CPYTHON INTERPRETER

A MINI PROJECT REPORT

Submitted by

Shaik Hussain Ahamed[RA2011003010439]
S.Jahnavi [RA2011003010457]

Under the guidance of

Dr.Jeya R

(Assistant Professor, Department of
ComputingTechnologies)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

APRIL 2023

INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report **“Cpython Interpreter”** is the bonafide work of **“Shaik Hussain Ahamed(RA2011003010439) , S. Jahnavi (RA2011003010457)”** of III Year/VI Sem B. Tech (CSE) who carried outthe mini project work under my supervision for the course 18CSC304J- Compiler Design in SRM Institute of Science and Technology during the academic year 2022-2023 (Even semester).

SIGNATURE	**SIGNATURE**
Dr.Jeya R	Dr.Pushpalatha M
Assistant Professor	Head of Department
Department of computing	School of Computing

TABLE OF CONTENT

1. ABSTRACT

CPython is the default and most widely used implementation of the Python programming language. It is an interpreter that executes Python code and provides a runtime environment for Python applications. CPython is written in C and is open source software, which means that its source code is freely available for modification and redistribution. The interpreter is designed to be highly portable and can be run on various operating systems, including Windows, macOS, and Linux. CPython includes a standard library that provides a wide range of functionality, from basic data types and containers to modules for working with databases, web protocols, and GUIs. Overall, CPython is a powerful and flexible interpreter that has played a critical role in making Python one of the most popular programming languages in use today.

CHAPTER 2

EXISTING SYSTEMS

2.1 Existing Systems and Drawbacks

The Python programming language, which is extensively used in business and open-source communities, is now implemented in the CPython interpreter system. The high-level, interpreted language CPython was created with simplicity, readability, and quick development in mind. It is written in C and serves as the standard implementation of the Python language.

A comprehensive standard library for CPython is available, with modules for file I/O, networking, threading, and more. It works with many different operating systems, including Windows, macOS, Linux, and many more. Additionally, CPython has a sizable and vibrant developer and user community that actively contributes to the growth of the language and its ecosystem.

Python code is parsed and instantly interpreted by the CPython interpreter before being executed. It creates optimised bytecode using a bytecode compiler, and the Python virtual machine interprets that code. Additionally, the interpreter has a garbage collector that controls memory deallocation and allocation automatically.

Every release of CPython adds new features and upgrades as it is being developed and enhanced. CPython is maintained and developed by the Python core development team, with assistance from the larger community.

In conclusion, the current CPython interpreter system provides a mature and reliable environment for Python development, with a significant user base and developer community.

Drawbacks:

Although CPython is a well-known and frequently used version of the Python language, the current setup still has significant flaws and restrictions, such as:

Performance: CPython's performance, particularly for CPU-intensive operations, can be a bottleneck for some applications. Thread concurrency is restricted by the Global Interpreter Lock (GIL) in CPython, which can also have an impact on how well multi-threaded programmes execute.

Memory Usage: Large-scale applications may experience performance challenges due to memory fragmentation and ineffective memory management in CPython.

Limited Concurrency: As was already established, CPython's GIL restricts thread concurrency, which might be problematic for multi-core machines.

Lack of Support for Alternative Platforms: CPython supports a broad variety of platforms, but not all hardware architectures, especially those used in embedded systems or portable electronic devices.

Issues with Compatibility with Other Implementations: Although CPython is the standard implementation of the Python programming language, there are other alternatives available, including PyPy and Jython, which may not be entirely compatible with the CPython ecosystem and libraries.

Lack of Support for Asynchronous Programming: Although the asyncio library in CPython provides some support for asynchronous programming, it may not be as efficient or flexible as other languages that have native asynchronous programming support.

Lack of Static Typing Support: CPython lacks built-in support for static typing, which can cause problems with readability and maintainability of code, especially for big codebases.

The CPython development team has made it a priority to address these issues, and they are constantly working to enhance concurrency, memory utilisation, performance, and compatibility with various systems and implementations.

2.2 Gap Identified Problem Statement

Gap Identification:

Alternative implementations, such PyPy and Jython, have been developed as a result of the CPython interpreter's drawbacks. These solutions seek to improve efficiency and parallelism while addressing some of CPython's drawbacks.

A high-performance Python implementation that nevertheless maintains compatibility with the CPython ecosystem and libraries, however, is currently lacking in the market.

Problem Proposition:

The issue is that Python applications' scalability and performance might be negatively impacted by the existing CPython implementation's performance, memory usage, and concurrency restrictions. Alternative implementations might not be compatible with the CPython ecosystem and libraries, despite the fact that they do have some advantages.

2.3 Objectives

Performance: The CPython interpreter strives to deliver fast Python programme execution. This entails making the virtual machine, compiler, and parser as efficient as feasible in order to run Python code.

A wide number of Python features and modules are supported by the CPython interpreter, which tries to be compliant with the Python language standard.

Extensibility: The goal of the CPython interpreter is to be extensible, enabling programmers to create and use unique modules and extensions written in C or other languages.

Reliability: With a low chance of crashes or other mistakes, the CPython interpreter strives to be dependable and stable.

Cross-platform compatibility: The CPython interpreter intends to support a wide range of hardware platforms and operating systems, enabling the creation and execution of Python programmes on a number of different devices and architectures.

Open source: Because the CPython interpreter is open source, programmers can contribute to its development and enhance its performance and usefulness.

CHAPTER 3

PROPOSED METHODOLOGY

Several phases go into the creation of CPython, including planning, design, coding, testing, and release. Agile and Waterfall techniques are combined in the CPython development process.

Agile development practises place a strong emphasis on incremental and iterative development, frequent releases, and close cooperation between developers and stakeholders. This strategy is used by the Python core development team, which updates Python on a regular basis and engages in open communication with the public to solicit feedback and proposals for enhancements.

The team also manages the production of significant releases using the Waterfall technique at the same time. The Waterfall model is a sequential design process that follows a linear approach; each stage must be finished before going on to the next. For large Python releases that require sizable changes to the language, this technique is employed.

The following steps are part of the CPython development process:

Planning: The Python core development team decides which enhancements and new features will be incorporated into the upcoming version.

Design: The group chooses the best ways to implement the updated functions and features.

Coding: The new features and enhancements are coded by the developers.

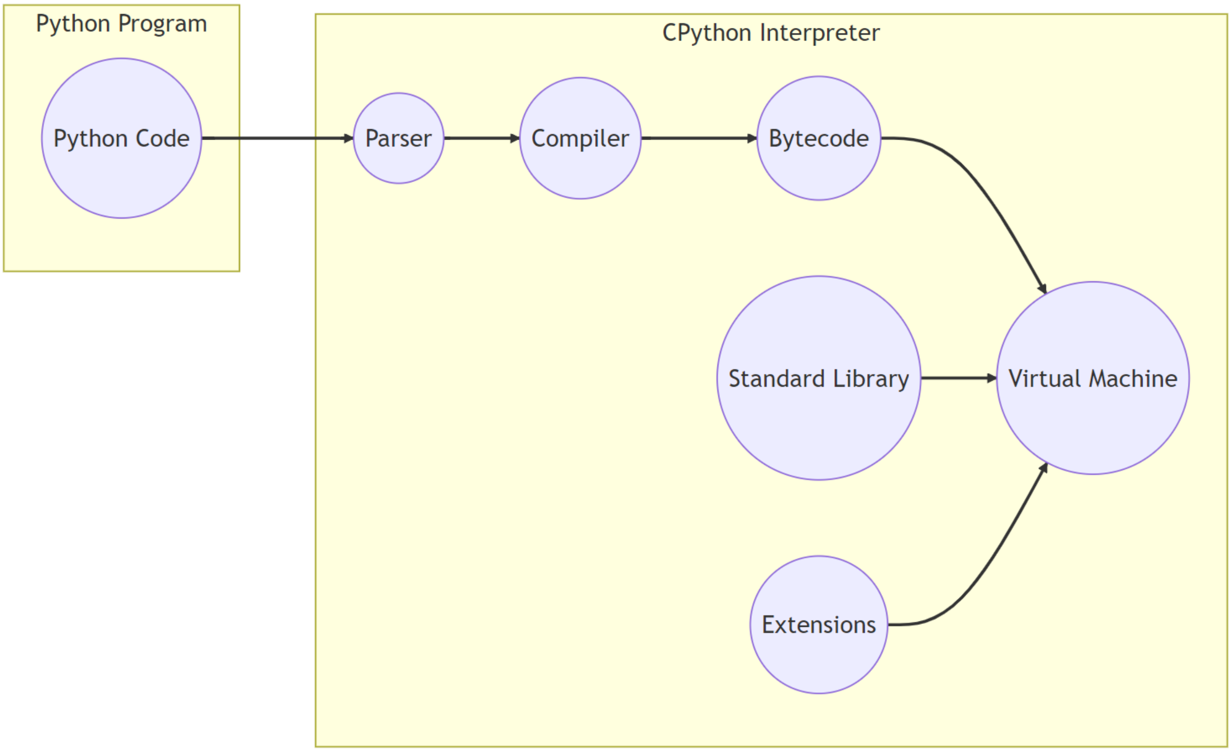
Testing: To make sure the new code functions as intended and doesn't add any errors, the developers test it.

