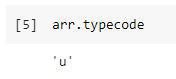
**Q1. What are the benefits of the built-in array package, if any?**

**Functions of Array**

* Get the Type Code. Since the type code is very important to an array, which determined what kind of elements that the array can have. ...
* Get Size of Array Item. ...
* Count the Number of Occurrences. ...
* Append and Extend. ...
* Manipulating the Index. ...
* Array to List.

## 1. Get the Type Code

Since the type code is very important to an array, which determined what kind of elements that the array can have. We can get the type code by accessing its property typecode.

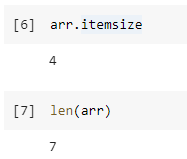


**2. Get Size of Array Item**

As above-mentioned, unlike Python List, the Array is much more rigid. That is, the items that an array contains must be of the same type. Therefore, the size of the item will also be the same. We can check the size of each item.

arr.itemsize

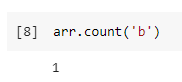
Don’t be confused that this is not the length of the array, but the size of a single item in the array.



**3. Count the Number of Occurrences**

Like Python List, the Array also has the count() function that will count the number of occurrences of a certain item in the array. Maybe you don’t even know that Python List has this? Try it out :)

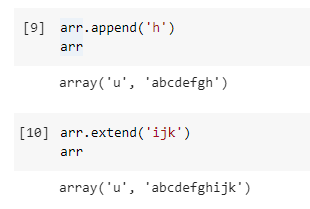
arr.count('b')



**4. Append and Extend**

If we want to add new items into an array, we can either use the append() or the extend() function. The former will add a single element, while the latter can add multiple items at one time.

arr.append('h')  
arr.extend('ijk')



**Q2. What are some of the array package's limitations?**

An array which is formed will be homogeneous. That is, in an integer array only integer values can be stored, while in a float array only floating value and character array can have only characters. Thus, no array can have values of two data types.

# Python program to demonstrate

# Creation of Array

# importing "array" for array creations

import array as arr

# creating an array with integer type

a = arr.array('i', [1, 2, 3])

# printing original array

print ("The new created array is : ", end =" ")

for i in range (0, 3):

    print (a[i], end =" ")

print()

# creating an array with double type

b = arr.array('d', [2.5, 3.2, 3.3])

# printing original array

print ("The new created array is : ", end =" ")

for i in range (0, 3):

    print (b[i], end =" ")

### **Complexities for Creation of Arrays:**

**Time Complexity:**O(1)

**Auxiliary Space:**O(n)

**Q3. Describe the main differences between the array and numpy packages.**

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original. The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

c **=** **[]**

**for** i **in** range**(**len**(**a**)):**

c**.**append**(**a**[**i**]\***b**[**i**])**

This produces the correct answer, but if a and b each contain millions of numbers, we will pay the price for the inefficiencies of looping in Python. We could accomplish the same task much more quickly in C by writing (for clarity we neglect variable declarations and initializations, memory allocation, etc.)

**for** **(**i **=** **0;** i **<** rows**;** i**++)** **{**

c**[**i**]** **=** a**[**i**]\***b**[**i**];**

**}**

This saves all the overhead involved in interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of our data. In the case of a 2-D array, for example, the C code (abridged as before) expands to

**for** **(**i **=** **0;** i **<** rows**;** i**++)** **{**

**for** **(**j **=** **0;** j **<** columns**;** j**++)** **{**

c**[**i**][**j**]** **=** a**[**i**][**j**]\***b**[**i**][**j**];**

**}**

**}**

NumPy gives us the best of both worlds: element-by-element operations are the “default mode” when an ndarray is involved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy

c **=** a **\*** b

does what the earlier examples do, at near-C speeds, but with the code simplicity we expect from something based on Python. Indeed, the NumPy idiom is even simpler! This last example illustrates two of NumPy’s features which are the basis of much of its power: vectorization and broadcasting.

