Astroid Documentation

Release 0.24.4

Logilab S.A.

February 26, 2014

Contents

Contents:

CHAPTER 1

Inference on the AST in Astroid

1.1 Introduction

1.1.1 What/where is 'inference' ?

Well, not *inference* in general, but inference within *astroid* in particular... Basically this is extracting information about a node of the AST from the node's context so as to make its description richer. For example it can be most useful to know that this identifier node *toto* can have values among 1, 2.0, and "yesterday".

The inference process entry-point is the `NodeNG.infer()` method of the AST nodes which is defined in `NodeNG` the base class for AST nodes. This method return a generator which yields the successive inference for the node when going through the possible execution branches.

1.1.2 How does it work ?

Todo

double chek this `infer()` is monkey-patched point

The `NodeNG.infer()` method either delegates the actual inference to the instance specific method `NodeNG._explicit_inference()` when not *None* or to the overloaded `_infer()` method. The important point to note is that the `_infer()` is *not* defined in the nodes classes but is instead *monkey-patched* in the `inference.py` so that the inference implementation is not scattered to the multiple node classes.

Note: The inference method are to be wrapped in decorators like `path_wrapper()` which update the inference context.

In both cases the `infer()` returns a *generator* which iterates through the various *values* the node could take.

Todo

introduce the `inference.infer_end()` method and terminal nodes along with the recursive call

In some case the value yielded will not be a node found in the AST of the node but an instance of a special inference class such as `_Yes`, `Instance`,etc. Those classes are defined in `bases.py`.

Namely, the special singleton `YES()` is yielded when the inference reaches a point where t can't follow the code and is so unable to guess a value ; and instances of the `Instance` class are yielded when the current node is infered to be an instance of some known class.

1.1.3 What does it rely upon ?

In order to perform such an inference the `infer()` methods rely on several more global objects, mainly :

MANAGER is a unique global instance of the class `AstroidManager`, it helps managing and reusing inference needed / done somewhere else than the current invocation node.

InferenceContext Instances of this class can be passed to the `infer()` methods to convey aditionnal information on the context of the current node, and especially the current scope.

Todo

Write something about `Scope` objects and `NodeNG.lookup()` method.

1.2 API documentation

Here is the annotaded API documentation extracted from the source code of the `inference`.

Todo

actually annotate the doc to structure its approach

this module contains a set of functions to handle inference on astroid trees

class `inference.`**CallContext**(*args*, *starargs*, *dstarargs*)
when inferring a function call, this class is used to remember values given as argument

infer_argument(*funcnode*, *name*, *context*)
infer a function argument value according to the call context

`inference.`**infer_arguments**(*self*, *context=None*)

`inference.`**infer_ass**(*self*, *context=None*)
infer a AssName/AssAttr: need to inspect the RHS part of the assign node

`inference.`**infer_augassign**(*self*, *context=None*)

`inference.`**infer_binop**(*self*, *context=None*)

`inference.`**infer_callfunc**(*self*, *context=None*)
infer a CallFunc node by trying to guess what the function returns

`inference.`**infer_empty_node**(*self*, *context=None*)

`inference.`**infer_end**(*self*, *context=None*)
inference's end for node such as Module, Class, Function, Const...

`inference.`**infer_from**(*self*, *context=None*, *asname=True*)
infer a From nodes: return the imported module/object

`inference.`**infer_getattr**(*self*, *context=None*)
infer a Getattr node by using getattr on the associated object

`inference.`**infer_global**(*self*, *context=None*)

`inference.`**infer_import**(*self*, *context=None*, *asname=True*)
infer an Import node: return the imported module/object

`inference.`**infer_index**(*self*, *context=None*)

inference.**infer_name**(*self*, *context=None*)
infer a Name: use name lookup rules

inference.**infer_name_module**(*self*, *name*)

inference.**infer_subscript**(*self*, *context=None*)
infer simple subscription such as [1,2,3][0] or (1,2,3)[-1]

inference.**infer_unaryop**(*self*, *context=None*)

CHAPTER 2

Extending Astroid Syntax Tree

Sometimes Astroid will miss some potentially important information users may wish to add, for instance with the standard library *hashlib* module. In some other cases, users may want to customize the way inference works, for instance to explain Astroid that calls to *collections.namedtuple* are returning a class with some known attributes.

Modifications in the AST are now possible using the using the generic transformation API. You can find examples in the *brain/* subdirectory, which are taken from the the pylint-brain project.

Transformation functions are registered using the *register_transform* method of the Astroid manager:

`AstroidManager.`**`register_transform`**(*node_class*, *transform*, *predicate=None*)
Register *transform(node)* function to be applied on the given Astroid's *node_class* if *predicate* is None or return a true value when called with the node as argument.

The transform function may return a value which is then used to substitute the original node in the tree.

To add filtering based on the *as_string* representation of the node in addition to the type, the `astroid.AsStringRegexpPredicate` predicate object can be used.

class `astroid.`**`AsStringRegexpPredicate`**(*regexp*, *expression=None*)
Class to be used as predicate that may be given to *register_transform*

First argument is a regular expression that will be searched against the *as_string* representation of the node onto which it's applied.

If specified, the second argument is an *attrgetter* expression that will be applied on the node first to get the actual node on which *as_string* should be called.

Last but not least, the `inference_tip()` function is there to register a custom inference function.

`astroid.`**`inference_tip`**(*infer_function*)
Given an instance specific inference function, return a function to be given to MANAGER.register_transform to set this inference function.

Typical usage

```
MANAGER.register_transform(CallFunc, inference_tip(infer_named_tuple),
                           AsStringRegexpPredicate('namedtuple', 'func'))
```

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

i

A

C

I

R