CE49X: Introduction to Computational Thinking and Data Science for Civil Engineers

Week 2: Advanced Python Programming - Control Flow, Functions, and More

Dr. Eyuphan Koc

Department of Civil Engineering
Bogazici University

Fall 2025

Based on "A Whirlwind Tour of Python" by Jake VanderPlas
Chapters 07-12: Control Flow, Functions, Errors, Iterators, List Comprehensions, and Generators
`https://github.com/jakevdp/WhirlwindTourOfPython`

Week 2 Outline

Control Flow: The Foundation of Programming Logic

What is Control Flow?

Control flow determines the order in which code statements are executed:

- **Sequential**: Default top-to-bottom execution
- **Conditional**: Execute code based on conditions
- **Iterative**: Repeat code blocks multiple times

Civil Engineering Applications

- Check if structural loads exceed design limits
- Process multiple soil samples in a dataset
- Iterate through different design scenarios
- Handle different material properties based on conditions

Conditional Statements: if-elif-else

```python
# Basic conditional structure
load = 1500  # kN    # <-- Try changing this value!
design_capacity = 1200  # kN

if load <= design_capacity:
    print("Structure is safe")
elif load <= design_capacity * 1.1:
    print("Structure needs inspection")
else:
    print("Structure is overloaded - immediate action required")
    safety_factor = design_capacity / load
    print(f"Safety factor: {safety_factor:.2f}")
```

Key Points

- Use : after each condition
- Indentation defines code blocks
- elif is short for "else if"
- else is optional

For Loops: Iterating Over Sequences

```python
# Processing multiple measurements
concrete_strengths = [25, 30, 28, 32, 27]  # MPa

print("Concrete strength analysis:")
for strength in concrete_strengths:
    if strength >= 30:
        grade = "High grade"
    elif strength >= 25:
        grade = "Standard grade"
    else:
        grade = "Low grade"
    print(f"Strength: {strength} MPa - {grade}")
```

For Loops: Iterating Over Sequences

Range Function

```python
# Generate loading scenarios
for load_factor in range(1, 6):  # 1 to 5
    applied_load = load_factor * 100  # kN
    print(f"Load Factor {load_factor}: {applied_load} kN")
```

[LIVE] Coding Challenge 1: Beam Safety Analysis

Your Task (3 minutes)

Write code to analyze multiple beam loads and determine their safety status:

- Given: `beam_loads = [850, 1200, 950, 1400, 750]` (in kN)
- Given: `design_capacity = 1000` kN
- Calculate safety factor for each beam
- Classify as: "Safe" (SF > 1.2), "Warning" (1.0 < SF <= 1.2), or "Failed" (SF <= 1.0)

Starter Code

```
beam_loads = [850, 1200, 950, 1400, 750]  # kN
design_capacity = 1000  # kN

# YOUR CODE HERE
# Hint: Use a for loop and if-elif-else
```

While Loops: Condition-Based Iteration

```python
beam_depth = 200  # mm
max_stress = 0
target_stress = 150  # MPa

while max_stress < target_stress:
    # Calculate stress (simplified)
    max_stress = 50000 / (beam_depth ** 2) * 1000  # Convert to MPa

    if max_stress < target_stress:
        beam_depth += 10
        print(f"Increasing depth to {beam_depth} mm")
    else:
        print(f"Final design: {beam_depth} mm depth")
        print(f"Max stress: {max_stress:.1f} MPa")
```

Caution

Always ensure the loop condition will eventually become False to avoid infinite loops!

Loop Control: break and continue

```python
# Finding first acceptable design  # <-- Students: Can you spot the logic issue?
materials = ['steel', 'concrete', 'timber', 'aluminum']
costs = [150, 80, 60, 200]  # $/m^3
budget_limit = 100

print("Searching for materials within budget:")
for material, cost in zip(materials, costs):
    if cost > budget_limit:
        print(f"Skipping {material} - too expensive (${cost})")
        continue  # Skip to next iteration

        print(f"Found suitable material: {material} at ${cost}/m^3")
    if material == 'concrete':
        print("Concrete selected - stopping search")
        break  # Exit loop entirely

print("Material selection complete")
```

Advanced Loop Feature: else Clause

```python
# Sieve of Eratosthenes - finding prime numbers (useful for optimization)
def find_primes_up_to(n):
    primes = []
    for num in range(2, n):
        for factor in primes:
            if num % factor == 0:
                break  # Not prime
        else:  # No break occurred - number is prime
            primes.append(num)
    return primes

# Find first 10 primes
first_primes = find_primes_up_to(30)
print("Prime numbers:", first_primes)
```

Loop-else Pattern

The `else` block executes only if the loop completes naturally (no `break`)

Functions: Building Reusable Code

Why Functions?

