Capture and analysis of side effects in a running Python program for the purpose of unit test generation

Michał Kwiatkowski

June 15, 2011

Abstract

Research on prospects of test generation for programs with side effects. Adding side effects tracing capabilities to Pythoscope and implementing a method of using acquired information during test generation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Unit test generation for legacy systems

Unit testing, when properly practiced, has been shown to decrease number of defects [25], prevent regressions and provide a safety net for experimentation[1], making the task of software development and maintenance easier. On the other hand unit testing, as any other development activity, takes time, thus increasing overall cost of making software. While there is no hard data on unit testing ROI (*Return On Investment*) there most certainly is a point where time spent on unit testing is wasted, i.e. unit testing adds no additional value to the project. Moreover, writing unit tests for *legacy systems*[2] has a very high cost and only long-term benefits. For those reasons anything, a practice or a tool, that helps reduce the time spent writing and maintaining unit tests has a potential of giving unit testing a satisfying ROI value.

One of the solutions to this problem is an *automatic test generation*. Model-based testing [6] derives tests from a model of the system, usually built even before the development process starts. Tests can also be generated from application source code or directly from recorded interactions of a running system with the user[3]. Recorded tests or tests generated directly from source code are called *characterization tests*, because they characterize the actual behavior of the system[4]. While useful in detecting regressions those tests can't validate the correct (or intended) behavior of the SUT (*System Under Test*). Test generation is useful in improving testing ROI, but cannot be thought of as a substitute for good software testing practices.

[1]See *Tests as Safety Net* in [17], page 24.

[2]For a definition of a legacy system see section 2.4 on page 15.

[3]More on this topic can be found in [17] in a section about *Recorded Test* pattern.

[4]More on characterization tests in section 2.5 on page 15.

1.2 Side effects

One of the aspects of writing good unit tests is ensuring isolation. Gerard Meszaros calls it the *Isolate the SUT* principle [17]:

> Whatever application, component, class, or method we are testing, we should strive to isolate it as much as possible from all other parts of the software that we choose not to test.

More often than not making perfectly isolated tests is hard. Application code can depend on a lot of things: other functions, methods, classes, modules, libraries, configuration files, databases, network, etc. Anything that is not an explicit SUT input can be thought of as an *external component*, and all operations on those components as *side effects*[5]. Every dependency needs to be accounted for during testing: classes initialized, modules imported, libraries loaded, configuration files placed under the right paths and with the right content, and so on. Problem of side effects for test generation is twofold. First, side effects need to be recognized in the application and analyzed. Only after that test cases with proper handling of those side effects can be generated. Capture and analysis of side effects and generation of unit tests from gathered data is the main topic of this thesis.

1.3 Technology involved

Python is a dynamic programming language with object oriented as well as functional capabilities, with a distinctive readable syntax and focus on imperative style of programming. Types of objects are settled on runtime, meaning static analysis of Python programs is less reliable and typically will yield less information than it could in case of languages with static typing like Java or C++. All of this makes analysis and test generation for Python programs a challenge, but hopefully techniques presented in this work can be transmitted to other dynamic languages, like Ruby or Perl.

There is a comprehensive set of testing-related tools for Python, including:

1. test runners: nose, py.test, zope.testing
2. test double libraries: mock, python-mock, PyMock
3. code coverage tools: coverage, figleaf
4. debuggers: CodeInvestigator, pdb

[5]For a stricter definition of a side effect refer to chapter 3 on page 18.

5. code metrics and analysis tools: pylint, pychecker, pycallgraph, pymetrics

but only one actively maintained unit test generator: Pythoscope[6]. First version of Pythoscope was developed by Michał Kwiatkowski with the support of Walt Disney Animation Studios and released in September 2008 on an open source license. The latest version of Pythoscope before this thesis supported both static and dynamic analysis of Python programs and was capable of generating unit test stubs as well as complete test cases of pure (meaning free of side effect) code. In this thesis, all implementation work will be based on Pythoscope - side-effects-aware extensions will be built on top of Pythoscope's codebase.

1.4 Characterization testing with Pythoscope

All function and method calls in Python can be described in a purely functional manner, as a mathematical construct: transformation of input into output.

$$f : I \to O$$

Domain of I and codomain of O can be further divided up into explicit and implicit. Function's arguments (A) constitute an explicit input while its return value (R) or raised exception (E) is an explicit output. State of the external entities (S) constitute an implicit input, while any changes made by the function to those external entities is an implicit output, or - as previously defined - *side effects* (S').

$$f : (A, S) \to (R, E, S')$$

First step in automatic test generation is capturing all calls, with complete inputs and outputs. Figure 1.1 illustrates call as a transformation. In gray have been marked subdomains that a version of Pythoscope prior to this thesis has been blind to. One of the goals of this thesis was implementing a method of capturing all inputs and outputs of a function, including the implicit ones.

[6]For a more comprehensive list of testing tools see Python testing tool taxonomy [43].

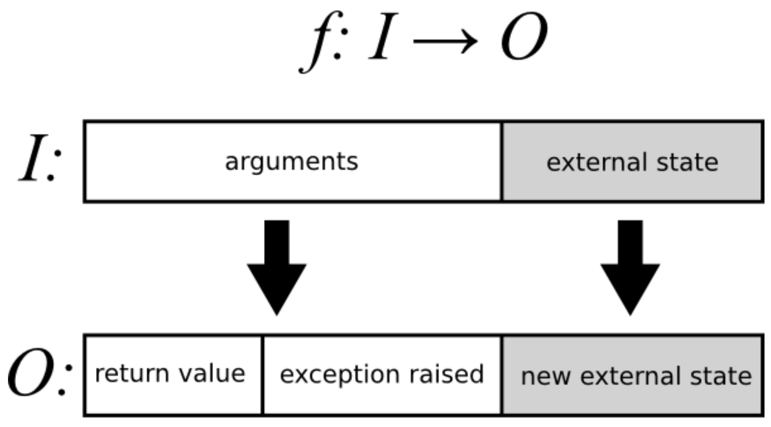


Figure 1.1: Call as a transformation

Goal of *characterization testing* is to assert that

$$\forall i \in \mathbf{I} : f(i) = f'(i)$$

where f' is a new (modified) version of f (also after refactoring [9]). Since it is impossible to check all values from the domain, a representative subset must be chosen to form test cases from. Moreover, it would be best to choose a *minimal* subset, so that a number of test cases correlate with complexity of the function being tested. This problem of picking out test cases from an infinite set of possible transformations has not been resolved in Pythoscope. Instead, Pythoscope simply converts all transformations captured during dynamic analysis into test cases.

The final problem of automatic generation of test cases has to do with test dependencies. Each test case has to reconstruct the input needed to exercise the function. Arguments need to be created and state of the world prepared. Pythoscope has to resolve all those dependencies when creating a test case and that requires knowledge about connections between objects and the external world. The problem gets even harder when test readability is of concern[7].

It is important to note that Pythoscope is not a fully automatic solution. To provide a starting point for dynamic analysis, the user has to do some setup work first.[8]

[7]Refer to section 5.10 on page 53 for details.
[8]Refer to 6.4.1 on page 66 for details.

1.5 Thesis goal and contributions

The main purpose of this thesis is to implement and describe modifications that were needed to make Pythoscope understand and generate tests aware of side effects.

There are four main contributions of this thesis. One of them is about theory while the other three deal with the implementation. Since all implementation work has been based on Pythoscope, it is necessary to provide a little bit of information about Pythoscope first.

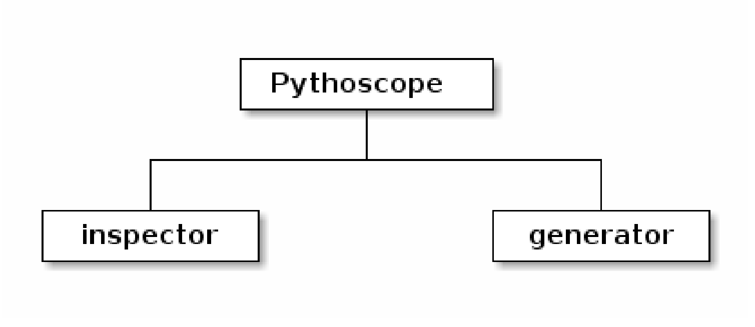


Figure 1.2: Two main parts of Pythoscope

In the most simplified view, Pythoscope consists of two parts: the inspector and the generator. The inspector is responsible for gathering information about the application. Inspector may analyze the static structure of a program, and it may also run the program to analyze its dynamic behavior. The generator is responsible for using data gathered by the inspector to generate unit test cases that verify the behavior of a program. Pythoscope project as a whole forms a combination of well researched areas, such as program profiling techniques [11] used by the inspector or compiler design [1] applied to the generator, and innovative application - unit test generation for a modern dynamic programming language. The pragmatic setting of the Python language makes the implementation non-trivial and thus deserve detailed treatment. More detailed description of Pythoscope's structure can be found in chapter 6.

1.5.1 Bytecode tracer for Python

One of the main shortcomings of Pythoscope's inspector prior to this thesis was its inability to trace bytecodes and calls into functions implemented in C programming language. The old inspector could only trace Python function calls. Bytecode tracing is necessary for tracing all kind of side effects, so the first contribution of this thesis was implementing a bytecode tracer for Python and merging it into Pythoscope. Bytecode tracer has been described in detail in chapter 4.

1.5.2 Side effects extension to Pythoscope's inspector

Once the bytecode tracer has been merged into Pythoscope it was necessary to implement a rule-based system inside the inspector, so that it could recognize different types of side effects. The rule-based system itself as well as a description of different sources of side effects can be found in chapter 4. Types of side effects has been described in chapter 3.

1.5.3 Side effects extension to Pythoscope's test generator

Just as the inspector had to be modified to recognize side effects the test generator also had to be altered to generate valid test cases that not only replayed side effects but also validate their appearance. New version of Pythoscope is able to prepare the external environment before the test case and then check for any side effects that occurred when exercising the SUT, as already described in section 1.4. Test generation process has been described in chapter 5.

1.5.4 A model of analysis for Python side effects

The last objective of this thesis was to describe types of side effects occurring in Python programs and present a model of capture and analysis of those side effects in the context of unit test generation.

1.6 Structure of this thesis

This thesis has been divided into seven chapters. First chapter is an introduction to the problem of test generation and the approach taken by the author to improve the state of the art. Chapter 2 describes basic definitions needed to understand rest of this work. Chapter 3 categorizes and describes different kinds of side effects. Chapter 4 presents a model for capture and analysis of side effects and describes its implementation in the Pythoscope project. Chapter 5 moves unto a topic of generating unit tests for applications with side effects. A model is followed by overview of the implementation, based on the Pythoscope project. Chapter 6 delves into more detail of Pythoscope architecture. Results of this thesis are summarized in the last chapter, where quality of generated test cases is measured based on a sample of open source applications written in Python.

Chapter 2

Unit testing: definitions

This chapter contains definitions that form a base for the rest of this work and are used throughout the whole thesis. References are listed for further reading. Other definitions are introduced when needed.

2.1 System under test

Concept central to the idea of unit testing is a **system under test** (SUT), that Gerard Meszaros defines as follows [17]:

> Whatever thing we are testing. The SUT is always defined from the perspective of the test. When we are writing unit tests, the SUT is whatever class (also known as CUT), object (also known as OUT), or method (also known as MUT) we are testing; when we are writing customer tests, the SUT is probably the entire application (also known as AUT) or at least a major subsystem of it. The parts of the application that we are not verifying in this particular test may still be involved as a depended-on component (DOC).

2.2 Unit test

In the context of this work, **a unit test** will be taken to mean **an automated developer test exercising a single unit of the SUT in isolation**. Each part of this definition is described in detail below.

Automation refers to the way tests are executed. Manual tests require a human to run, while automated tests can be run without any supervision. Automated tests can be much faster and precise than manual ones, but programming complex setup, interactions and checks can be often problematic.[1]

Developer tests are tests used during development. They are written, managed and run by developers.

A unit is a loosely defined term, and it depends on two things: the level of detail a developer wants and programming paradigm used in the application (that can be limited by the choice of a programming language). A unit may refer to a function, procedure, method, class or even a module or a package. Unit tests are not suitable for reading by a customer, as they usually don't map very well to high-level requirements.

Isolation means testing a component without its dependencies. Isolation is key when trying to achieve bug localization (see below). Dependencies are usually replaced with test doubles (see section 2.3).

Michael Feathers lists two qualities of a good unit test [7]:

1. short execution time
2. easy bug localization

Execution time is relevant because of the sheer number of unit tests in a project: they can often go into hundreds and thousands[2]. Feathers draws a line on 1/10th of a second - if a unit test takes longer than that it is considered slow.

Second feature of a good unit test is quick and easy bug localization. Failure of a test case should point directly at its cause. It should be done in such a way that when a developer gets a test failure, she can almost instantly interpret its meaning and localize error in code that caused the test to fail. If SUT is sufficiently covered with unit tests it creates a very tight feedback loop, which aids developers every time a code change is required.

2.3 Test double

The term **test double** was introduced by Gerard Meszaros in [17] as a generic term for all kinds of specialized objects used during testing as substitutes of real objects. Meszaros defines test double as follows:

[1]For more examples of manual and automated testing scenarios see [2].

[2]Pythoscope, which is a middle-sized Python project with over 3k LOC has over 400 test cases. CPython standard library in version 2.7 has over 1300 test cases, although not all of them are unit tests.

A Test Double is any object or component that we install in place of the real component for the express purpose of running a test.

Types of test doubles include dummies, stubs, mocks and fakes. They are used to test components in isolation of other components as well as external state like databases, network, file system, etc. More theory and practice regarding test doubles can be found in [17] and [8]. Changes to the external state in context of unit testing are covered in chapter 3.

2.4 Legacy system

Michael Feathers defines [7] **legacy system** simply as code without tests and that definition will be used in this thesis. He justifies by stating that without tests it is not possible to know whether any given code change is beneficial or detrimental. With unit tests in place feedback is immediate and clear, while the lack of tests inhibits change. At the same time it is hard to add tests to a legacy system, because application not constructed with testing in mind will need to be refactored to be testable. Michael Feathers calls that *The Legacy Code Dilemma*:

When we change code, we should have tests in place. To put tests in place, we often have to change code.

This chicken and egg problem makes the task of maintaining legacy systems very hard. Main purpose of test generators like Pythoscope is to get the project over the initial hump, when there are no tests at all.

2.5 Characterization test

Characterization test is a test that describes and checks for the actual behavior of the SUT. It's different from a **regression test** in that the latter verify the correct (or intended) behavior of the SUT, usually derived from requirements and design documents. Characterization tests can be constructed when no documentation is available, just by means of interacting with the SUT. The main purpose of characterization tests is not to find bugs, but to detect change. A developer may write a set of characterization tests to act as a safety net during refactoring. Those tests will help her preserve behavior of the system during development.

2.6 Test coverage

Test coverage is a measure of completeness of a test suite, and it forms a basis for strategy of *white-box testing*. Glenford J. Myers sums up main goal of white-box testing in “The Art of Software Testing” [19]:

> White-box testing is concerned with the degree to which test cases exercise or cover the logic (source code) of the program.

There are a couple of methods of measuring test coverage.

Statement (line) coverage measures number of statements (or lines) that were executed at least once during the testing process. This is a very weak criterion for white-box testing, yet the most easy to verify. When counting lines of code, empty lines and comments are usually ignored.

Decision (branch) coverage puts focus on decision statements, like `if-else` and `while`. A branch is covered when during a test run it had both a true and a false outcome at least once.

Somewhat more detailed is **condition coverage**, as it focuses on individual conditions in a decision statements. For example, the following snippet of Python code has one decision (branch) composed of two conditions:

```python
if a > 0 and b == 2:
    ...
```

Both `a > 0` and `b == 2` are conditions forming a single decision (branch) point. It can be said that a condition is covered when it takes all possible outcomes at least once during the whole test run.

It is worth noting that it is possible to achieve condition coverage without branch coverage. Using the example above, the following two test cases will achieve full coverage for both conditions, but will fail to cover the branch itself:

1. a=0, b=2
2. a=1, b=1

To overcome this flaw, both types of coverage can be combined. Full **decision-condition coverage** can be achieved when each condition in a decision takes on all possible outcomes at least once, and each of those decisions also take on all possible outcomes at least once during a test run.

There are cases when decision-condition coverage is not enough to sufficiently test the SUT and they all have to do with the way `and` and `or` operators are implemented. Consider the following two test cases, still using the same code example:

1. a=0, b=1
2. a=1, b=2

Theoretically they fully cover both conditions and the decision, but in reality we could argue that the test cases don't really cover all conditions completely. It's because to the interpreter a combination of two conditions with an `and` really correspond to the following code:

```
if a > 0:
    if b == 2:
        ...
    ...
```

With this definition in mind, two test cases mentioned above don't cover all conditions: `b == 2` is never false, because `a > 0` being false masks rest of the code, preventing the second condition from executing.

That leads to a definition of **multiple-condition coverage**, which is fulfilled when during a test run all possible combinations of condition outcomes in all decisions are invoked at least once.

Chapter 3

Types of side effects

This chapter will show kinds of side effects that can be spotted in Python programs. Real-world examples will be presented along with a commentary. But before that, a few terms need to be defined.

A **side effect** may be defined as an interaction with some external state. Interaction does not have to mean modification - an external state could only be read and that would still count as a side effect. The key point here is that the state can change at any time, outside of the current execution context (and thus the adjective "external"). When side effects are absent from an expression we can substitute any part of it with the computed value without changing its meaning. In other words, when there are no side effects, time is not a factor. Order of operations start to matter only in presence of side effects.

Side effects can be divided into two groups, depending on their reach. Side effects that are contained within the language interpreter and do not affect the outside world are **internal**. Side effects that extend beyond the interpreter are **external**. This basic classification can then be made more detailed by looking at exact targets of side effects. For example, object mutation is only a subset of internal side effects, just as variables rebinding. Internal side effects are strictly limited by language capabilities, and are in fact defined by it. Anything that deals with external state - state outside of the interpreter - is an instance of an external side effect, so in practice possibilities here are more vast.

There were two factors that played a role in choosing this particular categorization of side effects over any other:

1. inspection method, and

2. treatment of those side effects during testing.

Some side effects can be recognized by the function being called, while others are always caused by the same bytecode instruction - the inspection method influences the way side effects are categorized. Still, some side effects that utilize the same inspection method can be treated differently during testing. For example, a file system side effect is captured by a function call analysis, but requires totally different setup and teardown code than a mere mutation (which also uses function call analysis). Refer to chapter 4 for details.

