equip Documentation

Release 0.1

Romain Gaucher

August 04, 2015

Contents

equip is a small library that helps with Python bytecode instrumentation. Its API is designed to be small and flexible to enable a wide range of possible instrumentations.

The instrumentation is designed around the injection of bytecode inside the bytecode of the program to be instrumented. However, the developer does not need to know anything about the Python bytecode.

The following example shows how to write a simple instrumentation tool that will print all method called in the program, along with its arguments:

```
import sys
import equip
from equip import Instrumentation, MethodVisitor, SimpleRewriter

BEFORE_CODE = """
print ">> START"
print "[CALL] {file_name}::{method_name}:{lineno}", {arguments}
print "<< END"
"""

class MethodInstr(MethodVisitor):
  def __init__(self):
    MethodVisitor.__init__(self)

  def visit(self, meth_decl):
    rewriter = SimpleRewriter(meth_decl)
    rewriter.insert_before(BEFORE_CODE)

instr_visitor = MethodInstr()
instr = Instrumentation(sys.argv[1])
if not instr.prepare_program():
  return
instr.apply(instr_visitor, rewrite=True)
```

This program requires the path to the program to instrument, and will compile the source to generate the bytecode to instrument. All bytecode will be loaded into its representation, and the `MethodInstr` visitor will be called on all method declarations.

When a change is required (i.e., the code actually needs to be instrumented), the `Instrumentation` will overwrite the `pyc` file.

Running the instrumented program afterwards does not require anything but executing it as you would usually do. If the injected code has external dependencies, you can simply modify the `PYTHONPATH` to point to the required modules.

Contents:

CHAPTER 1

Installation

equip does not have any dependencies and is available on PyPi:

```
$ pip install equip
```

You can also install equip using the `setup.py`:

```
$ git clone https://github.com/neuroo/equip.git
$ cd equip
$ python setup.py develop
```

1.1 Current Limitations

The current version of equip only supports Python 2.7. It has not been tested on any other versions. Actually, if you try to run it on a different version, you'll get an exception complaining about the mismatching version.

The more practical way to use equip is however to leverage *virtualenv*.

1.2 virtualenv

During testing and to instrument different part of the program, it is useful to deploy the program under a virtual env. Here are the few steps to create a virtualenv:

```
$ sudo pip install virtualenv
$ mkdir project
$ cd project
$ virtualenv test-env
$ . test-env/bin/activate
```

Under this virtual environment, you can install equip the same way:

```
$ pip install equip
```

CHAPTER 2

Getting Started

equip has a simple interface that contains a handful of important classes to work with:

- Instrument
- SimpleRewriter
- MethodVisitor

2.1 Instrument

Main interface for the instrumentation. It triggers the conversion from the bytecode to the internal representation, as well as executing the visitors and writing back the resulting bytecode.

The workflow of `Instrument` requires the following steps:

1. Pass the location (or locations) to the Instrument:

```
instr.location = ['path/to/module', 'path/to/other/module']
```

2. Ask `Instrument` to prepare the program by compiling the sources (if necessary or requested) and creating a list of bytecode files that can be instrumented:

```
if not instr.prepare_program():
  raise Exception('Error while compiling the code...')
```

3. Apply the visitor on all bytecode files and persist the new bytecode:

```
instr.apply(my_visitor, rewrite=True)
```

The compilation of the program is not performed by default as the program might already be compiled, and the bytecode ready to consume. If however, we want to force rebuilding the bytecode for the entire application, we can set the force-rebuild option between step 1 and 2:

```
instr.set_option('force-rebuild')
```

2.2 Visitors

The `Instrument` creates a representation for each pyc file that contains different `Declaration` objects. A visitor can be created to iterate over these `Declaration`.

The most commonly used visitor is the `MethodVisitor` that is triggered over all method declarations found in the bytecode.

Here's an example of a visitor that prints the start and end line for each method:

```
class MethodLinesVisitor(MethodVisitor):
  def __init__(self):
    MethodVisitor.__init__(self)

  def visit(self, meth_decl):
    print "Method %s: start=%d, end=%d" \
          %  (meth_decl.method_name, meth_decl.start_lineno, meth_decl.end_lineno)
```

2.3 SimpleRewriter

Handles the insertion of bytecode, and generation of proper bytecode. The rewriter allows for multiple operations such as:

- Insert generic bytecode
- Insert import statements
- Insert on_enter/on_exit callbacks

The rewriter is called from within a visitor or any other way to get a particular `Declaration`. It consumes the `Declaration` and allows for inserting bytecode at any desired point in the original bytecode.

For example, we can add create an instrumentation to insert for all returns in a method:

```
ON_AFTER = """
print "Exit {method_name}, return value := %s" % repr({return_value})
"""

class ReturnValuesVisitor(MethodVisitor):
  def __init__(self):
    MethodVisitor.__init__(self)

  def visit(self, meth_decl):
    rewriter = SimpleRewriter(meth_decl)
    rewriter.insert_after(ON_AFTER)
```

Note that the `Instrument` is currently responsible for applying the changes, which means serializing the declarations of the current bytecode.