- **Reusability**: Write once, use many times
- **Organization**: Break complex problems into smaller parts
- **Testing**: Easier to test individual components
- **Collaboration**: Share functionality between team members

Engineering Applications

- Calculate structural properties (moment of inertia, section modulus)
- Convert between units (metric/imperial, different stress units)
- Perform repetitive design calculations
- Implement standard engineering formulas

Basic Function Definition

```python
def calculate_beam_moment(load, length):
    """Calculate maximum moment in simply supported beam.

    Args:
        load (float): Uniformly distributed load in kN/m
        length (float): Beam span in meters

    Returns:
        float: Maximum moment in kN*m
    """
    max_moment = (load * length**2) / 8
    return max_moment

# Using the function
udl = 10  # kN/m
span = 6  # m
moment = calculate_beam_moment(udl, span)
print(f"Maximum moment: {moment} kN*m")
```

Functions with Default Parameters

```python
def calculate_concrete_strength(fc_28=25, age_days=28, cement_type='OPC'):
    """Calculate concrete strength at different ages."""
    # Simplified maturity model
    if cement_type == 'RHPC':
        k = 0.25  # Rapid hardening
    elif cement_type == 'PPC':
        k = 0.15  # Pozzolanic
    else:
        k = 0.20  # Ordinary Portland Cement

    strength_ratio = (age_days / (k + 0.95 * age_days))
    return fc_28 * strength_ratio

# Usage examples
print(f"7-day: {calculate_concrete_strength(age_days=7):.1f} MPa")
print(f"RHPC: {calculate_concrete_strength(age_days=7,
    cement_type='RHPC'):.1f} MPa")
```

[QUICK] Challenge: Design Your Function

Pair Programming Exercise (4 minutes)

With your neighbor, write a function to check column slenderness:

- Function: `check_column_slenderness(height, width, depth)`
- Calculate slenderness ratio: `SR = height / min(width, depth)`
- Return classification:
  - "Short" if SR < 12
  - "Intermediate" if 12 <= SR <= 50
  - "Slender" if SR > 50
- Also return the actual slenderness ratio

Multiple Return Values

```python
def analyze_beam_section(width, depth, material='steel'):
    """Calculate section properties of rectangular beam.
    Returns:
        tuple: (area, moment_of_inertia, section_modulus)
    """
    area = width * depth  # mm^2
    moment_of_inertia = (width * depth**3) / 12  # mm^4
    section_modulus = moment_of_inertia / (depth/2)  # mm^3
    return area, moment_of_inertia, section_modulus
# Unpack multiple return values
w, h = 200, 400  # mm
A, I, S = analyze_beam_section(w, h)

print(f"Section properties:")
print(f"Area: {A:,.0f} mm^2")
print(f"Moment of Inertia: {I:,.0f} mm^4")
print(f"Section Modulus: {S:,.0f} mm^3")
```

Variable Arguments: *args and **kwargs

```python
def calculate_total_load(*loads, safety_factor=1.5, **load_types):
    """Calculate total design load with safety factors."""
    total_service_load = sum(loads)

    # Add named loads with specific factors
    for load_name, (load_value, factor) in load_types.items():
        factored_load = load_value * factor
        total_service_load += factored_load
        print(f"{load_name}: {load_value} kN * {factor} = {factored_load}
            kN")

    design_load = total_service_load * safety_factor
    return total_service_load, design_load

# Usage
service, design = calculate_total_load(50, 30, 20, safety_factor=1.6,
                                       wind=(25, 1.2), seismic=(40, 1.0))
print(f"Service: {service} kN, Design: {design} kN")
```