Table 3.1: Types of side effects in Python

Internal	External
Mutation of a built-in type	Keyboard input
Instance variable rebinding	Terminal output
Global variable read	File system access
Global variable rebinding	Other
Class variable read	
Class variable rebinding	

Most types of side effects will be shown on the same sample application, called **A-A-P**. The application has a similar functionality to `make`, a popular Unix dependency-tracking build utility. **A-A-P** was written by Bram Moolenaar and is licensed under GNU GPL. Source code of **A-A-P** can be obtained from its homepage at `http://www.a-a-p.org/`. All code samples come from a module named `RecPython.py` that contains Python functions that can be used in "recipes", which in **A-A-P** are taken to mean instructions on how to build a program. Line numbers refer to this file as acquired from version 1.068 of the **A-A-P** package.

Examples for class variables read and rebinding will be shown on an implementation of the *Singleton pattern* [10] written by this thesis author.

3.1 Internal side effects

Figure 3.1 presents a simplified view of the interpreter and its interaction with the application code and the external world. In this view the **interpreter** is a program, with its internal state as memory, Python source code as the input data and bindings to the surrounding environment. **Internal state** includes things like bytecode intstruction pointer, application stack, state of application's data structures, but also private interpreter's data structures

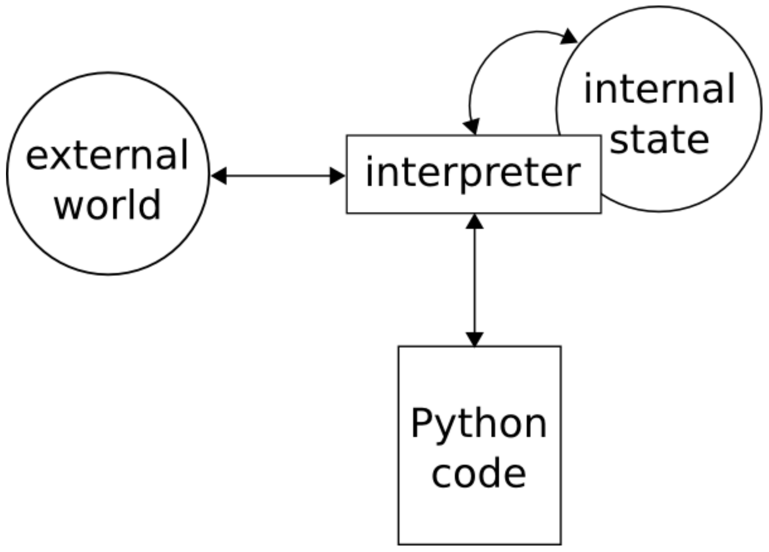


Figure 3.1: Interaction of the interpreter, application code and the external world

and bookeeping information, never visible to the running program. Any change to internal state of the interpreter could be considered an internal side effect. In this manner, an execution of a `NOP` instruction should be considered side effecty, as it increments the bytecode instruction pointer. Usually though, when internal side effects are considered, changes to the state that is invisible to the interpreted program are ignored.

The main joint between the interpeter and the program being interpreted are native data structures of a programming language. In Python those are integers, floats, strings, lists, dictionaries, etc. In CPython location of objects in memory is constant throughout execution, so the only visible change is by means of mutation. Mutation of the most basic Python data structures can manifest itself in multiple ways. Explicit change originating from the program itself (like an explicit call to `list.append` to add an element to a list) is an easy case, but others are more prevalent. All namespaces, including instance and class attributes, are implemented using a standard dictionary object (called `__dict__`) [14]. Thus, the mutation of this dictionary is a sign of a change in name binding. Moreover, since user-defined classes are constructed only from those basic data structures, any kind of change in user objects will be reflected in change to instances of those basic data types. Unfortunately, due to the way CPython is implemented, there is no uniform way to capture all kinds of side effects. For example, binding changes of instance variables manifest differently than mutation of other dictionaries. More on the topic can be found in chapter 4.

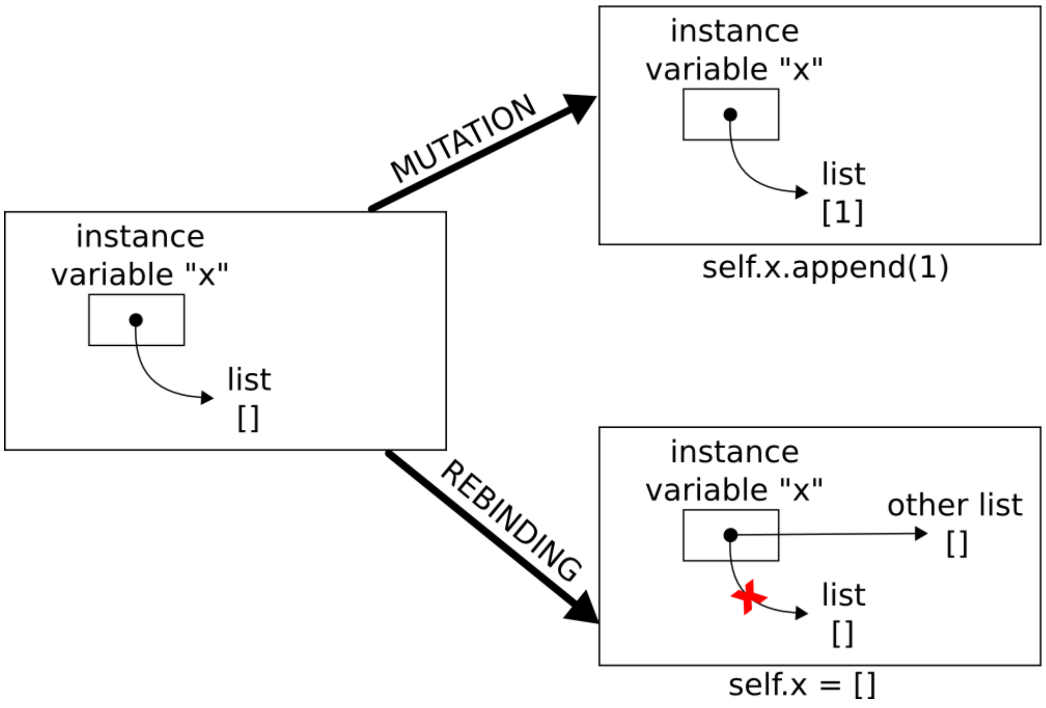


Figure 3.2: Mutation and rebinding

The difference between rebinding and mutation has been illustrated on figure 3.2.

3.1.1 Mutation of a built-in type

The most basic type of a side effect is mutation, i.e. a change to the state of a mutable data structure. For example, adding an element to a list is a mutation.

Python supports a few basic mutable data types, including lists, dictionaries and sets and for each of those there are many ways to modify them. A list can be sorted in place, a dictionary can be cleared and a set can be updated. Strings are not mutable in Python.

```python
class ImgHTMLParser(htmllib.HTMLParser):
    def __init__(self, formatter, verbose=0):
        htmllib.HTMLParser.__init__(self, formatter, verbose)
        self.img_list = []      # list of images so far
```

...

```
843     def handle_image(self, src, alt, *args):
844         if src:
845             from Remote import url_split3
846             scheme, mach, path = url_split3(src)
847             if scheme == '' and not os.path.isabs(src):
848                 # img file name is relative to html file dir.
849                 # Remove things like "../" and "./".
850                 n = os.path.normpath(os.path.join(self.img_dir, src))
851                 if not n in self.img_list:
852                     self.img_list.append(n)
```

Code listing above shows a fragment of a function that uses mutation on `img_list` instance variable. During object initialization `img_list` starts off empty and when a call to `handle_image` happens, based on a value of the `src` parameter the list can get a new element. Mutation happens on line 852 - method call `append` adds an element to a list.

3.1.2 Instance variable rebinding

Objects in Python are basically dictionaries, using `__dict__` attribute as the storage object with some metadata attached. Objects manifest behavior through special treating of the `__class__` attribute, and the related class hierarchy. When a method is called on an object or an attribute is referenced, contents of `__dict__` and methods defined on superclasses define the actual behavior. Encapsulation of attributes or methods is not enforced by the language, making all accesses legal and direct.

```
836 class ImgHTMLParser(htmllib.HTMLParser):
837     def __init__(self, formatter, verbose=0):
838         htmllib.HTMLParser.__init__(self, formatter, verbose)
839         self.img_list = []      # list of images so far
840     def set_dir(self, dir):
841         self.img_dir = dir      # dir of image file
```

Code listing above shows a simple implementation of a class with a constructor (in Python always called `__init__`) and a setter on attribute `img_dir`. The setter is an example of a method that has a side effect on instance variable binding. The effect of calling `set_dir` method is changing the binding of the `img_dir` instance variable. The side effect happens on line 841. It's worth noting that this side effect doesn't affect the original list at all, and thus is totally different from what was described in section 3.1.1.

Objects that utilize `__slots__` and don't have `__dict__` are treated the same way, i.e. storage method of attributes doesn't affect side effects. See 4.4.3 on page 34 for a description of the way attribute side effects are captured.

In Python classes are also objects - each class is an instance of a metaclass, abstracting behavior of all classes. Since classes are objects, the same treatment of attributes rebinding applies to them. More on that topic in the following sections.

3.1.3 Global variable read

There are more namespaces in Python than the scope of a user-defined object. There are other built-in types that have `__dict__` attribute and thus behave like a namespace. Each module has its own global namespace[1]. All top-level functions and class definitions are referenced by module's `__dict__`. The same is true for global variables - variables defined at the top-level of a module.

```
668 lastportname = None
669
670 def do_BSD_port(name, target):
671     """
672     Try using the BSD port system for the package "name".
673     "target" is "install" to install and "all" to build only.
674     Will ask the user when root permissions are required.
675     Returns non-zero for success.
676     """
677     global lastportname
```

...

```
690     if lastportname != name and not os.access(dirname, os.W_OK):
691         r = raw_input(("\nThe %s port appears to exist.\n" % dirname) +
692                       "Do you want to become root and use the port? (y/n) ")
```

Example code above shows a global variable defined at line 668 and a piece of code it is referenced by - condition at line 690 uses `lastportname`'s value.

[1]Each source file with `.py` extension is a module in Python. More on physical structure of a Python application can be found in section 6.3.1 on page 61.

3.1.4 Global variable rebinding

There are no constants in Python, at least not in the sense of constants in languages like C or Java. All global variables can be rebinded, what means that potentially any part of the codebase can modify a global variable.

```
726 lastpkgname = None
727 
728 def do_Debian_pkg(name, target):
729     """
730     Try using the Debian apt-get command for the package "name".
731     "target" is "install" to install and "all" to, eh, do nothing.
732     Will ask the user when root permissions are required.
733     Returns non-zero for success.
734     """
735     global lastpkgname

    ...

766     if target == "install":
767         cmd = "apt-get install %s" % name
768         if os.access("/", os.W_OK):
769             from Commands import aap_system
770             aap_system(0, recdict, cmd)
771         else:
772             do_as_root(recdict, [], cmd)
773         print "apt-get finished"
774 
775     lastpkgname = name
776 
777     return 1
```

Code listing above shows a fragment of an imperative function called `do_Debian_pkg`. It performs some actions and just before returning it rebinds a global variable `lastpkgname`. Side effect of calling this function is a change to the value of a global variable, happening exactly at line 775.

3.1.5 Class variable read

Similarly to modules described in previous sections, each class has its own scope. All methods and attributes defined at the class-level are available (and thus can be rebinded) through class' `__dict__`.

```
1 class Singleton(object):
2     instance = None
3     def __new__(cls):
4         if cls.instance is None:
5             cls.instance = super(Singleton, cls).__new__(cls)
6         return cls.instance
```

Implementation of the *Singleton pattern* [10] above shows a canonical example of class variables usage in Python. Class variable called `instance` is initiated at line 2 and then accessed two times: first at line 4 inside the condition and then at line 6 acting as a return value.

3.1.6 Class variable rebinding

Rebinding of class variables works the same as rebinding of instance variables, the only difference being that the receiver of the change is a class instead of a plain object.

```
1 class Singleton(object):
2     instance = None
3     def __new__(cls):
4         if cls.instance is None:
5             cls.instance = super(Singleton, cls).__new__(cls)
6         return cls.instance
```

Using the implementation of the *Singleton pattern* [10] again, the rebinding is clearly visible at line 5. Result of the call to `super` will be the new value of the `instance` class variable.

Special case of class variables rebinding is a technique called *monkey patching* [35] - modifying a module or a class definition at runtime. Since modules and classes in Python are mutable objects their definition can be altered at any time. Use of that technique is discouraged, but nevertheless possible.

3.2 External side effects

3.2.1 Keyboard input

Probably the easiest way to implement human-computer interaction interface in Python is to use keyboard for input and terminal for output. Output is described in the next section, but first keyboard input will be described.

There are many methods to get keyboard input in Python. A programmer can use a built-in function `raw_input` to read a whole line of input or access `sys.stdin` stream directly to do more complicated operations. The side effects in both cases are very similar. Read operations usually block the process and trim the read buffer, returning a string of characters.

```
690 if lastportname != name and not os.access(dirname, os.W_OK):
691     r = raw_input(("\nThe %s port appears to exist.\n" % dirname) +
692                   "Do you want to become root and use the port? (y/n) ")
693     if not r or (r[0] != 'y' and r[0] != 'Y'):
694         # User doesn't want this.
695         lastportname = None
696         return 0
```

Fragment of a function presented above queries a user for an answer. If the answer is not positive, a global variable `lastportname` is cleared and the function returns. Keyboard input side effect happens on line 691.

3.2.2 Terminal output

Terminal output is the most basic method of displaying output for a Python program. Even if the application has a GUI, terminal output is often used for debugging purposes. `Print` statement is one of the first things a Python programmer learns. As with keyboard input, there are more sophisticated methods of accessing the output streams: `sys.stdout` and `sys.stderr`, although they will not be discussed here.

```
766 if target == "install":
767     cmd = "apt-get install %s" % name
768     if os.access("/", os.W_OK):
769         from Commands import aap_system
770         aap_system(0, recdict, cmd)
771     else:
772         do_as_root(recdict, [], cmd)
773     print "apt-get finished"
```

Snippet of a function quoted above shows a common source of a terminal output side effect - the use of `print` statement. Line 773, when executed, will append a new line to a stream of output characters.

3.2.3 File system access

The last class of side effects covered in this work has to do with the file system. Creating files, reading files, descending directory tree, checking permissions are just a tip of the iceberg of all possible operations that can be performed on the file system. Ways to interact with the file system in Python are scattered across the interpreter and the standard library. To perform basic operations on files, a built-in `open` function will suffice, but more involved operations require use of the `os` module.

```
746 try:
747     if not os.path.isfile("/etc/debian_version"):
748         return 0    # Not a Debian system.
749     if not program_path("apt-get"):
750         return 0    # apt-get is not available (or $PATH wrong)
751 except:
752     # Can't use os.access().
753     return 0
```

Code listing above shows an example use of the `os.path.isfile` function. This function takes a string and queries the file system in search of a file with a given name. If found, `True` is returned, otherwise the result is `False`. This operation clearly depends on state of the file system, making it a good example of a file system side effect. `Os.path.isfile` invocation happens on line 747.

3.2.4 Others

There is plenty of other types of external side effects not covered in this work. Practically any extension implemented in C (thus outside of reach of the pythoscope's dynamic inspector) is a potential source of side effects. Pseudorandom number generators, database bindings, sockets, graphical display and interfaces, etc. - all are great real-world examples of extensions that can bring new types of side effects into a program. Handling all those different types of side effects is a good topic for more research, or possibly even another thesis. They are not listed and categorized here, because they are outside scope of this work.

Chapter 4

Capture and analysis of side effects

The concept of side effects does not exist in Python. Unlike languages like Eiffel[1] or Haskell[2], side effects in Python are not transparent to the programmer, i.e. there are no special language constructs to express whether a given statement or expression causes a side effect. Identification and description of different kinds of side effects has been a vital part of this thesis just as was actual implementation of their handling in Pythoscope. Types of side effects has been described in chapter 3. This chapter will focus on describing a method of capturing those side effects and their further analysis.

4.1 Bytecode interpretation, virtual machine and tracing

Bytecode is a representation of a program suitable for execution on a virtual machine. There are over 100 different bytecode instructions in Python, the exact number varies between interpreter versions.

Bytecode is produced from source code by the **compiler**[3]. For example, the following source code:

[1]Eiffel implements a concept called *Design by contract* described by Bertrand Meyer in [18], chapter 11, which allows to specify side effects as a part of method's contract.

[2]Haskell uses its type system to describe side effects through a language construct called monads. Refer to [20], chapter 7 for details.

[3]In fact, convertion of the source code into bytecode has two steps, first one involving a parser and an intermediate representation: Abstract Syntax Tree, but that doesn't bring anything into the discussion of tracing and can be safely omitted.

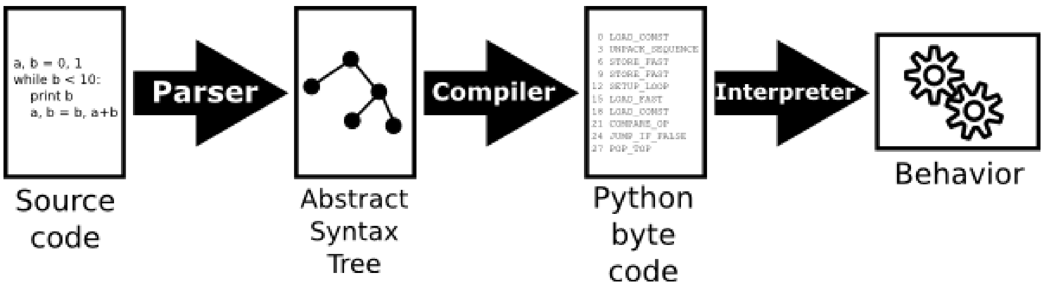
```
def myfunc(alist):
    return len(alist)
```

will get converted into the following sequence of bytecodes:

```
LOAD_GLOBAL              0 (len)
LOAD_FAST                0 (alist)
CALL_FUNCTION            1
RETURN_VALUE
```

Three of those bytecodes take arguments (listed above next to the bytecode name), while the last one takes none.

Process of parsing, compilation and execution has been presented on figure 4.1.



```
a, b = 0, 1
while b < 10:
    print b
    a, b = b, a+b
```

```
 0 LOAD_CONST
 3 UNPACK_SEQUENCE
 6 STORE_FAST
 9 STORE_FAST
12 SETUP_LOOP
15 LOAD_FAST
18 LOAD_CONST
21 COMPARE_OP
24 JUMP_IF_FALSE
27 POP_TOP
```

Figure 4.1: Parsing, compilation and execution in CPython

A virtual machine (VM) interprets each bytecode instruction by executing a machine code corresponding to the bytecode's meaning. CPython interpreter is a portable virtual machine implemented in C programming language[4].