CHAPTER 3

Examples

Several examples are available in the git repository under `examples/`.

CHAPTER 4

API

4.1 equip package

4.1.1 Subpackages

equip.analysis package

Subpackages

equip.analysis.graph package

Submodules

equip.analysis.graph.dominators Dominator tree

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class `equip.analysis.graph.dominators.`**`DominatorTree`**(*cfg*)
Bases: `object`

Handles the dominator trees (dominator/post-dominator), and the computation of the dominance frontier.

`build`()

`cfg`
Returns the CFG used for computing the dominator trees.

`dom`
Returns the dict containing the mapping of each node to its immediate dominator.

`frontier`
Returns the dict containing the mapping of each node to its dominance frontier (a set).

`post_dom`
Returns the dict containing the mapping of each node to its immediate post-dominator.

`print_tree`(*post_dom=False*)

equip.analysis.graph.graphs Graph data structures

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.analysis.graph.graphs.**DiGraph**(*multiple_edges=True*)
Bases: object

A simple directed-graph structure.

add_edge(*edge*)

add_node(*node*)

edges

has_node(*node*)

in_degree(*node*)

in_edges(*node*)

inverse()
Returns a copy of this graph where all edges have been reversed

make_add_edge(*source=None*, *dest=None*, *kind=None*, *data=None*)

make_add_node(*kind=None*, *data=None*)

static make_edge(*source=None*, *dest=None*, *kind=None*, *data=None*)

static make_node(*kind=None*, *data=None*)

multiple_edges

nodes

out_degree(*node*)

out_edges(*node*)

remove_edge(*edge*)

remove_node(*node*)

to_dot()

class equip.analysis.graph.graphs.**Edge**(*source=None*, *dest=None*, *kind=None*, *data=None*)
Bases: object

GLOBAL_COUNTER = 0

data

dest

gid

inverse()

inversed

kind

source

class equip.analysis.graph.graphs.**Node**(*kind=None*, *data=None*)
Bases: object

GLOBAL_COUNTER = 0

data

gid

kind

equip.analysis.graph.io Outputs the graph structures

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.analysis.graph.io.**DotConverter**(*graph*)
Bases: object

add_edge(*edge*)

add_node(*node*)

get_node_id(*node*)

static process(*graph*)

run()

equip.analysis.graph.traversals DFS/BFS and some other utils

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.analysis.graph.traversals.**EdgeVisitor**
Bases: object

visit(*edge*)

class equip.analysis.graph.traversals.**Walker**(*graph*, *visitor*, *backwards=False*)
Bases: object

Traverses edges in the graph in DFS.

graph

traverse(*root*)

visitor

equip.analysis.graph.traversals.**dfs_postorder_nodes**(*graph*, *root*)

Module contents

equip.analysis.graph Graph based operators.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

Submodules

equip.analysis.block Basic block for the bytecode.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class `equip.analysis.block.`**BasicBlock**(*kind*, *decl*, *index*)
Bases: `object`

Represents a basic block from the bytecode.

ENTRY = 1

EXCEPT = 6

IF = 5

IMPLICIT_RETURN = 2

LOOP = 4

UNKNOWN = 3

add_jump(*jump_index*, *branch_kind*)

clear_jumps()

decl

end_target

fallthrough

has_return_path

index

jumps

kind

length

equip.analysis.flow Extract the control flow graphs from the bytecode.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.analysis.flow.**ControlFlow**(*decl*)

Bases: object

Performs the control-flow analysis on a Declaration object. It iterates over its bytecode and builds the basic block. The final representation leverages the DiGraph structure, and contains an instance of the DominatorTree.

BLOCK_NODE_KIND = {1: 'ENTRY', 2: 'IMPLICIT_RETURN', 3: 'UNKNOWN', 4: 'LOOP', 5: 'IF', 6: 'EXCEPT'}

CFG_TMP_BREAK = -2

CFG_TMP_RAISE = -3

CFG_TMP_RETURN = -1

E_COND = 'COND'

E_END_LOOP = 'END_LOOP'

E_EXCEPT = 'EXCEPT'

E_FALSE = 'FALSE'

E_FINALLY = 'FINALLY'

E_RAISE = 'RAISE'

E_RETURN = 'RETURN'

E_TRUE = 'TRUE'

E_UNCOND = 'UNCOND'

N_CONDITION = 'CONDITION'

N_ENTRY = 'ENTRY'

N_EXCEPT = 'EXCEPT'

N_IF = 'IF'

N_IMPLICIT_RETURN = 'IMPLICIT_RETURN'

N_LOOP = 'LOOP'

N_UNKNOWN = 'UNKNOWN'

analyze()

Performs the CFA and stores the resulting CFG.

block_indices_dict

Returns the mapping of a bytecode indices and a basic blocks.

static block_kind_from_op(*op*)

block_nodes_dict

Returns the mapping of a basic bocks and CFG nodes.

blocks

Returns the basic blocks created during the control flow analysis.

decl

dominators

Returns the DominatorTree that contains:

- Dominator tree (dict of IDom)

- Post dominator tree (doc of PIDom)
- Dominance frontier (dict of CFG node -> set CFG nodes)

entry

entry_node

exit

exit_node

static find_targets(*bytecode*)

frames

static get_kind_from_block(*block*)

static get_pairs(*iterable*)

graph
Returns the underlying graph that holds the CFG.

static make_blocks(*decl*, *bytecode*)
Returns the set of `BasicBlock` that are encountered in the current bytecode. Each block is annotated with its qualified jump targets (if any).

