Using Python in Verification

ALPINUM CONSULTING

Agenda

- Intro to Python
- Using Python for verification
- CoCoTB
- UVM
- Python for UVM

Intro to Python

ALPINUM CONSULTING

Introduction to Python

- Released in 1991, Python is a high-level, interpreted, programming language
- that is simple to write, read and use (syntax & semantics resemble C & C++)
- Now widely used in web development, data science, AI/ML, hardware, …
  - Design & Verification

TIOBE Index, Sept 2024
(https://www.tiobe.com/tiobe-index/)

Sep 2024	Sep 2023	Change	Programming Language	Ratings	Change
1	1		Python	20.17%	+6.01%
2	3	↑	C++	10.75%	+0.09%
3	4	↑	Java	9.45%	-0.04%
4	2	↓	C	8.89%	-2.38%
5	5		C#	6.08%	-1.22%
6	6		JavaScript	3.92%	+0.62%

“Hello World” ☺

Python

```
print("Hello, World!")
```

Java

```
public class Hello {

  public static void main(String[] args) {
    System.out.println("Hello, World!");
   }

}
```

Basic input

- An input statement, variable = input(“message"), has three parts:
  - variable = value stored in memory.
  - The input() function reads one line of input from the user. A function is a named, reusable block of code that performs a task when called. The input is stored in the computer's memory and can be accessed later using the variable.
  - message indicates the program is waiting for input.

Keywords (these are reserved)

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
synch	elif	if	or	yield

Operator Precedence

Operator	Description	Example	Result
()	Parentheses	(1 + 2) * 3	9
**	Exponentiation	2 ** 4	16
+, –	Positive, negative	-math.pi	-3.14159
*, /	Multiplication, division	2 * 3	6
+, –	Addition, subtraction	1 + 2	3

Types in Python

Category	Basic Type
int	Integer numbers
float	Floating-point numbers
complex	Complex numbers
str	Strings and characters
bool	Boolean values

Category	Types
Sequence	`list`, `tuple`, `range`, `str`
Mapping	`dict`
Set	`set`, `frozenset`
Binary	`bytes`, `bytearray`, `memoryview`

Libraries in Python

- A library is a group of modules that contain functions, classes and methods to perform common tasks like data manipulation, math operations, web scraping and more.

```
# Importing entire libraries (This is a comment)
import math
import pandas as pd
```

Rank	Library	Primary Use Case
1	NumPy	Scientific Computing
2	Pandas	Data Analysis
3	Matplotlib	Data Visualization
4	SciPy	Scientific Computing
5	Scikit-learn	Machine Learning
6	TensorFlow	Machine Learning/AI
7	Keras	Machine Learning/AI
8	PyTorch	Machine Learning/AI
9	Flask	Web Development
10	Django	Web Development
11	Requests	HTTP for Humans
12	BeautifulSoup	Web Scraping
13	Selenium	Web Testing/Automation
14	PyGame	Game Development
15	SymPy	Symbolic Mathematics
16	Pillow	Image Processing
17	SQLAlchemy	Database Access
18	Plotly	Interactive Visualization
19	Dash	Web Applications
20	Jupyter	Interactive Computing

Objects in Python

- In Python, objects are instances of classes.
- A class serves as a blueprint for creating objects, encapsulating data and methods.