Release: After extensive testing and evaluation, the new code is made available to public.

CHAPTER 4

ARCHITECTURE AND DESCRIPTION

Architecture:



Description:

The Python programming language is implemented by the CPython interpreter as the standard. It is written in C and offers a platform for running Python programmes on a range of hardware and operating systems.

There are numerous key elements that make up the CPython interpreter's architecture:

Parser: The parser analyses the source code for Python and produces a number of Abstract Syntax Trees (ASTs) that show the organisation of the programme. The compiler is then given the ASTs.

Compiler: The compiler converts the ASTs the parser produced into bytecode. The interpreter can run the program's lower-level representation, or bytecode, on the fly.

Virtual Machine (VM): The bytecode produced by the compiler is executed by the virtual machine (VM). It features a garbage collector for memory management and a stack-based execution mechanism.

The CPython interpreter comes with a robust standard library that offers a variety of features, such as file I/O, networking, regular expressions, and more.

Extension Mechanisms: CPython has a number of extension mechanisms, including the ctypes module, which enables calling shared library functions directly from Python code, and C extensions, which are compiled as shared libraries and loaded dynamically into the interpreter at runtime.

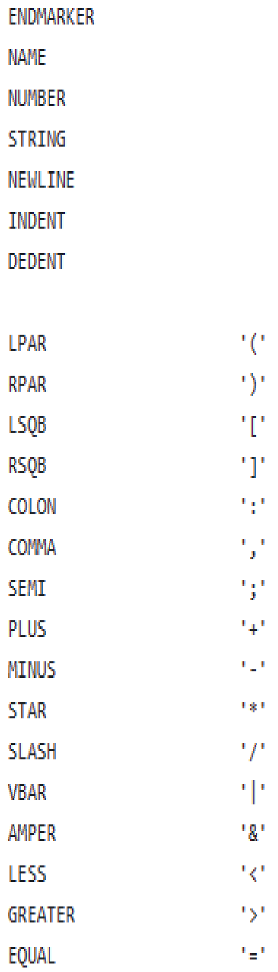
The CPython interpreter's architecture is made to be very flexible and extendable. The extension techniques enable developers to construct unique extensions and interface with other languages, while the standard library offers a robust set of functionality that can be utilised in Python programmes.

Overall, the CPython interpreter's architecture offers a strong and adaptable platform for running Python code on a variety of operating systems and for a variety of applications.

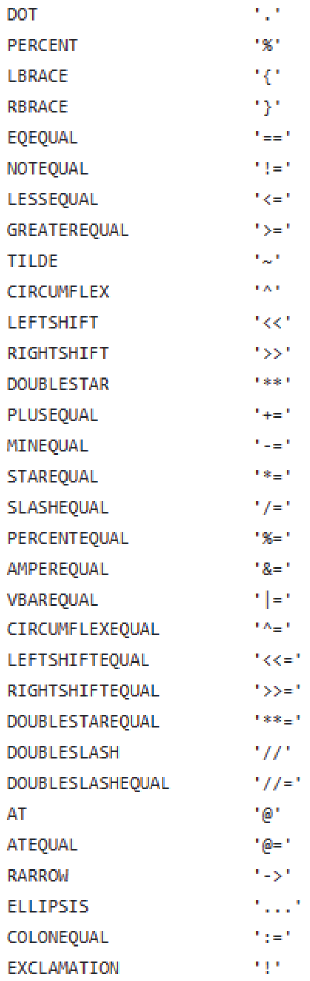
CHAPTER 5

MODULES DESCRIPTION
AND IMPLEMENTATION

5.1 Grammar-Tokens:

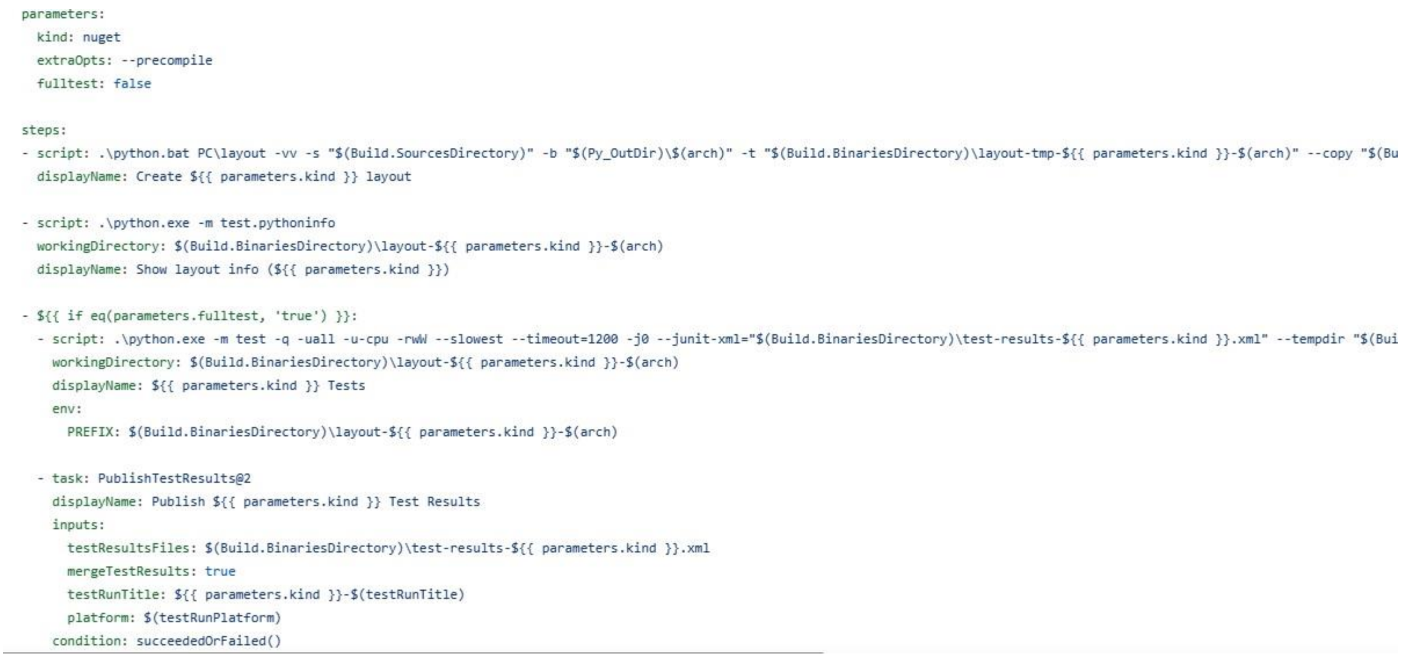


```
ENDMARKER
NAME
NUMBER
STRING
NEWLINE
INDENT
DEDENT

LPAR                    '('
RPAR                    ')'
LSQB                    '['
RSQB                    ']'
COLON                   ':'
COMMA                   ','
SEMI                    ';'
PLUS                    '+'
MINUS                   '-'
STAR                    '*'
SLASH                   '/'
VBAR                    '|'
AMPER                   '&'
LESS                    '<'
GREATER                 '>'
EQUAL                   '='
```



