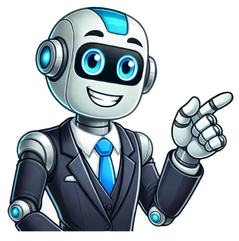
Continue



Math python pi

Return True if the values a and b are close to each other and False otherwise. Whether or not two values are considered close is determined according to given absolute and relative tolerances. If no errors occur, the result will be: abs(a-b) import os >>> dir_fd = os.open('somedir', os.O_RDONLY) >>> def opener(path, flags): ... return os.open(path, flags, dir_fd=dir_fd) ... >>> with open('spamspam.txt', 'w', opener=opener) as f: ... print('This will be written to somedir/spamspam.txt', file=f) ... >>> os.close(dir_fd) # don't leak a file descriptor The type of file object returned by the open() function depends on the mode. When open() is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a subclass of io.TextIOBase (specifically io.TextIOWrapper). When used to open a file in a binary mode with buffering, the returned class is a subclass of io.BufferedIOBase. The exact class varies: in read binary mode, it returns an io.BufferedReader; in write binary and append binary modes, it returns an io.BufferedWriter, and in read/write mode, it returns an io.BufferedRandom. When buffering is disabled, the raw stream, a subclass of io.RawIOBase, io.FileIO, is returned. See also the file handling modules, such as fileinput, io (where open() is declared), os, os.path, tempfile, and shutil. Raises an auditing event open with arguments path, mode, flags. The mode and flags arguments may have been modified or inferred from the original call. Changed in version 3.3: The opener parameter was added. The 'x' mode was added. IOError used to be raised, it is now an alias of OSError. FileExistsError is now raised if the file opened in exclusive creation mode ('x') already exists. Changed in version 3.4: The file is now non-inheritable. Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an InterruptedError exception (see PEP 475 for the rationale). The 'namereplace' error handler was added. Changed in version 3.6: Changed in version 3.11: The 'U' mode has been removed. Page 4Decorator to wrap a function with a memoizing callable that saves up to the maxsize most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments. The cache is threadsafe so that the wrapped function can be used in multiple threads. This means that the underlying data structure will remain coherent during concurrent updates. It is possible for the wrapped function to be called more than once if another thread makes an additional call before the initial call has been completed and cached. Since a dictionary is used to cache results, the positional and keyword arguments to the function must be hashable. Distinct argument patterns may be considered to be distinct calls with separate cache entries. For example, f(a=1, b=2) and f(b=2, a=1) differ in their keyword argument order and may have two separate cache entries. If user_function is specified, it must be a callable. This allows the lru_cache decorator to be applied directly to a user function, leaving the maxsize at its default value of 128: @lru_cache def count_vowels(sentence): return sum(sentence.count(vowel) for vowel in 'AEIOUaeiou') If maxsize is set to None, the LRU feature is disabled and the cache can grow without bound. If typed is set to true, function arguments of different types will be cached separately. If typed is false, the implementation will usually regard them as equivalent calls and only cache a single result. (Some types such as str and int may be cached separately even when typed is false.) Note, type specificity applies only to the function’s immediate arguments rather than their contents. The scalar arguments, Decimal(42) and Fraction(42) are be treated as distinct calls with distinct results. In contrast, the tuple arguments ('answer', Decimal(42)) and ('answer', Fraction(42)) are treated as equivalent. The wrapped function is instrumented with a cache_parameters() function that returns a new dict showing the values for maxsize and typed. This is for information purposes only. Mutating the values has no effect. To help measure the effectiveness of the cache and tune the maxsize parameter, the wrapped function is instrumented with a cache_info() function that returns a named tuple showing hits, misses, maxsize and currsize. The decorator also provides a cache_clear() function for clearing or invalidating the cache. The original underlying function is accessible through the __wrapped__ attribute. This is useful for introspection, for bypassing the cache, or for rewrapping the function with a different cache. The cache keeps references to the arguments and return values until they age out of the cache or until the cache is cleared. If a method is cached, the self instance argument is included in the cache. See How do I cache method calls? An LRU (least recently used) cache works best when the most recent calls are the best predictors of upcoming calls (for example, the most popular articles on a news server tend to change each day). The cache’s size limit assures that the cache does not grow without bound on long-running processes such as web servers. In general, the LRU cache should only be used when you want to reuse previously computed values. Accordingly, it doesn’t make sense to cache functions with side-effects, functions that need to create distinct mutable objects on each call (such as generators and async functions), or impure functions such as time() or random(). Example of an LRU cache for static web content: @lru_cache(maxsize=32) def get_pep(num): 'Retrieve text of a Python Enhancement Proposal' resource = f' num:04d}' try: with urllib.request.urlopen(resource) as s: return s.read() except urllib.error.HTTPError: return 'Not Found' >>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991: ... pep = get_pep(n) ... print(n, len(pep)) >>> get_pep.cache_info() CacheInfo(hits=3, misses=8, maxsize=32, currsize=8) Example of efficiently computing Fibonacci numbers