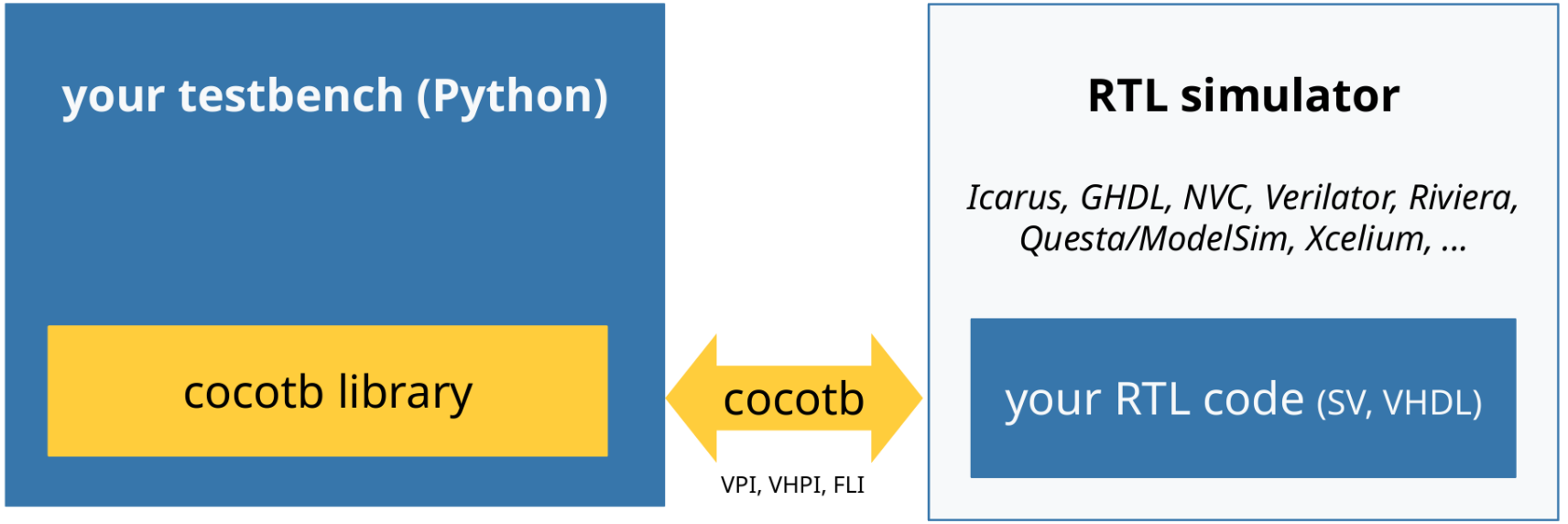
```
# Define a class
class Dog:
  def __init__(self, name, age):
    self.name = name # Instance attribute
    self.age = age # Instance attribute


# Create an object (instance) of the class
dog1 = Dog("Buddy", 3)


# Access object attributes
print(dog1.name) # Output: Buddy
print(dog1.age) # Output: 3
```

Python in verification and CoCoTB

ALPINUM CONSULTING

What is cocotb?

- An RTL simulator plugin
- A Python library for writing synchronous logic



Why Python for verification?

- Productive to write, easy to read
- Easy to interface with
- Huge existing ecosystem
- Popular language: easy to find engineers
- cocotb is open source
  - You are free to use and modify at will.
  - Developed, maintained, and supported by volunteers.
  - You can contribute. Just open a pull request on GitHub.
  - You should contribute to get improvements into cocotb.
  - The FOSSi Foundation provides a legal and administrative home for the project.

Our first cocotb test

ALPINUM CONSULTING

```
// adder.sv

`timescale 1ns/1ps

module adder   (
 parameter integer DataWidth = 4
) (
 input    logic  [DataWidth-1:0]  a_i,
 input    logic  [DataWidth-1:0]  b_i,
 output logic [DataWidth:0]   x_o
);

 assign x_o = a_i + b_i;
endmodule
```

```
  # test_adder_1.py

import cocotb
from cocotb.triggers import Timer


@cocotb.test
async def adder_basic_test(dut):
    """Test for 5 + 10"""

    dut.a_i.value = 5
    dut.b_i.value = 10

    await Timer(2, units="ns")

    assert dut.x_o.value.to_unsigned() == 14
```

A closer look at design access

Let's look at `dut.x_o.value.to_unsigned()`:

`dut`
: The *root handle* (the VHDL/Verilog toplevel). Passed as first argument to any `@cocotb.test` function. "dut" stands for "device under test".

`dut.x_o`
: A *handle* (pointer) to a top-level signal named `x_o` (could be a port or an internal signal).

`dut.x_o.value`
: A `LogicArray` instance representing the value of signal `x_o`.

`dut.x_o.value.to_unsigned()`
: A Python integer representing the value of the `x_o` as unsigned integer.