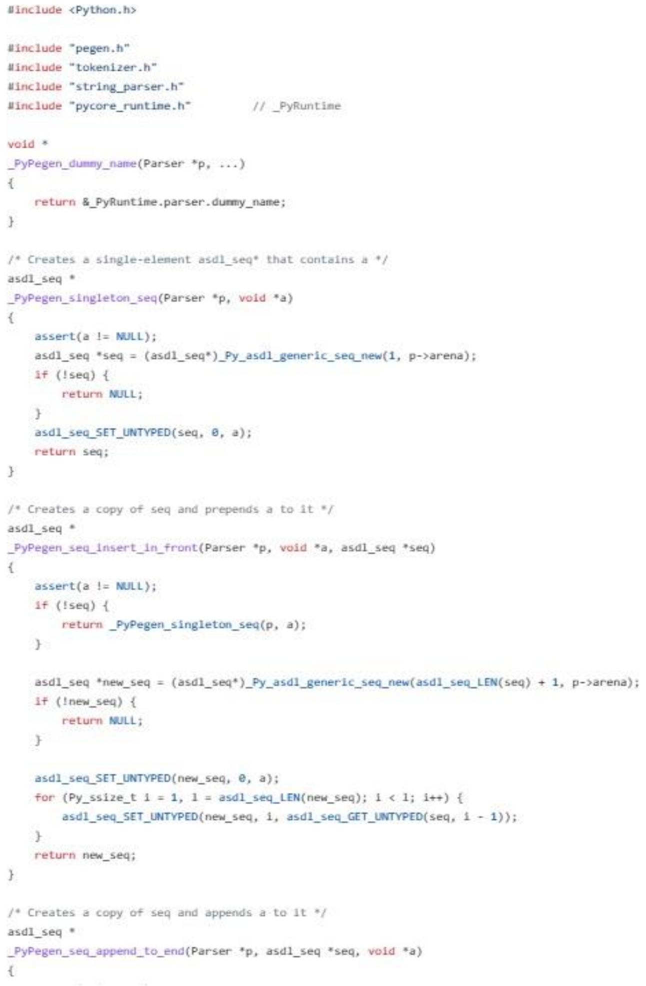
```
DOT                     '.'
PERCENT                 '%'
LBRACE                  '{'
RBRACE                  '}'
EQEQUAL                 '=='
NOTEQUAL                '!='
LESSEQUAL               '<='
GREATEREQUAL            '>='
TILDE                   '~'
CIRCUMFLEX              '^'
LEFTSHIFT               '<<'
RIGHTSHIFT              '>>'
DOUBLESTAR              '**'
PLUSEQUAL               '+='
MINEQUAL                '-='
STAREQUAL               '*='
SLASHEQUAL              '/='
PERCENTEQUAL            '%='
AMPEREQUAL              '&='
VBAREQUAL               '|='
CIRCUMFLEXEQUAL         '^='
LEFTSHIFTEQUAL          '<<='
RIGHTSHIFTEQUAL         '>>='
DOUBLESTAREQUAL         '**='
DOUBLESLASH             '//'
DOUBLESLASHEQUAL        '//='
AT                      '@'
ATEQUAL                 '@='
RARROW                  '->'
ELLIPSIS                '...'
COLONEQUAL              ':='
EXCLAMATION             '!'
```

5.2 Windows-layout-steps:

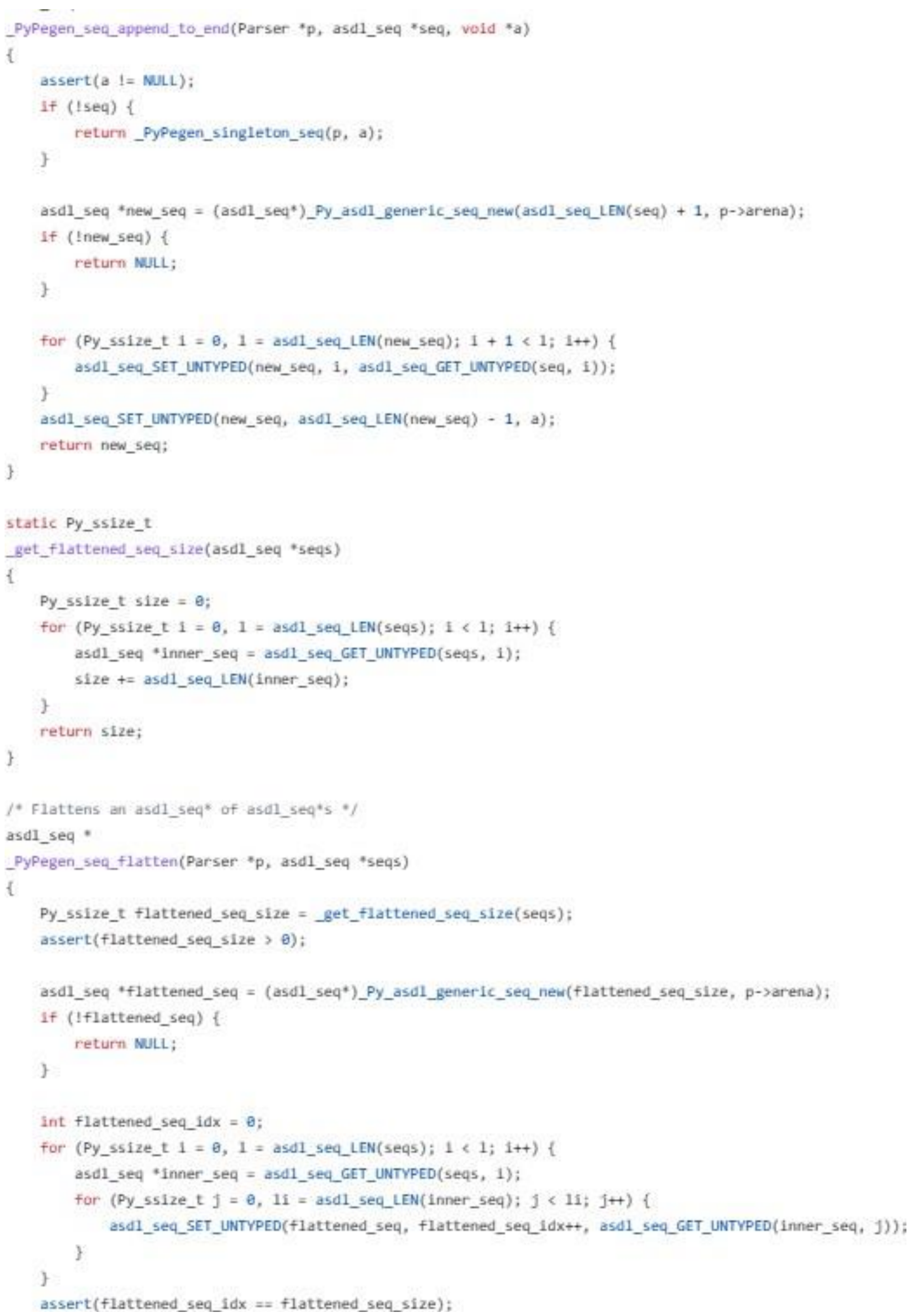


```yaml
parameters:
  kind: nuget
  extraOpts: --precompile
  fulltest: false

steps:
- script: .\python.bat PC\layout -vv -s "$(Build.SourcesDirectory)" -b "$(Py_OutDir)\$(arch)" -t "$(Build.BinariesDirectory)\layout-tmp-${{ parameters.kind }}-$(arch)" --copy "$(Bu
  displayName: Create ${{ parameters.kind }} layout

- script: .\python.exe -m test.pythoninfo
  workingDirectory: $(Build.BinariesDirectory)\layout-${{ parameters.kind }}-$(arch)
  displayName: Show layout info (${{ parameters.kind }})

- ${{ if eq(parameters.fulltest, 'true') }}:
  - script: .\python.exe -m test -q -uall -u-cpu -rwW --slowest --timeout=1200 -j0 --junit-xml="$(Build.BinariesDirectory)\test-results-${{ parameters.kind }}.xml" --tempdir "$(Bui
    workingDirectory: $(Build.BinariesDirectory)\layout-${{ parameters.kind }}-$(arch)
    displayName: ${{ parameters.kind }} Tests
    env:
      PREFIX: $(Build.BinariesDirectory)\layout-${{ parameters.kind }}-$(arch)

  - task: PublishTestResults@2
    displayName: Publish ${{ parameters.kind }} Test Results
    inputs:
      testResultsFiles: $(Build.BinariesDirectory)\test-results-${{ parameters.kind }}.xml
      mergeTestResults: true
      testRunTitle: ${{ parameters.kind }}-$(testRunTitle)
      platform: $(testRunPlatform)
    condition: succeededOrFailed()
```

