IMPERIAL COLLEGE LONDON
DEPARTMENT OF COMPUTING

SPLAT

Simple Python Lazy Automated Tester

Final Year Individual Project

MEng Project Report

Lee Wei Yeong

lwy08@doc.ic.ac.uk

Supervisor: Prof. Susan Eisenbach

Second Marker: Dr. Tristan Allwood

June 2012

Abstract

Writing unit test suites has become an indispensable part of the software engineering process. These test cases, collectively, aim to provide confidence to the client in the final delivery of the shipped product, as well as reduce costs by detecting and fixing bugs in the earlier stages of software development.

Unfortunately, it can be rather tedious to write comprehensive unit tests by hand, especially if the work contributing to this cost is hidden from the customer. Therefore, this project attempts to automate this software testing process, in order to make software development more efficient overall.

Traditionally, the core of software development often revolves around programming in statically typed languages like C/C++, Objective-C or Java/Scala. However, in recent times, it is increasingly common to find their dynamic counterparts, such as Javascript, Python or Ruby, growing in popularity, and employed in any software stack, not only just for quick scripting tasks.

This then necessitates the exploration of automated testing tools for dynamic languages, which is the motivation for this project, that is, to automatically generate unit tests using Python, and also address the present inadequate availability of such tools.

In addition, several techniques for accomplishing this have been suggested in papers, although many of which, including *lazy instantiation*, were applied extensively to static languages, like Haskell. Hence, this project intends to research into how a hybrid of these existing ideas could possibly inspire and be adapted to solve the automated unit test generation problem for this domain, in such a way that consistently high code coverage can be achieved across a variety of test programs.

In this paper, these techniques are implemented in the context of the dynamically typed programming language Python, and the experimental results of its performance for some programs is presented. These results show that this method is feasible and practical to some extent.

Acknowledgements

First of all, I would like to thank Professor Susan Eisenbach for supervising my project, and giving me invaluable advice and guidance throughout my work.

Secondly, I thank Dr. Tristan Allwood for his support, ideas, encouragement, and the useful discussions that we had regarding the direction of my project.

Also, many thanks to Chong-U Lim for his insight during our conversations, as well as to Yangfan Zhang, a close friend, without whom this project would not be what it is.

Finally, I would like to thank my family for their continued full support during the course of my university studies.

Contents

Chapter 1

Introduction

This is the Report for my Final Year Project, *Simple Python Lazy Automated Tester*, hence the acronym 'SPLAT' appearing on the cover page, which represents the name of the tool created from this research.

1.1 Motivation

Professional software engineers often write tests while developing code, especially for large complex codebases. These tests are highly beneficial for generating confidence in a bug-free solution delivery.

However, writing tests is not always easy to get right, and can be quite costly. It is reported that testing code is responsible for *approximately half* the total cost of software development [**?**][**?**][**?**].

Furthermore, this task becomes gradually more time-consuming as software grows in terms of complexity. Given similar resource constraints, it can become increasingly difficult to consistently achieve high test code coverage.

Moreover, a significant proportion of overall development time is spent writing test code not eventually included in production. Hence this work, though critical to assuring the quality of software [**?**], is ultimately invisible to the client, and sometimes difficult to justify this expenditure, as far as billing and accountability is concerned.

This has led to a large body of work on automatically generating unit test suites, particularly notable within the imperative programming community [**?**], in order to reduce the effort of unit testing required, to encourage wider adoption by developers.

Even then, the present need for manual testing indicates that there still remains much scope for improvement. A recent example supporting this claim is Google handing out a record $26k in bug bounties for security researchers reporting Chrome vulnerabilities [**?**].

Therefore, this example raises the question of whether full automatic discovery [**?**] for all these bugs could be possible, in order to eliminate the cost of manual testing, let alone any bug exploits.

1.2 Automated software testing for dynamic languages

Whilst research in this field is typically devoted to statically typed programming languages such as C/C++, Objective-C or Java/Scala, relatively less emphasis is placed on their dynamic counterparts like Javascript, Python or Ruby.

This observation is made in contrast to the rapid growth in popularity of dynamic languages in recent years, especially Python. Python was named the 'The Importance Of Being Earnest' (TIOBE) Programming Language of the Year, both in 2007 and 2010 [**?**]. Therefore, Python is the subject of this paper.

One paper implements the search-based software testing (SBST) technique, to automatically generate test scenarios for Ruby code, using genetic algorithms [**?**]. There is neither any equivalent tool targeting Python however, nor any chance of porting these tests for Python programs.

In that paper, the authors claimed success in achieving consistent and significantly high code coverage over a preselected set of test inputs with their tool, when compared against the naïve random test case generator. Would it be possible to improve this using a suitable adaptation of existing techniques, and/or to maintain this coverage across a more extensive range of programs written for different problem domains?

As Python, like Ruby, is a reflective, dynamically typed language, it would seem logical to adopt a similar approach in solving this problem, specifically by generating test scenarios via *runtime code analysis* [**?**].

1.3 The Python programming language

Growing numbers of commercial and closed source applications are being developed using the Python programming language [**?**]. Hence, this project would also be looking at automatically generating unit tests for Python bytecode, instead of from Python source code traditionally.

The Python programming language contains a variety of interesting features which encourage rapid experimentation with automatic testing techniques. This is primarily because Python is an open source, general purpose, multi-paradigm, cross-platform compatible, dynamically typed language, offering duck typing, and in active development and support. It also provides *excellent builtin introspection and reflection capabilities*, to inspect and manipulate code at runtime.

At the heart of the language design philosophy [**?**], there should be one – and preferably only one – obvious way to do things. The importance of readability promotes a *clean, concise and elegant syntax*, which makes demonstrating 'proof of concept' code easy.

For instance, the following Python code snippet certainly reads more fluently than its C# counterpart:

Sample C# code

```
if ("hello".indexOf("e") >= 0)
{
   return true;
}
```

Equivalent Python code

```
if 'e' in 'hello':
   return True
```

Python features a fundamental testing infrastructure toolset based on unittest, doctest and py.test. However, there is limited availability of testing support tools built on top of those. Many of these either target outdated versions of Python, or are discontinued. There are a few candidate tools for automated testing, for instance, pythoscope and pytestsgenerator, which generate tests by performing static code analysis. However, there are no tools which perform dynamic test case generation.

1.4 Project contributions

Within the context given above, this project makes the following key contributions:

- A discussion of the possible ways considered for automated software testing, focusing on test data generation by using information gathered at runtime
- A motivating example describing automated lazy testing in Python
- Implementation as a Python module, to automatically generate consistently high coverage test suites, primarily evaluated against Python libraries like python-graph, and other Python module implementations of famous algorithms
- Investigating effectiveness of the *lazy instantiation* testing technique, as illustrated by IRULAN in Haskell [**?**], for Python
- Further advance the work in the field of automated software testing, especially for dynamic languages

To this end, we take advantage of the main features of the core Python language, ie. *strong introspection and reflective capabilities*, combined together with its extensive tool support from the Python Package Index (PyPI) repository.

The concepts discussed in this paper are concretely demonstrated in a tool called SPLAT, a high coverage test suite generator for Python modules, written in Python, but portable to target other languages. This tool has been successfully applied to some of the most popular frameworks, achieving the initial objective of consistently high test code coverage, comparable to those manual unit tests written by hand, and even potentially discovering several bugs in the process as well. The tool has also been extended to support regression testing, where reports on a sample of case studies are included in this report.

1.5 Report organisation

Firstly, relevant background material is reviewed in Chapter **??**. Thereafter, the various algorithms and techniques used to automatically generate tests are formally introduced in Chapter **??**. These ideas presented are then implemented in the tool SPLAT, constituting the subject of Chapter **??**. This is accompanied by a detailed description of SPLAT's software design architecture, together with several worked examples, for clarification purposes. A summary of the extent of success of the project is discussed in Chapter **??**. Lastly, some final conclusions are drawn, before suggestions are given to possible future work in Chapter **??**.

Chapter 2

Background

2.1 Introduction

This part of the paper intends to provide an overview and discussion of literature relevant to this project, forming the basis for the reader to follow on later content. Firstly, Sections **??** to **??** review the general field of automated software testing. Section **??** deals with the papers that inspired and influenced this project. Relevant characteristics of dynamically typed programming languages, i.e. those related to Python, are then discussed in Section **??**. Finally, the associated technical difficulties are highlighted in Section **??**.

2.2 Definition of terms

Software testing delivers quality assurance in the product to the customer. It reveals faults by producing observable failures, and verifies that the provided implementation complies with the original client specification. The following terms commonly found in *automated test data generation* research are defined below.

2.2.1 General software testing

- *Test data*: data specifically identified for use in testing the software
- *Test case*: set of conditions under which the correct behaviour of an application is determined
- *Test suite*: a collection of test cases
- *Test automation*: use of software to control test execution, comparison of actual and expected results, setting up of test preconditions, and other test control and reporting functions
- *Test coverage*: measurement of extent to which software has been exercised by tests

2.2.2 Modelling programs as graphs

- *Input variable*: variable which appears as an input argument to the function being tested, usually one which is used in the function body
- *Program input*: cross product of the domains of the collection of input variables

- *Node*: an atomic, single entry, single exit, executable program bytecode instruction
- *Edge* $n_i \rightarrow n_j$: represents a possible transfer of execution control from node n_i to n_j
- *Control flow graph (CFG)*: a directed graph $G = (nodes, edges, start\_nodes, end\_nodes)$ for a program F
- *Path*: sequence of nodes and edges. If a path begins from the entry node, and terminates at the exit node, then it is a *complete* path.
- *Branch predicate*: condition in a node leading to either a true or false path
- *Path predicate*: collection of branch predicates which are required to be true, in order to traverse the path
- *Feasible path*: path with valid input for execution
- *Infeasible path*: path with no valid input for execution
- *Constraint*: an expression of conditions imposed on variables to satisfy
- *Definition (of a variable v)*: a node which modifies the value of v, for example, an assignment or input statement
- *Use (of a variable v)*: a node in which v is referenced, for example, in an assignment or output statement, or branch predicate expression
- *Definition-clear path (with respect to variable v)*: path within which v is not modified
- *Post domination*: a node z is *post-dominated* by a node y in G if and only if every path from y to the exit node e contains z
- *Control dependent*: a node z is *control dependent* on y if and only if z post-dominates one of the branches of y, and z does not post-dominate y
- *Control dependency graph (CDG)*: graph describing the reliance of a node's execution on the outcome at previous branching nodes

2.3 Overview

These survey papers [**?**] [**?**] [**?**] provide a high level overview of all the different kinds of general software testing techniques, as outlined in the following diagram:

The proposed test data generation techniques can be broadly classified into either Functional or Structural testing, or a combination of both. In this section, each approach is to be explained, and a single or two representative test data generation methods of each approach is to be described. This paper intends to focus on the *non-systematic, dynamic, structural* branch of software testing.

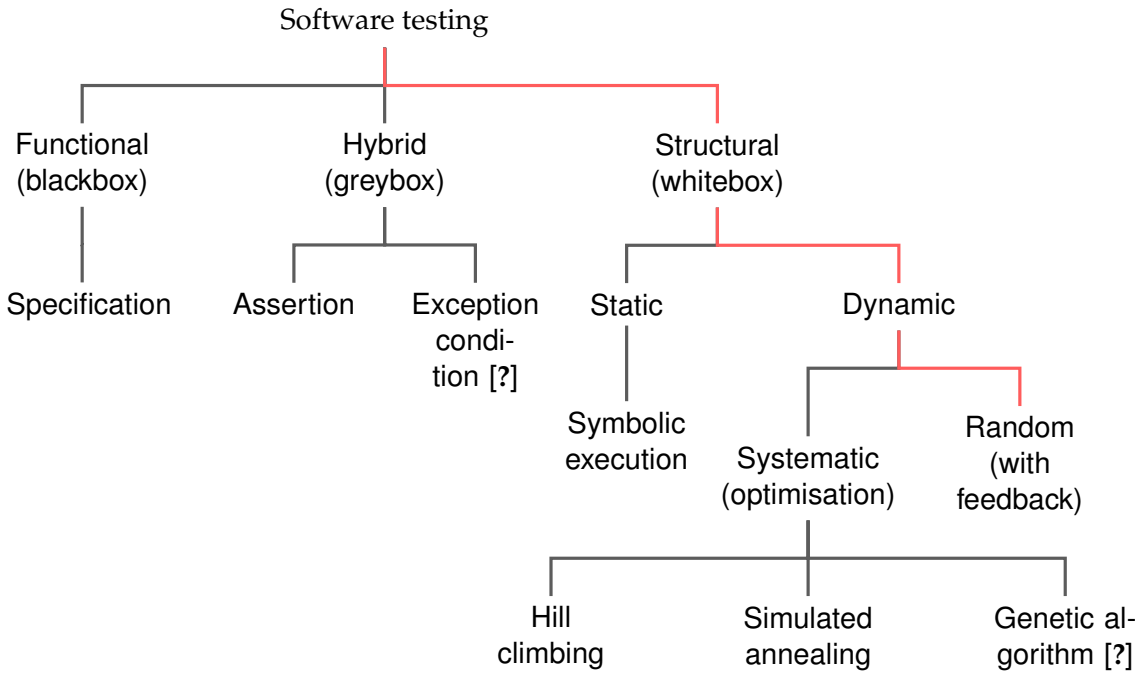


Figure 2.1: High level overview of software testing

2.3.1 Functional testing

Functional testing is one of the fundamental approaches to identifying suitable test cases. It is concerned with testing for logical system behaviour conforming to a prescribed specification. For tests derived this way, a present barrier to complete automation is the fact that a mapping needs to be provided from the abstract model of the specification to the concrete form of the implementation. A suggested solution for this is the use of innovative encodings of such information, but there has been little activity in this area of research.

