Exceptions

Up to this point in the semester, we have assumed that the input to our functions are always correct, and have thus 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 within 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 is how to structure `try` and `except` statements:

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

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 instance of the `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 question 2.

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

>>> safe_square('hello' * 5)

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

>>> safe_square(1 * 2.5)

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

4. Consider the following:

```
>>> def hello_append(lst):
        try:
            if type(lst) != list:
                raise HelloAppendError(lst)
            lst.append('hello')
        except HelloAppendError as e:
            print('type of arg was', type(e.contents))
```

Create a `HelloAppendError` class that would create the following output given the definition of `hello_append` above:

```
>>> x = 5
>>> hello_append(x)
type of arg was <class 'int'>
```

When writing big programs, it's usually a good idea to have robust error handling logic that details the state of the program so that you, as the programmer, can debug your code more effectively. Additionally, by gracefully responding to errors, your program could potentially continue to run even when errors or unexpected input occur. In this class, we are primarily interested in using exceptions as part of the error handling logic that we will implement when we start creating interpreters in Python.

Orders of Growth

(Notes developed based on those of George Wang, Jon Kotker, Seshadri Mahalingam, Chung Wu, and Justin Chen)

When we talk about the efficiency of a procedure (at least for now), we're often interested in how much more expensive it is to run the procedure with a larger input. That is, as the size of the input grows, how do the speed of the procedure and the space its process occupies grow?

For expressing all of these, we use what is called the Big-Theta notation. For example, if we say the running time of a procedure *foo* is in $\Theta(n^2)$, we mean that the time it takes to process the input grows as the square of the size of the input. More generally, we can say that *foo* is in some $\Theta(f(n))$ if there exist some constants *k1* and *k2* such that and some constants *c1* and *c2* such that,

*k1 * f(n) < running time of foo for some n > c1, and*
*k2 * f(n) > running time of foo for some n > c2*

To prove, then, that *foo* is in $\Theta(f(n))$, we only need to find constants *k1* and *k2* where the above holds. Fortunately for you, in 61A, we're not that concerned with rigor, and you probably won't need to know exactly how to do this (you will get the painful details in 61B!) What we want you to have in 61A, then, is the intuition of guessing the orders of growth for certain procedures.

Kinds of Growth

Here are some common ones: $\Theta(1)$ – constant time (takes the same amount of time regardless of input size); $\Theta(\log n)$ – logarithmic time; $\Theta(n)$ – linear time; $\Theta(n^2)$, $\Theta(n^3)$, etc – polynomial time; $\Theta(2^n)$ – exponential time ("intractable"; these are really, really horrible).

Orders of Growth in Time

"Time", for us, basically refers to the number of recursive calls or the number of times the suite of a `while` or `for` loop executes. Intuitively, the more recursive calls we make, the more time it takes to execute the function.

- If the function contains only primitive procedures like + or *, then it is constant time – $\Theta(1)$.
- If the function is recursive, you need to:
    a. Count the number of recursive calls there will be given input n
    b. Count how much time it takes to process the input per recursive call

The answer is usually the product of the above two. For example, given a fruit basket with 10 apples, how long does it take for me to process the whole basket? Well, I'll recursively call my eat procedure which eats one apple at a time (so I'll call the procedure 10 times). Each time I eat an apple, it takes me 30 minutes. So the total amount of time is just 30*10 = 300 minutes!

- If the function contains calls of helper functions that are not constant-time, then you need to take the orders of growth of the helper functions into consideration as well. In general, how much time the helper function takes would be factored.
- When we talk about orders of growth, we don't really care about constant factors. So if you get something like $\Theta(1000000n)$, this is really $\Theta(n)$. We can also usually ignore lower-order terms. For example, if we get something like $\Theta(n^3 + n^2 + 4n + 399)$, we take it to be $\Theta(n^3)$.