Execution of bytecode by the CPython interpreter utilizes a *stack* for holding intermediate computation data, like arguments for operators or functions and return values. This makes the CPython VM a *stack machine*. Later in this chapter a name of **value stack** will be used to refer to the interpreter's stack.

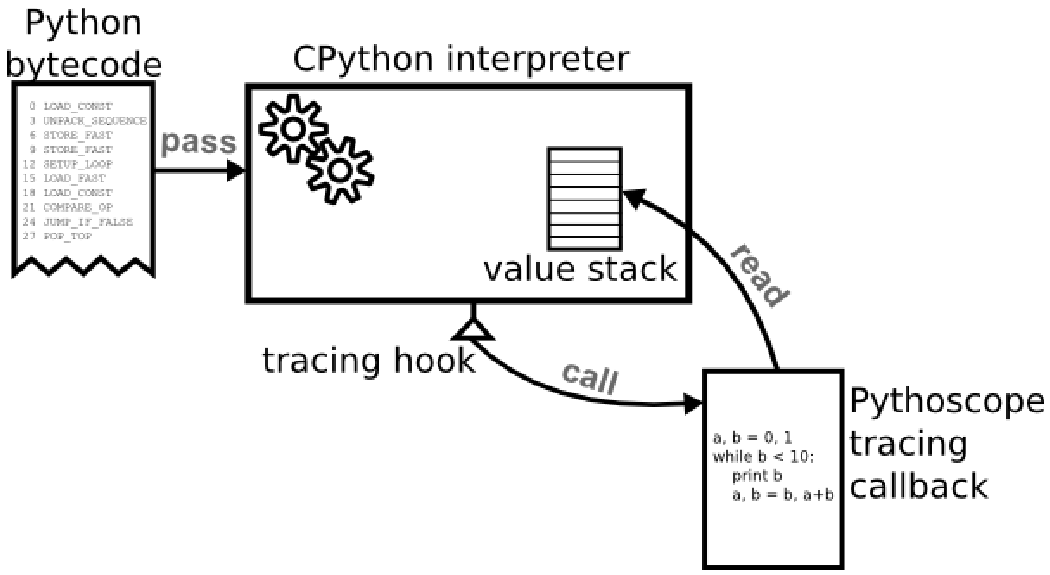
The CPython interpreter provides a number of **tracing hooks** that allow writing profilers, debuggers and other tools that need to perform analysis of a running program. Each hook corresponds to a different execution event: a function call, a function return, exception propagation or a step into a new source code line. The same tracing hooks allow Pythoscope to inspect side

[4]For more on CPython refer to section 6.2.1 on page 57

effects, as well as perform rest of dynamic analysis: creating execution graph and gathering values.

Tracing hooks can be used by registering **tracing callbacks** - Python functions that take event details as input and are run in a separate context, what means that their instructions are not being traced, avoiding an infinite loop.

4.2 Bytecode instrumentation



```
 0 LOAD_CONST
 3 UNPACK_SEQUENCE
 6 STORE_FAST
 9 STORE_FAST
12 SETUP_LOOP
15 LOAD_FAST
18 LOAD_CONST
21 COMPARE_OP
24 JUMP_IF_FALSE
27 POP_TOP
```

```
a, b = 0, 1
while b < 10:
    print b
    a, b = b, a+b
```

Figure 4.2: Bytecode, interpreter, tracing hook and callback

Each bytecode instruction to be executed is first passed over to a registered callback that can analyze it. Once the tracing function finishes processing the context switches back to the interpreter which executes the next bytecode instruction.

Knowledge of the bytecode instruction itself is not enough to reason about the behavior of a program under inspection. Since CPython is a stack-based interpreter it utilizes a value stack for most of its processing. In particular, arguments to bytecode instructions are passed by the stack. For example, to execute a `BINARY_ADD` instruction, the CPython interpreter does the following:

1. pops value of the first addend from the value stack

2. pops value of the second addend from the value stack

3. adds the two objects
4. pushes the computed sum to top of the value stack

In effect, execution of the `BINARY_ADD` instruction causes the stack to change. This is the case with many other bytecode instructions.

In order to properly discover and analyze side effects the tracing callback need not only recognize the bytecode, but also access the value stack. The interaction between bytecode, the interpreter and a tracing callback is presented on figure 4.2.

4.3 A model for capturing side effects

All side effects in Python can be tracked back to a single point: the underlying interpreter implementation, which in case of CPython is code implemented in C. Means of invoking that code from a Python program are twofold. The most common case is directly invoking a function implemented in C that causes side effects. For example, by calling the built-in function `open` a multiple filesystem-related side effects will be invoked: file will be checked for existence, possibly created and opened afterwards, for reading or writing depending on options passed to the function by the programmer. Second method of invoking a side effecty code in CPython is by a particular bytecode instruction. For example, a `print "Hello"` statement maps to `PRINT_ITEM` bytecode, which does a side effect on terminal output.

Based on that description of sources of side effects a **rule-based system** can be constructed that will be able to recognize side effects. A set of simple rules like “when function X sucessfully returns assume side effect Y happened” or “when bytecode X is executed assume side effect Y happened” can be prepared to cover all cases of side effects. There are four possible rule templates, presented below.

1. side effect X happens always when function F is called (bytecode instruction B is executed)
2. side effect X happens when function F is called (bytecode instruction B is executed) with arguments A
3. side effect X happens when external entity E is in state S at the time of execution of function F (bytecode instruction B)
4. side effect happens only at certain times that cannot be reliably determined (e.g. activation of a side effect depends on the internal state of a module implemented in C)

First three cases can be implemented, but the last one cannot. If there is no way to tell if a side effect happened or not inspection will never be complete. The only answer to that problem is to ignore those side effects, don't inspect them and don't consider them during test generation.

Preparation of such side effect inspection rules requires careful consideration and knowledge about both interpretation of bytecode by the CPython interpreter and behavior of standard library functions. This method can also be extended to external libraries, but just as in the case of the standard library side effects will have to be explicitly defined by a set of rules. In summary, this technique of inspecting side effects is very general and can be used in many contexts, but requires significant implementation work, as for each side effect a set of rules is required. This follows from a practically infinite number of side effects types, mentioned in chapter 3.

4.4 Bytecode instructions with side effects

Most of Python bytecodes don't cause side effects. Because of that only a small set of bytecodes has to be monitored by Pythoscope. Those can be divided into three groups, each described in their own subsection.

Descriptions are accompanied by tables that list all inputs and outputs of each bytecode, according to transformation schema presented in section 1.4. The only difference is that arguments in case of bytecodes are taken from the *value stack*, while return value is placed on the *value stack*.

4.4.1 `CALL_FUNCTION` family of bytecodes

This group includes four bytecodes: `CALL_FUNCTION`, `CALL_FUNCTION_VAR`, `CALL_FUNCTION_KW` and `CALL_FUNCTION_VAR_KW`. All result in a function call, but each one has a slightly different calling convention. `CALL_FUNCTION` bytecodes themselves don't read or modify external state, although functions that they call may do those things. Captured side effects depend strictly on rules, as described in the previous section. Value stack plays a crucial role in determining side effects during function calls.

Table 4.1: Inputs and outputs for CALL_FUNCTION family of bytecodes

Bytecode	Takes arguments	Reads external state	Returns value	Raises exception	Modifies external state
CALL_FUNCTION	x		x	x	
CALL_FUNCTION_VAR	x		x	x	
CALL_FUNCTION_KW	x		x	x	
CALL_FUNCTION_VAR_KW	x		x	x	

4.4.2 PRINT family of bytecodes

This group includes four bytecodes: `PRINT_ITEM`, `PRINT_ITEM_TO`, `PRINT_NEWLINE` and `PRINT_NEWLINE_TO`. Each of them causes a side effect on a file-like object[5]. Usually it means printing to a terminal output, but not always - it all depends either on an object bound to `sys.stdout` variable (in case of `PRINT_ITEM` and `PRINT_NEWLINE`) or on an object explicitly passed to the print statement (in case of `PRINT_ITEM_TO` and `PRINT_NEWLINE_TO`). For example, when a `StringIO` object[6] has been passed a side effect is a mutation of that object and no filesystem-related side effect occurs.

Table 4.2: Inputs and outputs for PRINT family of bytecodes

Bytecode	Takes arguments	Reads external state	Returns value	Raises exception	Modifies external state
PRINT_ITEM	x	x		x	x
PRINT_ITEM_TO	x			x	x
PRINT_NEWLINE		x			x
PRINT_NEWLINE_TO	x			x	x

[5]The Python Language Reference [48] defines "file-like object" as an object with a `write()` method.

[6]See `StringIO` documentation page [47] for a description of a `StringIO` object.

4.4.3 Bytecodes that deal with object attributes and global variables

This group includes three bytecodes related to attribute access (`LOAD_ATTR`, `STORE_ATTR` and `DELETE_ATTR`) and three bytecodes related to global variables access (`LOAD_GLOBAL`, `STORE_GLOBAL` and `DELETE_GLOBAL`). Execution of all of those bytecodes unconditionally causes a corresponding side effect. `STORE` bytecodes modify the binding of a variable, `DELETE` bytecodes remove the binding itself, while `LOAD` is a read operation.

Table 4.3: Inputs and outputs for bytecodes that deal with object attributes and global variables

Bytecode	Takes arguments	Reads external state	Returns value	Raises exception	Modifies external state
`LOAD_GLOBAL`	x	x	x	x	
`STORE_GLOBAL`	x				x
`DELETE_GLOBAL`	x	x		x	x
`LOAD_ATTR`	x	x	x	x	
`STORE_ATTR`	x			x	x
`DELETE_ATTR`	x	x		x	x

4.5 Module import mechanism

Import mechanism in Python can have two sources: it can be invoked by certain bytecode instructions (`IMPORT_STAR`, `IMPORT_NAME` and `IMPORT_FROM`) or it can be invoked by calling the built-in `__import__` function. In this regard it is neither a direct source of side effects (a concrete bytecode instruction or a C function is) nor it is a type of side effect (as it can cause many different types of them). It has been included in this chapter because it constitute an important part of Python, and no matter the source, the mechanism behaves the same way.

The import mechanism can cause a plethora of side effects of different kinds. In the most basic case an import statement refereces contents of a global variable, in the more involved it can query and modify the file system, modify global state and execute arbitrary Python code. Importing process is illustrated below with possible side effects listed next to the description.

1. Import arguments are parsed. There are a number of alternatives of the import statement. Examples of valid import statements include:

   - `import sys`
   - `from itertools import count, groupby`
   - `from os.path import basename`
   - `from stat import *`
   - `import curses.ascii`

   Each version has different semantics. Parsing import statements doesn't cause any side effects.

2. `sys.modules` global dictionary is checked to see if the module that is needed has been imported before. If yes, it is returned immediatelly and the import process stops.

3. If the module hasn't been imported before, it is looked up on the file system. Rules for querying the file system grown complicated over the version of Python, but there is no reason to list them all here. Important thing is that in this step file system is read extensively: directories' contents are listed, files are read and their properties (e.g. last modification time) are checked.

4. All Python source files required by the import statement are compiled. Compilation doesn't cause any side effects. This step can be skipped if an already compiled version of the module is present.

5. If `.pyc` file is not present or outdated a write to a file is attempted. Failures are generally ignored (a file write may fail for any number of reasons, e.g. because of unsufficient permissions).

6. All Python modules required by the import are executed. Since a module can contain any Python code all kinds of side effects can happen during this stage. Modules that cause side effects during import that extend beyond the scope of the module itself are often called to be *not import-safe*.

7. Finally, `sys.modules` global dictionary is updated with the information about the just imported module. Any future imports from this module will use that information, speeding up the process and preventing from any module to be imported more than once.

In summary, the import procedure itself can cause the following list of side effects[7]:

- global variable (`sys.modules`) read
- file system access
- built-in type (dictionary referenced by `sys.modules`) mutation

in addition to any side effects caused by the execution of a module being imported.

Current version of Pythoscope ignores side effects listed above and only traces execution of application modules. In other words, the import procedure itself is treated as being side-effects-free, which strictly is erroneous, but will cause problems in practice only for programs that reconfigure the importing process itself[8]. What Pythoscope *does* trace is side effects inside the application modules themselves. In other words, Pythoscope treats import mechanism as an internal concern of the interpreter.

4.6 Sources of side effects

As mentioned earlier in section 4.3 there are two general sources of side effects in CPython: calls to certain C functions and invocations of certain bytecode instructions. Those have been described in previous sections, so now it is possible to connect all types of side effects described in chapter 3 with ways of invoking them. Table 4.4 puts side effects and their sources together.

4.7 Bytecode tracing example

Figure 4.3 presents an interaction between the interpreter, bytecode tracer callback and the value stack during tracing of a sample code presented earlier in this chapter (see section 4.1 on page 28). Before start of the inspection, the bytecode tracer callback is registered with the interpreter tracing hook. After that point each bytecode executed can be captured and analyzed by the bytecode tracer. Some bytecodes (like `LOAD_GLOBAL` in this case) are always interpreted to cause side effects, others are always ignored (like `LOAD_FAST` in this example) and others are conditional - their assesment usually involves

[7]Refer to chapter 3 for a description of those side effects.

[8]Python documentation [42] lists an import hook as "an advanced function that is not needed in everyday Python programming".

Table 4.4: Side effects and their sources

Side effect	Sources
Mutation of a built-in type	C functions (e.g. `list.append`),
Instance variable rebinding	`STORE_ATTR` and `DELETE_ATTR` bytecodes, C functions (e.g. `setattr`)
Global variable read	`LOAD_GLOBAL` bytecode, C functions (e.g. `getattr`),
Global variable rebinding	`STORE_GLOBAL` and `DELETE_GLOBAL` bytecodes, C functions (e.g. `setattr`),
Class variable read	`LOAD_ATTR` bytecode, C functions (e.g. `getattr`)
Class variable rebinding	`STORE_ATTR` and `DELETE_ATTR` bytecodes, C functions (e.g. `setattr`)
Keyboard input	C functions (e.g. `raw_input`)
Terminal output	`PRINT` family of bytecodes, C functions (e.g. `sys.stdout.write`)
File system access	C functions (e.g. `open`),
Other	C functions

checking the value stack. In the example a `len` built-in function is invoked. Rules in Pythoscope define that it doesn't cause side effects, so the call is eventually ignored.

The same process is carried over for all code executed after registering the tracer callback. Each bytecode is analyzed and extracted side effects are stored for later, to be utilized during test generation.

4.8 Implementation of the bytecode tracer

Vanilla CPython is not capable of tracing individual bytecode instructions. It has mechanisms for tracing lines of code andfunction calls, which is enough to implement debuggers, profilers and code coverage tools, but doesn't allow tracing side effects, crucial to Pythoscope. The problem of adding bytecode tracing capabilities to CPython can be divided into two subproblems: the

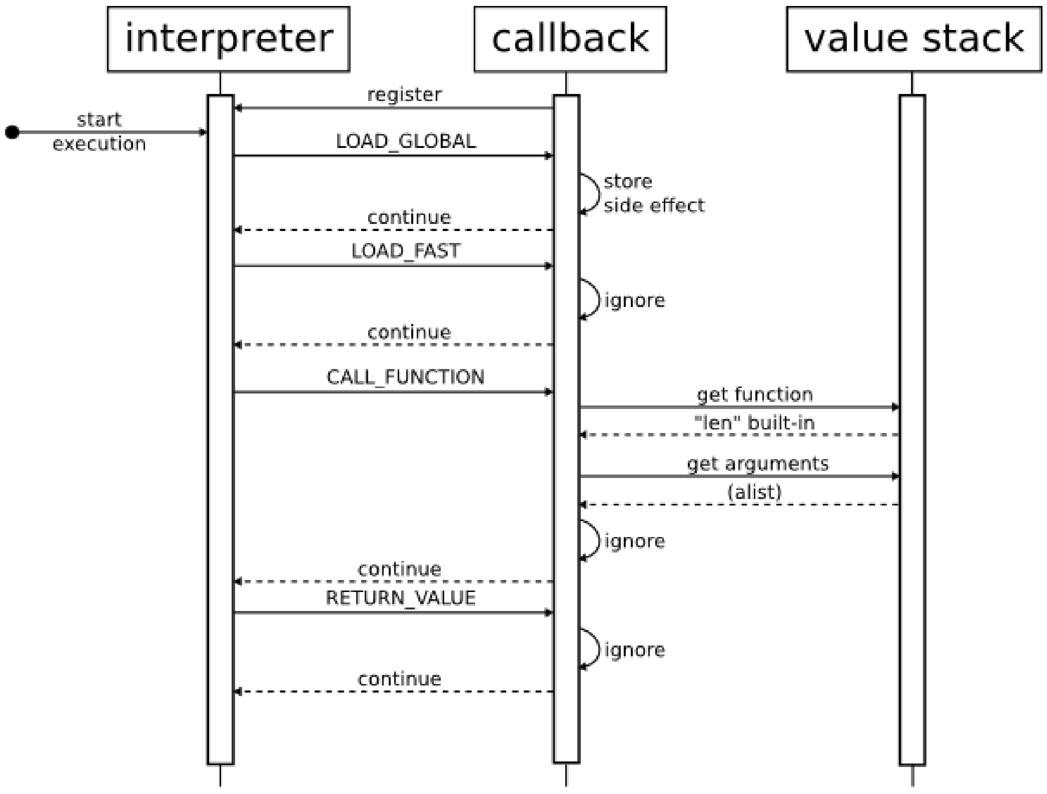


Figure 4.3: Bytecode tracing sequence diagram

problem of forcing the interpreter to invoke a callback between execution of bytecodes and the problem of access to interpreter's internal state.

A base for capturing execution of bytecodes was a technique published in 2008 by Ned Batchelder [3]. To describe the method an explanation of the structure of Python `code` objects is first needed.

Python code objects (not to be confused with bytecode) are used internally by the interpreter, yet are exposed to the programmer[9]. Code objects contain description of a function required for its execution, like its arguments count, number and names of local variables, stack size, bytecode instructions and a special mapping from bytecode offsets to line numbers, called `lnotab`. This mapping is then used by the interpreter during line tracing to determine when it should execute a registered trace callback. The check is done after each bytecode instruction, but callback is invoked only after a new source code line is entered. What interpreter considers a new line is controlled by `lnotab`.

