Qualified Name for Classes and Functions

Qualifed Team Name Hunter Haskins, Jose Escobar, Michael Crozier, Steven Funk

Python's Scoping Issues

- Python's introspection facilities have long had poor support for nested classes due to only having 3 defined namespaces:
  - Local
  - Global
  - Built-in

Python's Scoping Issues-continued

- The following slides provide a brief history of how a solution is formed
  - Initial attempt at a solution: PEP 227 - Statically Nested Scopes
  - Second attempt: PEP 3104 - Access to Names in Outer Scopes
  - Latest attempt: PEP 3155 - Qualifed Name for Classes and Functions

PEP 227 - Statically Nested Scopes

- The addition of nested scopes allows resolution of unbound local names in enclosing function's namespaces
- According to Python 2.0: for a function foo defined within a function bar, the names bound in bar are not visible in foo

PEP 227 - Statically Nested Scopes - continued

- The proposal PEP 227 then changes the rules so that names bound in bar are visible in foo, unless foo conatins a name binding that hides the binding in bar

PEP 227 Specifications

- Changes made by PEP 227 address two problems:
  1. The limited utility of lambda expressions & nested expressions in general
  2. The inability to define reursive functions except at a modular level
- New Feature: Names are introduced by the name binding operations:
  - Operations include: argument declarations, assignments, class and functions definitinos, for statements, and except clauses

PEP 227 Specifications - continued

- Results:
  - If a name is bound anywhere within a code block, all uses of the name within the block are treated as a references to the current block
    - Negative side effects: This can lead to errors when a name is used within a block before it is bound
  - If the global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace

PEP 227 code example

```
def foo():
    def factorial(n):
        if n < 2:
            return 1
        return n * factorial(n-1)
    print factorial(10)
```

Nested functions example

PEP 3104 - Access to Names in Outer Scopes

- Up to this update python code can only rebind names in 2 scopes:
  - Local scope by simple assignment
  - Module-global scope by using a global declaration
- In Python, though functions are usually defined at the top level, a function definition can be executed anywhere
  - This gave Python the syntactic apperance of nested scoping without the semantics

PEP 3104 - Access to Names in Outer Scopes - continued

- Proposed Solution:
  - Add scope override declaration in the referring (inner) scope
  - Declared as: `nonlocal x`
  - This works because it prevents x from becoming a local name in the current scope
- SyntaxErrors:
  - Occurs when there is no pre-existing binding in an enclosing scope
  - If a nonlocal declaration collides with the name of a formal parameter in the local scope

PEP 3104 Pre-Solution Code Example

```
def scoreboard(score = 0):
    def increment():
        score = score + 1 """fails with UnboundLocalE
```

This simple code shows that the function increment within scoreboard can't update score because it is not a local variable in increment

PEP 3104 - Initial Solution

- Argued that such a feature isn't necessary, as a rebindable outer variable can be simulated by wrapping it in a mutable object

```
class Namespace:
    pass
def scoreboard(score = 0)
    ns = Namespace()
    ns.score = 0
    def increment():
        ns.score = ns.score + 1
```

- But having to make additional namespaces to make up for missing functionality is a pain

PEP 3104 Post-Solution Code Example

```
def scoreboard(nonlocal score = 0):
    def increment():
        score = score + 1
```

This time around score is declared as nonlocal, and can be changed within increment

PEP 3155 - Qualified Name for Classes and Functions

- Given a class it was impossible to tell if it was defined within another class or at the top-level
- In the former case, it was also impossible to tell which class it was specifically defined in

PEP 3155 - Qualifed Name for Classes and Functions - Proposal

- Add the attribute **qualname** to functions and classes
- At top-level **qualname** is the same as **name**
- For nested functins and clases **qualname** contains a dotted path leading to the object from the top-level
- A function's local namespace is represented in the dotted path by a component called

PEP 3155 Pre-Solution

In Python 2 you could use `im_class` to see where things were defined

```
class C:
    def foo():
```

If you did `C.foo.im_class` you'd get 'class '**main**.C' But in Python 3 this was taken out and with the same thing you would get AttributeError: 'function' object has no attribute 'im_class'

PEP 3155 - Post-Solution Code Example

```
>>> class foo:
    def bar(): pass
    class A:
        def b(): pass

>>> foo.__qualname__
'foo'
>>> foo.bar.__qualname__
'foo.bar'
>>> foo.A.__qualname__
'foo.A'
>>> foo.A.b.__qualname__
'foo.A.b'
```

Limitations of Qualified Name

- The output of qualname function is not programmatically walkable
- This means that the usefulness of qualname will be limited to giving the programmer more information, becuase it can not be used in any program logic

Wrap Up

- PEP 227: Added statically nested scoping, which allowed fro the variables bound in the scope that surrounds the current scope to be referenced
- Pep 3104: Added scope override declaration in the referring inner scope
- Pep 3155: Added qualname attribute to give the programmer a way to see the path to a variable by showing the list of the namesapces before where the classes or function is defined

Citations

- PEP 3104: http://www.python.org/dev/peps/pep-3104/
- PEP 227: http://www.python.org/dev/peps/pep-0227/
- PEP 3155: http://www.python.org/dev/peps/pep-3155/