Lambda Functions: Quick Anonymous Functions

```python
# Lambda functions for simple calculations
stress_to_strain = lambda stress, E: stress / E
unit_weight_concrete = lambda fc: 22.5 + 0.12 * fc  # kN/m^3

# Using lambda with built-in functions
loads = [120, 85, 150, 200, 95]
safety_factors = [1.4, 1.6, 1.2, 1.8, 1.5]
design_loads = list(map(lambda x, y: x * y, loads, safety_factors))
print("Design loads:", design_loads)

# Sort materials by cost-effectiveness
materials = [{'name': 'Steel', 'strength': 250, 'cost': 800},
             {'name': 'Concrete', 'strength': 30, 'cost': 150}]
efficient_materials = sorted(materials,
                             key=lambda m: m['strength']/m['cost'],
                                 reverse=True)
for mat in efficient_materials:
    print(f"{mat['name']}: {mat['strength']/mat['cost']:.3f} ratio")
```

Types of Programming Errors

Three Categories of Errors

- **Syntax Errors**: Invalid Python code structure
- **Runtime Errors**: Code fails during execution
- **Semantic Errors**: Code runs but produces wrong results

Engineering Context

- **Syntax**: Forgetting colons in if statements
- **Runtime**: Division by zero in safety factor calculations
- **Semantic**: Using wrong formula for beam deflection

Focus on Runtime Errors

We'll focus on handling runtime errors using Python's exception handling framework

Basic Exception Handling: try-except

```python
def calculate_safety_factor(capacity, demand):
    """Calculate safety factor with error handling."""
    try:
        safety_factor = capacity / demand
        status = "Safe" if safety_factor >= 1.5 else "Check"
        return safety_factor, status
    except ZeroDivisionError:
        return None, "Zero demand error"
    except TypeError:
        return None, "Type error"

# Examples
print(calculate_safety_factor(1000, 500))  # (2.0, 'Safe')
print(calculate_safety_factor(1000, 0))    # (None, 'Zero demand error')
```

[DEBUG] Together: Find and Fix the Errors

Collaborative Debugging (5 minutes)

This engineering calculation has 3 bugs. Can you spot and fix them?

```python
def calculate_beam_stress(moment, width, height):
    """Calculate maximum bending stress in a rectangular beam."""
    # Bug #1: What happens with the inputs?
    section_modulus = width * height^2 / 6

    # Bug #2: Check the calculation
    stress = moment / section_modulus * 1000  # Convert to MPa

    # Bug #3: Error handling issue
    try:
        if stress > 250:
            status = "Overstressed"
        else:
            status = "Safe"
    except:
```

Specific Exception Handling

```python
def load_material_properties(filename):
    """Load material properties with specific error handling."""
    try:
        with open(filename, 'r') as file:
            return file.read()
    except FileNotFoundError:
        print(f"File '{filename}' not found!")
    except PermissionError:
        print(f"Permission denied: '{filename}'")
    except UnicodeDecodeError:
        print(f"Invalid encoding: '{filename}'")
    except Exception as e:
        print(f"Unexpected: {type(e).__name__}")
    return None

# Usage example
data = load_material_properties("steel_props.txt")
print("Loaded successfully" if data else "Using defaults")
```

Raising Custom Exceptions

```python
def validate_dimensions(width, height, length):
    """Validate structural dimensions."""
    if width <= 0 or height <= 0 or length <= 0:
        raise ValueError("Dimensions must be positive")

    if width > height * 3:
        raise ValueError("Width/height ratio exceeds limit")

    slenderness = length / min(width, height)
    if slenderness > 200:
        raise ValueError("Slenderness ratio too high")
    return True
# Usage
try:
    validate_dimensions(200, 400, 6000)
    print("Valid dimensions")
except ValueError as e:
    print(f"Error: {e}")
```

Complete Exception Handling: try-except-else-finally

```python
def process_analysis(input_file, output_file):
    """Complete error handling example."""
    file_handle = None
    try:
        file_handle = open(input_file, 'r')
        data = file_handle.read()
        with open(output_file, 'w') as f:
            f.write(analyze_structure(data))
    except FileNotFoundError:
        return False
    except Exception:
        return False
    else:
        return True
    finally:
        if file_handle:
            file_handle.close()
```

Understanding Iterators

What are Iterators?

