How To Solve It

"If there's a problem you can't solve, then it contains a simpler problem that you can solve. *Find it!*"

-- George Pólya (1887 –1985)

Professor of Mathematics at Stanford University
Quiz: What is the output?

```python
bob = ['4'] * 5
print(bob)
```

a) '4' * 5  
b) 20  
c) 20.0  
d) ['4', '4', '4', '4', '4']  
e) Error: Cannot multiply a list by a number.

Python Lists

In Python, a list is an ordered collection of objects. 

The code below defines the list `s` containing five strings.

```python
s = ['Jon', 'Bran', 'Rickon']
```

In python, a list is a mutable object. That means its contents can be changed:

```python
s[0] = 'Robb'
print(s)
```

['Robb', 'Bran', 'Rickon']
Python Lists: Access a Sublist

```python
>>> a = [3, 6, 9, 12, 15, 18]
>>> print(a)
[3, 6, 9, 12, 15, 18]
>>> print(a[0])
3
>>> print(a[4])
15
>>> print(a[:4])
[3, 6, 9, 12]
>>> print(a[4:]
[15, 18]
```

- Creates a list.
- All elements of list.
- Only element 0 of list
- Only element 4 of list
- Elements 0 through, but not including, element 4.
- Elements 4 through the last element of the list.

Python Lists Can Contain Different Types

```python
>>> myList = ['fish', 1, 'pig', 3.14]
>>> print(myList[0])
fish
>>> print(myList[1])
1
>>> print(myList[3])
3.14
```

- `fish` is a string
- `1` is an integer
- `3.14` is a float
Quiz: List

```python
>>> a = ['one', 'fish', 'two', 'fish']
>>> print(a[2], a[2:])
```

The output of the above print statement is:

- a) two ['two', 'fish']
- b) fish ['two', 'fish']
- c) fish ['fish']
- d) two ['two', 'fish', 'two', 'fish']
- e) fish ['fish', 'fish']

Quiz: List

```python
>>> a = ['red', 'fish', 'blue', 'fish']
>>> print(a[1], a[1:])
```

The output of the above print statement is:

- a) fish ['fish', 'fish']
- b) fish ['fish']
- c) red ['red']
- d) red ['red', 'fish', 'blue', 'fish']
- e) fish ['fish', 'blue', 'fish']
Quiz: What is the output?

```python
chin = ['3'] * 6
print(chin)
```

a) Error: Cannot multiply a list by a number.
b) '3' * 6
c) 18
d) 18.0
e) ['3', '3', '3', '3', '3', '3']

List + operator: Append

```python
>>> a = ['fish', 'cat', 1]
>>> a = a + [3.14]
>>> print(a)
['fish', 'cat', 1, 3.14]

>>> print(a[3])
3.14

>>> print(a[3] + 2.5)
5.640000000000001
```

Both a and [3.14] are lists. The + operator appends the 2nd to the end of 1st.
Both a[3] and 2.5 are floats. The + operator adds the two floats.

What is this? Can't Python add correctly?
Quiz: List

```python
a = [2, 3, 5, 7]
print(a + [1])
```

The output of the above print statement is:

a) [2, 3, 5, 7, 1]
b) [3, 4, 6, 8]
c) [3, 3, 5, 7]
d) [2, 3, 5, 8]
e) [2, 3, 5, 7][1]

---

Loop through a List with `while` `logicalExp`

```python
a = ['fish', 'cat', 1, 3.14]
print(len(a))
4
```

```
i = 0
while i<len(a):
    print(a[i])
i = i + 1
```

Output:

```
fish
cat
1
3.14
```

The `len()` function works on both strings and lists.
Loop through a List with **for item in list**

```python
a = ['fish', 'cat', 1, 3.14]

for myItem in a:
    print(myItem)
```

Output:

```
fish
cat
1
3.14
```

Quiz (we did this before with turtles)

Output:

```
fish
cat
1
3.14
```
3D Tic-Tac-Toe

This is an example 3D board in the format your program needs to draw it.

```
    O-  |  OOX  |  XXO
    --- |  - -  |  ---
    --- |  --X  |  ---
```

Calling this board “3D” is just one way to think if it.

