Program Analysis: Towards the Analysis of CPython Bytecode

André Theuma

Supervisor: Dr. Neville Grech

Faculty of ICT

University of Malta

22/04/2022

Submitted in partial fulfillment of the requirements for the degree of B.Sc. I.C.T. (Hons.)

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

Program Analysis: Towards the Analysis of CPython Bytecode

submitted is my work, except where acknowledged and referenced.

André Theuma

31/05/2022

Acknowledgements

your acknowledgements

Abstract

Analysis tooling systems are automated tools used to generate observations regarding useful characteristics of a program. These tooling systems provide insight in a program, showing the developer the inner workings of the individual operations which take place during program execution. These systems simulate different paths a software system can take, helping prevent vulnerabilities which may not appear during run-time.

PATH (Python Analysis Tooling Helper) is a static analysis tool created in this project, which generates a standardized Intermediary Representation for any given function, allowing analysis metrics to be generated from the facts produced by the tool. The purpose of this project was to create a framework that successfully conducts simple, accurate analysis of functions which do not require the complexity that there currently is in the available frameworks. PATH would disassemble CPython bytecode into a more 'digestible' representation, making any further possible analyses that would take place on the function possible.

The technologies that were used to make this project possible were Python V3.10, alongside the CPython interpreter.

Contents

List of Figures

List of Tables

1. Introduction

As an increasing amount of people are becoming reliant on complex software systems, the lack of security and analysis tooling frameworks is unacceptable in this day and age. Analysis tooling frameworks provide vital insight in software systems; allowing for further development and refinement of the said system. With an analysis tooling framework core vulnerabilities may be exposed and optimizations can be implemented. Analysis tooling frameworks already exist in the industry (such as Pylint [18], and Bandit [8]), but produce analyses which are highly specific and catered for certain use-cases.

PATH (Python Analysis Tooling Helper) provides general metrics for functions which are passed through it, aiding in analysis by generating a standardized IR (Intermediary Representation) and a Control Flow Graph.

1.1 Preliminary Overview

1.1.1 Python & CPython

Python is a high-level, object-oriented scripting language [19], suited for a wide range of development domains; from text processing [5] to machine learning [13] to game development [30]. The language's wide adoption (TIOBE's Language of the Year: 2007, 2010, 2018, 2020, 2021; [14]) may be attributed to the fact that it is based on the English language [26], making it easy to learn; aiding in the production of relatively complex programs. It is used extensively for rapid prototyping, as well developing fully-fledged real-world applications.

The most common implementation of Python is known as CPython [33], which is a bytecode interpreter for Python; written in C [1]

1.1.2 Program Analysis

Complex programs come hand-in hand with complex behaviours. These behaviours may need to be analysed, as they might highlight certain vulnerabilities, and also possibly indicate where optimizations can be carried out in the program. This area of interest is known as Program Analysis. Program Analysis provides answers for the following questions;

- Can the provided code be optimized?
- Is the provided code free of errors?
- How does data flow through the program & in what order do instructions get executed (Control-Flow)?

Naturally, as an increasing amount of modern day systems and frameworks are developed in Python, the need for properly conducting program analysis on these systems is ever-growing. There are two main approaches regarding program analysis; Dynamic Program Analysis & Static Program Analysis. Dynamic analysis is the testing and evaluation of an application during runtime, whilst static analysis is the testing and evaluation of an application by examining the code, producing facts and deducing possible errors in the program from the facts produced; without code execution [15]. Since all (significant) properties of the behaviour of programs written in today's' programming languages are mathematically undecidable [24], one must involve approximation for an accurate analysis of programs. This kind of analysis cannot be carried out by a Dynamic analysis as carrying out a runtime analysis only reveals errors but does not show the absence of errors [20]; which is the main motivation behind Static analysis. With the right kind of approximations,

[1] It is assumed that whenever a Python reference is made in this paper, Python V3.10 with the CPython implementation is in question.

Static analysis provides guarantees regarding properties of all the possible execution paths a program can take, giving it a clear advantage over Dynamic analysis; thus will be the main topic of interest in this paper.

1.2 Aims & Objectives

The essential aim of this dissertation is the production of an analytical tool (PATH) which can be easily implemented in existing software systems. The tool is to be used on Python V3.10 [33] systems which are interpreted with the CPython [33] interpreter. The tool needs to be able to scale up to larger systems, and still provide accurate metrics. The analytical tool must also generate a standardized IR of the functions present in the system being analysed. The standardized IR that is generated has to summarize the block analysis that is done by PATH.

The creation of this tool is vital as there is a lack of security and analysis tooling for what is the worlds most used programming language [14]; giving developers an increased freedom of choice in regard to choosing an analysis framework for their projects.

1.3 Chapters Overview

This paper is composed of five chapters, which can be broken down in the following manner:

Chapter 1 contains the introductory content, along with a brief overview of the technologies and ideas that are going to be delved into further along the paper. The introduction also gives a run-down of the main objectives that are to be met in this project.