Iterators provide a way to access elements of a collection sequentially without exposing the underlying structure:

- **Memory efficient**: Process one item at a time
- **Lazy evaluation**: Items generated only when needed
- **Uniform interface**: Same pattern for different data types

Engineering Applications

- Process large datasets of sensor readings
- Iterate through multiple design alternatives
- Handle streaming data from monitoring systems
- Generate sequences of load combinations

Basic Iterator Concepts

```python
material_costs = [150, 200, 180, 220, 160] # Lists are iterable

# --> Predict the output before running!
for cost in material_costs: # Direct iteration
    print(f"Material cost: {cost}")

# Manual iteration using iterator
cost_iterator = iter(material_costs)
print(next(cost_iterator))  # 150
print(next(cost_iterator))  # 200
print(next(cost_iterator))  # 180

# Range is an iterator (not a list!)
load_factors = range(1, 6)  # 1, 2, 3, 4, 5
print(type(load_factors))   # <class 'range'>

load_list = list(load_factors) # Convert to list if needed
print(load_list)  # [1, 2, 3, 4, 5]
```

Useful Iterator Functions: enumerate

```python
# Processing with position tracking
deflections = [2.5, 3.1, 1.8, 4.2, 2.9]  # mm
max_allow = 5.0  # mm

for i, defl in enumerate(deflections):
    beam_id = f"B{i+1:02d}"
    if defl > max_allow:
        status = "FAIL"
    elif defl > max_allow * 0.8:
        status = "WARN"
    else:
        status = "OK"
    print(f"{beam_id}: {defl:.1f} mm [{status}]")

# Find maximum
max_val = max(deflections)
max_idx = deflections.index(max_val)
print(f"Max: {max_val} mm at B{max_idx+1:02d}")
```

Useful Iterator Functions: zip

```python
# Combining related datasets with zip
beam_ids = ['B01', 'B02', 'B03']
moments = [120, 95, 140]  # kN*m
shears = [45, 38, 52]     # kN

print("Beam Analysis:")
for beam, M, V in zip(beam_ids, moments, shears):
    print(f"{beam}: M={M} kN*m, V={V} kN")

# Create summary dictionary
summary = {beam: {'moment': M, 'shear': V, 'ratio': M/150}
           for beam, M, V in zip(beam_ids, moments, shears)}

# Find critical beam
critical = max(summary.items(), key=lambda x: x[1]['ratio'])
print(f"Critical beam: {critical[0]} (ratio: {critical[1]['ratio']:.2f})")
```

Advanced Iterators: map and filter

```python
# Convert and filter loads
loads_kips = [12.5, 8.3, 15.7, 6.2, 11.4]
kips_to_kN = 4.448

loads_kN = list(map(lambda x: x * kips_to_kN, loads_kips)) # Convert to kN
print("kN:", [f"{load:.1f}" for load in loads_kN])

limit = 60 # Filter critical
critical = list(filter(lambda x: x > limit, loads_kN))
print(f"Critical (>{limit}):", critical)

def analyze(load): # Analysis
    sf = 80 / load
    return (load, sf, "OK" if sf >= 1.5 else "CRITICAL")

for load, sf, status in filter(lambda x: x[2] == "CRITICAL", map(analyze,
        loads_kN)):
    print(f"{load:.1f} kN, SF: {sf:.2f}")
```

```python
from itertools import combinations, product

dead = [50, 60]     # kN  # Load combinations
live = [30, 40]     # kN
wind = [20, 25]     # kN

for i, (D, L, W) in enumerate(product(dead, live, wind), 1):
    total = D + L + W
    print(f"LC{i}: {D}+{L}+{W} = {total} kN")

# Critical cases
critical = [(D, L, W) for D, L, W in product(dead, live, wind) if D+L+W >
    110]
print(f"Critical (>110): {len(critical)}")

members = [’A’, ’B’, ’C’, ’D’] # Connection pairs
connections = list(combinations(members, 2))
print(f"Connections: {len(connections)}")
```

List Comprehensions: Elegant List Creation

What are List Comprehensions?