**Q4. Explain the distinctions between the empty, ones, and zeros functions**.

empty, unlike zeros, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

empty, unlike zeros, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

**np.zeros**

Return a new array setting values to zero.

>>> np.zeros((2, 2))

array([[0., 0.],

[0., 0.]])

**np.empty**

Return a new uninitialized array.

>>> np.empty((2, 2))

array([[1.35807735e-312, 1.35807731e-312],

[1.99637364e-310, 8.69169476e-311]])

**Q5. In the fromfunction function, which is used to construct new arrays, what is the role of the callable argument?**

function : [callable] The function is called with N parameters, where N is the rank of shape. Each parameter represents the coordinates of the array varying along a specific axis. shape : [(N, ) tuple of ints] Shape of the output array, which also determines the shape of the coordinate arrays passed to function.

**numpy.fromfunction()** function construct an array by executing a function over each coordinate and the resulting array, therefore, has a value fn(x, y, z) at coordinate (x, y, z).

*Syntax : numpy.fromfunction(function, shape,  dtype)*

*Parameters :*

*function : [callable] The function is called with N parameters, where N is the rank of shape. Each parameter represents the coordinates of the array varying along a specific axis.*

# Python program explaining

# numpy.fromfunction() function

# importing numpy as geek

import numpy as geek

gfg = geek.fromfunction(lambda i, j: i \* j, (4, 4), dtype = float)

print(gfg)

Python program explaining

# numpy.fromfunction() function

# importing numpy as geek

import numpy as geek

gfg = geek.fromfunction(lambda i, j: i == j, (3, 3), dtype = int)

print(gfg)

**Q6. What happens when a numpy array is combined with a single-value operand (a scalar, such as an int or a floating-point value) through addition, as in the expression A + n?**

NumPy, short for Numerical Python, is the fundamental package required for high performance scientific computing and data analysis. It is the foundation on which nearly all of the higher-level tools in this book are built. Here are some of the things it provides:

* ndarray, a fast and space-efficient multidimensional array providing vectorized arithmetic operations and sophisticated broadcasting capabilities
* Standard mathematical functions for fast operations on entire arrays of data without having to write loops
* Tools for reading / writing array data to disk and working with memory-mapped files
* Linear algebra, random number generation, and Fourier transform capabilities
* Tools for integrating code written in C, C++, and Fortran

The last bullet point is also one of the most important ones from an ecosystem point of view. Because NumPy provides an easy-to-use C API, it is very easy to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays. This feature has made Python a language of choice for wrapping legacy C/C++/Fortran codebases and giving them a dynamic and easy-to-use interface.

While NumPy by itself does not provide very much high-level data analytical functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools like pandas much more effectively. If you’re new to Python and just looking to get your hands dirty working with data using pandas, feel free to give this chapter a skim. For more on advanced NumPy features like broadcasting, see [Chapter 12](https://www.oreilly.com/library/view/python-for-data/9781449323592/ch12.html).

For most data analysis applications, the main areas of functionality I’ll focus on are:

* Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
* Common array algorithms like sorting, unique, and set operations
* Efficient descriptive statistics and aggregating/summarizing data
* Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
* Expressing conditional logic as array expressions instead of loops with if-elif-else branches

While NumPy provides the computational foundation for these operations, you will likely want to use pandas as your basis for most kinds of data analysis (especially for structured or tabular data) as it provides a rich, high-level interface making most common data tasks very concise and simple. pandas also provides some more domain-specific functionality like time series manipulation, which is not present in NumPy.

