gem5 and Python Programming

The purpose of this session is to introduce you to Python programming and how it is used in gem5.

Key idea: gem5 Interprets Python

The gem5 simulator can be thought of as a CPP program that interprets a Python script which defines the simulation.

(**WARNING**: This is a simplification.
Some simulation configuration information is present in the C++ code.
However this simple idea can serve as a useful mental model.)

The Python script *imports* simulated components (**SimObjects**) and uses the Python configuration script to specify their configuration and interconnections to other SimObjects.

```
from m5.objects import CPU, L1Cache

cpu = CPU() # Create the CPU SimObject
cpu.clock = '1GHz' # Set it's parameters
cpu.l1_cache = L1Cache(size="64kB") # Connect it to other SimObjects.

# ... more configuration ...
```

`CPU` and `L1Cache` are not real SimObjects but this serves as an example of how to use the Python configuration script.

Running a script with gem5

The following is an example of how to run a gem5 simulation with a Python configuration script.
The `gem5` binary is used to run the simulation and has parameters (like `--outdir`) which can be set to configure the simulation.
The Python configuration script is passed to `gem5`.
All the arguments after the Python configuration script are passed to the Python script.

```
gem5 --outdir=mydir my-simulation.py --num-cores 4
```

The syntax is `gem5 {parameters to gem5} {gem5 python config script} {script parameters}`.

While the gem5 configuration script is mostly Python, it has some special features and limitations.
We will cover these in this session.
The most important distinction is that the gem5 binary gives the script the `m5` module which provides the interfaces between the configuration script and the gem5 simulator.

Exercise: Literally "Hello world" in gem5

1. Create a file called "mysim.py" and add the following:

```
print("hello from the config script")
```

Execute the script with gem5:

```
gem5 mysim.py
```

Notes

- End the python file name with ".py".
- Don't use dashes in the file name (use underscores instead). Python can't import files with dashes in the name so best to avoid them.

Getting Started with Python: Primitives

At its innermost level, Python has 4 primitive data types.
All other data types are built on top of these.

- Integers (int).
- Floating point numbers (float).
- Strings (str).
- Booleans (bool).

These are the basic building blocks of all Python programs and can be set and used in operations in various ways.

Primitive: Integers

materials/01-Introduction/03-python-background/02-primitives-int.py can be used as a reference for basic integer usage.
This tutorial will cover the basics.

Declaring Integers

```
x = 1
y = 2
```

Basic Integer Operations

```
a = x + y
b = x - y
c = x * y
d = x / y

# Values printed using f-strings.
print(f"a: {a}, b: {b}, c: {c}, d: {d}")
```

On f-strings: f-strings are a way to format strings in Python.
They are defined by the `f` before the string and allow you to insert variables into the string by wrapping them in curly braces `{}`.
We're jumping ahead a bit here but they are very useful and we recommend using them to output variables in your code.

Primitive: Floats

materials/01-Introduction/03-python-background/03-primitives-float.py can be used as a reference for basic float usage.

Declaring Floats

Floats are a primitive data type in Python. They are "real numbers" and are declared like so. Here we declare a variable `x` and assign it the literal value `1.5`.

```
x = 1.5
```

Basic Float Operations

```
# Like integers, floats can be set using arithmetic operations.

# Set variable `y`` to `10.5 + 5.5`.`
y = 10.5 + 5.5

# Set variable `z` to `y - x` (in this case, 16 - 1.5).
z = y - x
print(f"Value of z: {z}")
```

```
# Multiplication
multi_xy = x * y
print(f"Value of multi_xy: {multi_xy}")

# Division
div_xy = y / x
print(f"Value of div_xy: {div_xy}")
```

Primitive: Strings

materials/01-Introduction/03-python-background/04-primitives-string.py can be used as a reference for basic string usage.

```
# Strings are a primitive data type in Python. They are sequences of characters
# and are declared like so. Here we declare a variable `x` and assign it the
# literal value `"Hello World!"`.

x = "Hello World!"
print(x)

# Concatenation of two strings
# Note the use of the a literal string ("GoodBye!") and the variable `x`.
y = x + " GoodBye!"
print(y)
```

```
# We use the "f-string" syntax to insert the value strings inside other
# strings. The contents between the curly braces are evaluated as Python.
# In the following we concatenate x with " GoodBye " and the value of x + y
# ("Hello World! GoodBye!"). This z will be set to
# "Hello World! GoodBye Hello World! Goodbye!"
z = f"{x} GoodBye {x + y}"
print(z)
```