2.3.2 Structural testing

Structural testing is the process of deriving tests from the internal structure of the SUT. Studies have proven that it has been successfully applied, although a majority of them are limited to simple input types, such as numerical values, since numbers are common as input data in real-world software. Structural testing can be further subdivided into two classes: Static and Dynamic approaches.

2.3.2.1 Static approach

The Static approach does not execute the SUT, but rather generates test data by gathering static program analysis information. A path in the program's CFG is chosen, from which its predicate is derived as a set of constraints on input symbolic values, that is subsequently solved to generate a test case that exercises this path, as it satisfies that path

predicate.

Symbolic execution is a typical example of the Static approach. By executing a program symbolically, an algebraic expression including symbolic input variables for a given path can be obtained, instead of working with actual values for these input variables in a given path. The problem is then reduced to solving the path constraint generated to find suitable test data satisfying the test requirement.

Unfortunately, a number of problems may arise with symbolic execution. For example, due to features of programming languages, infeasible paths in loops with a variable number of iterations cannot be detected, indefinite loops may be encountered, or difficulty might be experienced dealing with a situation involving dynamic memory allocation and pointer management in structured programming languages. Furthermore, building a language-specific symbolic evaluator is no trivial task. The *complexity* of verifying feasibility of path traversal conditions, especially those involving non linear constraints, also proves too high to be tackled efficiently and automatically.

Symbolic execution is probably useful for the basic test data generation of straight forward code, but these inherent problems prevent it from possibly being applied to the general applications practically. Thereby, this lack of generality compared to the dynamic approaches causes its exclusion from this study.

2.3.2.2 Dynamic approach

The relationship between input data and internal variables for structural test data generation is difficult to analyse statically in the presence of loops and computed storage locations. Instead of using *variable substitution*, the Dynamic approach runs the program in question (usually in more than one pass) with some starting randomly selected input, and then simply observe the results via some form of program instrumentation.

Code instrumentation will also monitor and report if program execution follows an intended path. Search methods can be varied to pursue more “interesting" paths. Variables are then updated each time before the next execution, until the desired path is achieved, at which point, the associated test case is generated.

Consequently, the values of variables at any time of the execution are used to find more adequate test data. This paradigm is based on the idea that even if some test requirement is not satisfied, data collected during that phase of program execution is still useful as additional information to guide the test inputs towards coming closer to satisfying the given test requirement. With the help of such feedback mechanisms, test inputs can be incrementally refined until desired.

Since array subscripts and pointer values are known at runtime, this approach does not suffer from many of the problems associated with static approaches like symbolic ex-

ecution. However, it is not without drawbacks: the *lack of scalability* accounts for the great cost associated with possibly requiring many iterations, before a suitable test input is found. This also incurs additional execution time of the SUT, considering that the search space for test inputs is so vast that exhaustive enumeration is infeasible for any reasonably-sized program.

Future direction of research in this area might extend the limited problem domain from testing programs of a purely numerical nature, to supporting ones involving strings of special values in predefined orders, like date time values, dynamic data structures, such as lists or trees, containing characteristic “shape" information, or objects with internal state.

Systematic (optimisation)

The systematic methodology combines the results of actual program executions with a search strategy, for instance [**?**], and optimising for condition/decision coverage of complex C/C++ programs [**?**]. It does this by taking a more radical view: it transforms test data generation into another function minimisation problem, conducting an ordered heuristic search over the program test space for inputs which minimise the desired objective function, which represents the specified test criteria, until a particular branch in a path is taken. This causes the search to be directed into potentially promising areas of the search space first.

Hill climbing is a well known local search algorithm based on the concept of progressional improvement of an initial randomly chosen solution from the program search space as a starting point by investigating its neighbourhood of other candidate solutions. Simulated annealing is similar in principle to hill climbing, but reduces dependence on the starting solution, to avoid getting stuck in a local minimum.

One of the most successful search strategies in this category is using simulated evolution to evolve candidate solutions, using operators inspired by genetics and natural selection, otherwise known as genetic algorithms [**?**]. Genetic algorithms, probably the best known form of evolutionary algorithms, basically encode complicated data structures into simple representation of bit strings, which subsequently undergo transformations to eventually yield test cases. The key ingredients to this process include:

- chromosomal representation of solution to the problem
- initial population of solutions
- evaluation function for rating the “fitness" of solutions
- genetic operators to alter the structure across each generation

- parameters of the algorithm - population size, probability of applying genetic operators to solutions

Genetic algorithms have also been applied to automate unit test data generation for object-oriented software [**?**] [**?**], namely with respect to maximising program branch coverage, based on initial random guesses and typical usage profiles, followed by collecting previous execution traces.

This success attributed could be because unlike other function minimisation solutions, a global, as opposed to a local, minimum is sought for the value of interest. There is already a substantial amount of existing work in this that it has been experimentally shown to be of the best overall performance [**?**], so this paper would not be considering further exploring this technique at all.

Random

In Random test data generation, inputs are produced at random until a useful input is found. It is the simplest by far of all test data generation techniques, applicable to any sort of programs, and best used as a starting point for research in this field. It is commonly reported in literature, easy to implement, and therefore frequently used as a benchmark for other research work.

However, due to its simplicity, it is also unable to perform well reliably, in terms of test code coverage criteria, against a series of sample programs written for various different use cases. This is because of the even probability distribution over function input argument value selection, so it follows that pathological values, representing a small percentage of program input, are highly unlikely to be chosen to generate test data from.

Several mitigations exist: supplying a feedback mechanism to random testing in RANDOOP [**?**], or diversifying test configurations over picking optimal ones in *Swarm testing* [**?**].

Random Tester for Object Oriented Programs (RANDOOP) builds inputs incrementally by randomly selecting a method call to apply, and finding arguments from among the history of previously constructed inputs. This factor constitutes the feedback-directed variation in random generation, and has been experimented generating test inputs for Java and .NET container classes. The conclusion of this paper finds this technique scales viably to large systems, quickly discovers errors in heavily-tested, widely-deployed applications, and achieves behavioural coverage on par with existing systematic techniques.

Swarm testing is a novel and inexpensive way to do this by deliberately omitting certain API calls or input features. In this manner, it is more likely to trigger a capacity bug in a stack Abstract Data Type (ADT) with too many `push()` operations invoked in a test

case.

2.3.3 Hybrid testing

The hybrid approach seeks to adopt flavours of both the functional and structural testing methods to gain each of their advantages, and mitigate the disadvantages. This form of solving the problem does not require as many executions to search for an appropriate test input satisfying the test requirement. Here tests are derived from a specification of desired behaviour, but with reference to the implementation details, like finding test cases that violate assertion conditions, which can be infused by the programmer into program code.

It also attempts to overcome the limitations of symbolic execution such as handling arrays and pointers, by making known the index of the array during program execution when those structures are used. This results in some sort of iterative refinement of arbitrarily chosen test inputs, more formally termed as an 'iterative relaxation method' (IRM), much like that in numerical analysis, which improves upon an approximate solution to a given predicate equation.

Assertions specify constraints that apply to some state of a computation. When an assertion evaluates to false, an error has been found in the program. Assertions can be embedded within comment regions, either as boolean conditions or as executable code. By way of illustration, this is precisely the mechanism the generated unit tests use to indicate the status of observable failures. Tools have been written [**?**] which automatically generate assertions for runtime errors such as division by zero, array boundary violations and overflow, as well as find input test data to simulate error conditions where variables are uninitialised, yet used somewhere in the program code.

Exceptions are defined as as runtime error conditions in code. An example of exception condition testing can be found in one of the components of a search based automated test data generation framework for safety-critical systems [**?**].

The exception handling code is, on the whole, the least documented, tested and understood part of any system. Weaknesses tend to occur undiscovered here since exceptions are expected to occur only rarely during normal program execution, leading to bug exploit vulnerabilities, and these execution paths are commonly the first to be under attack. Yet failure to produce test data which checks the correct handling of conditions by raising exceptions could incur severe losses, although some care still has to be taken not to generate test cases which are impossible in practice during actual system operation. Test data can also be generated to inspect the structural coverage of the exception handler.

2.3.4 Testability transformation

Another interesting paper promotes the idea of testability transformation [**?**], where source code is refactored to facilitate software testing, like unrolling loops for example. A testability transformation is a source-to-source program transformation ("rewrite rules") that aims to improve the ability of a given test generation method to generate test data for the original program. It is rather ingenious in that a preprocessor first instruments the incoming program source to better suit the testing framework, before unit tests are generated for it, while still allowing many traditional transformation rules to be applied. It is hoped thereby that this algorithm would improve the performance and adequacy of the test data generated by the testing framework in this way.

2.4 Existing tools

After briefly reviewing the general field of automated software testing techniques, this section will now go into more specific details, by describing similar tools related to this project, beginning with an automated unit test data generation tool which uses genetic algorithms to test Ruby code, written in Ruby. This will be followed on by evaluating the available Python unit test generators.

2.4.1 RuTeG

The closest automated unit test generator in a dynamic language is Ruby Test case Generator (RuTeG) [**?**], a tool written in Ruby, which uses genetic algorithms on Ruby source code to automatically create unit tests in Ruby. Apart from the description in the paper, there is no source code available online to try it out.

The tool adopts a evolutionary search-based software testing (SBST) approach for dynamically typed programming languages, in this case Ruby, that is capable of generating test scenarios ranging from simple to more complex test data. It improves upon existing work in structural testing by supporting additional input data types that are frequently used, especially in object-oriented programs, where often parameters are objects themselves that maintain an internal state, or are complex and compound data structures that require appropriate initialisation. In such situations, test data generation pursues a specific goal.

The paper claims that the tool has successfully been applied, where experiments on real-world Ruby projects show that it achieves full or higher statement coverage on more cases and does so faster than randomly generated test cases.

2.4.2 Python

With respect to automated test generation in Python, it can be said that there is only minimal work done. The most recent tools - pythoscope v0.4.3 (Feb 2010) and pytests-generator v0.2 (Feb 2009) - perform static analysis on Python source code, as opposed to dynamic testing on Python bytecode (*.pyc files). The next subsections are dedicated to examining these existing tools in greater detail.

2.4.2.1 FizzBuzz (sample Python module)

First of all, this paper will be using a Python variant of the proverbial FizzBuzz program to demonstrate and measure the effectiveness of these Python automated unit tests generator tools, so this subsection will introduce this program to a sufficient level of detail.

The FizzBuzz program, just under 10 lines long, is a common basic technical interview question posed when hiring entry-level programmers. The adapted problem statement reads:

> “Write a function that takes a single input numeric argument, and returns "Fizz" if it is a multiple of three, "Buzz" if it is a multiple of five, and "FizzBuzz" if it is a multiple of both three and five.”

Listing 1 FizzBuzz Python module

```
1 def fizzbuzz(i):
2     if i % 15 == 0:
3         return 'FizzBuzz'
4     elif i % 3 == 0:
5         return 'Fizz'
6     elif i % 5 == 0:
7         return 'Buzz'
8     else:
9         return i
```

Nodes corresponding to decision statements (for example an `if` or `while` statement) are referred to as branching nodes. In this example, node #5 is one such branching node. Outgoing edges from these nodes are referred to as *branches*. The condition determining whether a branch is taken is referred to as the *branch predicate*. For the branch from node #5, the branch predicate corresponds to the boolean expression `i % 15 == 0`.