In [13]: data1 = [6, 7.5, 8, 0, 1]

In [14]: arr1 = np.array(data1)

In [15]: arr1

Out[15]: array([ 6. , 7.5, 8. , 0. , 1. ])

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [17]: arr2 = np.array(data2)

In [18]: arr2

Out[18]:

array([[1, 2, 3, 4],

[5, 6, 7, 8]])

In [19]: arr2.ndim

Out[19]: 2

In [20]: arr2.shape

Out[20]: (2, 4)

Unless explicitly specified (more on this later), np.array tries to infer a good data type for the array that it creates. The data type is stored in a special dtype object; for example, in the above two examples we have:

In [21]: arr1.dtype

Out[21]: dtype('float64')

In [22]: arr2.dtype

Out[22]: dtype('int64')

In addition to np.array, there are a number of other functions for creating new arrays. As examples, zeros and ones create arrays of 0’s or 1’s, respectively, with a given length or shape. empty creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

In [23]: np.zeros(10)

Out[23]: array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [24]: np.zeros((3, 6))

Out[24]:

array([[ 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0.]])

In [25]: np.empty((2, 3, 2))

Out[25]:

array([[[ 4.94065646e-324, 4.94065646e-324],

[ 3.87491056e-297, 2.46845796e-130],

[ 4.94065646e-324, 4.94065646e-324]],

[[ 1.90723115e+083, 5.73293533e-053],

[ -2.33568637e+124, -6.70608105e-012],

[ 4.42786966e+160, 1.27100354e+025]]])

**Q7. Can array-to-scalar operations use combined operation-assign operators (such as += or \*=)? What is the outcome?**

With scalars:

>>>

>>> a = np.array([1, 2, 3, 4])

>>> a + 1

array([2, 3, 4, 5])

>>> 2\*\*a

array([ 2, 4, 8, 16])

All arithmetic operates elementwise:

>>>

>>> b = np.ones(4) + 1

>>> a - b

array([-1., 0., 1., 2.])

>>> a \* b

array([2., 4., 6., 8.])

>>> j = np.arange(5)

>>> 2\*\*(j + 1) - j

array([ 2, 3, 6, 13, 28])

These operations are of course much faster than if you did them in pure python:

>>>

>>> a = np.arange(10000)

>>> %timeit a + 1

10000 loops, best of 3: 24.3 us per loop

>>> l = range(10000)

>>> %timeit [i+1 for i in l]

1000 loops, best of 3: 861 us per loop

Comparisons:

>>>

>>> a = np.array([1, 2, 3, 4])

>>> b = np.array([4, 2, 2, 4])

>>> a == b

array([False, True, False, True])

>>> a > b

array([False, False, True, False])

Array-wise comparisons:

>>>

>>> a = np.array([1, 2, 3, 4])

>>> b = np.array([4, 2, 2, 4])

>>> c = np.array([1, 2, 3, 4])

>>> np.array\_equal(a, b)

False

>>> np.array\_equal(a, c)

True

Logical operations:

>>>

>>> a = np.array([1, 1, 0, 0], dtype=bool)

>>> b = np.array([1, 0, 1, 0], dtype=bool)

>>> np.logical\_or(a, b)

array([ True, True, True, False])

>>> np.logical\_and(a, b)

array([ True, False, False, False])

Transcendental functions:

>>>

>>> a = np.arange(5)

>>> np.sin(a)

array([ 0. , 0.84147098, 0.90929743, 0.14112001, -0.7568025 ])

>>> np.log(a)

array([ -inf, 0. , 0.69314718, 1.09861229, 1.38629436])

>>> np.exp(a)

array([ 1. , 2.71828183, 7.3890561 , 20.08553692, 54.59815003])

Shape mismatches

>>>

>>> a = np.arange(4)

>>> a + np.array([1, 2])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: operands could not be broadcast together with shapes (4) (2)

Broadcasting? We’ll return to that [later](https://scipy-lectures.org/intro/numpy/operations.html#broadcasting).

Transposition:

>>>

>>> a = np.triu(np.ones((3, 3)), 1) # see help(np.triu)

>>> a

array([[0., 1., 1.],

[0., 0., 1.],

[0., 0., 0.]])

>>> a.T

array([[0., 0., 0.],

[1., 0., 0.],

[1., 1., 0.]])

