2026 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SANTA CLARA, CA, USA
MARCH 2 – 5, 2026

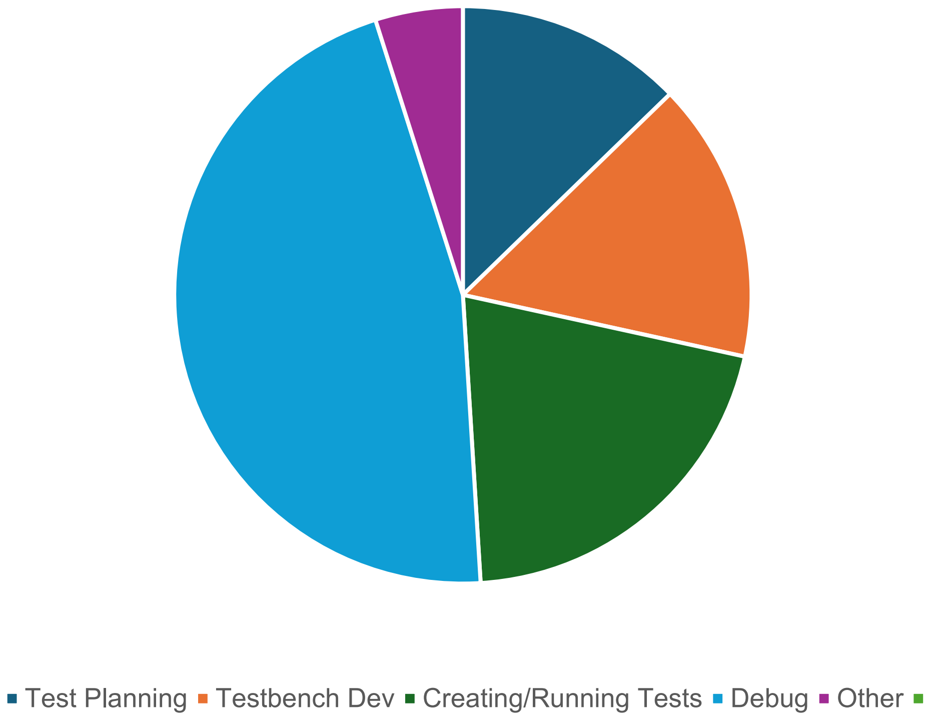
Properly Introducing Python to your UVM Testbench

Matthew Ballance

AMD

Python in Hardware Verification

- 36% of verification activities are software-like
  - Testbench development
  - Creating Tests
  - Scoreboards and data analysis
- Using a software language, like Python, is attractive
  - Library ecosystem
  - Development tools
  - Developer population
- UVM testbench integration can be a challenge
- PyHDL-IF integrates easily and enables UVM reuse



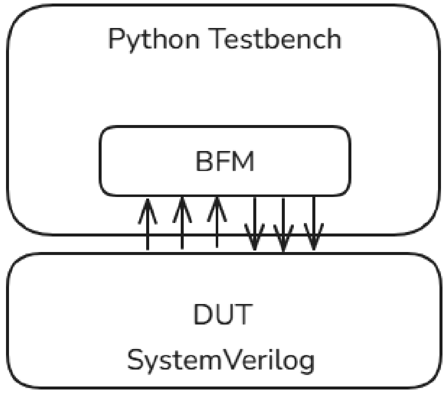
Python Integration Challenge

- Two existing approaches – each with trade-offs
  - **Signal Level:** cocotb is a well-known example
  - **Function-Wrapper:** generated Python/SV interface API

Approach	Ease of Integration	UVM Reuse
Signal Level	High	Low
Function-Wrapper	Low	High

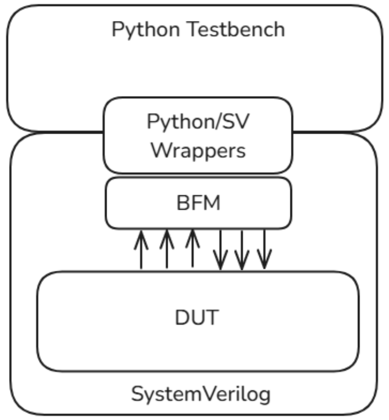
Signal-Level Integration

- Python access to signals via VPI/VHPI
  - **cocotb** is a well-known example
- Bulk of testbench is implemented in Python
  - Tests
  - Scoreboards
  - Protocol Bus Functional Models (BFMs)
- **Pro:** Very easy to integrate
- **Con:** Can’t reuse task/function-driven components



