**Introduction to Simpy (simulation py)-01**

[[Gaurav Kumar](https://gaurav-adarshi.medium.com/?source=post_page-----97b05d1376b9--------------------------------)](https://gaurav-adarshi.medium.com/?source=post_page-----97b05d1376b9--------------------------------)

[Gaurav Kumar](https://gaurav-adarshi.medium.com/?source=post_page-----97b05d1376b9--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff319c9e06fc1&operation=register&redirect=https%3A%2F%2Fgaurav-adarshi.medium.com%2Fintroduction-to-simpy-simulation-in-python-01-97b05d1376b9&user=Gaurav+Kumar&userId=f319c9e06fc1&source=post_page-f319c9e06fc1----97b05d1376b9---------------------post_header-----------)

15 min read

·

Nov 1

1

1

In the realm of computer programming, simulation plays a pivotal role in understanding complex systems, conducting experiments, and making informed decisions. SimPy, short for Simulation Python, is a powerful and versatile simulation framework that allows developers and researchers to create and analyze discrete-event simulations using Python. Whether you’re a novice or an experienced programmer, SimPy offers an intuitive and efficient way to model and simulate various scenarios, making it an indispensable tool in a wide range of fields, from operations research to logistics, and from epidemiology to manufacturing.

This is going to be an extensive n part series on becoming proficient with simpy.

**What is SimPy?**

SimPy is an open-source Python library designed for modeling and simulating discrete-event systems. It provides the building blocks for creating simulations, allowing you to model the flow of time and events in a system accurately. Unlike continuous simulations, where variables change continuously over time, discrete-event simulations focus on events that occur at specific points in time. These events trigger state changes in the system, making SimPy particularly suited for modeling processes involving discrete, often unpredictable events.

**Some key features of SimPy include:**

1. **Processes:** In SimPy, entities in the simulation are represented as processes. These processes can be anything from people in a queue, vehicles in a traffic network, or tasks in a computer system. You can define processes with Python generator functions, making it easy to model complex, asynchronous behaviors.
2. **Events:** Events are the core of SimPy’s modeling approach. Events can be scheduled to occur at specific times or after specific conditions are met. You can create event chains and manage the flow of time in your simulation by scheduling events to trigger changes in the system.
3. **Resources:** In many simulations, resources are limited and need to be shared among different entities. SimPy provides a resource class that allows you to model resource allocation and contention. This is invaluable for modeling scenarios like the allocation of servers in a data center or machines in a factory.
4. **Statistics and Data Collection:** SimPy offers a range of tools for collecting data during the simulation. This includes monitoring events, tracking the time entities spend in various states, and recording other relevant information. These statistics are crucial for analyzing and understanding system behavior.
5. **Integration:** SimPy is highly compatible with other Python libraries and tools. You can use it in conjunction with data analysis libraries like NumPy and Pandas to analyze simulation results and make data-driven decisions.

**SymPy — Installation**

SymPy has one important prerequisite library named **mpmath**. It is a Python library for real and complex floating-point arithmetic with arbitrary precision. However, Python’s package installer PIP installs it automatically when SymPy is installed as follows −

pip install sympy

import sympy  
sympy.\_\_version\_\_  
  
Output:  
'1.12'

**SymPy — Symbolic Computation**

Symbolic computation pertains to the creation of algorithms designed to manipulate mathematical expressions and various mathematical entities. It merges the realms of mathematics and computer science to resolve mathematical expressions by employing mathematical symbols. A Computer Algebra System (CAS), like SymPy, precisely assesses algebraic expressions by utilizing the very symbols employed in conventional manual techniques. To illustrate, consider the computation of the square root of a number through Python’s math module, as demonstrated below −

import math   
print (math.sqrt(25), math.sqrt(7))  
  