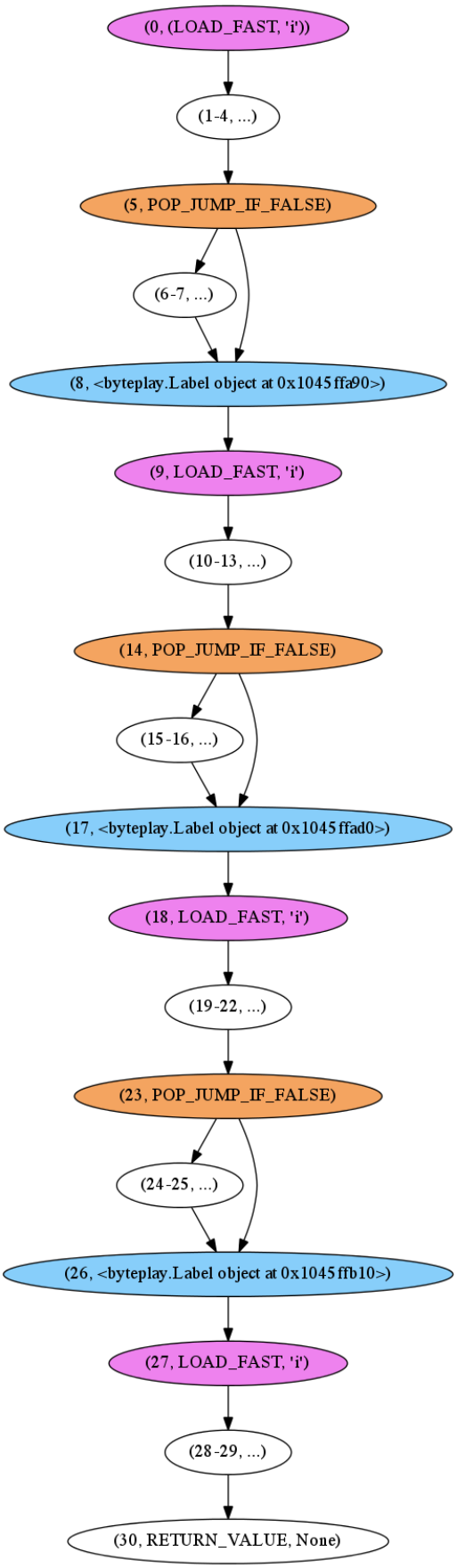


Figure 2.2: FizzBuzz Python module CFG

2.4.2.2 Pythoscope

Pythoscope is an open source unit test generator for Python code, written in Python, licensed under the MIT license. Ideas were contributed by Paul Hildebrandt and Titus Brown, and most of the code so far has been written by Michal Kwiatkowski.

This commandline tool, though extremely easy to setup and use, does not perform the expected automated unit test generation as advertised, but instead only produces a very rudimentary unit test stub below:

Listing 2 Unit test suite generated by Pythoscope

```
1  import unittest
2
3  class TestTriangle(unittest.TestCase):
4      def test___init__(self):
5          # triangle = Triangle(a, b, c)
6          assert False # TODO: implement your test here
7
8  class TestClassifyTriangle(unittest.TestCase):
9      def test_classify_triangle(self):
10         # self.assertEqual(expected, classify_triangle(triangle))
11         assert False # TODO: implement your test here
12
13 if __name__ == '__main__':
14     unittest.main()
```

2.4.2.3 Pytestsgenerator

This automated unit test case generator creates unit tests for Python modules. The authors Vijakumar and Karthikeyan developed this tool on 32-bit Linux, and it only works for that platform and architecture. Its purpose is to simplify usage of the existing PyUnit framework, and generate logical test cases for classes and methods. WxPython is required as it powers the GUI, but there is also a CLI offered at the same time. The application is packaged for distribution using the distutils module.

According to accompanying documentation, this tool is intended to accomplish the following objectives:

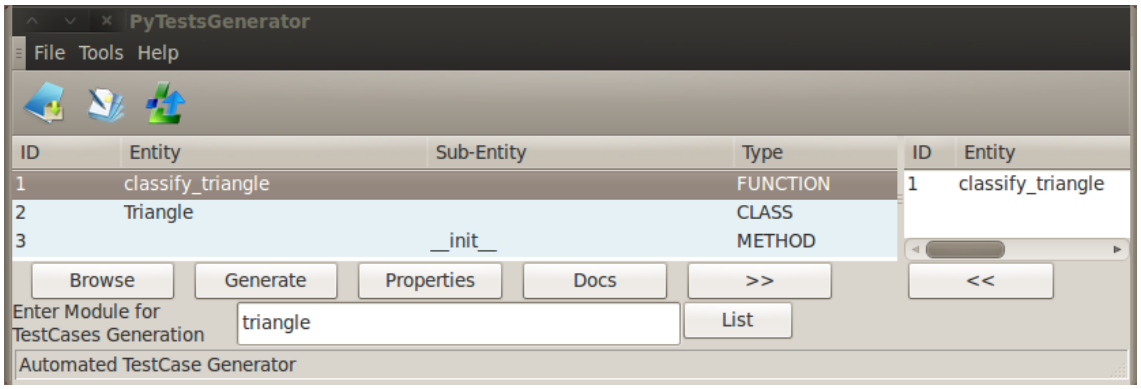
- Read a specified python module
- List the Classes, Functions and Properties of that module (for the user's selection)
- Drill down the Classes for methods and properties

- Generate basic set of test cases for each class or method selected

The predetermined logic for the test cases to be generated include:

- Number of arguments
- Valid arguments
- Invalid arguments
- Custom logic

Following is an example demonstration of the software targeting the sample Python module:



Using this tool then creates the following unit test stub, included below for comparison:

Listing 3 Unit test suite generated by pytestsgenerator

```
###Generated using PyTestGenerator
#!/usr/bin/env python

#Format
#test_<Test_Number>_<Entity_Name>[<Arg_Status>]
#         <Predicted_Status>_<Comment>

import unittest
import sys
import triangle

class PyUnitframework(unittest.TestCase):
        '''Test Cases generated for triangle module'''

if __name__=="__main__":
        testlist=unittest.TestSuite()
        testlist.addTest(unittest.makeSuite(PyUnitframework))
        result = unittest.TextTestRunner(verbosity=2) \
                .run(testlist)
        if not result.wasSuccessful():
                sys.exit(1)
        sys.exit(0)
```

As evident from the code snippets of the generated unit test stubs, both these tools are still fairly inadequate for automatically discovering unit tests for arbitrary Python programs.

2.5 The Python programming language

Python is a general-purpose, multi-paradigm (imperative/object-oriented/functional programming styles), high-level programming language, whose design philosophy emphasises code readability. It features a fully dynamic type system and automatic memory management. Its syntax is said to be clear and expressive. It is often used as a scripting language, but can also be applied in a wide range of non-scripting contexts, popular with the numeric and scientific community.

Python has a large and comprehensive standard library. Using third-party tools, Python code can be packaged into standalone executable programs. Python interpreters are available for many operating systems. Its reference implementation is CPython.

Unlike statically typed languages, Python features a number of expected runtime charac-

teristics unique to the class of dynamic languages: dynamic typing, interpretation (seamless source code compilation when needed), introspection/reflection and runtime modification.

2.5.1 Dynamic typing

The beauty of dynamic languages is dynamic typing, otherwise also expressed as 'duck typing', meaning objects are described by what they can or cannot do, i.e. by the methods it responds to at runtime, instead of being associated to a specific type.

Pythonic programming style that determines an object's type by inspection of its method or attribute signature rather than by explicit relationship to some type object. By emphasising interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. Instead, it typically employs the EAFP (Easier to Ask Forgiveness than Permission) style of programming.

Values possess types instead of variables, and often the type of objects sent in as function arguments or produced in method invocations as return results are not strictly checked against.

When code is executed, the execution environment need not care what type an object has, only if it implements the methods that are called on it. This makes Python an attractive prototyping language for this study, yet at the same time poses complications (difficulty in identifying input data for method invocations, because arguments can be used in different ways) in searching for adequate test cases, due to its dynamic nature. It is therefore essential to limit the size of the search space to be considered to maintain a reasonable execution time for generating tests.

2.5.2 Introspection/reflection

This makes it much easier to collect relevant information about classes and methods at runtime. It is not only possible to query for the availability of methods during object creation as well as the code implementation behind it, but also to search for methods that may change the internal state of an object as well as identifying their arguments.

2.5.3 Runtime modification

Runtime modification is a central characteristic of dynamic languages. A type, value or code object can typically be altered (changed/added) during runtime in a dynamic language (actual allowed behaviours vary between languages). This means generating new objects from a runtime definition, creation and loading of entire new system modules, or changing the inheritance or type tree, thus changing the way existing types behave,

especially with respect to method invocations.

The use of this feature usually only exists in a small percentage of code like creating this automated unit test generator, whilst the rest is designed in a more traditional manner. For these reason, the testing of code including the `exec()` or `eval()` keywords are excluded from this study.

2.5.4 Data model

This subsection offers a summary of Python's data model.

2.5.4.1 Objects

Everything in Python is an object. Objects are Python's abstraction for data. All data is represented by objects or relations between objects. Objects are never explicitly destroyed, but garbage collected (according to the reference counting scheme for the standard implementation CPython).

Objects are modelled as entities, each ascribed with three properties:

1. **identity**

   assigned permanently upon its creation, like a fixed memory address location

   queried using `id()`

   comparison is made using `is`

2. **type**

   unchangeable

   queried using `type()`

   determines operations it supports, e.g. `len()`

3. **value**

   may be mutable, as in the case for `dict` and `list`

   or immutable, as in `numeric`, `basestring`, or `tuple`

For immutable types, the operation computing a new value may actually return a reference to an existing object with the required type and value (if available), while this is not allowed for a mutable object.

2.5.4.2 Type objects

Type objects share the following traits:

1. they are used to represent *abstract data types* in programs

2. they can be *subclassed*

   `<type 'type'>` is a subclass of `<type 'object'>`

   `<type 'object'>` is a subclass of no object

3. they can be *instantiated*

   an instance of the object is created using the `__new__()` and `__init__()` methods of the type object

   existing type object becomes the `__class__` for the new instance of this object

   the type object to use is specified through the `__metaclass__` class attribute, but the type object specified must be a subclass of the type of the base object

4. the type of any type object is `<type 'type'>`

5. `<type 'type'>` is the *type of all types*, from which `<type 'object'>`, the *base of all other types*, inherits

6. *type* is equivalent to *new-style class*

7. an object is only a *type* if it is an instance of `<type 'type'>`, e.g. `<type 'type'>` is an instance of itself

2.5.4.3 Containers

Containers are objects containing references to other objects, such as `tuple`, `list` and `dict`.

The full description is attached in the Appendix **??**.

2.5.4.4 Bytecode

The following describes one sample of the intricate relationship between Python bytecode and code objects.

Listing 4 Variable scope in bytecode

```
1 def outer(aa):
2         global aa
3         def inner():
4                 bb = 1
5                 return aa + bb + cc
6         dd = aa + inner()
7         return dd
```

Global variables, like `cc`, are unbound and loaded using the `LOAD_GLOBAL` bytecode instruction, found in the `co_freevars` attribute of the code object.

Local bound variables, like `bb`, are represented by `LOAD_FAST` bytecode instruction, and equivalently found in the `co_varnames` attribute of the code object.

Variables bound in outer scope, like `aa` in the context of `inner()`, is loaded using the `LOAD_DEREF` bytecode instruction, and accessed through the `co_cellvars` attribute of enclosing code object.

The constant `1` on line 4, is loaded by `LOAD_CONST` and found in `co_consts` attribute of the code object.

The complete listing of code object properties is at Appendix **??**, and opcodes is at Appendix **??**.

2.6 Challenges

There are several facets of complexity to this problem, which this work hopes to tackle.

2.6.1 Function argument instantiation

Function arguments can range from basic primitive types, to dynamic data structures like lists, maps, and trees, to objects. Depending on available resources, scope for this project might be restricted to supporting only numeric types and objects.

These classes of values present distinct challenges, when they appear as input arguments to functions being tested. Intuitively, the search space for a primitive integer type, for instance, extends in one direction towards positive infinity if unsigned, and in both directions if signed. However, for complex data structures like trees, the notion of infinity manifests itself via nesting as well.

Even the usage of basic types may become quite complex, especially when there is only a small solution space, in which a certain condition can be satisfied. This might extend beyond single arguments to a combination of them, and this is exacerbated when arguments can depend on other values, or previous argument values.

This is especially applicable not only to class constructors, when creating appropriate objects, but also in automatically generating initial function input arguments.

It is vital to ensure that an exhaustive enumeration of the search space in search of pathological test cases is not performed, because there would quickly be an exponential blow up, especially in functions with multiple input arguments, as well as being inefficient, due to the side effect of generating many redundant or subsumed test cases.