Data Structure for the “2D” Board

The given 2D version represents the board as a list of single character strings:

- `board = [' '] * 10`
- Empty spot the player may move: `' '`
- Spot where move has been made: `'X'` or `'O'`
- The game locations correspond to the board list indices:

```
7 8 9
4 5 6
1 2 3
```

Therefore, a game in this state will have a `board` list with these values:

```
board=[' ', 'O', ' ', ' ', ' ', 'O', 'X', ' ', ' ', 'X']
```
Data Structure for the “3D” Board

This is an example 3D board in the format your program needs to draw it:

O- | OX | XXO
--- | - - | ---
--- | - X | ---

One data structure that could represent this board uses the ' - ' for open spots and the ' ' for illegal spots:

```python
board = ['-'] * 30  # List with 10 elements per level.
board[0] = board[10] = board[30] = board[15] = ' '  # Where the game locations corresponding to the board list indices are:
```

<table>
<thead>
<tr>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 8 9</td>
<td>14 15 16</td>
<td>27 28 29</td>
</tr>
<tr>
<td>4 5 6</td>
<td>11 12 13</td>
<td>24 25 26</td>
</tr>
<tr>
<td>1 2 3</td>
<td></td>
<td>21 22 23</td>
</tr>
</tbody>
</table>

First Step: Rewrite the 2D `drawBoard(board)`

A great first step would be to get the 2D game working with the required display format for level 1 of the 3D game.

Getting the 2D game working with this format requires changing `drawBoard()`, AND also every place where the ' ' character is used to represent an open move location.

After the same 3 moves in a 2D game, this is the how the board would be displayed using the format required for one level of the 3D game.
2D drawBoard(board)

```python
def drawBoard(board):
    print('   ||   ')  
    print('   |   |')
    print('-----------')
    print('  |   |')
    print('  |   |')
    print('-----------')
    print('   |   |')
    print('   |   |')
```

It is very easy to change this to print the 2D board in three rows with the three characters in each row.

English First! Extending getPlayerMove() to 3D

1) The function must loop until the user provides good input.

2) Inside the loop, the function must:
   a) Get the user's move (call inputTuple())
   b) If input spot is less then 1 or greater than 9, then input is bad so continue back to the top of the loop.
   c) If input level is less than 1 or greater than 3, then input is bad so continue back to the top of the loop.
   d) Convert the two input numbers (spot and level) to a single board index.
   e) If the board has an open move character '-', at the board index calculated from the user input, then the move is legal, so exit the loop by returning the board index.
   f) If the loop did not exit, then continue back to the top of the loop.
Calculate `boardIdx` from `spot` and `level`

```python
if (spot == 1) and (level == 1) boardIdx = 1
elif (spot == 1) and (level == 2) boardIdx = 11
elif (spot == 1) and (level == 3) boardIdx = 21
elif (spot == 2) and (level == 1) boardIdx = 2
elif (spot == 2) and (level == 2) boardIdx = 12
elif (spot == 2) and (level == 3) boardIdx = 22
elif (spot == 3) and (level == 1) boardIdx = 3
elif (spot == 3) and (level == 2) boardIdx = 13
elif (spot == 3) and (level == 3) boardIdx = 23
elif (spot == 4) and (level == 1) boardIdx = 4
elif (spot == 4) and (level == 2) boardIdx = 14
elif (spot == 4) and (level == 3) boardIdx = 24
```

Only 15 `elif` statements left for you to figure out on your own...

OR... maybe you can think of a statement that can handle several cases at once?

---

Extending `getPlayerMove()` to 3D

```python
def getPlayerMove(board):
    while (True):
        playerMoveStr = input("Enter move (1-9)")
        boardIdx = int(playerMoveStr)

        if (boardIdx < 1) or (boardIdx > 9):
            continue

        if isSpaceFree(board, boardIdx):
            return boardIdx
```

Replace with `inputTuple()`
Then extract spot and level from the returned tuple.

Check that spot (not `boardIdx`) is not less than 1 or greater than 9.

Calculate `boardIdx` after checking spot and level.

Why?
Extending `isSpaceFree()` to 3D

This function would not need to be changed at all if we were drawing the board in the same style as does the textbook. However, since we draw the board as:

```
X-- | X- X | ---
--- | O X  | ---
--- | OOO  | ---
```

For us, a free space is a ' - ' and an illegal move is ' '.

```python
#=============================================
# Return true if the given move is free on
# the given board.
def isSpaceFree(board, move):
    # Notice that this tricky statement a
    # boolean expression (not an assignment).
    return board[move] == ' '
```

Extending `isBoardFull()` to 3D

Extending this function to 3D requires one small change.

Note: The 3D extension requires that `isSpaceFree()` will return False when `i` equals 10, 15 and 20.

