EXCEPTIONS, CALCULATOR 10

COMPUTER SCIENCE 61A

November 5th, 2012

We are beginning to dive into the realm of interpreting computer programs - that is, writing programs that understand programs. In order to do so, we'll have to examine programming languages in-depth. The **Calculator** language, a subset of Scheme, will be the first of these examples.

In today's discussion, we'll be looking at implementing Calculator using regular Python. We'll also take a look at **Exceptions**, a mechanism for handling unexpected execution - quite common when handling user input.

1 Exceptions

Up to this point in the semester, we have assumed that the input to our functions are always correct, and thus have not done any error handling. However, functions can often have large domains, and we want our functions to handle erroneous input gracefully. This is where exceptions come in.

Exceptions provide a general mechanism for adding error-handling logic to programs. **Raising an exception** is a technique for interrupting the normal flow of execution in a program, signaling that some exceptional circumstance has arisen.

An **exception** is an object instance of a class that inherits, either directly or indirectly, from the `BaseException` class. The following is an example of how to raise an exception:

```
>>> raise Exception('An error occurred')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
Exception: An error occurred
```

Notice how the string 'An error occurred' is an argument to the `Exception` object being created, and the string is part of what Python prints out in response to the exception being raised.

If the exception is raised while with a `try` statement, then the interpreter will immediately look for an `except` statement that handles the type of exception being raised. `try` and `except` statements allow programs to respond to unexpected arguments and other errors gracefully, rather than terminating entirely.

Here's how to structure `try` and `except` statements:

```
try:
   <try suite>
except <exception class> as <name>:
   <except suite>
      except <exception class> as <name>:
      <except suite>
```

1.1 Questions

1. Fill in all the blanks to produce the desired output:

```
>>> try:
        x = ______
    except ________________ as ___:
        print(handling a , type(e))
        x = ____
handling a <class ZeroDivisionError>
>>> x
9001
```

2. Write the function `safe_square` that uses exceptions to print 'Incorrect argument type' when anything other than an `int` or `float` class is given as an argument. Otherwise, `safe_square` should multiply the argument by itself. A useful fact is that a `TypeError` is raised when * is given incorrect arguments.

```
def safe_square(x):
```

3. Predict the output of each of the following lines, assuming `safe_square` is implemented as described in the previous question.

```
>>> safe_square('hello')

>>> safe_square('hello * 5)

>>> safe_square('hello' * 'hello')

>>> safe_square(1 * 2.5)

>>> safe_square(1/ 0)
```

2 Calculator

For now, our Calculator language will be a Scheme-syntax language that can handle the four basic arithmetic operations. These operations can be nested and can take varying numbers of arguments. Here's a couple examples of Calculator in action:

```
> (+ 2 2)
4

> (- 5)
-5

> (* (+ 1 2) (+ 2 3))
15
```

Our goal now is to write an interpreter for this Calculator language. The job of an interpreter is, given an expression, evaluate its meaning. So let's talk about expressions.

2.1 Representing Expressions

There are two kinds of expressions. A **call expression** is a Scheme list - the first element is the operator, and each subsequent element is an operand. A **primitive expression** is an operator symbol or number. When we type a line at the Calculator prompt and hit enter, we've just sent an expression to the interpreter.

To represent Scheme lists in Python, we'll be using `Pair` objects. The class definition is below. Note the usage of exceptions:

```
class Pair(object):
```

```
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __len__(self):
        n, second = 1, self.second
        while isinstance(second, Pair):
                n += 1
                second = second.second
        if second is not nil:
                raise TypeError("length attempted on improper list")
        return n

     def __getitem__(self, k):
        if k < 0:
                raise IndexError("negative index into list")
        j, y = 0, self
        while j < k:
                if y.second is nil:
                        raise IndexError("list index out of bounds")
                elif not isinstance(y.second, Pair):
                        raise TypeError("ill-formed list")
                j, y = j + 1, y.second
        return y.first

    def map(self, fn):
        """Returns a Scheme list after mapping Python function
        fn over self."""
        mapped = fn(self.first)
        if self.second is nil or isinstance(self.second, Pair):
                return Pair(mapped, self.second.map(fn))
        else:
                raise TypeError("ill-formed list")

   def to_py_list(self):
        """Returns a Python list containing the elements of this
        Scheme list."""
        y, result = self, [ ]
        while y is not nil:
            result += [y.first]
            if not isinstance(y.second, Pair) or y.second is not nil:
```

```
                raise TypeError("ill-formed list")
            y = y.second
        return result

class nil(object):
    """The empty list"""

    def __len__(self):
        return 0

    def map(self, fn):
        return self

nil = nil() #nil now refers to a single instance of nil class
```

2.2 Questions

1. Translate the following Python representation of Calculator expressions into the proper Scheme-syntax:

```
>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))

>>>Pair('+', Pair('1', Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```

2. Translate the following Calculator expression into calls to the `Pair` constructor.

```
> (+ 1 2 (- 3 4))
```

2.3 Evaluation

So what is evaluation? Evaluation discovers the form of an expression and executes a corresponding evaluation rule.

Primitive expressions are evaluated directly. Call expressions are evaluated recursively: (1) Evaluate each operand expression, (2) Collect their values as a list of arguments, and (3) **Apply** the named operator to the argument list.

Here's `calc_eval`:

```
def calc_eval(exp):
    if not isinstance(exp, Pair): #expression is primitive
        return exp
    else:
```

```
        operator, operands = exp.first, exp.second
        args =  operands.map(calc_eval).to_py_list()
        return calc_apply(operator, args)
```

As you can see, all we've done is follow the rules of evaluation outlined above. If the expression is primitive (i.e. not a Scheme list), simply return it. Else, evaluate the operands and apply the operator to the evaluated operands.

How do we apply the operator? We'll use `calc_apply`, with dispatching on the operator name:

```
def calc_apply(operator, args):
    if operator == '+':
        return sum(args)
    elif operator == '-':
        if len(args) == 1:
            return -args[0]
        else:
            return sum(args[0], [-args for args in args[1:]])
    elif operator == '*':
        return reduce(mul, args, 1)
```

Depending on what the operator is, we can match it to a corresponding Python call. Each conditional clause above handles the application of one operator.

Something very important to keep in mind: `calc_eval` deals with **expressions**, `calc_apply` deals with **values**.

2.4 Questions

1. Suppose we typed each of the following expressions into the Calculator interpreter. How many calls to `calc_eval` would they each generate? How many calls to `calc_apply`?

   ```
   > (+ 2 4 6 8)

   > (+ 2 (* 4 (- 6 8)))
   ```

2. The − operator will fail if given no arguments. Add error handling to raise an exception when this situation is encountered (the type of exception is unimportant).

3. We also want to be able to perform division, as in `(/ 4 2)`. Supplement the existing code to handle this. If division by 0 is attempted, or if there are less than 2 arguments supplied, you should raise an exception (the type of exception is unimportant).

4. Alyssa P. Hacker and Ben Bitdiddle are also tasked with implementing the `and` operator, as in `(and (= 1 2) (< 3 4))`. Ben says this is easy: they just have to follow the same process as in implementing `*` and `/`. Alyssa is not so sure. Who's right?

5. Now that you've had a chance to think about it, you decide to try implementing `and` yourself. You may assume the conditional operators (e.g. <, >, =, etc) have already been implemented for you.