There are several possible ways to conduct the search for such corner cases. Previous algorithms range from naïve systematic enumeration of all possible values to variants of random testing.

Another complexity factor is added when there are dependency between functions, arising from the argument to one having to be constructed by the other. This naturally enforces a fixed sequence in which to order the function invocations and entails that the automated unit test generation framework respect this ordering so as to create significant test cases.

Therefore, the task here is to come up with a more efficient way of prioritising pathological boundary parameter value generation, under real time and space constraints. Some leading intuition follows.

2.6.1.1 Lazy instantiation

It might be reasonable to begin with "lazy instantiation" [?], where dummy nullified objects are passed in initially. Test data is only generated for return values when the methods on them are actually invoked. This supposes multiple runs through the same code block, and using feedback from previous iteration to direct future execution.

As for implementing the idea of *lazy instantiation*, IRULAN [?] is the canonical reference tool written to demonstrate this concept in Haskell. This project intends to investigate further into the feasibility of applying this concept to Python.

A sample execution to discover errors in the following code snippet

```
module IntTreeExample where
data IntTree
        = Leaf
        | Branch IntTree Int IntTree
insert :: Int -> IntTree -> IntTree
insert n Leaf = Branch Leaf n Leaf
insert n (Branch left x right)
        | n < x = Branch (insert n left) x right
        | n > x = Branch left x (insert n right)
```

produces the following output:

```
$ irulan --ints='[0,1]' --enable-case-statements -a --
   maximumRuntime=1 source IntTreeExample
...
insert 1 (Branch ? 1 ?1) ==> !
IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in
   function insert
insert 0 (Branch ? 0 ?1) ==> !
IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in
   function insert
case insert 0 (Branch (Branch ? 0 ?1) 1 ?2) of
```

```
Branch x _ _ -> x ==> !
IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in
   function insert
case case insert 0 (Branch (Branch (Branch ? 0 ?1) 1 ?2) 1 ?3)
   of
Branch x _ _ -> x of
Branch x _ _ -> x ==> !
IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in
   function insert
...
```

2.6.1.2 Runtime in-memory manipulation

It is also envisioned that the dynamic language features of Python be exploited in order to rapidly generate useful test data. One idea is to manipulate and observe the behaviour of code blocks in memory at runtime, by monkeypatching or hotswapping code under test (CUT) for stubs, but with a hook to log incoming parameters during a sample execution, in order to determine their initial starting range & types.

On a related note, a cross-cutting concern such as logging may be implemented using the concept of Aspect Oriented Programming (AOP), with tools like pytilities, Aspyct, aspects.py or PythonDecoratorLibrary.

2.6.1.3 Random testing

Apart from random testing with feedback RANDOOP [**?**], and preferring configuration diversity over a single optimal test configuration in Swarm testing [**?**], another suggestion is to inspect stack frames of previous executions to grasp a better initial starting point for generating parameters.

2.6.2 Optimising search space coverage

The suggestion to parallelise the search space for interesting values over the entire range of integers for example, is to use the General Purpose Graphics Processing Unit (GPGPU) toolkit like Nvidia's CUDA or HADOOP cluster, of which its feasibility remains to be determined.

Alternatively, parallelism has already been achieved [**?**], by running multiple processes simultaneously on a network of workstations or on a single multi-core processor, with a user-determined numerical parameter. In that experiment, each process running on separate workstations communicated to maintain synchronisation via a software facility.

The benefit is clear: reduction in execution time by a factor of the number of parallel

processes. This seems possible, given a way to partition the test case generation workload into several balanced independent components, according to the available resources.

2.6.3 Testing a dynamically typed language

Much of the body of work in the software testing community concerns testing against static languages, rather than dynamic languages, or even Python in particular.

Dynamically typed languages are characterised by values having types, but not variables, hence a variable can refer to a value of any type, which can possibly cause test data generation to become more complicated. Python therefore heavily employs duck typing, to determine an object's type by inspection of its method or attribute signatures.

Tools arising from research efforts into testing for static languages lacks adequate support for code written in dynamic languages, including typical features such as `eval()`, closure, continuations, functional programming constructs, and macros, thus this paper aims to look into this further, in the context of Python.

2.6.4 Non-terminating program executions

Another difficulty associated with this problem domain is detecting infinite executions when generating test code. This can be most commonly attributed to (the error of) infinite loops present, which may even be nested. It is impossible to detect all kinds of loops fully automatically, but many such can [**?**]. An immediate solution is to implement timeouts, with custom duration according to CUT. Early detection so as to improve efficiency is difficult.

2.6.5 Early detection of path infeasibility

The paper [**?**] claims one of the most time consuming task of automatic test data generation is the detection of infeasible path after execution of many statements. Hence, backtracking on path predicates [**?**], satisfiability of a set of symbolic constraints [**?**], selectively exploring a subset of "best" paths [**?**] are some of the past attempts at solving this issue. This is a major problem of test data generation based on actual value, incurring both costly and unnecessary computation.

2.6.6 Improving code coverage

Achieving consistently high code coverage over a wide range of programs (not to mention running within reasonable time and space) via generated test cases ultimately defines the extent of success of this project. This allows for effective fault detection, which may be of different types. An alternative measurement of code coverage improvement involves identifying error prone regions of code where more rigorous testing would prove bene-

ficial [?] [?]. There already exists other empirical studies for code coverage in different test data generation algorithms documented, providing some competitive standards to match up to [?] [?] [?].

2.7 Summary

In this chapter, the relevant background literature and theory to understand this project has been explained in sufficient detail. The next chapter presents the theoretical foundation underpinning this project's contributions to the field of software testing, including the problem specification, approach taken, and algorithms and techniques used, to name a few.

Chapter 3

Contributions

This section will highlight some of the more pertinent aspects of this project, and discuss certain design choices made during its development.

3.1 Key characteristics

The strategy emphasises dynamic test data generation, where intermediate runtime data is gathered, represented in some suitable form, and used to guide subsequent testing iterations.

Tests are mainly generated at a function level granularity, largely for top-level functions, in a sensible order as the runtime engine discovers such that type inference information is maximised, and the tool does not require any additional user input other than the target Python module or package to be tested.

SUT is modelled as a combination of one `__main__` program entry point, top-level functions, together with classes containing fields and methods, as illustrated below.

Listing 5 Shape of SUT

```
class A(object):
        def __init__(self, attr1,...,attrN):
                self.attr1 = attr1
                ...
                self.attrN = attrN
        def method1(self, param1,...,paramN):
                <method body>

def function1(arg1,...,argN='default'):
        <function body>
        return result

if __name__ == '__main__':
        function1(1, '2')
```

Functions are restricted to well-formed entities which receive some input arguments, uses some or all of these in its body for computation, and finally returns some of these

values.

Listing 6 Shape of a test function

```
def function1(arg1, arg2, ... argN='default'):
        <body>
        return result
```

The output of unit tests generated commonly consist of the following parts (embedded within a unit test suite with custom helper methods like `assertNotRaises()`:

Listing 7 Shape of generated unit test

```
def test_<function name>_<test name>(self,<additional parameters>):
        <statements to setup and initialise parameters>
        <execute function, storing return value>
        <assertions on return value, e.g. assertIsNone()>
```

3.2 Limitations

This project assumes unobsfucated (so reverse engineering and code reconstruction lies out of the scope of this investigation), compiled SUT, containing only valid syntax. For example, names and identifiers conform to the Python 2.7 Grammar specifications laid out in Appendix **??**. Code is without any Exceptions under typical execution, like raising a `ZeroDivisionError` for binary operators `/`, `//` or `%`. There must also be the absence of any underlying non-termination conditions like infinite loops and missing base cases in recursions.

Moreover, we are dealing only with Object Oriented (OO) style Python modules, i.e. those involving functions, classes and objects only. Only new-style classes will be considered (explanation of difference between new-style and classic/old-style classes can be found in Appendix **??**), as the old-style classes will be deprecated in Py3k. Internal methods, i.e. those surrounded by '__', are also ignored during test generation.

The scope is also restricted to exclude test data generation to cover branches with string predicates, for which another study addresses string equality, string ordering and regular expression matching [**?**].

3.3 Approach

The proposed solution is a Mac OS X compatible, Python 2.7 module, developed incrementally, which performs bytecode inspection with runtime construction of control flow graph (CFG) / control dependence graph (CDG).

The key deliverable from this project will be unit test suites, in terms of a language-neutral Javascript Object Notation (JSON) format, consisting of various assertions, capture expressions, and value assignments. This affords flexibility in later system extensions to target other dynamic programming languages.

An API may be exposed if there are reusable components, eg. algorithms, developed in this tool. It is also planned to provide visualisation of this process, in the form of a PyQt-powered GUI frontend.

The resulting end product can be applied to regression testing as well, to report changes in behaviour across different versions, as software evolves over time. It will also try to generate unit test for algorithms, as these tend to be the places where software bugs are more likely to happen, with the slightest of changes. Other code will normally be API calls encapsulating these functions, and are less likely to have bugs in themselves, but rather expose the hidden bugs in the algorithms used behind it.

3.3.1 Testing recursive functions

A recursive function is modelled as one which returns a numeric value, and at most 1 numeric input argument is active in the recursive calls of the same function. It is properly formed, meaning the base and recursive cases are well-defined. There is an arbitrary, configurable recursion depth limit of 1024 calls on the stack at any one time and a maximum of 3s runtime imposed. Some example functions include the factorial, fibonacci, towers_of_hanoi, greatest_common_divisor, binary_search, quicksort, and mergesort programs.

3.4 Challenges

This subsection details several major challenges faced during this project.

3.4.1 Python modules with own imports

In the course of this project, many difficulties arose as a result of developing the solution. One such concern is how to test Python modules which contain their own imports. Python modules could not be successfully imported if their dependencies were not first imported.

This is evident when testing Python packages, like the Python graph library. In the end, this issue was resolved by adding the base package directory to Python's `sys.path` list, where Python searches for its required Python modules, and also using the `__import__` keyword to dynamically import modules by supplying their name as a string.

3.4.2 Lack of existing tool support

There exists an abundance of tools to parse, manipulate and execute Python source code, but hardly anything equivalent for Python 2.7 bytecode. Many of the tools were developed for outdated versions of Python, and then later became abandoned projects, or their source is no longer available online.

Hence, this project had to write a bytecode function parser, manipulator and reconstructor on top of existing libraries like byteplay or Peak Bytecode Assembler. It also contains tracing capabilities to measure bytecode instruction coverage, as opposed to conventional tools measuring Python source code statement coverage.

3.4.3 Vague error messages

Metaprogramming is instrumental to the success of this project. However, there is limited documentation and support for this available online, so many of the extensions had to be experimentally developed. One of the greatest problems occurs in type inference during lazy instantiation. Consider the following code fragment, a simple function meant return the result of adding two of its input arguments:

Listing 8 Simple addition function

```
1 def add(a,b,c):
2         return a + c
```

Because the Python source is unknown at the time of test generation, and only the Python bytecode is given to be executed over, all possible permutations of types in Python (see Appendix **??** for a complete listing) need to be exhaustively enumerated for all the input arguments actively used within the function.

Due to the way the `+` operator is defined, it is usually able to only act upon operands derived from the same base type. If the types of the operands are incorrect, for instance, the call `add('a',1,2)` throws the following error:

```
TypeError:cannot concatenate 'str'and 'int'objects
```

The error object produced is not only just an arbitrary `String` message, but also only carries with it extremely little information. It does notify the function caller of the conflicting types of the offending arguments causing the addition failure, but is totally ambiguous with respect to exactly which of the arguments in fact need to be corrected (whether the second or third argument in the order listed), especially when the source is invisible.

In order to solve this, the project implements proxy meta-parameters, which throw a more descriptive error message, with the relative index of the function arguments causing the error, in this situation. This situation is also triggered when object fields or meth-

ods are missing, thus the internal `__getattr__(self, name)` method is overridden to throw the more descriptive version of the error message in such a case.

3.4.4 Relationship between input arguments

It is not clear how to efficiently determine the way input arguments to functions are related, to each other and the function, but being able to grasp this concept will prove beneficial in generating unit tests. Some complications are detecting when an input argument to a function listed later reuses an earlier argument in its initialisation, a redefinition of an input argument in the function body (namespace pollution), and which input arguments are actually (directly or indirectly) influencing the function's return value.

3.4.5 Various kinds of programs

There are many sorts of programs, such as those involving loops/Python generators, list/set/dict, conditionals, recursive calls, input data type - numeric: integer or floating point, or classes and objects.