Note

The transposition is a view

The transpose returns a view of the original array:

>>>

>>> a = np.arange(9).reshape(3, 3)

>>> a.T[0, 2] = 999

>>> a.T

array([[ 0, 3, 999],

[ 1, 4, 7],

[ 2, 5, 8]])

>>> a

array([[ 0, 1, 2],

[ 3, 4, 5],

[999, 7, 8]])

**Q8. Does a numpy array contain fixed-length strings? What happens if you allocate a longer string to one of these arrays?**

NumPy builds on (and is a successor to) the successful Numeric array object. Its goal is to create the corner-stone for a useful environment for scientific computing. NumPy provides two fundamental objects: an N-dimensional array object (ndarray) and a universal function object (ufunc).

The dtype of any numpy array containing string values is the maximum length of any string present in the array. Once set, it will only be able to store new string having length not more than the maximum length at the time of the creation. If we try to reassign some another string value having length greater than the maximum length of the existing elements, it simply discards all the values beyond the maximum length.

In this post we are going to discuss ways in which we can overcome this problem and create a numpy array of **arbitrary length**.

Let’s first visualize the problem with creating an arbitrary length numpy array of string type.

# importing numpy as np

import numpy as np

# Create the numpy array

country = np.array(['USA', 'Japan', 'UK', '', 'India', 'China'])

# Print the array

print(country)

**Problem #1 :** Create a numpy array of arbitrary length.

**Solution :** While creating the array assign the ‘object’ dtype to it. This lets you have all the behaviors of the python string.

|  |
| --- |
| # importing the numpy library as np  import numpy as np    # Create a numpy array  # set the dtype to object  country = np.array(['USA', 'Japan', 'UK', '', 'India', 'China'], dtype = 'object')    # Print the array  print(country) |

**Q9. What happens when you combine two numpy arrays using an operation like addition (+) or multiplication (\*)? What are the conditions for combining two numpy arrays?**

We have seen lots of operators in our Python tutorial. Of course, we have also seen many cases of operator overloading, e.g. "+" for the addition of numerical values and the concatenation of strings.

42 + 5

"Python is one of the best " + "or maybe the best programming language!"

We will learn in this introduction that the operator signs are overloaded in Numpy as well, so that they can be used in a "natural" way.

We can, for example, add a scalar to an ndarrays, i.e. the scalar will be added to every component. The same is possible for subtraction, division, multiplication and even for applying functions, like sine, cosine and so on, to an array.

It is also extremely easy to use all these operators on two arrays as well.

## Using Scalars

Let's start with adding scalars to arrays:

import numpy as np

lst = [2,3, 7.9, 3.3, 6.9, 0.11, 10.3, 12.9]

v = np.array(lst)

v = v + 2

print(v)

### OUTPUT:

[ 4. 5. 9.9 5.3 8.9 2.11 12.3 14.9 ]

Multiplication, Subtraction, Division and exponentiation are as easy as the previous addition:

print(v \* 2.2)

### OUTPUT:

[ 8.8 11. 21.78 11.66 19.58 4.642 27.06 32.78 ]

print(v - 1.38)

### OUTPUT:

[ 2.62 3.62 8.52 3.92 7.52 0.73 10.92 13.52]

print(v \*\* 2)

print(v \*\* 1.5)

### OUTPUT:

[ 16. 25. 98.01 28.09 79.21 4.4521 151.29 222.01 ]

[ 8. 11.18033989 31.14962279 12.2015163 26.55125232 3.06495204

43.13776768 57.51477202]

We started this example with a list lst, which we turned into the array v. Do you know how to perform the above operations on a list, i.e. multiply, add, subtract and exponentiate every element of the list with a scalar?

