

inline-test_functions.py

Link

<https://github.com/bitcoin/bitcoin/blob/e25af11225d9d94ecf7068bf7a9a359268786fbe/contrib/binaries/verify.py#L267C1-L267C1>

▼ Code

```
def test_assorted(self):
    table1 = table("mytable", column("myid", Integer))

    table2 = table("myothertable", column("otherid", Integer))

    self.assert_compile(
        func.lala(3, 4, literal("five"), table1.c.myid) * table2.c.otherid,
        "lala(:lala_1, :lala_2, :param_1, mytable.myid) * "
        "myothertable.otherid",
    )

    self.assert_compile(
        select(func.count(table1.c.myid)),
        "SELECT count(mytable.myid) AS count_1 FROM mytable",
    )

    self.assert_compile(
        select(func.foo.bar.lala(table1.c.myid)),
        "SELECT foo.bar.lala(mytable.myid) AS lala_1 FROM mytable",
    )

    self.assert_compile(
        select(func.foo.bar.lala(12)),
        "SELECT foo.bar.lala(:lala_2) AS lala_1",
    )

    self.assert_compile(func.lala.hoho(7), "lala.hoho(:hoho_1)")

    self.assert_compile(
        func.my_func(1, 2, None, 3),
        "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
    )

    f1 = func.my_func(1, 2, None, 3)
    f1._generate_cache_key()

    self.assert_compile(
        pickle.loads(pickle.dumps(f1)),
        "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
    )
```

```

try:
    func.__bases__
    assert False
except AttributeError:
    assert True

```

▼ Documentation

```

def test_assorted(self):
    table1 = table("mytable", column("myid", Integer))

    table2 = table("myothertable", column("otherid", Integer))

    # test an expression with a function
    self.assert_compile(
        func.lala(3, 4, literal("five"), table1.c.myid) * table2.c.otherid,
        "lala(:lala_1, :lala_2, :param_1, mytable.myid) * "
        "myothertable.otherid",
    )

    # test it in a SELECT
    self.assert_compile(
        select(func.count(table1.c.myid)),
        "SELECT count(mytable.myid) AS count_1 FROM mytable",
    )

    # test a "dotted" function name
    self.assert_compile(
        select(func.foo.bar.lala(table1.c.myid)),
        "SELECT foo.bar.lala(mytable.myid) AS lala_1 FROM mytable",
    )

    # test the bind parameter name with a "dotted" function name is
    # only the name (limits the length of the bind param name)
    self.assert_compile(
        select(func.foo.bar.lala(12)),
        "SELECT foo.bar.lala(:lala_2) AS lala_1",
    )

    # test a dotted func off the engine itself
    self.assert_compile(func.lala.hoho(7), "lala.hoho(:hoho_1)")

    # test None becomes NULL
    self.assert_compile(
        func.my_func(1, 2, None, 3),
        "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
    )

    f1 = func.my_func(1, 2, None, 3)
    f1._generate_cache_key()

    # test pickling
    self.assert_compile(

```

```

        pickle.loads(pickle.dumps(f1)),
        "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
    )

    # assert func raises AttributeError for __bases__ attribute, since
    # its not a class fixes pydoc
    try:
        func.__bases__
        assert False
    except AttributeError:
        assert True

```

Prompt

As a code documentation assistant, you are assigned to document at the in-line level. When in-line comments are needed, your task is to insert comments within the code itself. The output should be the entire code along with the documentation you've added.

The code is as follows:

```

"""
def test_assorted(self):
    table1 = table("mytable", column("myid", Integer))

    table2 = table("myothertable", column("otherid", Integer))

    self.assert_compile(
        func.lala(3, 4, literal("five"), table1.c.myid) * table2.c.otherid,
        "lala(:lala_1, :lala_2, :param_1, mytable.myid) * "
        "myothertable.otherid",
    )

    self.assert_compile(
        select(func.count(table1.c.myid)),
        "SELECT count(mytable.myid) AS count_1 FROM mytable",
    )

    self.assert_compile(
        select(func.foo.bar.lala(table1.c.myid)),
        "SELECT foo.bar.lala(mytable.myid) AS lala_1 FROM mytable",
    )

    self.assert_compile(
        select(func.foo.bar.lala(12)),

```

```

        "SELECT foo.bar.lala(:lala_2) AS lala_1",
    )

    self.assert_compile(func.lala.hoho(7), "lala.hoho(:hoho_1)")

    self.assert_compile(
        func.my_func(1, 2, None, 3),
        "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
    )

    f1 = func.my_func(1, 2, None, 3)
    f1._generate_cache_key()

    self.assert_compile(
        pickle.loads(pickle.dumps(f1)),
        "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
    )

    try:
        func.__bases__
        assert False
    except AttributeError:
        assert True
"""