Each of these programming constructs may occur simultaneously, nested within each other, presenting an additional factor of complexity when trying to maximise code coverage in a consistent manner across these various programs, as it is not always possible to reuse partial unit tests generated for a code block in its parent code.

3.4.6 Range of values for testing

Even after the types of function input arguments are perfectly inferred, there still remains a massive and virtually infinite range of values to be tested for, even if the function just takes a single numeric input argument, not to mention for an input vector.

As the search eventually takes too long to complete, one possible solution is to extract constants from the Python bytecode as candidate values for testing first, then try the minimum, midway, and maximum range of the data type, as these are more likely the pathological values used in boundary value checks in code, then randomly generate test data with feedback later, based on the types of these values, and in the region of the range of these values.

In addition, hints can also be gathered from existing test cases manually written for the SUT for common test patterns.

3.5 Available tools

3.5.1 Parsers

The most popular among the parsing tools by far is pyparsing. However, it only parses Python source code to Abstract Syntax Tree (AST), so it is not sufficient to handle Python bytecode.

Instead, a custom bytecode function parser, using a 'context stack', was written on top of the byteplay Python package. The byteplay package is excellent in transforming raw bytecode into a standard list of (opcode, argument) pairs for subsequent easy manipulation.

3.5.2 Unit test frameworks

It is fairly common to write unit tests in the main program entry point of a Python module. This is great, except that it can only be run when within that particular module. Therefore, this has led to the creation of unit test discovery and execution frameworks, so that tests can be added to existing code, executed, and a report generated simply and quickly.

In addition to finding and running tests, such frameworks allow selective execution of certain tests, capture and collation of error output, and coverage and profiling information. There are many unit test frameworks in Python, but the Python nosepackage, a wrapper around the Python unittests module, is actively developed, stable, integrates well with distutils, easily extensible with its plug-in architecture, and used on a number of major projects, so this project has chosen to use nose too.

3.5.3 Graphical User Interface (GUI) toolkits

There are several options, namely, PyQt/PySide, PyGTK, wxPython and Tkinter. The GUI demo frontend was written in PyQt for this project, by preference, and because of the advantages of it being cross-platform, provide native widgets style-able by Cascading StyleSheets (CSS), and a QtDesigner tool. PyGTK and wxPython are also possible, but Tkinter was too simple and least feature complete for this purpose.

3.5.4 Code coverage

Coverage.py is a tool for measuring code coverage of Python modules. It monitors your program, noting which parts of the code have been executed, then analyses the source to identify and mark these sections of code. Coverage measurement is typically used to gauge the effectiveness of tests. It can show which parts of your code are being exercised by tests, and which are not. It is still in active development and supported on Python 2.7,

so this project is using this to fingerprint test target Python modules.

3.5.5 Alternative implementations

Psyco is a Python extension module which can greatly speed up the execution of any Python code. It has since given way to alternative Python interpreters like PyPy and Unladen Swallow. These have been experimented with to try and increase the speed of execution for automatic unit test generation due to the large amount of computation needed, and not sacrifice the comprehensive library support at the same time.

3.5.6 Supporting utilities

virtualenv/pip are the indispensable supporting tools for every Python project. virtualenv is a tool to create isolated Python environments, while pip is a tool for installing and managing Python packages, as found on the PyPI package repository.

3.6 Design decisions

This is a short section describing the preference of using one Python function over another.

3.6.1 callable() vs. hasattr(obj, "__call__")

Classes can be used as argument types in Python's builtin `callable()`, which returns `True` only for callable types specified in Appendix **??**, but not `hasattr(obj, "__call__")`, which checks whether an object has a particular specified attribute.

The builtin function is more easily readable. The attribute lookup first tries to get that attribute, discarding the result, catching and handling any underlying Exceptions, while the builtin function only examines the type's structure, so the builtin function performs more quickly than the attribute lookup.

3.7 Algorithm

Test data generation uses the branch coverage algorithm to select a path that may reach the targeted branch and obtains constraint information for the selected path to then generate the test data inputs for the resulting test cases. This only applies to the relevant 'control points', i.e. absolute and relative jumps to labels in bytecode, where the conditional is somewhat dependent on the incoming function arguments.

3.7.1 Example

A simple example shall be provided to illustrate the basic approach of test data generation. Consider the program abs, a function to return the absolute value of a given input (numeric) argument.

The code is attached below:

Listing 9 absolute function

```
1 def abs(n):
2     return n if n >= 0 else -n
```

The function's CFG is as follows:

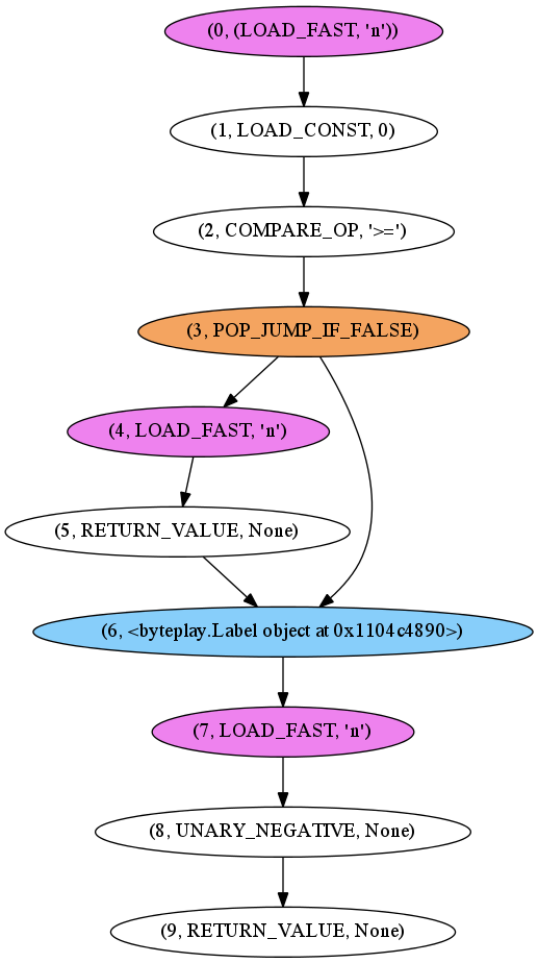


Figure 3.1: absolute function CFG

The goal of test data generation to optimise code coverage is then to find a particular input argument vector which will cause each of the program paths to be traversed.

The program is executed given a non-negative integer, and one of the subpaths is traversed successfully (edges of the path are highlighted in red):

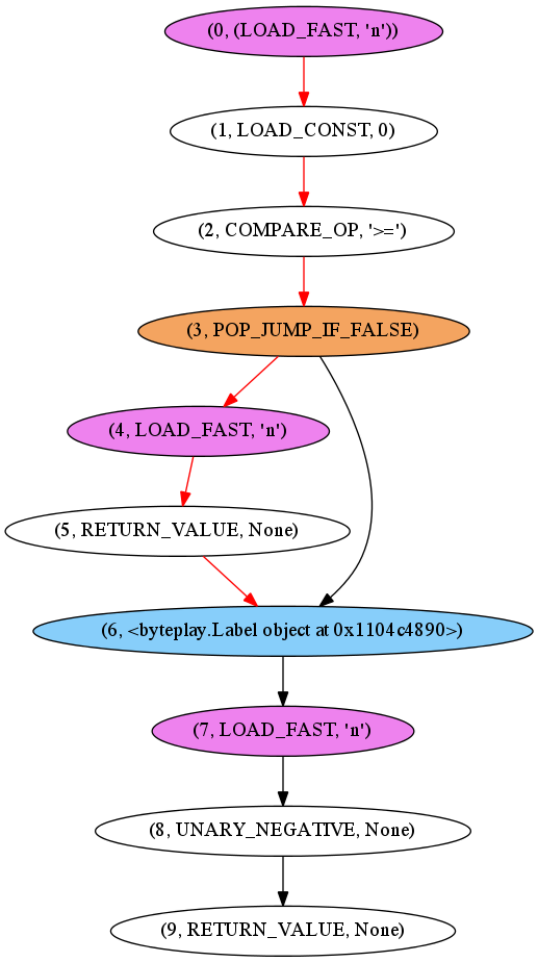


Figure 3.2: absolute function CFG - nonnegative n

3.8 Summary

In this chapter, the current research work has been highlighted. In order to demonstrate the idea of automated lazy testing in Python, an example usage of the tool is provided, using the ideas found in this chapter. The structure of this tool, and the examples, is the subject of the next chapter.

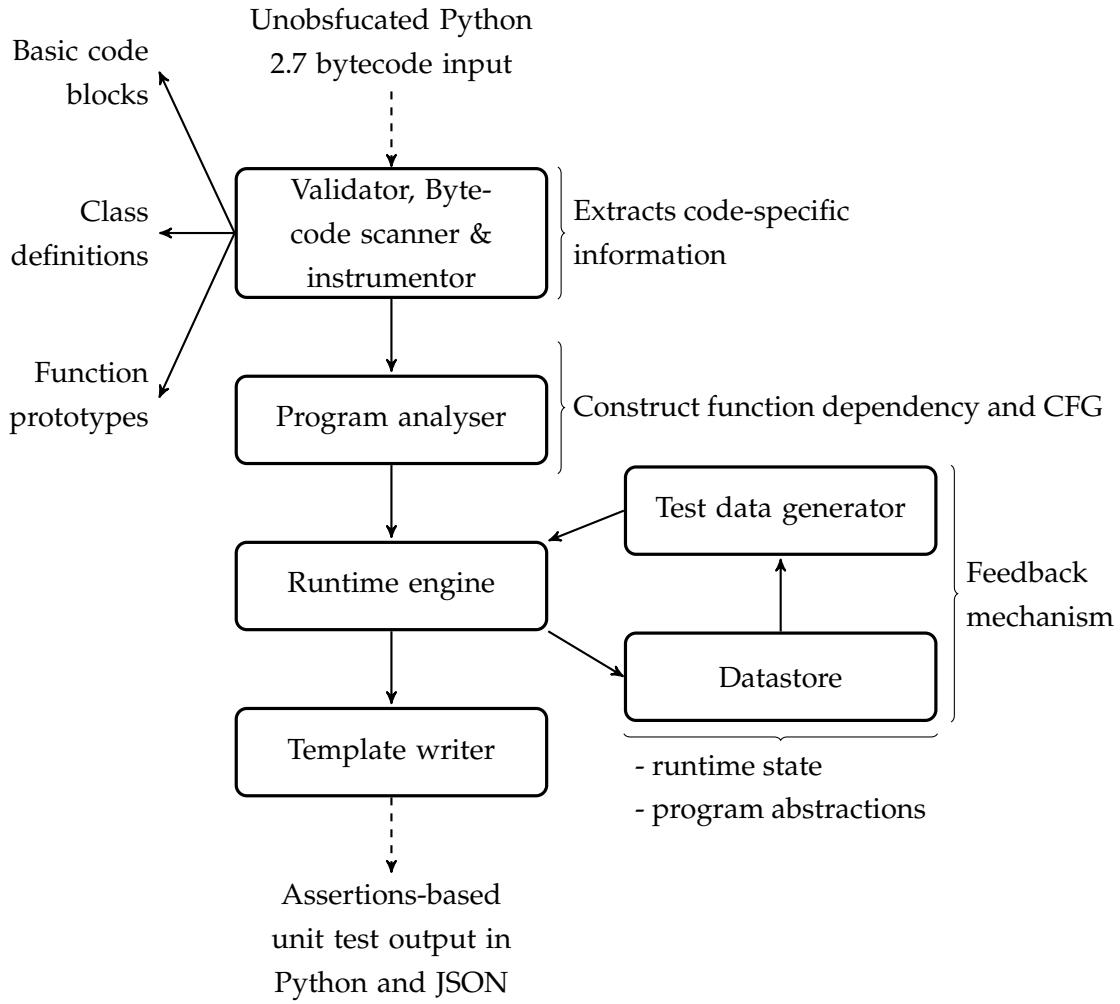
Chapter 4

SPLAT

In this section, the tool SPLAT is introduced, together with its various components.

4.1 Architecture

The software design architecture, except for the main program script and GUI demo frontend not being reflected, is outlined below:



4.2 Components

This section will review the various components that make up the tool. The tool is packaged up in a private namespace as a Python package and made available on PyPI.

4.2.1 Main run scripts

The tool comes with a GUI demonstration frontend, which is basically a PyQt-powered, fullscreen image viewer for visualising the CFGs of the generated unit tests. There is also the main program script which will execute all the dependent components in order to produce unit tests for the target Python module.

4.2.2 Validator

The validator is the first sanity check to ensure that the supplied Python bytecode file (*.pyc) or Python package directory is indeed valid, and thereafter record some relevant information regarding the package/module name, paths, and import locations for later use.