Parameters

- **decl** – The current declaration object.
- **bytecode** – The bytecode associated with the declaration object.

Module contents

equip.analysis Operators and simple algorithms to perform analysis on the bytecode.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

equip.bytecode package

Submodules

equip.bytecode.code Parsing and representation of the supplied bytecode.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class `equip.bytecode.code.`**BytecodeObject**(*pyc_file*, *lazy_load=True*)
Bases: `object`

This class parses the bytecode from a file and constructs the representation from it. The result is:

•One module (type: `ModuleDeclaration`)

•The bytecode expanded into intelligible structure.

•Construction of nested declarations, and hierarchy of declaration types.

accept (*visitor*)
Runs the visitor over the nested declarations found in the this module, or the entire bytecode if it's a *BytecodeVisitor*.

add_enter_code (*python_code*, *import_code=None*)
Adds enter callback in the module. The callback code (both `import_code` and `python_code`) is wrapped in a main test if statement:

```
if __name__ == '__main__':
  import_code
  python_code
```

Parameters

- **python_code** – Python code to inject before the module gets executed (if it's executed under main). The code is not executed if it's not under main.
- **import_code** – Python code that contains the import statements that might be required by the injected `python_code`. Defaults to None.

add_exit_code (*python_code*, *import_code=None*)
Adds exit callback in the module. The callback code (both `import_code` and `python_code`) is wrapped in a main test if statement:

```
if __name__ == '__main__':
  import_code
  python_code
```

Parameters

- **python_code** – Python code to inject after the module gets executed (if it's executed under main). The code is not executed if it's not under main.
- **import_code** – Python code that contains the import statements that might be required by the injected `python_code`. Defaults to None.

build_representation ()
Builds the internal representation of declarations and how they relate to each other. It works by creating a map of type/method declaration indices, and then associate the bytecode for each of them.

When all declarations are created, the parenting process runs and creates the tree structure of the decalrations, such as:

```
ModuleDeclaration()
  - TypeDeclaration(name='SomeClass')
     - MethodDeclaration#lineno(name='methodOfSomeClass')
     - MethodDeclaration#lineno(name='otherMethodOfSomeClass')
```

This representation is required to run the visitors.

static build_tree (*root*, *indent=''*)
Returns a string that represents the tree of `Declaration` types.

declarations
Returns a set of all the declarations found in the current bytecode.

static finalize_decl_object (*kind*, *acc_data*)

static find_classes_methods(*bytecode*)
Finds the indices of the classes and methods declared in the bytecode. This is done by matching code_object of the declaration and the `MAKE_FUNCTION` or `BUILD_CLASS` opcode.

get_bytecode()
Returns the current translated bytecode.

get_decl(*code_object=None*, *method_name=None*, *type_name=None*)
Returns the declaration associated to the code_object `co`, or supplied name.

Warning: This is only valid until the rewriter is called on the declarations.

Parameters

- **code_object** – Python code object type
- **method_name** – Name of the method.
- **type_name** – Name of the type.

static get_formal_params(*code_object*)
Returns the ordered list of formal parameters (arguments) of a method.

Parameters **code_object** – The code object of the method.

static get_imports_from_bytecode(*code_object*, *bytecode*)
Parses the import statements from the bytecode and constructs a list of `ImportDeclaration`.

static get_last_import_ref(*bytecode*, *code_object*)
Find the last reference of an import statement in the bytecode.

get_module()
Returns the ModuleDeclaration associated with the current bytecode.

static get_parsed_code(*code_object*)

has_changes
Returns *True* if any change was performed on the module. This is used to know if we need to rewrite or not a pyc file.

load_bytecode(*code_object*)

static next_code_object(*bytecode*, *index*)

parse()
Parses the binary file (pyc) and extract the bytecode out of it. Keeps the magic number as well as the timestamp for serialization.

parse_code(*co*)
Parses a Python code object. Mostly useful for testing.

static parse_code_object(*code_object*, *bytecode*)
Parses the bytecode (`co_code` field of the code object) and dereferences the `oparg` for later analysis.

Parameters

- **code_object** – The code object containing the bytecode to analyze
- **bytecode** – The list that will be used to append the expanded bytecode sequences.

static parse_imports(*main_module*, *bytecode*)
Extracts and adds import statements to the `ModuleDeclaration`.

static prev_code_object(*bytecode*, *index*)

write()
Persists the changes in the bytecode. This overwrites the current file that contains the bytecode with the new bytecode while preserving the timestamp.