Chapter 2 gives an in-depth literature review of the content that is briefly touched upon in the introductory chapter [Chapter 1]. The literature review consists of (but is not limited to) the following works; Python, the uses of Python, Python's bytecode & CPython's *ceval.c* interpreter, and Static Analysis Tooling.

Chapter 3 delves into the methodology and implementation of the disassembler

(PATH). The design of choice is further discussed together with the reasons for ongoing into such a design are highlighted.

Chapter 4 evaluates the results produced by PATH and answers specific research questions, ensuring that the Analytical toolkit works as intended. A couple of case-studies are included testing the scalability and ease-of use of PATH.

Chapter 5 presents the conclusions of the project and any suggestions for further work that can be done.

2. Background & Literature Review

2.1 Python

2.1.1 History

Python being a general-purpose [28], high-level programming language [37, pp.2–4], makes producing complex software systems a relatively non-trivial task when compared to the production complexity that comes along with other comparable programming languages such as C [29].

The language was developed to be a successor of the ABC programming language [12] and was initially released as Python 0.9.0 in 1991. Similarly to ABC, Python has an English-esque syntax but differs in its application domain; Python being a tool that is intended to be used in a more professional environment, whilst ABC is geared towards non-expert computer users [35, pp.285–288].

2.1.2 Features & Philosophy

The simplicity of Python enables it to be an extremely popular language to use in the industry of software development [14]. It was designed with the intention to be highly readable, thus removing the 'boilerplate' style used in the more traditional languages, such as Pascal. Python uses indentation for block delimiters, (off-side rule [33, pp.4–5]) which is unusual among other popular programming languages

[33, pp.2–3]; new blocks increase in indentation, whilst the end of a current block is signified by the decrease of an indentation. It supports a dynamic type system, enabling the compiler to run faster and the CPython interpreter to dynamically load new code. Dynamic type systems such as Python offer more flexibility; allowing for simpler language syntax, which in turn leads to a smaller source code size [31]. Although dynamically typed, Python is also strongly-typed disallowing operations which are not-well defined. Objects are typed albeit variable names are untyped.

One of Python's most attractive features is that it offers the freedom to allow the developer to use multiple programming paradigms [38]; appealing to a wider audience. Python also includes a cycle-detecting garbage collector [38], freeing up objects as soon as they become unreachable [36, pp.9–10]. Objects are not explicitly freed up as the collector requires a significant processing overhead [39, pp.27-30], and re-allocating memory to objects every time an object is required is resource consuming. Python has module support in its design philosophy formulating a highly extensible language. Modules can be written in C/C++ [28] and imported as libraries in any Python project; highlighting the extensibility of the language[1]. There are plenty [2] of readily available third-party libraries suited for many tasks, ranging from Web Development [9] to complex Machine Learning frameworks [22], further increasing ease-of use and supporting the quick and simple development philosophy of Python.

2.1.3 Implementations

There are several environments which support the Python language; known as implementations. The default, most feature comprehensive Python implementation is CPython[3] [34] ,written and maintained by Guido van Rossum. Other popular re-implementations include PyPy [4], Jython [16] and IronPython [21]. This paper

[1]The language the module is being written in depends on the interpreter that is being used. In this paper it is assumed that the CPython interpreter is being used as it is part of Python's default implementation

[2]Over 329,000 packages as of September 2021 [38]

[3]The terms CPython and Python are typically used interchangeably. This will be the case in this paper, unless specified otherwise.

will focus on the CPython implementation of the Python language and will not cover any of the other alternate implementations.

2.2 CPython

2.2.1 Overview

CPython is one of the many environments that support the Python language. It [4] is written in C, and [5] compiles Python source code into bytecode (See **Bytecode**). There are four main compilation stages of the CPython compiler [23]; the grammar parsing, compilation of the parse tree to the bytecode, bytecode optimizations[6], and finally bytecode interpretation (as seen in Figure 2.2).

CPython works transparently, via the PVM(Python Virtual Machine); an interpreter loop is run [7] and there is no direct translation between the Python code to C [1, pp.1–2]. The PVM is known as a stack machine, whereby PVM instructions retrieve their arguments from the stack, just to be placed back onto the stack after the instruction is executed. It can be said that the python compiler generates PVM code[8] for the python VM to execute. The CPython interpreter resembles a classic interpreter with a straightforward algorithm [1, pp.2–4]:

1. Firstly, the opcode of a VM instruction is fetched, along with any necessary arguments.

2. Secondly, the instruction is then executed.

3. Finally, steps 1-2 are repeated till no more opcodes can be fetched. This is done by raising an exception when an invalid [9] opcode is found.