4.2.3 Bytecode scanner & Instrumentor

The bytecode scanner then builds upon this data cache to examine each of the Python bytecode files for which tests are requested, by enumerating all the possible information that can be gathered from the Python bytecode file, including decompiling the code objects contained therein. It also *instruments* the bytecode by hacking the offsets relative to line numbers, and reloads this new version into the `sys.module` namespace.

4.2.4 Program analyser

This is divided up into two portions: the function dependency analyser and the function CFG builder. The function dependency analyser scans the list of Python bytecode files accumulated previously for suitable functions to test, and then constructs a relationship diagram, depending on the type of these functions.

For instance, recursive functions are represented by a reflexive graph node, while an outgoing arrow from node x to node y indicates that the function represented by node x uses the one represented by node y. Hence, it is better to generate unit tests for the function represented by node x first, and track the arguments supplied to the implicit call to the function represented by node y, for type inference purposes, and also to minimise the number of unit tests generated.

The function CFG builder works at a far finer level of granularity, i.e. per function basis. It maps out the flow of the function's bytecode instructions into a coloured graph for further

analysis later during unit test generation. The outcome of these scans are manifested as graphs drawn using the pydot Python package.

4.2.5 Unit test generator

The unit test generator lies at the heart of the entire tool. It consists of a few distinct unit test generation phases, after setting up the respective “environment" variables.

4.2.5.1 all `None`s

As a start, as many `None`s as the number of required arguments in the function signature are passed into the function call.

4.2.5.2 all `None`s, with structure

This time, the list of arguments is enhanced with the right fields for objects, initialised with `None`s again, whose structure is derived from the Python bytecode.

4.2.5.3 `MetaParams`, with structure and defaults

The `None`s in the object fields, and other uninitialised used arguments, previously are now enriched with the proxy metaparameters mentioned before. Optional arguments used, but without a supplied value, are replaced by their defaults.

4.2.5.4 Lazy instantiation

Finally, the main program loop to automatically generate unit tests is reached. This loop runs until a predetermined maximum number of iterations, each time executing the same function, with an updated list of arguments, and terminating when no more `Exception`s are raised. This is followed by a series of random value unit tests conducted.

Within an execution cycle, `TypeError`s, `MetaAttributeError`s and other `Exception`s are caught and handled by changing the list of arguments where necessary. Unfortunately, error messages are only displayed as arbitrary strings, hopefully compatible between versions, so regular expression matching on these error messages containing proxy metaparameter and type error details is the primary mechanism adopted, through which additional information is captured to improve the overall code coverage.

4.2.5.5 Argument list to UnitTestObject

After the argument list is finalised in each case, it is all passed to this conversion function, which works by first recursively creating the equivalent unit test statements that will produce the proxy metaparameter objects for the relevant arguments when the unit test suite is executed.

Thereafter, it produces assertions related to whether a return value exists from the function invocation with that list of arguments, or an `Exception` is raised. If an `Exception` is raised, the function then figures out if that is a builtin or user-defined `Exception`, for which it will necessarily generate statements in the unit test to import the `Exception` Python module prior to asserting the `Excception`.

However, if no `Exception` is raised, then this counts also as an assertion, and the function will proceed to check if the return value is `None`, a primitive, or an object, for which it will output statements verifying value equality for primitives, and class and fields equality on objects. All these statements are put into a list, that is encapsulated by a custom UnitTestObject, to be read in by the Template Writer to automatically generate the unit test suite.

4.2.5.6 Recursive function tester

If a recursive function is identified, the testing logic is slightly varied. A `TRACE_DICT` is a dictionary containing important tracing information like the name of the recursive function being executed, recursion depth, execution stack frames, and UnitTestObjects generated so far, being propagated along the recursive function calls.

Each run of the function is wrapped within a `multiprocessing.Process`, so that an appropriate timeout can be enforced as needed, together with a logging facility, yet still retaining the original result collection and tracing capabilities.

4.2.6 Template writer

The template writer takes in the list of UnitTestObjects, each abstracting a single unit test automatically generated, together with the basic location information from the Validator, and populates a Pystache (Mustache templating system in Python) unit test suite template file with this data, writing both the Python unittests and JSON version out into a 'tests' directory suffix of the SUT. The output of generated unit tests are converted into language neutral JSON format, for translating to other languages in future.

4.2.7 Auxiliary tools

These range from the bytecode modifier, tracer to measure bytecode instruction coverage, and the proxy metaparameter Python class. There are also utility functions to perform common tasks such as adding timing aspects to functions, serialising and deserialising objects to dictionaries, prettyprinting in Terminal using the blessings Python package, formatting debug output.

There is a centralised constants store to keep information about the types of bytecode instructions and operators, the predicates for the inspect Python package, module types recognised by the imp Python package, and enumerations for colouring graph nodes.

4.3 Summary

In this chapter, the working automated testing tool has been shown, but exactly how well does it actually perform against current standards? This notion will be made more precise, both quantitatively and qualitatively, in the next chapter focusing on the evaluation of this work.

Chapter 5

Evaluation

5.1 Experiment

Experimental evaluation entails the following:

i. comparison with existing work, e.g. Pythoscope (2010), PyTestsGenerator (2009)

ii. benchmark against popular Python libraries, e.g. python-graph, and Python module implementations of famous algorithms

iii. measure quality of test cases generated using metrics, e.g. bugs and crash discovery

5.2 Evaluation criteria

An explanation of and in depth discussion into the adequacy of the different strands of test code coverage criteria and how the various metrics established are used to grade the effectiveness of this tool against the benchmark suite in automatically generating unit test data forms the theme of this section.

Throughout this section, P is a program under test and C is a selected test coverage criteria. T is a set of test data which satisfies 100% of test requirements of C for a P. M is a test data generation method and T' is a set of test data found by M for a P and C.

5.2.1 Code coverage

Code coverage is a measure used in software testing to describe the rigour to which the target program code has been tested. To quantify how well the program is exercised by a given unit test suite, one or more *coverage criteria* is used. Test coverage criteria serve as a means to explicitly state the degree to which a test requirement (i.e. statements, branches, or conditions) has been examined. There are a number of coverage criteria, with the main ones being:

5.2.1.1 Basic coverage criteria

- **Function coverage**: Has each relevant function in the program been invoked?
- **Statement coverage**: Has each node in the CFG of the program been executed?
- **Decision coverage**: Has every edge in the CFG of the program been traversed?

- **Condition coverage**: Has each boolean sub-expression evaluated both to true and false (where possible)?
- **Parameter value coverage**: In a function taking arguments, has all permutations of the common values for such arguments been considered? For example, a string could take any of these legal values - null, empty, whitespace (space, tab, newline), valid/invalid string, single/double-byte string, string in UTF-8 encoding

5.2.1.2 Additional coverage criteria

- **Path coverage**: Has every feasible path through the CFG of the given part of code been executed?
- **Entry/exit coverage**: Has every possible call and return of the function been executed?
- **Loop coverage**: Has every possible loop been executed for zero, once and multiple iterations?
- **Linear code sequence and jump (LCSAJ)** or **JJ-Path coverage**: software analysis method used to identify structural units in SUT

5.2.2 Performance

The performance of a test data generation method should be viewed in two different aspects, the first of which is the **effectiveness** of a test data generator - the fraction of test requirements covered by T', ignoring unreachable branches, occurring likely due to logical program errors.

The other aspect measured is the **efficiency** of a test data generation method in terms of its *space* and *runtime* complexity. Space efficiency is affected by the amount of information stored during the process of gathering test data for generation. In general, plenty of space is required for the static approach, especially symbolic execution-based methods, whereas the dynamic approach requires relatively less space in comparison, because of the difference between static analysis information and runtime data.

In contrast, the static approach is more economical than the dynamic approach as far as time requirements are concerned. While the dynamic approach needs many iterations, and a single execution of the SUT is the most significant computational cost, the solution can be derived quickly in a single pass in the static approach, neglecting the fact that obtaining a solution from the set of algebraic expressions could result in a complex computation.

5.2.3 Quality of test data

The quality of test data is related to how many faults are detected by T'. If a set of test data T_1' can reveal more faults in P than the other set of test data T_2' for a given M and C, T_1' is of a better quality than T_2'. Therefore, the quality of test data generated is measured by seeding errors into P via a mutation testing technique.

5.2.3.1 Mutation testing

Mutation testing is a fault-based method of software testing designed to create effective test data. It works by randomly modifying programs source code or bytecode in slight but critical ways. Any tests which pass after code has been mutated are considered defective, called as 'mutations'.

This procedure is established on well-defined mutation operators that either mimic typical software or human error, and supposedly leads to the creation of more valuable unit tests.

Its purpose is to evaluate the effectiveness of the automatic test data generation strategy, especially when it comes down to the 'weaker' parts of code that are seldom or never accessed during normal program execution, and so probably less extensively tested.

The basic steps involved here are as follows:

- begin with the test suite generated automatically, and the one written manually by hand.
- once verified that both pass on a given piece of program code, apply a mutation to the bytecode of this SUT.
- the extent of mutation can vary, from the very elementary substitution of a logical operator with its complement. For instance, == can be transformed into !=, while < would turn into >=.
- the more complex operations would be as drastic as reordering code execution or removing parts of code completely.
- however, mutations of this degree frequently cause compiler errors, defeating the purpose of this evaluation altogether, so it is often more advisable to perform the simpler mutations mentioned instead.
- after this, both the original test suites are re-run against this program.
- if the test suites were effective, they should now be expected to fail in covering the mutated program.
- otherwise the test is not well written, as it is creating false positives, and needs to be revisited.

- of course, a scoring metric can be invented to denote partial success in generating unit tests, should it be the case that only some of the tests pass on the mutant program.

The standard comparison operators, according to the Python 2.7 Grammar (Appendix **??**), are listed below to be used for demonstrating mutation testing:

- **<**: less than
- **>**: greater than
- **==**: compares value equality
- **>=**: greater than or equal to
- **<=**: less than or equal to
- **(<> | !=)**: not equal to
- **in**: checks element for membership in collection
- **is**: compares identity of two objects

5.2.4 Generality

The generality of a test data generation method indicates its ability to function in a wide selection and practical range of situations. Ideally, the test data generation method should function in the presence of arbitrarily complex programs.

The less the test data generation method is restricted by language constructs and target languages in which the program is written, the more generally applicable the test data generation method is, which is why this project sets out to generate unit tests in a language-neutral format like JSON.

The test data generation method should work on the complex program to be used in practice. The coverage rate in P by T' according to the complexity of each program needs to be examined. If a test data generator covers all target branches on the complex program, it is said to be generally applicable, which is the desired characteristic.

5.3 Selection of programs

Before preparation for experimentation, candidate programs have to be selected as unit test data generation targets. Besides calculating the basic Source Lines of Code (SLoC) statistic, the features considered are loops/Python generators, list/set/dict, conditionals, recursive calls, input data type - numeric: integer or floating point, or classes and objects, as well as the complexity of a program, referring to the cyclomatic complexity

metric, nesting (of conditionals) and condition complexity (number of boolean expressions within a conditional).

Cyclomatic complexity (CC) directly measures the number of linearly independent paths through program code, and is computing using its CFG. It is more precisely defined by the following equation:

$$M = E - N + 2P$$

where M is the cyclomatic complexity, E is the number of edges in the flowgraph, N the number of nodes of the graph, and P is the number of connected components (exit nodes).

5.3.1 Fingerprinting Python modules

A collection of trivial Python modules was put together with the following characteristics for the purpose of benchmarking as test targets:

Total Source Lines of Code (SLOC) = **71**

Cyclomatic complexity (CC)

Type	Count	Complexity
Files	7	8
Classes	3	3
Methods	8	9
Functions	6	17
Total	24	37

The **trivial** Python package consists of the following modules, and this paragraph briefly explains the purpose of each of these pieces of code. The **bank_account** Python module is a simple class-based model of a real life bank account, with the typical expected functions like `deposit()` and `withdraw()`. The **factorial** program is simply one that computes the factorial of a given input numeric argument. The **fib** program stands for "fibonacci", and recursively computes the fibonacci sum for a given input integer. The **triangle** program is the classic basic triangle test often used for benchmarking. Given the lengths of the three sides of an arbitrary triangle, it attempts to classify and return the type of triangle as a string, of one of the following values: {'notvalid', 'equilateral', 'isoceles', 'scalene'}.

