1 Recursive functions

Recall from the previous handout that every recursive solution has this structure:

(1) **base case**, where the problem is simple enough to be solved directly

(2) **recursive case**, which has three components

  (a) **divide** problem into one or more simpler or smaller parts of the problem,

  (b) **call** the function (recursively) on at least one part, and

  (c) **combine** the solutions of the parts into a solution for the problem

2 Hand tracing recursive function call

This recursive function is called `m` for mystery. It does not compute anything interesting... but it's a good function on which to practice hand-tracing the recursion. In other words, execute this function by hand (no computer) and try to figure out what it will print.

```
def m(n):
    if n < 0:
        m(-n)
    elif n < 10:
        print n,
    else:
        m(n / 10)
        digit = n % 10
        print ",", digit % 3,
```

Hand-tracing a recursive function call can be difficult, but *it is very important you understand how python executes a recursive function*. Using indentation can help, as in these examples.

```
m(743):
    m(74):
        m(7):
            print 7
        digit = 4
        print , 1
    digit = 3
    print , 0
```

Therefore, `m(743)` prints `7, 1, 0`.

3 Writing recursive functions

If you are asked to solve a problem using recursion, follow these steps:

1) (Doc) Write the docstring first... trust me, it helps!
2) (Base) Figure out the base case: think of inputs where the answer is easy. If the input is a number, this is often 0 or 1. If it's a list or a string, this is often the empty string or list, or sometimes a string/list with just one letter/item.
3) For the recursive case:
   a) (Divide) Break the problem into two pieces: a piece you can "handle" easily and another piece which is a *smaller* version of the *same* problem.
   b) (Recurse) Follow the "have faith" principle. Make a recursive call and have faith the function will work correctly. This is where the docstring is helpful.
   c) (Combine) Take the result of the recursive call and the solution to the other smaller piece and combine them into a complete solution.

Exercises

Some solutions are presented in class and also included in the moodle version of this handout.

1. Write a recursive function `count_e` that takes a string `s` and returns the number of times `'e'` occurs in `s`.

Solution:

```
def count_e(s):
    '''(str) -> int
    Returns the number of times 'e'
    occurs in s.
    >>> count_e('abc')
    0
    >>> count_e('bees knees')
    4
    '''
    if len(s) == 0:
        return 0
    elif s[0] == 'e':
        return 1 + count_e(s[1:])
    else:
        return count_e(s[1:])
```

2. Write a recursive function `reverse` that takes a string `s` and returns the string in reverse.

Solution:

```
def reverse(s):
    '''(str) -> str
    Returns the reverse of s.
    >>> reverse('abc')
    'cba'
    >>> reverse('bees')
    'seeb'
    '''
    if len(s) == 0:
        return ''
    else:
        first = s[0]
        rest = s[1:]
        rev_of_rest = reverse(s[1:])
        return rev_of_rest + first
```

3. Write a recursive function `no_duplicate_e` that takes a string `s` and returns the string after replacing all duplicate occurrences of `'e'` with a single `'e'`. So `no_duplicate_e('eeee zeee')` returns `'e ze'`.

Solution:

```
def no_duplicate_e(s):
    '''(str) -> str
    Returns s after replacing all duplicate occurrences
    of 'e' with a single 'e'.
    >>> no_duplicate_e('abc')
    'abc'
    >>> no_duplicate_e('free bees pleez')
    'fre bes plez'
    '''
    if len(s) <= 1:      # s can't have duplicates
        return s
    else:                # s has at least one character
        first = s[0]
        rest = s[1:]
        no_dupes_rest = no_duplicate_e(rest)
        # don't add first to result if it will
        # create a duplicate 'e'
        if first == 'e' and no_dupes_rest[0] == 'e':
```

```
            return no_dupes_rest
        else:
            return first + no_dupes_rest
```

4. Write a recursive function `mirror` that takes a string `s` and returns the string in "mirrored", as in `mirror('ah')` returns `'ahha'`.

Solution:

```
def mirror(s):
    '''(str) -> str
    Returns the mirror of s.
    >>> mirror('abc')
    'abccba'
    >>> mirror('bees')
    'beesseeb'
    '''
    if len(s) == 0:
        return ''
    else:
        first = s[0]
        rest = s[1:]
        mirror_rest = mirror(s[1:])
        return first + mirror_rest + first
```

5. Write a recursive function `duplicate` that takes a string `s` and returns the string with each letter duplicated, as in `duplicate('ah')` returns `'aahh'`.

Solution:

```
def duplicate(s):
    '''(str) -> str
    Returns s with each letter duplicated.
    >>> duplicate('abc')
    'aabbcc'
    >>> duplicate('bees')
    'bbeeeess'
    '''
    if len(s) == 0:
        return ''
    else:
        first = s[0]
```

```
        rest = s[1:]
        dup_rest = duplicate(s[1:])
        return first*2 + dup_rest
```

Definition of recursion adapted from NIST, `http://xlinux.nist.gov/dads//HTML/recursion.html`. Mystery function adapted from Stuart Reges.