Primitive: Booleans

materials/01-Introduction/03-python-background/05-primitives-bool.py can be used as a reference for basic boolean usage.

```
# Bools are a primitive data type in Python. They are "True" or "False" and are
# declared like so. Here we declare a variable `x` and assign it the literal
# value `True`.
x = True
print(f"Value of x: {x}")
```

```
# Bools can be set using logical operations of literals or other bool
# variables. These logical operations are `is` `and`, `or`, and `not` and are
# used to compare values.

# Set `y`` to True if  `x and True`.
y = x and True
print(f"Value of y: {y}")

# Set `z`` to True if `x or False`.
z = x or False
print(f"Value of z: {z}")

# Set `a`` to True if `not x`.
a = not x
print(f"Value of a: {a}")
```

Boolean Comparison

```
# The ==, !=, <, >, <=, and >= operators can be used to compare values of other
# primitive data types. The result of these operations is a bool.

# Set `b` to True if `1 + 1` is equal to `2`.
b = (1 + 1) == 2
print(f"Value of b: {b}")

# Set `c` to True if `1 + 1` is not equal to `2`.
c = (1 + 1) != 2
print(f"Value of c: {c}")
```

```
# Set `d` to True if `1 + 1` is less than `3`.
d = (1 + 1) < 3
print(f"Value of d: {d}")

# Set `e` to True if `1 + 1` is greater than `3`.
e = (1 + 1) > 3
print(f"Value of e: {e}")

# Set `f` to True if `1 + 1` is less than or equal to `2`.
f = (1 + 1) <= 2

# Set `g` to True if `1 + 1` is greater than or equal to `2`.
g = (1 + 1) >= 2
```

Getting started with Python: Collections

Python has many built-in collection types though the most commonly used are lists, dictionaries, and sets. In all cases they serve to store multiple variables within the single collection variable

Lists are ordered collections of variables. Duplicates allowed.
The are set in square brackets.

```
a_list = [1, 1, 2]
```

More on lists can be found at [materials/01-Introduction/03-python-background/06-collections-list.py]

Sets are unordered collections of variables. Duplicates are not allowed.
They are set in curly braces.

```
a_set = {"one", "two", "three", "four", "five"}
```

More on examples with sets can be found at materials/01-Introduction/03-python-background/07-collections-set.py.

Dictionaries are collections of key-value pairs. These are effectively sets with each value in the set (the 'key') mapping to another variable (the 'value'). Duplication of keys is not allowed (but duplication of values is).

```
a_dict = {1: "one", 2: "two"}
```

More on examples with dictionaries can be found at materials/01-Introduction/03-python-background/08-collections-dict.py.

Python collections usage

```
# The collections examples from the past few slides
a_list = [1, 1, 2]
a_set = {"one", "two", "three", "four", "five"}
a_dict = {1: "one", 2: "two"}

# Accessing elements in a list
# Each element has an index which can be used to access the element. Indices start at 0
print(a_list[0])
print(a_list[1])

# Add an element to the end of the list. `a_list` will be set to `[1, 1, 2, 1]`.
a_list.append(1)

# Accessing elements in a set. No index is used to access elements in a set.
for element in a_set:
    print(element)
```

```
# Add an element to the set.
a_set.add("six")

# Adding the same to the set again will not change the set.
a_set.add("six")

# Accessing elements in a dictionary
# Elements are accessed by their key.
print(a_dict[1]) # "one"

# Add an element to the dictionary.
# `a_dict` will be set to `{1: "one", 2: "two", 3: "three"}`.
a_dict[3] = "three"

# Adding the same key the dictionary will overwrite the value.
# `a_dict` will be set to `{1: "one", 2: "two", 3: "four"}`.
a_dict[3] = "four"
```

More on Python primitives and collections

Python is rather unique in that it comes with a lot of built-in functionality.
While useful, this does normally mean that there are many ways to do the same thing.

For example, in the following snippets, `dict_1`, `dict_2`, and `dict_3` are all equivalent.

```
dict_1 = {'key_one': "one", 'key_two': "two"}

dict_2 = dict(key_one="one", key_two="two")

dict_3 = dict()
dict_3['key_one'] = "one"
dict_3['key_two'] = "two"
```

Those of you new to Python may want to dedicate some time this evening to go through examples using Python's "built-in" functions.

List Comprehensions

List comprehensions are a way to create lists in Python, and commonly used in gem5.