To enable bytecode tracing then it is necessary to modify the `lnotab` in

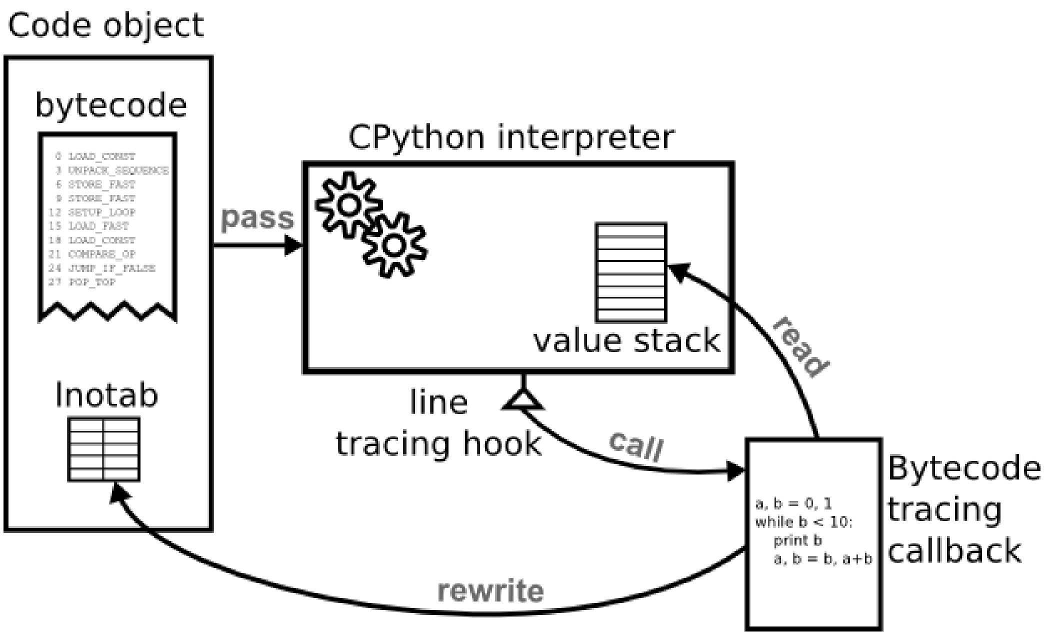
[9]For a description of code objects refer to chapter 3 of [48]

such a way that the interpreter will treat *every* bytecode instruction as a new line, effectively turning line tracing into bytecode tracing. The bytecode tracer does exectly that as a step preceding execution - it modifies `lnotab` of the code to be executed. In a dynamic language like Python is isn't enough though. Because of the capability to define functions on-the-fly and dynamically load (import) any code, throughout execution the bytecode tracer has to control any new code objects and rewrite them before they are executed. This *dynamic rewriting* of code objects is a key technique making bytecode tracing in Python possible.

The only problem when this technique requires special treatment is import of modules. Import process from the point of view of the tracer function is atomic: before the execution of an import statement the module is not loaded, and after: the module is present and initialized. The problem is that the bytecode tracer needs to rewrite module's code object *after* it is read, but *before* it is executed. To achieve that a custom import hook has been used [33].

The problem of accessing interpreter's internal state is strictly related to the *value stack*, as this is the structure where lies all data important for bytecode tracing. Value stack is not normally accessible from the level of a Python application. The bytecode tracer implemented in Pythoscope uses ctypes [31] extension for accessing values of the interpreter's stack, allowing to capture and analyze arguments to all bytecodes and calls, what ultimately enables dynamic analysis of side effects (refer to section 4.7 for an example).

Figure 4.4 is a view of figure 4.2 specific to the context of bytecode tracing. Code object has been marked as a container for bytecode and `lnotab`, which is being rewritten by the callback.



```
 0 LOAD_CONST
 3 UNPACK_SEQUENCE
 6 STORE_FAST
 9 STORE_FAST
12 SETUP_LOOP
15 LOAD_FAST
18 LOAD_CONST
21 COMPARE_OP
24 JUMP_IF_FALSE
27 POP_TOP
```

```
a, b = 0, 1
while b < 10:
    print b
    a, b = b, a+b
```

Figure 4.4: Interaction of bytecode tracer and `lnotab` during tracing

Chapter 5

Test generation from collected data

Before delving into a topic of test generation, a structure of a good unit test case has to be presented first. Tests generated by Pythoscope should obey all rules of manually written test cases, so they are just as easy to maintain and extend. There are two basic rules of unit testing structure: one have to do with the layout of a single test case, and the other with composition of test cases into test suites.

5.1 Four-phase test cases

Gerard Meszaros in his book "xUnit Test Patterns" [17] presents a structure of a typical unit test case. He calls it a *Four-Phase Test* pattern.

> We design each test to have four distinct phases that are executed in sequence: fixture setup, exercise SUT, result verification, and fixture teardown.
>
> 1. In the first phase, we set up the test fixture (the "before" picture) that is required for the SUT to exhibit the expected behavior as well as anything you need to put in place to be able to observe the actual outcome (such as using a *Test Double*).
> 2. In the second phase, we interact with the SUT.
> 3. In the third phase, we do whatever is necessary to determine whether the expected outcome has been obtained.

4. In the fourth phase, we tear down the test fixture to put the world back into the state in which we found it.

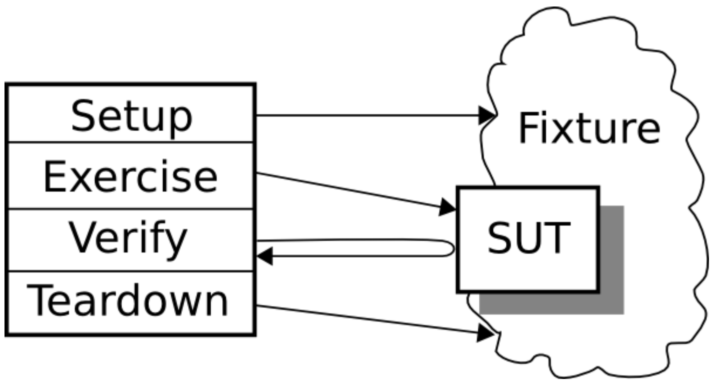


Figure 5.1: *Four-Phase Test* pattern

Exercise step invokes the SUT, while the **verify** step compares SUT's output with expected outcome. In case of a mismatch, a test failure is reported.

Setup puts the SUT in a state required to carry out a particular test scenario. That usually involves preparation of the environment and initialization of objects.

Teardown phase is responsible for bringing the world back to a pristine state. Not all actions performed during the setup phase need a corresponding teardown - some resources tear down automatically. One of such things are objects - Python as a garbage-collected language doesn't need to explicitly destroy objects and free memory allocated to them. External resources usually require explicit teardown, for example files created during setup need to be deleted before the test completes. Teardown phase should be unconditional - meaning any possible test errors in the exercise or verify phases should not interfere with it. In Python this is usually achieved by a use of a `try-except` block.

Setup and teardown phases are optional - only exercise and verification are required to form a valid test case.

In Python unit test cases are written as methods - name of a method need to start or end with the word "test", while the method body need to contain the four phases. A sample Python unit test case is presented below.

```python
def test_empty_database_has_no_users(self):    # method name
    conn = Database.open()                     # setup
    count = conn.select('count', 'users')      # exercise
    assert count == 0                          # verify
    conn.close()                               # teardown
```

5.2 Test suites as classes

Second pattern of structuring unit tests organizes the way test cases are bundled together into sets, or the so called **test suites**. The idea is to employ a way of gathering executable pieces of code together natural to the programming language. In the case of Python, a language with strong object-oriented support, this involves utilizing classes and methods. Gerard Meszaros calls this pattern *Testcase Class* pattern [17] and the idea is to group a set of related *Test Methods* on a single *Testcase Class*.

Using this pattern, a sample test suite in Python can look like this:

```
class TestUser(unittest.TestCase):
    def test_user_does_this(self):
        ...
    def test_user_does_that(self):
        ...
```

There are a few things worth noticing here. First, a test class name has to contain the word "Test". Second, this class has to inherit from a general `unittest.TestCase` class, which implements functionality common to all test classes.[1] Third, each test case is a separate method with a unique name.

This pattern of constructing test suites, besides keeping related test cases together, additionally allows to archieve some kind of testing abstraction by utilization of inheritance. The `unittest.TestCase` class itself contains some tools helpful during testing. Those include:

1. `setUp` and `tearDown` methods that define setup and teardown for all test methods in a class,
2. set of common assertions, like `assertEqual` or `assertRaises`,
3. an algorithm for running all test methods in a class.

By making specialized subclasses of `unittest.TestCase` a developer can abstract common patterns that appear in her test suites. This is usually an

[1]This requirement of inheriting from `unittest.TestCase` is specific to the unittest library [49]. Other Python unit testing environments usually don't care about superclass of a test class. This behavior can be found in nose [36] and py.test [39] testing frameworks.

extension of the default set of assertions or specific setup and teardown procedures. Other abstractions are also possible, e.g. overriding order in which test methods are run. Extensibility of the Python unit testing framework is its big strength, allowing the growing test suites to stay manageable and understandable.

5.3 Setup and teardown

As was mentioned in chapter 2 isolation is a key characteristic of a good unit test case. Means of achieving isolation will be presented here.

One aspect of isolation is environment, both internal and external. Non-pure code can be sensitive to certain parts of the environment and a valid test case would have to depend on a careful preparation of that environment. But preparation is only half of the story - in order to keep unit tests separated, a cleanup procedure has to also be taken into consideration. Those two phases of environment preparation and cleanup are enclosed within setup and teardown of a test case, mentioned at the beginning of this chapter.

Setting up the environment to satisfy requirements of a test case is not the only way to achieve isolation. Another way is to use test doubles[2] in place of environment dependencies. Test doubles are usually implemented as standard Python objects that are garbage-collected meaning they don't need any explicit teardown. Also, they are usually easier to setup than real external dependencies. Lastly, test code that uses test doubles generally runs faster than code that uses real objects. For example, in-memory database mock object will behave faster (and more reliably) than a real database communicating with a test case by a socket. The only downside of using a test double is a risk of testing the wrong thing. A test double that behaves differently than a real object may conceal a bug in the application code.

As an example of isolation code in test cases generated by Python, consider the following application code:

```python
# module.py
no_calls = 0

def fun():
    global no_calls
    no_calls += 1
    return True
```

[2]See section 2.3 on page 14 for a definition of a test double.

A function `fun` modifies a global variable `no_calls` each time it is called, always returning `True`. To properly test such a function it is necessary to prepare a state of `no_calls` global variable before the test (*setup*) and restore it to its previous value after the test (*teardown*). If a single call to `fun` has been caught during dynamic inspection, Pythoscope will generate the following test case:

```
1  from module import fun
2  import unittest
3  import module
4
5  class TestFun(unittest.TestCase):
6      def test_fun_returns_true(self):
7          old_module_no_calls = module.no_calls
8          module.no_calls = 0
9          self.assertEqual(True, fun())
10         self.assertEqual(1, module.no_calls)
11         module.no_calls = old_module_no_calls
```

Setup consists of storing the old value of `no_calls` (line 7) and setting its value to 0 (line 8). After the assertions, teardown can be run which takes care of restoring `no_calls` to its former state (line 11). This way other tests that depend on the state of this global variable won't be affected.

5.4 Exercise step

Pythoscope combines the exercise step with a part of the verification step through *assertions*. In testing, **assertion** is a function that ensures that a certain condition holds, and if it doesn't hold a test failure is reported. For example, the following assertion checks that the return value of a call to `plusone(10)` will be 11.

```
self.assertEqual(11, plusone(10))
```

Call to `plusone(10)` is the exercise step, while the whole assertion itself forms the verification step. While it would be possible to split the above assertion into two lines:

```
result = plusone(10)
self.assertEqual(11, result)
```

hereby putting the exercise and verification steps on separate lines, it is customary to combine them as in the first example. This is also the convention that Pythoscope uses.

5.5 Verify step for pure functions

Wikipedia defines a pure functions as a function that meets the following two conditions [50]:

1. The function always evaluates the same result value given the same argument value(s). The function result value cannot depend on any hidden information or state that may change as program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices.
2. Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices.

Testing pure functions is substantially easier than testing other code. First off, usually no setup or teardown is required, as the function only depends on the arguments passed. Second, all that needs to be asserted about such function is its return value. By definition a pure function does no side effects, so there's no need to check for mutation of input, globals or change to any other external state.

Consider the following definition:

```
def plusone(x):
    return x + 1
```

Given that Pythoscope captured `plusone(10)` call during dynamic inspection the generator will be able to form a complete test case presented below.

```
class TestPlusOne(unittest.TestCase):
    def test_plusone_returns_11_for_10(self):
        self.assertEqual(11, plusone(10))
```

As long as the arguments list is short and constructing it brings close to none dependencies the test cases will be as readable as the one presented above. While they may require a complex setup, test cases for pure functions will always contain only one assertion.

For the purposes of Pythoscope's inspection, functions that raise exceptions can still be considered pure, as long as they fulfill the two conditions mentioned earlier. Consider the following example:

```
def raise_if_too_big(x):
    if x > 100:
        raise ValueError
```

Given that Pythoscope captured a call to `raise_if_too_big(101)` during dynamic inspection the generator will be able to form a test case with a `assertRaises` assertion that checks if the provided exception has really been raised.

```
class TestRaiseIfTooBig(unittest.TestCase):
    def test_raise_if_too_big_raises_value_error_for_101(self):
        self.assertRaises(ValueError, lambda: raise_if_too_big(101))
```

Those two types of assertions (`assertEqual` and `assertRaises`) are all that is needed for testing output of pure functions, no matter if it is a return value or an exception.

5.6 Verify step for functions with side effects

Once side effects enter the picture it is no longer possible to test the SUT only with assertions on the return value (mentioned in the previous section). All side effects have to be treated as behavior of the SUT and tested accordingly. In Pythoscope this is achieved by assertions that check state transitions - they assert that state has been properly changed after exercising the SUT.

A sample generated test case from section 5.3 will be used as an example.

```
from module import fun
import unittest
import module

class TestFun(unittest.TestCase):
    def test_fun_returns_true(self):
        old_module_no_calls = module.no_calls
        module.no_calls = 0
        self.assertEqual(True, fun())
        self.assertEqual(1, module.no_calls)
        module.no_calls = old_module_no_calls
```

Calling function `fun` causes a side effect: a change in binding of a global variable `no_calls`. To test that such side effect actually happens, two elements are needed. First, before the exercise step, preparing an initial state of the `no_calls` variable becomes a part of the setup procedure (already described in section 5.3). Here, `no_calls` is set to 0 in line 8. After the exercise step (line 9) an assertion on the new changed state of `no_calls` variable is made. Here, the `assertEqual` in line 10 asserts that `no_calls` has now a value of 1.

All internal side effects can be handled by an equality assertion of some kind. External side effects require more sophisticated methods of checking state, usually involving use of test doubles. For example, to test terminal output[3] it is best to mock the standard output stream and assert changes to the mock objects after the exercise step.

5.7 Requirements for test generation

Test generation exhibits a few classical challenges related to code generation. First of all, each test case as well as the whole test suite file, have to be **valid Python code**, i.e. all of the code has to be compilable and executable by the standard CPython interpreter.

Second, a lot of care has been put into generation of **readable code**. Test cases generated by Pythoscope are meant to be modified by a programmer, so all typical conventions of good code style[4] have to be obeyed.

Third, the whole test suite have to be **executable in the test environment** of the user's choice. *unittest* environment from the Python's standard library is the default, although other test environment are supported as well[5].

Finally, the test cases should not only be executable, **all test cases should actually pass**, given that the application code has not been altered after test generation. All those areas of interest put unique constraints on the test generation process, at times forcing hard decisions between simplicity and correctness.

5.8 Test generation process

Data gathered during inspection has a hierarchical structure. It is presented on figure 5.2.

Static inspection recognizes the application as being composed of many modules and each module contains standalone functions and classes. After

[3]Refer to section 3.2.2 on page 26.

[4]The most popular and basic code style conventions for Python are described in the PEP-8 document [22].

[5]For alternatives to unittest see *nose* [36] and *py.test* [39].

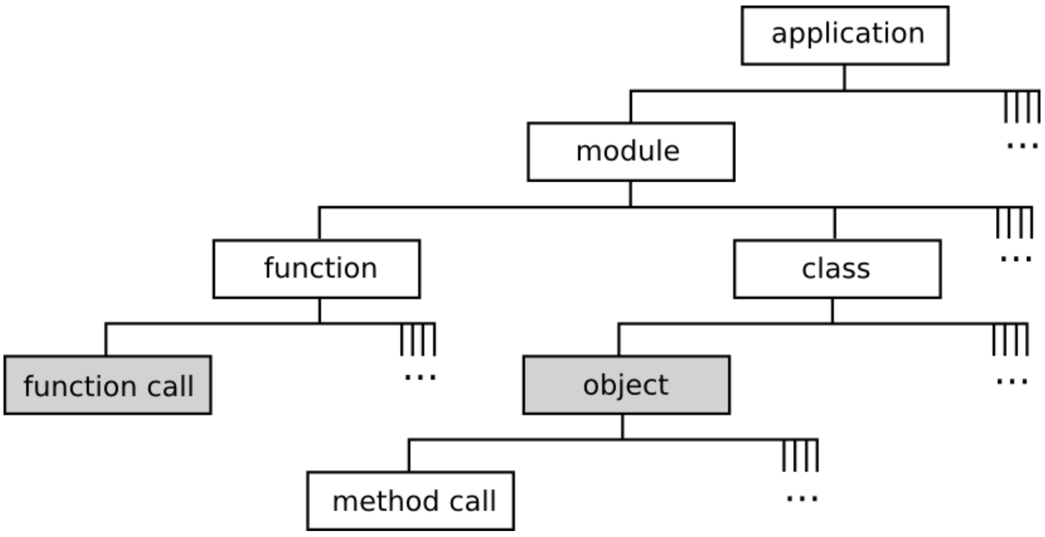


Figure 5.2: Application structure after inspection

dynamic inspection that view is enriched with information on actual uses of those functions and classes. Functions are called yielding function calls, while classes are instantiated giving objects, each of which is related to one or more method calls. Each call, be it function or method call, can be summarized with four parameters:

1. arguments
2. return value
3. exception
4. side effects

Since side effects are interactions with the external state, they cover both state before and after the call. With this in mind it is clear that a description of a call after dynamic inspection is consistent with a theoretical view of a call as a transformation, described in section 1.4.

Testable interactions - each interaction can be a base for a single test case - has been marked on figure 5.2 in gray. Read on for an explanation how those interactions are used in the test generation process.

