W02_06_Errors_Exceptions_Tracebacks

September 12, 2022

1 Errors, Exceptions, and Tracebacks

1.1 Exceptions

An *exception* is a way of signalling that something unusual ("exceptional") has happened while the code is running.

- How to raise an exception
- How to handle one.

That might mean an error, for example. If the code doesn't know how to recover from the error, it can raise (or "throw") an exception. That changes the control flow of the program. If the exception is not handled (or "caught") inside the currently-executing function, then the function exits – immediately, and without returning any value. The function that called this one then has a chance to handle (or "catch") it. If not, it continues to the function that called *that* one, and so on. If it is never caught, it propagates "all the way up", the program exits ("crashes"), and you see a Traceback on your screen.

For unrecoverable errors, that is probably just fine. Other times, we will want to handle an exception before the program exits, either within the current function, or in an enclosing one.

So, there are two main mechanisms we need to know about: how to raise an exception, and how to handle one.

[1]:
```
def get_mobile_number(operator, user):
    """Given an operator and a user number,
    return the full mobile number. For example:

    >>> get_mobile_number("meteor", "1234567")
    '0851234567'

    """

    # We have a mapping from operators to prefixes.
    d = {"virgin": "087", "three": "086",
         "meteor": "085"}
    return d[operator] + user
```

This works as follows:

[2]:
```
get_mobile_number("virgin", "1234567")
```

[2]:
```
'0871234567'
```

But if we run something like the following, we'll get a KeyError, because T-Mobile doesn't exist:

[3]:
```
get_mobile_number("T-Mobile", "1234567")
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-3-5c616191561f> in <module>
----> 1 get_mobile_number("T-Mobile", "1234567")

<ipython-input-1-390f2cbf7649> in get_mobile_number(operator, user)
     10     # We have a mapping from operators to prefixes.
     11     d = {"virgin": "087", "three": "086", "meteor": "085"}
---> 12     return d[operator] + user

KeyError: 'T-Mobile'
```

That KeyError is an *exception*. It was raised by the code that implements dictionaries to indicate the requested key wasn't found. Our variable d is a dictionary. When we tried to look up the value of "T-Mobile", it found there was no such key in d. It (the dict) didn't print out an error message, or return a special error value, or anything like that. All the dict did was raise an exception, one called KeyError. Because our code didn't handle the exception, it propagated all the way to the console and caused a crash.

1.2 Catching exceptions

But what if we wanted to do something else with a bad operator, instead of crashing the program? Let's say we're printing out mobile numbers, and anytime we get an unrecognised operator, we should just print nothing, instead. We can do that by "catching" the exception, using a try-except block.

A try-except block looks a bit like an if-statement. That reflects the fact that it affects control flow, just like an if statement does. It means something like: run this, and if there's an exception, clean up by running something else instead.

We can either catch the error from outside the function:

[4]:
```
user = "1234567"
oper = "T-Mobile"
try:
    # try this code...
    print(get_mobile_number(oper, user))
except KeyError:
    # ... and if a KeyError happens
    # ... do this instead.
    print("Operator " + oper + " not found")
```

```
Operator T-Mobile not found
```

Or we can catch it inside the function. It depends what we want to achieve. The following new version won't crash. If there is a KeyError while we're "trying" the first piece of code, then we'll just return None instead.

[5]:
```
def get_mobile_number(operator, user):
    # We have a mapping from operators to prefixes. Note that we use a
    # string for the number and the prefix -- why?
    d = {"virgin": "087", "three": "086", "meteor": "085"}
    try:
        return d[operator] + user # try this code...
    except KeyError: # ... and if a KeyError happens...
        return None # ... do this instead.

print(get_mobile_number(oper, user))
```

```
None
```

Exercise: Write a function which accepts two numbers and returns the first divided by the second. Observe the exception that is thrown if you try to divide by zero. Now use a try-except block to catch that exception and print a sensible error message, instead of crashing the program.