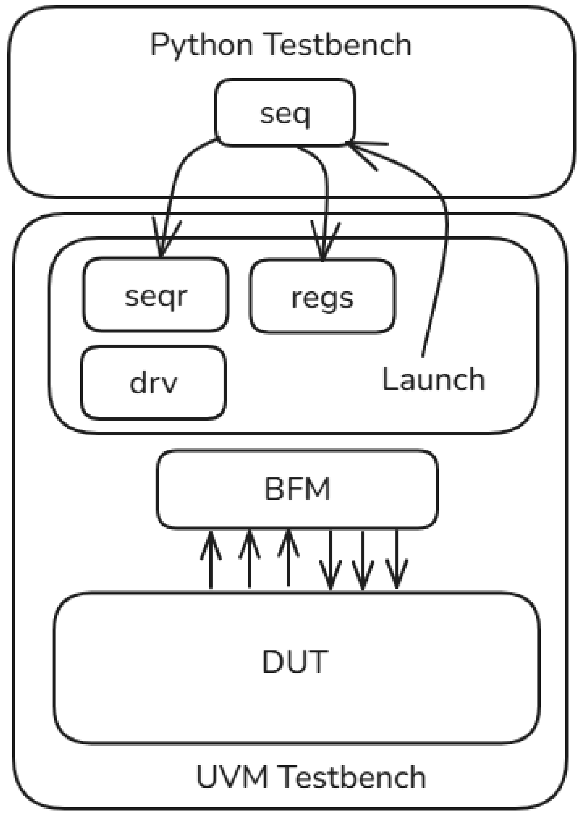
Function-Wrapper Integration

- Integrates via generated function wrapper code
  - PyHDL-IF, pysv, pybind are examples
- Focus Python on tasks without reusable SV
  - Tests
  - Scoreboards
- **Pro:** Enables reuse of all SystemVerilog classes
- **Con:** High effort to create and maintain wrappers



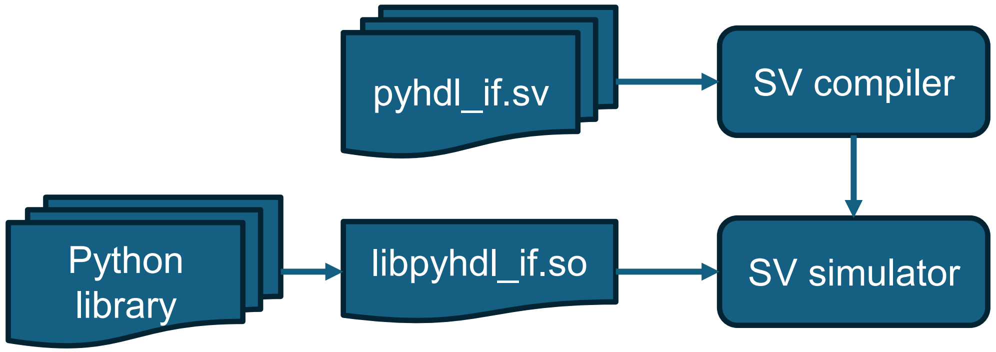
PyHDL-IF UVM: Best of Both

- Low integration effort
- High UVM infrastructure reuse
- Leverages commonality of the UVM API
- Three key capabilities
  - Run Python behavior from UVM
  - Pre-defined APIs for interacting with UVM infrastructure
  - Support Python development tools



Adding PyHDL-IF UVM to your Testbench

- Three Components
  - SystemVerilog packages
  - DPI library
  - Python library
- Steps
  - Compile SV packages
  - Use DPI library



```
% compile-sv pyhdl_if.sv pyhdl_uvm.sv
```

```
% hdlsim -sv_lib pyhdl_if
```