5.3 Parser_actions_help:



```c
#include <Python.h>

#include "pegen.h"
#include "tokenizer.h"
#include "string_parser.h"
#include "pycore_runtime.h"         // _PyRuntime

void *
_PyPegen_dummy_name(Parser *p, ...)
{
    return &_PyRuntime.parser.dummy_name;
}

/* Creates a single-element asdl_seq* that contains a */
asdl_seq *
_PyPegen_singleton_seq(Parser *p, void *a)
{
    assert(a != NULL);
    asdl_seq *seq = (asdl_seq*)_Py_asdl_generic_seq_new(1, p->arena);
    if (!seq) {
        return NULL;
    }
    asdl_seq_SET_UNTYPED(seq, 0, a);
    return seq;
}

/* Creates a copy of seq and prepends a to it */
asdl_seq *
_PyPegen_seq_insert_in_front(Parser *p, void *a, asdl_seq *seq)
{
    assert(a != NULL);
    if (!seq) {
        return _PyPegen_singleton_seq(p, a);
    }

    asdl_seq *new_seq = (asdl_seq*)_Py_asdl_generic_seq_new(asdl_seq_LEN(seq) + 1, p->arena);
    if (!new_seq) {
        return NULL;
    }

    asdl_seq_SET_UNTYPED(new_seq, 0, a);
    for (Py_ssize_t i = 1, l = asdl_seq_LEN(new_seq); i < l; i++) {
        asdl_seq_SET_UNTYPED(new_seq, i, asdl_seq_GET_UNTYPED(seq, i - 1));
    }
    return new_seq;
}

/* Creates a copy of seq and appends a to it */
asdl_seq *
_PyPegen_seq_append_to_end(Parser *p, asdl_seq *seq, void *a)
{
```

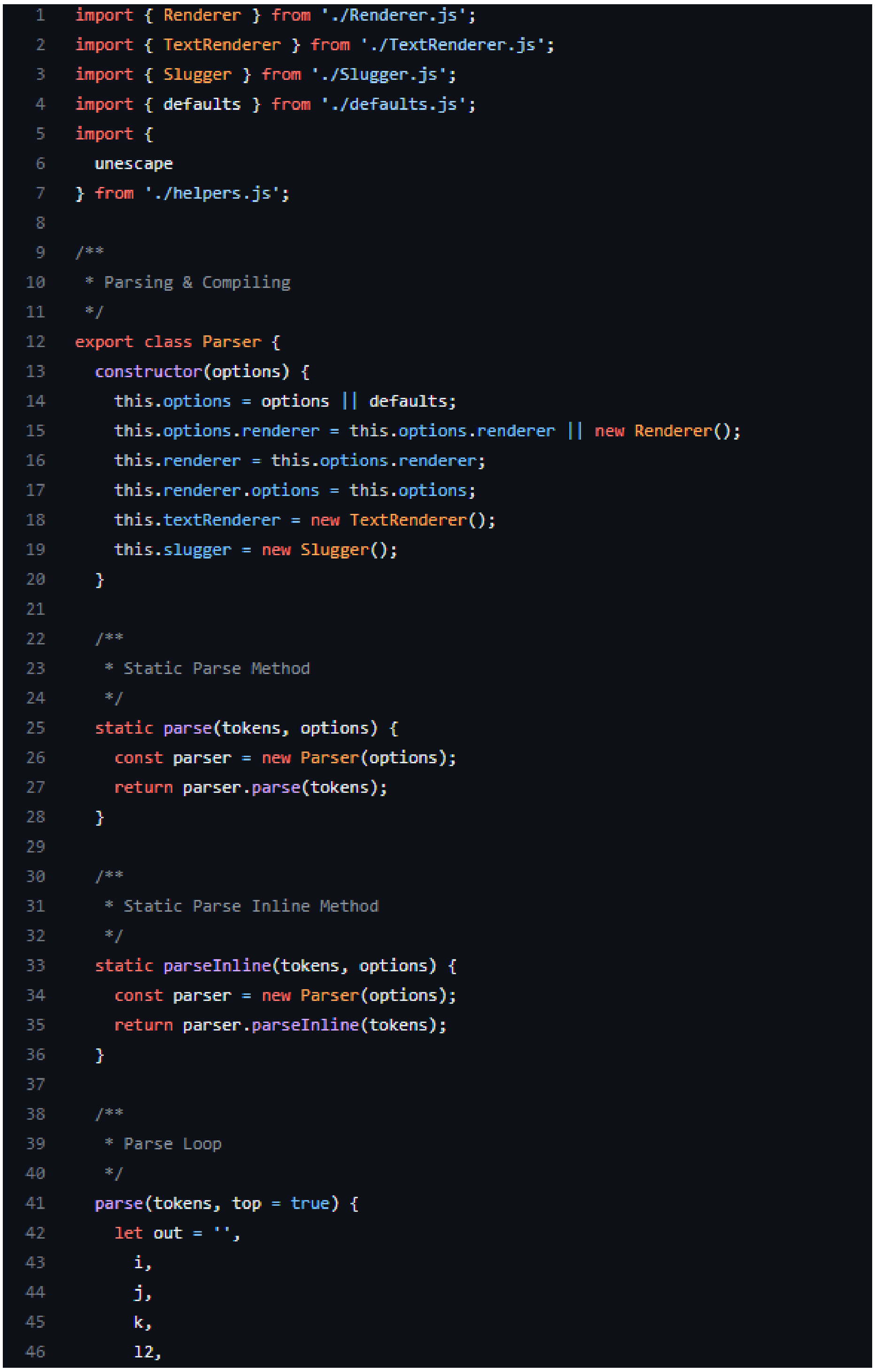


```
_PyPegen_seq_append_to_end(Parser *p, asdl_seq *seq, void *a)
{
    assert(a != NULL);
    if (!seq) {
        return _PyPegen_singleton_seq(p, a);
    }

    asdl_seq *new_seq = (asdl_seq*)_Py_asdl_generic_seq_new(asdl_seq_LEN(seq) + 1, p->arena);
    if (!new_seq) {
        return NULL;
    }

    for (Py_ssize_t i = 0, l = asdl_seq_LEN(new_seq); i + 1 < l; i++) {
        asdl_seq_SET_UNTYPED(new_seq, i, asdl_seq_GET_UNTYPED(seq, i));
    }
    asdl_seq_SET_UNTYPED(new_seq, asdl_seq_LEN(new_seq) - 1, a);
    return new_seq;
}

static Py_ssize_t
_get_flattened_seq_size(asdl_seq *seqs)
{
    Py_ssize_t size = 0;
    for (Py_ssize_t i = 0, l = asdl_seq_LEN(seqs); i < l; i++) {
        asdl_seq *inner_seq = asdl_seq_GET_UNTYPED(seqs, i);
        size += asdl_seq_LEN(inner_seq);
    }
    return size;
}

/* Flattens an asdl_seq* of asdl_seq*s */
asdl_seq *
_PyPegen_seq_flatten(Parser *p, asdl_seq *seqs)
{
    Py_ssize_t flattened_seq_size = _get_flattened_seq_size(seqs);
    assert(flattened_seq_size > 0);

    asdl_seq *flattened_seq = (asdl_seq*)_Py_asdl_generic_seq_new(flattened_seq_size, p->arena);
    if (!flattened_seq) {
        return NULL;
    }

    int flattened_seq_idx = 0;
    for (Py_ssize_t i = 0, l = asdl_seq_LEN(seqs); i < l; i++) {
        asdl_seq *inner_seq = asdl_seq_GET_UNTYPED(seqs, i);
        for (Py_ssize_t j = 0, li = asdl_seq_LEN(inner_seq); j < li; j++) {
            asdl_seq_SET_UNTYPED(flattened_seq, flattened_seq_idx++, asdl_seq_GET_UNTYPED(inner_seq, j));
        }
    }
    assert(flattened_seq_idx == flattened_seq_size);
```