We could use a for loop for this purpose. Let us do it for the addition without loss of generality. We will add the value 2 to every element of the list:

lst = [2,3, 7.9, 3.3, 6.9, 0.11, 10.3, 12.9]

res = []

for val in lst:

res.append(val + 2)

print(res)

### OUTPUT:

[4, 5, 9.9, 5.3, 8.9, 2.11, 12.3, 14.9]

Even though this solution works it is not the Pythonic way to do it. We will rather use a list comprehension for this purpose than the clumsy solution above. If you are not familar with this approach, you may consult our [chapter on list comprehension](https://python-course.eu/advanced-python/list-comprehension.php) in our Python course.

res = [ val + 2 for val in lst]

print(res)

### OUTPUT:

[4, 5, 9.9, 5.3, 8.9, 2.11, 12.3, 14.9]

Even though we had already measured the time consumed by Numpy compared to "plane" Python, we will compare these two approaches as well:

v = np.random.randint(0, 100, 1000)

%timeit v + 1

### OUTPUT:

869 ns ± 32.1 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

lst = list(v)

%timeit [ val + 2 for val in lst]

Arithmetic Operations with two Arrays

If we use another array instead of a scalar, the elements of both arrays will be component-wise combined:

import numpy as np

A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])

B = np.ones((3,3))

print("Adding to arrays: ")

print(A + B)

print("\nMultiplying two arrays: ")

print(A \* (B + 1))

### OUTPUT:

Adding to arrays:

[[12. 13. 14.]

[22. 23. 24.]

[32. 33. 34.]]

Multiplying two arrays:

[[22. 24. 26.]

[42. 44. 46.]

[62. 64. 66.]]

"A \* B" in the previous example shouldn't be mistaken for matrix multiplication. The elements are solely component-wise multiplied.

## Matrix Multiplication:

For this purpose, we can use the dot product. Using the previous arrays, we can calculate the matrix multiplication:

np.dot(A, B)

**Q10. What is the best way to use a Boolean array to mask another array?**

Masks are an array that contains the list of boolean values for the given condition.  
...  
**Steps Required**

1. Import the library.
2. Create a function for masking.
3. Masking can be done by following two approaches:- ...
4. Then return the masked from the function.
5. Now create the main function.
6. Create two arrays one for masking another.

*numpy.ma.getmask(arr)*

*numpy.ma.masked\_array(arr, mask=)*

*where,*

***condition:****condition for masking*

***arr:****arr to be masked*

***mask:****result of masked array*

### Steps Required

* Import the library.
* Create a function for masking.
* Masking can be done by following two approaches:-
  + **Using masked\_where() function:** Pass the two array in the function as a parameter then use *numpy.ma.masked\_where()* function in which pass the condition for masking and array to be masked. In this we are giving the condition for masking by using one array and masking the another array for that condition.
  + **Using masked\_where(), getmask() and masked\_array() function:**Pass the two array in the function as a parameter then use *numpy.ma.masked\_where()* function in which pass the condition for masking and array to be masked in this we are using the same array for which we are giving condition for making and the array to be masked and store the result in the variable, then use *numpy.ma.getmask()* function in which pass the result of marked\_where function and store it in the variable named as ‘res\_mask’.
* Then return the masked from the function.
* Now create the main function
* Create two arrays one for masking another.
* Then call the function as we have created above and pass both the arrays in the function as a parameter and store the result in a variable let named ‘masked’.
* Now for getting the array as a 1-d array we are using *numpy.ma.compressed()* which passes the masked as a parameter.
* Then print the Masked array.

importing the library

import numpy as np

# function to create masked array

def masking(ar1, ar2):

  # masking the array1 by using array2

  # where array2 mod 7 is true

  mask = np.ma.masked\_where(ar2%7,ar1)

  return mask

# main function

if \_\_name\_\_ == '\_\_main\_\_':

  # creating two arrays

  x = np.array([1,2,4,5,7,8,9])

  y = np.array([10,12,14,5,7,0,13])

  # calling masking function to get

  # masked array

  masked = masking(x,y)

  # getting the values as 1-d array which

  # are non masked

  masked\_array = np.ma.compressed(mask)

  # printing the resultant array after masking

  print(f'Masked Array is:{masked\_array}')

**Q11. What are three different ways to get the standard deviation of a wide collection of data using both standard Python and its packages? Sort the three of them by how quickly they execute.**

In general, a low standard deviation means that the data is very closely related to the average, thus very reliable and a high standard deviation means that there is a large variance between the data and the statistical average, thus not as reliable.

