

Open-source Software

Much software is proprietary and the intellectual property of whoever created it (a company or a person). But some software is **open-source**, meaning the code is available publicly to anyone online.

Open-source software isn't just free to use – it's also available for developers to build on it. A developer can branch a piece of software to make their own version with modifications, or they can contribute updates to the original version by sending them to the people who maintain the main version.

Major examples of open-source software include the operating system kernel [Linux](#), the browser [Firefox](#), and the media player [VLC](#).



Problem Decomposition

15-110 – Wednesday 10/07

Quizlet4

Learning Goals

- Use **problem decomposition** to break a large problem into reasonably-sized parts.
- Use **docstrings** to describe a Python function.
- **Prompt** and **iterate** using AI tools to create a Python project of moderate scale.

Problem Decomposition

Case Study: Memory Game

Try it here:

<https://www.helpfulgames.com/subjects/brain-training/memory.html>

You want to build a memory matching game using AI. You use the following prompt:

"Please implement a simple memory game using Python. The game has a deck made of pairs of cards - for example, a deck could have two cards that both say "dog" and two cards that both say "cat". All cards are initially face down. The user can click on two cards to flip them face up. If they match, they stay face up; if they don't, they flip face down again. The user keeps playing until all cards are face up, then is given a score based on how many flips they took."

The LLM produces code that works, but it's a lot of text and a bit overwhelming to understand.

Working With AI

How can we ensure the AI produces results we can understand?

Instead of offloading all the work to the AI, do some of the **planning** in advance, and provide that plan to the agent. This ensures you and the AI are more in sync regarding expectations and how the project will work.

Problem decomposition is an especially helpful approach when working with projects that will require 20+ lines of code.

Problem Decomposition

Problem Decomposition is the process of taking a complex problem and breaking it down into more manageable tasks.

Each task is assigned its own **function**. Some tasks may be so complicated that they need to be broken down into further subtasks, which will be implemented in further functions.

If we provide a set of required functions to an AI and we already know what they're supposed to do, interpreting the results may be easier.

Example – Tic Tac Toe

`playTicTacToe()` # implements the whole game



`setupBoard()` # creates an initial board



`makeMove(board, player)` # player chooses where to place token



`findWinner(board)` # determines the winner



`checkTie(board)` # are all spaces filled?

Designing Good Functions

How do we know when we can stop breaking down functions?

A good function:

- Accomplishes a single clearly-defined task
- Implements an algorithm that is simple and short enough that we can understand it easily (< 20 lines)
- Encourages reusability instead of hardcoding information

Counterexample: multiple tasks

Here we're trying to solve a word search puzzle.

The function to the right is trying to do too many things. Moving part of the function into another function would improve readability and modularity.

```
def solveWordSearch(puzzle, words):
    solution = { }
    rows, cols = len(puzzle), len(puzzle[0])
    for r in range(rows):
        for c in range(cols):
            for w in words:
                if w not in solution and w[0] == puzzle[r][c]:
                    dirs = [(-1, -1), (-1, 0), (-1, 1), (0, -1),
                            (0, 1), (1, -1), (1, 0), (1, 1)]
                    for (dr, dc) in dirs: # delta row, delta col
                        finalRow = r + dr*(len(w)-1)
                        finalCol = c + dc*(len(w)-1)
                        if 0 <= finalRow < rows and \
                            0 <= finalCol < cols: # in bounds
                            works = True
                            for i in range(len(w)):
                                newRow = r + dr*i
                                newCol = c + dc*i
                                if w[i] != puzzle[newRow][newCol]:
                                    works = False
                                    break
                            if works == True:
                                solution[w] = ((r, c),
                                                (finalRow, finalCol))
                                break
    return solution
```

Tasks in a Word Search

What is a natural breakdown of tasks when solving a wordsearch?

The steps taken when checking a word from a starting position seem distinct from the more general steps taken to perform a search.

P	Y	T	H	O	N	A
R	L	U	K	J	I	B
O	E	C	O	D	E	C
G	C	D	B	N	G	D
R	A	M	F	H	U	E
A	Q	W	E	R	T	G
M	N	O	P	I	Y	G

Words:
PYTHON – PROGRAM – CODE

Example: modified solveWordSearch

Split the program into two functions:

`solveWordSearch` and `findEndFromStart`.

Now our main program is a more reasonable length and does a clear thing: checks whether each word starts at each of the possible starting positions.

```
def solveWordSearch(puzzle, words):
    solution = { }
    rows, cols = len(puzzle), len(puzzle[0])
    for r in range(rows):
        for c in range(cols):
            for w in words:
                if w not in solution and \
                    w[0] == puzzle[r][c]:
                    result = findEndFromStart(puzzle, w, r, c)
                    if result != None:
                        solution[w] = ((r, c),
                                       (result[0], result[1]))
    return solution
```

Example: modified solveWordSearch

Is the other function small enough?

