Self Programming Language Explanation

Table of Contents

1. Overview
2. Why Python?
3. Architecture & Design
4. Required Capabilities
5. Advanced Features
6. Control Structures
7. Parent Relationships & Method Invocation
8. Building and Running
9. Examples
10. Implementation Notes

1 Overview

This is a complete implementation of the Self programming language mechanics as described in "SELF: The Power of Simplicity" by Ungar & Smith (1987). Self is a prototype-based object-oriented language that simplifies object-oriented programming by eliminating classes and unifying state access with message passing.

Key Principles of Self

1. **Everything is an object** – No primitive types separate from objects
2. **Objects contain slots** – Named slots hold references to other objects
3. **Message passing is fundamental** – No direct variable access, only messages
4. **Inheritance through parents** – Some slots are marked as "parents"
5. **Prototypes instead of classes** – Objects are created by cloning existing objects
6. **Behavioral simplicity** – Fewer concepts, more expressive power

2 Why Python?

I chose Python for this implementation for several reasons:

Advantages:

- **Dynamic typing** matches Self's runtime typing approach
- **Flexible object model** allows easy implementation of slot-based objects
- **Rich data structures** (dictionaries, sets) perfect for implementing slots and parent relationships
- **Garbage collection** handles memory management automatically like Self
- **Clear syntax** makes the implementation easy to understand and maintain
- **Built-in deep copying** supports Self's clone operation
- **Exception handling** helps implement proper error handling for message lookup

Trade-offs:

- **Performance** – Python is interpreted, so slower than a native implementation
- **Memory overhead** – Python objects have more overhead than optimal Self objects
- **Not Self-hosting** – This is an implementation *of* Self, not *in* Self

3 Architecture & Design

Core Classes

SelfObject

The central class representing all objects in the Self system. Contains:

- `slots`: Dictionary mapping slot names to other `SelfObject` instances
- `parent_slots`: Set of slot names that are marked as parents for inheritance
- `messages`: Optional list of messages to send when object is evaluated (for methods)
- `primitive_value`: Optional primitive data (int, float, string, etc.)
- `primitive_function`: Optional Python function to execute when evaluated

SelfSystem

High-level utility class that creates common objects like numbers and points, and manages the root object that provides basic behavior shared by all objects.

Key Design Decisions

1. **Unified Object Model**: All entities (numbers, methods, classes) are `SelfObject` instances with different configurations
2. **Receiver Context**: Method invocation properly sets up receiver context using the "self" slot
3. **Breadth-First Search**: Parent lookup uses BFS to handle multiple inheritance correctly
4. **Cycle Detection**: Object printing and inheritance chains handle cycles properly
5. **Deep Copying**: Clone operations create truly independent copies of objects

4 Required Capabilities

All 8 required capabilities are fully implemented and tested:

1. `evaluate()` Evaluates an object according to Self semantics:

- Objects with message lists: copy self, send messages, return last result
- Objects with primitive functions: copy self and call function with receiver context
- Objects with primitive values: return copy with same value
- Regular objects: return self

2. `copy()` Creates a deep copy of an object, implementing Self's fundamental clone operation. All slots, parent relationships, and data are copied to create an independent object.

3. `send_message(message_name)` Sends a message to an object:

- Look up slot with `message_name` in this object
- If not found, search parent objects breadth-first
- Evaluate the found object with proper receiver context
- Return the result

4. `send_message_with_parameter(message_name, parameter)` Sends a message with a parameter:

- Find the slot object as above
- Copy the slot object and set its "parameter" slot
- Evaluate with receiver context
- Return the result

5. `assign_slot(slot_name, obj)` Assigns an object to a named slot. This is how state is stored in Self since there are no variables.

6. `make_parent(slot_name)` Marks an existing slot as a parent slot for inheritance. The slot must exist before being marked as a parent.

7. `assign_parent_slot(slot_name, obj)` Convenience method that combines `assign_slot()` and `make_parent()` - assigns an object to a slot and marks it as a parent.