Running Python Behavior from UVM

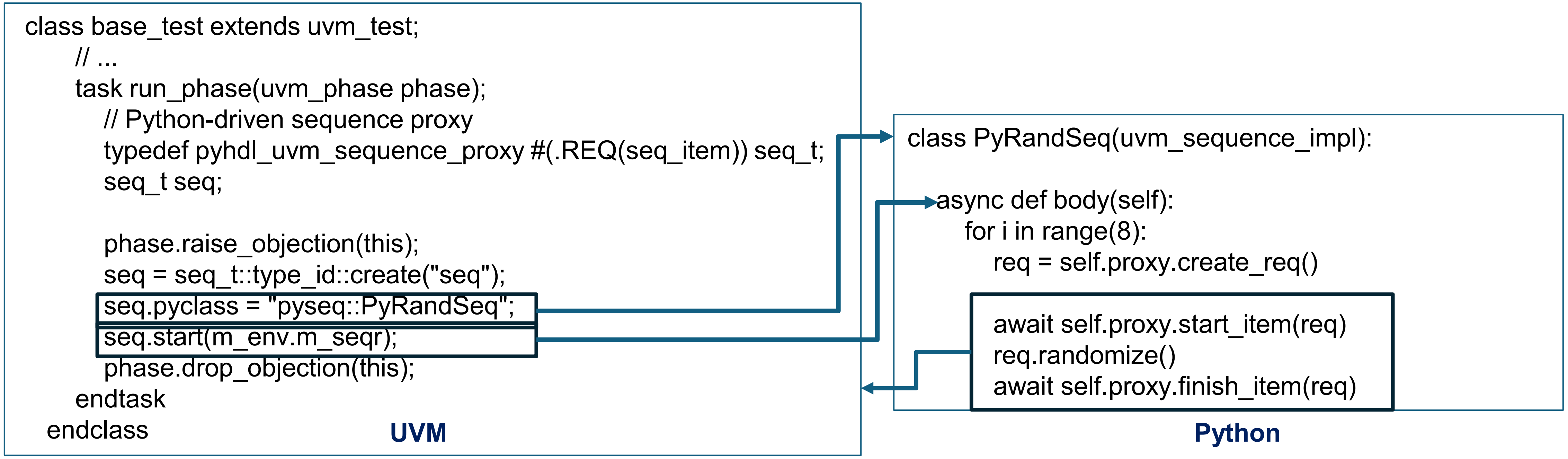
- PyHDL-IF provides SystemVerilog *proxy* classes
  - pyhdl_uvm_sequence_proxy
  - pyhdl_uvm_component_proxy

- Allow UVM class instances to be implemented in Python
  - Sequence: pre_body, body, post_body
  - Component: phases

- Easily synchronize Python behavior with UVM behavior

Proxy Class Example

- UVM environment creates *proxy* class instance
  - Specifies Python class to run
- Use proxy-class instance like a normal UVM class



```
class base_test extends uvm_test;
      // ...
      task run_phase(uvm_phase phase);
         // Python-driven sequence proxy
         typedef pyhdl_uvm_sequence_proxy #(.REQ(seq_item)) seq_t;
         seq_t seq;

         phase.raise_objection(this);
         seq = seq_t::type_id::create("seq");
         seq.pyclass = "pyseq::PyRandSeq";
         seq.start(m_env.m_seqr);
         phase.drop_objection(this);
      endtask
   endclass
```

```
class PyRandSeq(uvm_sequence_impl):

    async def body(self):
        for i in range(8):
            req = self.proxy.create_req()

            await self.proxy.start_item(req)
            req.randomize()
            await self.proxy.finish_item(req)
```

Mapping UVM/SV APIs to Python

- Most UVM APIs map directly to Python without change
- Some SV conventions must be changed to be *Pythonic*
- Example: multiple return values
  - SV generally uses *output* parameters
  - Python generally returns a *tuple* of values

SystemVerilog

```
function bit get_config_object (
    string field_name,
    inout uvm_object value,
    input bit clone=1);
```

```
function void my_c::build_phase(uvm_phase phase);
  uvm_object obj;
  bit has;

  has = comp.get_config_object("cfg", obj);

endfunction
```

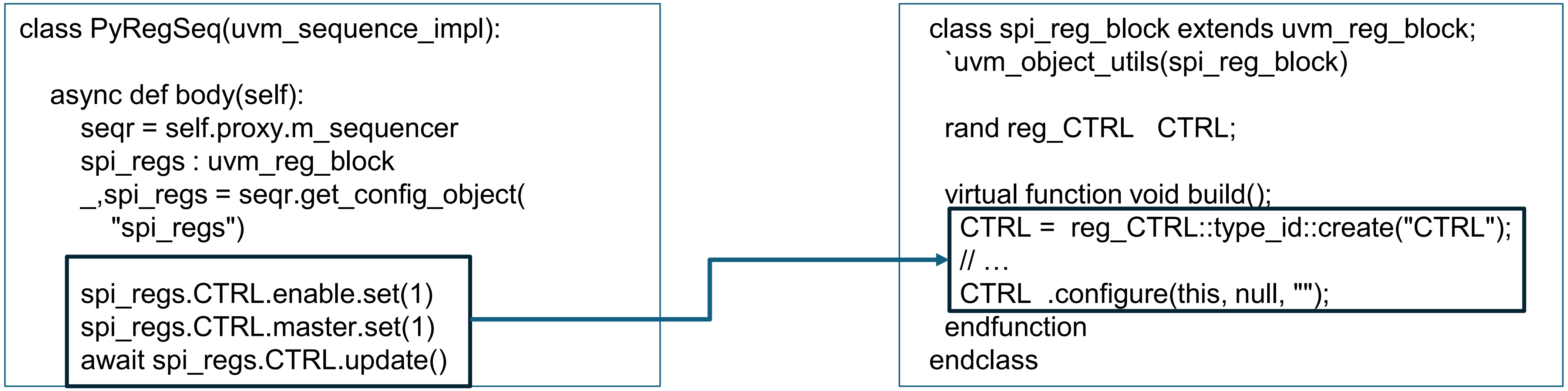
Python

```
def get_config_object(self,
  name : str,
  clone : bool=True) -> Tuple[bool, uvm_object]:
```

```
def build_phase(self, phase):
  has, obj = comp.get_config_object("cfg")
```