A Python script to run Icarus Verilog

```
# run_test_adder_1.py

from cocotb_tools.runner import Icarus


def test_adder_1():
    """Simulate the adder with Icarus Verilog"""

    sim = Icarus()
    sim.build(
        sources=["adder.sv"],
        hdl_toplevel="adder",
        always=True,
    )
    sim.test(hdl_toplevel="adder", test_module="test_adder_1")

if __name__ == "__main__":
    test_adder_1()
```

And go! Oops.

```
❯ python3 run_test_adder_1.py
     -.--ns INFO     gpi                                ..mbed/gpi_embed.cpp:108  in set_program_name_in_venv         Using Python virtual environ
ment interpreter at /home/philipp/src/cocotb-tutorial/.direnv/python-3.11/bin/python
     -.--ns INFO     gpi                                ../gpi/GpiCommon.cpp:101  in gpi_print_registered_impl        VPI registered
     0.00ns INFO     cocotb                             Running on Icarus Verilog version 12.0 (stable)
     0.00ns INFO     cocotb                             Running tests with cocotb v2.0.0.dev0+5d0f4f76 from /path/to/site-packages/cocotb
     0.00ns INFO     cocotb                             Seeding Python random module with 1726854133
     0.00ns INFO     cocotb.regression                  running test_adder_1.adder_basic_test (1/1)
                                                            Test for 5 + 10
     2.00ns INFO     cocotb.regression                  test_adder_1.adder_basic_test failed
                                                        Traceback (most recent call last):
                                                          File "/path/to/01.adder/test_adder_1.py", line 16, in add
er_basic_test
                                                            assert dut.x_o.value.to_unsigned() == 14  # really?
                                                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                                                        AssertionError: assert 15 == 14
                                                         +  where 15 = <bound method LogicArray.to_unsigned of LogicArray('01111', Range(4, 'down
to', 0))>()
                                                         +    where <bound method LogicArray.to_unsigned of LogicArray('01111', Range(4, 'downto'
, 0))> = LogicArray('01111', Range(4, 'downto', 0)).to_unsigned
                                                         +      where LogicArray('01111', Range(4, 'downto', 0)) = LogicObject(adder.x_o).value
                                                         +        where LogicObject(adder.x_o) = HierarchyObject(adder).x_o
     2.00ns INFO     cocotb.regression                  ****************************************************************************************
                                                        ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                                        ****************************************************************************************
                                                        ** test_adder_1.adder_basic_test    FAIL           2.00           0.00       1705.66  **
                                                        ****************************************************************************************
                                                        ** TESTS=1 PASS=0 FAIL=1 SKIP=0                    2.00           0.01        226.25  **
                                                        ****************************************************************************************
```

Recap: our first cocotb test

ALPINUM CONSULTING

What did we learn?

- A test is a `async` Python function decorated with `@cocotb.test`.
- Access any signal in the design through the `dut` root handle.
- Make time pass by using `await Timer()`.
- Use `assert` for checking.
- Invoke the simulator to run the test through a Python script using `cocotb_tools.runner`.

What's new in cocotb 2.0?

- The Python data type of `dut.my_signal.value` changed from `BinaryValue` to `LogicArray`. The interfaces are similar, but you need to be more explicit at times (e.g., when casting to an integer).
- The `yield` syntax is gone.
- `raise TestFailure` is gone.
- Makefiles continue to be provided, but give the new Python runner a try!

Click to edit Master title style

Basic randomization and multi-simulator support

Configure the test runner

```
# test_adder_2.py

import random
import cocotb
from cocotb.triggers import Timer


@cocotb.test
async def adder_randomised_test(dut):
    """Test for adding 2 random numbers multiple times"""

    for _ in range(10):
        a = random.getrandbits(4)
        b = random.getrandbits(4)

        dut.a_i.value = a
        dut.b_i.value = b

        await Timer(2, units="ns")

        assert dut.x_o.value.to_unsigned() == a + b
```