8. `print_object()` Produces a hierarchical string representation of the object showing:

    - Primitive values and functions
    - Message lists
    - Parent slots (marked)
    - All slots with proper indentation
    - Handles circular references

5 Advanced Features

Multiple Inheritance

Objects can have multiple parent slots, and the breadth-first search will find methods from any parent in the inheritance hierarchy.

```
child = SelfObject()
parent1 = SelfObject()
parent2 = SelfObject()

parent1.assign_slot("method1", SelfObject(primitive_value="from_parent1"))
parent2.assign_slot("method2", SelfObject(primitive_value="from_parent2"))

child.assign_parent_slot("p1", parent1)
child.assign_parent_slot("p2", parent2)

# Child can now access both methods
result1 = child.send_message("method1")  # "from_parent1"
result2 = child.send_message("method2")  # "from_parent2"
```

Cyclic Inheritance

The implementation handles cyclic inheritance relationships properly by tracking visited objects during breadth-first search.

Breadth-First Slot Lookup

When a message is not found in the immediate object, the search continues through all parents at the current level before moving to grandparents. This ensures predictable method resolution order.

Primitive Functions

Built-in operations like arithmetic are implemented as primitive functions that operate on objects with primitive values:

- `primitive_add()` – Addition
- `primitive_subtract()` – Subtraction
- `primitive_multiply()` – Multiplication
- `primitive_less_than()` – Comparison
- `primitive_equals()` – Equality
- `primitive_clone()` – Cloning
- `primitive_print()` – Printing

6 Control Structures

Self implements control structures using its uniform object model, without special syntax:

If-Then-Else Statements

Boolean objects (`true` and `false`) have `ifTrue:` methods with different behavior:

```
# True object executes the parameter (true block)
def true_if_true(obj):
    return obj.slots["parameter"].evaluate()

# False object does nothing (returns self)
def false_if_true(obj):
    return obj.slots["self"]

true_obj = SelfObject(primitive_value=True)
true_obj.assign_slot("ifTrue:", SelfObject(primitive_function=true_if_true))

false_obj = SelfObject(primitive_value=False)
false_obj.assign_slot("ifTrue:", SelfObject(primitive_function=false_if_true))
```

Recursion

Recursion works naturally through message sending. Objects can send messages to themselves:

```
# Factorial implementation
def factorial_compute(obj):
    n = obj.slots["parameter"].primitive_value
    if n <= 1:
        return SelfObject(primitive_value=1)
    else:
        n_minus_1 = SelfObject(primitive_value=n - 1)
        sub_result = obj.slots["self"].send_message_with_parameter("compute:", n_m
        return SelfObject(primitive_value=n * sub_result.primitive_value)
```

```
factorial = SelfObject()
factorial.assign_slot("compute:", SelfObject(primitive_function=factorial_compute))

# Usage: factorial.send_message_with_parameter("compute:", SelfObject(primitive_val
```

Loops

Loops are implemented using recursion and conditionals:

```
# Countdown loop
def countdown(obj):
    n = obj.slots["parameter"].primitive_value
    print(f"Count: {n}")
    if n > 0:
        n_minus_1 = SelfObject(primitive_value=n - 1)
        return obj.slots["self"].send_message_with_parameter("countdown:", n_minus_
    else:
        return SelfObject(primitive_value=0)

counter = SelfObject()
counter.assign_slot("countdown:", SelfObject(primitive_function=countdown))
```

7 Parent Relationships & Method Invocation

The Problem

When a method (function body) is invoked, it needs access to:

1. The receiver object (the object that received the message)
2. Local variables and parameters
3. The lexical environment where the method was defined

Self's Solution

In Self, when a message is sent:

1. **Method Lookup**: Find the method object in the receiver's slots or parents
2. **Method Copy**: Copy the method object (creating an activation record)
3. **Context Setup**: Set the copy's parent to the receiver
4. **Parameter Binding**: Set the "parameter" slot with the argument
5. **Evaluation**: Evaluate the method copy

This approach means:

- **Method objects are prototypes** of activation records
- **Each invocation creates a new activation** by copying
- **The parent link provides access** to the receiver's slots
- **Local variables are slots** in the activation record

Why This Matters

This design enables:

1. **Closure-like behavior** – Methods capture their environment
2. **Consistent object model** – No special cases for methods vs objects
3. **Dynamic method modification** – Methods are regular objects that can be changed
4. **Inheritance of behavior** – Methods can be inherited through parent links

Alternative Approaches

Other possible approaches and their trade-offs:

1. **Stack-based activation records** (like C/Java)
    - More memory efficient
    - Faster invocation
    - But less flexible, harder to implement closures
2. **Lexical scoping with environments** (like Lisp)
    - Clear separation of concerns
    - Well-understood semantics
    - But requires separate environment mechanism
3. **Class-based method dispatch** (like Smalltalk)
    - Familiar model
    - Efficient implementation
    - But less uniform, requires separate class concept

Self's approach provides **maximum uniformity and flexibility** at the cost of some performance overhead.