Accessing Name-Registered UVM Objects

- Some UVM objects are registered by name with UVM
  - Component instances
  - Register blocks, registers, register fields



```
class PyRegSeq(uvm_sequence_impl):

    async def body(self):
        seqr = self.proxy.m_sequencer
        spi_regs : uvm_reg_block
        _,spi_regs = seqr.get_config_object(
            "spi_regs")

        spi_regs.CTRL.enable.set(1)
        spi_regs.CTRL.master.set(1)
        await spi_regs.CTRL.update()
```

```
class spi_reg_block extends uvm_reg_block;
  `uvm_object_utils(spi_reg_block)

  rand reg_CTRL   CTRL;

  virtual function void build();
    CTRL =  reg_CTRL::type_id::create("CTRL");
    // …
    CTRL  .configure(this, null, "");
  endfunction
endclass
```

- User code generally accesses these as named fields
- PyHDL-IF supports the same user experience

Accessing Data Fields

- Many UVM objects use plain-data fields
  - Sequence-item fields
  - Sequence control knobs
- Goal: Allow access from Python
  - See transaction-field values -- scoreboards
  - Control stimulus and randomization
- PyHDL-IF Requirements
  - Obtain field names and types
  - Get/Set values

```
class seq_item extends uvm_sequence_item;
  rand bit [7:0]   addr;
  rand bit         write; // 1=write, 0=read
  rand bit [31:0] data;
  rand bit [3:0]   tid;

  `uvm_object_utils_begin(seq_item)
    `uvm_field_int(addr , UVM_ALL_ON)
    `uvm_field_int(write, UVM_ALL_ON)
    `uvm_field_int(data , UVM_ALL_ON)
    `uvm_field_int(tid  , UVM_ALL_ON)
  `uvm_object_utils_end

endclass
```

Accessing Data Fields: Field Layout

- Must identify field names and types
- UVM *sprint* displays object contents
  - uvm_field_* macros automate
- PyHDL-IF parses to extract names/types
  - On-demand, per-type, for efficiency

```
class seq_item extends uvm_sequence_item;
  rand bit [7:0]   addr;
  rand bit         write; // 1=write, 0=read
  rand bit [31:0]  data;
  rand bit [3:0]   tid;
  // …
endclass
```

```
------------------------------------
Name       Type       Size  Value
------------------------------------
seq_item  seq_item  -     @550
  addr     integral  8     'h43
  write    integral  1     'h0
  data     integral  32    'ha8136875
  tid      integral  4     'hc
------------------------------------
```

Accessing Data Fields: Field Values

- UVM defines pack/unpack field-data methods
  - Saves/restores value of fields
  - Automated with uvm_field* macros
  - Caller must know data layout

```
class uvm_object;
  function int pack_ints (
    ref int unsigned intstream[],
    input uvm_packer packer=null);

  function int unpack_ints (
    ref int unsigned intstream[],
    input uvm_packer packer=null);
```

- PyHDL-IF implements pack/unpack
  - Pack: Returns dynamically-defined Python type
  - Unpack: Accepts object with named fields
- Supports direct field access as well

Example: Accessing Data Fields

- Sequence ‘knobs’ control randomization
- Field access enables Python control

```
class seq_item extends uvm_sequence_item;
  bit          ctrl_addr_page;
  bit[1:0]     addr_page;

  rand bit [7:0]   addr;
  // …
  constraint addr_page_c {
    (ctrl_addr_page) -> addr[7:6] == addr_page;
  }
endclass
```

UVM

```
async def body(self):
  # Exercise each page in turn
  for i in range(4):
    req = self.proxy.create_req()

    # Configure the knobs
    req.ctrl_addr_page = 1
    req.addr_page = i

    # Randomize with control knobs
    await self.proxy.start_item(req)
    req.randomize()
    await self.proxy.finish_item(req)
```