[4] The CPython implementation.
[5] At a very high level.
[6] These optimizations can be omitted, depending on the arguments given to the compiler.
[7] *ceval.c*
[8] PVM code is also known as a *.pyc* file.
[9] In the case that there are no more opcodes, this in itself is considered to be an invalid opcode

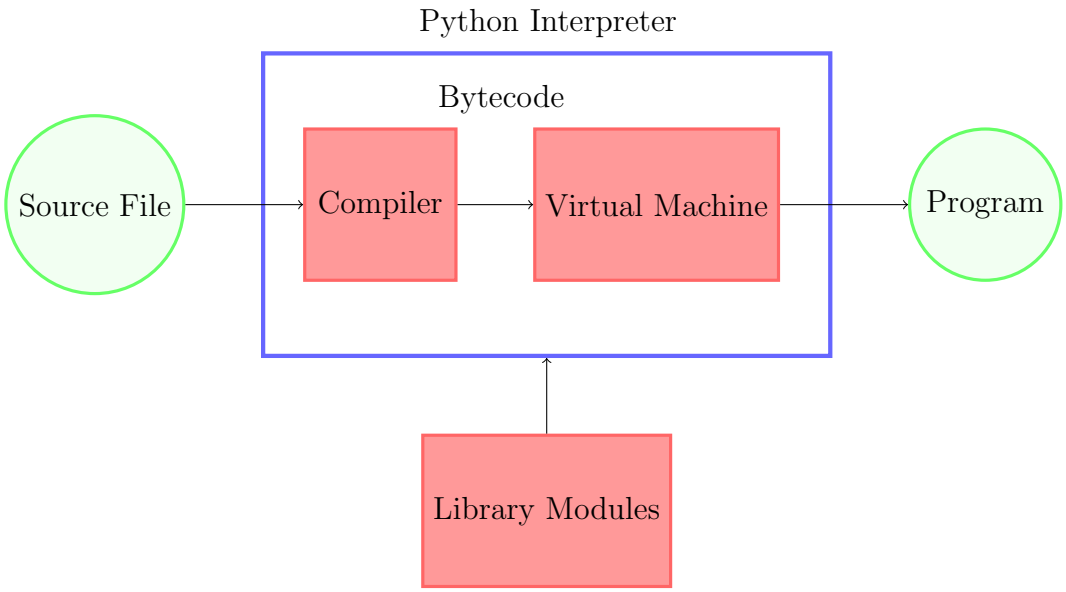

Figure 2.1: Python Code Execution

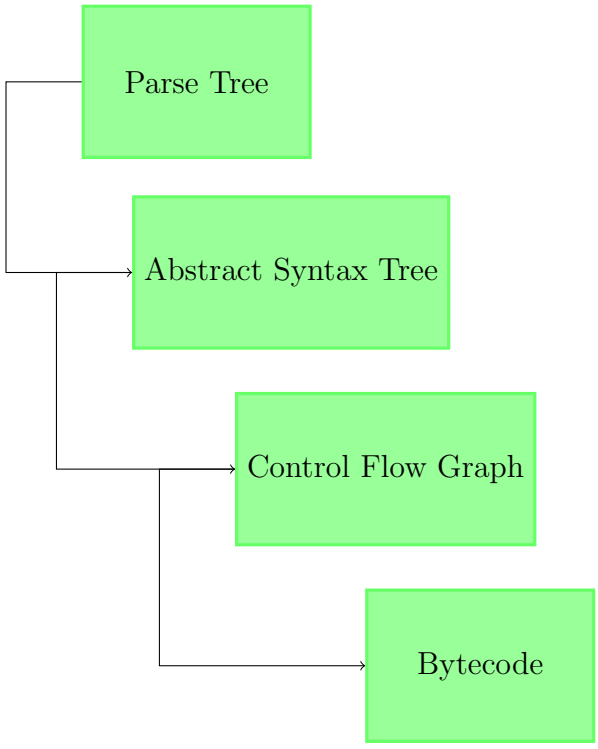

Figure 2.2: Compiler Innards

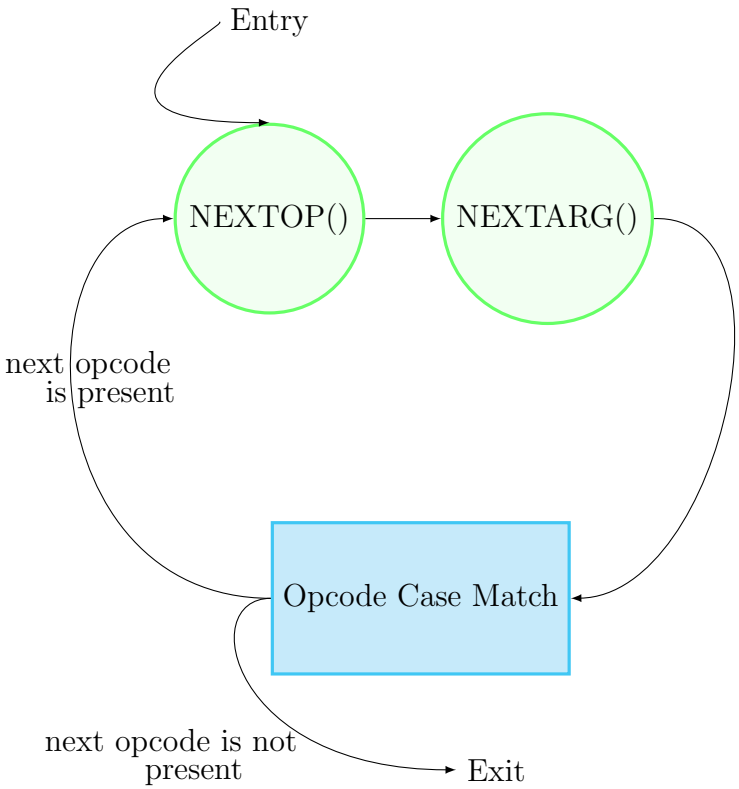

Figure 2.3: VM Innards

This simple algorithm is known as the CPython Evaluation Loop [7]. The evaluation loop [7] is formulated in the following manner[10]:

Evaluation is computed frame by frame (See **Frames**), with what is essentially a (very long) switch statement; reading every opcode and delegating accordingly.

2.2.2 Stacks

A stack data structure is a dynamic structure that operates with a LIFO[11] policy [7]. Since CPython does not directly interact with the hardware for compilation, this makes both its call stack and stack frames rely on the PVM. In CPython there is one main stack that the PVM requires for proper functionality, this being the call stack. The other two stacks (Value Stack and Call Stack) are essential for the proper computation of any variables that there are in the frame (See **Frames**).

[10] Actual code differs from what is presented. The source code has been edited as to be more readable and concise.

[11] Last in First out.

```
for(**indefinite condition**){
    oparg=null;
    opcode = NEXTOP();
    if(ARGUMENT_PRESENT(opcode)){
        oparg = NEXTARG();
    }
    switch(opcode){
        case **opcode_name**:
            manipulate stack & set variables accordingly
        ...
        ...
        ...
        default:
            raise error
    }
}
```

Figure 2.4: Evaluation Loop

Most instructions manipulate the value stack and the call stack [3].

Call Stack

The call stack contains call frames[13] (See **Frames**). This is the main structure of running a program. A function call results into a pushed frame onto the call stack whilst a return call results into a pop of the function frame off of the stack [6]. A visual representation of a call stack in action is shown in Figure 2.5. In this figure a sample script can be seen run, step by step, showing the frames being pushed onto the call stack and popped from the call stack as is required. Regardless of the code provided, the first frame pushed onto the frame stack is called the `__main__` call frame.

Value Stack

This stack is also known as the evaluation stack. It is where the manipulation of object happens when object-manipulating opcodes are evaluated[12] A value stack is found in a call-frame; with every call-frame having a value stack. Any manipulations that are performed on this stack (unless they are namespace related) are

[12]When the stack of the current frame is modified.

Python Test Script

```
def foo():
    print("Hello")

def intermediary():
    foo()

def start():
    intermediary()

start()
```