They allow for creation of a list in a single line of code by interating over another list and applying an operation to each element.

The following code creates a list of even numbers from 1 to 20:

```
even_numbers = [x for x in range(1, 21) if x % 2 == 0]
```

Its non-comprehension equivalent would be:

```
even_numbers = []
for x in range(1, 21):
    if x % 2 == 0:
        even_numbers.append(x)
```

The syntax for a list comprehension is:

```
[expression for item in iterable if condition]
```

For example, let's say we want a list of all the prices for items in a store that are less than $10. Let's assume `store` is a collection of items and each item has a function `get_price` that returns the price of the item and a function `get_name` that returns the name of the item.

The following would get the the list of items.

```
item_under_10 = [item.get_name() for item in store if item.get_price() < 10]
```

It is also possible to nest list comprehensions.

For example, let's say we have a list of lists and we want to flatten it.

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened_list = [item for sublist in list_of_lists for item in sublist]
```

In the above we have a list of lists.
The outer loop iterates over each sublist in the list of lists.
The inner loop iterates over each item in the sublist.

The non-comprehension equivalent would be:

```
flattened_list = []
for sublist in list_of_lists:
    for item in sublist:
        flattened_list.append(item)
```

Python `if`

```
condition = True and False

if condition:
    print("The condition is True")
else:
    print("The condition is False")
```

materials/01-Introduction/03-python-background/09-if-statements.py can be used as a reference for basic `if` usage.

`if` Notes

- Python doesn't use braces to define blocks of code. Instead it uses indentation.
- The `print` statements are indented to show that they are part of the `if` block (for example, "The condition is True" `print` will only be execute if `condition` is `True`).
- Make sure your indentation is consistent. Python will throw an error if it is not.
- In gem5 we use 4 spaces for indentation.

Python `for`

`for` iterates through a collection of items.

```
for value in [1, 2, 3]:
    print(value)
```

Again, the `print` statement is indented to show that it is part of the `for`.

materials/01-Introduction/03-python-background/10-for-loops.py can be used as a reference for basic `for` usage.

Python `while`

`while` will execute a block of code until a condition is `False`.

```
counter = 0
while counter < 3:
    print(counter)
    counter += 1
```

materials/01-Introduction/03-python-background/11-while-loop.py can be used as a reference for basic `while` usage.

Note: The `counter += 1` line is shorthand for `counter = counter + 1`.
This sets the counter value to the current counter value plus 1. E.g.,
if the counter is 0, `counter += 1` will set the `counter` variable to 1.

Python functions

Functions are defined using the `def` keyword.

```
def my_function(arg1, arg2):
    return arg1 + arg2

result = my_function(1, 2)
print(result) # 3
```

The following style of explicitly referencing the arguments is also common.

```
def my_function(arg1: int, arg2: int) -> int:
    return arg1 + arg2
```

We strongly recommend using typing hints in your functions. This improves code readability and helps catch errors.

materials/01-Introduction/03-python-background/12-function.py can be used as a reference for basic function usage.

Importing Code

Python allows you to import code from other files.

Let's say we have functions `add`, `subtract`, and `multiply` in a file called `math_funcs.py`:

```
# Contents of math_funcs.py
def add(a: int b: int) -> int:
    return a + b

def subtract(a: int, b: int) -> int:
    return a - b

def multiply(a: int, b: int) -> int:
    return a * b
```

We can import the functions and use them with:

```
from math_funcs import add, subtract, multiply

print(add(1,2))
print(subtract(4,2))
print(multiply(3,3))
```

If math_funcs.py is in a directory, say "math_dir" we can use:

```
from math_dir.math_funcs import add, subtract, multiply
```

A completed and extended example can be found at materials/01-Introduction/03-python-background/13-importing-code.py.

Python Generators

Generators are a way to create iterators in Python. They are similar to functions but instead of returning all the values at once, they yield them one at a time.

```
def my_generator():
    yield 1
    yield 2
    yield 3

for value in my_generator():
    print(value)
```

The difference in syntax is the `yield` keyword.
This is used to return a value from the generator.

In addition to being more memory efficient, generators are useful for creating infinite sequences.

The advantage of generators is you can create infinite sequences.

