Lecture #24: More Scheme, Exceptional Conditions

Announcements

- My Thursday office hours this week (only) are 5–6PM.
- Quiz to be released today. Due Friday. Part of preparation for next test.

Some List Hacking in Scheme

```
(define (shuffle L1 L2)
    "The list consisting of the first element of L1, then
    the first of L2, then the second of L1, etc., until
    the elements of one or the other list is exhausted."
)
```

Simple For Loop

- Example of converting Python to Scheme:

```
def isPrime(x):
  if x < 2:
    return False
  elif x == 2:
    return True
  else:
    for k in \
        range(2, int(sqrt(x)+2)):
      if x % k == 0:
        return False
    return True
```

```
(define (prime? x)
    (define (no-factor? k lim)
        (cond ((>= k lim) #t)
              ((= (remainder x k) 0) #f)
              (#t (no-factor? (+ k 1) lim))))

    (cond ((< x 2) #f)
          ((= x 2) #t)
          (#t (no-factor? 2
               (floor (+ (sqrt x) 2))))))
```

Can We Generalize?

- The **for** loop translated to a recursion.
- A common idiom, but somewhat bulky. Can we generalize?

```
(define (find? start limit body)
    "True iff (BODY k) yields #t for any START <= k < LIMIT."
    ???
)
```

- Want to be able to write

```
(no-factor? 2 (floor (+ (sqrt x) 2)))} as
(not (find? 2 (floor (+ (sqrt x) 2))
           (lambda (k) (= (remainder x k) 0))))
```

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({\it DESCRIPTION\})
```

- By default, `__debug__` is true. **python3 -O...** makes it false.

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 throws exceptions, and try statements catch them:

```
def f0(...):
    try:
        g0(...)        # 1.  Call of g...
        OTHER STUFF    # Skipped
    except:
        handle oops    # 3.  Handle problem
...
def g1(...): # Eventually called by g0, possibly many calls down
    if detectError():
        raise Oops()     # 2.  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.

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."