Note that the magic number if changed to be the one from the current Python version that runs the instrumentation process.

equip.bytecode.decl Structured representation of Module, Types, Method, Imports.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.bytecode.decl.**Declaration**(*kind*, *_code_object*)
Bases: object

Base class for the declaration types of object.

FIELD = 4

IMPORT = 5

METHOD = 3

MODULE = 1

TYPE = 2

accept(*visitor*)

add_child(*child*)
Adds a child to this declaration.

Parameters **child** – A Declaration that is a child of the current declaration.

bytecode
Returns the bytecode associated with this declaration.

bytecode_object

children
Returns the children of this declaration.

code_object

end_lineno
Returns the end line number of the declaration.

get_start_lineno()

has_changes

is_field()

is_import()

is_method()

is_module()

is_type()

kind

lines
A tuple of start/end line numbers that encapsulates this declaration.

parent
Returns the parent of this declaration or `None` if there is no parent (e.g., for a `ModuleDeclaration`).

parent_class
Returns the parent class (a `TypeDeclaration`) for this declaration.

parent_method
Returns the parent method (a `MethodDeclaration`) for this declaration.

parent_module
Returns the parent module (a `ModuleDeclaration`) for this declaration.

start_lineno
Returns the start line number of the declaration.

update_nested_code_object(*original_co*, *new_co*)

class equip.bytecode.decl.**FieldDeclaration**(*field_name*, *code_object*)
Bases: *equip.bytecode.decl.Declaration*

field_name

class equip.bytecode.decl.**ImportDeclaration**(*code_object*)
Bases: *equip.bytecode.decl.Declaration*

Models an import statement. It handles relatives/absolute imports, as well as aliases.

aliases

dots

live_names

root

star

class equip.bytecode.decl.**MethodDeclaration**(*method_name*, *code_object*)
Bases: *equip.bytecode.decl.Declaration*

The declaration of a method or a function.

body

formal_parameters

is_lambda

labels

method_name

nested_types

class equip.bytecode.decl.**ModuleDeclaration**(*module_path*, *code_object*)
Bases: *equip.bytecode.decl.Declaration*

The module is the object that captures everything under one pyc file. It contains nested classes and functions, as well as import statements.

add_import(*importDecl*)

classes

functions

imports

module_path

class equip.bytecode.decl.**TypeDeclaration**(*type_name*, *code_object*)
Bases: *equip.bytecode.decl.Declaration*

Represent a class declaration. It has a name, as well as a hierarchy (superclass). The type contains several methods and fields, and can have nested types.

fields

methods
Returns a list of MethodDeclaration that belong to this type.

nested_types
Returns a list of TypeDeclaration that belong to this type.

superclasses

type_name
Returns the name of the type.

equip.bytecode.utils Utilities for bytecode interaction.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

equip.bytecode.utils.**get_debug_code_object_dict**(*code_object*)

equip.bytecode.utils.**get_debug_code_object_info**(*code_object*)

equip.bytecode.utils.**show_bytecode**(*bytecode*, *start=0*, *end=4294967296*)

equip.bytecode.utils.**update_nested_code_object**(*main_co*, *original_co*, *new_co*)

Module contents

equip.bytecode Operations and representations related to parsing the bytecode and extracting its structure.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

equip.rewriter package

Submodules

equip.rewriter.merger Responsible for merging two bytecodes at the specified places, as well as making sure the resulting bytecode (and code_object) is properly created.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.rewriter.merger.**CodeObject**(*co_origin*)
Bases: object

Class responsible for merging two code objects, and generating a new one. This effectively creates the new bytecode that will be executed.

JUMP_OP = [93, 110, 120, 121, 122, 143, 111, 112, 113, 114, 115, 119]

MERGE_BACKLIST = ('co_code', 'co_firstlineno', 'co_name', 'co_filename', 'co_lnotab', 'co_flags', 'co_argcount')
List of fields in the code_object not to merge. We only keep the ones from the original code_object.

add_get_cellvars_freevars(*varname*)

add_get_constant(*const*)

add_get_names(*name*)

add_get_tuple(*value*, *field_name*)

add_get_varnames(*const*)

add_global_name(*global_name*)
Adds the global_name as a known imported name. The instrument bytecode will get modified to change any LOAD_* to a LOAD_GLOBAL when finding this name.

Parameters **global_name** – The imported global name.

append(*op*, *arg*, *bc_index=-1*, *lineno=-1*)

emit(*op*, *oparg*, *arg=None*, *lineno=-1*)
Writes the bytecode and lnotab.

get_instruction_size(*op*, *arg=None*, *bc_index=0*)

get_op_oparg(*op*, *arg*, *bc_index=0*)
Retrieve the opcode (*op*) and its argument (*oparg*) from the supplied opcode and argument.