8 Building and Running

Requirements

- Python 3.6 or later
- No external dependencies (uses only Python standard library)

Running the Implementation

```
# Make the file executable
chmod +x self_implementation.py

# Run the comprehensive tests and demonstrations
python3 self_implementation.py
```

Using as a Module

```
from self_implementation import SelfSystem, SelfObject

# Create a Self system
system = SelfSystem()

# Create and use objects
num1 = system.create_number(10)
num2 = system.create_number(5)
result = num1.send_message_with_parameter("+", num2)
print(f"10 + 5 = {result.primitive_value}")

# Create points
point1 = system.create_point(3, 4)
point2 = system.create_point(1, 2)
point_sum = point1.send_message_with_parameter("+", point2)
point_sum.send_message("printString")  # Prints: Point(4, 6)
```

9 Examples

Creating a Simple Object

```
# Create an object with some slots
person = SelfObject()
person.assign_slot("name", SelfObject(primitive_value="Alice"))
person.assign_slot("age", SelfObject(primitive_value=30))

# Access slots by sending messages
```

```
name = person.send_message("name")
print(name.primitive_value)  # "Alice"
```

Implementing Inheritance

```
# Create a parent object with shared behavior
animal = SelfObject()

def animal_speak(obj):
    print("Some animal sound")
    return obj.slots["self"]

animal.assign_slot("speak", SelfObject(primitive_function=animal_speak))

# Create a child that inherits from animal
dog = SelfObject()
dog.assign_parent_slot("parent", animal)
dog.assign_slot("name", SelfObject(primitive_value="Rex"))

# Dog can use inherited method
dog.send_message("speak")  # Prints: "Some animal sound"
```

Creating Custom Methods

```
# Create a greeting method
def greet(obj):
    receiver = obj.slots["self"]
    name = receiver.send_message("name")
    print(f"Hello, I'm {name.primitive_value}")
    return receiver

greeter = SelfObject()
greeter.assign_slot("name", SelfObject(primitive_value="Bob"))
greeter.assign_slot("greet", SelfObject(primitive_function=greet))

greeter.send_message("greet")  # Prints: "Hello, I'm Bob"
```

10 Implementation Notes

Performance Considerations

This implementation prioritizes correctness and clarity over performance:

- **Deep copying** is used for safety but is expensive

- **Dictionary lookups** are used for slots (could be optimized with arrays)
- **Breadth-first search** is thorough but not optimized for common cases
- **No caching** of method lookups or optimized dispatch

Memory Management

Python's garbage collection handles memory management automatically. In a production Self implementation, you would need:

- **Generational garbage collection** for handling object cycles
- **Weak references** for some parent relationships to break cycles
- **Copy-on-write** semantics for efficient cloning

Error Handling

The implementation includes proper error handling for:

- **Missing messages** – throws AttributeError
- **Type mismatches** – throws TypeError for arithmetic on non-numbers
- **Invalid operations** – throws ValueError for malformed operations

Testing

The implementation includes comprehensive tests covering:

- All 8 required capabilities
- Multiple inheritance scenarios
- Cyclic inheritance handling
- Control structure implementations
- Primitive function operations
- Object creation and cloning

Extensions

Possible extensions to this implementation:

- **Garbage collection visualization** to show object relationships
- **Debugger integration** to step through message sending

- **Performance profiling** to identify optimization opportunities
- **Syntax parser** to accept Self-like textual syntax
- **Persistence** to save and restore object worlds
- **Distribution** to handle objects across network boundaries

11 Conclusion

This implementation demonstrates that Self's radical simplification of object-oriented programming is both feasible and powerful. By eliminating classes and unifying state access with message passing, Self provides a remarkably expressive foundation for object-oriented computation.

The key insights from Self that influenced later languages include:

1. **Prototype-based inheritance** (used in JavaScript)
2. **Uniform message passing** (influenced method dispatch optimizations)
3. **Dynamic object modification** (influenced dynamic languages)
4. **Closure-like methods** (influenced function objects in Python and others)

This implementation captures the essence of Self's design while providing a solid foundation for further experimentation with prototype-based programming.