Lecture #17: Abstraction Support: Exceptions, Operators, Properties

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:
  
  ```python
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 Responses

• 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 ⇒ 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.

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

Cleaning Up and Reraising

- Sometimes we catch an exception in order to clean things up before the real handler takes over.
  ```python
  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:
  ```python
  for i in range(100)
    try:
      setTimer(10)  # Set time limit
      if found():
        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.

Standard Exceptions

- See the Python library for a complete rundown.
- We'll often encounter `ValueError` (inappropriate values), `AttributeError` (x.foo, where there is no foo in x), `TypeError`, `OSError` (bad system call), `IOError` (such as nonexistent files).
- Other exceptions are not errors, but are used because `raise` is a convenient way to achieve some effect:
  - `StopIteration`: see last lecture.
  - `SystemExit`: Results from `sys.exit(n)`, which is intended to end a program.
Example: Implementing Iterators

- **An iterator** is an abstraction device for hiding the representation of a collection of values.
- The **for** statement is actually a generic control construct with the following meaning (well, Python adds a few more bells and whistles):
  ```python
tmp_iter = C.__iter__()
try:
  while True:
    x = tmp_iter.__next__()
S
except StopIteration:
  pass
```
- The **__next__** method can use the **raise StopIteration** statement to cause the loop to exit.
- Types that implement **__iter__** are called **iterable**, and those that implement **__next__** are **iterators**.
- The built-in functions **iter(x)** and **next(x)** are defined to call **x.__iter__()** and **x.__next__()**.

Problem: Reconstruct the range class

- Want **Range(1, 10)** to give us something that behaves like a Python range, so that this loop prints 1-9:
  ```python
  for x in Range(1, 10):
    print(x)
  ```

```python
class Range:
    def __init__(self, low, high):
        self._low = low
        self._high = high
    def __iter__(self):
        return RangeIter(self)

class RangeIter:
    def __init__(self, limits):
        self._bound = limits._high
        self._next = limits._low
    def __next__(self):
        if self._next >= self._bound:
            raise StopIteration
        else:
            self._next += 1
        return self._next-1
```

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."

Back To Rationals

- Before, we implemented rational numbers as functions. The "standard" way is to use a class.
- There are a few interesting problems along the way, at least if you want to make something that meets our natural expectations.
- Python has defined a whole bunch of library classes to capture different kinds of number (see `numbers` and `fractions`), but we're going to build our own here.

Some Basics

- We'd like rational numbers, with the usual arithmetic.
- Furthermore, we'd like to integrate rationals with other numeric types, especially `int` and `float`.
- So, let's start with the constructor:

```python
class rational:
    def __init__(self, numer=0, denom=1):
        if type(numer) is not int or type(denom) is not int:
            raise TypeError("numerator or denominator not int")
        if denom == 0:
            raise ZeroDivisionError("denominator is 0")
        d = gcd(numer,denom)
        self._numer, self._denom = numer // d, denom // d
```

Arithmetic

- Would be nice to use normal syntax, such as `a+b` for rationals.
- But we know how to do that from early lectures:

  ```python
  def __add__(self, y):
    return rational(self._numer * y._denom + self._denom * y._numer,
        self._denom * y._denom)
  ```

  ```python
  def __add__(self, y):
    y = rational._coerceToRational(y)
    return rational(self._numer * y._denom + self._denom * y._numer,
        self._denom * y._denom)
  ```

- What do we do if `y` is an `int`?
- One solution: **Coercion**:

```python
def _add__add__(self, y):
    return rational(self._numer * y._denom + self._denom * y._numer,
        self._denom * y._denom)
```
Coercion

In programming languages, coercion refers to conversions between types or representations that preserve abstract values.

```python
@staticmethod
def _coerceToRational(y):
    if type(y) is rational:
        return y
    else:
        return ?
```

Type Dispatching

- But now what about 3 + rational(1,2)? Ints don't know about rationals.
- This is a general problem with object-oriented languages. I call it "worship of the first parameter." It's the type of the first parameter (or that left of the dot) that controls what method gets called.
- Others use the phrase "the expression problem," because it arises in the context of arithmetic-expression-like things.
- There are various ways that languages have dealt with this.
- The brute-force solution is to introduce multimethods as a language feature (functions chosen on the basis of all parameters’ types.)
- Or one can build something like this explicitly:

```python
_add_dispatch_table = { (rational, int): _addri,
                        (int, rational): _addir, ...}

def __add__(self, y):
    _add_dispatch_table[(type(self), type(y))](self, y)
```

A Python Approach

- The dispatch-table requires a lot of cooperation among types.
- Python uses a different approach that allows extensibility without having to change existing numeric types.
- The expression x+y first tries x.__add__(y).
- If that throws the exception Not Implemented Error, it next tries y.__radd__(x).
- The __add__ functions for standard numeric types observe this, and throw Not Implemented Error if they can’t handle their right operands.
- So, in rational:

```python
def __radd__(self, y):
    return rational._coerceToRational(y).__add__(x)
```

Syntax for Accessors

- Our previous implementation of rational numbers had functions for accessing the numerator and denominator, which now might look like this:

```python
def numer0(self):
    return self._numer
numer = property(numer0) # numer is now a descriptor
```

Properties

- To help class implementors control syntax, Python provides an egregiously general mechanism known as descriptors.
- An attribute of a class that is set to a descriptor object behaves differently from usual when selected.
- Descriptors, in their full details, are wonders to behold, so we’ll stick with simple uses.
- If we define

```python
def numer0(self):
    return self._numer
numer = property(numer0) # numer is now a descriptor
```

```python
>>> 3 + rational(1,2)
7/2
```

Properties (contd.)

- The usual shorthand for writing this is to use property as a decorator:

```python
@property
def numer(self):
    """My numerator in lowest terms.""
    return self._num
```

```python
def __init__(self, low, high):
    self._low, self._high, self._x = low, high, low
    self._getx = _getx
    self._setx = _setx
```

```python
x = RestrictedInt(1, 10)
print(x)
```

```python
7
```

```python
x._setx(12)
print(x)
```

```python
12
```