Parameters

- **op** – The current opcode.
- **arg** – The current dereferenced argument.
- **bc_index** – The current bytecode index.

insert(*index*, *op*, *arg*, *bc_index=-1*, *lineno=-1*)

static is_jump_op(*op*)

merge_fields(*co_other*)
Merges fields from the code_object. The only fields that aren't merged, are listed in *MERGE_BACKLIST*.

Parameters **co_other** – The other code_object to merge the *co_origin* with.

prepend(*op*, *arg*, *bc_index=-1*, *lineno=-1*)

reset_code()

to_code()

class equip.rewriter.merger.**Merger**
Bases: object

AFTER = 2
Only valid for MethodDeclaration. This specifies that the instrument code should be injected before each return of the method (i.e., before each encountered RETURN_VALUE in the bytecode).

AFTER_IMPORTS = 6
Valid for `ModuleDeclaration` or `MethodDeclaration`. This specifies that the instrument code should be injected after the encountered imports.

BEFORE = 1
Only valid for `MethodDeclaration`. This specifies that the instrument code should be injected before the body.

BEFORE_IMPORTS = 5
Valid for `ModuleDeclaration` or `MethodDeclaration`. This specifies that the instrument code should be injected before the encountered imports.

INSTRUCTION = 4
Valid for all `Declaration`. This specifies that the instrument code should be injected after each instrument.

LINENO = 3
Valid for all `Declaration`. This specifies that the instrument code should be injected each time the current line number changes.

MODULE_ENTER = 8
Valid for `ModuleDeclaration`. This specifies that the code should be injected at the beginning of the module.

MODULE_EXIT = 9
Valid for `ModuleDeclaration`. This specifies that the code should be injected at the end of the module.

RETURN_VALUES = 7
Unused.

UNKNOWN = 0
Error case for the kind of location for the merge.

static already_instrumented(*bc_source*, *bc_input*)
Checks if the instrumentation in bc_input is already in bc_source

static build_bytecode_offsets(*new_co*, *bytecode*)

static get_final_bytecode(*bc_source*, *bc_input*, *co_source*, *co_input*, *location*, *ins_lineno*, *ins_offset=-1*)
Computes the final sequences of opcodes and keep old values. It also tracks what sequences come from the instrument code or the original code, so we can resolve jumps.

Parameters

- **bc_source** – The bytecode of the orignal code.
- **bc_input** – The instrument bytecode to inject.
- **co_source** – The orignal code object.
- **co_input** – The instrument code object.
- **location** – The location of the instrumentation. It should be either: `BEFORE`, `AFTER`, `LINENO`, etc.
- **ins_lineno** – The line number to inject the instrument at. Only valid when the injection location is `LINENO`.
- **ins_offset** – Not used.

static inline_instrument(*dst_bytecode*, *src_bytecode*, *original_lineno*, *instr_counter=-1*, *template=None*, *location=0*)
Inline the instrument bytecode in place of the current state of `dst_bytecode`.

Parameters

- **dst_bytecode** – The list that contains the final bytecode.
- **src_bytecode** – The bytecode of the instrument.
- **original_lineno** – The line number from the original bytecode, so we always map the instrument code line numbers to the code being instrumented.
- **instr_counter** – A counter to track the frames of the different instrumentation code being inlined. This is used to resolve jump targets.
- **template** – An instrumentation can follow a template, if so, the actual template is supplied here. An example is the instrumentation `AFTER` which requires to capture the return value. Defaults to None.

static merge(*co_source*, *co_input*, *location=0*, *ins_lineno=-1*, *ins_offset=-1*, *ins_import_names=None*)

The merger makes sure that the bytecode is properly inserted where it should be, but also that the consts/names/locals/etc. are re-indexed. We will always append at the end of the current tuples.

We need to first compute the new bytecode resolve the jumps, and then dump it... if we just emit it as right now, we have an issue since we cannot know where an absolute/relative jump will land since some instr code can be inserted in between.

static merge_exit(*new_co*, *bc_source*, *bc_input*, *ins_import_names=None*)

Special handler for inserting code at the very end of a module.

static resolve_jump_targets(*bytecode*, *new_co*)

Resolves targets of jumps. Since we add new bytecode, absolute (resp. relative) jump address (resp. offset) can change and we need to track the changes to find the new targets.

The resolver works in two phases:

1. Create the list of bytecode indices based on the size of the opcode and its argument.
2. For each jump opcode, take its argument and resolve it in the same part of the bytecode (e.g., instrument bytecode or original bytecode).

Parameters

- **bytecode** – The structure computed by `get_final_bytecode` which overlays the final bytecode sequences and its origin.
- **new_co** – The currently created `CodeObject`.

`equip.rewriter.merger.`**RETURN_CANARY_NAME = '_______0x42024_retvalue'**

This global name is always injected as a new variable in `co_varnames`, and used to carry the return values. We essentially add:

```
    STORE_FAST '_______0x42024_retvalue'
    ... instrument code that can use `{return_value}`
    LOAD_FAST  '_______0x42024_retvalue'
    RETURN_VALUE
```

as specified by the `RETURN_INSTR_TEMPLATE`.