5.3.2 Coverage results

The code coverage results, according to coverage.py, is tabulated below for each Python module within the trivial package:

	pyprimes 0.1.1a	pyutilib.math 3.3	quixey challenge
Source Lines of Code	385	132	187
Functions	38	14	17
Cyclomatic Complexity	147	43	80
Original	63%	63%	97%
Generated	54%	70%	91%

A sample graph depicting the cumulative increase in code coverage over iterations is shown below:

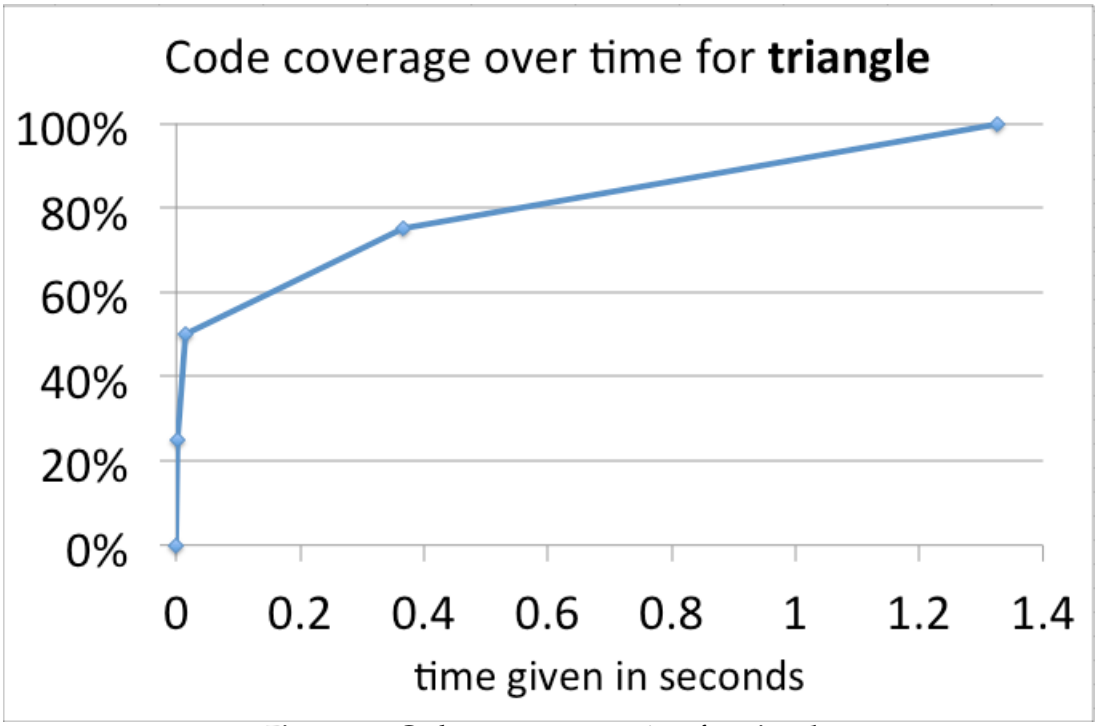


Figure 5.1: Code coverage over time for **triangle**

5.3.3 Discussion

The results gathered, though inconclusive because of the small sample size, yet are indeed very encouraging and promise greater potential for future work to be done in order to truly investigate the full extent of effectiveness in using this methodology to automatically generate unit tests in Python.

However, it is worthy to note that even if the high code coverage attained here could be extended to a wider range of Python software, this is still probably insufficient, unless a mixture of the different kinds of code coverage criteria like Parameter value coverage or LCSAJ is measured.

For example, even though a test case for function accepting a single numeric argument exercises the entire function's code paths, it does not automatically mean that the function is bug-free for any value in that numeric range. The wisdom of which of this criteria most appropriate to the intended use case, for example to test safety critical systems, to select also becomes a significant factor.

As expected, the dynamic method of testing certainly incurs a fair bit of space and runtime cost for being rather effective at automatically generating unit tests, to store data of the execution history and for the many iterative refinements to go through.

Compared to the existing Python tools, i.e. Pythoscope, PyTestsGenerator, this project trivially outperforms those tools in every test code coverage criteria, as those tools do not automatically discover unit tests (even though they claim to do so), whereas this tool does.

There were two other larger Python packages originally intended for benchmark - Python graph library pygraph and a collection of famous algorithms from Quixey Challenge, but due to limited resources, meaningful results could not be obtained for these benchmarks, hence they are not reflected here.

This is so, since not all the `TypeError` cases were handled, neither the bytecode versions of the programming constructs found in Appendix **??**, although at least a sample of each technique was included in this project's tool as a prototype.

Also, only the groundwork for the mutation testing explained previously managed to be carried out, i.e. being able to track the execution of a modified fragment of Python bytecode. The concept was to follow that as specified, by inverting boolean operators in conditional expression within the Python bytecode. Succeeding in doing this will ascertain the quality of the test data generated.

5.4 Summary

The results achieved and comparison of desired outcomes with original expectations are examined here in this chapter. To finish off, the next chapter summarises this entire paper and hints at what might be possible future research directions in this area of automated software testing.

Chapter 6

Conclusion & Future Work

In this study, the rationale behind the inspiration for this project was first introduced. Consequently, relevant background information was provided to set the stage for the rest of the project. This was made up of everything from the fundamentals, i.e. basic graph theory applied to software testing, overview of the general field of software testing, evaluation of existing unit test generation tools, intrinsics and subtleties of the Python programming language resulting in the challenges faced by this project. Following which, the theoretical foundations of this project were highlighted, all culminating in the description of the SPLAT tool implemented by this project. Lastly, the performance of this tool was evaluated and discussed in relation to various test coverage criteria.

Unfortunately, this paper is by no means a standalone, and there still remains much material to explore further. Some guiding suggestions for its future direction may include: exposing an API for other developers to contribute to the development of this tool, comprehensive documentation on the internal workings and usage of the tool, with examples provided as well, or improve sophistication of the tool to generate more robust and thorough tests, test code containing more complex interaction of language constructs, or deal with new language features in Py3k.

The efficiency of the tool in test generation can be optimised by compiling on a faster Python implementation, or coming up with some way of load division that encourages parallelism. Moreover, deeper analysis of the tool can be achieved by conducting benchmarks across a wider range of different Python frameworks and libraries, exploring other techniques and algorithms to attempt to improve overall test code coverage, and by extending the bytecode parser to automatically generate unit tests for all the scenarios found in Appendix **??**.

Appendix A

Python 2.7 Grammar (EBNF)

```
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER
decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef)
funcdef: 'def' NAME parameters ':' suite
parameters: '(' [varargslist] ')'
varargslist: ((fpdef ['=' test] ',')*
              ('*' NAME [',' '**' NAME] | '**' NAME) |
              fpdef ['=' test] (',' fpdef ['=' test])* [','])
fpdef: NAME | '(' fplist ')'
fplist: fpdef (',' fpdef)* [',']
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | print_stmt  | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | exec_stmt | assert_stmt)
expr_stmt: testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
print_stmt: 'print' ( [ test (',' test)* [','] ] |
                      '>>' test [ (',' test)+ [','] ] )
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test [',' test [',' test]]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
import_from: ('from' ('.'* dotted_name | '.'+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
```

```
exec_stmt: 'exec' expr ['in' test [',' test]]
assert_stmt: 'assert' test [',' test]
compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef
     | classdef | decorated
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
except_clause: 'except' [test [('as' | ',') test]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
testlist_safe: old_test [(',' old_test)+ [',']]
old_test: or_test | old_lambdef
old_lambdef: 'lambda' [varargslist] ':' old_test
test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom trailer* ['**' factor]
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [listmaker] ']' |
       '{' [dictorsetmaker] '}' |
       '`' testlist1 '`' |
       NAME | NUMBER | STRING+)
listmaker: test ( list_for | (',' test)* [','] )
testlist_comp: test ( comp_for | (',' test)* [','] )
lambdef: 'lambda' [varargslist] ':' test
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: '.' '.' '.' | test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: expr (',' expr)* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [','])) |
                  (test (comp_for | (',' test)* [',')) )
```

```
classdef: 'class' NAME ['(' [testlist] ')'] ':' suite
arglist: (argument ',')* (argument [',']
                         |'*' test (',' argument)* [',' '**' test]
                         |'**' test)
argument: test [comp_for] | test '=' test
list_iter: list_for | list_if
list_for: 'for' exprlist 'in' testlist_safe [list_iter]
list_if: 'if' old_test [list_iter]
comp_iter: comp_for | comp_if
comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_if: 'if' old_test [comp_iter]
testlist1: test (',' test)*
yield_expr: 'yield' [testlist]
```

Appendix B

Type hierarchy

The standard list of built-in types is:

1. **None**

   Singleton

   signifies the absence of a value

2. **NotImplemented**

   Singleton

   return this if the operation for operands provided is not implemented

3. **Ellipsis**

   Singleton

   indicates presence of '...' in a slice

4. **numbers.Number**

   numbers.Integral: Plain, Long, Boolean

   numbers.Real(Float): represents machine-level double precision floating point numbers

   numbers.Complex: pair of doubles (real, imaginary components)

5. **Sequence**

   finite ordered sets indexed by non-negative numbers

   responds to `len()`, returning number of items in sequence

   supports slicing

   examples of immutable sequences include `str`, `unicode`, `tuple`

   whereas mutable sequences are `list`, `bytearray`

6. **Set**

   unordered, finite set of unique immutable objects

   responds to `len()`, returning number of items in set

no indexing operations supported

mutable version is `set()`, immutable `frozenset()`

7. **Mapping**

   `dict`

   mutable

   finite set of objects

   indexed by nearly arbitrary keys

   except those values containing `list`/`dict`/other mutable types

8. **Callable type**

   User-defined function

   created by a function definition

   called with an argument list as long as the function's formal parameter list

   support getting and setting custom attributes, which can be used for attaching metadata to functions, for example, using regular dot-notation

   User-defined method

   combination of a class, class instance (or `None`), and any callable

   Generator function

   a function or method which contains the **`yield`** keyword

   always returns an iterator object

   providing the `next()` value

   `StopIteration` exception is raised upon reaching the end

   Built-in functions

   wrapper around a C function, like `len()`, `math.sin()`

   Built-in methods

   a built-in function, with an additional object implicitly passed to the C function as an extra argument

   Class

   created by a class definition

   contains a namespace implemented by a dictionary object

attribute references are translated to lookups in this dictionary

attribute assignments update the module's namespace dictionary

There are classic and new-style classes. New-style classes were introduced in Python 2.2 to unify classes and types. A new-style class is but just a user-defined type. The major motivation for this change is to provide a unified object model with a full meta-model. The method resolution order for old-style classes first searches for the attribute in the class itself, then continues in the base classes in a depth-first, left-to-right order, according to declaration of base class list. For new-style classes, this is detailed in the *C3 MRO*. More in-depth details regarding customising classes by overriding special method names can be found in the official Python 2.7.x documentation.

Class instance

created by calling a class object

9. **File**

10. **Module**

contains a namespace implemented by a dictionary object

attribute references and assignments treated the same way as objects

does not contain the code object used to initialise the module

11. **Internal type**

Code object

represent byte-compiled executable Python code, otherwise known as bytecode

immutable

contain no references to mutable objects

contains no context, unlike a function object, which has an explicit reference to its globals (module in which it was defined)

does not contain default argument values

Frame object

represent execution frames

may occur in traceback objects (see next item)

Traceback object

represent stack trace of an exception

created when an exception occurs, and search for an exception handler begins

one traceback object per level of the execution stack

Slice object

represent slices when *extended slice syntax* is used, i.e. using two colons, or multiple slices or ellipses separated by commas

Static method object

wrapper around another (usually) user-defined method object

created by built-in `staticmethod()` constructor

Class method object

wrapper around another object, altering retrieval process from classes and class instances

created by built-in `classmethod()` constructor

Appendix C

Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError...
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      ...
```

Appendix D

Python objects

D.1 frame object

frame	f_back	next outer frame object (this frame's caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_exc_traceback	traceback if raised in this frame, or None
	f_exc_type	exception type if raised in this frame, or None
	f_exc_value	exception value if raised in this frame, or None
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_restricted	0 or 1 if frame is in restricted execution mode
	f_trace	tracing function for this frame, or None

D.2 code object

code	co_argcount	number of arguments (not including * or ** args)
	co_cellvars	tuple of the names of local variables referenced by nested functions
	co_code	string of raw compiled sequence of bytecode instructions
	co_consts	tuple of literal constants used in the bytecode
	co_filename	name of file in which this code object was created
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap: 1=optimized | 2=newlocals | 4=*varg | 8=**kwarg | 20=generator
	co_freevars	tuple containing the names of free variables
	co_lnotab	encoded mapping of line numbers to bytecode indices
	co_name	name with which this code object was defined
	co_names	tuple of names of local variables used by bytecode
	co_nlocals	number of arguments and local variables used by the function
	co_stacksize	virtual machine stack space required
	co_varnames	tuple containing argument names, then names of the local variables

Appendix E

Python 2.7 opcodes