```
def infinite_flip_flop() -> Generator[bool]:
    bool val = True
    while True:
        yield val
        val = not val
```

The above generator will yield `True`, `False`, `True`, `False`, `True`, `False`, and so on indefinitely.

Though returning a list can be tempting, if you want to iterate over a sequence of values one at a time, a generator is the way to go.

gem5 and Object Oriented design

gem5 utilizes Object Oriented Design (OOD) to model the components of a computer system. This is a powerful way to model complex systems and is a common design pattern in software engineering. In short, it is a way to encapsulate data and functions that logically belong together in entities called "objects".

Classes allow you to create your own data types in Python. They are a way to bundle data and functionality together in a single unit. A class is a blueprint for an object. It defines the attributes and methods object instances of the class will have. For example, we can have class `Car` with attributes like `color`, `make`, `model`, and methods like `drive`, `stop`, `park`.
When we create an object of the class `Car`, we can set the attributes of the car object like `color`, `make`, `model` and call the methods like `drive`, `stop`, `park`.
Though each object of the class `Car` will have the same attributes and methods, the values of the attributes can be different for each object.

Basic class

Code for this example can be found at materials/01-Introduction/03-python-background/14-basic-class-and-object.py.

Let's create a simple class with some object instantiations.

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def eat(self, food):
        print(f"{self.name} is eating {food}")

    def sleep(self):
        print(f"{self.name} is sleeping")
```

This defines the class.

The `__init__` method is a special method that is called when an object is created.
It is used to initialize the object's attributes.
Now we can create objects from this class.

```
dog = Animal("Dog", 5)
cat = Animal("Cat", 6)
```

We can access attributes like so:

```
print(f"Name of animal: {dog.name}")
print(f"Age of animal: {dog.age}")
```

and call its methods like so:

```
dog.eat("meat")
dog.sleep()
```

Despite having different values for `name` and `age`, both `dog` and `cat` objects have the have the same type, `Animal`.
Therefore they can be passed to functions that expect an `Animal` object.

```
def feed_animal(animal):
    animal.eat("food")

feed_animal(dog)
feed_animal(cat)
```

Object Oriented Design Terminology you should know

- **Class**: A blueprint for an object. It defines the attributes and methods object instances of the class will have.
  (i.e., `Animal` in our example).
- **Object**: An instance of a class.
  (i.e., `dog` and `cat` in our example).
- **Member Variable**: A variable that is encapsulated within a specific object.
  (i.e., `weight`, `height`, and `name` in our example).
- **Member Function**: A function that is encapsulated within a specific object.
  (i.e., `eat` and `sleep` in our example).
- **Constructor**: A special method that is called to create an object.
  (i.e., `__init__` in our example).
- **Instantiation/Construction**: The creation of an object from a class.
  (i.e., `dog = Animal(100, 5, "Dog")` in our example).

Inheritance

Inheritance allows for a class to be defined in relation to another class. This other class is referred to as the base class, parent class, or super class, with the new class being the derived class, child class, or sub class.

There are many instances where a new class is needed but shares many of the same attributes and methods as an existing class. In these cases it is possible to inherit from the existing class and extend it with new attributes and methods.

Let's imagine we want to add an elephant object using to our Animal class. We want a new member variable `trunk_length` and a new member function `trumpet`. The insight here is Elephants are Animals, but not all Animals are Elephants. An Elephant will always have all the common attributes and methods of an Animal, but not all Animals will have the attributes and methods of an Elephant.

The code for this section can be found at materials/01-Introduction/03-python-background/15-inheritance.py

```
class Elephant(Animal):
    def __init__(self, name, age, trunk_length):
        # Call the constructor of the parent class
        super().__init__(name, age)
        self.trunk_length = trunk_length

    def trumpet(self):
        print("Trumpeting")
```

The Elephant class inherits from the Animal class. This means that the Elephant class has all the attributes and methods of the Animal class. Not only does this save a lot of typing and time by borrowing the attributes and methods of the Animal class, but it also makes the code more readable and maintainable.

Of most importance, an Elephant can be passed as an Animal to any function
So:

```
def print_animal(animal):
    print(f"Name: {animal.name}")
    print(f"Age: {animal.age}")

dog = Animal("Dog", 10)
elephant = Elephant("Dumbo", 10)
print_animal(elephant)
print_animal(dog)
```

However a function that expects an Elephant object will not accept an Animal object. This is because an Elephant is an Animal, but an Animal is not an Elephant.

```
def toot_horn(elephant):
    elephant.trumpet()

# This will work
toot_horn(elephant)

# This will not work
toot_horn(dog)
```