Questions:

Give the order of growth in time for the following functions:

1.
```
def sum_of_factorial(n):
      if n == 0:
            return 1
      else:
            return factorial(n) + sum_of_factorial(n-1)
```

2.
```
def fib_iter(n):
      prev, curr, i = 0, 1, 1
      while i < n:
            prev, curr = curr, prev + curr
            i += 1
      return curr
```

3.
```
def mod_7(n):
      if n % 7 == 0:
            return 0
      else:
            return 1 + mod_7(n-1)
```

4. Given:

```
def bar(n):
      if n % 2 == 1:
            return n + 1
      return n

def foo(n):
      if n < 1:
            return 2
      if n % 2 == 0:
            return foo(n-1) + foo(n-2)
      else:
            return 1 + foo(n-2)
```

What is the order of growth of `foo(bar(n))`?

Calculator

We are beginning to dive into the realm of *interpreting* computer programs. In order to do so, we'll have to examine some new programming languages. The *Calculator* language, invented for this class, is the first of these examples.

Our Calculator language is actually nearly identical to Python, except it only handles arithmetic operations - add, sub, mul, div, etc. Also unlike Python's arithmetic operators, we'll let Calculator's operators each take an arbitrary number of arguments.

Here's a few examples of Calculator in action:

```
calc> 6
6

calc>mul()
1

calc>add(1, mul(3, sub(3, 7)))
-11
```

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

Representing Expressions

When we type a line at the Calculator prompt and hit enter, we've just sent an expression to the interpreter. We can represent an Expression as an object:

```
class Exp(Object):
      def __init__(self, operator, operands):
            self.operator = operator
            self,operands = operands
      def __repr__(self):
            return 'Exp({0}, {1})'.format(repr(self.operator), repr(self.operands)))
      def __str__(self):
            operand_strs = ', '.join(map(str, self.operands))
            return '{0}({1})'.format(self.operand, operand_strs)
```

Don't worry about what `repr` and `str` do for an Exp right now, the most important thing to note is that every Exp by its operator and operands. Also another thing to note: an Exp's operands are always themselves Exps as well.

Example: If I wanted to represent the Calculator expression `add(2, 3)`, I would do this by calling the `Exp` constructor as follows: `Exp('add, [2, 3])`.

Our Calculator language is only concerned with two kinds of expressions: numbers, which are *self-evaluating* expressions, and call expressions, which involve an operator acting on some number of arguments, each of which are expressions.

What does it mean for a number to be self-evaluating? If we evaluate an expression that is a number, the value of the expression (i.e. the result returned by evaluating it) is that number. Simple!

What about evaluating call expressions? We can follow this straightforward two-step process.

1. Evaluate the operands.
2. Apply the operator, to the arguments (the values of the operands)

(Sound familiar?)

Using this two-step process, we can interpret any Calculator expression. The first step will be handled by a function called `calc_eval`, and the second step will be handled by a function called `calc_apply`.

Here's `calc_eval`:

```
def calc_eval(exp):
      if type(exp) in (int, float): # if the expression is a number
            return exp
      else:
            arguments = list(map(calc_eval, exp.operands)) #evaluate the operands
            return calc_apply(exp.operator, arguments) #apply the function to operands
```

As you can see, all we've done is follow the rules of evaluation outlined above. If the expression is a number, return it. Else, evaluate the operands and apply the operator the evaluated operands.

How do we apply the operator? With `calc_apply`:

```
def calc_apply(operator, args):
      if operator in ('add', '+'):
            return sum(args)
      if operator in ('sub', '-'):
            if len(args) == 0:
                  raise TypeError(operator + ' requires at least 1 argument')
            if len(args) == 1:
                  return -args[0]
            return sum(args[0], [-args for args in args[1:]])
      if operator in ('mul', '*'):
            return reduce(mul, args, 1)
```

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

Something very important but may not have been obvious: calc_exp deals with *expressions*, calc_apply deals with *values*.

Questions

Let's say we want to make the variable a contain the object representation of the Calculator expression: `add(4, 5, mul(3, 2))`. Fill in the blank:

```
>>>a = Exp(_____________________________________________________________)
```

Suppose we typed the following expression into the calc interpreter:

```
calc> add(1, mul(3, sub(3, 7)))
```

How many calls to `calc_eval` does this generate? How many calls to `calc_apply`?

The above implementation of the `calc_exp` and `calc_apply` does not handle calls to `div`. Add to them so we can evaluate Calculator `div` calls:

Example:

```
calc>div(8, 4)
2
```

If a div expression is not given exactly two arguments, you should raise a TypeError.