The process of test generation can be summarized in the following three steps (also presented on figure 5.3):

1. Each module selected by the user for test generation is queried for *testable interactions*: function calls and objects. Each such interaction

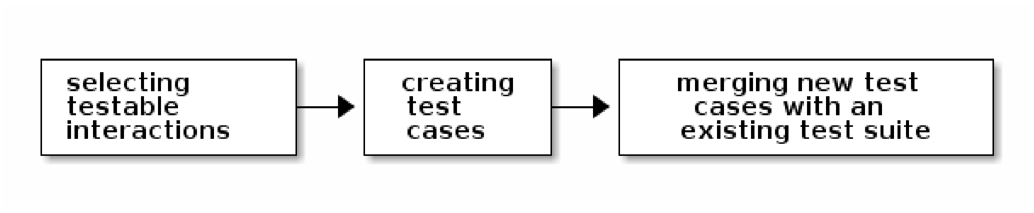

selecting testable interactions
creating test cases
merging new test cases with an existing test suite

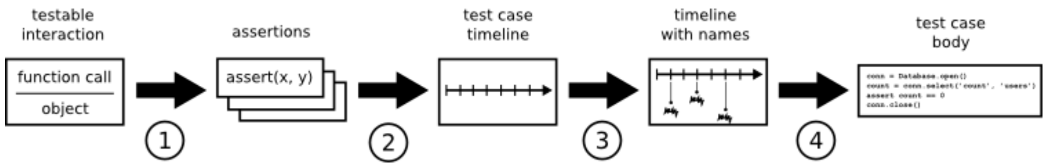
Figure 5.3: Test generation process

describes a transformation (in case of a function call) or a chain of related transformations (in case of an object and its method calls) that can form a single test case.

2. Each *testable interaction* forms a *test method*, and a set of test methods for the same group of interactions forms a *test class*. All four phases of a test case are generated solely based on the information gathered during dynamic inspection: runtime dependencies form setup and teardown, calls form an exercise phase and expected outcome (transformation) forms the verification phase.

3. For each newly generated test class, Pythoscope finds a proper test file to contain it (or creates a new one if necessary) and then puts the new test class at the end. If a test class already existed, contents of both classes are merged. Test methods named the same can be either skipped or overwritten, according to user's preference.

While selection and merging steps are straightforward, the second step deserves a detailed treatment. It can be further divided into four distinct phases, illustrated on figure 5.4.



```
conn = Database.open()
count = conn.select('count', 'users')
assert count == 0
conn.close()
```

Figure 5.4: Creating test cases

1. Generating assertions is a process of taking the *testable interaction* and converting all of its outputs (both explicit and implicit) into checks. For example, if at runtime a function `addone` returned `2` for argument `1` we can form the following assertion: `assert_equal(2, addone(1))`.

The same principle applies to exceptions and side effects. Every change to an external resource made by the testable interaction will generate a separate assertion. Assertions usually embody both exercise and verification phases of a four-phase test case.

2. **Resolving dependencies** is a step where assertions are laid down on a timeline along with all their dependencies. It is a moment where setup and teardown is prepared. This timeline is a test scenario, an abstract representation of actions that will happen during testing. Refer to section 5.9 for a list of dependency resolving rules.

3. **Naming objects** phase takes the test timeline and gives names to objects that need one. If an object is used only once in a test case it doesn't need a name. By adhering to this rule the test contents are kept short and readable. See notes on test cases readability in the next section.

4. **Generating test contents** is the last phase of constructing a test case body. In this phase each entry on the timeline is converted to a test code fragment. End result is a complete string of Python test code, ready to be embedded in a test method. After that a test method is named and put into a test class.

5.9 Test dependencies and hard-to-test code

One of the problems of test generation is resolving test dependencies. Exercise and verify steps of a unit test define dependencies that need to be met in order for the test to run successfuly. Dependencies need to be resolved recursively: if object X is required by a call in the exercise step and constructing it requires object Y, both X and Y need to be set up. Any dependencies of Y will also have to be resolved and so on, until no more objects are needed. The same is true for external state setup - any related side effects and objects need to be resolved as well.

During dynamic inspection Pythoscope tracks relations between objects and side effects that affect them. Having that information is crucial to resolving dependencies during test generation. Dependency resolving rules used during test generation has been presented below.

- When a built-in composite object (like a list, a tuple or a dictionary) is required, all its elements are required as well.

- When an instance of a user class is required, call to its constructor (usually a method called `__init__`) is required as well.
- When a call (function and method calls are treated equally here) is required all its arguments are required as well.
- When any kind of mutable object (be it built-in or user-defined) is required all side effects affecting it are required as well.
- When a side effect is required, all objects it affects are required as well.

Last rule may seem redundant, but it is not: a single side effect may affect more than one object, in which case the rule will bring the missing object into the list of dependencies.

Test dependencies are at a heart of a problem of developer testing in general, not only in the context of automatic test generation. Gerard Meszaros names the problem simply *Hard-to-Test Code* and lists highly coupled code as the most prominent cause[6]. Michael Feathers gives examples of highly coupled code and shows how it affects difficulty of writing unit tests[7]. He discusses C++ and Java programming languages, and while not all of his examples apply to Python, some are more general. They have all been listed below.

Irritating Parameter is a parameter of a type that is very hard to instantiate or shouldn't be instantiated in a unit test, e.g. because it is slow or causes unpredictable side effects (or both).

Hidden Dependency is a dependency that is not on parameter list of any function or method explicitly called in a test case, but is required for its proper execution.

Construction Blob is case of a constructor or a call that uses a large number of parameters, to a point of becoming unreadable. Although Pythoscope can handle 50 parameters just as easily as 5, there is a problem of readability of generated test cases. Refer to section 5.10 for details.

Irritating Global Dependency is similar in scope to *Hidden Dependency* and notes a place where code uses a global variable or resource, which doesn't work well in a testing environment. Michael Feathers lists an example of a *Singleton Pattern* [10] and problems it causes during testing.

[6]Details on *Hard-to-Test Code* can be found in [17], p. 209.

[7]Examples of highly coupled code are described in [7], chapter 9.

Horrible Include Dependencies are specific to C++ and the problem does not appear in Java or Python.

Onion Parameter is a parameter that has a lot of (and often deep) dependencies - to create it a large number of other objects need to be created first. Just as the *Construction Blob* the *Onion Parameter* makes the generated test cases much less readable.

Aliased Parameter describes a case of a class where doing the *Extract Interface* refactoring [9] would create superfluous number of interfaces, which is bad design. *Subclass and Override Method* refactoring [7] is presented as an alternative that avoids the problem of too many interfaces. Since in Python *Extract Interface* is not needed to break dependencies, this technique also does not apply.

5.10 Readability concerns

As mentioned in one of the previous sections, highly coupled code is a problem for Pythoscope because of a loss in readability of generated test cases. Setup code for objects can easily span multiple lines, dominating test contents. There are three main properties of objects that affect readability of their setup code, listed below.

Number of dependencies This the number of other objects the constructor requires. As the list of parameters gets bigger, readability drops significantly. Current version of Pythoscope doesn't limit number of parameters or setup code needed, and in effect may generate convoluted test cases. The canonical solution to this problem is using the *Creation Method* pattern [17]. An object with a high number of dependencies is called a *Construction Blob* (introduced earlier in this chapter).

Depth of dependencies If a constructor parameter requires other objects to be created, which in turn require other objects, and so on, it is a case of a deep dependency structure. One way of breaking it is by cutting it at the first level and introducing a *Test Double*. Current version of Pythoscope doesn't use any test doubles, making *Onion Parameter* a serious problem for readability of test cases it generates. Other way to mitigate the problem is by a proper naming method. Pythoscope names objects both for validity and readability reasons, what in some cases works around problems caused by the *Onion Parameter* pretty well.

Number of calls on dependencies When a *Test Double* is used it needs a strict specification of behavior to adhere to. If this specification is complicated, the test double suddenly starts to occupy a lot of space in the test and makes the whole test case harder to read. Since Pythoscope doesn't use test doubles it doesn't have this problem, yet it is an issue worth considering, especially since it forces a certain amount of heuristic reasoning when deciding between generating a test with a test double and generating a test with a real object.

Readability doesn't only concern contents of test cases, names of test classes and methods are also important. A good name should suggest a behavior that is expected from the SUT and tested in this particular test case.

Pythoscope generates name of test classes based on the name of an application class or function. For example, all test cases for class `UserAccount` will be gathered in a test class called `TestUserAccount`. Test cases for function `execute_transaction` will be contained inside a test class called `TestExecuteTransaction`.

Names of test methods are much more descriptive. In general, Pythoscope tries to keep the names short yet unique and informative. For example, a test case that contains the following assertion:

```
self.assertEqual(11, plusone(10))
```

will be named `test_plusone_returns_11_for_10`. That can be read almost verbatim as "test that a function called `plusone` returns 11 when 10 is given as an argument". Similarly, longer chains of assertions give longer test names, e.g. this chain:

```
user_account = UserAccount(0)
self.assertEqual(None, user_account.deposit(100))
self.assertEqual(None, user_account.withdraw(50))
```

gives a test case named `test_deposit_and_withdraw_after_creation_with_0`.

Chapter 6

Pythoscope architecture

This chapter delves into technical details of Pythoscope's implementation. Methods of capture, analysis and test generation for code with side effects has been described in detail in previous chapters. Here a bigger picture will be presented - a view of the Pythoscope project itself, without the focus on side effects. Requirements for the project are listed first, what is followed by a list of dependencies and a description of each part of Pythoscope: from the two inspectors, through the middle layer and ending with the test generator. This chapter is closed by a discussion of a few chosen areas in Pythoscope that were challenging in implementation, including topics such as portability and performance.

6.1 Requirements and design principles for Pythoscope

Pythoscope development was never driven only by the need to add new features, instead a concrete set of qualities were always kept in mind. Those qualities defined not only the architecture but also usage scenarios for Pythoscope.

First of all, Pythoscope should support incremental usage. A programmer should be able to focus her testing efforts only on a subset of the system and after that still be able to address other parts when the need arises. Moreover, a programmer should be able to extend test suites generated by Pythoscope without any configuration effort - modification and running of test cases should work the usual way. When the system under inspection changes, Pythoscope should be able to detect and isolate those changes and generate only enough test cases to complement the already existing test suite. More generally, Pythoscope should be usable both for generating big test suites for the whole system all at once as well as iterative case-by-case test generation based on evolving behavior of the system. The hope is to make a tool like a text editor or a test runner, which will be used by a programmer every day, and not a single-use toy that will be useless once the test cases are generated.

Second, Pythoscope should support a wide range of Python versions. Legacy systems, which Pythoscope addresses, often have a lot of dependencies and interpreter upgrade is not always possible. By requiring a more recent version of Python, Pythoscope would be out of reach for those use cases. Early on, a decision was made to support Python 2.3 and higher. Since dynamic inspection procedures are run on the same process as the legacy code itself, this means Pythoscope also has to be written in Python 2.3. Forward compatibility is also of concern, although much easier to achieve than backward compatibility. Pythoscope supports Python versions up to 2.6.

On the other hand, complexity of the dynamic inspector implementation forced Pythoscope development to focus on a single interpreter implementation. A canonical Python implementation, CPython in version 2, has been chosen, as the most widespread one [16]. Support for Python 3, Jython, IronPython or PyPy, while possible in the future, is not a current focus.

Interpreter version is but a single piece of a bigger issue of dependencies management. In more general terms, Pythoscope tries to keep its own dependencies as simple as possible, so it can be installed and used on a wide variety of systems without much work on the developer side. Right now the only two dependencies of Pythoscope are the Python interpreter and the ctypes library [31]. While Pythoscope supports setuptools [46], it doesn't require them for installation.

Of course dependencies for developing Pythoscope are wider. Here, one needs setuptools [46], nose [36], mock [34] and docutils [32], as well as all supported interpreter versions installed, so a whole unit test suite can be run across all supported versions.

Last two design principles relate more closely to the Pythoscope's architecture and coding style than the previous ones. Modularity is a reason for inspector-generator split and a more general tendency to maintain loose coupling. This style of programming hopefully encourages reusability and allows other projects to borrow Pythoscope's code. For example, the inspector could be used as a backend for a Python IDE, providing method signatures, usage examples and debugger aids to a programmer. The generator could also be reused, for exampe in a model checking tool [5]. The second and related principle is simplicity. When achieved it greatly eases porting and supports experimentation, which is crucial in any research project.

6.2 Pythoscope dependencies

In order to run the Pythoscope application one needs to fulfill five dependencies, listed below.

- CPython interpreter
- ctypes library
- lib2to3 library
- bytecode tracer library
- pythoscope package

6.2.1 CPython interpreter

Pythoscope runs only on the canonical Python implementation called CPython[1]. CPython is implemented in C and runs on Windows, Linux/Unix and Mac OS X platforms[2]. CPython was the first implementation of the Python language and is still the most popular one [16]. Since its first public release in 1991[3] it went through many revisions, up to the re-licensed and very successful 2.x line and finally to the new, revamped and backwards-incompatible line 3.x. Although a stable version of a 3.x line has been available since December of 2008, 2.x line of interpreters is still in widespread use.

Pythoscope supports CPython versions 2.3 through 2.6. Compatibility issues related to interpreter version are described on page 74.

6.2.2 Ctypes library

Ctypes [31] is a foreign function library[4] for Python. It allows calling functions in DLLs or shared libraries directly from Python, without a need for an intermediary C extension. It also allows to read and write binary structures, dereference pointers and do other unsafe memory operations, normally unavailable at Python level. Ctypes library has been a part of CPython standard library since version 2.5 and is also available as a standalone package, supporting versions of Python 2.3 and higher.

[1]Per Python glossary [40].

[2]As listed on the Python home page [41].

[3]The release was numbered 0.9.0 and was posted by Guido van Rossum to alt.sources usenet group, as mentioned in the HISTORY document in CPython source code.

[4]Also often called a foreign function interface (or FFI).

Pythoscope uses this library to access internal interpreter data, normally hidden from a running program. Structures read by Pythoscope this way include a wrapper objects[5], generator objects[6] and frame objects[7].

6.2.3 Lib2to3 library

Lib2to3 was initially a library intended for a very specific purpose. Written by Guido van Rossum, it formed a back end for the 2to3 tool [26] which purpose is to ease transition from Python 2 to the new and backward incompatible version 3. 2to3 can automatically convert Python 2 source code into Python 3 code. Its ability to read and write back slightly modified Python code proved to be very useful in the context of test generation.

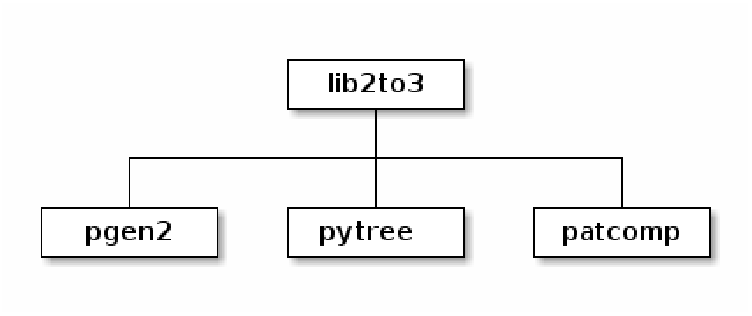


Figure 6.1: Structure of the lib2to3 library

The core of a lib2to3 implementation is the `pytree` module. In a form of `Node` and `Leaf` class definitions it defines the structure and behavior of the abstract syntax tree (AST for short). This tree can be traversed in either breadth-first or depth-first manner using `pre_order` and `post_order` methods respectively. It can also be modified by appending or replacing existing nodes using `set_child`, `insert_child` and `append_child` methods of `Node`. A string can be produced from the AST by simply calling the `str` function on it.

The opposite conversion, from the source code to the AST, is achieved by a parser, implemented in the `pgen2` subpackage. The parser follows a

[5]Wrapper objects are created for visibility of a C functions at Python level, e.g. reference to a built-in function will point at a wrapper object. More details can be found in [48], chapter 3 "Data model".

[6]Generator objects are objects returned by generator functions. For more details see [48], chapter 6 "Simple statements", section on the `yield` statement.

[7]For a description of frame objects see [48], chapter 3 "Data model", section on frame objects.

grammar, defined in a separate file, named `Grammar.txt`, taken directly from the CPython interpreter's repository. In Pythoscope, a grammar of Python 2.6 is used.

Lastly, lib2to3 defines a query language that can be used to find and extract certain pieces of the syntax tree. Module responsible for that task is called `patcomp`, a shorthand for "pattern compiler".

6.2.4 Bytecode tracer library

CPython has a built-in method of tracing application code, allowing the tool builders to implement debuggers and profilers[8]. Bytecode tracer library extends possibilities of CPython to trace individual bytecode instructions. Developed in 2010 as a part of this thesis, it soon has been merged into Pythoscope. While still usable as a separate tool it really shines when used in combination with traditional tracing methods.

6.2.5 The pythoscope package

The pythoscope package is divided into four major parts: the inspector, the generator, the middle layer between them and a command-line interface. **The inspector** is responsible for gathering information about an application – its logical structure – layout of modules, class definitions, function signatures, as well as its dynamic behaviour. **The generator** uses that information to generate test cases and merge them with an existing test suite. **Intermediate data** is stored as a collection of in-memory Python objects, serialized to disk with the pickle module [38] when necessary. A complete tool is available to the developer through a **command-line interface**.

All of the techniques used by the inspector to gather information may be divided into two categories, depending on whether they execute the application code or not. **Static inspection** refers to analysis performed without execution of the application code. This basically means gathering information from the source code of an application. To make the process easier, source code is first converted to more convenient representations. Pythoscope uses lib2to3 parser and CPython compiler to generate abstract syntax trees and bytecodes respectively. While the inspection of ASTs yields a lot of information regarding structure of the code (i.e. how does it look like), bytecode analysis may additionally provide hints about dynamic characteristics of the code (i.e. what it does). Unfortunately, due to the dynamic nature of Python[9] even static bytecode analysis doesn't yield enough information to

[8]See section 4.1 on page 28 for a detailed description of *tracing hooks* in the interpreter.

[9]Described in section 1.3 on page 7.