Overriding Methods

Finally subclasses can override methods of the parent class. This is useful when the method of the parent class does not make sense for the subclass. For example, the Elephant class could override the eat method of the Animal class to print a different message when an Elephant eats.

```
class Elephant(Animal):
    def __init__(self, name, age, trunk_length):
        super().__init__(name, age)
        self.trunk_length = trunk_length

    def trumpet(self):
        print("Trumpeting")

    def eat(self, food): # Overriding the eat method
        print(f"{self.name} is eating {food} with its trunk")
```

A piece of code which expects an `Animal` can therefore execute completely different code depending on the type of the object passed to it.

```
def feed_animal(animal):
    animal.eat("food")

feed_animal(dog)
feed_animal(elephant)
```

To return to everyday Object Oriented Design terminology, the `Animal` class is the base class and the `Elephant` class is the derived class.
The derived class can override the methods of the base class.
This means a function which expects a base class object can execute completely different code depending on the type of the object passed to it.

Abstract Classes

In the past couple of examples we've envisioned a simple class `Animal` that has object instantiations. There are cases where you don't want there to be any object instantiations of a class. This is where Abstract Base Classes are useful.
In our case it makes no sense to have a generic Animal when we can create a subclass for each animal type.

Abstract Base Classes are classes that are meant to be inherited from, but not instantiated. They are used to define a common interface for subclasses to implement.

The `abc` module in Python provides the `ABC` class that can be inherited from to create an Abstract Base Class.
Methods do not have to be implemented in the Abstract Base Class, but they can be. This is useful for cases where you wish to enforce that a method is defined in the subclass.

The code for this section can be found at materials/01-Introduction/03-python-background/16-inheritance-with-abstract-base.py

```
from abc import ABC, abstractmethod

class Animal(ABC):
    """
    An abstract class that represents an animal.
    """

    def eat(self, food):
        print("Is eating {food}")

    @abstractmethod
    def move(self):
        raise NotImplementedError("move method not implemented")
```

We can then add animals. Let's say a Dog and a Cat:

```
class Dog(Animal):
    def move(self):
        print("Dog is running")

class Cat(Animal):
    def move(self):
        print("Cat is walking")
```

All we needed to do was specify the unimplemented methods of the Animal class in the subclasses.

We could add a subclass to cat. Let's say "LazyCat", which has
a new method "sleep", unique to it while sharing all other Cat methods.

```
class LazyCat(Cat):
    def sleep(self):
        print("Cat is sleeping")
```

We can instantiate these classes and call their methods, everything except
the abstract base class.

```
dog1 = Dog()
dog2 = Dog()
cat = Cat()
lazy_cat = LazyCat()

dog1.eat("meat")
dog1.move()

dog2.eat("bones")
dog2.move()

cat.eat("fish")
cat.move()

lazy_cat.eat("milk")
lazy_cat.move()
lazy_cat.sleep()
```

More Object Oriented Design terminology

- **Inheritance**: The ability of a class to inherit attributes and methods from another class.
  (i.e., `Elephant` inherits from `Animal` in our example).
- **Base Class**: The class from which attributes and methods are inherited.
  This can also be called the parent class or super class.
  (i.e., `Animal` in our example).
- **Derived Class**: The class that inherits attributes and methods from another class.
  This can also be called the child class or sub class.
  (i.e., `Elephant` in our example).
- **Overriding**: The ability of a subclass to provide a specific implementation of a method that is already provided by one of its superclasses.
  (i.e., `Elephant` overrides the `eat` method of `Animal` in our example).

- **Abstract class**: A class that is meant to be inherited from, but not instantiated directly.
  (i.e., `Animal` in our example).
- **Abstract Method**: A method that is declared in an abstract class, but not implemented.
  To be implemented by subclasses.
  (i.e., `move` in our example).

SimObjects and Object Oriented Design

A SimObject is an object in gem5 that represents a component of the simulated system.
They are instantiated from classes which inherit from the `SimObject` abstract class and encapsulate parameters for the simulated component (e.g, memory size), and the methods it uses to interact with other components in a standard way.

As each of these shares a common base class, gem5 can handle them in consistent manner, despite simulating a wide variety of components.
If a new component is needed then we simply create a child from an the most logical, existing component and extend it with the new functionality.

Worth noting: gem5 also has special parameters called "Ports" which are used to define communication channels between SimObjects.
More on this in future sessions.

A SimObject OO Design example

It's useful in gem5 to take a SimObject and extend it to add new functionality.
gem5 should ideally be open **for extension but closed for modification**.
Modifying gem5 code directly can be difficult to maintain and can lead to
merge conflicts when updating to new versions of gem5.