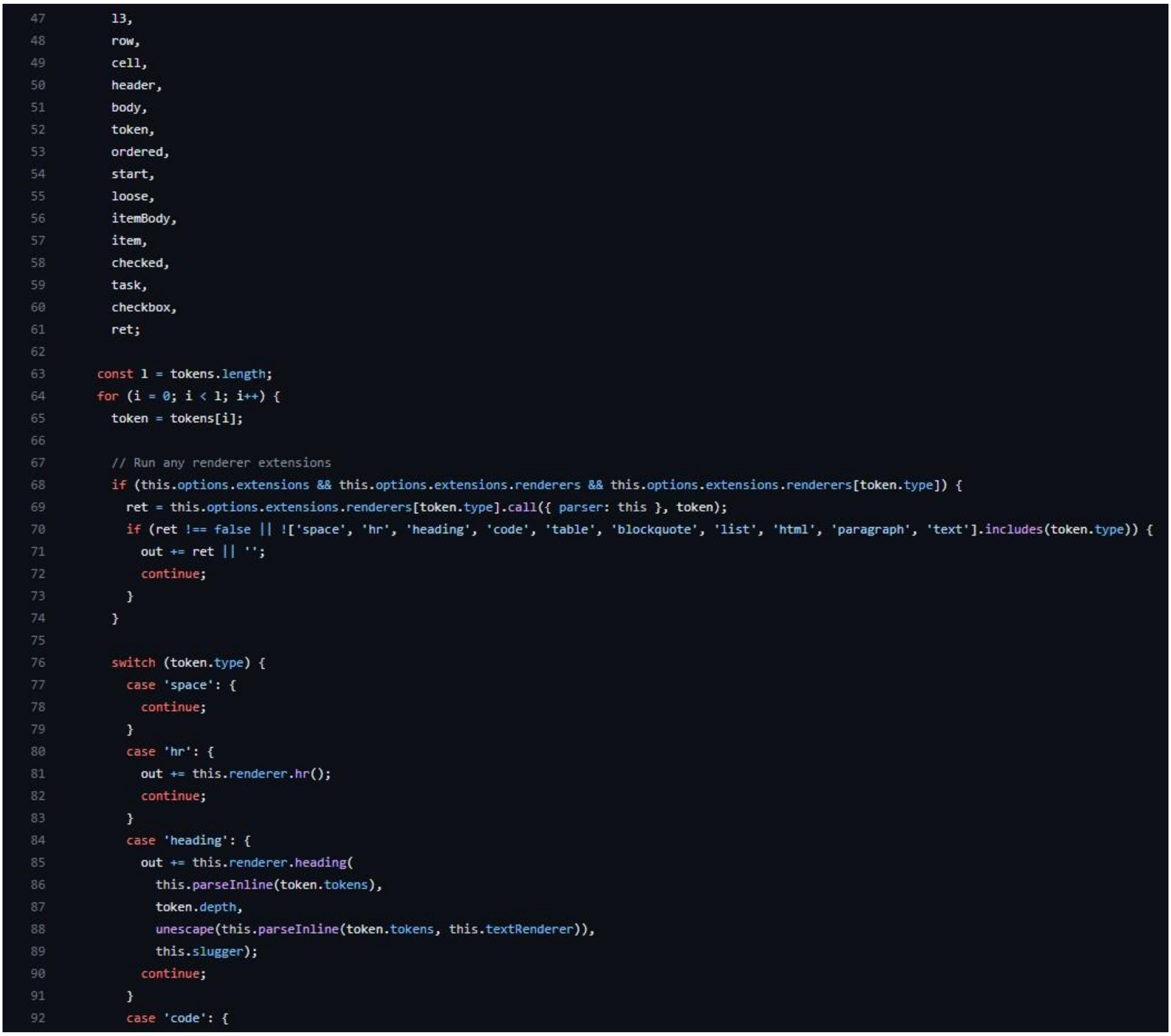
CHAPTER 6

RESULTS AND DISCUSSION

6.1 Parser

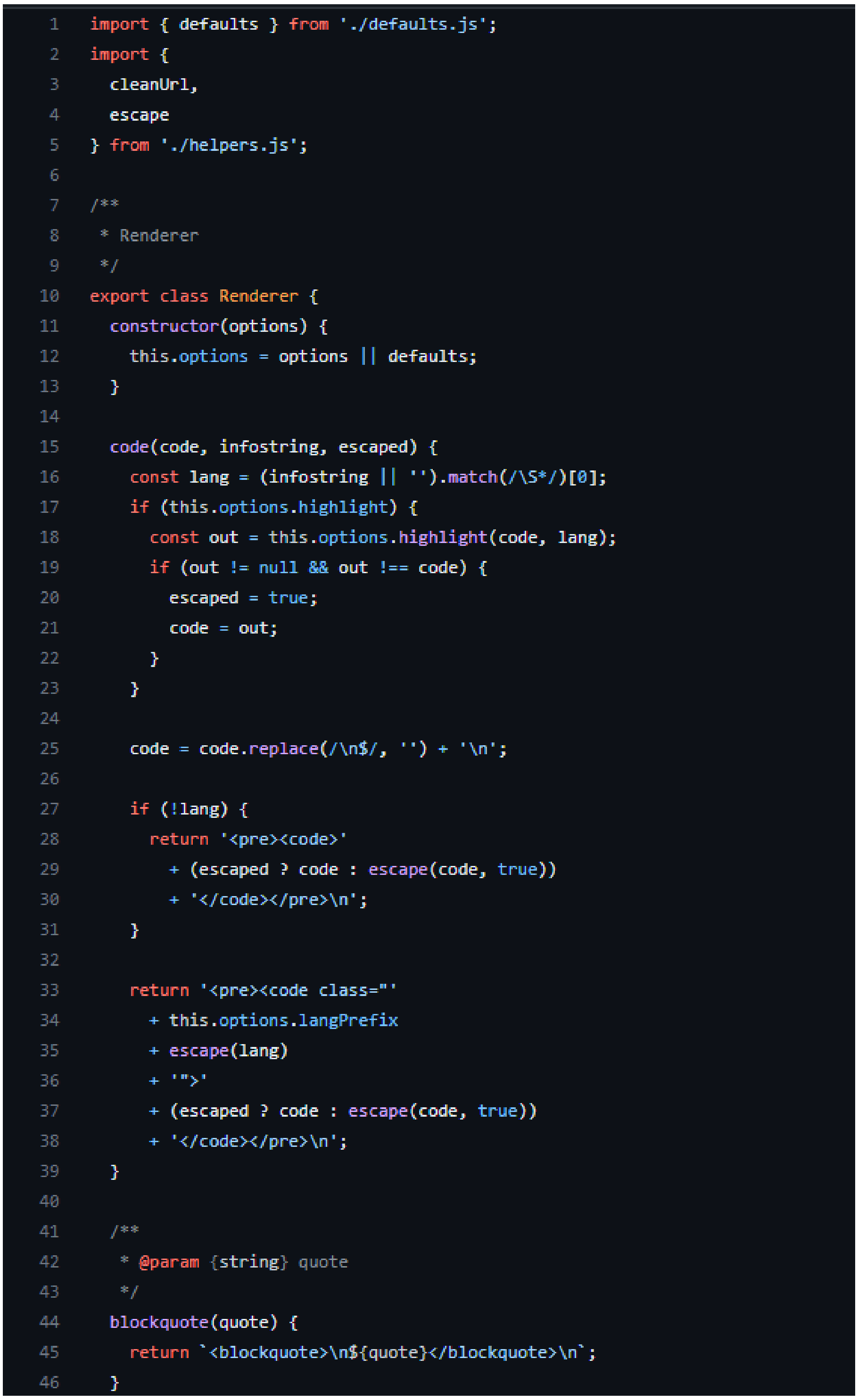


```javascript
import { Renderer } from './Renderer.js';
import { TextRenderer } from './TextRenderer.js';
import { Slugger } from './Slugger.js';
import { defaults } from './defaults.js';
import {
  unescape
} from './helpers.js';

/**
 * Parsing & Compiling
 */
export class Parser {
  constructor(options) {
    this.options = options || defaults;
    this.options.renderer = this.options.renderer || new Renderer();
    this.renderer = this.options.renderer;
    this.renderer.options = this.options;
    this.textRenderer = new TextRenderer();
    this.slugger = new Slugger();
  }

  /**
   * Static Parse Method
   */
  static parse(tokens, options) {
    const parser = new Parser(options);
    return parser.parse(tokens);
  }

  /**
   * Static Parse Inline Method
   */
  static parseInline(tokens, options) {
    const parser = new Parser(options);
    return parser.parseInline(tokens);
  }

  /**
   * Parse Loop
   */
  parse(tokens, top = true) {
    let out = '',
      i,
      j,
      k,
      l2,
```