Python

Supporting Development Tools

- Python has a rich development-tool ecosystem
  - Integrated Development Environments (IDEs)
  - Type checkers
  - AI Assistants
- Development tools operate on Python source
- PyHDL-IF supports development tools by
  - Generating Python *projection* of UVM types
  - Providing tools a Python view of the UVM testbench

Creating Python Projections of UVM Classes

- PyHDL-IF provides a UVM test for type extraction
- Uses Factory to discover registered types
- Generates corresponding Python classes

```
- name: extract-py-mirrors
  uses: "hdlsim.${{ sim }}.SimImage"
  needs: [sim-img, pyhdl-if.DpiLib]
  with:
    plusargs:
    - UVM_TESTNAME=pyhdl_uvm2py
    - pyhdl_outdir=${{ rootdir }}/python
```

Sim-Run workflow

```
@dc.dataclass
class MemTx(uvm_sequence_item):
    addr : int = dc.field(default=0)
    we : bool = dc.field(default=False)
    data : int = dc.field(default=0)
    size : int = dc.field(default=0)
```

Python projection class

Example: Creating a Scoreboard with AI

- Provide Python API, Python class, and prompt

```
Implement the write method in mem_scoreboard:
- Must check that accesses are only to 0..0xFF,
  0x200..0x2FF, 0x400..0x4FF, 0x600..0x6FF
- Writes may only be to the ranges 0x600..0x6FF
- Track writes and compare reads against the
  expected value.
- Report an error on mismatch

Write some unit tests in mem_scoreboard/tests.
Exercise corner cases and ensure tests pass
before concluding the task
```

Prompt

Scoreboard API

```python
class MemScoreboard(object):
    def write(self, t : MemTx):
        pass
```

```python
def test_write_mask_size1_and_mismatch():
    sb = make_sb()
    # Only low byte should be stored/compared
    sb.write(MemTx(addr=0x600, we=True,
        data=0x1234, size=1))
    sb.write(MemTx(addr=0x600, we=False,
        data=0x34, size=1))
    assert sb.errors == 0
    # Mismatch on the next read
    sb.write(MemTx(addr=0x600, we=False,
        data=0x35, size=1))
    assert sb.errors == 1
```

Scoreboard Test

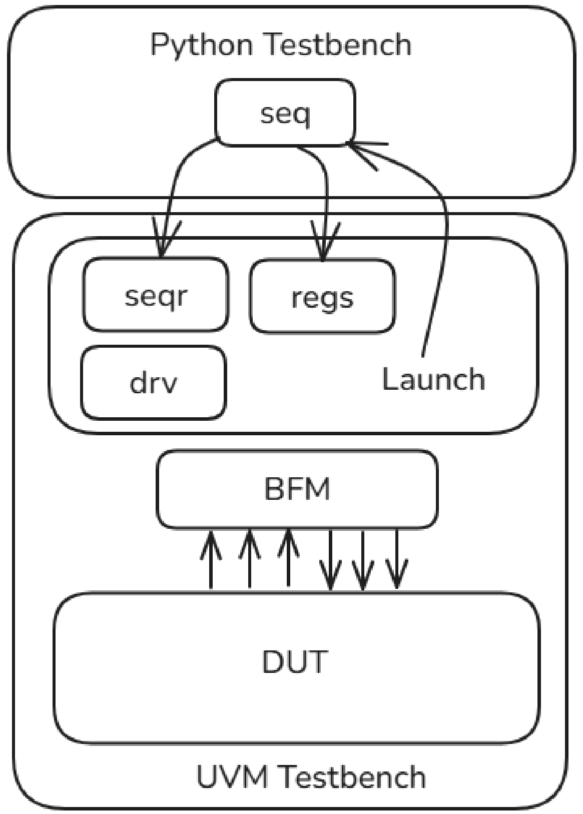
- Python class provides key context
- AI assistant creates scoreboard+tests

Properly Introducing Python to your UVM Testbench

- Using Python in UVM Testbench environments has many benefits
  - Robust ecosystem of libraries
  - Large set of development tools
  - Popular, well-known language

- PyHDL-IF UVM simplifies UVM reuse with Python
  - Call Python behavior from UVM
  - Dynamically access UVM objects and fields
  - Read/write registers and run sequences
  - Support development tools with a Python view of UVM

- PyHDL-IF is Apache 2.0: https://github.com/fvutils/pyhdl-if



COPYRIGHT AND DISCLAIMER

2026

DESIGN AND VERIFICATION™

DVCON

CONFERENCE AND EXHIBITION

UNITED STATES

SANTA CLARA, CA, USA
MARCH 2 – 5, 2026