A concise way to create lists by applying an expression to each item in an iterable:

- **Compact syntax**: Replace multiple lines with one
- **Readable**: Often more Pythonic than loops
- **Efficient**: Generally faster than equivalent loops
- **Functional style**: Express what you want, not how to get it

Engineering Applications

- Transform measurement units across datasets
- Filter structural members meeting design criteria
- Generate design parameter combinations
- Process sensor data with mathematical transformations

Basic List Comprehension Syntax

```
steel_grades = [250, 300, 350, 400, 450] # Traditional approach with loop
    # --> Challenge: Convert to one line!
yield_stresses = []
for grade in steel_grades:
    yield_stresses.append(grade * 1.0)  # MPa
print("Traditional:", yield_stresses)

yield_stresses_lc = [grade * 1.0 for grade in steel_grades]
print("List comp:", yield_stresses_lc) # List comprehension approach

beam_depths = [200, 250, 300, 350, 400]  # mm
beam_width = 150  # mm

section_moduli = [width * depth**2 / 6 for depth in beam_depths
                  for width in [beam_width]]

print("Section moduli (*10^3 mm^3):")
for depth, S in zip(beam_depths, section_moduli):
    print(f"Depth {depth} mm: S = {S/1000:.1f} *10^3 mm^3")
```

[PRACTICE] Your Turn: Transform Sensor Data

Individual Practice (3 minutes)

Process this sensor data using list comprehensions:

```python
# Raw strain gauge readings (microstrain)
raw_data = [245, -12, 389, 421, -5, 367, 298, 412, -8, 335]

# Task 1: Filter out negative values (noise)
valid_data = # YOUR CODE HERE

# Task 2: Convert to strain (divide by 1,000,000)
strain_values = # YOUR CODE HERE

# Task 3: Calculate stress (E = 200 GPa)
stress_MPa = # YOUR CODE HERE

# Task 4: Find readings above 70 MPa (one line!)
critical = # YOUR CODE HERE
```

List Comprehensions with Conditions

```python
# Filter and transform concrete test data
cylinders = [
    {'id': 'C01', 'strength': 28.5},
    {'id': 'C02', 'strength': 32.1},
    {'id': 'C03', 'strength': 24.8},
    {'id': 'C04', 'strength': 30.2}
]
# Filter acceptable cylinders (>= 25 MPa)
acceptable = [c['id'] for c in cylinders if c['strength'] >= 25.0]
print("Acceptable:", acceptable)

# Calculate ratios for acceptable cylinders
target = 30.0
ratios = [c['strength']/target for c in cylinders if c['strength'] >= 25.0]

for cyl_id, ratio in zip(acceptable, ratios):
    status = "OK" if ratio >= 1.0 else "Low"
    print(f"{cyl_id}: {ratio:.2f} ({status})")
```

Nested List Comprehensions

```python
# Load combination matrix
factors = {'dead': [1.2, 1.4], 'live': [1.6, 1.8], 'wind': [1.0, 1.3]}
base = {'dead': 100, 'live': 80, 'wind': 50}  # kN

# Nested comprehension
combinations = [
    {'case': f"LC{i+1}", 'total': base['dead']*df + base['live']*lf +
        base['wind']*wf}
    for i, (df, lf, wf) in enumerate([
        (df, lf, wf) for df in factors['dead']
        for lf in factors['live'] for wf in factors['wind']
    ])
]
# Critical cases
critical = [lc for lc in combinations if lc['total'] > 300]
print(f"Critical (>300): {len(critical)}")
for lc in critical[:2]:
    print(f"{lc['case']}: {lc['total']:.1f} kN")
```

Advanced List Comprehensions: Conditional Expressions

```python
# Beam classification with conditional expressions
moments = [85, 120, 95, 140, 75, 160]  # kN*m
design_moment = 125  # kN*m

# Classifications
classes = [f"B{i+1}: {m} kN*m ({'OK' if m <= design_moment else 'OVER'})"
           for i, m in enumerate(moments)]
for c in classes[:3]:
    print(c)

# Reinforcement ratios
ratios = [m/design_moment * 0.01 if m <= design_moment else m/design_moment * 0.015
          for m in moments]

for i, (m, rho) in enumerate(zip(moments[:3], ratios[:3])):
    area = rho * 200 * 400
    print(f"B{i+1}: rho={rho:.4f}, As={area:.0f} mm^2")
```