```

Responses

ChatGPT

```

def test_assorted(self):
    # Creating two tables, 'table1' and 'table2'
    table1 = table("mytable", column("myid", Integer))
    table2 = table("myothertable", column("otherid", Integer))

    # Testing a complex expression involving 'table1' and 'table2'
    self.assert_compile(
        func.lala(3, 4, literal("five"), table1.c.myid) * table2.c.otherid,
        "lala(:lala_1, :lala_2, :param_1, mytable.myid) * "
        "myothertable.otherid",
    )

    # Testing a simple SELECT statement with COUNT function
    self.assert_compile(
        select(func.count(table1.c.myid)),
        "SELECT count(mytable.myid) AS count_1 FROM mytable",
    )

```

```

)

# Testing a SELECT statement with nested function calls
self.assert_compile(
    select(func.foo.bar.lala(table1.c.myid)),
    "SELECT foo.bar.lala(mytable.myid) AS lala_1 FROM mytable",
)

# Testing a SELECT statement with a constant value
self.assert_compile(
    select(func.foo.bar.lala(12)),
    "SELECT foo.bar.lala(:lala_2) AS lala_1",
)

# Testing a function call with a single argument
self.assert_compile(func.lala.hoho(7), "lala.hoho(:hoho_1)")

# Testing a function call with multiple arguments and NULL
self.assert_compile(
    func.my_func(1, 2, None, 3),
    "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
)

# Generating a cache key for 'f1' function
f1 = func.my_func(1, 2, None, 3)
f1._generate_cache_key()

# Testing pickling and unpickling of 'f1' function
self.assert_compile(
    pickle.loads(pickle.dumps(f1)),
    "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
)

# Testing the handling of AttributeError
try:
    func.__bases__
    assert False
except AttributeError:
    assert True

```

GPT4

```

def test_assorted(self):
    # Create a table object 'table1' with a single column 'myid' of type Integer.
    table1 = table("mytable", column("myid", Integer))

    # Create another table object 'table2' with a single column 'otherid' of type Integer.
    table2 = table("myothertable", column("otherid", Integer))

    # Test the SQL compilation of a custom function 'lala' with parameters and a multiplication
    self.assert_compile(
        func.lala(3, 4, literal("five"), table1.c.myid) * table2.c.otherid,
        "lala(:lala_1, :lala_2, :param_1, mytable.myid) * myothertable.otherid",
    )

```

```

)

# Test the SQL compilation of a SELECT statement using 'func.count' on 'table1.c.myid'.
self.assert_compile(
    select(func.count(table1.c.myid)),
    "SELECT count(mytable.myid) AS count_1 FROM mytable",
)

# Test the SQL compilation of a nested function call 'foo.bar.lala' on 'table1.c.myid'.
self.assert_compile(
    select(func.foo.bar.lala(table1.c.myid)),
    "SELECT foo.bar.lala(mytable.myid) AS lala_1 FROM mytable",
)

# Test the SQL compilation of the same nested function call with a constant value.
self.assert_compile(
    select(func.foo.bar.lala(12)),
    "SELECT foo.bar.lala(:lala_2) AS lala_1",
)

# Test the SQL compilation of a custom function 'lala.hoho' with a constant parameter.
self.assert_compile(func.lala.hoho(7), "lala.hoho(:hoho_1)")

# Test the SQL compilation of another custom function 'my_func' with various parameters.
self.assert_compile(
    func.my_func(1, 2, None, 3),
    "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
)

# Create an instance of 'my_func' and generate its cache key.
f1 = func.my_func(1, 2, None, 3)
f1._generate_cache_key()

# Test the serialization and deserialization (using pickle) of the 'my_func' instance.
self.assert_compile(
    pickle.loads(pickle.dumps(f1)),
    "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
)

# Try accessing a non-existent attribute '__bases__' of 'func', expecting an AttributeError.
try:
    func.__bases__
    assert False # This line should not be reached; if it is, the test should fail.
except AttributeError:

```

Bard

```

"""

```

This defines a test class for assorted functions.

Args:

self: The current test class instance.

