Lecture #21: Exceptional Conditions

Part of the contract between the implementor and client is the set of preconditions under which a function, method, etc. is supposed to operate. Example:

class Rational:
def init(self, x, y):
    """The rational number x/y. Assumes that x and y are ints and y != 0."""

Here, "x and y are ints and y!=0" is a precondition on the client.

So what happens when the precondition is not met?

Failed preconditions

Possible Responses

Programming Errors

When something goes wrong, the program usually displays some sort of error message.

Exceptional Conditions

Numerous programming languages, including Python, support a general notion of exceptional condition or exception with supporting grammar and syntax that separate error-handling from main program flow. Exceptional conditions come in many forms:

- Errors in the code (e.g., a function tries to divide by zero)
- Exceptional conditions returned by libraries (e.g., exceptions thrown by a network library)

Python has its own exceptions:

- Syntax errors (e.g., "def" without a colon)
- Run-time errors (e.g., division by zero)
- User errors (e.g., "print" without parentheses)

Possible Responses

Assertions

The Python assert statement provides a standard way to check for programmer errors. Two forms:

assert CONDITION
assert CONDITION, DESCRIPTION

These are equivalent to:

if not CONDITION:
    raise AssertionError
if not CONDITION:
    raise AssertionError(DESCRIPTION)

By default, assert statements are turned off when Python is compiled with -O or -O2.

Because assert statements are turned off in release builds, they are not appropriate for detection of user errors or other errors that the program is deliberately designed to handle.
An exception mechanism is a control structure that
– Halts execution at one point in a program (called raising or throwing an exception).
– Resumes execution at some other, previously designated point in the program (called catching or handling an exception).

In Python, the `raise` statement raises (or throws exceptions, and the `try` statement catches them.

```python
def f0(...):
    try:
        g0(...)
        # 1. Call of g0...
        OTHER STUFF # Skipped
    except:
        handle oops # 4. Handle problem

def g1(...):
    # 2. Called by g0, possibly many calls down
    if detectError():
        raise Oops() # 3. Raise exception
    MORE # Skipped
```

Last modified: Thu Mar 16 16:52:45 2017 CS61A: Lecture #21

Standard Exceptions

• Exceptions are objects of builtin class `BaseException` or a subtype of it.
• The Python language and its library uses several predefined sub-classes, such as:
  - `TypeError` A value has the wrong type for an operation.
  - `IndexError` Out-of-bounds list or tuple index (e.g.).
  - `KeyError` Nonexistent key to dictionary
  - `ValueError` Other inappropriate values of the right type.
  - `AssertionError` An `assert` statement with a false assertion.
  - `IOError` Non-existent file, e.g.
  - `OSError` Bad operand to an operating-system call.

Communicating the Reason

• Normally, the handler would like to know the reason for an exception.
• “Reason,” being a noun, suggests we use objects, which is what Python does.
• Python defines the class `BaseException`. It or any subclass of it may convey information to a handler. We’ll call these `exception classes`.
• `BaseClassException` carries arbitrary information as if declared:
```python
class BaseException:
def init(self, *args):
    self.args = args
```
• The `raise` statement then packages up and sends information to a handler:
```python
raise ValueError("x must be positive", x, y)
raise ValueError # Short for raise ValueError()
e = ValueError("exceptions are just objects!")
raise e # So this works, too
```

Handlers

• A function indicates that something is wrong; it is the client (caller) that decides what to do about it.
• The `try` statement allows one to provide one or more handlers for a set of statements, with selection based on the type of exception object thrown.
```python
try:
    assorted statements
except ValueError:
    print("Something was wrong with the arguments")
except EnvironmentError:
    # Also catches subtypes IOError, OSError
    print("The operating system is telling us something")
except:
    # Some other exception
    print("Something wrong")
```

Retrieving the Exception

• So far, we’ve just looked at exception types.
• To get at the exception object, use a bit more syntax:
```python
try:
    assorted statements
except ValueError as exc:
    print("Something was wrong with the arguments:", exc)
```

Cleaning Up and Reraising

• Sometimes we catch an exception in order to clean things up before
  • returning the exception

```
```
Finally Clauses

More generally, we can clean things up regardless of how we leave the `try` statement:

```python
for i in range(100):
    try:
        setTimer(10)  # Set time limit
    if found(i):
        break
    longComputationThatMightTimeOut()
finally:
    cancelTimer()  # Continue with 'break' or with exception
```

This fragment will always cancel the timer, whether the loop ends because of `break` or a timeout exception. After which, it carries on whatever caused the `try` to stop.

This can be effected concisely with:

```python
with open(filename, "w") as out:
    out.write(text)
```

because of a `Garbage Collection (GC) Trampolining Mechanism)`

Other Uses of Exceptions

- We've described a software-engineering motivation for exceptions: dealing with erroneous situations.
- In effect, specifying possible exceptions is therefore part of the interface.
- Exceptions are a way of returning information from a function "out of band," and allowing programmers to cleanly separate error-handling from normal cases.
- In effect, specifying possible exceptions is therefore part of the interface.
- Exceptions are a way of returning information from a function "out of band," and allowing programmers to cleanly separate error-handling from normal cases.

Summary

- Exceptions are a way of returning information from a function "out of band," and allowing programmers to cleanly separate error-handling from normal cases.
- In effect, specifying possible exceptions is therefore part of the interface.
- Exceptions are a way of returning information from a function "out of band," and allowing programmers to cleanly separate error-handling from normal cases.
- In effect, specifying possible exceptions is therefore part of the interface.