This caused the ship's computer systems and the ship's propulsion systems to shut down. This would have been avoided with good software engineering and exception handling. You can find more at https://en.wikipedia.org/wiki/USS_Yorktown_(CG-48)#Smart_ship_testbed.

By US Navy, unknown scene camera operator - This media is available in the holdings of the National Archives and Records Administration, cataloged under the National Archives Identifier (NAID) 6393040., Public Domain, Link

3 Raising exceptions

Raising an exception is what happens when we trigger an exception. It is sometimes called *throwing an exception*. Python peforms some checks for us. For example, if we have a list of length 5:

```
colours = ["green", "blue", "yellow", "red", "orange"]
```

and we attempt to index beyond the end of the list:

```
c = colours[6]
```

we will see the error message:

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-26-322e6eedcde5> in <module>()
      1 colours = ["green", "blue", "yellow", "red", "orange"]
----> 2 c = colours[6]
IndexError: list index out of range
```

Python can check for some errors, but not all. We need to include checks in our programs to raise exceptions when something unexpected happens or invalid input is received. This reduces the likelihood of our engineering programs computing incorrect results, which could have serious consequences, and it makes it easier to debug programs. We will start with raising exceptions, and move later to how we can handle exceptions when one is raised.

Below we build on some examples from earlier notebooks to add exception handling.

3.1 Parameter validity checking

In the first notebook we looked at the gravitational potential V of a body of mass m (point mass) at a distance r from a body of mass M:

$$V = \frac{GMm}{r}$$

where G is the *gravitational constant*. This expression makes sense only for $G, M, m \geq 0$ and $r > 0$. Implementing computation of the gravitational potential as a function:

[1]:
```
def gravity_potential(G, M, m, r):
    return G*M*m/r

V = gravity_potential(6.674e-11, 1.65e12, 6.1e2, 7e3)
print(V)
```

```
9.59625857142857
```

A user could easily make a typographical error and make G, M, or m negative, or we might encounter a case where r becomes zero. In any of these cases, the computed gravitational potential would not be sensible but the above program would return a value and proceed.

Rather than return an obviously wrong result, we can guard against easy-to-detect errors by checking the validity of the arguments and raising exceptions in the case of invalid data, e.g.:

[2]:
```
def gravity_potential(G, M, m, r):
    if G < 0:
        raise ValueError("Gravitational constant must be greater than or equal␣
↪to zero")
    if M < 0:
        raise ValueError("Mass M must be greater than or equal to zero")
    if m < 0:
        raise ValueError("Mass m must be greater than or equal to zero")
    if r <= 0:
        raise ValueError("Distance r must be greater than zero")
    return G*M*m/r

V = gravity_potential(6.674e-11, 1.65e12, 6.1e2, 7e3)
```

Now, if we attempt

```
V = gravity_potential(-6.674e-11, 1.65e12, 6.1e2, 7e3)
```

we would see:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-20-ef1cff047ffd> in <module>()
     10     return G*M*m/r
     11
---> 12 V = gravity_potential(-6.674e-11, 1.65e12, 6.1e2, 7e3)

<ipython-input-20-ef1cff047ffd> in gravity_potential(G, M, m, r)
      1 def gravity_potential(G, M, m, r):
      2     if G < 0:
----> 3         raise ValueError("Gravitational constant must be greater than or equal to zero
      4     if M < 0:
      5         raise ValueError("Mass M must be greater than or equal to zero")

ValueError: Gravitational constant must be greater than or equal to zero
```

The error is detected and a message is printed that makes clear what the problem is. The message also shows where in the program the exception was raised.

Be liberal with the addition of checks in your programs.

3.1.1 Exception types

Above we have used the exception type '`ValueError`'. There are multiple built-in exception types (see https://docs.python.org/3/library/exceptions.html), and we can even write our own. Pick the exception type that best describes the type of error encountered. Above, we have used the exception type `ValueError` because it describes what we were checking for. If none of the specific built-in exceptions accurately describe the error, '`RuntimeError`' is often the best choice.

3.2 Averting the Ariane 5 explosion

Recall from Activity 03 that an Ariane 5 rocket exploded shortly after takeoff due to an integer overflow - the navigation software attempted to convert a 64-bit float (it was a velocity component) to a 16-bit integer. The largest value a 16-bit integer can store is 32767, and the velocity of the rocket exceeded 32767.

The below snippet of code would have raised an informative exception in pre-launch testing, which could then have been easily fixed.