One of the most important applications of standard deviation is in comparing two datasets. If two datasets have the same average, it does not mean necessarily they are exactly the same, right? For example, the datasets 199, 200, 201 and 0, 200, 400 both have the same average (200), however the first dataset has a very small standard deviation (s=1) compared to the second dataset (s=200).

# Standard Deviation for a sample or a population

A **population**dataset contains all members of a specified group (the entire list of possible data values). For example, the population may be “ALL people living in Canada”. A **sample**dataset contains a part, or a subset, of a population. The size of a sample is always less than the size of the population from which it is taken. For example, the sample may be “SOME people living in Canada”.

We are normally interested in knowing the **population** **standard deviation** as it contains all the values we want to analyze. We would normally calculate the population standard deviation if we have the entire population or we have a sample of a larger population, but we do not want to generalize our findings to the population. In many cases, we have a sample of data from which we would like to generalize the analysis to a population though. In general, if you have only a sample of data and you want to make a statement about the population standard deviation from which the sample is drawn, you need to use the sample standard deviation. In this article, I will focus on the population standard deviation. The **population standard deviation** formula is the following:

# 

# **Standard Deviation in Python**

The population mean and standard deviation of a dataset can be calculated using Numpy library in Python. The following code shows the work:

import numpy as np  
dataset=[13, 22, 26, 38, 36, 42,49, 50, 77, 81, 98, 110]print('Mean:', np.mean(dataset))  
print('Standard Deviation:', np.std(dataset))Mean:53.5  
Standard Deviation: 29.694275542602483

**12. What is the dimensionality of a Boolean mask-generated array?**

one-dimensional

Boolean Arrays as Masks

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is True .

import numpy as np

import pandas as pd

# use pandas to extract rainfall inches as a NumPy array

rainfall = pd.read\_csv('data/Seattle2014.csv')['PRCP'].values

inches = rainfall / 254.0 # 1/10mm -> inches

inches.shape

Out[1]:

(365,)

The array contains 365 values, giving daily rainfall in inches from January 1 to December 31, 2014.

As a first quick visualization, let's look at the histogram of rainy days, which was generated using Matplotlib (we will explore this tool more fully in [Chapter 4](https://jakevdp.github.io/PythonDataScienceHandbook/04.00-introduction-to-matplotlib.html)):

In [2]:

%matplotlib inline

import matplotlib.pyplot as plt

import seaborn; seaborn.set() # set plot styles

In [3]:

plt.hist(inches, 40);

### **Boolean operators**

We've already seen how we might count, say, all days with rain less than four inches, or all days with rain greater than two inches. But what if we want to know about all days with rain less than four inches and greater than one inch? This is accomplished through Python's bitwise logic operators, &, |, ^, and ~. Like with the standard arithmetic operators, NumPy overloads these as ufuncs which work element-wise on (usually Boolean) arrays.

For example, we can address this sort of compound question as follows:

In [23]:

np.sum((inches > 0.5) & (inches < 1))

| **Operator** | **Equivalent ufunc** |  | **Operator** | **Equivalent ufunc** |
| --- | --- | --- | --- | --- |
| & | np.bitwise\_and |  | | | np.bitwise\_or |
| ^ | np.bitwise\_xor |  | ~ | np.bitwise\_not |

Using these tools, we might start to answer the types of questions we have about our weather data. Here are some examples of results we can compute when combining masking with aggregations:

In [25]:

print("Number days without rain: ", np.sum(inches == 0))

print("Number days with rain: ", np.sum(inches != 0))

print("Days with more than 0.5 inches:", np.sum(inches > 0.5))

print("Rainy days with < 0.2 inches :", np.sum((inches > 0) &

(inches < 0.2)))

Number days without rain: 215

Number days with rain: 150

Days with more than 0.5 inches: 37

Rainy days with < 0.2 inches : 75