Set and Dictionary Comprehensions

```
grades = [250, 300, 250, 350, 300, 400] # Set comprehension - unique values
unique = {grade for grade in grades}
print("Unique grades:", sorted(unique))

# Dictionary comprehension
steel_grades = [250, 300, 350, 400]
properties = {
    grade: {'fy': grade, 'fu': grade * 1.3, 'E': 200000}
    for grade in steel_grades
}

# Display properties
for grade, props in list(properties.items())[:3]:
    print(f"Grade {grade}: fy={props['fy']}, fu={props['fu']:.0f}")

# Safety factors
sf = {grade: 2.5 if grade < 350 else 2.2 for grade in steel_grades}
print("SF:", sf)
```

Generators: Memory-Efficient Iterators

What are Generators?

Generators are special iterators that generate values on-the-fly:

- **Memory efficient**: Don't store all values in memory
- **Lazy evaluation**: Values computed only when needed
- **Single-use**: Can only be iterated once
- **Infinite sequences**: Can represent unbounded data

Engineering Applications

- Process large datasets without loading everything into memory
- Generate infinite sequences of design parameters
- Stream real-time sensor data
- Create custom iteration patterns for optimization algorithms

Generator Expressions vs List Comprehensions

```python
loads = [10, 15, 20, 25] # List vs Generator

list_sq = [load**2 for load in loads] # List - all in memory
print("List:", list_sq)

gen_sq = (load**2 for load in loads) # Generator - on demand
print("Gen:", gen_sq)

# Use generator
for sq in gen_sq:
    print(f"^2: {sq}")

# Exhausted after use
print("Reuse:", list(gen_sq))  # Empty!

import sys # Memory
print(f"List: {sys.getsizeof([x**2 for x in range(100)])} bytes")
print(f"Gen: {sys.getsizeof((x**2 for x in range(100)))} bytes")
```

Generator Functions with yield - Part 1

```python
def fibonacci_generator(max_value):
    """Generate Fibonacci sequence."""
    a, b = 0, 1
    while a <= max_value:
        yield a
        a, b = b, a + b

def load_combo_generator(dead, live_loads, factors):
    """Generate load combinations."""
    for live in live_loads:
        for factor in factors:
            total = dead + live * factor
            yield {'dead': dead, 'live': live, 'factor': factor, 'total':
                total}
```

Generator Functions

Generator functions use `yield` instead of `return` to produce a sequence of values

Generator Functions with yield - Part 2

```
# Using generator functions
print("Fibonacci numbers <= 100:")
fib_gen = fibonacci_generator(100)
for num in fib_gen:
    print(num, end=' ')

print("\n\nLoad combinations:")
load_gen = load_combination_generator(
    dead_load=50,  # kN
    live_loads=[30, 40, 50],  # kN
    load_factors=[1.2, 1.4, 1.6]
)

for i, combo in enumerate(load_gen, 1):
    if combo['total'] > 100:  # Filter critical combinations
        print(f"LC{i:02d}: {combo['dead']} +
            {combo['live']}*{combo['factor']} "
              f"= {combo['total']:.1f} kN")
```

Advanced Generator: Prime Sieve - Part 1

```python
def sieve_of_eratosthenes(limit):
    """Generate prime numbers using Sieve algorithm."""
    is_prime = [True] * (limit + 1)
    is_prime[0] = is_prime[1] = False
    for num in range(2, int(limit**0.5) + 1):
        if is_prime[num]:
            for multiple in range(num * num, limit + 1, num):
                is_prime[multiple] = False
    for num in range(2, limit + 1):
        if is_prime[num]:
            yield num

primes = sieve_of_eratosthenes(30) # Test the generator
print("Primes up to 30:", list(primes))
```

