

1 Mutation

- 1.1 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.

```
>>> 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
```

2 Recursion

- 2.1 Write a procedure `merge(s1, s2)` which takes two sorted (smallest value first) lists and returns a single list with all of the elements of the two lists, in ascending order. Use recursion.

Hint: If you can figure out which list has the smallest element out of both, then we know that the resulting merged list will have that smallest element, followed by the merge of the two lists with the smallest item removed. Don't forget to handle the case where one list is empty!

```
def merge(s1, s2):
    """ Merges two sorted lists
    >>> merge([1, 3], [2, 4])
    [1, 2, 3, 4]
    >>> merge([1, 2], [])
    [1, 2]
    """
    if _____:

        return s2

    elif _____:

        return s1

    elif _____:

        return _____

    else:

        return _____
```

2.2 Implement a function to solve the subset sum problem: you are given a list of integers and a number k . Is there a subset of the list that adds up to k ?

```
def subset_sum(lst, k):
    """
    >>> subset_sum([], 0)
    True
    >>> subset_sum([], 4)
    False
    >>> subset_sum([2, 4, 7, 3], 5)      # 2 + 3 = 5
    True
    >>> subset_sum([1, 9, 5, 7, 3], 2)
    False
    >>> subset_sum([1, 1, 5, -1], 3)
    False
    """

    if _____:

        return True

    elif _____:

        return False

    else:

        return _____
```

3 Streams

- 3.1 Write a function `merge` that takes 2 sorted streams `s1` and `s2`, and returns a new sorted stream which contains all the elements from `s1` and `s2`. Assume that both `s1` and `s2` have infinite length.

```
(define (merge s1 s2)
```

```
  (if -----  
      -----  
      -----))
```

- 3.2 (Adapted from Fall 2014) Implement `cycle` which returns a stream repeating the digits 1, 3, 0, 2, and 4, forever. Write `cons-stream` only once in your solution!

Hint: `(3+2) % 5 == 0`.

```
(define (cycle start)
```

```
  -----)
```

4 Generators

- 4.1 Implement `accumulate`, which takes in an `iterable` and a function `f` and yields each accumulated value from applying `f` to the running total and the next element.

```
from operator import add, mul
```

```
def accumulate(iterable, f):
    """
    >>> list(accumulate([1, 2, 3, 4, 5], add))
    [1, 3, 6, 10, 15]
    >>> list(accumulate([1, 2, 3, 4, 5], mul))
    [1, 2, 6, 24, 120]
    """
    it = iter(iterable)
```

```
for _____:
```

6 Final Review

- 4.2 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.

```
def repeated(f):
```

```
    """
```

```
    >>> double = lambda x: 2 * x
```

```
    >>> funcs = repeated(double)
```

```
    >>> identity = next(funcs)
```

```
    >>> double = next(funcs)
```

```
    >>> quad = next(funcs)
```

```
    >>> oct = next(funcs)
```

```
    >>> quad(1)
```

```
    4
```

```
    >>> oct(1)
```

```
    8
```

```
    >>> [g(1) for _, g in
```

```
    ... zip(range(5), repeated(lambda x: 2 * x))]
```

```
    [1, 2, 4, 8, 16]
```

```
    """
```

```
g = -----
```

```
while True:
```

```
-----
```

```
-----
```

- 4.3 Ben Bitdiddle proposes the following alternate solution. Does it work?

```
def ben_repeated(f):
```

```
    g = lambda x: x
```

```
    while True:
```

```
        yield g
```

```
        g = lambda x: f(g(x))
```

5 SQL

- 5.1 You're starting a new job at an animal shelter, and you've been tasked with keeping track of all the cats that are up for adoption!

We'll start with an empty table:

```
CREATE TABLE cats(name, weight DEFAULT 1, notes DEFAULT "meow");
```

- (a) What would SQL display?

```
sqlite> INSERT INTO cats(name) VALUES ("Tom"), ("Whiskers");
sqlite> SELECT * FROM cats;
```

```
sqlite> INSERT INTO cats VALUES
...> ("Mittens", 2, "Actually likes shoes"),
...> ("Rascal", 4, "Prefers to associate with dogs"),
...> ("Magic", 2, "Expert at card games");
sqlite> SELECT * FROM cats ORDER BY weight, name;
```

```
sqlite> UPDATE cats SET notes = "A cat" WHERE notes = "meow";
sqlite> SELECT name FROM cats WHERE notes = "A cat";
```

- (b) Cats of different weights require different quantities of food. We have the following table:

```
CREATE TABLE food AS
SELECT 1 AS cat_weight, 0.5 AS amount UNION
SELECT 2           , 2.5           UNION
SELECT 3           , 4.0           UNION
SELECT 4           , 4.5;
```

Write a query that calculates the total amount of food required to feed all the cats (this should work for any table of cats, not just the one we created above). In our example, we have two cats of weight 1, two cats of weight 2, and one cat of weight 4. The total food required is $2 \times 0.5 + 2 \times 2.5 + 1 \times 4.5 = 10.5$.

```
SELECT -----
  

FROM -----
  

WHERE -----;
```

6 Macros

- 6.1 Write the `let` special form as a macro called `let-macro`. Recall that `let` takes in a list of bindings and a body expression. It creates a temporary frame containing the given bindings, and returns the result of evaluating the body in this temporary frame. Do not use the `let` special form in your solution.

You may use the provided `cadr` procedure in your solution.

Hint: The built-in `map` procedure takes in a one-argument function and a list and returns the result of mapping the function to every element in the list.

```
(define-macro (let-macro bindings body)
```

```
)
```

```
(define (cadr lst) (car (cdr lst)))
```

```
scm> (define x 3)
```

```
x
```

```
scm> (let-macro ((x 1) (y 2)) (+ x y))
```

```
3
```

```
scm> (let-macro ((x 2) (y x)) (* x y))
```

```
6
```