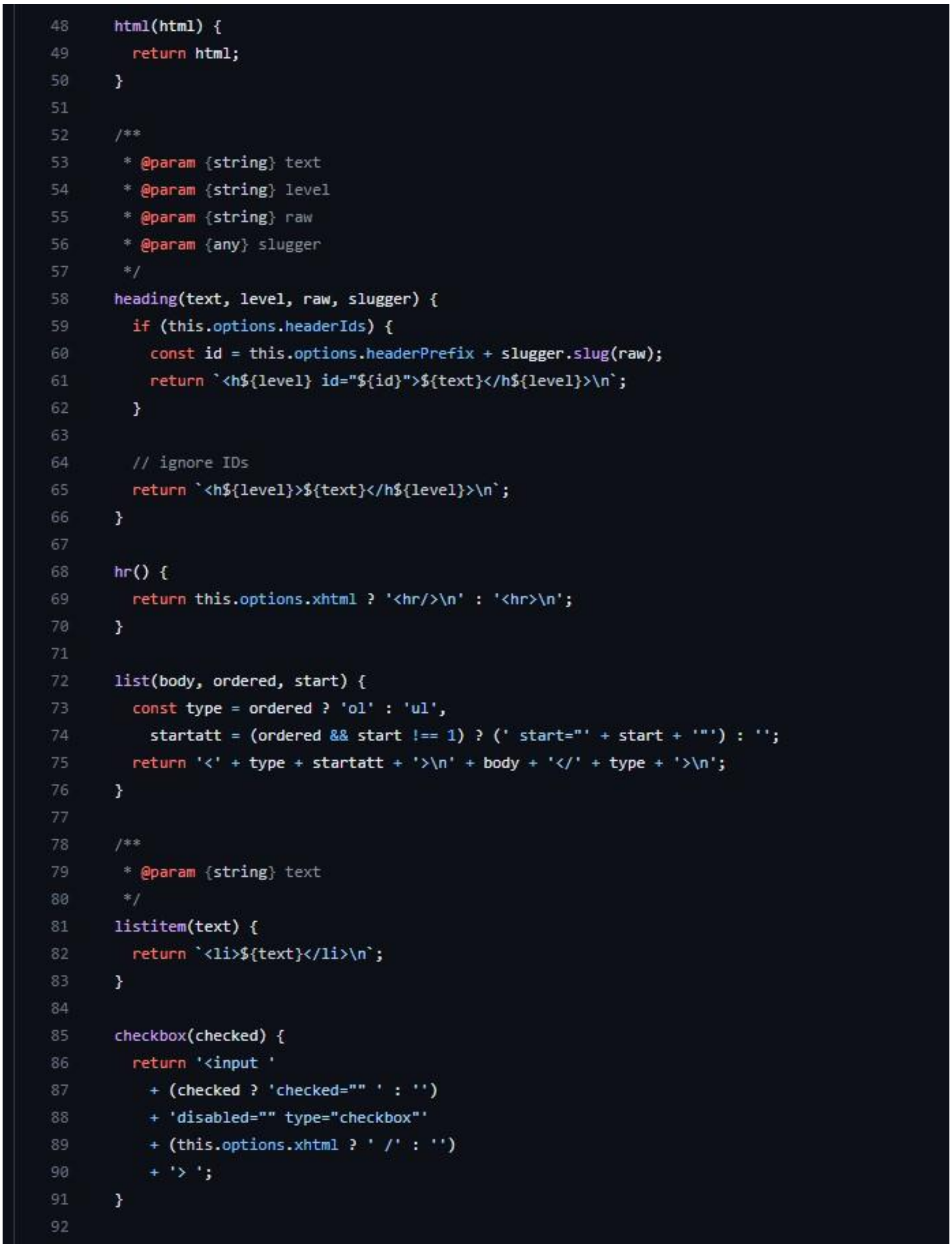


```
      l3,
      row,
      cell,
      header,
      body,
      token,
      ordered,
      start,
      loose,
      itemBody,
      item,
      checked,
      task,
      checkbox,
      ret;

    const l = tokens.length;
    for (i = 0; i < l; i++) {
      token = tokens[i];

      // Run any renderer extensions
      if (this.options.extensions && this.options.extensions.renderers && this.options.extensions.renderers[token.type]) {
        ret = this.options.extensions.renderers[token.type].call({ parser: this }, token);
        if (ret !== false || !['space', 'hr', 'heading', 'code', 'table', 'blockquote', 'list', 'html', 'paragraph', 'text'].includes(token.type)) {
          out += ret || '';
          continue;
        }
      }

      switch (token.type) {
        case 'space': {
          continue;
        }
        case 'hr': {
          out += this.renderer.hr();
          continue;
        }
        case 'heading': {
          out += this.renderer.heading(
            this.parseInline(token.tokens),
            token.depth,
            unescape(this.parseInline(token.tokens, this.textRenderer)),
            this.slugger);
          continue;
        }
        case 'code': {
```

6.2 Renderer

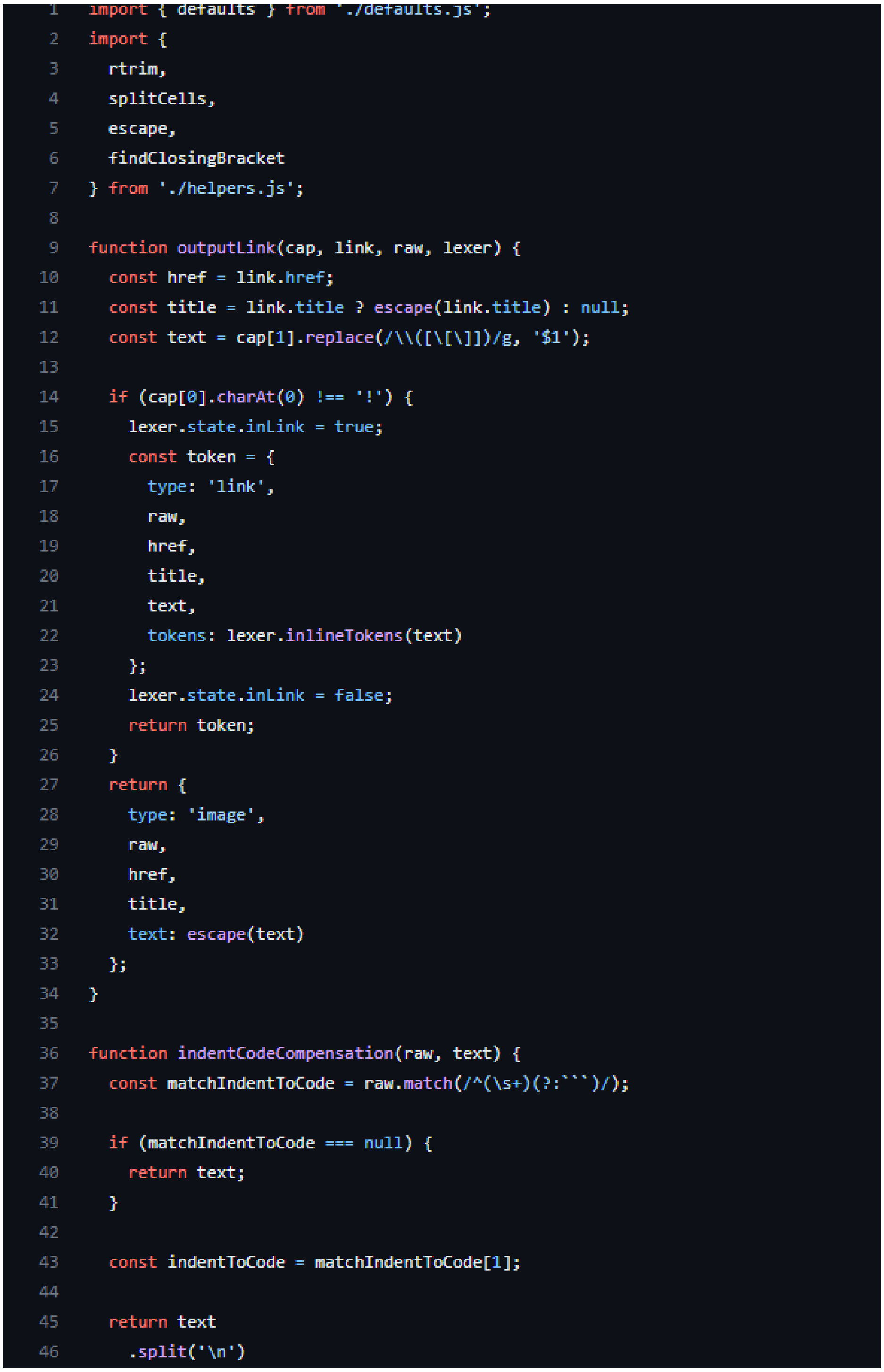


```
import { defaults } from './defaults.js';
import {
  cleanUrl,
  escape
} from './helpers.js';

/**
 * Renderer
 */
export class Renderer {
  constructor(options) {
    this.options = options || defaults;
  }

  code(code, infostring, escaped) {
    const lang = (infostring || '').match(/\S*/)[0];
    if (this.options.highlight) {
      const out = this.options.highlight(code, lang);
      if (out != null && out !== code) {
        escaped = true;
        code = out;
      }
    }

    code = code.replace(/\n$/, '') + '\n';

    if (!lang) {
      return '<pre><code>'
        + (escaped ? code : escape(code, true))
        + '</code></pre>\n';
    }

    return '<pre><code class="'
      + this.options.langPrefix
      + escape(lang)
      + '">'
      + (escaped ? code : escape(code, true))
      + '</code></pre>\n';
  }

  /**
   * @param {string} quote
   */
  blockquote(quote) {
    return `<blockquote>\n${quote}</blockquote>\n`;
  }
```