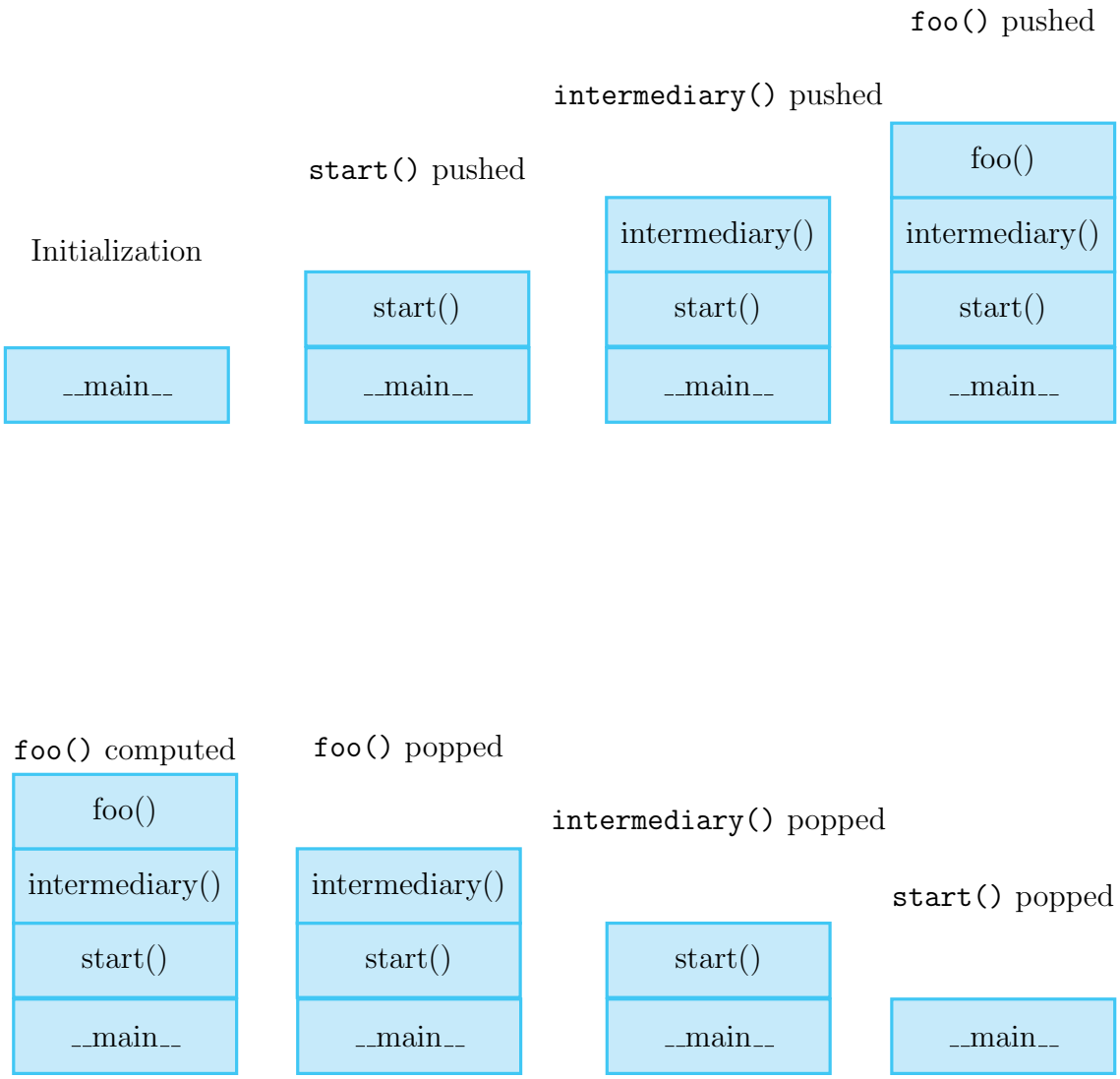

Figure 2.5: Call Stack

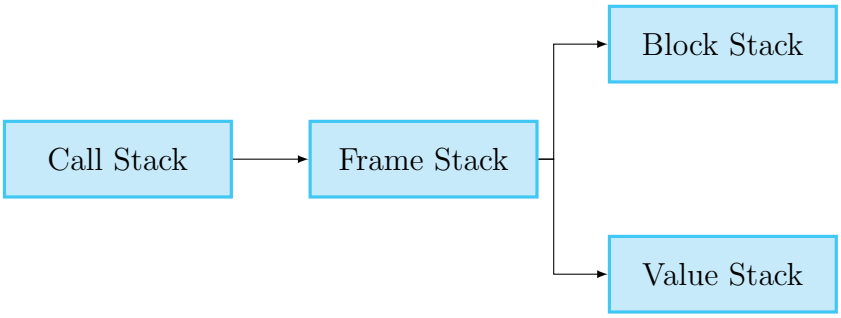

Figure 2.6: Overview of CPython stacks

independent of other stacks and do not have the permissions to push values on other value stacks.

Block Stack

The block stack keeps track of different types of control structures, such as; loops, try/except blocks and with blocks. These structures push entries onto the block stack, which are popped whenever the said structure is exited. The block stack allows the interpreter to keep track of active blocks at any moment

2.2.3 Frames

A frame[13] (in Pythonic terms), is an object which represents a current function call (subprogram call); more formally referred to as a code object. It is an internal type containing administrative information useful for debugging and is used internally by the interpreter [36, pp.18–19]. Frame objects are tightly coupled with the three main stacks (See **Stacks**) and every frame is linked with another frame. Every frame object has two frame-specific stacks; value stack (See **Value Stack**) and the block stack (See **Block Stack**). Frames are born from function calls, and die when that function is returned.

[13] Also called a call-frame.

Frame Attributes

Along with the properties mentioned above, a frame object would have the following attributes [14]:

- `f_back`: this is[15] the previous stack frame object (return address).
- `f_code`: this is[15] the current code object (See **Code Objects**) being executed, in the current frame.
- `f_builtin`: this is[15] the builtin symbol table.
- `f_globals`: this is[15] the dictionary used to look up global variables.
- `f_locals`: this is[15] the symbol table used to look up local variables.
- `f_valuestack`: this is a pointer, which points to the address after the last local variable.
- `f_valuestack`: this holds[15] the value of the top of the value stack.
- `f_lineno`: this gives the line number of the frame.
- `f_lasti`: this gives the bytecode instruction offset of the last instruction called.
- `f_blockstack`: contains the block state, and block relations.
- `f_localsplus`: is a dynamic structure that holds any values in the value stack, for evaluation purposes.