Returns:

None

"""

```
def test_assorted(self):
```

```
    """
```

```
    Tests the `lala()` function with various arguments.
```

```
    This test ensures that the `lala()` function can handle different types of arguments,
    including integers, strings, and table columns. It also checks that the function
    generates the expected SQL statements.
```

```
    """
```

```
    table1 = table("mytable", column("myid", Integer)) # Define a table object for 'mytable' wi
```

```
    table2 = table(
```

```
        "myothertable", column("otherid", Integer)
```

```
    ) # Define a table object for 'myothertable' with an 'otherid' column
```

```
    self.assert_compile(
```

```
        func.lala(3, 4, literal("five"), table1.c.myid) * table2.c.otherid,
```

```
        "lala(:lala_1, :lala_2, :param_1, mytable.myid) * myothertable.otherid",
```

```
    ) # Test 'lala()' with integers, a string, and a table column
```

```
    self.assert_compile(
```

```
        select(func.count(table1.c.myid)),
```

```
        "SELECT count(mytable.myid) AS count_1 FROM mytable",
```

```
    ) # Test 'count()' with a table column
```

```
    self.assert_compile(
```

```
        select(func.foo.bar.lala(table1.c.myid)),
```

```
        "SELECT foo.bar.lala(mytable.myid) AS lala_1 FROM mytable",
```

```
    ) # Test nested function call with a table column
```

```
    self.assert_compile(
```

```
        select(func.foo.bar.lala(12)),
```

```
        "SELECT foo.bar.lala(:lala_2) AS lala_1",
```

```
    ) # Test nested function call with an integer
```

```
    self.assert_compile(
```

```
        func.lala.hoho(7),
```

```
        "lala.hoho(:hoho_1)",
```

```
    ) # Test 'hoho()' method with an integer
```

```
    self.assert_compile(
```

```
        func.my_func(1, 2, None, 3),
```

```
        "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
```

```
    ) # Test 'my_func()' with integers and None
```

```
    f1 = func.my_func(1, 2, None, 3) # Create an instance of 'my_func'
```

```
    f1._generate_cache_key() # Call the '_generate_cache_key()' method on the instance
```

```

self.assert_compile(
    pickle.loads(pickle.dumps(f1)),
    "my_func(:my_func_1, :my_func_2, NULL, :my_func_3)",
) # Test pickling and unpickling of the instance

try:
    func.__bases__
    assert False
except AttributeError:
    assert True # Check if the 'func' class has no base classes

```

LLama2

```

def test_assorted(self):
    # Create tables
    table1 = table("mytable", column("myid", Integer))
    table2 = table("myothertable", column("otherid", Integer))

    # Test assertion with function call
    self.assert_compile(
        func.lala(3, 4, literal("five"), table1.c.myid) * table2.c.otherid,
        "lala(:lala_1, :lala_2, :param_1, mytable.myid) * "
        "myothertable.otherid"
    )

    # Test assertion with SELECT statement
    self.assert_compile(
        select(func.count(table1.c.myid)),
        "SELECT count(mytable.myid) AS count_1 FROM mytable"
    )

    # Test assertion with nested functions
    self.assert_compile(
        select(func.foo.bar.lala(table1.c.myid)),
        "SELECT foo.bar.lala(mytable.myid) AS lala_1 FROM mytable"
    )

    # Test assertion with constant argument
    self.assert_compile(
        select(func.foo.bar.lala(12)),
        "SELECT foo.bar.lala(:lala_2) AS lala_1"
    )

    # Test assertion with function call without arguments
    self.assert_compile(func.lala.hoh

```

StarChat

Here's what I've done so far:

```

def test_assorted(self):
    # This function tests various aspects of SQLAlchemy's expression language, including
    # inline commenting and caching

```



```

# Create two tables for use in this function
table1 = table("mytable", column("myid", Integer))
table2 = table("myothertable", column("otherid", Integer))

# Assert that an expected SQL statement can be generated from a given set of expressions
self.assert_compile(
    func.lala(3, 4, literal("five"), table1.c.myid) * table2.c.otherid,
    """# Comment: This line multiplies together four values using the LALA
    # function, which takes three arguments (the first two being integers, while
    # the third argument is a string). The fourth parameter is a ColumnElement object,
    # indicating the value of 'yid' in the 'ytable' table."""
    "lala(:lala_1, :lala_2, :param_1, mytable.myid) * "
    "# Comment: This line accesses the 'otherid' column in the 'yothertable' table."
    "myothertable.otherid",
)

# Assert that an expected SQL SELECT statement can be generated
self.assert_compile(
    select(func.count(table1.c.myid)),
    """# Comment: This line generates a COUNT aggregation over the 'yid' column
    # in the 'ytable' table."""
    "SELECT count(mytable.myid) AS count_1 FROM mytable",
)

# Assert that an expected SQL SELECT statement can be generated, where the function name include
self.assert_compile(
    select(func.foo.bar.lala(table1.c.myid)),
    """# Comment: This line invokes the 'foo.bar.lala()' method on the 'yid' column
    # in the 'ytable' table."""
    "SELECT foo.bar.lala(mytable.myid) AS lala_1 FROM mytable",
)

# Assert that another expected SQL SELECT statement can be generated, where the function only ac
self.assert_compile(
    select(func.foo.bar.lala(12)),
    """# Comment:

```