using a cache to implement a dynamic programming technique: @lru_cache(maxsize=None) def fib(n): if n < 2: return n return fib(n-1) + fib(n-2) >>> [fib(n) for n in range(16)] [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610] >>> fib.cache_info() CacheInfo(hits=28, misses=16, maxsize=None, currsize=16) Changed in version 3.3: Added the typed option. Changed in version 3.8: Added the user_function option. Changed in version 3.9: Added the function cache_parameters() In the world of programming, mathematical constants like pi play a vital role in various applications, from scientific calculations to graphical representations. Python, known for its simplicity and versatility, provides several ways to use the value of pi in your projects. Whether you’re a beginner looking to understand the basics or an experienced developer needing a refresher, this tutorial will guide you through using pi effectively in Python. We’ll explore the different methods available, including importing from libraries and using built-in functions. By the end of this article, you’ll have a solid understanding of how to implement pi in your Python code seamlessly. Using the math Module to Access Pi One of the most straightforward ways to access the value of pi in Python is by using the built-in math module. This module includes a constant math.pi, which provides the value of pi to a high degree of accuracy. To use this method, you first need to import the math module. Here’s how you can do it. import math radius = 5 area = math.pi * (radius ** 2) print(f"The area of a circle with radius {radius} is {area}.") Output: The area of a circle with radius 5 is 78.53981633974483. In this example, we first import the math module, which gives us access to the constant math.pi. We then define a radius for our circle and calculate the area using the formula A = πr². The result is printed out, showing the area of the circle. This method is highly efficient and widely used in various mathematical computations, making it a go-to option for many Python developers. Using NumPy for Advanced Mathematical Operations If you’re working on more complex mathematical tasks, the NumPy library is another excellent choice. This library not only provides the value of pi but also offers a wide range of mathematical functions for array and matrix operations. To use pi from NumPy, you need to install the library if you haven’t already. Here’s how you can utilize pi with NumPy. import numpy as np angle_degrees = 90 angle_radians = np.deg2rad(angle_degrees) sine_value = np.sin(angle_radians) print(f"The sine of {angle_degrees} degrees is {sine_value}.") Output: The sine of 90 degrees is 1.0. In this example, we import the NumPy library and convert an angle from degrees to radians using np.deg2rad(). We then calculate the sine of that angle with np.sin(). The result indicates that the sine of 90 degrees is 1.0, which aligns with trigonometric principles. Using NumPy is particularly beneficial when dealing with arrays or performing more advanced mathematical operations, making it a powerful tool in your Python arsenal. Defining Your Own Pi Constant For those who prefer a more hands-on approach, you can also define your own constant for pi. While this is not necessary for most applications, it can be useful for educational purposes or when you want to avoid importing external libraries. Here’s how you can define and use your own pi constant. PI = 3.141592653589793 diameter = 10 circumference = PI * diameter print(f"The circumference of a circle with diameter {diameter} is {circumference}.") Output: The circumference of a circle with diameter 10 is 31.41592653589793. In this code snippet, we define our own constant PI with a value of 3.141592653589793. We then calculate the circumference of a circle using the formula C = πd, where d is the diameter. The output shows the circumference of the circle based on our defined constant. While this method works well, it’s important to note that using the math module or NumPy is generally more reliable for precision in mathematical calculations. Using SymPy for Symbolic Mathematics If you’re delving into symbolic mathematics, the SymPy library is an excellent option. It allows for symbolic computation, which means you can manipulate mathematical expressions symbolically rather than numerically. Here’s how to use pi with SymPy. from sympy import pi, symbols x = symbols('x') expression = pi * x**2 print(f"The symbolic expression for the area of a circle is: {expression}.") Output: The symbolic expression for the area of a circle is: pi*x**2. In this example, we import pi from SymPy and create a symbolic variable x. We then define an expression for the area of a circle symbolically as πx². The output displays the symbolic expression rather than a numerical value. This method is particularly useful in fields such as algebra and calculus, where you may need to manipulate formulas or solve equations symbolically. Conclusion Using pi in Python is straightforward and versatile, thanks to the various libraries and methods available. Whether you choose to use the built-in math module, leverage the power of NumPy, define your own constant, or explore symbolic mathematics with SymPy, understanding how to work with pi can enhance your programming skills and mathematical computations. By incorporating these methods into your projects, you can ensure accurate and efficient calculations, making your Python applications more robust and effective. FAQ What is the value of pi in Python? The value of pi in Python can be accessed using the math module as math.pi, which is approximately 3.14159. Can I use pi in NumPy? Yes, you can use pi in NumPy by accessing it via np.pi, which provides the same value as the math module. Is it necessary to import a library to use pi in Python? While it is common to import libraries like math or NumPy to access pi, you can also define your own constant if needed. What is