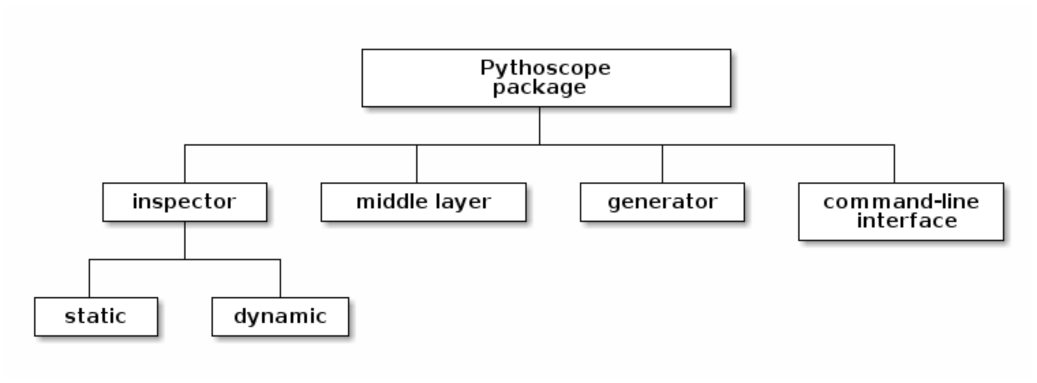


Figure 6.2: Structure of the pythoscope package

generate interesting test cases.

This is where **dynamic inspection** has to be applied. By running application code in a controlled environment, different kinds of information can be gathered, from program call graphs [23] to input-output pairs of all functions' and methods' invocations, exceptions raised, as well as other operations normally hidden by the CPython bytecode interpreter. That kind of detail is necessary to generate a proper unit test case, with the necessary setup, precise assertions and appropriate teardown, as described earlier in the chapter on test generation[10]. Implementation of the dynamic inspection is very tighly bound to the underlying language implementation. Mere difficulty of implementing a dynamic inspector as well as amount of information that can be gathered that way depends in great deal on tracing capabilities of a Python interpreter. Thus, while the rest of architecture notes can be reused in other projects similar in scope to Pythoscope, the section on dynamic inspection is inescapably very focused on CPython.

The middle layer is what holds all information gathered at the inspection stage. Most of the information is stored in domain-specific objects, defined by Pythoscope, which map to structure and behaviour of the application under inspection. At the top, we have a `Project` class which holds references to `Module` objects, which in turn reference `Classes` and `Functions`. This static view is enriched with information about traced behaviour, like `FunctionCalls`, `MethodCalls`, `GeneratorObjectInvocations` and so on, each of those objects holding information about inputs, outputs, exceptions and side effects. Combining static and dynamic view of the project allows to derive properties of the code, and ultimatelly use that knowledge to generate test cases.

[10]See chapter 5 on page 41.

Test generation based on data stored in the middle layer is exactly what **the test generator** is responsible for. For each small subset of gathered information a unit test case can be generated. Dependencies of each test case are resolved, environment setup code is included and becomes a part of the test case. For each output and change that need to be tested the generator spawns an assertion. Finally a teardown code is included, ensuring proper separation of test cases. From some perspective, the test generator is a compiler that converts inspection information into Python test code. This point of view is especially useful when implementation of Pythoscope's generator is considered, as many canonical compiler patterns are present there.

6.3 Static inspector

The main role of the static inspection is to prepare ground for later dynamic inspection. Logical structure of the program is analyzed and stored first by the static inspector, what is followed by the dynamic inspection step, described in the next section. But before that, static inspection techniques will be described in detail.

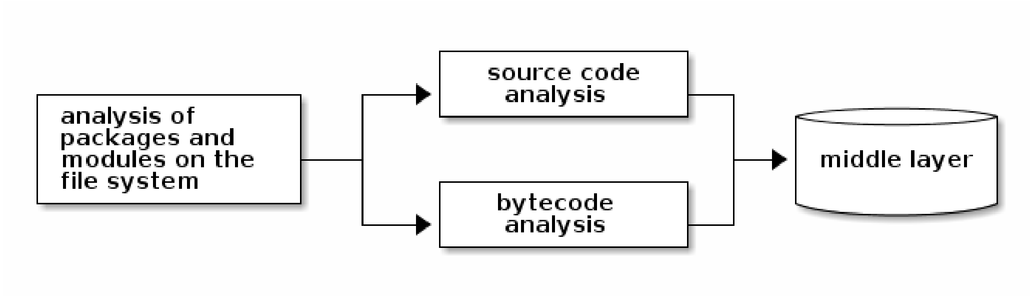


Figure 6.3: Static inspection process

6.3.1 Analysis of packages and modules on the file system

Logical structure of a Python program is defined by layout of source code files on the file system as well as by the contents of those files. *Aptus*, a sample medium-sized Python application is used as an illustration[11]. *Aptus* is a

[11]Full source code of the *Aptus* program can be obtained from `http://nedbatchelder.com/code/aptus/` under the MIT license.

Mandelbrot set viewer and renderer, written by Ned Batchelder in Python. It uses three popular Python libraries: Numpy for calculations, PIL for image rendering and wxPython library for GUI. When compiled and installed, *Aptus* directory structure looks like this:

```
aptus/                     Top-level package
    __init__.py            Initialize the aptus package
    cmdline.py
    compute.py
    engine.so*
    ggr.py
    gui/                   Graphical user interface subpackage
        __init__.py
        computepanel.py
        dictpanel.py
        help.py
        ids.py
        juliapanel.py
        mainframe.py
        misc.py
        palettespanel.py
        pointinfo.py
        resources.py
        statspanel.py
        viewpanel.py
        youarehere.py
    importer.py
    options.py
    palettes.py
    progress.py
    settings.py
    timeutil.py
    tinyjson.py
```

The `aptus` package contains an initialization file (which must always be called `__init__.py` and be present in the package directory[12]), ten Python modules, one module compiled from C (`engine.so`) and one subpackage (`gui/`), which in turn contains its own initialization file and 13 more Python modules. The logical structure, as visible to the running Python program,

[12]More on module and packages can be found in the chapter 6 of "The Python Tutorial" [44].

follows almost exactly the file system structure. For example, to import the `mainframe` module that resides in the `gui` subpackage, one must use the following import statement:

```
import aptus.gui.mainframe
```

At import time there is more to it than the directory structure of the application itself. Most applications use external libraries (or at least Python's standard library) which reside in a global environment, so in reality all imports go through an importer that dynamically resolves all names. This doesn't matter in most cases though - since all modules are safely contained inside the `aptus` package, the application structure and behaviour of its imports can be predicted by looking at the file system.

All Python modules are discovered by recursively descending project directory tree. All files with `.py` extension are collected and returned. Binary modules (like the `engine.so` from *Aptus*) are ignored, because their structure cannot be inspected as easily as the structure of the pure Python modules. Internal files of version control systems (like `.svn/` directories in case of Subversion) are ignored as well. In the end, Pythoscope holds a list of all Python modules inside a project. Contents of those modules are not inspected directly - instead Pythoscope uses ASTs and bytecode to gather information.

6.3.2 Source code analysis

All source code, especially from the point of view of a programmer, is a sequence of characters. While this representation is easy to modify and transfer, it is not efficient for program evaluation or compilation. Because of that compilers use a *syntax tree* during the semantic analysis phase. Syntax trees contain all information needed to check[13] and compile a given unit of code. They were also shown to be very useful representation for just-in-time compilation inside interpreters, even more so than bytecode [12]. The same principle applies to static inspection performed by Pythoscope.

Pythoscope uses lib2to3 library described earlier[14], but not directly. Two additional layers has been developed: one for traversing ASTs, named `astvisitor`, and the other for constructing and modifying them, named `astbuilder`. Static inspector uses both: `parse` function from the `astbuilder` module is used to convert source code to a syntax tree, while the `ASTVisitor` class

[13]Those checks include but are not limited to checking for type errors and determining variables' binding.

[14]See 6.2.3 on page 58.

from the astvisitor module is used to analyze it. Both modules are also used during test generation, see section 6.6 on page 73 for more details.

During static inspection each file identified as holding Python code, as described in section 6.3.1, is converted to AST and then analyzed. A valid AST is traversed in search of standard building blocks of an application: classes, functions, etc. ASTVisitor class from the astvisitor module, when subclassed, can be used to collect information about elements of a program. Static inspector uses two such subclasses. One, called ModuleVisitor is used to identify program elements at the module level, while the ClassVisitor is used to identify elements at the class level.

Elements of the program structure identified at the module level have been summarized in table 6.1.

At the class level, where ClassVisitor is used, all but the method definitions are skipped. This means inner classes are currently ignored by Pythoscope as an implementation detail of a class to be used internally by its instances.

As the AST is traversed and elements are identified, Pythoscope creates its own tree of objects. Details of objects created at that point can be found in section 6.5 on page 70. The important thing is that, once static inspector finishes its work on a file, Pythoscope has a reference to an instance of a Module class that holds all information about the structure of a module, what includes, but is not limited to, a list of all defined classes and functions.

6.3.3 Bytecode analysis with CPython compiler

There are cases where AST inspection is either insufficient or too convoluted, at least when compared to bytecode inspection. In those cases Pythoscope uses CPython compiler facilities to collect the necessary information.

An example of such area is a case of differentiating between normal function and generator functions[15], implemented in Pythoscope by a function named is_generator_definition from the inspector.static module. In Python, a generator definition can be identified by the presence of the yield expression. This means that this code defines a normal function:

```python
def function():
    return 42
```

while the following defines a generator:

[15]Generator functions are described in detail in [48], chapter 6 “Simple statements”, section on the yield statement.

Table 6.1: Elements of a program at the module level

Element	Example	Description

class definitions	`class AClass(object):`
	`    pass`
		Three elements of a class are identified and remembered by Pythoscope: its name, its ancestors (superclasses) and a list of methods.

function definitions	`def a_function(arg1, arg2):`
	`    return 42`
		Besides function name and a list of parameters' names Pythoscope keeps a reference to the function body for later inspection. Inspection of the function body may reveal, among other things, whether a function is pure or not.

import statements	`import os`
	
	`from sets import Set`
		Imports are parsed, so that when new ones are added by the test generator, duplicates can be avoided.

`__main__` snippet	`if __name__ == '__main__':`
	`    unittest.main()`
		The documention for unittest [49] suggests the use of a `__main__` snippet and in fact that is a very common way to run tests written in unittest, as can be shown on the example of CPython standard library. Pythoscope uses that snippet, if present in a file, as a reference point for adding new test classes. The idea is to keep that snippet at the very bottom of a file at all times.

```python
def function():
    yield 42
```

While it would be possible to traverse the syntax tree of the function body to find a `yield` expression, a more straightforward method is to compile the function definition and inspect attributes of the generated bytecode. The attribute useful in this case is called `co_flags`. When bit `0x20` of this attribute is set the function is a generator.[16]

6.4 Dynamic inspector

From the test generation perspective dynamic inspection is the phase where most data is gathered. Objects being created, functions invoked, exceptions being raised and caught, locks acquired and freed, files created and extended - all of those actions define a behavior of a program, which forms a base for test generation. The job of a tool like Pythoscope is to carefully intercept and interpret those actions in their current dynamic context, so that later each single piece of behavior can be replayed in a unit test.

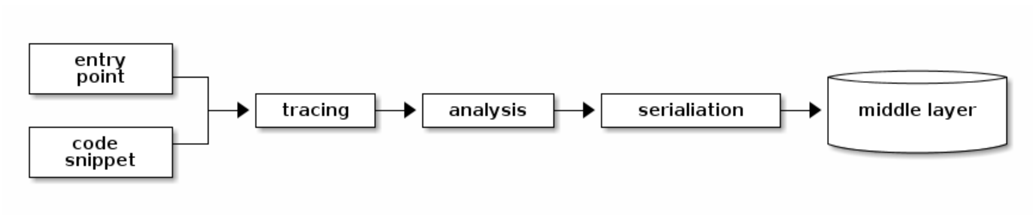


Figure 6.4: Dynamic inspection process

6.4.1 Ways to invoke dynamic inspection

There are two main ways to invoke the dynamic inspector in Pythoscope: through an *entry point* or via a use of a special *code snippet*. First method allows Pythoscope to invoke dynamic inspection when it needs to, what makes it similar to the way static inspection is done. Second method puts more power into the hands of a programmer: dynamic inspection is invoked when the programmer manually runs her application. Both methods give the same end results in terms of amount and precision of gathered information.

Point of entry is a small Python module that calls the rest of the application code that needs to be inspected. If there is a main function in the

[16]Per [48], chapter 3 "Data model", section on code objects.

application, a call to it would make a good entry point. Existing high-level regression tests are also good candidates for becoming entry points. Pythoscope imports entry points from directory `.pythoscope/points-of-entry/`. A point of entry should essentially be a Python script, that when invoked manually calls into the system or its part. The reasons for the existence of entry points are twofold. First, they provide Pythoscope with a working starting point for further dynamic inspection. Second, they are a signal from a developer that wrote the entry point that the code in question is safe, i.e. it won't destroy important data, initiate a production deploy or something similarly disastrous.

Code snippet constitutes an easier and quicker alternative to entry points. All the programmer needs to do is to find places in the application code where it begins execution and the place it ends. At the top of the first application module to be imported, which is usually the first place to be run by the Python interpreter, a programmer needs to place the following snippet:

```
import pythoscope
pythoscope.start()
```

It imports pythoscope and initializes the dynamic inspector. All code executed after that line will be traced and remembered by Pythoscope. As the application is running, Pythoscope analyzes and keeps all information in memory. Thus, before the application exists, that information has to be saved into disk. A programmer must put in another snippet, this time at the place where the application ends execution. Most of the time this place is simply at the end of the main script file, but sometimes it is a line before a call to `sys.exit`.

```
pythoscope.stop()
```

Once those two snippets are placed, a programmer may prepare the environment and run her application in the usual way. All interactions with the application are intercepted by Pythoscope and saved to disk right before application's exit. After that a programmer may use the pythoscope tool to generate test cases based on just gathered data.

It is important to understand the distinction of context between both approaches. With points of entry, a user runs the pythoscope command, which in turn may (if it needs to) import and call into the entry point's code. Pythoscope tool is in charge here. In fact, many entry points may be invoked one after another (or the same entry points multiple times) during a single run of a pythoscope tool. On the other hand, when using the code snippet, it is the application under inspection that is running, and pythoscope acts as a library that merely installs its hooks after the application starts and saves the results right before the application ends.

6.4.2 Tracing calls and bytecodes

The layer that is closest to the interpreter during dynamic inspection is the tracer. The tracer is the receiver of events originating from the interpreter. All the tracer does is enriching trace data with additional information, to be passed and interpreted by higher layers of Pythoscope. See the next section to learn more.

Implementation of the bytecode tracer for CPython has been described in *Capture and analysis of side effects* chapter, in section 4.8. History and dependencies of the bytecode tracer has been mentioned in section 6.2.

6.4.3 Object and call analysis

On top of tracer an analysis layer has been constructed. From the stream of events reported by the tracer it serializes objects, builds call graphs and handles side effects. This is the point where raw tracer data is converted into domain objects, like `FunctionCall`.

A base for correct inspector work is the static view of the project, containted within the Project tree[17]. Based on raw execution data provided by the tracer the inspector finds an object that is acted upon and annotates it with just aquired dynamic information. Below follows an example of a tracer event being recognized and saved as a `FunctionCall` object within the Project tree. Points 1 through 5 are handled by the tracer, while points 6 through 11 are responsibility of the inspector. The whole process has also been illustrated on figure 6.5.

1. 'call' event is reported by the Python interpreter with the reference to the current frame
2. the tracer makes sure that the event is a function call (`Tracer.should_ignore_frame`)
3. tracer extracts function name from the frame (`frame.f_code.co_name`)
4. tracer extracts call arguments information from the frame (`input_from_argvalues`)

[17]Refer to section 6.5 on page 70 for details.

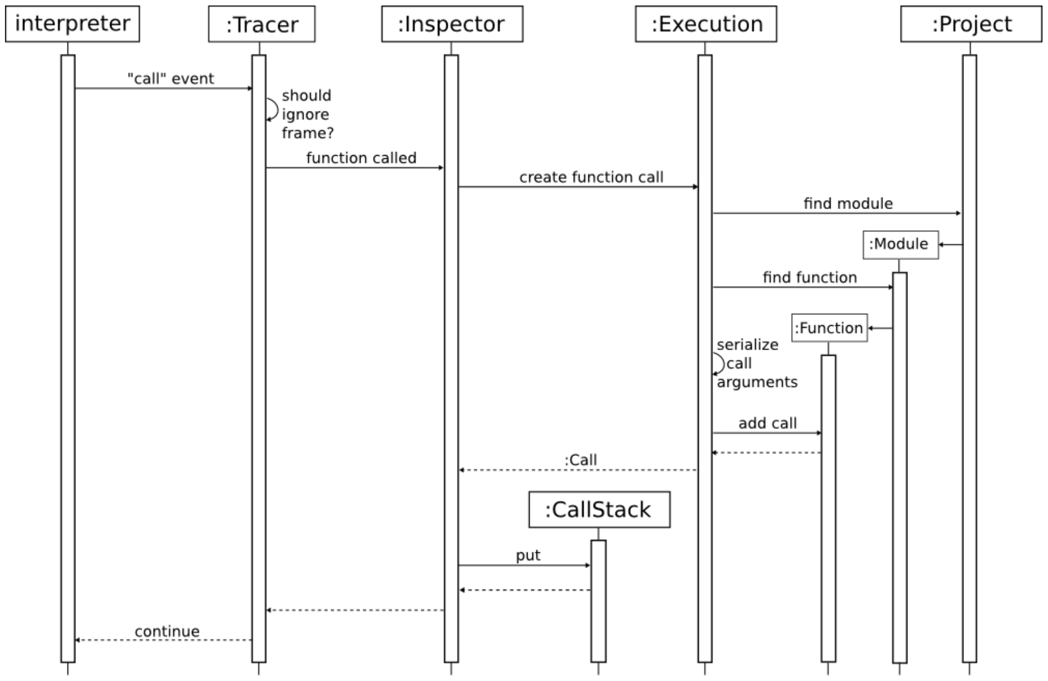


Figure 6.5: Call analysis example

5. tracer calls `function_called` on an Inspector object with function name, arguments, code and frame object as arguments

6. inspector passes all arguments to `create_function_call` method of an `Execution` object

7. `Execution` object extracts filename from the `code` and looks for a corresponding `Module` in the Project tree

8. if the `Module` is found, its contents are used to look for a `Function` object corresponding to the name that was passed

9. if the `Function` is found, a `FunctionCall` object is initialized and attached to the rest of the Project tree

10. arguments of the function call are serialized using `serialize_call_arguments` method of the `Execution` object

11. inspector puts the call on the `CallStack`, what eventually builds a full call graph

This example highlights an important feature of the `Inspector` class: it is very thin and delegates most of the work to other classes.

A `CallStack` object tracks calls, returns and exceptions, forming a virtual call stack that corresponds to the interpreter's call stack. The difference between the two is that interpreter "forgets" about previous stack levels that were already unwound, while the `CallStack` keeps all of this information. In effect, a complete call graph is constructed. Nodes of this graph are ordered, so that it is possible to determine order of all calls.