The following shows an example of of specializing a gem5 SimObject to create an abstract L1 cache. This is then used as a base class for L1 instruction cache.

The code for the following example can also be found at materials/01-Introduction/03-python-background/17-inheriting-from-a-simobject.py

```
from m5.objects import Cache
from abc import ABC

class L1Cache(type(Cache), type(ABC)):
    """Simple L1 Cache with default values"""

    def __init__(self):
        # Here we set/override the default values for the cache.
        self.assoc = 8
        self.tag_latency = 1
        self.data_latency = 1
        self.response_latency = 1
        self.mshrs = 16
        self.tgts_per_mshr = 20
        self.writeback_clean = True
        super().__init__()
```

```
    # We extend the functionality. In this case by adding a method to aid in
    # Adding the cache to a bus and processor.
    # Connecting to the cpu is left unimplemented as it will be different for
    # each type of cache.
    def connectBus(self, bus):
        """Connect this cache to a memory-side bus"""
        self.mem_side = bus.cpu_side_ports

    def connectCPU(self, cpu):
        """Connect this cache's port to a CPU-side port
        This must be defined in a subclass"""
        raise NotImplementedError
```

```
class L1ICache(L1Cache):
    """Simple L1 instruction cache with default values"""

    def __init__(self):
        # Set the size
        self.size = "32kB"
        super().__init__()

    # This is the implementation needed for the L1ICache to connect to the CPU.
    def connectCPU(self, cpu):
        """Connect this cache's port to a CPU icache port"""
        self.cpu_side = cpu.icache_port
```

Sometimes gem5 is a bit different

While the configuration scripts are mostly Python, there are some differences between Python and gem5's Python.
Here are some important differences to keep in mind:

gem5 has a special module `m5`

The `m5` module is a special module that provides the interfaces between the configuration script and the gem5 simulator. This is *compiled into the gem5 binary* and therefore is not a standard Python module. The most common complaint is that `import m5` will be considered an error by most Python IntelliSense tools. However, it is a valid import when the script is interpreted by gem5.

SimObject parameter assignments are special

In most cases Python allows this:

```
class Example():
    hello = 6
    bye = 6

example = Example()
example.whatever = 5
print(f"{example.hello} {example.whatever} {example.bye}")
```

Here we've added another variable the object.

However, gem5 will throw an error if you try to do this with a SimObject.

```
AttributeError: 'example' object has no attribute 'whatever'
```

There are rules on what you can and cannot assign to a SimObject.

SimObjects only allow parameter assignment in 3 cases:

1. The parameter exists in the list of parameters. Ergo you are setting the parameters (`simobject.param1 = 3`).
2. The value you are setting is a SimObject and its variable name does not conflict with a SimObject parameter (`simobject.another_simobject = Cache()`).
3. The parameter name starts with `_`. gem5 will ignore this (`simobject._new_variable = 5`).

SimObject Port assignments are special

Ports are a special type of SimObject variable.
They are used to connect SimObjects together.
The syntax for setting a response and request port is `simobject1.{response_port} = simobject1.{request_port}` (or vice versa).
This is not a traditional `=` assignment and is instead calling a `connect` function on the port.

SimObject Vector parameters are immutable

Vector parameters are vectors of parameter values of other SimObjects.
They are a special type of SimObject parameter.
They are used to store multiple values in a single parameter.
However, unlike typical Python Lists once created, they cannot be changed.
You can't add or remove SimObjects from a vector after it has been created.

```
simobject = ASimObject()
simobject.vector_param = [1, 2]
simobject.vector_param = [3, 4] # This is ok but is just overriding the previous value.
simobject.vector_param.append(5) # This is not permitted.
simobject.vector_param.remove(1) # This is not permitted.
```

The following is a common mistake:

```
processor_simobject.cpus = []
for cpu in range(4):
    processor_simobject.cpus.append(CPU())
```

The correct way to do this is to set the vector parameter in one go:

```
simobject.cpus = [CPU() for _ in range(4)]
```

After simulation initialization, you can't add new variables to a SimObject

```
simobject = ASimObject()
simobject.var1 = 5
simobject.var2 = 6

m5.instantiate()
# Could also be `Simulator`'s `run()` function.

simobject.var3 = 7 # This is not permitted
```

In some cases this may not fail but the change in the SimObject configuration will not be reflected in the simulation.