`equip.rewriter.merger.`**RETURN_INSTR_TEMPLATE = ((125, '_______0x42024_retvalue'), (-2, None), (124, '_______0x4**

The template that dictates how return values are being captured.

equip.rewriter.simple A simplified interface (yet the main one) to handle the injection of instrumentation code.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class `equip.rewriter.simple.`**`SimpleRewriter`**(*decl*)
Bases: `object`

The current main rewriter that works for one `Declaration` object. Using this rewriter will modify the given declaration object by possibly replacing all of its associated code object.

`KNOWN_FIELDS` = **('method_name', 'lineno', 'file_name', 'class_name', 'arg0', 'arg1', 'arg2', 'arg3', 'arg4', 'arg5', 'arg6**
List of the parameters that can be used for formatting the code to inject. The values are:

- `method_name`: The name of the method that is being called.
- **`lineno`: The start line number of the declaration object being** instrumented.
- `file_name`: The file name of the current module.
- `class_name`: The name of the class a method belongs to.

static `format_code`(*decl*, *python_code*, *location*)
Formats the supplied `python_code` with format string, and values listed in *KNOWN_FIELDS*.

Parameters

- **`decl`** – The declaration object (e.g., `MethodDeclaration`, `TypeDeclaration`, etc.).
- **`python_code`** – The python code to format.
- **`location`** – The kind of insertion to perform (e.g., `Merger.BEFORE`).

static `get_code_object`(*python_code*)
Actually compiles the supplied code and return the `code_object` to be merged with the source `code_object`.

Parameters **`python_code`** – The python code to compile.

static `get_formatting_values`(*decl*, *location*)
Retrieves the dynamic values to be added in the format string. All values are statically computed, but formal parameters (of methods) are passed by name so it is possible to dereference them in the inserted code (same for the return value).

Parameters

- **`decl`** – The declaration object.
- **`location`** – The kind of insertion to perform (e.g., `Merger.BEFORE`).

static `indent`(*original_code*, *indent_level=0*)
Lousy helper that indents the supplied python code, so that it will fit under an if statement.

`insert_after`(*python_code*)
Insert code at each *RETURN_VALUE* opcode. See *insert_before*.

`insert_before`(*python_code*)
Insert code at the beginning of the method's body.

The submitted code can be formatted using `fields` declared in `KNOWN_FIELDS`. Since `string.format` is used once the values are dumped, the injected code should be property structured.

Parameters **python_code** – The python code to be formatted, compiled, and inserted at the beginning of the method body.

insert_enter_code(*python_code*, *import_code=None*)
Insert generic code at the beginning of the module. The code is wrapped in a `if __name__ == '__main__'` statement.

Parameters

- **python_code** – The python code to compile and inject.
- **import_code** – The import statements, if any, to add before the insertion of *python_code*. Defaults to None.

insert_enter_exit_code(*python_code*, *import_code=None*, *location=9*)

insert_exit_code(*python_code*, *import_code=None*)
Insert generic code at the end of the module. The code is wrapped in a `if __name__ == '__main__'` statement.

Parameters

- **python_code** – The python code to compile and inject.
- **import_code** – The import statements, if any, to add before the insertion of *python_code*. Defaults to None.

insert_generic(*python_code*, *location=0*, *ins_lineno=-1*, *ins_offset=-1*, *ins_module=False*, *ins_import=False*)
Generic code injection utils. It first formats the supplied `python_code`, compiles it to get the *code_object*, and merge this new *code_object* with the one of the current declaration object (`decl`). The insertion is done by the `Merger`.

When the injection is done, this method will go and recursively update all references to the old *code_object* in the parents (when a parent changes, it is as well updated and its new `code_object` propagated upwards). This process is required as Python's code objects are nested in parent's code objects, and they are all read-only. This process breaks any references that were hold on previously used code objects (e.g., don't do that when the instrumented code is running).

Parameters

- **python_code** – The code to be formatted and inserted.
- **location** – The kind of insertion to perform.
- **ins_lineno** – When an insertion should occur at one given line of code, use this parameter. Defaults to -1.
- **ins_offset** – When an insertion should occur at one given bytecode offset, use this parameter. Defaults to -1.
- **ins_module** – Specify the code insertion should happen in the module itself and not the current declaration.
- **ins_import** – True of the method is called for inserting an import statement.

insert_import(*import_code*, *module_import=True*)
Insert an import statement in the current bytecode. The import is added in front of every other imports.

inspect_all_globals()

Module contents

equip.rewriter Utilities to merge and rewrite the bytecode.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

equip.utils package

Submodules

equip.utils.files

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

`equip.utils.files.`**`file_extension`**(*filename*)

`equip.utils.files.`**`good_ext`**(*fext*, *l=None*)

`equip.utils.files.`**`list_dir`**(*directory*)

`equip.utils.files.`**`scan_dir`**(*directory*, *files*, *l_ext=None*)

equip.utils.log

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

`equip.utils.log.`**`enableLogger`**(*to_file=None*)