ALPINUM CONSULTING

Intro to UVM

Features of UVM

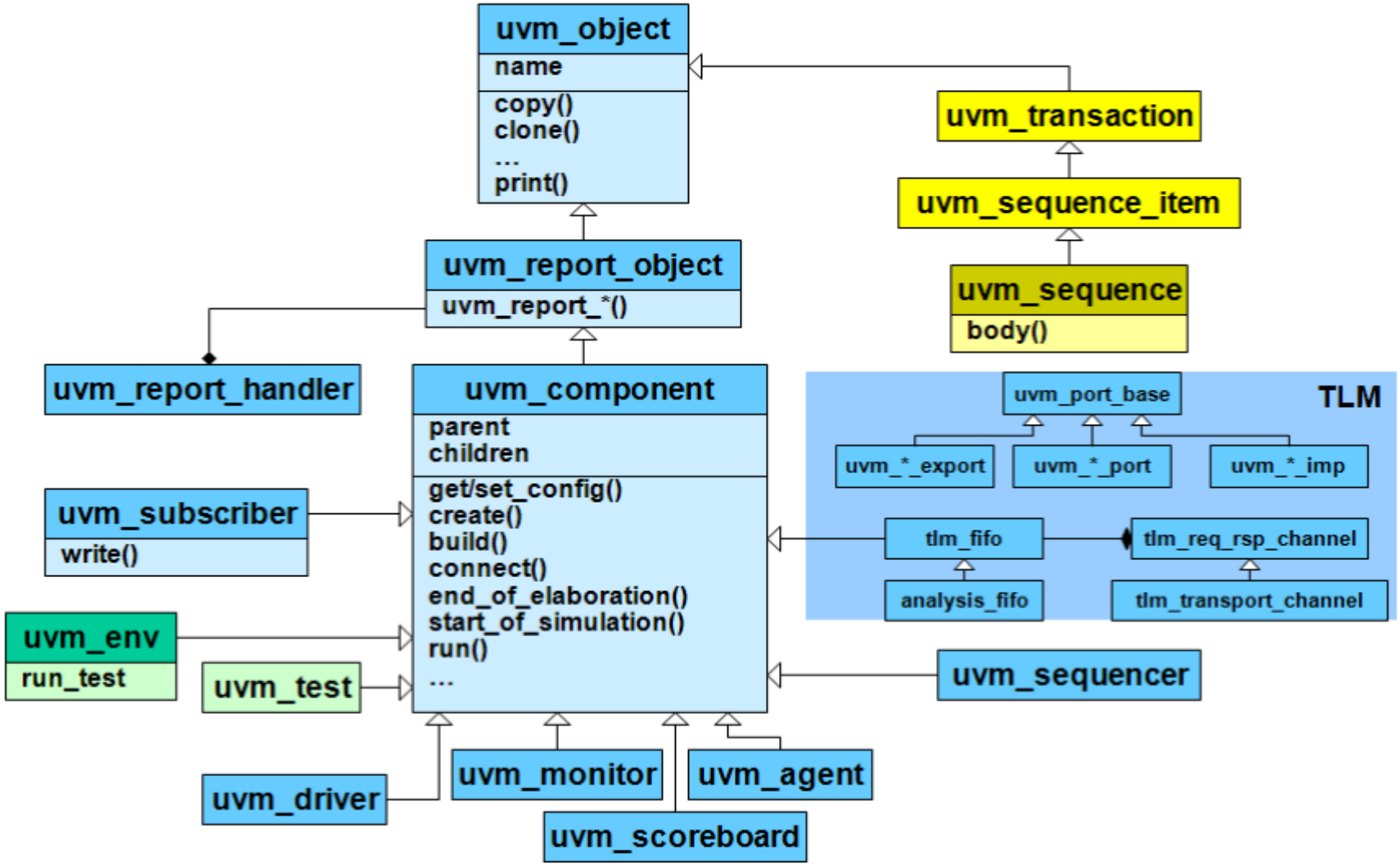
- Standardized Base Classes promotes reuse and consistency
- Constrained Random Testing is automated, thorough stimulus generation
- Functional Coverage ensures comprehensive verification progress
- Factory Mechanism enables flexible testbench configuration
- Modularity and Reusability reduces development time for future projects
- TLM Interfaces simplifies and speeds up component interaction
- Suitable for small to complex designs