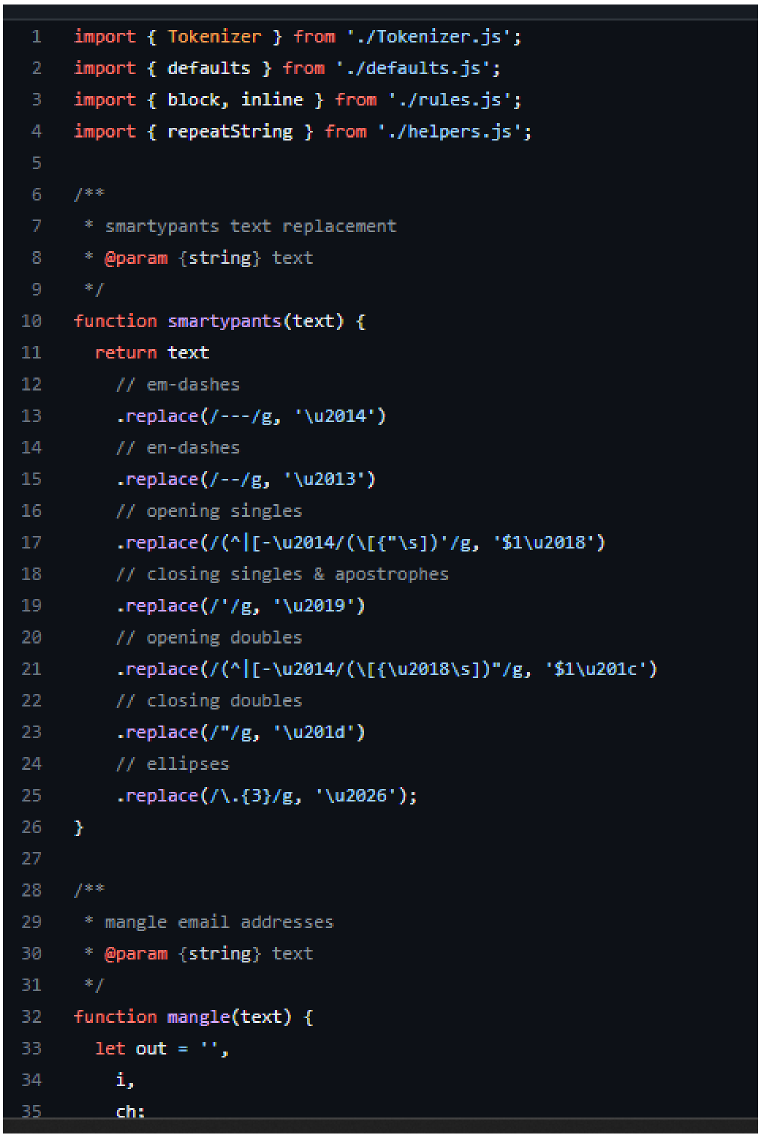
```
html(html) {
  return html;
}

/**
 * @param {string} text
 * @param {string} level
 * @param {string} raw
 * @param {any} slugger
 */
heading(text, level, raw, slugger) {
  if (this.options.headerIds) {
    const id = this.options.headerPrefix + slugger.slug(raw);
    return `<h${level} id="${id}">${text}</h${level}>\n`;
  }

  // ignore IDs
  return `<h${level}>${text}</h${level}>\n`;
}

hr() {
  return this.options.xhtml ? '<hr/>\n' : '<hr>\n';
}

list(body, ordered, start) {
  const type = ordered ? 'ol' : 'ul',
    startatt = (ordered && start !== 1) ? (' start="' + start + '"') : '';
  return '<' + type + startatt + '>\n' + body + '</' + type + '>\n';
}

/**
 * @param {string} text
 */
listitem(text) {
  return `<li>${text}</li>\n`;
}

checkbox(checked) {
  return '<input '
    + (checked ? 'checked="" ' : '')
    + 'disabled="" type="checkbox"'
    + (this.options.xhtml ? ' /' : '')
    + '> ';
}

```

6.3 Tokenizer

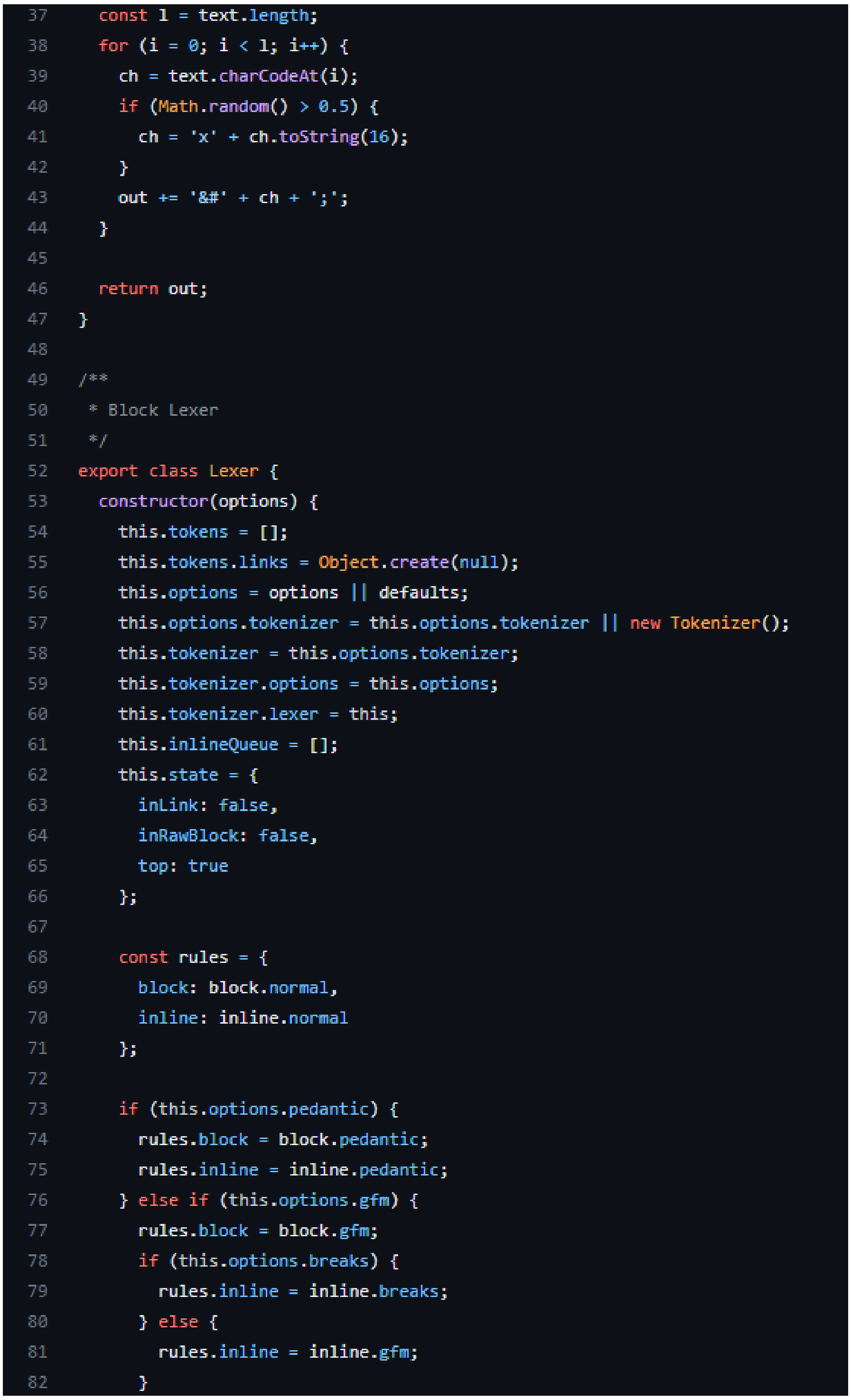


````
import { defaults } from './defaults.js';
import {
  rtrim,
  splitCells,
  escape,
  findClosingBracket
} from './helpers.js';

function outputLink(cap, link, raw, lexer) {
  const href = link.href;
  const title = link.title ? escape(link.title) : null;
  const text = cap[1].replace(/\\([\[\]])/g, '$1');

  if (cap[0].charAt(0) !== '!') {
    lexer.state.inLink = true;
    const token = {
      type: 'link',
      raw,
      href,
      title,
      text,
      tokens: lexer.inlineTokens(text)
    };
    lexer.state.inLink = false;
    return token;
  }
  return {
    type: 'image',
    raw,
    href,
    title,
    text: escape(text)
  };
}

function indentCodeCompensation(raw, text) {
  const matchIndentToCode = raw.match(/^(\s+)(?:```)/);

  if (matchIndentToCode === null) {
    return text;
  }

  const indentToCode = matchIndentToCode[1];

  return text
    .split('\n')
````