Frame Stack

An important data structure relating to the concept of Frames is the stack frame. The stack frame is a collection of all the current frames in a call-stack-like data

[14] Retrieved from declaration found in *./Include/frameobject.h*

[15] In this context; "...this is..." or "...this holds..." does not mean the actual value is being stored in the variable, but a pointer to the address of the value is being stored.

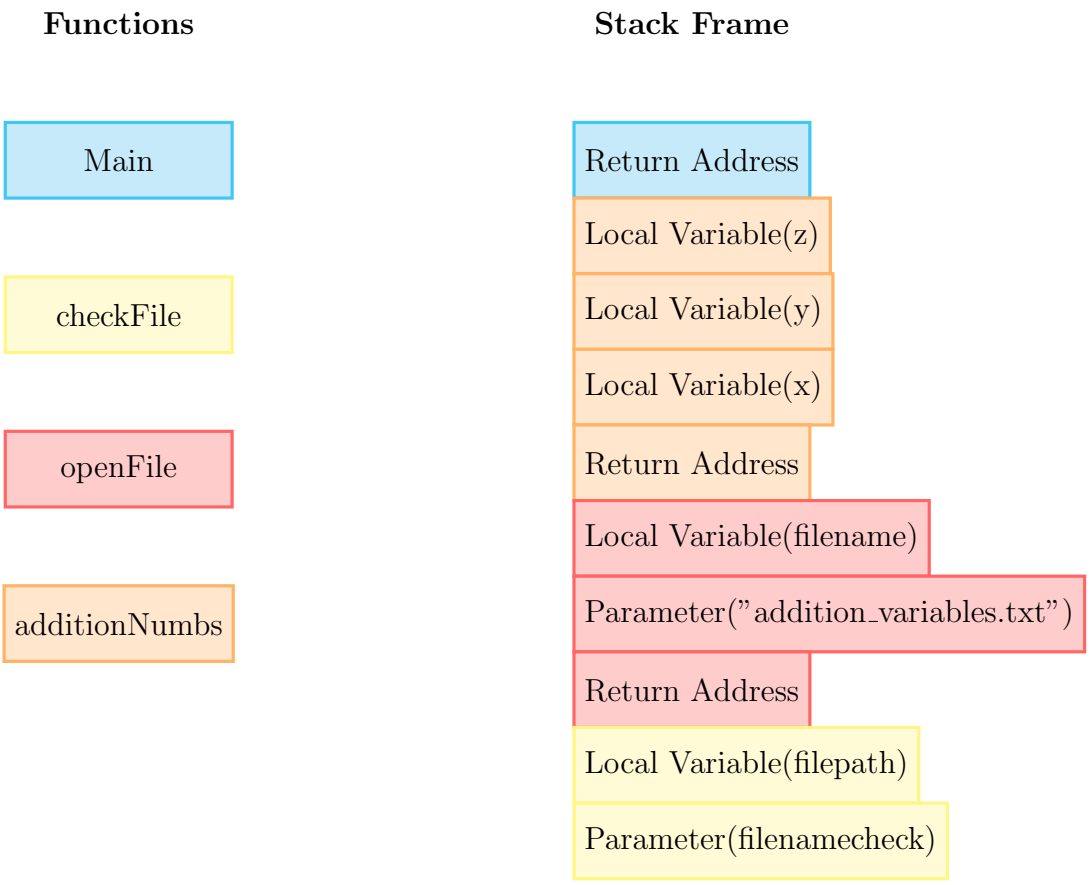

Figure 2.7: Stack frames example

structure (See **Call Stack**). A stack frame exists for every function call (as typically, every function has its own unique frame) and are stacked as shown in Figure 2.7.

The stack frame contains a frame pointer which is another register that is set to the current stack frame. Frame pointers resolve the issue that is created when operations (pushes or pops) are computed on the stack hence changing the stack pointer, invalidating any hard-coded offset addresses that are computed statically, prior to run-time [2]. The code can then refer to the local variables as offsets from the frame pointer not the stack pointer.

2.2.4 Code Objects

A code object is a low-level detail of the CPython implementation. When Python code is parsed, it is compiled into a code object to be run on the PVM. A code

object contains a list of instructions which directly interact with the CPython VM; hence why it is called a low-level detail. Typically, code objects are of the type `PyCodeObject`[16] and each section of the code object represents a chunk of executable code that has not been bound into a function [10]. The structure of the type of these code objects [16] change throughout different CPython versions, thus there is no set composition [10]. For reference, the source code in Figure 2.8 produces the following code object:

```
<code object addition_numbers at 0x1047f10b0, file "filepath", line 3>
```

Which follows this convention:

$$codeobject < functionName > at < address >, file < path >, line < firstLineNo. > \tag{2.1}$$

Disassembler

Code objects are expanded by using the `dis` module in Python. This module contains several analysis functions which; all of which directly convert the input code object into the desired output [11]. The function that is of a particular interest in this paper is the `dis.dis` function which disassembles a code object into its respective bytecodes, alongside some other information, as seen in Figure 2.8. When applying the analysis function `dis.dis`, the disassembled code object takes the following format:

$$< lineNumber >< label >< instructionOffset >< opname >< opargs >< var > \tag{2.2}$$

It is interesting to note that the value for `opargs` is computed in little-endian order. Typically, as is shown in ... ,the arguments associated with the instructions are used for specific stack manipulations.

