Powerset.py Write a python function to compute the powerset of a given set or list (the function should work for both; hint: do a list(S) inside the function). Return a list of lists.

```python
def pow(S):
    """Powerset of a set L. Since sets/lists are unhashable, we convert the set to a list, perform the powerset operations, leaving the result as a list (can't convert back to a set).
    pow(set(['ab', 'bc'])) --> [['ab', 'bc'], ['bc'], ['ab'], []]
    """
    L=list(S)
    if L==[]:
        return([[]])
    else:
        pow_rest0 = pow(L[1:])
        pow_rest1 = list(map(lambda ls: [L[0]]+ls, pow_rest0))
        return(pow_rest0 + pow_rest1)
```

MkDFA.py Define a function `mk_dfa` whose definition is sketched below. A DFA is represented using a dict of the form:

```
{"Q":Q, "Sigma":Sigma, "Delta":Delta, "q0":q0, "F":F})
```

Here, Q is a non-empty set of strings (state names), Sigma is a set of non-empty single-character strings (alphabet), q0 is a state belonging to Q, and F is a possibly non-empty set of states, and is also a subset of Q. Delta is a total function represented as a hash-table, mapping a pair (q, c) (where q in Q and c in Sigma) to a new state q1 where q1 is also in Q.

Implement all the checks in boldface font given above as asserts in Python. Test that all the checks are working. Submit this terminal session of the checks happening as file MkDFATests.txt.

```python
def fst(p):
    """ First of a pair."""
    return p[0]

def snd(p):
    """ Second of a pair."""
    return p[1]

def fn_dom(F):
    """ For functions represented as hash-maps (dicts), return their domain as a set.
    """
    return {k for k in F.keys()}

def fn_range(F):
    """ For functions represented as hash-maps (dicts), return their range as a set.
    """
    return {v for v in F.values()}
```
Design (on paper) a DFA that accepts all strings over $\Sigma = \{0, 1\}$ that end in 0101.

**Solution:** Here is the solution outline with comments - after this, I provide the real details of the DFA coding.

This is only my "sketch" as I would sketch on paper. The real DFA coding begins after this.

- $s_0$ # Start from

$s_0 \rightarrow 1 \rightarrow s_0$ # In state $s_0$, upon a 1, stay in $s_0$
$s_0 \rightarrow 0 \rightarrow s_1$ # upon a 0, move to $s_1$

$s_1 \rightarrow 0 \rightarrow s_1$ # in $s_1$, another 0 is not helping move along. stay in $s_1$
$s_1 \rightarrow 1 \rightarrow s_2$ # Move on 1 to $s_2$

$s_2 \rightarrow 1 \rightarrow s_0$ # Upon 1 fail back to $s_0$
$s_2 \rightarrow 0 \rightarrow s_3$ # upon 0, make progress to $s_3$

$s_3 \rightarrow 0 \rightarrow s_1$ # upon a 0, fail but don’t go clear back; go upto $s_1$ as we can "re" use this 0 # toward a 0101
s3 --> s4  # finally go to s4
s4 --> s3  # fail to s3 because we can have 010101 etc, and the third 0 is a promising 0
        # because it may help advance 01010 to a 010101
s4 --> s0  # fail back to s0

Q1 = {'S0', 'S1', 'S2', 'S3', 'S4'}

Sigma1 = {'0', '1'}

Delta1 = {('S0', '0'): 'S1',
          ('S0', '1'): 'S0',
          ('S1', '0'): 'S1',
          ('S1', '1'): 'S2',
          ('S2', '1'): 'S0',
          ('S2', '0'): 'S3',
          ('S3', '0'): 'S1',
          ('S3', '1'): 'S4',
          ('S4', '0'): 'S3',
          ('S4', '1'): 'S0'}

q01 = 'S0'

F1 = {'S4'}

The generated DFA is