6.4 Lexer



```
import { Tokenizer } from './Tokenizer.js';
import { defaults } from './defaults.js';
import { block, inline } from './rules.js';
import { repeatString } from './helpers.js';

/**
 * smartypants text replacement
 * @param {string} text
 */
function smartypants(text) {
  return text
    // em-dashes
    .replace(/---/g, '\u2014')
    // en-dashes
    .replace(/--/g, '\u2013')
    // opening singles
    .replace(/(^|[-\u2014/(\[{"\s])'/g, '$1\u2018')
    // closing singles & apostrophes
    .replace(/'/g, '\u2019')
    // opening doubles
    .replace(/(^|[-\u2014/(\[{\u2018\s])"/g, '$1\u201c')
    // closing doubles
    .replace(/"/g, '\u201d')
    // ellipses
    .replace(/\.{3}/g, '\u2026');
}

/**
 * mangle email addresses
 * @param {string} text
 */
function mangle(text) {
  let out = '',
    i,
    ch;
```



```
  const l = text.length;
  for (i = 0; i < l; i++) {
    ch = text.charCodeAt(i);
    if (Math.random() > 0.5) {
      ch = 'x' + ch.toString(16);
    }
    out += '&#' + ch + ';';
  }

  return out;
}

/**
 * Block Lexer
 */
export class Lexer {
  constructor(options) {
    this.tokens = [];
    this.tokens.links = Object.create(null);
    this.options = options || defaults;
    this.options.tokenizer = this.options.tokenizer || new Tokenizer();
    this.tokenizer = this.options.tokenizer;
    this.tokenizer.options = this.options;
    this.tokenizer.lexer = this;
    this.inlineQueue = [];
    this.state = {
      inLink: false,
      inRawBlock: false,
      top: true
    };

    const rules = {
      block: block.normal,
      inline: inline.normal
    };

    if (this.options.pedantic) {
      rules.block = block.pedantic;
      rules.inline = inline.pedantic;
    } else if (this.options.gfm) {
      rules.block = block.gfm;
      if (this.options.breaks) {
        rules.inline = inline.breaks;
      } else {
        rules.inline = inline.gfm;
      }
```

CHAPTER 7

CONCLUSION

The Python programming language is extensively used by organisations and developers worldwide, and the CPython interpreter is the standard implementation of the language. Because of its versatile, extendable, and platform-independent architecture, it is the best option for creating Python applications.

The main parts of the CPython interpreter are the parser, compiler, virtual machine, standard library, and extension methods. The compiler converts the ASTs that the parser produces from reading Python source code into bytecode. The virtual machine, which controls memory and offers a stack-based execution mechanism, subsequently executes the bytecode. The extension techniques enable developers to construct unique extensions and interface with other languages, while the standard library offers a robust set of functionality that can be utilised in Python programmes.

Although the CPython interpreter has several downsides and restrictions, such as the Global Interpreter Lock (GIL), which can reduce the parallelism of multithreaded Python programmes, it is nonetheless a well-liked and often used tool for Python development. It is a dependable and useful tool for the Python environment because of its open-source design and vibrant developer community, which ensure its ongoing growth and improvement.

CHAPTER 8

FUTURE ENHANCEMENT

CPython is anticipated to advance and change in the future. Future developments could bring forth some of the following improvements:

1. Better concurrency support: Future editions of CPython are anticipated to provide improved concurrency support. As a result, software developers will be able to create systems that are more productive, scalable, and can utilise numerous processors and cores.

2. Quicker startup times: Although CPython has already made substantial strides in this regard, more may be done. The goal of CPython's upcoming versions could be to speed up startup times even more.

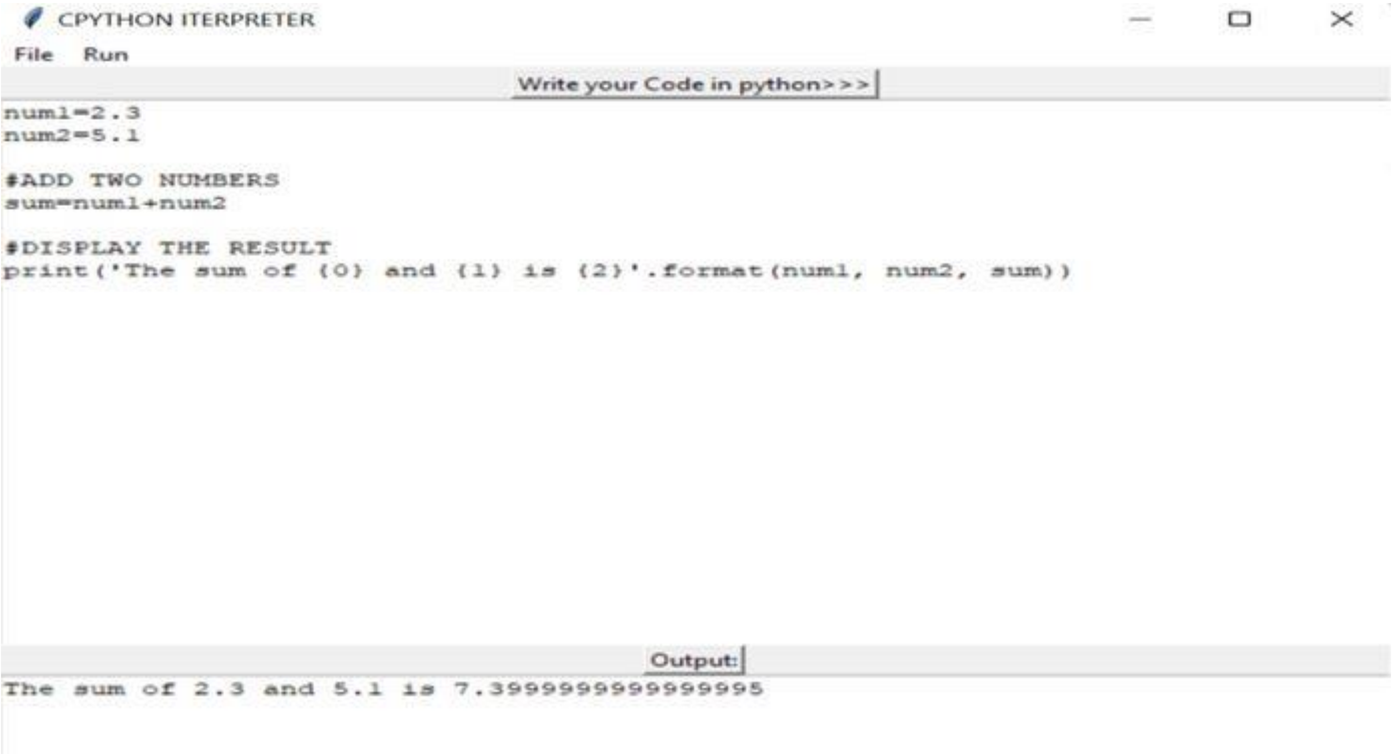
3. Improved error management: CPython may include improved techniques for handling errors that give developers more detailed error messages. Developers will be able to debug their code and solve problems more quickly as a result.

4. Better memory management: Python programmes could become more effective and scalable by using better memory management approaches that minimise their memory footprint.

Overall, CPython has advanced significantly since its introduction and is still a useful tool for developers. With its rich set of libraries and frameworks, performance improvements, and enhanced security features, CPython is expected to remain a popular choice for developing software applications in the years to come.

CHAPTER 9

OUTPUT SCREENSHOT



```
num1=2.3
num2=5.1

#ADD TWO NUMBERS
sum=num1+num2

#DISPLAY THE RESULT
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

```
The sum of 2.3 and 5.1 is 7.3999999999999995
```

CHAPTER 10

REFERENCES

1. "CPython Internals: A Beginner's Guide" by Anthony Shaw, which provides a comprehensive overview of CPython's architecture, data structures, and memory management.

2. "A Study of CPython's Memory Management" by Andrew Dalke, which investigatesCPython's memory allocation and garbage collection mechanisms.

3. "Improving CPython Performance" by Mark Shannon and Victor Stinner, which describes various techniques for improving the performance of CPython, including optimizations to the bytecode execution engine and improved memory management.

4. "Python and Rust: a match made in heaven?" by Armin Ronacher, which explores the use of Rust for writing Python extensions, and its potential benefits in terms of performance and reliability.

5. "CPython, PyPy, and type annotations: A performance comparison" by Sebastian Witowski and Mateusz Haligowski, which compares the performance of CPython and PyPy with and without type annotations.

These papers provide a deeper understanding of the inner workings of CPython and exploreways to improve its performance and reliability.