[16]The type is the C structure of the object which is used to describe a code object.

Python source code

```
1        import dis
2
3        def addition_numbers(x,y):
4            z=x+y
5            return z
6
7        dis.dis(addition_numbers)
```

Disassembly of Function

```
4         0 LOAD_FAST        0(x)
          2 LOAD_FAST        1(y)
          4 BINARY_ADD
          6 STORE_FAST       2(z)

5         8 LOAD_FAST        2(z)
         10 RETURN_VALUE
```

Figure 2.8: Disassembly of a Function

Bytecode

Bytecode is a form of portable code, that is executed on a virtual machine. The origin of the concept of bytecode was introduced when a generalized way of interpreting complex programming languages was required, so as to simplify information structures that can, instead, be characterized in their essentials [17]. This ideology gave birth to portable code[17], which was a generalized form of code, that can cross-compile, as long as the Virtual Machine that interpreted it was compatible with the native machine architecture. In today's CPython[1], there are over 100 low-level bytecode instructions [11]. These bytecode operations can be classified in the following manner;

- Unary instructions.
- Binary instructions.
- Inplace instructions.
- Coroutine instructions.

[17]Portable code, p-code and bytecode can be used interchangeably.

- Argument instructions.
- Miscellaneous instructions.

All these instructions are discussed with much further detail in the ... section

2.2.5 Execution of Code Objects

The evaluation stage firstly makes use of the public API `PyEval_EvalCode()` [25, lines 716–724] which is used for evaluating a code object created at the end of the compilation stages (Figure 2.2). This API constructs an execution frame from the top of the stack by calling `_PyEval_EvalCodeWithName()`. The first execution frame that is constructed must conform to these requirements:

- The resolution of keyword[18] and positional arguments[19].
- The resolution *args[20] and **kwargs[20] in function definitions.
- The addition of arguments as local variables to the scope[21].
- The creation of Co-routines and Generators [32, pp.2–3].

Code execution in CPython is the evaluation and interpretation of code object. Below we delve into a more detailed description of how frame objects are created, and executed [27].

Thread State Construction

Prior to execution, the frame would need to be referenced from a thread. The interpreter allows for many threads to run at any given moment. The thread structure that is created is called `PyThreadState`.

[18] A keyword argument is a value that, when passed into a function, is identifiable by a specific parameter name, such as variable assignment.

[19] A positional argument is a value that is passed into a function based on the order in which the parameters were listed during the function definition.

[20] *args and **kwargs allow multiple arguments to be passed into a function via the unpacking (*) operator.

[21] A scope is the membership of a variable to a region

Frame Construction

Upon constructing the frame, the following arguments are required:

- `_co`: A `PyCodeObject` (code object).
- `globals`: A dictionary relating global variable names with their values.
- `locals`: A dictionary relating local variable names with their values.

It is important to note that there are other arguments that might be used but do not form part of the basic API, thus will not be included.

Keyword & Positional Argument Handling

If the function definition contains a multi-argument keyword argument [20], then a new keyword argument dictionary [22] must be created in the form of a `PyDictObject` Similarly, if any positional arguments[20] are found they are set as local variables.

The dictionary[22] that was created is now filled with the remaining keyword arguments which do not resolve themselves to positional arguments. This resolution comes after all the other arguments have been unpacked. In addition, missing positional arguments [23] are added to the `*args`[20] tuple. The same process is followed for the keyword arguments; values are added to the `**kwargs` dictionary and not a tuple.

Final Stage

Any closure names are added to the code object's list of free variable names, and finally, generators and coroutines are handled in a new frame. In this case, the frame is not pre-evaluated but it is evaluated only when the generator/coroutine method is called to execute its target.

[22]Denoted as a `kwdict` dictionary.

[23]Positional arguments that are provided to a function call, but are not in the list of positional arguments.

2.2.6 Execution of Frame Objects

The local and global variables are added to the frame preceding frame evaluation, which is handled by `_PyEval_EvalFrameDefault()`.

`_PyEval_EvalFrameDefault()` is the central function which is found in the main execution loop. Anything that is executed in CPython goes through this function and forms a vital part of interpretation.

2.3 Analysis

3. Methodology

4. Evaluation

5. Conclusion

A. This chapter is in the appendix

A.1 These are some details

References

[1] J. Aycock. Converting python virtual machine code to c. In *Proceedings of the 7th International Python Conference*, pages 76–78, 1998.

[2] J. W. Bacon. The stack frame, 2011. URL: `http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch10s07.html` [cited on: 25-04-2022].

