1. Implement the functions `max_product`, which takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```python
def max_product(lst):
    """Return the maximum product that can be formed using lst without using any consecutive numbers
    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    ""

    Solution:
    if lst == []:
        return 1
    elif len(lst) == 1:
        return lst[0]
    else:
        return max(max_product(lst[1:]), lst[0]*max_product(lst[2:]))
```
2. (Fall 2012) Draw the environment diagram for the following code:

```python
def horse(mask):
    horse = mask
def mask(horse):
    return horse
return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```

Solution:

---

![Environment Diagram](image)

---
3. Draw the environment diagram for the following code:

doug = "ni"

def cat(dog):
    def rug(rat):
        doug = lambda doug: rat(doug)
        return doug
    return rug(dog)("ck")

cat(lambda rat: doug + rat)
4. Define a function `foo` that takes in a list `lst` and returns a new list that keeps only the even-indexed elements of `lst` and multiplies each of those elements by the corresponding index.

```python
def foo(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> foo(x)
    [0, 6, 20]
    """
    return [_________________________________________________
    
    Solution:
    
    ```python
    return [i * lst[i] for i in range(len(lst)) if i % 2 == 0]
    ```

5. Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner (M-1, N-1). The insect is only capable of moving right or up. Write a function `paths` that takes a grid length and width and returns the number of different paths the insect can take from the start to the goal. (There is a closed-form solution to this problem, but try to answer it procedurally using recursion.)

```python
def paths(m, n):
    """
    >>> paths(2, 2)
    2
    >>> paths(117, 1)
    1
    """
    if m == 1 or n == 1:
        return 1
    return paths(m - 1, n) + paths(m, n - 1)
```

Solution:
6. An **expression tree** is a tree that contains a function for each non-leaf label, which can be either `'+` or `'*'`. All leaves are numbers. Implement `eval_tree`, which evaluates an expression tree to its value. You may want to use the functions `sum` and `prod`, which take a list of numbers and compute the sum and product respectively.

```python
def eval_tree(tree):
    """Evaluates an expression tree with functions as root
    >>> eval_tree(tree(1))
    1
    >>> expr = tree('*', [tree(2), tree(3)])
    >>> eval_tree(expr)
    6
    >>> eval_tree(tree('+', [expr, tree(4), tree(5)]))
    15
    """
    if is_leaf(tree):
        return label(tree)
    args = [eval_tree(subtree) for subtree in branches(tree)]
    if label(tree) == '+':
        return sum(args)
    elif label(tree) == '*':
        return prod(args)
```

**Solution:**
```python
if is_leaf(tree):
    return label(tree)
args = [eval_tree(subtree) for subtree in branches(tree)]
if label(tree) == '+':
    return sum(args)
elif label(tree) == '*':
    return prod(args)
```
7. (Spring 2015) Implement the memory function, which takes a number \( x \) and a single-argument function \( f \). It returns a function with a peculiar behavior that you must discover from the doctests. You may only use names and call expressions in your solution. You may not write numbers or use features of Python not yet covered in the course.

```python
def memory(x, f):
    """Return a higher-order function that prints its memories."
    >>> f = memory(3, lambda x: x)
    >>> f = f(square)
    3
    >>> f = f(double)
    9
    >>> f = f(print)
    6
    >>> f = f(square)
    3
    None
    """
    def g(h):
        print(f(x))
        return ________________
    return g
```

**Solution:**
```python
def memory(x, f):
    def g(h):
        print(f(x))
        return memory(x, h)
    return g
```