the difference between math.pi and numpy.pi? Both provide the same value of pi, but numpy.pi is often used in the context of array operations and mathematical functions provided by NumPy. Can I perform symbolic mathematics with pi in Python? Yes, you can use the SymPy library to perform symbolic mathematics with pi, allowing for manipulation of mathematical expressions. So you want to use pi or π in your python code, then you are in for a treat. We will be covering Python’s math.pi method of math module. We will also be looking at other ways to use pi(π), for instance, numpy.pi and scipy.pi methods of numpy and scipy libraries also do the same job, so we will examine when to use which respective methods. Before jumping on the topic, let’s take a brief refresher on pi(π): Pi or π(greek letter) is the ratio of a circle’s circumference to its diameter. It is an irrational number. In other words, the number of digits beyond the decimal point is infinite. 3.1415929 or 22/7 are two ways to write it. 14th March is observed as pi(π) day as 3, 1 & 4 are the significant digits of pi(π). The math module provides the pi(π) constant value: numerous other constants and mathematical functions like power and logarithmic, trigonometric, special constants, etc. # to use it like math.pi import math # to use directly as pi from math import pi Importing pi from the math module import math radius = int(input("Enter radius:")) def cal_circum(radius:int): return f"Circumference:{2*math.pi*radius} unit" print(cal_circum(radius)) The formula gives the circumference of a circle- 2πR, where 2 and π are constants and R is the circle’s radius. Finding circumference using math.pi import math radius = int(input("Enter radius:")) def cal_area(radius:int): return f"Area:{math.pi*(radius**2)} unit sq." print(cal_area(radius)) The formula gives the area of a circle – π*R2, where π is a constant and R is the circle’s radius. Finding areas using math.pi math.pi by default provides 15 places of precision for the value of pi. However, if you want to get a more precise pi(π) value, you can do so by using a decimal of Python’s standard library. from decimal import * import math print(len(str(math.pi))) getcontext().prec = 31 pi_precision = Decimal(22)/Decimal(7) print(len(str(pi_precision))) Precision comparison By excluding ‘3’ at the one’s place and the decimal, we get 15 and 30 digits of precision for math.pi and decimal, respectively. The most accurate value of pi is 62,831,853,071,796 digits and was achieved by the University of Applied Sciences (Switzerland) in Chur, Switzerland, on 19 August 2021. You can use the value of pi(π) without using the math module. Let’s see how can we achieve this: According to Python’s conventions, we can define constants using capital variable names. The only drawback is that anyone can easily modify the value. Output 1 Another way is to use the pconst library, which helps create constant values that can’t be modified, just like other languages like c, c++, java, etc. Firstly you need to install the pconst library. Code below, we try to change the constant value to 5, but the pconst library throws an error “Constant value of ‘PI’ is not editable”, thus preventing any changes. Pconst can be used to define our own constants, which we don’t want to be modified. pip install pconst from pconst import const const.PI = 22/7 try: const.PI = 5 except Exception as e: print(const.PI) print(e) pconst library doesn’t allow modifications If you want or work with arrays and matrices, the NumPy library is a clear choice. It has functions for working in the domain of linear algebra, Fourier transform, etc. It provides the constant pi(π). However, to use it, you must install it, depending on the case if you have Anaconda installed or not. # if you have anaconda installed in your system conda install numpy # if you have only python installed pip install numpy pi(π) constant value using NumPy library SciPy is a scientific computation library that uses NumPy underneath. SciPy stands for Scientific Python. It provides more utility functions for optimization, stats, and signal processing. # if you have anaconda installed in your system conda install scipy # if you have only python installed pip install scipy pi(π) constant value using Scipy library Q1. Should I use math.pi, numpy.pi or scipy.py? The math.pi is part of the built-in math library of Python, while the rest of the two have to be installed to make use of them. All three of them provide the same value. If you are looking to work on large data sets with numerous calculations, numpy or scipy are great options; otherwise, math.pi is the way to go. To use pi in python, use: math.pi in your statement. It will give the value of pi as float – 3.141592653589793. If you’re getting the error message “math pi is not defined” in Python, it might be because you’re trying to use the constant pi without importing it from the math module. Here’s an example of how to use pi correctly in Python: import math # Calculate the circumference of a circle of #radius 5 radius = 5 peri = 2 * math.pi * radius print (peri) In this example, we first import the math module using the import command. We then use the math.pi constant to calculate the circumference of a circle of radius 5. Prefixing pi with math tells Python to use the pi constant from the math module. Without this import statement, we would get the error message “NameError: name ‘pi’ is not defined”. The value of pi with both numpy and math modules is above fifteen digits accuracy. It’s a float value and accessing the pi value from both modules fetches the same result. The only difference is that math module doesn’t need any type of dependency but with numpy we need many additions. In case, you are using numpy functionalities beforehand, go for np.pi. How do you define pi in Numpy? Just import the numpy library. It is a predefined function. pip install numpyimport numpy as npans= np.piprint(ans)