Why did the 2D version not need to worry about what `isSpaceFree()` would return for the unused board[0]?

```python
#=============================================
# Return True if every space on the board has
# been taken. Otherwise return False.
def isBoardFull(board):
    for i in range(1, 10):
        if isSpaceFree(board, i):
            return False
    return True
```

What simple change would make this function work in both versions?
Some Change Free Functions

In our 3D game, there should be no surprise that functions implementing parts of the game that are identical in both versions require no modification when moved to the 3D game:

```python
def whoGoesFirst():
def playAgain():
def inputPlayerLetter():

The `makeMove()` function might be expected to require modification for the 3D game. Happily, it does not:

```python
##============================================
def makeMove(board, letter, move):
    board[move] = letter
```

Extending `isWinner()` to 3D

```python
def isWinner(board, c):
    return False
```

The 2D game has 8 ways to make three-in-a-row.
The 3D game has 8 ways on the 1st layer, 8 on the 3rd plus 4 on the 2nd, ...
plus 8 columns, plus 8 vertical diagonals.

This can be checked with 36 if statements.

OR shorten by using loops. No need to make loops exclude `board[15]`
since the space character will never match an 'X' or 'O'.
Extending `getComputerMove()` (1 of 4)

```python
# Given a board and the computer's letter, # determine where to move and return that move.
def getComputerMove(board, computerLetter):
    if computerLetter == 'X':
        playerLetter = 'O'
    else:
        playerLetter = 'X'
```

Getting the AI to play a smart game of 3D Tic-Tac-Toe takes some thinking - but that is left for extra credit.

Getting the AI to play a not-totally-stupid game only requires extending the existing 2D logic to include the larger board.

If you understand the book's 2D logic, then extending is easy.

If you do not understand the book's logic, then extending it is only about as hard getting a monkey to type the full text of Hamlet on a keyboard.

Extending `getComputerMove()` (2 of 4)

```python
# Here is our algorithm for our Tic Tac Toe AI:
# First, check if we can win in the next move
for i in range(1, 10):
    copy = getBoardCopy(board)
    if isSpaceFree(copy, i):
        makeMove(copy, computerLetter, i)
    if isWinner(copy, computerLetter):
        return i
```

What is going on in this first part of the AI?
Why does it make 9 of copies of the board?
Why does it call each of the functions that it calls?
What, if anything, needs to change?
Extending `getComputerMove()` (3 of 4)

```python
# Check if the player could win on the next move. If so, block that move.
for i in range(1, 10):
    copy = getBoardCopy(board)
    if isSpaceFree(copy, i):
        makeMove(copy, playerLetter, i)
        if isWinner(copy, playerLetter):
            return i
```

What is going on in this second part of the AI?
Why does it make another bunch of copies of the board?
Why does it call each of the functions that it calls?
What, if anything, needs to change?

Extending `getComputerMove()` (4 of 4)

```python
# If open, take one of the corners.
move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
if move != None:
    return move

# If open, take the center.
if isSpaceFree(board, 5):
    return 5

# Move into an open side spot.
return chooseRandomMoveFromList(board, [2, 4, 6, 8])
```

When extending this to 3D, there are a few slightly different strategies that could all earn full credit (but not extra credit).
What are they? Are they equally good strategies?
Extending the main game loop to 3D

- Where is the program initialized?
- Where is the main game loop?
  - Where does it start?
  - What is the last line within the main game loop?
  - How does the main game loop exit?
- What changes need to be made to the main game loop?

Quiz: What is the Output?

```python
board=['-']*30
board[27]='X'
for i in range(1,8):
    if (board[i]==board[i+10]==board[i+20]):
        print(i, i+10, i+20, board[i])
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11 21 -</td>
</tr>
<tr>
<td>2</td>
<td>12 22 O</td>
</tr>
<tr>
<td>3</td>
<td>13 23 X</td>
</tr>
<tr>
<td>4</td>
<td>18 28 -</td>
</tr>
<tr>
<td>5</td>
<td>14 24 -</td>
</tr>
<tr>
<td>6</td>
<td>16 26 O</td>
</tr>
<tr>
<td>7</td>
<td>17 27 X</td>
</tr>
</tbody>
</table>
A List's `reverse()` Method

```python
a = ['fish', 'cat', 1, 3.14]
a.reverse()
print(a)
```

Output:

```python
[3.14, 1, 'cat', 'fish']
```

- Python does not have a `reverse()` method.
- Every Python list object has a `reverse()` method.

This statement tells Python to use `a`'s `reverse()` method on `a`.

A list's `reverse()` and a string's `upper()`

```python
a = ['fish', 'cat', 1, 3.14]
a.reverse()
print(a)  # [3.14, 1, 'cat', 'fish']
a.upper() # ERROR: 'list' object has no attribute 'upper'
```
A list's `reverse()` and a string's `upper()`

```python
a = ['fish', 'cat', 1, 3.14]
a.reverse()
print(a)  # [3.14, 1, 'cat', 'fish']

a[3] is the string 'fish'.

String's `upper()` method returns a new string converted to uppercase, but does not change the string.

```python
print(a)  # [3.14, 1, 'cat', 'FISH']
```

"My dad says persistence is the key to success. So I'm going to keep giving you the same wrong answer until it becomes the right answer!"
"The reason I'm successful is because I'm lucky. But I didn't get lucky until I started working 90 hours a week."