`equip.utils.log.`**`removeOtherHandlers`**(*to_keep=None*)

Module contents

equip.visitors package

Submodules

equip.visitors.bytecode

Callback the visitor method for each encountered opcode.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.visitors.bytecode.**BytecodeVisitor**
Bases: object

A visitor to visit each instruction in the bytecode. For example, the following code:

```
class CallFunctionVisitor(BytecodeVisitor):
  def __init__(self):
    BytecodeVisitor.__init__(self)

  def visit_call_function(self, oparg):
    print "Function call with %d args" % oparg
```

Prints whenever a CALL_FUNCTION opcode is visited and prints out its number of arguments (the oparg for this opcode).

static toMethodName(*name*)

visit(*index*, *op*, *arg=None*, *lineno=None*, *cflow_in=False*)
Callback of the visitor. It dynamically constructs the name of the specialized visitor to call based on the name of the opcode.

Parameters

- **index** – Bytecode index.
- **op** – The opcode that is currently visited.
- **arg** – The expanded oparg (i.e., constants, names, etc. are resolved).
- **lineno** – The line number associated with the opcode.
- **cflow_in** – True if the current index is the target of a jump.

visit_binary_add()

visit_binary_and()

visit_binary_divide()

visit_binary_floor_divide()

visit_binary_lshift()

visit_binary_modulo()

visit_binary_multiply()

visit_binary_or()

visit_binary_power()

visit_binary_rshift()

visit_binary_subscr()

visit_binary_subtract()

visit_binary_true_divide()

visit_binary_xor()

visit_break_loop()

visit_build_class()

visit_build_list(*oparg*)

visit_build_map(*oparg*)

visit_build_set(*oparg*)

visit_build_slice(*oparg*)

visit_build_tuple(*oparg*)

visit_call_function(*oparg*)

visit_call_function_kw(*oparg*)

visit_call_function_var(*oparg*)

visit_call_function_var_kw(*oparg*)

visit_compare_op(*compare*)

visit_continue_loop(*jump_abs*)

visit_delete_attr(*name*)

visit_delete_fast(*local*)

visit_delete_global(*name*)

visit_delete_name(*name*)

visit_delete_slice_0()

visit_delete_slice_1()

visit_delete_slice_2()

visit_delete_slice_3()

visit_delete_subscr()

visit_dup_top()

visit_dup_topx(*oparg*)

visit_end_finally()

visit_exec_stmt()

visit_extended_arg(*oparg*)

visit_for_iter(*jump_rel*)

visit_get_iter()

visit_import_from(*name*)

visit_import_name(*name*)

visit_import_star()

visit_inplace_add()

visit_inplace_and()

visit_inplace_divide()

visit_inplace_floor_divide()

visit_inplace_lshift()

visit_inplace_modulo()

visit_inplace_multiply()

visit_inplace_or()

visit_inplace_power()

visit_inplace_rshift()

visit_inplace_subtract()

visit_inplace_true_divide()

visit_inplace_xor()

visit_jump_absolute(*jump_abs*)

visit_jump_forward(*jump_rel*)

visit_jump_if_false_or_pop(*jump_abs*)

visit_jump_if_true_or_pop(*jump_abs*)

visit_list_append(*oparg*)

visit_load_attr(*name*)

visit_load_closure(*free*)

visit_load_const(*constant*)

visit_load_deref(*free*)

visit_load_fast(*local*)

visit_load_global(*name*)

visit_load_locals()

visit_load_name(*name*)

visit_make_closure(*oparg*)

visit_make_function(*oparg*)

visit_map_add(*oparg*)

visit_nop()

visit_pop_block()

visit_pop_jump_if_false(*jump_abs*)

visit_pop_jump_if_true(*jump_abs*)

visit_pop_top()

visit_print_expr()

visit_print_item()

visit_print_item_to()

visit_print_newline()

visit_print_newline_to()

visit_raise_varargs(*oparg*)

visit_return_value()

visit_rot_four()

visit_rot_three()

visit_rot_two()

visit_set_add(*oparg*)

visit_setup_except(*jump_rel*)

visit_setup_finally(*jump_rel*)

visit_setup_loop(*jump_rel*)

visit_setup_with(*jump_rel*)

visit_slice_0()

visit_slice_1()

visit_slice_2()

visit_slice_3()

visit_stop_code()

visit_store_attr(*name*)

visit_store_deref(*free*)

visit_store_fast(*local*)

visit_store_global(*name*)

visit_store_map()

visit_store_name(*name*)

visit_store_slice_0()

visit_store_slice_1()

visit_store_slice_2()

visit_store_slice_3()

visit_store_subscr()

visit_unary_convert()

visit_unary_invert()

visit_unary_negative()

visit_unary_not()

visit_unary_positive()

visit_unpack_sequence(*oparg*)

visit_with_cleanup()

visit_yield_value()

equip.visitors.classes Callback the visit method for each encountered class in the program.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.visitors.classes.**ClassVisitor**
Bases: object

A class visitor that is triggered for all encountered TypeDeclaration.