An `Execution` object represents a single run of an application, no matter the method[18]. It holds a reference to the Project tree and queries it when calls are made and objects are serialized. In this way static information greatly simplifies and channels effort during dynamic inspection. Static information acts as a reference point for all information gathered during runtime.

The `Inspector` object is discarded once the inspection is done and effects of the run are contained within the `Execution` object. So the `Execution` object not only registers the events, but also is responsible for storing and browsing through them. Each `Execution` object forms a net of calls and dynamic connections built on top of the static Project tree. Those interweaving nets are then used by the test generator. More on the dynamic section of the Project tree can be found in section 6.5 on page 70.

6.5 The middle layer

Data gathered by the inspector is stored in a tree-like structure, serialized to disk with the pickle module [38]. It has been presented on figure 6.6. Part of the tree that is constructed during static inspection is in white, while gray has been used to mark elements that are created only during dynamic inspection.

The root of this object tree is an instance of a `Project` class. Besides some bookkeeping data it contains a set of references to `Module` instances, which represent Python module of the inspected application. There is no artificial distinction between application modules and test modules. All modules can contain both application code and test cases and Pythoscope was constructed to work with this arbitrary arrangement. Each module contains a set of domain objects, each representing a single element of a program: a class, a function or a test class.[19] Classes are containers for methods.

`Project` delegates management of `CodeTree` instances to a `CodeTreesManager` instance. Refer to section 6.10 for details and notes on performance. Each `CodeTree` instance corresponds to exactly one `Module` instance.

[18]For description of different methods of invoking the dynamic inspection see 6.4.1.

[19]See table 6.1.

That structure information is later enhanced with dynamic data. At the beginning of dynamic inspection `Execution` instance is created to act as a root for all objects created during this round of tracing. This way Pythoscope can gather and manage multiple execution contexts and differentiate between them without problem. During dynamic inspection two concurrent processes occur: calls are recorder on a call graph and objects are serialized and laid out on a timeline[20]. Those calls and objects are at the same time hooked up to the `Project` tree, so the test generator has an easier time finding and using them. For example, a `Function` object contains references to all calls that were registered and `UserObject`s are connected with `Class` instances they were instantiated from.

The most basic unit of an execution is a `Call` object. Pythoscope identifies four different types of calls in Python: function calls, method calls, generator invocations and calls into C[21]. Each call includes reference to the caller (which is an another `Call`), serialized call arguments, references to other calls made inside this one (subcalls), list of side effects made by this call, and finally either an output value or a raised exception (serialized in either case).

As just there are many types of `Call`s there are many types of serialized objects. Figure 6.6 shows inheritance hierarchy for `SerializedObject` and its derivatives. Only `UserObject` is connected with the rest of the tree - other types represent types built into the interpreter and their usage during test generation doesn't require any context information in regards to the rest of the application.

Although filled-in separately, both models (the static model and the dynamic model) share structure which is essential for the generator work to be effective. Only combination of both inspection sources gives good results.

[20]Timestamping objects is crucial for later test generation, refer to section 5.8 on page 48 for details.

[21]There is also a possibility of calls Pythoscope can't classify - they are saved as `UnknownCall`s and are not bind to any other middle layer structure. They presence is motivated only by the need to not leave any holes in the call graph.

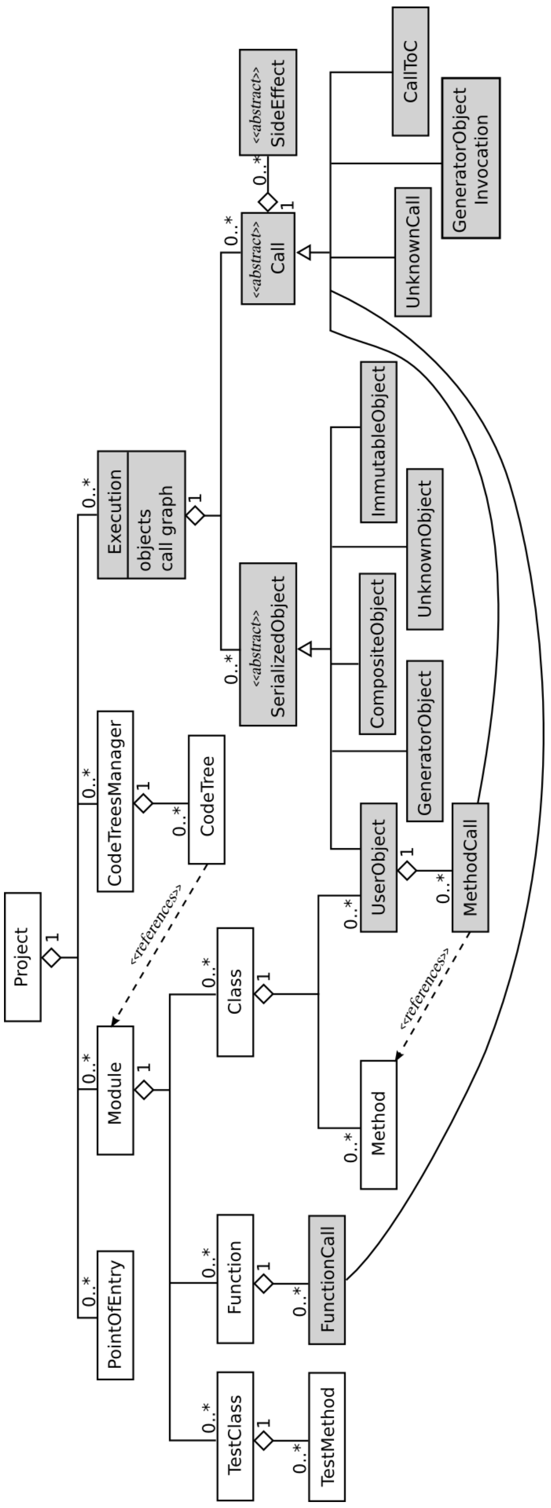


Figure 6.6: Middle layer class diagram

6.6 Test generator

This section describes details of the test generator implementation. For a high-level overview of the test generation process, see section 5.8 on page 48.

Test generator is divided into many submodules, of which the most important 9 will be described here. Overall structure of the test generator architecture has been shown on figure 6.7.

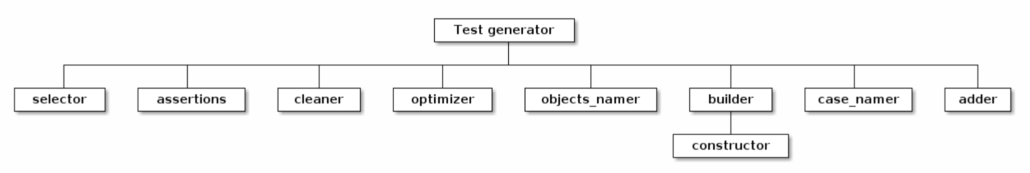


Figure 6.7: Submodules of the test generator

Each submodule takes care of one step in the test generation process, as described in section 5.8. Combined, they convert information gathered by the inspector into valid test cases. All submodules have been described below, in the order they are invoked during test generation.

Selector chooses suitable *testable interactions* based solely on the data contained within the `Project` tree.

Assertions submodule takes each *testable interaction* and prepares a set of assertions that validate its behavior. Many kinds of assertions are supported, including equality assertions, exception assertions and assertion stubs. After that, assertions are expanded into a test case timeline which contains all statements, assertions and possible dependencies of a test case.

Cleaner purges redundant dependencies from the test case timeline.

Optimizer takes the timeline and compresses it even further by combining sequences of operations. For example, instead of replaying side effects on a list, like so: `x = []; x.append(1); x.append(2)` the optimizer compiles that down to a single list creation statement: `x = [1, 2]`. Such optimizations never change semantics of a test case, only improve its readability.

Objects namer annotates the timeline with names for objects that need them. Each object gets a unique yet descriptive name, as an another method of keeping tests readable.

Builder takes the test case timeline and compiles it down to a single `CodeString`, which will constitute the test case body. It uses the *constructor* as its submodule. See also section 6.11 for a further description of the `CodeString` class.

Constructor has a very specific purpose: it compiles a `Call` object into a `CodeString`, in this way turning interactions captured at runtime into executable code.

Case namer is responsible for giving names to newly generated test classes and methods. The problem of naming test cases has been described in more detail in section 5.10.

Adder merges new test cases into the existing test suite. At the end of the code generation process, the final `CodeString` is converted to AST. In this form the generated code can be merged into rest of the test suite: a test method can be made a part of a test class, while a test class can be put into a right place in a module. Lib2to3 is utilized here again, as it allows extending source code of existing modules.

6.7 Compatibility issues

Pythoscope supports Python interpreter's versions 2.3 through 2.6. This imposes certain limits on the way Pythoscope is implemented. First of all, only features present in 2.3 can be used during coding. This means language features like decorators[22], generator expressions[23], conditional expressions[24] or the "with" statement[25] cannot be used. Moreover, any differences between implementations of features that existed since Python 2.3 have to be handled by Pythoscope. One such example is support for sets in the standard library. In Python 2.3 this data structure is supported by a pure Python implementation in the `sets` module. Python 2.4 and higher has a built-in type `set` implemented in C. Old `sets` module is still present, but importing it raises a `DeprecationWarning` in Python 2.6.

[22]More on decorators can be found in "What's New in Python 2.4" document by A.M. Kuchling [15], as well as in PEP-318 [24].

[23]Generator expressions were introduced in Python 2.4 and described in detail in PEP-289 [13].

[24]Conditional expressions were introduced in Python 2.5 and described in detail in PEP-308 [21].

[25]The "with" statement was introduced in Python 2.5, after adding a `__future__` directive to code. Since Python 2.6 the directive is no longer needed and support for "with" statement is enabled by default.

For reasons mentioned the compatibility code is scattered across the whole codebase. Each compatibility code is annotated with a comment describing in detail differences between versions. Functions and types added in later versions of Python that are nevertheless needed in Pythoscope has been backported and gathered in a module called `compat`. It includes functions like `sorted`, `all`, `any` and `groupby` that are useful iteration idioms. Moreover, all of the functions in `util` module are implemented in an interpreter-agnostic way.[26] Even the tracing mechanism can differ between versions. The `tracer` module has two main classes: `StandardTracer` and `Python23Tracer` which is a subclass of the former. `StandardTracer` is used under most interpreters. Because Python's 2.3 tracer reports "exception" events differently a specific version is needed that will work around the differences and make them transparent to the rest of the code base.

To keep Pythoscope working for all those interpreter versions a unit testing suite is kept up-to-date and runnable across all versions during development. Before commiting any change to the code base, a full test suite has to be run on all supported versions of CPython, and only after all compile and pass without errors, the change is accepted.

6.8 Comparison of parsing libraries for Python

Table 6.2 lists available libraries capable of generating and modifying syntax trees for Python code.

After considering all pros and cons of each library, lib2to3 was chosen as the only library capable of generating from a modified AST a source code that would still contain original whitespace and comments. All other libraries discard whitespace and comments, so a full reconstruction of a file based on AST is not possible. Since Pythoscope is intended to be used in an iterative manner, ability to add test cases to existing classes is vital. With lib2to3 Pythoscope can not only analyze code, but also discover existing test suites and extend them with new test cases.

6.9 Performance implications of dynamic analysis

Important aspect of dynamic inspection is the ability to distinguish objects at runtime. Garbage-collected environment without a direct access to memory

[26]An example of such a function is `generator_has_ended`. Interpreter versions 2.5 and 2.6 expose an interface that allowed this function to be implemented fully in Python, but versions 2.3 and 2.4 lacked that interface, forcing an implementation at a lower level, via means of a C extension.

Table 6.2: Abstract Syntax Tree libraries in Python

Library name	Characteristics
compiler module [29]	part of the standard library deprecated does not preserve comments or whitespace
ast module [27]	available only in Python 2.5 and higher does not preserve comments or whitespace
byteplay [28]	external package does not preserve comments or whitespace
peak.util.assembler [37]	external package does not preserve comments or whitespace
lib2to3	external package written in pure Python, which makes it slower than parsers in C, but more portable preserves comments and original whitespace

makes this requirement a challenge.

To illustrate this problem consider an example of a function `append_and_return` presented below.

```python
def append_and_return(x, e):
    x.append(e)
    return e
```

When called in the following context:

```python
alist = []
append_and_return(alist, 1)
```

the tracer will, among other things, catch the following events:

- call of the `append_and_return` function with a list as a first argument and 1 as the second, and
- call to the `append` method on the same list object with a number 1 as an argument.

In order to link those two events together the inspector needs to know that the list passed to `append_and_return` is the same object as the list which `append` is called on. This recognition is crucial to generation of test cases that deal with side effects. In this case, catching a mutation of an object during `append` is necessary to register the side effect.[27] Mutation gets us back to the problem of object's identity.

Python documention describes a function called `id` used to determine identity of objects at runtime:

> Return the "identity" of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.

In CPython value returned by `id` is simply the address of an object in memory. The object's lifetime limitation of this utility function is crucial. It is very likely that a new object will get an ID number of an old, just-garbage-collected one. In context of Pythoscope, this could lead the inspector to assume connections between objects that in reality are distict, which would inevitably lead to generation of faulty test cases.

To overcome this problem, Pythoscope's dynamic inspector has to prevent all serialized objects from being garbage collected. The easiest way to accomplish this is by holding a reference to them until the run is completed. Once the run is finished all references are released allowing the garbage collector to free unused objects.

This technique, while allowing correct operation of the dynamic inspector, has a high memory footprint, and in extereme cases could cause the otherwise working application to crash because of memory shortage. This is a general problem of profiling - observation is never entirely nonintrusive, and for each method of profiling a corner case exists with results which differ depending on whether the program is running with profiling or not.

Higher memory use is not the only side effect of running the inspector, another one is longer execution time. All operations performed by a program need to be recognized and registered, what occupies precious processing time. Time spent in the inspector is time taken away from the application, which means it will run significantly slower. It is not harmful by itself, but becomes important when time plays a factor in the application. For example, a network connection timeout may occur, caused by longer processing time, a situation which would not normally happen.

[27] For a description of mutation as a type of side effect, see 3.1.1 on page 21.

6.10 Efficient management of syntax trees

High memory footprint of the AST structures severly crippled performance of the early versions of Pythoscope. Big projects with lots of modules and objects inside them could spawn big pickle files, which in turn caused very high pythoscope startup times. Even 10 Mb pickle file on a modern computer caused a noticeable slowdown. While pickle is a very friendly storage method, it has two main flaws. First, it is inefficient storage in terms of memory and processing needed for read/write operations, at least compared to a dedicated data file format or any of the popular databases. Second, cherry-picking of data is not possible - each time the whole pickle file has to be read into memory before usage. This is unfortunate for Pythoscope, which uses only a small subset of data at any given time.

Since most of the memory was being taken by lib2to3 ASTs, the solution was to pull them out from the main `Project` object into a separate `CodeTree` objects, which could be pickled and stored separately. This way, the main pickle file could remain small and thus fast to operate on, while ASTs could be written and read on-demand. This improvement made Pythoscope ready to be used on big projects - a memory usage during full static inspection stays within a stable limit, disregarding number or size of modules under inspection. Ryan Freckleton reported [45] a stable memory usage of about 30Mb during Pythoscope run on the Python's standard library modules.

Implementation of that storage scheme lies within the `CodeTreesManager` class. The manager handles `CodeTree` instances through four basic operations. An AST represented by the `CodeTree` instance can be remembered (stored on the file system), forgot (permanently deleted) and recalled (restored from the file system into memory). Moreover, a manager supports a `clear_cache` method, which frees memory of all ASTs currently residing in memory. Currently, only a single AST is ever kept in memory at once. When another one is needed, current one is saved to disk and freed, making room for the new one.

The main pickle file is stored in `.pythoscope/project.pickle`, while the AST pickle files lie inside the `.pythoscope/code-trees/` directory.

`CodeTree` objects are more than mere Abstract Syntax Trees for a module. Beside the tree, instances of this class also contain references to interesting elements of the module, in particular function and class definitions, `__main__` snippet location (see table 6.1) and location of import statements. Thanks to those references it is trivial for the generator to replace a piece of code, like a test class, with its extended version, without affecting the rest of the module. Because of that `CodeTree` instances could be called ASTs with annotations.

6.11 `CodeString`, a basic test case building block

It has been mentioned that abstract syntax trees are a convenient representation for purposes of static inspection.[28] The same is not true from the point of view of code generation. In Python, as well as in other non-homoiconic[29] scripting languages, it is easier to operate on strings rather than on syntax trees. For this reason in Pythoscope code generator does most of its work on strings. Along with the code itself, two additional pieces of information are passed: set of imports needed for the code to run and a flag that denotes whether the code is runnable or just a template. Templates are used when real code cannot be generated - usually because of lack of data. The string, set of imports and a flag are kept together in a `CodeString` instance. Instances of this class behave like strings, but also support special concatenation operations that bear the set of imports and the completeness flag in mind. For example, concatenation of two `CodeString`s also merges sets of their imports, so that the resulting `CodeString` requires all imports that any of the components needed. A concatenation of an incomplete `CodeString` and a complete `CodeString` creates a new incomplete `CodeString`. Through the use of those special operations correctness of the generated code is ensured.

[28]Refer to section 6.3.2 on page 63.

[29]Source code of a program written in homoiconic language is also a data structure in a primitive type of the language itself. This makes it easy for such program to operate on other programs without resorting to intermediary representations, like abstract syntax trees.

Chapter 7

Survey on the quality of generated test cases on the example of five open source projects

7.1 Testing projects

Success of this thesis has been measured by testing Pythoscope on five sample applications. Selection criteria for those projects were the following:

1. A project has to be open source, as Pythoscope works only with source code, and a mere binary is not enough.
2. A project has to be easily installable and work as advertised in documentation. That excludes all packages that would not install on the test machine or ones that had critical bugs (one example of such a bug may be missing a crucial file in the source tarball).
3. Finally, a project has to be without tests. Process of writing code with tests is different and the resulting code looks different than a code written without tests.[4] Since generating tests for legacy systems is Pythoscope's main focus, legacy projects were chosen.

Moreover, care was taken to choose projects from a vast area of application. In particular the author wanted to include at least one project that deals with external APIs and one that does processing-intensive operations. The final selection can be found in table 7.1. To give an idea of relative sizes of each project, source code metrics has been gathered and presented in table 7.2. Lines of code counts exclude empty lines and comment lines.

