Lecture #21: Exceptional Conditions

Failed preconditions

- 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?

Programmer Errors

- Python has preconditions of its own.
- E.g., type rules on operations: `3 + (2, 1)` is invalid.
- What happens when we (programmers) violate these preconditions?

Outside Events

- Some operations may entail the possibility of errors caused by the data or the environment in which a program runs.
- I/O over a network is a common example: connections go down; data is corrupted.
- User input is another major source of error: we may ask to read an integer numeral, and be handed something non-numeric.
- Again, what happens when such errors occur?

Possible Repsonses

- One approach is to take the point of view that when a precondition is violated, all bets are off and the implementor is free to do anything.
  - Corresponds to a logical axiom: False $\Rightarrow$ True.
  - But not a particularly helpful or safe approach.
- One can adopt a convention in which erroneous operations return special error values.
  - Feasible in Python, but less so in languages that require specific types on return values.
  - Used in the C library, but can't be used for non-integer-returning functions.
  - Error prone (too easy to ignore errors).
  - Cluttered (reader is forced to wade through a lot of error-handling code, a distraction from the main algorithm).
- Numerous programming languages, including Python, support a general notion of *exceptional condition* or *exception* with supporting syntax and semantics that separate error handling from main program logic.

Assertions

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

```
assert CONDITION
assert CONDITION, DESCRIPTION
```

- Equivalent to either

```
if __debug__ and not CONDITION:
    raise AssertionError
if __debug__ and not CONDITION:
    raise AssertionError(DESCRIPTION)
```

- By default, `__debug__` is true. **python3 -O**. . . makes it false.
- Because it can be turned off, this is not appropriate for detection of user errors, or other errors that the program is deliberately designed to handle.

Exceptions

- 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.

```
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
```

Standard Exceptions

- Exceptions are objects of builtin class `BaseException` or a subtype of it.
- The Python language and its library uses several predefined subclasses, 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:

```
class BaseException:
    def __init__(self, *args):
        self.args = args
    ...
```

- The raise statement then packages up and sends information to a handler:

```
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.

```
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 objects, use a bit more syntax:

```
try:
    assorted statements
except ValueError as exc:
    print("Something was wrong with the arguments: {0}", exc)
```

Cleaning Up and Reraising

- Sometimes we catch an exception in order to clean things up before the real handler takes over.

```
inp = open(aFile)
try:
    Assorted processing
    inp.close()
except:
    inp.close()
    raise          # Reraise the same exception
```

Finally Clauses

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

```
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.

"With" Clauses

- The `finally` statement comes in useful in a number of standard places, such as generally
  - When the program reserves some *resource* for its use from a small set of such resources, and must be sure to return it to prevent deadlocking the system.
  - When the program creates some kind of persistant object (like a file) that requires some specific action before it is complete.
- Such situations are sufficiently common that Python's designers decided to provide a more concise and general construct to handle them.
- Just as `for` statements and generator definitions are associated with particular kinds of object type—iterator and iterables—this new construct is associated with a kind of object known as a *context manager.*

Example

- If you really want to be tidy about using a file, you need the following pieces, at least:

```
def writeAll(filename, text):
    """Create (or overwrite) a file named FILENAME with the string TEXT."""
    try:
        out = open(filename, "w")  # Open for writing
        out.write(text)
    finally:
        out.close()                # Make sure everything is written
```

- This can be effected concisely with

```
def writeAll(filename, text):
    """Create (or overwrite) a file named FILENAME with the string TEXT."""
    with open(filename, "w") as out:
        out.write(text)
```

- This is because Python files (returned by `open`) implement the methods required to be context managers: `__enter__` and `__exit__`.

With-Statement Details (Simplified)

- The statement

```
with E1 as VAR:
    STATEMENTS
```

  is essentially the same as

```
mgr = E1
VAR = mgr.__enter__()
ok = True
try:
    try:
        STATEMENTS
    except:
        ok = False
        if not mgr.__exit__(info about the exception):
            raise   # Re-raise the exception
finally:
    if ok:
        mgr.__exit__(None, None, None)
```

- [WARNING: This is not entirely correct, being simplified, but it gives the general idea.]

Other Uses of Exceptions

- We've described a software-engineering motivation for exceptions: dealing with erroneous conditions.
- But from a programming-language point of view, they're just another control structure.
- Python uses them in non-erroneous situations as well:
  - We've seen that *iterators* use `StopIteration` to indicate they have no more elements.
  - Alternatively, Python can create an iterator out of any object that has a `__getitem__` method, which (as usual) raises `IndexError` to indicate the end of a sequence.

Summary

- Exceptions are a way of returning information from a function "out of band," and allowing programmers to clearly separate error handling from normal cases.
- In effect, specifying possible exceptions is therefore part of the interface.
- Usually, the specification is implicit: one assumes that violation of a precondition might cause an exception.
- When a particular exception indicates something that might normally arise (*e.g.*, bad user input), it will often be mentioned explicitly in the documentation of a function.
- Finally, `raise` and `try` may be used purely as normal control structures. By convention, the exceptions used in this case don't end in "Error."