Output:  
5.0 2.6457513110645907

You may observe that the approximate calculation of the square root of 7 is presented here. However, in SymPy, square roots of numbers that lack perfect square roots remain uncomputed by default, as demonstrated below:

import sympy   
print (sympy.sqrt(7))  
  
Output:  
sqrt(7)

It is possible to simplify and show result of expression symbolically with the code snippet below −

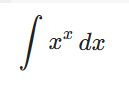
import math  
print (math.sqrt(12))  
  
Output:  
3.4641016151377544

You need to use the below code snippet to execute the same using sympy −

print (sympy.sqrt(12))  
  
Output:  
2\*sqrt(3)

When executed in a Jupyter notebook, SymPy code utilizes the MathJax library to display mathematical symbols in LaTeX format. This is demonstrated in the following code excerpt:

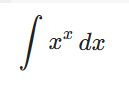
from sympy import \*   
x=Symbol ('x')   
expr = integrate(x\*\*x, x)   
expr



On executing the above command in python shell, following output will be generated −

Integral(x\*\*x, x)

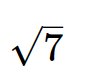
Which is equivalent to



The square root of a non-perfect square can be represented by Latex as follows using traditional symbol −

from sympy import \*   
x=7   
sqrt(x)

The output for the above code snippet is as follows −



A symbolic computation system like SymPy performs a wide array of symbolic calculations, including operations like finding derivatives, integrals, and limits, solving equations, and working with matrices. The SymPy package encompasses various modules that enable symbolic operations in areas such as plotting, rendering (e.g., in LATEX format), physics, statistics, combinatorics, number theory, geometry, logic, and more.

**SymPy — Numbers**

The core module in SymPy package contains Number class which represents atomic numbers. This class has two subclasses: Float and Rational class. Rational class is further extended by Integer class.

Float class represents a floating point number of arbitrary precision.

from sympy import Float   
Float(6.32)  
  
Output:  
6.32

SymPy can convert an integer or a string to float.

Float(10)  
  
Output:  
10.0

Float('1.33E5')# scientific notation  
  
Output:  
133000.0

While converting to float, it is also possible to specify number of digits for precision as given below −

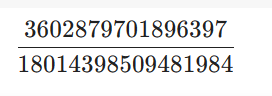
Float(1.33333,2)  
  
Output:  
1.3

A representation of a number (p/q) is represented as object of Rational class with q being a non-zero number.

Rational(3/4)  
  
Output:  
3  
-  
4  
​

If a floating point number is passed to Rational() constructor, it returns underlying value of its binary representation

Rational(0.2)



For simpler representation, specify denominator limitation.

Rational(0.2).limit\_denominator(100)  
  
Output:  
1  
-  
5

When you provide a string as an argument to the Rational() constructor, it yields a rational number with unlimited precision.

Rational("3.65")  
  
Output:  
73  
--  
20

A rational object can also be created when two numerical parameters are provided, and its attributes include the numerator and denominator components.

a=Rational(3,5)   
print (a)   
print ("numerator:{}, denominator:{}".format(a.p, a.q))  
  
Output:  
3/5  
  
numerator:3, denominator:5

SymPy introduces a RealNumber class, functioning as a synonym for Float. Additionally, SymPy establishes singleton classes for Zero and One, which can be accessed using S.Zero and S.One, respectively, as demonstrated below.

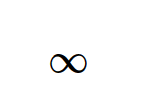
S.Zero  
  
Output:  
0

Other predefined Singleton number objects are Half, NaN, Infinity and ImaginaryUnit

from sympy import S   
print (S.Half)  
  
Output:  
1/2

Infinity is available as oo symbol object or S.Infinity

from sympy import oo   
oo



ImaginaryUnit number can be imported as I symbol or accessed as S.ImaginaryUnit and represents square root of -1

from sympy import I   
I  
  
Output:  
i

from sympy import sqrt   
i=sqrt(-1)   
i\*i  
  
Output:  
When you execute the above code snippet, you get the following output −  
  
-1

**SymPy — Symbols**

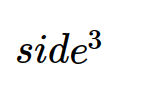
The Symbol class holds utmost significance within the symPy library. As previously indicated, symbolical calculations revolve around the use of symbols. SymPy variables are essentially instances of the Symbols class.

