Advanced Python Metaprogramming Using AST and Decorators for Runtime Code Transformation in Scalable AI Systems

---

High-Ranking Keywords Covered:

- advanced python examples
- python metaprogramming
- python AST tutorial
- python runtime code transformation
- python decorators for runtime logic
- python code instrumentation
- scalable ai system python
- maang-level python interview code
- rare python tricks
- how to impress hr with python

Filename: ast_code_transformer.py

"""

Advanced Python: Dynamic AST Code Transformation Using Metaprogramming

Author: Z. M. Sultana

Target Audience: Senior Python Developers, AI Engineers, MAANG Recruiters

This script showcases a rare and powerful Python technique: using Abstract Syntax Trees (AST) and decorators to dynamically modify functions at runtime.

Why it matters:

Demonstrates metaprogramming expertise

Enables runtime code instrumentation

Ideal for AI/ML profiling or live monitoring

Rare knowledge domain — perfect for high-tier interviews

"""

```
import ast
import inspect
import textwrap
import types
from functools import wraps

# ==============================
# Custom AST Node Transformer
# ==============================

class TimeLoggerInjector(ast.NodeTransformer):
    """
    Injects timing logs into any Python function dynamically using AST.
    Adds logging around the body of the function without altering original code manually.
```

```
    """

    def visit_FunctionDef(self, node):
        import_line = ast.parse("import time").body[0]
        start_time = ast.parse("start = time.time()").body[0]
        end_time = ast.parse(
            'print(f"[DEBUG] Execution time of function \'{node.name}\': {time.time() - start:.6f}s")'
        ).body[0]

        node.body.insert(0, start_time)
        node.body.append(end_time)
        return [import_line, node]

# ============================
# Runtime Code Transformer
# ============================

def transform_function(func):
    """
    Transforms the given function's AST to include timing logs.
    """
    # Get source and parse into AST
    source = textwrap.dedent(inspect.getsource(func))
    tree = ast.parse(source)
```

```
    # Modify AST using custom transformer
    transformer = TimeLoggerInjector()
    transformed_tree = transformer.visit(tree)
    ast.fix_missing_locations(transformed_tree)

    # Compile and evaluate transformed code
    code = compile(transformed_tree, filename="<ast>", mode="exec")
    new_env = {}
    exec(code, func.__globals__, new_env)
    return new_env[func.__name__]

# ================================
# Decorator Using AST Modifier
# ================================

def auto_profile(func):
    """
    Decorator that replaces the function at runtime with an AST-modified version
    that includes detailed execution time logging.
    """
    modified_func = transform_function(func)

    @wraps(func)
    def wrapper(*args, **kwargs):
        return modified_func(*args, **kwargs)
```

```
    return wrapper

# ============================
# Advanced Example Use Case
# ============================

@auto_profile
def simulate_ai_pipeline():
    """
    Simulates an AI pipeline stage such as:
    - Data cleaning
    - Model inference
    - Post-processing
    """

    print("[INFO] Starting AI inference simulation...")
    import time
    time.sleep(1.5)  # Simulate heavy computation
    print("[INFO] Model inference complete.")

if __name__ == "__main__":
    simulate_ai_pipeline()
```

Key Takeaways for MAANG HR & Engineers

Feature	Purpose
ast.NodeTransformer	Enables deep Python introspection and transformation
inspect.getsource()	Extracts function source dynamically
compile() + exec()	Re-injects transformed code safely
Decorators	Wraps logic without modifying the original callsite
Use Case	Perfect for profiling, logging, debugging in AI/ML pipelines