Lecture 16 - Error handling and exceptions

Computer Programming

Robert Vincent and Samia Hilal

Marianopolis College

May 2, 2022

Handling errors

- What to do when a function fails for some reason?
- We can use a special value to indicate something went wrong.
- This may be tricky - There may be no obvious choice for the special value.
- This approach requires all users of a function to be aware of possible failures, the special value, and act accordingly.

Using a special result 1

```
def my_index(my_list, value):
    for ind in range(len(my_list)):
        if value == my_list[ind]:
            return ind
    return -1 # special value if not found
```

- file: my_index_err.py
- If we don't find the value, we return -1.
- If we find the value, we return a number greater than or equal to zero.

Using a special result 2

```
x = [1, 3, 9, 8, 5, 1, 0]
n = my_index(x, 5)
if n >= 0:          # did this work?
    m = my_index(x, 2)
    if m >= 0: # did this work?
        print("Found both 5 and 2")
    else:
        print("Failed to find 2")
else:
    print("Failed to find 5")
```

- We must check every time we call the function.
- This can get complicated quickly.

Problems with this approach

- Checking every return value for an error can lead to verbose code.
- Different functions will use different special results, so we have to remember each case.
- Not all functions have obvious "error" values.
- Many error conditions are rare - why add so much code for them?
- **Another approach** would be to return error codes. However, it can be tricky to interpret.

Exceptions in Python

- You have already encountered some exceptions.
- Consider the built in function `int()` and the list method `index()`.

```
>>> x = int('abc')
ValueError: invalid literal for int() with base
10: 'abc'
>>> lst = [1,2,3]
>>> lst.index(4)
ValueError: 4 is not in list
```

- Or, if we divide by zero:

```
>>> x,y=10,0
>>> print(x/y)
ZeroDivisionError: division by zero
```

Exceptions in Python

- Other common exceptions you've probably seen:

```
>>> x = [1,2,3]
>>> x[2] = 4
>>> print(x)
[1, 2, 4]
>>> x[3] = 5
IndexError: list index out of range
```

- Or when we try to access an undefined variable:

```
>>> Temp = 10
>>> print(temp)
NameError: name 'temp' is not defined
```

Exception Handling

- If explicit errors (or other *exceptional* conditions) occur, they can be handled by special language syntax.
- Separate from normal program operation.
- Can be handled where convenient.
- Programs terminate if an exception is not handled.
- Common in most programming languages developed since the 1980's: C++, Java, etc.

Exception classes in Python

- `ValueError` and `ZeroDivisionError` are examples of *exceptions*.
- Exceptions are objects whose class is typically derived from the builtin class `Exception`.
- An exception is *raised* or *thrown* to indicate an error or unusual condition.
- If no provision is made for handling the exception, the program exits immediately.
- So how do we raise or handle exceptions?

Raising an exception in our own code

```
def my_index(my_list, value):
    for ind in range(len(my_list)):
        if value == my_list[ind]:
            return ind
    raise ValueError('Value ' + str(value) +
                     ' not found.')

>>> L = [1,2,3]
>>> x = my_index(L,4)
ValueError:  Value 4 not found.
```

- Function fails and raises `ValueError` exception.
- A `ValueError` is raised because we are trying to assign x to a value that has a problem.
- The string, and other information, will be made available to the exception handler.

Handling exceptions using try statement

```
def my_index(my_list, value):
    for ind in range(len(my_list)):
        if value == my_list[ind]:
            return ind
    raise ValueError('Value ' + str(value) +
                     ' not found.')
#Handling exceptions: Program does not fail
x = [1, 3, 9, 8, 5, 1, 0]
try:
    n = my_index(x, 8)
    m = my_index(x, 2)
except ValueError as ex:
    print(ex) # prints Value 2 not found
else:
    print("Found both 8 and 2")
```

The `try` statement

- Required first clause is `try:` with an indented statement list (code may cause an exception).
- One or more `except` clauses with their own statement lists.
- If an exception is raised in the `try` clause, the first matching except clause will be run.
- *At most* one of the `except` clauses will execute.
- An optional `else:` clause will run only if no exceptions are raised within the `try` clause.
- An optional `finally:` clause *always* runs after everything else.

The except clause

- Exceptions form a class hierarchy.
- The `except` clause usually includes the class name of an exception.
- There may also be the reserved word `as` followed by a local name to use for the exception object.
- When an exception is raised, each enclosing `except` clause is evaluated from top to bottom.
- If the exception's type matches the type given in the `except` clause, that clause will execute.

The try statement syntax

```
try:
    # Executes until an exception is raised.
    statement1
    statement2
    ...
except ValueError as name:
    # Executes iff ValueError raised!
    statement3
    statement4
    ...
else:
    # Executes iff no exception raised!
    statement5
    statement6
    ...
```

Exception Handling Execution

- When an exception occurs, `except` statements are checked in order and the first match only is executed.
- The `except Exception ...` statement is always a match so it is normally listed after all the others.
- If an exception occurs that cannot be matched to any of the `except` statements defined, the program fails.
- If no exception occurs and there is an `else` defined at the end of the list of exceptions, the `else` is executed.

Another example

```
w,x,y=10,2,8
try:
    print(w / x)          # Prints 5.0
    print(y / (x - 2))    # Exception!
    print(x / y)          # Skipped.
except Exception as e:
    print('Exception:', e)
else:
    print('OK')       # Never reached.
print('Done')         # Final message.
```

This code will print:

```
5.0
Exception: division by zero
Done
```

Two exception clauses

```
while True:
    try:
        x = input('OK? ')
        n = int(x) # Can raise ValueError
        print(10 / n) # Can divide by zero
    except ValueError as exc:
        print('Ex1',exc)
    except ZeroDivisionError as exc:
        print('Ex2',exc)
    else:
        print('OK!')
```

Single exception clause

```
while True:
    try:
        x = input('OK? ')
        n = int(x) # Can raise ValueError
        print(10 / n) # Can divide by zero
    except (ValueError,
            ZeroDivisionError) as exc:
        print('Ex1',exc)
    else:
        print('OK!')
```

Generic exception clause

```
while True:
    try:
        x = input('OK? ')
        n = int(x) # Can raise ValueError
        print(10 / n) # Can divide by zero
    except Exception as exc:
        print('Ex1',exc)
    else:
        print('OK!')
```

- Will catch *any* exception that derives from `Exception`, not just `ValueError` or `ZeroDivisionError`.

Empty exception clause

```
while True:
    try:
        x = input('OK? ')
        n = int(x) # Can raise ValueError
        print(10 / n) # Can divide by zero
    except:
        print('Ex1')
    else:
        print('OK!')
```

- Will not catch *any* exception at all. We can't even exit the loop!
- Can't examine the exception (no `as` clause).

Builtin exceptions

- `TypeError` - An object's type does not support a requested operation.
- `ValueError` - An object's value is not valid in a context.
- `IndexError` - A list subscript is out of range.
- `NameError` - A name is not defined.
- `OSError` - An operating system call failed.
- `IOError` - An input/output operation failed.
- `ZeroDivisionError` - The denominator in a division or remainder operation is zero.

Summary

- Exceptions are a general tool for handling errors.
- The `raise` statement triggers an exception.
- The `try` statement can handle exceptions.
- Exceptions themselves are objects that are part of a special class hierarchy in Python.