When you employ the Symbol() function, its argument should be a character string that represents a symbol that can be allocated to a variable.

from sympy import Symbol  
  
a = Symbol('a')  
b = Symbol('b')  
expression = a\*\*3 + b\*\*2  
expression



s=Symbol('side')   
s\*\*3



SymPy provides a convenient Symbols() function, which enables the simultaneous declaration of multiple symbolic variables. These variable names are specified within a string, separated by either commas or spaces.

from sympy import symbols   
x,y,z=symbols("x,y,z")

Within the SymPy library, the abc module encompasses all Latin and Greek alphabets, treating them as symbols. Consequently, this approach provides a convenient alternative to creating Symbol objects individually, thereby streamlining the process.

from sympy.abc import x,y,z

Nevertheless, it’s important to note that certain symbols like C, O, S, I, N, E, and Q have already been predefined. Moreover, the abc module doesn’t include definitions for symbols with multiple letters. In such cases, you should employ the Symbol object as demonstrated earlier. The abc module establishes reserved names that can identify conflicting definitions within the default SymPy namespace. Specifically, clash1 encompasses single-letter symbols, while clash2 encompasses multi-letter symbols that may potentially clash with existing definitions.

from sympy.abc import \_clash1, \_clash2   
\_clash1  
  
Output:  
{'C': C, 'O': O, 'Q': Q, 'N': N, 'I': I, 'E': E, 'S': S}

\_clash2  
  
Output:  
{'beta': beta, 'zeta': zeta, 'gamma': gamma, 'pi': pi}

Symbols that are indexed can be created using a syntax reminiscent of the range() function. Ranges are specified using a colon. The type of the range is determined by the character to the right of the colon. If this character is a digit, all the digits to the left of it are considered the starting value (which is nonnegative), and all the digits to the right are treated as one greater than the ending value.

from sympy import symbols   
symbols('a:7')  
  
Output:  
(a0, a1, a2, a3, a4, a5, a6)

symbols('mark(1:5)')  
  
Output:  
(mark1, mark2, mark3, mark4)

**SymPy — Substitution**

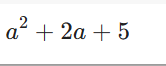
One fundamental operation when working with mathematical expressions involves substitution. In SymPy, the subs() function is employed to replace every instance of the first parameter with the second parameter.

from sympy import sin, cos  
from sympy.abc import x, a  
  
expr = sin(x) \* sin(x) + cos(x) \* cos(x)  
expr



This function proves valuable when we need to assess a specific mathematical expression. For instance, when we aim to determine the results of the given expression by replacing the variable ‘a’ with the value 5.

expr=a\*a+2\*a+5   
expr



expr.subs(a,5)  
  
Output:  
40

from sympy.abc import x   
from sympy import sin, pi   
expr=sin(x)   
expr1=expr.subs(x,pi)   
expr1  
  
Output:  
0

This function can also serve to substitute one subexpression with another. In the given example, ‘b’ is substituted with ‘a+b’.

from sympy.abc import a,b   
expr=(a+b)\*\*2   
expr1=expr.subs(b,a+b)   
expr1



**SymPy — sympify() function**

The sympify() function serves the purpose of transforming any arbitrary expression into a format that is compatible with SymPy, allowing it to be employed as a SymPy expression. It has the capability to convert regular Python objects like integers into their SymPy counterparts. Furthermore, even objects like strings that represent numbers or mathematical expressions can also undergo conversion into SymPy expressions.

from sympy import sympify  
expr="x\*\*2+3\*x+2"   
expr1=sympify(expr)   
expr1   
expr1.subs(x,2)  
  
Output:  
12

You can transform any Python object into a SymPy object, but it’s important to note that this conversion relies on the use of the eval() function internally. Therefore, caution must be exercised to ensure that the input expressions are properly sanitized. Failing to do so may result in the raising of a SympifyError.