Example, listing all types declared in the bytecode:

```
class TypeDeclVisitor(ClassVisitor):
  def __init__(self):
    ClassVisitor.__init__(self)

  def visit(self, typeDecl):
    print "New type: %s (parentDecl=%s)" \
          % (typeDecl.type_name, typeDecl.parent)
```

visit(*typeDecl*)

equip.visitors.methods Callback the visit method for each encountered method in the program.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.visitors.methods.**MethodVisitor**
Bases: object

A method visitor that is triggered for all encountered MethodDeclaration.

Example, listing all methods declared in the bytecode:

```
class MethodDeclVisitor(MethodVisitor):
  def __init__(self):
    MethodVisitor.__init__(self)

  def visit(self, methDecl):
    print "New method: %s:%d (parentDecl=%s)" \
          % (methDecl.method_name, methDecl.start_lineno, methDecl.parent)
```

visit(*methodDecl*)

equip.visitors.modules Callback the visit method for each encountered module in the program.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.visitors.modules.**ModuleVisitor**
Bases: object

visit(*moduleDecl*)

Module contents

equip.visitors Different visitor interfaces to traverse the bytecode, modules, classes, or methods.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

4.1.2 Submodules

equip.instrument

Main interface to handle the instrumentation and run the visitors.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class `equip.instrument.`**`Instrumentation`**(*location=None*)
Bases: `object`

Main class for handling the instrumentation. The typical workflow is:

1. Set the location from the ctor or using the *location* setter
2. Update options, such as *force-rebuild*
3. Call *prepare_program* to scan the file system for source/bytecode
4. Register any *on_enter*/*on_exit* instrumentation callbacks
5. *apply* the instrumentation using a customer visitor

`KNOWN_OPTIONS` **= ('force-rebuild',)**
The list of known options

`apply`(*visitor*, *rewrite=False*)
Runs the visitor over all matching types (e.g., MethodDeclaration, etc.).

Parameters

- **`visitor`** – The instance of the visitor to run over the program.
- **`rewrite`** – Whether the instrumentation should overwrite the bytecode file (pyc) at the end. Default is *False*.

`get_option`(*key*)
Gets the value of an option. Defaults to `None`.

Parameters **`key`** – The name of the option.

`instrument`(*visitor*, *bytecode_file*, *rewrite=False*)
Loads the representation of the bytecode in *bytecode_file*, and apply the visitor to the representation.

Parameters

- **`visitor`** – The instance of the visitor to run over the representation of the bytecode.
- **`bytecode_file`** – Absolute path of the file containing the bytecode (pyc).
- **`rewrite`** – Whether the instrumentation should overwrite the bytecode file (pyc) at the end. Default is *False*.

`location`
The path that contains the bytecode of the application to instrument. The path can either be a string or an iterable.

on_enter(*python_code*, *import_code=None*)
Inserts the `python_code` at the beginning of the module inside an if statement. The resulting injected code looks like this:

```
if __name__ == '__main__':
  python_code
```

Parameters

- **python_code** – Python code to inject before the module gets executed (if it's executed under main). The code is not executed if it's not under main.
- **import_code** – Python code that contains the import statements that might be required by the injected `python_code`. Defaults to None.

on_exit(*python_code*, *import_code=None*)
Inserts the `python_code` at the end of the module inside an if statement. The resulting injected code looks like this:

```
if __name__ == '__main__':
  python_code
```

Parameters

- **python_code** – Python code to inject after the module gets executed (if it's executed under main). The code is not executed if it's not under main.
- **import_code** – Python code that contains the import statements that might be required by the injected `python_code`. Defaults to None.

prepare_program()
Builds the representation of the program, and compiles all source files if it's either necessary (e.g., missing bytecode for existing source) or if the `force-rebuild` option is set.

set_option(*key*, *value=True*)
Sets one of the options used later one by the instrumentation. The available options are listed in *KNOWN_OPTIONS*.

Parameters

- **key** – The name of the option to set.
- **value** – The value of the option. Defaults to `True`.

validate()
Debugging info for the instrumented bytecode. Iterates again over all the bytecode and dumps the current (instrmented) bytecode.

equip.prog

Handles the current program for instrumentation.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

class equip.prog.**Program**(*instrumentation*)
Bases: object

Captures the sources and binaries from the current program to instrument.

bytecode_files
The list of pyc files.

compile_program()
Compiles the program.

create_program(*skip_rebuild=False*)
Creates the structure of the program with its source files and binary files. When the Instrument option force-rebuild is set, it will trigger the compilation of all python source files.

Parameters **skip_rebuild** – Force skipping the build. Mostly here due to the recursive nature of this function.

static split_program_source_bc(*lst*)

4.1.3 Module contents

equip

Bytecode instrumentation framework for Python.

copyright

3. 2014 by Romain Gaucher (@rgaucher)

license Apache 2, see LICENSE for more details.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

e

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W