```
def_op('STOP_CODE', 0)
def_op('POP_TOP', 1)
def_op('ROT_TWO', 2)
def_op('ROT_THREE', 3)
def_op('DUP_TOP', 4)
def_op('ROT_FOUR', 5)
def_op('NOP', 9)
def_op('UNARY_POSITIVE', 10)
def_op('UNARY_NEGATIVE', 11)
def_op('UNARY_NOT', 12)
def_op('UNARY_CONVERT', 13)
def_op('UNARY_INVERT', 15)
def_op('BINARY_POWER', 19)
def_op('BINARY_MULTIPLY', 20)
def_op('BINARY_DIVIDE', 21)
def_op('BINARY_MODULO', 22)
def_op('BINARY_ADD', 23)
def_op('BINARY_SUBTRACT', 24)
def_op('BINARY_SUBSCR', 25)
def_op('BINARY_FLOOR_DIVIDE', 26)
def_op('BINARY_TRUE_DIVIDE', 27)
def_op('INPLACE_FLOOR_DIVIDE', 28)
def_op('INPLACE_TRUE_DIVIDE', 29)
def_op('SLICE+0', 30)
def_op('SLICE+1', 31)
def_op('SLICE+2', 32)
def_op('SLICE+3', 33)
def_op('STORE_SLICE+0', 40)
def_op('STORE_SLICE+1', 41)
def_op('STORE_SLICE+2', 42)
def_op('STORE_SLICE+3', 43)
def_op('DELETE_SLICE+0', 50)
def_op('DELETE_SLICE+1', 51)
def_op('DELETE_SLICE+2', 52)
def_op('DELETE_SLICE+3', 53)
def_op('STORE_MAP', 54)
def_op('INPLACE_ADD', 55)
def_op('INPLACE_SUBTRACT', 56)
def_op('INPLACE_MULTIPLY', 57)
def_op('INPLACE_DIVIDE', 58)
def_op('INPLACE_MODULO', 59)
def_op('STORE_SUBSCR', 60)
def_op('DELETE_SUBSCR', 61)
def_op('BINARY_LSHIFT', 62)
def_op('BINARY_RSHIFT', 63)
def_op('BINARY_AND', 64)
def_op('BINARY_XOR', 65)
def_op('BINARY_OR', 66)
def_op('INPLACE_POWER', 67)
def_op('GET_ITER', 68)
def_op('PRINT_EXPR', 70)
def_op('PRINT_ITEM', 71)
def_op('PRINT_NEWLINE', 72)
def_op('PRINT_ITEM_TO', 73)
def_op('PRINT_NEWLINE_TO', 74)
def_op('INPLACE_LSHIFT', 75)
def_op('INPLACE_RSHIFT', 76)
def_op('INPLACE_AND', 77)
def_op('INPLACE_XOR', 78)
def_op('INPLACE_OR', 79)
def_op('BREAK_LOOP', 80)
def_op('WITH_CLEANUP', 81)
def_op('LOAD_LOCALS', 82)
def_op('RETURN_VALUE', 83)
def_op('IMPORT_STAR', 84)
def_op('EXEC_STMT', 85)
def_op('YIELD_VALUE', 86)
def_op('POP_BLOCK', 87)
def_op('END_FINALLY', 88)
def_op('BUILD_CLASS', 89)
HAVE_ARGUMENT = 90
name_op('STORE_NAME', 90)
name_op('DELETE_NAME', 91)
def_op('UNPACK_SEQUENCE', 92)
jrel_op('FOR_ITER', 93)
def_op('LIST_APPEND', 94)
name_op('STORE_ATTR', 95)
name_op('DELETE_ATTR', 96)
name_op('STORE_GLOBAL', 97)
name_op('DELETE_GLOBAL', 98)
```

```
def_op('DUP_TOPX', 99)
def_op('LOAD_CONST', 100)
hasconst.append(100)
name_op('LOAD_NAME', 101)
def_op('BUILD_TUPLE', 102)
def_op('BUILD_LIST', 103)
def_op('BUILD_SET', 104)
def_op('BUILD_MAP', 105)
name_op('LOAD_ATTR', 106)
def_op('COMPARE_OP', 107)
hascompare.append(107)
name_op('IMPORT_NAME', 108)
name_op('IMPORT_FROM', 109)
jrel_op('JUMP_FORWARD', 110)
jabs_op('JUMP_IF_FALSE_OR_POP', 111)
jabs_op('JUMP_IF_TRUE_OR_POP', 112)
jabs_op('JUMP_ABSOLUTE', 113)
jabs_op('POP_JUMP_IF_FALSE', 114)
jabs_op('POP_JUMP_IF_TRUE', 115)
name_op('LOAD_GLOBAL', 116)
jabs_op('CONTINUE_LOOP', 119)
jrel_op('SETUP_LOOP', 120)
jrel_op('SETUP_EXCEPT', 121)
jrel_op('SETUP_FINALLY', 122)
def_op('LOAD_FAST', 124)
haslocal.append(124)
def_op('STORE_FAST', 125)
haslocal.append(125)
def_op('DELETE_FAST', 126)
haslocal.append(126)
def_op('RAISE_VARARGS', 130)
def_op('CALL_FUNCTION', 131)
def_op('MAKE_FUNCTION', 132)
def_op('BUILD_SLICE', 133)
def_op('MAKE_CLOSURE', 134)
def_op('LOAD_CLOSURE', 135)
hasfree.append(135)
def_op('LOAD_DEREF', 136)
hasfree.append(136)
def_op('STORE_DEREF', 137)
hasfree.append(137)
def_op('CALL_FUNCTION_VAR', 140)
def_op('CALL_FUNCTION_KW', 141)
def_op('CALL_FUNCTION_VAR_KW', 142)
jrel_op('SETUP_WITH', 143)
def_op('EXTENDED_ARG', 145)
EXTENDED_ARG = 145
def_op('SET_ADD', 146)
def_op('MAP_ADD', 147)
```

Appendix F

Source code to Bytecode mapping

Source code fragment	Equivalent Bytecode (byteplay)

class <classname>(object):
  pass
	[
	  (LOAD_CONST, <classname>),
	  (LOAD_GLOBAL, 'object'),
	  (BUILD_TUPLE, 1),
	  (LOAD_CONST, <byteplay.Code>),
	  (MAKE_FUNCTION, 0),
	  (CALL_FUNCTION, 0),
	  (BUILD_CLASS, None),
	]

destructured assignment:
  a, b = 1, 0
	[
	  (LOAD_CONST, (1,0)),
	  (UNPACK_SEQ, 2),
	  (STORE_FAST, 'a'),
	  (STORE_FAST, 'b'),
	]

a <op>= b	[
	  ...
	  (INPLACE_*, None),
	]

del <var>	[(DELETE_FAST, <var>),]

lambda function definition:
  lambda _: None
	[
	  (LOAD_CONST, <byteplay.Code object>),
	  (MAKE_FUNCTION, 0),
	]

function invocation:
  <function symbol>(#n arguments)
	[
	  (LOAD_GLOBAL, <function symbol>),
	  ...loading n arguments...
	  (CALL_FUNCTION, n),
	]

tuples:
  (<constant>, <var>,)
	[
	  (LOAD_CONST, <constant>),
	  (LOAD_GLOBAL, <var>),
	  (BUILD_TUPLE, 2),
	]

Source code fragment	Equivalent Bytecode (byteplay)

`list() / set()`	`[`
	`  ...load n elements L-to-R order...`
	`  (BUILD_(LIST|SET), n),`
	`]`

`mapping: dict()`	`[`
	`  (BUILD_MAP, n),`
	`  ...load n elements R-to-L order...`
	`]`

`extended slicing syntax:`
`  x[0]`
`  x[:-1]`
`  x[0:-1]`
`  x[:0:-1]`
	`[`
	`  (BINARY_SUBSCR, None),`
	`  (SLICE_2, None),`
	`  (SLICE_3, None),`
	`  (BUILD_SLICE, 3),`
	`  (BINARY_SUBSCR, None),`
	`]`

`with <1> as <2>:`
`  <3>`
`<4>`
	`[`
	`  <1>,`
	`  (SETUP_WITH, <byteplay.Label object>),`
	`  (STORE_FAST, <2>),`
	`  <3>,`
	`  (POP_BLOCK, None),`
	`  (LOAD_CONST, None),`
	`  (WITH_CLEANUP, None),`
	`  (END_FINALLY, None)`
	`  <4>,`
	`]`

`raise <Exception>('<message>')`	`[`
	`  (LOAD_GLOBAL, <Exception>),`
	`  (LOAD_CONST, <message>),`
	`  (CALL_FUNCTION, 1),`
	`  (RAISE_VARARGS, 1),`
	`]`

`generator's 'yield' keyword`	`[`
	`  ... setup value preamble...`
	`  (YIELD_VALUE, None),`
	`]`

Source code fragment	Equivalent Bytecode (byteplay)

`if <1>:`
`  <1a>`
`[elif <2>:`
`  <2a>]*`
`else:`
`  <3>`
`<4>`
	`<cond_jmp> = POP_JUMP_IF_{TRUE|FALSE}|JUMP_IF_{TRUE|FALSE}_OR_POP`
	`[`
	`  <1>,`
	`  (<cond_jmp>, <byteplay.Label#1 object>),`
	`  <1a>,`
	`  (<byteplay.Label#1 object>, None),`
	`  <2>,`
	`  (<cond_jmp>, <byteplay.Label#2 object>),`
	`  <2a>,`
	`  (<byteplay.Label#2 object>, None),`
	`  <3>,`
	`  (<byteplay.Label#3 object>, None),`
	`  <4>,`
	`]`

`while <1>:`
`  <2>`
`else:`
`  <3>`
`<4>`
	`<cond_jmp> = POP_JUMP_IF_{TRUE|FALSE}|JUMP_IF_{TRUE|FALSE}_OR_POP`
	`[`
	`  (SETUP_LOOP, <byteplay.Label#1 object>),`
	`  (<byteplay.Label#2 object>, None),`
	`  <1>,`
	`  (<cond_jmp, <byteplay.Label#3 object>),`
	`  <2>,`
	`  (<byteplay.Label#3 object>, None),`
	`  (POP_BLOCK, None),`
	`  <3>,`
	`  (<byteplay.Label#1 object>, None),`
	`  <4>,`
	`  (BREAK_LOOP, None),`
	`  (JUMP_ABSOLUTE, <byteplay.Label#2 object>),`
	`]`

`for <i> in <1>:`
`  <2>`
`else:`
`  <3>`
`<4>`
	`[`
	`  (SETUP_LOOP, <byteplay.Label#1 object>),`
	`  <1>,`
	`  (GET_ITER, None),`
	`  (<byteplay.Label#2 object>, None),`
	`  (FOR_ITER, <byteplay.Label#3 object>),`
	`  <2>,`
	`  (STORE_FAST, <i>), - 1 per loop variable`
	`  (JUMP_ABSOLUTE, <byteplay.Label#2 object>),`
	`  (<byteplay.Label#3 object>, None),`
	`  (POP_BLOCK, None),`
	`  <3>,`
	`  (<byteplay.Label#1 object>, None),`
	`  <4>,`
	`]`

Source code fragment	Equivalent Bytecode (byteplay)

try:
  <1>
except (Exception,...) as (e,...):
  <2>
else:
  # no Exception raised
  <3>
finally:
  # always executed
  <4>
<5>
	[
	  (SETUP_FINALLY, <byteplay.Label#1 object>),
	  (SETUP_EXCEPT, <byteplay.Label#2 object>),
	  <1>,
	  (POP_BLOCK, None),
	  (JUMP_FORWARD, <byteplay.Label#3 object>),
	  (<byteplay.Label#2 object>, None),
	  (DUP_TOP, None),
	  (LOAD_GLOBAL, <Exception),
	  ...for n such exceptions...
	  (BUILD_TUPLE, n),
	  (COMPARE_OP, 'exception match'),
	  (<cond_jmp>, <byteplay.Label#4 object>),
	  (POP_TOP, None),
	  (UNPACK_SEQUENCE, m),
	  (STORE_FAST, 'e')...m times...
	  (POP_TOP, None),
	  <2>,
	  (<byteplay.Label#4 object>, None),
	  (END_FINALLY, None),
	  (<byteplay.Label#3 object>, None),
	  <3>,
	  (POP_BLOCK, None),
	  (LOAD_CONST, None),
	  (<byteplay.Label#1 object>, None),
	  <4>,
	  (END_FINALLY, None),
	  <5>,
	]