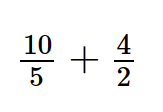
sympify("x\*\*\*2")

SympifyError occurs when attempting to perform a sympification of the expression ‘x\*\*\*2’ fails due to an exception being raised.

The sympify() function accepts the following parameters: \* strict: By default, it is set to False. When set to True, it only allows conversion for types with explicitly defined conversions. Otherwise, it raises a SympifyError when an undefined conversion is encountered. \* evaluate: When set to False, arithmetic operations and operators are transformed into their SymPy equivalents without actually evaluating the expression.

sympify("10/5+4/2")  
  
Output:  
4

sympify("10/5+4/2", evaluate=False)



**SymPy — evalf() function**

This function is designed to calculate a specified numerical expression with precision extending up to 100 decimal places. Additionally, it accepts a ‘subs’ parameter, which should be a dictionary containing numerical values assigned to symbolic variables. Take the following expression into consideration.

from sympy.abc import r   
expr=pi\*r\*\*2   
expr



To compute the expression above using the evalf() function and replacing ‘r’ with the value 5.

expr.evalf(subs={r:5})  
  
Output:  
78.5398163397448

The default floating-point precision is set to 15 decimal places, but this can be altered by specifying a precision value of up to 100. In this particular case, the given expression is calculated with a precision of 20 decimal places.

expr=a/b   
expr.evalf(20, subs={a:100, b:3})  
  
Output:  
33.333333333333333333

**SymPy — Lambdify() function**

The lambdify function serves as a means to transform SymPy expressions into Python functions. In situations where you need to compute an expression across a broad spectrum of values, using the evalf() function may not be the most efficient approach. Lambdify functions in a manner akin to lambda functions, with the added capability of mapping SymPy symbols to the corresponding symbols in a specified numerical library, typically NumPy. By default, lambdify utilizes the math standard library for its implementations.

expr=1/sin(x)   
f=lambdify(x, expr)   
f(3.14)  
  
Output:  
627.8831939138764

When dealing with expressions that involve multiple variables, you should note that when using the lambdify() function, the initial argument should be a list comprising these variables, and this should be followed by the expression you intend to compute.

expr=a\*\*2+b\*\*2   
f=lambdify([a,b],expr)   
f(2,3)  
  
Output:  
13

Nonetheless, in order to harness the numerical capabilities of the numpy library as a computational backend, we must specify it as a parameter for the lambdify() function.

f=lambdify([a,b],expr, "numpy")

We use two numpy arrays for two arguments a and b in the above function. The execution time is considerably fast in case of numpy arrays.

import numpy   
l1=numpy.arange(1,6)   
l2=numpy.arange(6,11)   
f(l1,l2)  
  
Output:  
array([ 37, 53, 73, 97, 125], dtype=int32)

**SymPy — Logical Expressions**

Boolean functions are created in the sympy.basic.booleanarg module. You can construct Boolean statements using the standard Python operators like & (And), | (Or), and ~ (Not), as well as >> and <<. These Boolean expressions are built on top of the Basic class, which is defined in SymPy’s core module.

Now, let’s discuss the **BooleanTrue function**. This function serves as the counterpart of the True value in regular Python. When you call it, it returns a unique instance that can be accessed through S.true.

For instance, consider the following code:

from sympy import \*  
x = sympify(true)  
x, S.true  
  
Output:  
(True, True)

In this code, we create a variable x and assign it the value of S.true, demonstrating the use of BooleanTrue in SymPy.

The **“BooleanFalse” function** serves as an equivalent representation of the Boolean False value in Python within the SymPy library. To access this functionality, you can utilize the symbol “S.false.”

