Introducing: Noggin

By: Joshua Newberry

What Makes Noggin Different?

- Noggin uses JavaScript/C++ type syntax, but with Python keywords and structure; looks like C, runs like Python
  - No type declaration for variables or return types
  - Uses semicolons and curly braces
- Noggin has a high-functioning interactive mode with memory, commands like “clear”, “allclear”, and “goback” to go along with the normal run by file mode
- Noggin includes native functions like min(), max(), and str()
- Noggin treats print as a function using parentheses, and allows for theoretically infinite arguments to string together in sequence. Ex. print(a, b, c, d, e, f, ... y, z);

Running from a file, and classes/str()

```
def fact(n)
{
    if (n <= 1)
    {
        return 1;
    }
    return n * fact(n - 1);
}

print("5! = " + str(fact(5)));
```

```
class Counter
{
    init(start)
    {
        this.value = start;
    }

    inc()
    {
        this.value = this.value + 1;
        return this.value;
    }
}

def c = Counter(10);
print(c.value);
print(c.inc());
print(c.inc());
print(c.inc());
c.value = 50;
print(c.value);
```

Improved REPL Shell and min()/max()

```
class Noggin:
    def __init__(self):
        self.environment = Environment()
        self.history = []
        self.clear = lambda: os.system('cls')
        self.previous_environment = []
        self.previous_history = []

    def run_file(self, path):
        with open(path) as f:
            self.run(f.read())

    def run_prompt(self):
        try:
            print(">>>>> PNoggin Interactive Shell <<<<<")
            while True:
                self.run(input("> "))
                ErrorHandler.had_error = False
                ErrorHandler.had_runtime_error = False
        except KeyboardInterrupt:
            print("\nExiting PNoggin Interactive Shell.")
```

```
def run(self, source):
    if source == "allclear":
        self.clear()
        self.previous_environment.append(self.environment)
        self.environment = Environment()
        self.previous_history.append(self.history)
        self.history = []
        return
    if source == "clear":
        self.clear()
        return
    if source == "goback":
        if len(self.previous_environment) >= 1:
            self.environment = self.previous_environment.pop()
        else:
            print("Cannot revert to a previous environment")
        if len(self.previous_history) >= 1:
            self.history = self.previous_history.pop()
        else:
            print("Cannot revert to a previous history")
        return
    scanner = Scanner(source, len(self.history))
    tokens = scanner.scan_tokens()
    parser = Parser(tokens)
    statements = parser.parse()
    if ErrorHandler.had_error:
        return
    interpreter = Interpreter(self.environment)
    resolver = None
    if isinstance(statements, List):
        resolver = Resolver(interpreter)
        resolver.resolve(self.history + statements)
    if ErrorHandler.had_error:
        return
    interpreter.interpret(statements)
    if ErrorHandler.had_runtime_error:
        return
    if resolver is not None:
        self.history.append(statements)
```

Statements/Control Flow

```
class Stmt:
    pass

class Expression(Stmt):
    def __init__(self, expression):
        self.expression = expression

class Print(Stmt):
    def __init__(self, exprList:List):
        self.exprList = exprList

class Def(Stmt):
    def __init__(self, name:Token, initializer):
        self.name = name
        self.initializer = initializer

class Class(Stmt):
    def __init__(self, name:Token, methods:List):
        self.name = name
        self.methods = methods

class Assignment(Stmt):
    def __init__(self, name:Token, expression):
        self.name = name
        self.expression = expression
```

```
class Function(Stmt):
    def __init__(self, name:Token, parameters:List, block:Block):
        self.name = name
        self.parameters = parameters
        self.block = block

class Return(Stmt):
    def __init__(self, keyword:Token, value=None):
        self.keyword = keyword
        self.value = value

class If(Stmt):
    def __init__(self, condition, block:Block, Else:None|If|Else = None):
        self.condition = condition
        self.block = block
        self.Else = Else

class Else(Stmt):
    def __init__(self, block:Block):
        self.block = block

class While(Stmt):
    def __init__(self, condition, block:Block):
        self.condition = condition
        self.block = block
```

Parsing for Statements

```
def statement(self) -> Print|Def|Class|Function|Assignment|Block|If|While|Return|Set|Expression:
    if self.match(TokenType.PRINT): ## Print
        return self.printStatement()
    elif self.match(TokenType.DEF): ## Declare and Function
        return self.declaration()
    elif self.match(TokenType.CLASS):
        return self.klass()
    elif self.is_start_of_assignment(): ## Assign
        return self.assignmentStatement()
    elif self.match(TokenType.LEFT_BRACE): ## Block
        return self.block()
    elif self.match(TokenType.IF): ## If
        return self.ifStatement()
    elif self.match(TokenType.WHILE): ## While
        return self.whileStatement()
    elif self.match(TokenType.FOR): ## For
        return self.forStatement()
    elif self.match(TokenType.RETURN):
        return self.returnStatement()
    else: ## Expression of some other kind
        return self.expressionStatement()
```

Cont.

- If, Else If, Else statements
- While loops
- For loops are sort of just a While loop

```
def ifStatement(self):
    self.consume(TokenType.LEFT_PAREN, "Expect '(' after if token")
    condition = self.expression()
    self.consume(TokenType.RIGHT_PAREN, "Expect ')' after if condition")
    self.consume(TokenType.LEFT_BRACE, "Expect '{' before if statement contents")
    block = self.block()
    if self.match(TokenType.ELSE):
        if self.match(TokenType.IF):
            return If(condition, block, self.ifStatement())
        self.consume(TokenType.LEFT_BRACE, "Expect '{' before else statement contents"
        else_block = self.block()
        return If(condition, block, Else(else_block))
    return If(condition, block)

def whileStatement(self):
    self.consume(TokenType.LEFT_PAREN, "Expect '(' after while token")
    condition = self.expression()
    self.consume(TokenType.RIGHT_PAREN, "Expect ')' after while expression")
    self.consume(TokenType.LEFT_BRACE, "Expect '{' before while statement contents")
    block = self.block()
    return While(condition, block)

def forStatement(self):
    self.consume(TokenType.LEFT_PAREN, "Expect '(' after for token")
    definition = None
    if self.match(TokenType.DEF):
        definition = self.varDeclaration()
    condition = self.expression()
    self.consume(TokenType.SEMICOLON, "Expect ';' after loop condition")
    modifier = self.statement()
    self.consume(TokenType.RIGHT_PAREN, "Expect ')' after for expression")
    self.consume(TokenType.LEFT_BRACE, "Expect '{' before for statement contents")
    block = self.block()
    block.statements.append(modifier)
    if definition:
        return Block([definition, # Define
                      While(condition, block)])
    return Block([While(condition, block)])
```

Conclusions/Challenges

- Type hinting can help immensely when debugging
- One mistake can break large portions of code
- Building a language can be broken down into smaller chunks if you choose a structure to follow
- Many features of modern languages feel “invisible” until you try making your own
    - Well implemented features are the ones that don’t always get noticed