This also does a single thing: checks every direction from a starting point to see if the word is in that direction.

We could maybe make another helper to split checking the letters from iterating over directions / checking if they're in bounds, but we can also stick with what's already here.

```
def findEndFromStart(puzzle, word, row, col):
    dirs = [(-1, -1), (-1, 0), (-1, 1), (0, -1),
            (0, 1), (1, -1), (1, 0), (1, 1)]
    rows, cols = len(puzzle), len(puzzle[0])
    for (dr, dc) in dirs: # delta row, delta col
        finalRow = row + dr*(len(word)-1)
        finalCol = col + dc*(len(word)-1)
        if 0 <= finalRow < rows and \
            0 <= finalCol < cols: # in bounds
            works = True
            for i in range(len(word)):
                newRow = row + dr*i
                newCol = col + dc*i
                if word[i] != puzzle[newRow][newCol]:
                    works = False
                    break
            if works == True: # all letters were valid
                return (finalRow, finalCol)
    return None # no solution found
```

Prompting AI – Input, Output, Steps

In a previous lecture we talked about how we can prompt an AI agent to generate code for a function by focusing on the three primary parts of an algorithm: the input, output, and goal.

As we start coding at scale, communicating the parts of every function clearly becomes especially important!

- **Input:** what are the types and purposes of the parameters?
- **Output:** what is the type and meaning of the returned value?
- **Steps:** what is the purpose of the function? What general algorithm is it following?

Describe Functions with Docstrings

We can communicate everything a person or an AI needs to know about a function using a **docstring**.

This is a triple-quote string written in the first line of a function. It does not add functionality; instead, it describes how the function works.

```
def quadraticFormula(a, b, c):  
    """  
    Calculates the two roots of a quadratic equation  
  
    Uses the quadratic formula:  
     $(-b \pm \sqrt{b^2 - 4ac}) / 2a$   
  
    Parameters:  
        a (float) - coefficient of  $x^2$   
        b (float) - coefficient of  $x$   
        c (float) - constant coefficient  
  
    Returns:  
        tuple[float] - the two roots of the equation  
    """  
    partial = (b**2 - 4*a*c)**0.5  
    root1 = (-b + partial) / (2 * a)  
    root2 = (-b - partial) / (2 * a)  
    return (root1, root2)
```

Describe Functions with Docstrings

summary (one line)



```
def quadraticFormula(a, b, c):  
    """  
    Calculates the two roots of a quadratic equation
```

more detailed
description (optional)



```
    Uses the quadratic formula:  
    (-b +/- sqrt(b^2 - 4ac)) / 2a
```

Inputs
name (type) - description



Parameters:

```
    a (float) - coefficient of x^2  
    b (float) - coefficient of x  
    c (float) - constant coefficient
```

Output
type - description



Returns:

```
    tuple[float] - the two roots of the equation  
    """
```

```
    partial = (b**2 - 4*a*c)**0.5  
    root1 = (-b + partial) / (2 * a)  
    root2 = (-b - partial) / (2 * a)  
    return (root1, root2)
```

You Do: Make a Docstring

You do: write the docstring for the function `solveWordSearch` from the previous example.

Make sure you include all parts – summary, parameters, and returned value.

```
def solveWordSearch(puzzle, words):  
    ...  
  
puzzle = [ "PYTHONA",  
           "RLUKJIB",  
           "OECODEC",  
           "GCDBNGD",  
           "RAMFHUE",  
           "AQWERTG",  
           "MNOPIYG" ]  
  
targetWords = ["PYTHON", "PROGRAM", "CODE" ]  
  
sol = solveWordSearch(puzzle, targetWords)  
print(sol)  
# { 'PYTHON': ((0, 0), (0, 5)),  
#   'PROGRAM': ((0, 0), (6, 0)),  
#   'CODE': ((2, 2), (2, 5)) }
```

Example: Memory Game

Let's put it all together to program the **memory game** from the start of lecture, but this time in a format that we can more easily understand.

1. What **tasks** can we break the game down into?
2. For each task, write a **docstring**.
3. Plug it into the AI and **test** the results.

Learning Goals

- Use problem decomposition to break a large problem into reasonably-sized parts.
- **Prompt** and **iterate** using AI tools to create a Python project of moderate scale.
- **Critique** code generated by an AI tool to identify possible places for improvement.