Here’s an example of how to use it:

from sympy import \*   
x = sy.symbols('false') # Creating a symbol using S.false  
x, S.false  
  
Output:  
(False, False)

In this code, we’ve created a symbol “x” using the S.false functionality, which represents the Boolean False value.

**AND Function**

The logical AND function operates by assessing its two input values and yields a result of False should either of them be False. This function effectively mimics the behavior of the “&” operator.

Here’s an example using the SymPy library in Python:

from sympy import symbols  
from sympy.logic.boolalg import And  
  
x, y = symbols('x y')  
x = True  
y = True  
  
result = And(x, y)  
symbolic\_result = x & y  
  
# The 'result' and 'symbolic\_result' variables both hold the value 'True'.  
  
Output:  
(True, True)

In this example, the logical AND operation is carried out both with the And function and the symbolic "&" operator, yielding a value of True because both 'x' and 'y' are set to True.

y=False   
And(x,y), x"&"y  
  
Output:  
(False, False)

**Or function**

This function takes two Boolean values as input and yields a True result if at least one of them is True. The ‘|’ operator conveniently replicates its functionality.

from sympy import \*  
from sympy.logic.boolalg import Or  
x, y = symbols('x y')  
x = True  
y = False  
Or(x, y), x | y  
  
Output:  
(True, True)

x=False   
y=False   
Or(x,y), x|y  
  
Output:  
(False, False)

**Not Function**

The Logical Not function, when applied to a Boolean argument, produces the opposite of the input value. In other words, it yields True if the argument is initially False, and False if the argument is initially True. This negation operation can also be achieved using the ~ operator, which performs the same logical inversion. An illustrative example is provided below:

from sympy import \*   
from sympy.logic.boolalg import Or, And, Not   
x, y = symbols('x y')   
x = True   
y = False   
Not(x), Not(y)  
  
Output:  
(False, True)

In this code snippet, Not(x) returns False because x is initially True, and Not(y) returns True as y is initially False.

Not(And(x,y)), Not(Or(x,y))  
  
Output:  
(True, False)

**Xor Function**

The Logical XOR (exclusive OR) function yields a True result when an odd number of its input arguments are True, with the remainder being False. Conversely, it produces a False result when an even number of the input arguments are True, and the remaining ones are False. The same behavior can be observed when using the ^ operator.

from sympy import \*   
from sympy.logic.boolalg import Xor   
x,y=symbols('x y')   
x=True   
y=False  
  
Xor(x,y), x^y  
  
Output:  
(True, True)

a,b,c,d,e=symbols('a b c d e')   
a,b,c,d,e=(True, False, True, True, False)   
Xor(a,b,c,d,e)  
  
Output:  
True

In the previous example, when there are three arguments (an odd number) that are set to True, the Xor operation yields a True result. Conversely, if the count of True arguments is even, the result will be False, as demonstrated below -

a,b,c,d,e=(True, False, False, True, False)   
Xor(a,b,c,d,e)  
  
Output:  
False

**Nand Function**

This function executes the Logical NAND operation, where it assesses its input values and yields a result of True when at least one of them is False, and False only when all of them are True.

from sympy import \*   
from sympy.logic.boolalg import Nand   
a,b,c=symbols('a b c')   
a,b,c=(True, False, True)   
Nand(a,b,c), Nand(a,c)  
  
Output:  
(True, False)

**Nor Function**

This function carries out the Logical NOR operation, wherein it assesses its inputs and yields a result of False if any of them are True, and True if all of them are, in fact, False.

from sympy import \*   
from sympy.logic.boolalg import Nor   
a,b,c=symbols('a b c')   
a,b,c=(True, False, True)   
Nor(a,b,c), Nor(a,c)  
  
Output:  
(False, False)

It’s worth noting that SymPy offers convenient operators like ‘^’ for Xor, ‘~’ for Not, ‘|’ for Or, and ‘&’ for And, but in standard Python usage, these symbols serve as bitwise operators. Consequently, if you apply them to integers, the outcomes will vary.