[3]:
```python
import numpy as np

def cast_velocity(v):
    "Cast velocity to 16-bit with checking for overflow"

    # Largest number a 16-bit int can store (http://docs.scipy.org/doc/numpy/
 ↪reference/generated/numpy.iinfo.html)
    max16_it = np.iinfo(np.int16).max

    # Assign float velocity to 16-bit int if safe, otherwise raise exception
    if abs(v) <= max16_it:
        return np.int16(v)
    else:
        raise OverflowError("Cannot safely cast velocity to 16-bit int.")


# Cast rocket velocity to 16-bit int. If velocity is greater than 32767 an␣
 ↪exception will be raised.
v = cast_velocity(32767.0)
```

Increase the velocity to over 32767 to see what happens.

This bug should have been detected during pre-launch testing. However, unexpected issues could arise during rocket flight and we cannot just shut down the control software. The program needs to attempt to recover the situation. Next we look at how to handle problems that arise during execution of a program.

4 Catching and handling exceptions

When an exception is raised, by default a program will exit. However, we do not always want to exit a program when we encounter an exception. Sometimes we want to 'catch' the exception and do something else.

4.0.1 Exception handling and the USS Yorktown

In the case of the USS Yorktown, the ship's software should not stop if input data leads to a divide-by-zero, nor should it proceed erroneously and without warning. The software needs to 'catch' the divide-by-zero exception, and do something else. This might be reducing propulsion and asking for revised input.

In Python we catch exceptions with

```
try:
    # Attempt to do something here that might raise an exception
except FooError:
    # If a 'FooError' exception is raised above, do something here. For other exception types
    # the program will exit (if FooError is left out, any exception will end up here)
```

This is the basic and most common construct. It can be made more elaborate.

4.1 Checking interactive user input

Say our program asks the user to enter their age, and they enter an invalid value (negative number, something other than a number, etc). A program should raise an exception when an invalid value is input, but rather than exiting it would be better to ask the user to enter their age again. Here is an example:

[4]:
```
def get_user_age():
    "Function that asks user for their age. If input is invalid, user is␣
 ↪prompted to try again"
    try:
        # Get age from user input - if conversion to int fails Python raises an␣
↪exception
        age = int(input('How old are you? '))

        # Conversion to int has been successful, but we need to check that age␣
↪is positive. Raise
        # exception if age is less than 0
        if age < 0:
            raise ValueError("Age must be a positive integer")

        return age
    except:
        # Getting age from user input unsuccessful, so print message
        print("Invalid age entered. Please try again")

        # Prompt user again to input age
        return get_user_age()


# Uncomment the below lines to test
# age = get_user_age()
# print(age)
```

4.2 Example: integer type conversion

Below is another (contrived) take on the Ariane 5 rocket. Our preference is to cast the velocity component (float) to a 16-bit integer, but if the value is too large for a 16-bit integer we want to cast to a 32-bit integer. If the float is too large for a 32-bit integer we want to exit the program with an error message.

We first provide two functions for converting to a integer: the first to a 16-bit integer and the second to a 32-bit integer:

[5]:
```
def cast_velocity16(v):
    "Convert to a 16-bit int. Raise exception if this will cause an overflow"
    if abs(v) <= np.iinfo(np.int16).max:
         return np.int16(v)
    else:
        raise OverflowError("Cannot safely cast velocity to 16-bit int.")


def cast_velocity32(v):
    "Convert to a 32-bit int. Raise exception if this will cause an overflow"
    if abs(v) <= np.iinfo(np.int32).max:
         return np.int32(v)
    else:
        raise OverflowError("Cannot safely cast velocity to 32-bit int.")
```

We now perform a conversion, trying first to convert to a 16-bit integer, and if that is unsuccessful we attempt a conversion to a 32-bit integer:

[6]:
```python
def cast_velocity(v):
    try:
        # Try to cast v to a 16-bit int
        return cast_velocity16(v)
    except OverflowError:
        # If cast to 16-bit int failed (and exception raised), try casting to a␣
 ↪32-bit int
        try:
            return cast_velocity32(v)
        except OverflowError:
            # If cast to 32-bit int failed, raise exception
            raise RuntimeError("Could not safely cast velocity to an available␣
 ↪int type.")

# Velocity to cast (too large for a 16-bit int)
v_int = cast_velocity(42767.0)
print(v_int)
print(type(v_int))

# Velocity to cast (small enough for a 16-bit int)
v_int = cast_velocity(3210.0)
print(v_int)
print(type(v_int))
```

```
42767
<class 'numpy.int32'>
3210
<class 'numpy.int16'>
```

5 Testing

Testing is a critical part of software engineering for enhancing program quality, and for building the confidence we and others will have in a program. Testing is not only performed when developing a new program. Programs should come with a suite of tests that can be run regularly. This helps detect errors that might inadvertently creep into a program. For large projects, these tests might be run nightly or even every time a change is made to a program (the latter is known as *continuous integration*). Any program used for engineering analysis and design should have an extensive suite of tests. It would be negligent to use a program in real-life engineering analysis that is not covered by extensive testing.

