Recall Week 8 - Dictionaries

You should be able to:

- Discuss the key characteristics of dictionaries
- Manipulate dictionary keys & values
- Apply Python methods to dictionaries
- Perform standard loops on dictionaries
- Assess membership and perform operations on dictionaries

Week 9 - Recursion

You should be able to:

- Evaluate when recursion is appropriate
- Identify and implement base and recursive cases
- Distinguish different types of recursive problems
- Trace and debug a recursive function
- Apply recursion on complex data structures

Let’s Define Recursion:

Function that calls itself
Doesnt end until task is finished - base case for stopping
Copies of itself - breaking down into smaller problems

And something else……..

Let’s Define Recursion:

Recursive functions are functions that repeatedly calls itself until some condition is met. They are of value when a complex problem can be broken down into repeating subproblems. In other words a problem that can be broken down into smaller versions of itself.

Multiplication

* operator can perform multiplication for us

```
def mult(a, b):
    return a * b
```

Iteration - performing the same action multiple times, often updating one or more state variables each time, until a goal or condition is met.

```
count = 0           # initial state
while count < 3:    # condition checked each iteration
    count += 1      # state change
```

Multiplication

iteration

Define a*b as a+a+a+a+a.....b times

```
def mult(a,b):
    total = 0          # initial state
    for n in range(b):
        total += a     # state change
    return total
```

Recursion

Recursion is a process in which a function solves a problem by repeatedly applying the same operation to progressively smaller portions of the original data or problem, with each call stored on the call stack. When the smallest unit (the base case) is reached and solved directly, each stored call finishes in reverse order, combining or returning results as the stack unwinds.

Recursion

How it works:

- **The Function Calls Itself**: A recursive function is one that invokes itself during its execution
- **Each Call Works on a Smaller Problem**: Every time the function calls itself, it works with a reduced version of the original problem
- **The Base Case Stops the Recursion**: Eventually, the problem becomes simple enough to solve directly, and the recursion stops
- **Results Build Up as Recursion Returns**: As each function call completes, it returns a result that contributes to solving the larger problem

Recursive Pattern
Multiplication

Define a*b as a+a+a+a+a.....b times

```
def mult(a,b):
    total = 0                # initial state
    for n in range(b):
        total += a  # state change
    return total
```

5*4 is 5+5+5+5
5+(5*3)
5+5+(5*2)
5+5+5+5+(5*1)

Recursive Pattern

Multiplication

Define a*b as a+a+a+a+a.....b times

```
def mult(a,b):
    total = 0                # initial state
    for n in range(b):
        total += a  # state change
    return total
```

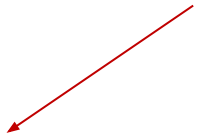
5*4 is 5+5+5+5

5+(5*3)

5+5+(5*2)

5+5+5+5+(5*1)

Something we know



https://ocw.mit.edu/courses/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/video_galleries/lecture-videos/

Recursive Pattern

Multiplication

$$a*b = \underbrace{a + a + a + a + a \ldots\ldots + a}_{b \text{ times}}$$

$$= a + \underbrace{(a + a + a + a \ldots\ldots + a)}_{b-1 \text{ times}}$$

$$= a + a * (b-1)$$

Recursive reduction

https://ocw.mit.edu/courses/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/video_galleries/lecture-videos/

Recursive Pattern

Breaking Down to Simplify

5*4

→ need to solve 5*3 first

→ need to solve 5*2 first

→ need to solve 5*1 first (BASE CASE)

Recursive Pattern

Unwinding Up with Answers

5*1 = 5	← base case returns 5
5*2 = 5 + (5*1) = 5 + 5 = 10	← uses previous result
5*3 = 5 + (5*2) = 5 + 10 = 15	← uses previous result
5*4 = 5 + (5*3) = 5 + 15 = 20	← uses previous result

https://ocw.mit.edu/courses/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/video_galleries/lecture-videos/

Recursive Pattern

Each function call is a separate scope/environment

- Same variable names but in separate scopes and different objects
- Control passes to previous scope once function returns value

Recursive Pattern
Multiplication

- Recursive step
  - If b != 1, a * b = a + a(b - 1)
- Base case
  - If b = 1, a * b = a

```
def mult_recur(a,b):
    If b == 1:
        return a
    else:
        Return a + mult_recur(a, b - 1)
```

Base case

Recursive step

https://ocw.mit.edu/courses/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/video_galleries/lecture-videos/

Recursion

Every recursive function has the same structure:

```
def recursive_function(data):

    # 1. BASE CASE: what is the simplest case? Can be more than 1
        if stopping_condition(data):
            return simple_answer

    # 2. RECURSIVE CASE: how to create smaller versions of same problem?
    smaller_piece = reduce_problem(data)
    result = recursive_function(smaller_piece)

    # 3. COMBINE: how to combine results from smaller problems?
    return combine(result)
```

Basic Types of Recursive Problems

- Linear Recursion
- Tree (Binary) Recursion
- Tail Recursion
- Divide and Conquer

Linear Recursion

A linear recursive function:

- One recursive call per function
- Creates a single chain of calls (straight line)
- Work happens during unwinding
- Common in summation, counting, or accumulation problems

Linear Recursion

```python
def reverse_string(s):
    if len(s) == 0:
        return ""
    return s[-1] + reverse_string(s[:-1])
```

```
reverse_string("hello")
→ "o" + reverse_string("hell")
→ "o" + "l" + reverse_string("hel")
→ "o" + "l" + "l" + reverse_string("he")
→ "o" + "l" + "l" + "e" + reverse_string("h")
→ "o" + "l" + "l" + "e" + "h" + reverse_string("")
→ "o" + "l" + "l" + "e" + "h" + ""
→ "olleh"
```

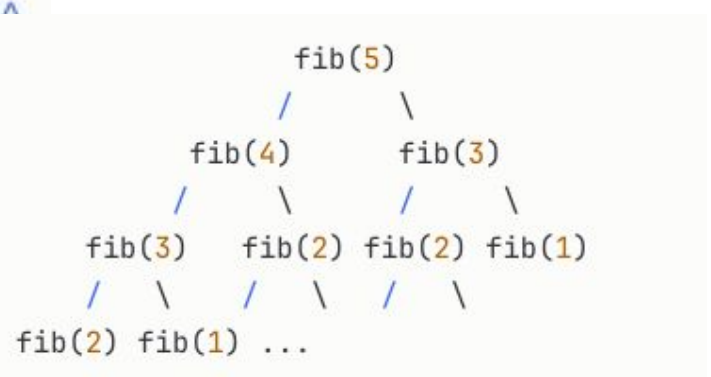
Tree Recursion

A tree (binary) recursive function:

- Multiple recursive calls per level, producing multiple subproblems
- Forms a recursive tree
- Can be inefficient without memoization
- Common examples include:
  - Generating combination
  - Traversing trees
  - Fibonacci sequence

Tree Recursion

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
           ^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^
```



Tail Recursion

A tail recursive function:

- Does all the work before the recursive call
- Recursive call is the last operation in the function
- Uses an accumulator variable to store progress
- Common in summation, counting nodes in linked list or tree, calculating total length or cumulative value

```
def factorial(n, accumulator=1):
    if n == 0:
        return accumulator
    return factorial(n - 1, accumulator * n)
```

Tail Recursion

```
def factorial(n, accumulator=1):
    if n == 0:
        return accumulator
    return factorial(n - 1, accumulator * n)
```

Divide & Conquer Recursion

A divide & conquer recursive function:

- Three phase structure:
  - Divides problem into smaller, independent, sub-problems of same type
  - Solves subproblems recursively
  - Merges solutions of subproblems to solve original problem
- Balanced splitting: problems are divided into roughly equal parts
- Typically shallower depth than linear recursion with reduced risk of stack overflow
- Common in sorting & searching algorithms

Divide & Conquer Recursion

```python
def binary_search(arr, target, left=0, right=None):
    if right is None:
        right = len(arr) - 1

    if left > right:
        return -1

    mid = (left + right) // 2

    if arr[mid] == target:
        return mid
    elif arr[mid] > target:
        return binary_search(arr, target, left, mid - 1)
    else:
        return binary_search(arr, target, mid + 1, right)
```

Recursion

Why it’s useful for working with data:

- Process data structures without knowing their size or depth in advance.

Recursion

Why it’s useful for working with data:

- Process data structures without knowing their size or depth in advance.
- Simplifies complex hierarchical operations. Works well when a problem can be broken into smaller versions of the problem. (e.g., lists -> sublists)

Recursion

Why it’s useful for working with data:

- Process data structures without knowing their size or depth in advance.
- Simplifies complex hierarchical operations. Works well when a problem can be broken into smaller versions of the problem. (e.g., lists -> sublists)
- Solutions often express a problem’s logic more directly than iterative loops.

Recursion

Why it’s useful for working with data:

- Process data structures without knowing their size or depth in advance.
- Simplifies complex hierarchical operations. Works well when a problem can be broken into smaller versions of the problem. (e.g., lists -> sublists)
- Solutions often express a problem’s logic more directly than iterative loops.
- Provides a natural mental and structural model for parent-child relationships (e.g., folder systems, nested JSON)

Recursion

What data related use cases lend themselves to recursive solutions?

Recursion

What data related use cases lend themselves to recursive solutions?

- Decision trees - classification or regression
- Product category hierarchies in commerce
- Web crawlers
- Software dependency graphs
- Social network traversal
- Various search algorithms
- Fractals and geometric problems

Problem Statement:
Write a recursive function that counts how many times a target value appears in a list.

```python
# Examples:
count_occurrences([1, 2, 3, 2, 2, 4], 2)  # Should return 3
count_occurrences([5, 5, 5], 5)           # Should return 3
count_occurrences([1, 2, 3], 9)           # Should return 0
count_occurrences([], 1)                  # Should return 0
```

```python
def count_occurrences(items, target):
```

Base Case:

Recursive Case:

Problem Statement:
Write a recursive function that checks if a list of numbers is sorted in ascending order.