**Equivalent function**

This function yields an equivalence relation. It will affirm “Equivalent(A, B)” as true exclusively when both A and B are either both true or both false. It provides a true outcome only when all the given arguments are logically equivalent, otherwise, it returns a false result.

from sympy import \*   
from sympy.logic.boolalg import Equivalent   
a,b,c=symbols('a b c')   
a,b,c=(True, False, True)   
Equivalent(a,b), Equivalent(a,c)  
  
Output:  
(False, True)

**ITE function**

This function functions similarly to the conditional “If-Then-Else” statement found in programming languages. When you call ITE(A, B, C), it assesses the truth value of A and then returns the result B if A is true, otherwise it returns the result C. It’s important to note that all of the arguments must be Booleans.

from sympy import \*   
from sympy.logic.boolalg import ITE   
a,b,c=symbols('a b c')   
a,b,c=(True, False, True)   
ITE(a,b,c), ITE(a,c,b)  
  
Output:  
(False, True)

**SymPy — Querying**

The SymPy package incorporates the assumptions module, which offers a set of tools for gathering information about mathematical expressions. This module introduces the ask() function, which serves the purpose of extracting such information.

sympy.assumptions.ask(property)

Following properties provide useful information about an expression −

**algebraic(x)**

For a number to be considered algebraic, it should satisfy a crucial condition: it must serve as a solution to a polynomial equation that has non-zero coefficients and only includes rational numbers. Take √2, for instance, which qualifies as algebraic because it serves as a solution to the equation x² — 2 = 0, where the coefficients are rational numbers.

**complex(x)**

Complex number predicate. It is true if and only if x belongs to the set of complex numbers.

**composite(x)**

The “ask(Q.composite(x))” function yields a true result for a given value ‘x’ only when ‘x’ is a positive integer and possesses at least one positive divisor besides 1 and the number itself, confirming that it is indeed a composite number.

**even, odd**

The ask() returns true of x is in the set of even numbers and set of odd numbers respectively.

**imaginary**

This property represents Imaginary number predicate. It is true if x can be written as a real number multiplied by the imaginary unit I.

**integer**

This property returned by Q.integer(x) returns true of x belong to set of even numbers.

**rational, irrational**

Q.irrational(x) is true if and only if x is any real number that cannot be expressed as a ratio of integers. For example, pi is an irrational number.

**positive, negative**

Predicates to check if number is positive or negative

**zero, nonzero**

Predicates to heck if a number is zero or not

from sympy import \*   
x=Symbol('x')   
x=10   
ask(Q.algebraic(pi))  
False  
  
ask(Q.complex(5-4\*I)), ask( Q.complex(100))  
(True, True)  
  
x,y=symbols("x y")   
x,y=5,10   
ask(Q.composite(x)), ask(Q.composite(y))  
(False, True)  
  
ask(Q.even(x)), ask(Q.even(y))  
(False, True)  
  
x,y= 2\*I, 4+5\*I   
ask(Q.imaginary(x)), ask(Q.imaginary(y))  
(True, False)  
  
x,y=5,10   
ask(Q.even(x)), ask(Q.even(y)), ask(Q.odd(x)), ask(Q.odd(y))  
(False, True, True, False)  
  
x,y=5,-5   
ask(Q.positive(x)), ask(Q.negative(y)), ask(Q.positive(x)), ask(Q.negative(y))  
(True, True, True, True)  
  
ask(Q.rational(pi)), ask(Q.irrational(S(2)/3))  
(False, False)  
  
ask(Q.zero(oo)), ask(Q.nonzero(I))  
(False, False)

**Basics End Here !!!**

Going forward we are going to delve into the depths of use cases and some pretty advanced functionalities. But hey, don’t forget to save this as you’ll need to revisit it couple of times as I had to.

And before we part ways do check out other articles from me and give me a follow!!! Happy Reading.

1

1