Programming for Science
Possible answers for workshop 8

Here are possible answers to the exercises in the Workshop 8. Usually these are not the only way of solving the problem, but they are relatively straightforward and clean ways.

1. Code for this was given and demonstrated in lectures, but for completeness here is a succinct version.

```
def factorial(n):
    """
    Compute the factorial of n (for integer n >= 0).
    """
    if not isinstance(n, int) or n < 0:
        raise ValueError('invalid argument for factorial:')

    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

2. Here is code to recursively calculate the sum of the integers to $0, \ldots, n$.

```
def rsum(n):
    """
    Recursively calculate the sum of the first n integers
    """
    if not isinstance(n, int) or n < 0:
        raise ValueError("argument is not a non-negative integer")

    if n == 0:
        return 0
    else:
        return n + rsum(n-1)

print rsum(10)
```

3. Code to calculate the sum of a list by adding the two ends and recursively summing the middle follows. Note that to take care of odd and even length lists we need two base cases.

```
def lsum(L):
    """
    Recursively calculate the sum of the list L
    """
    if not isinstance(L, list):
        raise ValueError("argument is not a list")

    n = len(L)
    if n == 0:
```

```
        return 0
    elif n == 1:
        return L[0]
    else:
        return L[0] + lsum(L[1:-1]) + L[-1]

print lsum([])
```

Note that recursively calculating a sum is definitely **not** an efficient way of finding the sum. An iterative calculation, that doesn't involve additional function calls is much more efficient.

4. Recursive calculation of the Fibonacci numbers is not very efficient either, and like this, many quantities that can be calculated recursively can be turned into a more efficient iterative calculation. Nonetheless, here is code to calculate the Fibonacci numbers, which shows some of the power of recursion.

```
def Fib(n):
    """
    Recursively calculate the nth Fibonacci number for non-negative n
    """
    if not isinstance(n, int) or n < 0:
        raise ValueError("argument is not a non-negative integer")

    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return Fib(n-1) + Fib(n-2)

print Fib(30)  # Prints 6765
```

This is particularly inefficient as it recursively calls `Fib` to calculate `Fib(n-1)` and `Fib(n-2)`, but `Fib(n-1)` itself must call `Fib(n-2)` (and `Fib(n-3)`). In fact to calculate `Fib(20)` requires the function to be called 21891 times!

Here is a more efficient version that uses "memoisation". To avoid having to pass the storage dictionary `memory`, the `Fib_memo` function is hidden by "wrapping" it in the function `Fib`.

```
def Fib_memo(n, memory):
    """
    Recursively calculate the nth Fibonacci number for non-negative n.
    Uses the dictionary memory for storing intermediate results
    """
    if memory.has_key(n):
        return memory[n]
    else:
        f = Fib_memo(n-1, memory) + Fib_memo(n-2, memory)
        memory[n] = f
        return f
```

```
def Fib(n):
    """
    Recursively calculate the nth Fibonacci number for non-negative n.
    This function just hides the memory dictionary and does the error chec
    """
    if not isinstance(n, int) or n < 0:
        raise ValueError("argument is not a non-negative integer")

    return Fib_memo(n, {0:0, 1:1})

print Fib(30)  # Prints 832040
```

The memoised code is much faster than the original: one my computer the original takes about 1.3s to calculate `Fib(30)`, but only 0.036s with the memoised version.

5. Unlike the Fibonacci numbers, the Koch curve much easier to draw recursively than iteratively. Here's some possible code.

```
from TurtleWorld import *

def koch(t, n=4, L=50):
    """
    Recursively draw a Koch curve using turtle t. The basic length of the
    curve is L (default 50) and the recursion continues for n
    levels.
    """
    if n == 0:                # End of the recursion, just draw the line of
        fd(t, L)
    else:
        L = L/3.0              # Length of segments next level down
        n = n - 1
        koch(t, n, L)
        lt(t, 60)
        koch(t, n, L)
        rt(t, 120)
        koch(t, n, L)
        lt(t, 60)
        koch(t, n, L)



world = TurtleWorld()
world.delay = 0                          # Make the turtle run fast!
bob = Turtle()

pu(bob)
bk(bob, 150)
pd(bob)
```

```
koch(bob, 5, 200)

wait_for_user()
```