Introduction to the UVM library

- The UVM library provides all the building blocks that we require to build modular, scalable, reusable, verification environments
- This library contains base classes, utilities, and macros to support the entire verification process

- To use UVM Library, the user needs to:
  - Compile uvm_pkg.sv file
  - Import uvm_pkg into the desired scope
  - Include a file that contains uvm macros

UVM Class Hierarchy



Foundational building blocks

Dynamic components

Top-level structure

△ is-a relationship

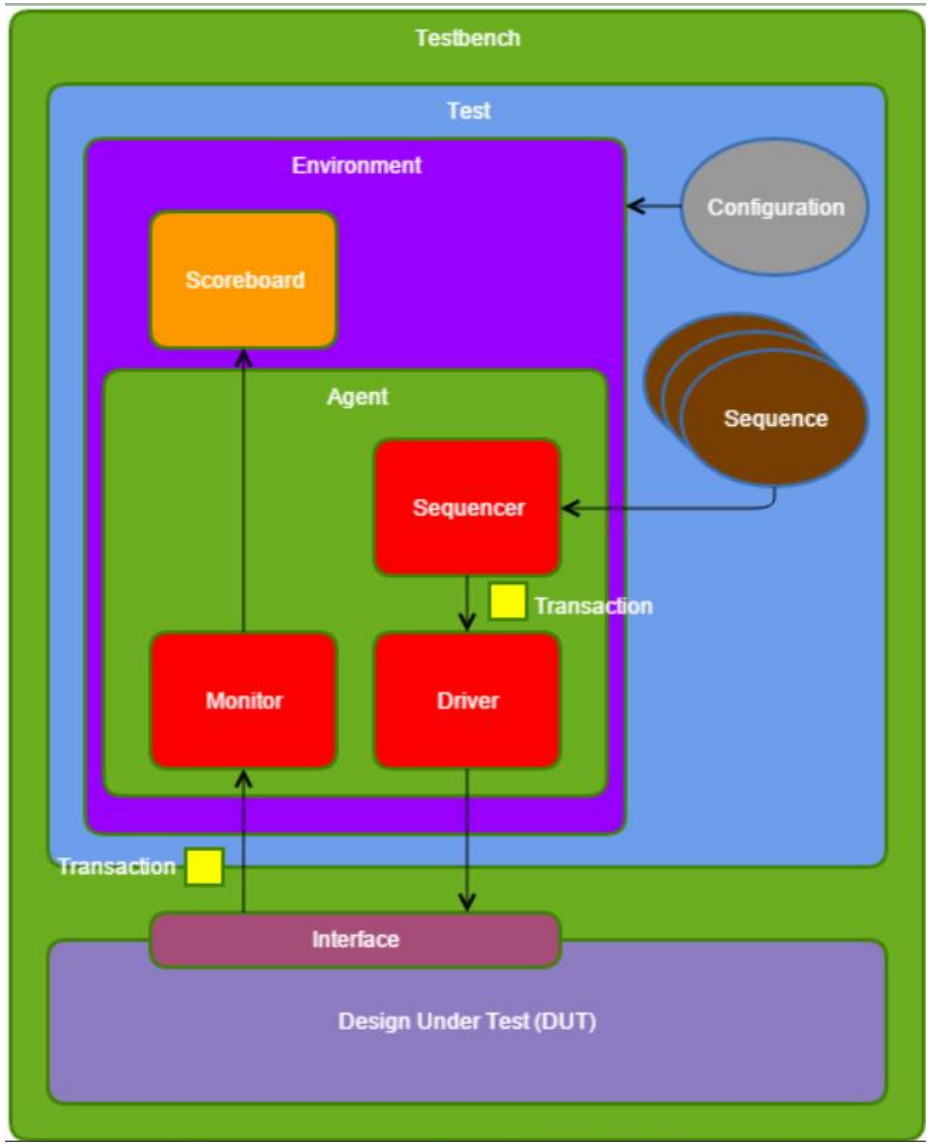
◆ has-a relationship

https://www.cse.scu.edu/~m1wang/verification/Verification.pdf

Structure of a UVM Test Bench

- A UVM testbench is a structured and reusable environment designed to verify a Design Under Test (DUT)
- It follows a modular approach and is composed of various components, each with specific roles in driving, monitoring, and analyzing the DUT's behavior