Sieve Algorithm

Efficiently generates prime numbers by eliminating multiples

Advanced Generator: Prime Sieve - Part 2

```python
def structural_optimization_sequence():
    """Generate optimization parameters using prime spacing."""
    primes = sieve_of_eratosthenes(50)
    base_dim = 200  # mm

    for prime in primes:
        if prime > 10:
            width = base_dim + prime * 5
            yield {'width': width, 'height': width * 1.5}
print("Optimization sequence:") # Generate structural dimensions
opt_gen = structural_optimization_sequence()
for i, params in enumerate(opt_gen):
    if i >= 4: break
    print(f"Option {i+1}: {params['width']}*{params['height']} mm")
```

Application

Using mathematical sequences for engineering optimization parameters

Generator State Preservation

```python
def load_test(max_load, increment):
    """Simulate loading with state preservation."""
    load, step = 0, 0
    while load <= max_load:
        step += 1
        stress = load / 10  # MPa
        status = 'elastic' if stress < 250 else 'plastic'
        yield {'step': step, 'load': load, 'stress': stress, 'status':
            status}
        load += increment
test = load_test(max_load=600, increment=100) # Run test
for result in test:
    print(f"Step {result['step']}: {result['load']} kN ->
        {result['stress']:.1f} MPa [{result['status']}]")
    if result['status'] == 'plastic':
        print("Yield reached")
        break
print("State preserved")
```

[COMPETITION] Mini-Challenge: Load Combination Optimizer

Challenge (5 minutes)

Write the most efficient code to generate and analyze load combinations!

```
# Given loads and factors
dead_loads = [100, 120]  # kN
live_loads = [40, 50, 60]  # kN
load_factors = {'dead': 1.2, 'live': 1.6, 'combo': 1.4}

# YOUR CHALLENGE:
# 1. Generate all combinations: D*1.2 + L*1.6 and (D+L)*1.4
# 2. Find the maximum factored load
# 3. Count how many exceed 200 kN
# 4. Use generators or comprehensions for efficiency!

# YOUR CODE HERE (aim for < 5 lines!)
```

What We've Covered This Week

Control Flow

- if/elif/else statements
- for and while loops
- break, continue, and else clauses

Functions

- Function definition and calling
- Parameters and return values
- Default arguments and *args/**kwargs

Error Handling

- Exception types and handling
- try/except/else/finally

Iterators

- Iterator protocol and concepts
- enumerate, zip, map, filter
- itertools module

List Comprehensions

- Basic syntax and patterns
- Conditional expressions
- Nested comprehensions

Generators

- Generator expressions
- Generator functions with yield

Key Programming Concepts Mastered

Core Programming Skills

- **Structured Programming**: Control flow and modular design
- **Functional Programming**: Functions as first-class objects
- **Error Resilience**: Robust code with proper exception handling
- **Efficient Iteration**: Memory-conscious data processing
- **Pythonic Code**: Idiomatic Python patterns and best practices

Engineering Problem-Solving Skills

- Design parameter optimization with generators
- Load combination analysis with iterators
- Robust structural calculations with error handling
- Efficient data processing with comprehensions
- Modular engineering functions for reusability

Next Week Preview: Modules and Data Science

Week 3 Topics

- **Modules and Packages**: Organizing larger projects
- **String Processing**: Text manipulation and regular expressions
- **Introduction to NumPy**: Numerical computing fundamentals
- **Introduction to Pandas**: Data analysis and manipulation
- **Basic Plotting**: Data visualization with Matplotlib

Engineering Applications Preview

- Process structural analysis reports (text processing)
- Handle large datasets of sensor measurements (NumPy)
- Analyze construction material test results (Pandas)
- Create engineering plots and visualizations (Matplotlib)

Practice Exercises

1. **Control Flow**: Write a program that classifies structural members based on slenderness ratios using nested if statements
2. **Functions**: Create a function library for common structural calculations (beam deflection, column capacity, etc.)
3. **Error Handling**: Implement robust input validation for a structural design program
4. **Iterators**: Process a large dataset of material test results using memory-efficient iteration
5. **List Comprehensions**: Generate load combinations and filter critical cases using comprehensions
6. **Generators**: Create a generator that produces optimization parameters for structural design

Challenge Project

Combine all concepts to create a structural analysis program with proper error handling, modular functions, and efficient data processing

Questions?

Thank you!

Dr. Eyuphan Koc
eyuphan.koc@bogazici.edu.tr

Next Week: Modules, Packages, and Introduction to Data Science Libraries