When testing a program, we should test for both valid and invalid input data. For the valid cases the computed result should be checked against a known correct solution. For the invalid data cases, tests should check that an exception is raised. We will consider the former, and address the latter in an optional section.

Ideally, tests for a large program should have different levels of granularity. Some tests should test small blocks (individual functions), ideally in isolation from other parts of a program. The cause of unexpected behaviour in a large program can then be pin-pointed quickly. Higher level tests should test a program as a whole, and would typically represent user cases.

The programming examples in the preceding notebooks included little or no checking/testing. From now, we want to add more checks to our programs. For an example of correctness testing, let's consider the Fibonacci series from Activity 04. The function for computing Fibonacci numbers is:

[7]:
```
def f(n):
    "Compute the nth Fibonacci number using recursion"
    if n == 0:
        return 0  # This doesn't call f, so it breaks out of the recursion loop
    elif n == 1:
        return 1  # This doesn't call f, so it breaks out of the recursion loop
    else:
        return f(n - 1) + f(n - 2)  # This calls f for n-1 and n-2 (recursion),␣
 ↪and returns the sum
```

To build our confidence that the function is correct, we can check a number of computed terms in the series against known results. A helpful tool for this is the `assert` keyword:

[8]:
```
assert f(0) == 0
assert f(1) == 1
assert f(2) == 1
assert f(3) == 2
assert f(10) == 55
assert f(15) == 610
```

If all the assertions are true, there should be no output. Try changing one of the checks to trigger an assertion failure.

5.1 Test frameworks (optional)

Testing is so important to good software engineering that there are many tools to help us to write and run tests. A popular and powerful testing library for Python is `pytest` (http://doc.pytest.org/en/latest/). Before using it, we need to check that it is installed. We try to import it, and if that fails we install it:

[9]:
```
import sys
try:
    import pytest
except:
    try:
        !{sys.executable} -m pip -q install pytest
        import pytest
    except ImportError:
        !{sys.executable} -m pip -q --user install pytest
```

We have seen some simple testing of the Fibonacci series that checks for correctness. For the gravity potential problem at the start of the notebook we could also add some checks for correctness. But, we also would like an automated process to test that an exception *is* raised when user input is invalid.

Consider the gravity potential problem from the start of the notebook:

[10]:
```
def gravity_potential(G, M, m, r):
    if G < 0:
        raise ValueError("Gravitational constant must be greater than or equal␣
 ↪to zero")
    if M < 0:
        raise ValueError("Mass M must be greater than or equal to zero")
    if m < 0:
        raise ValueError("Mass m must be greater than or equal to zero")
    if r <= 0:
        raise ValueError("Distance r must be greater than zero")
    return G*M*m/r
```

We now would like to add some tests that check that invalid data raises an exception, i.e. we want to exit with an error if the function `gravity_potential` *does not* raise an exception. We can do this with PyTest:

[11]:
```
# Use PyTest to check that ValueError is raised for invalid input data
import pytest

# Check that G < zero raises a ValueError
with pytest.raises(ValueError):
    gravity_potential(-6.674e-11, 1.65e12, 6.1e2, 7e3)

# Check that M < zero raises a ValueError
with pytest.raises(ValueError):
    gravity_potential(6.674e-11, -1.65e12, 6.1e2, 7e3)

# Check that m < zero raises a ValueError
with pytest.raises(ValueError):
    gravity_potential(6.674e-11, 1.65e12, -6.1e2, 7e3)

# Check that r < zero raises a ValueError
with pytest.raises(ValueError):
    gravity_potential(6.674e-11, 1.65e12, 6.1e2, -7e3)

# Check that r=0 raises a ValueError
with pytest.raises(ValueError):
    gravity_potential(6.674e-11, 1.65e12, 6.1e2, 0.0)
```

We should test that our programs compute correct results, and we should check that exceptions are indeed raised for invalid input data.

6 Exercises

Complete now the 09 Exercises notebook.