```python
# Examples:
is_sorted([1, 2, 3, 4, 5])      # Should return True
is_sorted([1, 3, 2, 4])         # Should return False
is_sorted([5])                  # Should return True
is_sorted([])                   # Should return True
is_sorted([1, 1, 2, 2, 3])      # Should return True (equal values OK)
```

```
def is_sorted(numbers):
```

Base Case:

Recursive Case:

File Retrieval

implement the recursive function `find_python_files()` that searches through all directories and subdirectories to find every Python file (.py) and return their complete paths in a list.

Base Case:

Recursive Case:

```
file_system = {
    'src': {
        'main.py': {'type': 'file'},
        'utils': {
            'helper.py': {'type': 'file'},
            'config.json': {'type': 'file'},
            'math': {
                'calculator.py': {'type': 'file'},
                'constants.py': {'type': 'file'}
            }
        },
        'tests': {
            'test_main.py': {'type': 'file'},
            'fixtures': {
                'data.csv': {'type': 'file'},
                'test_utils.py': {'type': 'file'}
            }
        }
    },
    'docs': {
        'readme.txt': {'type': 'file'},
        'api.py': {'type': 'file'}
    },
    'setup.py': {'type': 'file'}
}
```

The Problem: Nested comment thread

You receive JSON from a web API with nested lists of arbitrary depth. You need flat list of comments with all replies extracted.

```
api_response = [
    {"id": 1, "text": "Great post!"},
    {
        "id": 2,
        "text": "I agree",
        "replies": [
            {"id": 3, "text": "Me too"},
            {
                "id": 4,
                "text": "Same here",
                "replies": [{"id": 5, "text": "Definitely"}],
            },
        ],
    },
    {"id": 6, "text": "Thanks all"},
]
```

```
id: 1  Great post!

id: 2  I agree
 ├─ id: 3  Me too
 └─ id: 4  Same here
      └─ id: 5  Definitely

id: 6  Thanks all
```

What type of recursive function do should we use to solve this problem?

```
api_response = [
    {"id": 1, "text": "Great post!"},
    {
        "id": 2,
        "text": "I agree",
        "replies": [
            {"id": 3, "text": "Me too"},
            {
                "id": 4,
                "text": "Same here",
                "replies": [{"id": 5, "text": "Definitely"}],
            },
        ],
    },
    {"id": 6, "text": "Thanks all"},
]
```

The Problem: Nested comment thread

You receive JSON from a web API with nested lists of arbitrary depth. You need flat list of comments with all replies extracted.

```
api_response = [
    {"id": 1, "text": "Great post!"},
    {
        "id": 2,
        "text": "I agree",
        "replies": [
            {"id": 3, "text": "Me too"},
            {
                "id": 4,
                "text": "Same here",
                "replies": [{"id": 5, "text": "Definitely"}],
            },
        ],
    },
    {"id": 6, "text": "Thanks all"},
]
```

Base case?

Recursive case?

```python
def flatten_comments(comments):
    """
    Recursively flattens nested comment structures.
    Args:
        comments (list[dict]): List of comment objects, each possibly containing 'replies'.
    Returns:
        list[dict]: A flat list of all comments (no parent references).
    """

    flat_list = []

    for comment in comments:

        flat_list.append(
            {"id": comment["id"], "text": comment["text"]}
        )

        if "replies" in comment:
            flat_list.extend(flatten_comments(comment["replies"]))

    return flat_list
```

flat_list

[{'id': 1, 'text': 'Great post!'},
 {'id': 2, 'text': 'I agree'},
 {'id': 3, 'text': 'Me too'},
 {'id': 4, 'text': 'Same here'},
 {'id': 5, 'text': 'Definitely'},
 {'id': 6, 'text': 'Thanks all'}]

```python
def flatten_comments(comments, parent_id=None):
    """
    Recursively flattens nested comment structures while preserving parent relationships.

    Args:
        comments (list[dict]): List of comment objects, each possibly containing 'replies'.
        parent_id (int | None): The ID of the parent comment, if any.

    Returns:
        list[dict]: A flat list of all comments with parent IDs included.
    """
    flat_list = []

    for comment in comments:
        # Record the current comment and its parent
        flat_list.append(
            {"id": comment["id"], "text": comment["text"], "parent_id": parent_id}
        )

        # Recursively flatten replies, passing the current comment's ID as parent
        if "replies" in comment:
            flat_list.extend(flatten_comments(comment["replies"], parent_id=comment["id"]))

    return flat_list
```

Tracking Hierarchy

```
[{'id': 1, 'parent_id': None, 'text': 'Great post!'},
 {'id': 2, 'parent_id': None, 'text': 'I agree'},
 {'id': 3, 'parent_id': 2, 'text': 'Me too'},
 {'id': 4, 'parent_id': 2, 'text': 'Same here'},
 {'id': 5, 'parent_id': 4, 'text': 'Definitely'},
 {'id': 6, 'parent_id': None, 'text': 'Thanks all'}]
```

When is Recursion Appropriate?

- **Structure is truly hierarchical and depth is unknown**
- **Problem naturally breaks into smaller versions of itself**
- **Iterative solution would be significantly more complex**
- **Data depth is reasonable (< 100 levels typically)**