https://desn.org.uk/introduction-to-semiconductor-design-verification-dv/



ALPINUM CONSULTING

Python and UVM

Using a Python-only UVM Approach

ALPINUM CONSULTING

- Many UVM environments leverage Python-based frameworks like pyuvm or cocotb, which expect Python files to have the .py extension for proper execution.
- If you are using pyuvm (a Python implementation of UVM), your Python script might look like this:

```
# Filename: testbench.py
from pyuvm import *
@pyuvm.test()
class ExampleTest(uvm_test):
  def build_phase(self):
     self.env = AluEnv("env", self)
  async def run_phase(self):
     self.raise_objection()
     await self.env.run_test_sequence()
     self.drop_objection()
```

- Considerations for UVM Environments
  - When integrating Python with SystemVerilog UVM testbenches, ensure that your Python scripts use .py
    - with tools like cocotb or custom DPI (Direct Programming Interface) setups.
  - For hybrid environments (e.g., Python controlling UVM testbenches), frameworks like cocotb or uvm-python simplify the interaction between Python and HDL simulators.

Co-simulation Approach

- Using cocotb with UVM
  - Compile and run with a simulator that supports VPI (e.g., Questa, VCS, Icarus Verilog):
  - Bash: make SIM=questa TOPLEVEL=dut MODULE=test_dut
  - Interaction with UVM:
    - Your SV UVM environment can send/receive transactions via DUT ports or via DPI calls to Python.
    - Python can act as a scoreboard, stimulus generator, or coverage collector.
- Using uvm-python
  - uvm-python is a Python port of UVM 1.2.
  - You can write UVM-style classes in Python and connect them to your SV DUT via cocotb.
  - This lets you reuse UVM methodology but in Python syntax.
- DPI-C Direct Calls
  - If you want **tight coupling** between SV UVM and Python:
    1.Write a **C shim** that embeds Python (Python.h API).
    2.Import that C shim into SV via **DPI-C**.
    3.Call Python functions directly from UVM sequences or scoreboards, etc

```
from uvm import uvm_test, run_test
class MyTest(uvm_test):
  def build_phase(self, phase):
    super().build_phase(phase)
    # Build env, agents, etc.

  def run_phase(self, phase):
    phase.raise_objection(self)
    # Drive transactions here
    phase.drop_objection(self)

if __name__ == "__main__":
  run_test(MyTest)
```

Systemverilog

```
import "DPI-C" function void py_process(input int data);
initial begin
  py_process(42);
end
```

C (py_shim.c):

```
#include <Python.h>
void py_process(int data) {
        Py_Initialize();
        PyObject* pModule = PyImport_ImportModule("my_py_module");
        PyObject* pFunc = PyObject_GetAttrString(pModule, "process_data");
        PyObject* pArgs = Py_BuildValue("(i)", data);
        PyObject_CallObject(pFunc, pArgs);
        Py_Finalize();
}
```

Co-simulation Approach

ALPINUM CONSULTING

- Simulator Support
  - You need a simulator that supports:
    - VPI for cocotb (Icarus, Questa, VCS, Xcelium, Riviera-PRO)
    - DPI-C for direct Python embedding
    - PLI if using older flows
- Considerations
  - If you want minimal disruption to your existing SV UVM testbench
    - use cocotb to connect Python to your DUT and let UVM run as usual.
  - If you want to write UVM entirely in Python
    - use uvm-python
  - If you have an existing SV-UVM database, then co-simulate
    - for example, SystemVerilog UVM sequences can call a Python function to generate stimulus dynamically, etc

Summary

- Intro to Python
- Using Python for verification
- CoCoTB
- UVM
- Python for UVM