We can also use multiple `except` clauses, e.g.:

```
try:
    something
except KeyError:
    deal with KeyError
except ZeroDivisionError:
    deal with ZeroDivisionError
```

Exercise: Observe the exceptions that are thrown in the below cases, and then repair the function using multiple `except` clauses to do something sensible that doesn't crash.

[6]:
```
def subtract_one(L):
    L[0] -= 1
    return L
```

[7]:
```
subtract_one([0, 1, 2, 3, 4]) # should work ok
```

[7]:
```
[-1, 1, 2, 3, 4]
```

[8]:
```
subtract_one((0, 1, 2, 3, 4))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-377662191a46> in <module>
----> 1 subtract_one((0, 1, 2, 3, 4))

<ipython-input-6-3439f8f0c97e> in subtract_one(L)
      1 def subtract_one(L):
```

```
----> 2     L[0] -= 1
      3     return L

TypeError: 'tuple' object does not support item assignment
```

[9]:
```
subtract_one([])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-9ea726afb39b> in <module>
----> 1 subtract_one([])

<ipython-input-6-3439f8f0c97e> in subtract_one(L)
      1 def subtract_one(L):
----> 2     L[0] -= 1
      3     return L

IndexError: list index out of range
```

1.3 Raising exceptions

So far, we've seen how to handle ("catch") exceptions. What about raising ("throwing") them? The most common scenario is when our function receives arguments which are "impossible" in some way, or when processing data that doesn't make sense.

[31]:
```
from math import sqrt
def triangle_area(x, y, z):
    """Calculate the area of a triangle (not necessarily right-angled)
    given its three sides. Heron's formula tells us the area. However,
    not all values of x, y, and z are valid. If the two smallest of
    them add up to less than the largest, no such triangle exists.
    """
    # Order so that x <= y <= z
    x, y, z = sorted((x, y, z))
    # Check that the values are valid
    if x + y < z:
        # ... and if not, raise an exception.
        raise ValueError("No such triangle exists")

    # Heron's formula [https://www.mathsisfun.com/geometry/herons-formula.html]
    s = (x + y + z)/2.0
    area = sqrt(s * (s-x) * (s-y) * (s-z))
    return area
```

[32]:
```
print(triangle_area(5, 5, 5)) # should work ok
```

```
10.825317547305483
```

[33]:
```
print(triangle_area(10, 2, 2)) # we detect an impossible triangle and raise an␣
↪exception
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-86d7a7cf954e> in <module>
----> 1 print(triangle_area(10, 2, 2)) # we detect an impossible triangle and␣
↪raise an exception

<ipython-input-31-b45acaece40c> in triangle_area(x, y, z)
     11     if x + y < z:
     12         # … and if not, raise an exception.
---> 13         raise ValueError("No such triangle exists")
     14
     15     # Heron's formula [https://www.mathsisfun.com/geometry/
↪herons-formula.html]

ValueError: No such triangle exists
```

1.4 Exceptions and doctests

We can take account of exceptions in our *doctests*. That is, we can write a doctest where the expected result is a Traceback and exception. For example, we can write a doctest like the following. Note, we can put `...` to indicate some extra text that the doctest can ignore.

[34]:
```
def triangle_area(x, y, z):
    """
    A normal, working call:
    >>> triangle_area(5, 5, 5)
    10.825317547305483

    An impossible triangle: we hope to see a Traceback:
    >>> triangle_area(10, 2, 2)
    Traceback (most recent call last):
    ...
    ValueError: No such triangle exists
    """
    # Order so that x <= y <= z
    x, y, z = sorted((x, y, z))
    # Check that the values are valid
    if x + y < z:
        # ... and if not, raise an exception.
        raise ValueError("No such triangle exists")

    # Heron's formula [https://www.mathsisfun.com/geometry/herons-formula.html]
```

```
    s = (x + y + z)/2.0
    area = sqrt(s * (s-x) * (s-y) * (s-z))
    return area
```