Table 7.1: Projects chosen for testing

Name	Version	Description
Reverend	r17924	A simple Bayesian classifier
freshwall	1.1.2	GNOME wallpaper changer
http-parser	0.2.0	HTTP request/response parser
isodate	0.4.4	An ISO 8601 date/time/duration parser and formater
pyatom	1.2	A Python library for generating Atom 1.0 feeds

Source code for all of the projects can be downloaded from the Python Package Index, at `http://pypi.python.org`.

Table 7.2: Code metrics for sample projects

Name	Lines of Code	Functions no.	Classes no.	Methods no.
Reverend	936	3	15	101
freshwall	1223	55	18	99
http-parser	411	4	7	33
isodate	925	18	5	24
pyatom	341	3	2	13

7.2 Testing environment

Testing has been performed on a PC box running Ubuntu 10.10 with Python 2.6.

7.3 Testing procedure

The process of testing a single application can be described in six steps:

1. extract the application into a temporary directory
2. configure the package, install any missing dependencies and compile any C modules
3. do one of the following:
    (a) put points of entry that were prepared beforehand into `.pythoscope/points-of-entry` directory OR
    (b) put pythoscope code snippet[1] into the application binary and run the application on a sample data prepared beforehand
4. run pythoscope to generate tests for all application modules
5. execute generated tests and gather coverage information
6. remove the temporary directory with all its contents

Points of entry and sample data were all based on usage examples included in the applications' documentation. This way Pythoscope's capabilities were tested on a real-world scenario, yet not requiring any special preparation from the developer's side.

Number of points of entry and examples for snippet executions for each project were summarized in table 7.3.

Table 7.3: Projects and their testing methods

Project	Number of points of entry	Code snippet executions
Reverend	2	0
freshwall	0	1
http-parser	0	2
isodate	1	0
pyatom	1	0

[1]See 6.4.1 on page 66 for more details on the pythoscope code snippet.

7.4 Results

The results for each application are a set of the following metrics:

- inspection outcome (success/failure)
- test generation outcome (success/failure)
- total number of unit tests generated (including passing, failing and stubbed test cases)
- number of failing test cases
- number of test stubs
- line coverage percent[2] gathered by the coverage plugin for nose test runner[30].

Actual results for the five sample applications have been presented in table 7.4.

Table 7.4: Testing results by project

Project	Inspection	Test generation	Unit tests	failing	stubs	Coverage
Reverend	success	success	35	2	16	38%
freshwall	success	success	236	9	104	58%
http-parser	success	success	22	0	19	27%
isodate	success	success	51	6	31	50%
pyatom	success	success	20	0	7	81%

For comparison, results of the same tests made on the last version of pythoscope before this thesis (which was 0.4.3 released on February 28th 2010) has been presented in table 7.5.

[2]Refer to section 2.6 for the meaning of this metric.

Table 7.5: Pre-thesis testing results by project

Project	Inspection	Test generation	Unit tests	failing	stubs	Coverage
Reverend	failure	n/a	n/a	n/a	n/a	n/a
freshwall	static: success dynamic: failure	success	126	0	126	0%
http-parser	static: success dynamic: failure	success	29	0	29	0%
isodate	success	success	47	11	31	34%
pyatom	success	success	20	1	7	53%

7.5 Sample generated test cases

Three examples of generated test cases will be used to describe some of the characteristics and capabilities of the new Pythoscope, improved in this thesis. First example shows a test stub: an incomplete unit test, second a passing (correct) test case, while the last example shows a failing (erroneous) one together with the test runner's output.

Figure 7.1: Example of a test case stub: http-parser

```
from nose.tools import assert_equal
from http_parser.reader import SocketReader

class TestSocketReader:
    def test_readable_and_readinto_after_creation_with__socketobject_instance(self):
        # _socketobject = <TODO: socket._socketobject>
        # socket_reader = SocketReader(_socketobject)
        # assert_equal(True, socket_reader.readable())
        # assert_equal(3961, socket_reader.readinto(<TODO: __builtin__.bytearray>))
        pass
```

Figure 7.2: Example of a passing test case: pyatom

```
import datetime
from nose.tools import assert_equal
from pyatom import AtomFeed
from itertools import islice

class TestAtomFeed:
    def test_add_and_generate_and_to_string_after_creation_with_entries_equal_None_
    and_kwargs_equal_dict_and_title_equal_My_Blog(self):
        atom_feed = AtomFeed('My Blog', None,
                             author='Me',
                             feed_url='http://example.org/feed',
                             subtitle='My example blog for a feed test.',
                             url='http://example.org')
        dt = datetime.datetime(2011, 4, 27, 10, 15, 29, 201154, None)
        assert_equal(None,
                     atom_feed.add(author='Me',
                                   content='Body of my post',
                                   content_type='html',
                                   title='My Post',
                                   updated=dt,
                                   url='http://example.org/entry1'))
        assert_equal([u'<?xml version="1.0" encoding="utf-8"?>\n',
                ..., u'</feed>\n'],
            list(islice(atom_feed.generate(), 23)))
        assert_equal(u'<?xml version="1.0" encoding="utf-8"?> ... </feed>\n',
            atom_feed.to_string())
        assert_equal(dt, atom_feed.updated)
```

Figure 7.3: Example of a failing test case: isodate

```python
from nose.tools import assert_equal
from isodate.isotzinfo import build_tzinfo
from isodate.tzinfo import FixedOffset

class TestBuildTzinfo:
    def test_build_tzinfo_returns_fixed_offset_instance(self):
        assert_equal(FixedOffset(-4, 0, '-04'), build_tzinfo('-04', '-', 4, 0))
```

Test runner output after running this test case:

```
======================================================================
FAIL: test_isodate_isotzinfo.TestBuildTzinfo.test_build_tzinfo_returns_fixed_offset_instance
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/local/lib/python2.6/dist-packages/nose-0.11.1-py2.6.egg/nose/case.py", line 183,
      in runTest
    self.test(*self.arg)
  File "isodate/tests/test_isodate_isotzinfo.py", line 23,
      in test_build_tzinfo_returns_fixed_offset_instance
    assert_equal(FixedOffset(-4, 0, '-04'), build_tzinfo('-04', '-', 4, 0))
AssertionError: <FixedOffset '-04'> != <FixedOffset '-04'>
```

7.6 Types of side effects handled by Pythoscope

Side effects described in chapter 3 have been implemented in varying degrees. This section will touch details of that implementation. All of the following has been summarized in table 7.6.

Mutation of built-in types has been completely solved in term of Pythoscope mechanisms, but the actual support has been implemented only for operations on lists. Adding support for a new data type will require only declarative descriptions of all methods given data type defines.

Instance variable rebiding is handled properly both during dynamic inspection and test generation. Problem with tracking dependencies mentioned in the previous section is a more general one, as it applies to all kinds of references, including globals and class variables, thus leaving assessment of instance variable rebiding support at "full".

Similarly global variables read and rebiding is fully supported. As described in section 5.3 on page 44, global variables accesses are mapped to an appropriate setup and teardown code.

Class variables are inspected properly, just like instance variables, but currently Pythoscope misses the code needed for generation of test cases that include class variables manipulation. This may cause Pythoscope to generate failing test cases. Thus the support for class variables read and rebiding is only partial at this point.

Just like Pythoscope requires custom code to support new data types, the situation is the same for keyboard input and file system access. Currently the support for those functions is missing from Pythoscope.

Terminal output has a partial support, because all `prints` are inspected, but that information is not used in Pythoscope during test generation.

Table 7.6: Types of side effects handled by Pythoscope

Side effect	Support in Pythoscope
Mutation of a built-in type	Partial (not all types)
Instance variable rebinding	Full
Global variable read	Full
Global variable rebinding	Full
Class variable read	Partial (faulty generation)
Class variable rebinding	Partial (faulty generation)
Keyboard input	No
Terminal output	Partial (only inspection, no generation)
File system access	No

7.7 Conclusions

Results presented in chapter 7 indicate that the method of generating test cases based on a combination of static and dynamic analysis described in this thesis is capable of generating tests that handle most kinds of side effects.

The new bytecode tracer (first goal of this thesis, as per section 1.5.1) and its usage inside Pythoscope doesn't cause problems during inspection and generation. Despite its highly experimental status Pythoscope seems to behave rather stable across many different types and sizes of code bases. In the experiment Pythoscope handled without a flaw both pure Python code (like *pyatom*) and applications that mix Python and C code (like *http-parser*).

Side effects extension to Pythoscope's inspector and generator (second and third goal of this thesis, as per sections 1.5.2 and 1.5.3) has been partially implemented - problems and implementation details has been described in the previous section 7.6.

Finally, **a model of analysis for side effects in Python** (last goal of this thesis, as per section 1.5.4) has been described in detail in chapter 4. The model is very specific to CPython, so future improvements may include extending it to other Python implementations, or even other dynamic programming languages, like Ruby or Perl.

Improvement in generated test cases quality is clearly visible by comparing the two sets of results: for the new Pythoscope (table 7.4) and a version before this thesis (table 7.5). More real test cases (as opposed to stubs) are generated, what directly affects the coverage values which turn up higher than before. Dynamic inspection is more reliable: the old Pythoscope failed dynamic inspection on 3 out of 5 projects tested, while the new one correctly inspected all five.

Coverage value and ratio of stubs to real test cases vary greatly between projects (in particular coverage varies from 27% to 81% in this experiment). Careful study showed that this value is strictly dependant on the quality of point of entry and the test data that was used during inspection. Points of entry or test data that cause the application to traverse most of its code branches will generate high coverage values and a high number of real test cases. On the other hand, a point of entry that invokes only a subset of the whole application will result in a low coverage value of the generated test suite.

Standard library objects require extension on the Pythoscope's side itself. If the support is implemented they are handled well, otherwise Pythoscope puts stubs in their place. Two sample test cases quoted in section 7.5 provide an explanation. The pyatom sample test case, on figure 7.2, shows how Pythoscope properly captured and was able to recreate a `datetime` object, from the `datetime` library. The object is created on line 14. At the same time, http-parser sample test case, on figure 7.1, shows how Pythoscope behaves when support for a given standard library object is not implemented. Line 6 shows a stub constructor for the `socket` object. In this case developer is forced to complete the test case by herself.

Another problem discovered during testing was a difficulty of tracking attribute bindings during test generation. In Python attributes bindings change dynamically, what makes it hard to track object references that can be used during testing. In particular, in a generated test, references to objects that needs to be checked may be lost during execution, and consequently those objects may be garbage-collected. Ideally Pythoscope should resolve all those dependencies and inject its own references to those objects into a generated test. At that time this feature is not implemented, causing Pythoscope to occasionally generate failing test cases.

Last issue with the generated tests was related to the way Pythoscope handles equality of Python objects. Python standard library objects have a well-defined equality semantics. In general, two different objects that are the same internally (e.g. two lists that contain the same elements) are equal. Unfortunately that logic doesn't apply to user-defined classes by default. This means any two separate user objects are never equal. In other words, a user object is only equal to itself and nothing else. Equality problem is very important during testing, as it is at the heart of all assertions. So it comes without a surprise that any issue in this area will cause Pythoscope to generate faulty test cases. One example of such a test case is presented on figure 7.3.

All of the problems described above make Pythoscope a semi-automatic tool: rather a help in the hands of a developer working on a legacy system than a complete solution. A study of programmers' productivity with Pythoscope is a possible follow-up to this thesis.

The results may not seem very optimistic, so it is important to note that the goal of this thesis was to improve Pythoscope's capabilities by adding support for side effects. While that goal has been reached there is still a lot of work required to make Pythoscope a stable and production-ready tool.

Bibliography

[1] Aho, Sethi, Ullman, “Compilers: Principles, Techniques, and Tools”, Addison-Wesley, 1986

[2] James Bach, “Test Automation Snake Oil”, version 2.1, 1999, `http://www.satisfice.com/articles/test_automation_snake_oil.pdf`

[3] Ned Batchelder, “Wicked hack: Python bytecode tracing”, document available at `http://nedbatchelder.com/blog/200804/wicked_hack_python_bytecode_tracing.html`, retrieved on September 30th 2010

[4] Kent Beck, “Test-Driven Development: by Example”, Addison Wesley, 2003

[5] Edmund M. Clarke, Jr., Orna Grumberg, Doron A. Peled, “Model Checking”, MIT Press, 1999

[6] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, B. M. Horowitz, “Model-based testing in practice” icse, pp.285, 21st International Conference on Software Engineering (ICSE’99), 1999

[7] Michael C. Feathers, “Working Effectively with Legacy Code”, Prentice Hall, 2005

[8] Martin Fowler, “Mocks Aren’t Stubs”, revision from January 2nd 2007, `http://martinfowler.com/articles/mocksArentStubs.html`

[9] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley Professional, 1st edition, 1999

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns”, Addison-Wesley, 1995

[11] Graham, Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. “gprof: a call graph execution profiler” in Proc. SIGPLAN ’82 Symp. on Compiler Construction, SIGPLAN Notices 17(4), pp. 120-126, 1982.

[12] Kade Hansson, “Java: Trees Versus Bytes”, thesis available at `http://central.kaserver5.org/Kasoft/Typeset/JavaTree/`, retrieved on September 30th 2010

[13] Raymond Hettinger, “PEP-289: Generator Expressions”, document available at `http://www.python.org/dev/peps/pep-0289/`, retrieved on September 30th 2010

[14] Andrew Kuchling, “Python’s Dictionary Implementation: Being All Things to All People”, chapter 18 in “Beautiful Code”, O’Reilly Media, 1st edition, 2007

[15] A.M. Kuchling, “What’s New in Python 2.4”, document available at `http://docs.python.org/whatsnew/2.4.html`, retrieved on September 30th 2010

[16] Alex Martelli, “Python in a Nutshell”, Second Edition, O’Reilly Media, 2006

[17] Gerard Meszaros, “XUnit Test Patterns, Refactoring Test Code”, Addison-Wesley, 2007

[18] Bertrand Meyer, “Object-Oriented Software Construction” 2nd edition, Prentice Hall, 1997

[19] Glenford J. Myers, “The Art of Software Testing”, 2nd edition, Wiley, 2004

[20] Bryan O’Sullivan, Don Stewart, John Goerzen, “Real World Haskell”, O’Reilly Media, 2008

[21] Guido van Rossum, Raymond Hettinger, “PEP-308: Conditional Expressions”, document available at `http://www.python.org/dev/peps/pep-0308/`, retrieved on September 30th 2010

[22] Guido van Rossum, Barry Warsaw, “PEP-8: Style Guide for Python Code”, document available at `http://www.python.org/dev/peps/pep-0008/`, retrieved on September 30th 2010

[23] B.G. Ryder, "Constructing the Call Graph of a Program", IEEE Transactions on Software Engineering, vol. SE-5, no. 3, pp. 216-226, May 1979

[24] Kevin D. Smith, Jim J. Jewett, Skip Montanaro, Anthony Baxter, "PEP-318: Decorators for Functions and Methods", document available at `http://www.python.org/dev/peps/pep-0318/`, retrieved on September 30th 2010

[25] L. Williams, G. Kudrjavets, N. Nagappan, "On the Effectiveness of Unit Test Automation at Microsoft", International Symposium on Software Reliability Engineering (ISSRE) 2009, Mysuru, India, pp. 81-89

[26] "2to3 - Automated Python 2 to 3 code translation", documentation page available at `http://docs.python.org/library/2to3.html`, retrieved on September 30th 2010

[27] "Abstract Syntax Trees", documentation page available at `http://docs.python.org/library/ast.html`, retrieved on September 30th 2010

[28] "byteplay" library, homepage available at `http://code.google.com/p/byteplay/`, retrieved on September 30th 2010

[29] "Python compiler package", documentation page available at `http://docs.python.org/library/compiler.html`, retrieved on September 30th 2010

[30] "coverage" nose plugin, documentation page available at `http://somethingaboutorange.com/mrl/projects/nose/doc/plugin_cover.html`, retrieved on September 30th 2010

[31] "ctypes — A foreign function library for Python", documentation page available at `http://docs.python.org/library/ctypes.html`, retrieved on September 30th 2010

[32] "docutils" library, homepage available at `http://docutils.sourceforge.net/`, retrieved on September 30th 2010

[33] "Import utilities", documentation page available at `http://docs.python.org/library/imputil.html`, retrieved on September 30th 2010

[34] "mock" test double library, homepage available at `http://www.voidspace.org.uk/python/mock/`, retrieved on September 30th 2010

[35] “Monkey patch” glossary entry available at http://plone.org/documentation/glossary/monkeypatch, retrieved on September 30th 2010

[36] “nose” unit testing framework, homepage available at http://somethingaboutorange.com/mrl/projects/nose/, retrieved on September 30th 2010

[37] “peak.util.assembler” library, homepage available at http://peak.telecommunity.com/DevCenter/BytecodeAssembler, retrieved on September 30th 2010

[38] “pickle module”, documentation page available at http://docs.python.org/library/pickle.html, retrieved on September 30th 2010

[39] “py.test” unit testing framework, homepage available at http://pytest.org, retrieved on September 30th 2010

[40] “Python glossary”, documentation page available at http://docs.python.org/glossary.html, retrieved on September 30th 2010

[41] Python project home page available at http://www.python.org/, retrieved on September 30th 2010

[42] “The Python Standard Library”, documentation page available at http://docs.python.org/library/, retrieved on September 30th 2010

[43] “Python testing tools taxonomy”, wiki page available at http://pycheesecake.org/wiki/PythonTestingToolsTaxonomy, retrieved on September 30th 2010

[44] “The Python Tutorial, chapter 6: Modules”, document available at http://docs.python.org/tutorial/modules.html, retrieved on September 30th 2010

[45] “Running Pythoscope against the stdlib” post by Ryan Freckleton on pythoscope google group available at http://groups.google.com/group/pythoscope/msg/12aa72251b8030a2, retrieved on September 30th 2010

[46] “setuptools” library, homepage available at http://pypi.python.org/pypi/setuptools, retrieved on September 30th 2010

[47] "StringIO — Read and write strings as files", documentation page available at `http://docs.python.org/library/stringio.html`, retrieved on September 30th 2010

[48] "The Python Language Reference", documentation page available at `http://docs.python.org/reference/`, retrieved on September 30th 2010

[49] "unittest module", documentation page available at `http://docs.python.org/library/unittest.html`, retrieved on September 30th 2010

[50] "Pure function" wikipedia article available at `http://en.wikipedia.org/wiki/Pure_function`, retrieved on September 30th 2010

CD contents

`/cdcontents.txt` List of CD contents with descriptions

`/thesis` This thesis contents in pdf format

`/pythoscope` Pythoscope's source code

`/aap` Code of A-A-P used as an example in chapter 3

`/testapps` Code of applications used for testing in chapter 7

`/generated` Code of generated tests from chapter 7