1. For each row below, fill in the blanks in the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and expressions may affect later expressions.

```python
>>> cats = [1, 2]
>>> dogs = [cats, cats.append(23), list(cats)]
>>> cats

>>> dogs[1] = list(dogs)
>>> dogs[1]

>>> dogs[0].append(2)
>>> cats

>>> dogs[2].extend([list(cats).pop(0), 3])
>>> dogs[3]

>>> dogs
```
1. (Fall 2012) Draw the environment diagram.

```python
def box(a):
    def box(b):
        def box(c):
            nonlocal a
            a = a + c
        return (a, b)
    return box

gift = box(1)
return (gift(2), gift(3))

box(4)
```
3 The Gift & The Recurse

1. The quicksort sorting algorithm is an efficient and commonly used algorithm to order the elements of a list. We choose one element of the list to be the pivot element and partition the remaining elements into two lists: one of elements less than the pivot and one of elements greater than the pivot. We recursively sort the two lists, which gives us a sorted list of all the elements less than the pivot and all the elements greater than the pivot, which we can then combine with the pivot for a completely sorted list.

First, implement the quicksort_list function. Choose the first element of the list as the pivot. You may assume that all elements are distinct.

```python
def quicksort_list(lst):
    """
    >>> quicksort_list([3, 1, 4])
    [1, 3, 4]
    """

    if _______________________________________________________:
        _______________________________________________________

    pivot = lst[0]

    less = ________________________________________________

    greater = _____________________________________________

    return ________________________________________________
```

2. We can also use quicksort to sort linked lists! Implement the quicksort_link function, without constructing additional Link instances.

You can assume that the extend_links function is already defined. It takes two linked lists and mutates the first so that it points to the second.

```python
>>> 11, 12 = Link(1, Link(2)), Link(3, Link(4))
>>> 13 = extend_links(11, 12)
>>> 13
Link(1, Link(2, Link(3, Link(4))))
>>> 11 is 13
True
```
def quicksort_link(link):
    ""
    >>> s = Link(3, Link(1, Link(4)))
    >>> quicksort_link(s)
    Link(1, Link(3, Link(4)))
    ""

    if ____________________________________________________:
        return link

    pivot, ______ = ________________________________________

    less, greater = _________________________________________

    while link is not Link.empty:
        curr, rest = link, link.rest

        if __________________________________________________:
            _______________________________________

        else:

            _______________________________________

        link = _______________________________________

    less = ______________________________________________

    greater = _____________________________________________

    ___________________________________________________

    return ______________________________________________
4 Can You Take Me Higher?

1. (Fall 2013) Fill in the blanks in the implementation of paths, which takes as input two positive integers \( x \) and \( y \). It returns the number of ways of reaching \( y \) from \( x \) by repeatedly incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9.

```python
def inc(x):
    return x + 1

def double(x):
    return x * 2

def paths(x, y):
    """Return the number of ways to reach \( y \) from \( x \) by repeated incrementing or doubling.
    >>> paths(3, 5) # inc(inc(3))
    1
    >>> paths(3, 6) # double(3), inc(inc(inc(3)))
    2
    >>> paths(3, 9) # E.g. inc(double(inc(3)))
    3
    >>> paths(3, 3) # No calls is a valid path
    1
    """
    if x > y:
        return ________________________________
    elif x == y:
        return ________________________________
    else:
        return ________________________________
```
2. (Fall 2013) Fill in the blanks in the implementation of `pathfinder`, a higher-order function that takes an increasing function \( f \) and a positive integer \( y \). It returns a function that takes a positive integer \( x \) and returns whether it is possible to reach \( y \) by applying \( f \) to \( x \) zero or more times. For example, 8 can be reached from 2 by applying `double` twice. A function \( f \) is increasing if \( f(x) > x \) for all positive integers \( x \).

```python
def pathfinder(f, y):
    """
    >>> f = pathfinder(double, 8)
    >>> {k: f(k) for k in (1, 2, 3, 4, 5)}
    {1: True, 2: True, 3: False, 4: True, 5: False}
    >>> g = pathfinder(inc, 3)
    >>> {k: g(k) for k in (1, 2, 3, 4, 5)}
    {1: True, 2: True, 3: True, 4: False, 5: False}
    """

    def find_from(x):
        while __________________________________________:
            __________________________________________
        return ______________________________________

3. Write a generator function that yields functions that are repeated applications of a one-argument function \( f \). The first function yielded should apply \( f \) 0 times (the identity function), the second function yielded should apply \( f \) once, etc.

```python
def repeated(f):
    """
    >>> [g(1) for _, g in ...
        zip(range(5), repeated(double))]
    [1, 2, 4, 8, 16]
    """

    g = ______________________________

    while True:
        __________________________________
        __________________________________
```
4. Ben Bitdiddle proposes the following alternate solution. Does it work?

```python
def ben_repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = lambda x: f(g(x))
```

5. Slim Shady

1. Implement `widest_level`, which takes a `Tree` instance and returns the elements at the depth with the most elements.

```python
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...          Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    levels = []
x = [t]

    while __________________________________________________________:
        __________________________________________________________
        __________ = sum(__________________________, [])
    return max(levels, key=______________________________)
```
6 Scheming With a Broken Heart

1. Consider the following Scheme tree data abstraction.

(define (make-tree entry children) (cons entry children))
(define (entry tree) (car tree))
(define (children tree) (cdr tree))
(define tree 'below-example)

; +--------+--------+
; | | |
; 6 7 2
; +--+--+ | +--+--+
; | | | | |
; 9 8 1 6 4
; |
; |
; 3

Write a procedure `tree-sums` that takes a tree of numbers (like the one above) and outputs a list of sums from following each possible path from root to leaf.

**Hint:** You may find the `flatten` procedure helpful.

(define (flatten lst)
  (cond ((null? lst) nil)
        ((list? (car lst)) (append (flatten (car lst)) (flatten (cdr lst))))
        (else (cons (car lst) (flatten (cdr lst))))))

(define (tree-sums tree)
  (if _________________________________________________________
    _________________________________________________________
    (map (lambda (x) _____________________________________)
         ______________________________________________________))

scm> (flatten '(0 (1) ((2)) (3 ((4)))))
(0 1 2 3 4)
scm> (tree-sums tree)
(20 19 13 16 11)
1. Implement the append-stream procedure, which takes in two streams and returns a stream with the two streams concatenated. (Note that if the first stream is infinite, the result will not contain any elements from the second stream.)
   (define (append-stream s1 s2)

2. Now implement subset-stream, which takes in a normal Scheme list and returns a stream with every possible subset of that Scheme list.
   (define (subset-stream lst)
8 Turning Tables

1. You’re trying to re-organize your music library! The table tracks below contains song titles and the corresponding album. Create another table tracklist with two columns: the album and a comma-separated list of all songs from that album.

```sql
create table tracks as
    select "Human" as title, "The Definition" as album union
    select "Simple and Sweet", "The Definition" union
    select "Paper Planes", "Translations Through Speakers";

create table tracklist as
    with
        songs(album, total) as ( )
            ),
                ),
                    ),
                        )
                    )
            )
        select
            where
            sqlite3> select * from tracklist order by album;
            The Definition|Human, Simple and Sweet
            Translations Through Speakers|Paper Planes
```