[3] J. Bennett. An introduction to python bytecode, 2018. URL: `https://opensource.com/article/18/4/introduction-python-bytecode` [cited on: 26-04-2022].

[4] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

[5] V. Bonta and N. K. N. Janardhan. A comprehensive study on lexicon based approaches for sentiment analysis. *Asian Journal of Computer Science and Technology*, 8(S2):1–6, 2019.

[6] P. Conrad. Python function calls and the stack lesson 2, 2010. URL: `https://sites.cs.ucsb.edu/~pconrad/cs8/topics.beta/theStack/02/` [cited on: 26-04-2022].

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition edition, 2009.

[8] B. Developers. Bandit, 2022. URL: `https://bandit.readthedocs.io/en/latest/` [cited on: 20-04-2022].

[9] J. Forcier, P. Bissex, and W. J. Chun. *Python web development with Django*. Addison-Wesley Professional, 2008.

[10] P. S. Foundation. Code objects, 2022. URL: `https://docs.python.org/3/c-api/code.html` [cited on: 26-04-2022].

[11] P. S. Foundation. dis-disassembler for python bytecode, 2022. URL: `https://docs.python.org/3/library/dis.html#dis.Bytecode` [cited on: 26-04-2022].

[12] L. Geurts, L. Meertens, and S. Pemberton. *ABC programmer's handbook*. Prentice-Hall, Inc., 1990.

[13] P. Goldsborough. A tour of tensorflow. *arXiv preprint arXiv:1610.01178*, 2016.

[14] T. Index. The python programming language, 2022. URL: `https://www.tiobe.com/tiobe-index/python/` [cited on: 19-04-2022].

[15] Intel. Dynamic analysis vs. static analysis, 2013. URL: `https://www.cism.ucl.ac.be/Services/Formations/ICS/ics_2013.0.028/inspector_xe/documentation/en/help/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm` [cited on: 19-04-2022].

[16] J. Juneau, J. Baker, F. Wierzbicki, L. S. Muoz, V. Ng, A. Ng, and D. L. Baker. *The definitive guide to Jython: Python for the Java platform*. Apress, 2010.

[17] P. J. Landin. The mechanical evaluation of expressions. *The computer journal*, 6(4):308–320, 1964.

[18] P. Logilab. Pylint, 2022. URL: `https://pylint.pycqa.org/en/latest/` [cited on: 20-04-2022].

[19] M. Lutz. *Programming python*. " O'Reilly Media, Inc.", 2001.

[20] A. Møller and M. I. Schwartzbach. Static program analysis. *Notes. Feb*, 2012. URL: `https://cs.au.dk/~amoeller/spa/spa.pdf`.

[21] J. P. Mueller. *Professional IronPython*. John Wiley & Sons, 2010.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[23] R. Ramachandra. An insight into cpython compiler design. 2009.

[24] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[25] G. V. Rossum et al. python/cpython, 2022. URL: `https://github.com/python/cpython/blob/d93605de7232da5e6a182fd1d5c220639e900159/Python/ceval.c#L716` [cited on: 24-04-2022].

[26] A. S. Saabith, M. Fareez, and T. Vinothraj. Python current trend applications-an overview. *International Journal of Advance Engineering and Research Development*, 6(10), 2019.

[27] A. Shaw. Your guide to the cpython source code, 2022. URL: `https://realpython.com/cpython-source-code-guide/` [cited on: 25-04-2022].

[28] K. Srinath. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, 4(12):354–357, 2017.

[29] M. Summerfield. *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming (paperback)*. Pearson Education, 2007.

[30] A. Sweigart. *Making Games with Python & Pygame*. 2012.

[31] D. Thomas. Benefits of dynamic typing. Online, 2013. URL: `https://wiki.c2.com/?BenefitsOfDynamicTyping` [cited on: 21-04-2022].

[32] C. Tismer. Continuations and stackless python. In *Proceedings of the 8th international python conference*, volume 1, 2000.

[33] G. van Rossum. Python (programming language) 1 cpython 13 python software foundation 15.

[34] G. Van Rossum. Python reference manual. *Department of Computer Science [CS]*, (R 9525), 1995.

[35] G. van Rossum and J. de Boer. Interactively testing remote servers using the python programming language. *CWi Quarterly*, 4(4):283–303, 1991.

[36] G. Van Rossum and F. L. Drake Jr. Python reference manual, 1994.

[37] G. Van Rossum and F. L. Drake Jr. *Python tutorial*, volume 620. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 2020.

[38] G. Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, pages 1–36, 2007. URL: `https://www.wikizero.com/en/Python_(language)`.

[39] B. Zorn. *Barrier methods for garbage collection*. Citeseer, 1990.

[19] [26] [5] [13] [30] [14] [15] [24] [20] [33] [37] [18] [28] [29] [12] [35] [31] [36] [39] [22] [9] [34] [4] [16] [21] [23] [1] [32] [2] [7] [6] [3] [11] [10] [17]