[35]:
```
import doctest
doctest.testmod()
```

```
**********************************************************************
File "__main__", line 5, in __main__.get_last_n_elements
Failed example:
    get_last_n_elements("abcde", 8)
Expected:
    Traceback (most recent call last):
    …
    ValueError: Requested too many elements (8 versus 5)
Got:
    Traceback (most recent call last):
      File "/Users/jmmcd/anaconda3/lib/python3.7/doctest.py", line 1329, in
__run
        compileflags, 1), test.globs)
      File "<doctest __main__.get_last_n_elements[1]>", line 1, in <module>
        get_last_n_elements("abcde", 8)
      File "<ipython-input-27-5007f00f1252>", line 11, in get_last_n_elements
        raise ValueError(f"Requested too many elements ({n} versus {len(s)})")
    ValueError: Requested too many elements (8 versus 5)
**********************************************************************
1 items had failures:
   1 of   2 in __main__.get_last_n_elements
***Test Failed*** 1 failures.
```

[35]:
```
TestResults(failed=1, attempted=4)
```

Exercise: What happens here if s is shorter than n elements? Try it.

[15]:
```
def get_last_n_elements(s, n):
    return s[-n:]
```

[16]:
```
get_last_n_elements("abcde", 2)
```

[16]:
```
'de'
```

Now, change the code to raise a `ValueError` in that case. What if `n` was negative? Again, raise a `ValueError` in that case. Test it works correctly. Finally, add some doctests. Add both positive ones (normal operation) and negative ones (`ValueError` tracebacks)

1.4.1 Solutions

Write a function which accepts two numbers and returns the first divided by the second. Observe the exception that is thrown if you try to divide by zero. Now use a try-except block to catch that exception and print a sensible error message, instead of crashing the program.

[2]:
```
def divide(a, b):
    return a / b
```

[3]:
```
divide(10, 0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-ccc290a9b5cb> in <module>
----> 1 divide(10, 0)

<ipython-input-2-dc50b17f0ae8> in divide(a, b)
      1 def divide(a, b):
----> 2     return a / b

ZeroDivisionError: division by zero
```

[7]:
```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError: # note spelling, exactly as in the traceback above
        # contradicting a point I made in class :)
        # usually nicer not to print when a function is expected to return a␣
 ↪value
        print("Don't divide by zero!")
```

[6]:
```
divide(10, 0)
```

```
Don't divide by zero!
```

Observe the exceptions that are thrown in the below cases, and then repair the function using multiple except clauses to do something sensible that doesn't crash.

[11]:
```
def subtract_one(L):
    try:
        L[0] -= 1
    except TypeError:
        print("Warning, cannot change input L of type", type(L))
    except IndexError:
        print("Warning, cannot change empty list")
    return L
```

[12]:
```
subtract_one((0, 1, 2, 3, 4))
```

```
Warning, cannot change input L of type <class 'tuple'>
```

[12]:
```
(0, 1, 2, 3, 4)
```

[13]:
```
subtract_one([])
```

```
Warning, cannot change empty list
```

[13]:
```
[]
```

What happens here if s is shorter than n elements? Try it.

[14]:
```
def get_last_n_elements(s, n):
    return s[-n:]
```

[16]:
```
get_last_n_elements("abcde", 2) # ok
```

[16]:
```
'de'
```

[17]:
```
get_last_n_elements("abcde", 8) # no crash, but misleading for the user I think
```

[17]:
```
'abcde'
```

Now, change the code to raise a ValueError in that case. What if n was negative? Again, raise a ValueError in that case. Test it works correctly. Finally, add some doctests. Add both positive ones (normal operation) and negative ones (ValueError tracebacks)

[38]:
```
def get_last_n_elements(s, n):
    """
    >>> get_last_n_elements("abcde", 2)
    'de'
    >>> get_last_n_elements("abcde", 8)
    Traceback (most recent call last):
    ...
    ValueError: Requested too many elements (8 versus 5)
    """
    if n > len(s):
        raise ValueError(f"Requested too many elements ({n} versus {len(s)})")
    elif n < 0:
        raise ValueError("Requested negative number of elements")
    return s[-n:]
```

[39]:
```
import doctest
doctest.testmod()
```

[39]:
```
TestResults(failed=0, attempted=4)
```