Conditions

Numpy arrays can be created according to a (boolean) condition.

For instance, let's define a set of temperature values (°C).

In [1]:

```
import numpy as np
T = np.array([25, 27, 29, 24, 26, 18, 32])
```

Let's extract temperatures above 25°C:

In [2]:

```
print(T > 25)
print(T[T > 25])
```

```
[False  True  True False  True False  True]
[27 29 26 32]
```

One can replace these values using `np.where`:

- whenever the condition holds True, replace the value with 30
- whenever the condition holds False, keep the value

```
In [3]: np.where(T>25, 30, T)
```

```
Out[3]: array([25, 30, 30, 24, 30, 18, 30])
```

Advanced: note that `np.where` returns a copy of the object

Let's suppose now there are several temperature series:

In [4]:

```
T = np.stack([T, T+1, T-3])
T
```

Out[4]:

```
array([[25, 27, 29, 24, 26, 18, 32],
       [26, 28, 30, 25, 27, 19, 33],
       [22, 24, 26, 21, 23, 15, 29]])
```

Let's replace the values of each serie for which the maximum is not 33°C. Hence, maximum of `T` along axis 1 is calculated (since series are stacked along axis 0).

In [5]:

```
max_ = T.max(axis=1)
print(max_)
```

```
[32 33 29]
```

Then the condition is defined:

In [6]:

```
cond = max_ == 33
print(cond)
```

```
[False  True False]
```

And it is used in `np.where`.

In [7]:

```
np.where(cond, T, 0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 np.where(cond, T, 0)

ValueError: operands could not be broadcast together with shapes (3,) (3,7)
()
```

Error! The condition `cond` does not have the same shape than replacing values (`T` and `0`) Hence `numpy` cannot handle the condition.

This is because the maximum calculation removed one dimension. But this can be prevented using `keepdims=True`:

```
In [8]:
max_ = T.max(axis=1, keepdims=True)
print(max_)                                 # shape of dimension 1 is 1 instead of 7 (becau
                                            # but at least this dimension is kept
```

```
[[32]
 [33]
 [29]]
```

```
In [9]:
cond = max_ == 33
np.where(cond, T, 0)
```

```
Out[9]:
array([[ 0,  0,  0,  0,  0,  0,  0],
       [26, 28, 30, 25, 27, 19, 33],
       [ 0,  0,  0,  0,  0,  0,  0]])
```

Similarly: using `np.any`, one can look for series for which **at least one** value meets a criterium:

In [10]:

```
cond = (T >= 32).any(axis=1, keepdims=True)
np.where(cond, T, 0)
```

Out[10]:

```
array([[25, 27, 29, 24, 26, 18, 32],
       [26, 28, 30, 25, 27, 19, 33],
       [ 0,  0,  0,  0,  0,  0,  0]])
```

With `np.all`, all values must satisfy the criterium:

```
In [11]:
cond = (T < 32).all(axis=1, keepdims=True)
np.where(cond, T, 0)
```

```
Out[11]:
array([[ 0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0],
       [22, 24, 26, 21, 23, 15, 29]])
```

Find out duplicate values

`np.unique` eliminates values that occur several times:

```
In [12]: rng = np.random.default_rng(42)
         arr = rng.integers(0, 2, (12, 3))
         arr
```

```
Out[12]:   array([[0, 1, 1],
                  [0, 0, 1],
                  [0, 1, 0],
                  [0, 1, 1],
                  [1, 1, 1],
                  [1, 1, 0],
                  [1, 0, 1],
                  [0, 0, 1],
                  [1, 1, 0],
                  [1, 1, 0],
                  [0, 0, 0],
                  [1, 1, 0]])
```

```
In [13]: np.unique(arr, axis=0)      # there exists duplicated values along axis 0
```

```
Out[13]: array([[0, 0, 0],
                [0, 0, 1],
                [0, 1, 0],
                [0, 1, 1],
                [1, 0, 1],
                [1, 1, 0],
                [1, 1, 1]])
```

```
In [14]: np.unique(arr, axis=1)      # no duplicated values along axis 1
```

```
Out[14]:
array([[0, 1, 1],
       [0, 0, 1],
       [0, 1, 0],
       [0, 1, 1],
       [1, 1, 1],
       [1, 1, 0],
       [1, 0, 1],
       [0, 0, 1],
       [1, 1, 0],
       [1, 1, 0],
       [0, 0, 0],
       [1, 1, 0]])
```

```
In [15]: np.unique(arr)                          # no other values than 0 and 1 in the array
```

```
Out[15]: array([0, 1])
```

Element-wise maximum

`np.maximum` can calculate the maximum of several arrays **element-wise**.

note: this is very different from `np.max` that takes the maximum value of a single array.

In [16]:

```
v1 = np.array([1, 2, 3, 4])
v2 = np.array([3, 2, 1, 0])
```

In [17]:

```
_ = np.maximum(v1, v2)                    # a new array is created to store the result
print(_)
```

```
[3 2 3 4]
```

Advanced: `np.maximum` is one of the `numpy` function that can allocate memory in an already-defined variable. This is done using parameter `out`. This is useful to modify the content of a variable that has already been passed to several different functions.

In [18]:

```
out = np.full(4, 1000)       # an array of shape (4,) with constant value 1000
out
```

Out[18]:

```
array([1000, 1000, 1000, 1000])
```

In [19]:

```
_ = np.maximum(v1, v2, out=out)
print(_ is out)                     # the result of np.maximum is stored in the alre
out
```

```
True
```

Out[19]:

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

Index of maximum

Let's suppose `T` is an array with 1000 values. One can determine the index of the maximum value of `T` using `np.argmax`.

```
In [20]: T = np.sin(np.linspace(0, np.pi, 1000)) * 10
```

On peut trouver le pas de temps pour lequels la température est maximale.

```
In [21]: np.argmax(T)
```

```
Out[21]: 499
```

Vectorization of Python functions

Some Python operations have no meaning for `numpy`.

For instance, the function below is running fine for usual Python scalars:

In [22]:

```
def f(a, b):
    if a < b:
        return a + b
    else:
        return a * b

f(3, 5)
```

Out[22]:

```
8
```

But it does not work with `numpy` arrays as comparison of 2 arrays using `<` is unclear for `numpy`: it does not know if the condition must be met for all elements (`numpy.all`) or at least one (`numpy.any`):

In [23]:
```
arr1, arr2 = np.split(np.random.randint(1, 5, 16), 2)
print(arr1)
print(arr2)
f(arr1, arr2)
```

```
[2 2 2 1 4 4 4 1]
[4 2 4 1 1 4 4 4]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[23], line 4
      2 print(arr1)
      3 print(arr2)
----> 4 f(arr1, arr2)

Cell In[22], line 2, in f(a, b)
      1 def f(a, b):
----> 2     if a < b:
      3         return a + b
      4     else:

ValueError: The truth value of an array with more than one element is ambigu
ous. Use a.any() or a.all()
```

In this case, we want `f` to be applied element-wise. Thus best choice is to use `np.where`:

In [24]:

```
np.where(arr1 < arr2,
         arr1 + arr2 ,
         arr1 * arr2)
```

Out[24]:

```
array([ 6,  4,  6,  1,  4, 16, 16,  5])
```

Another way is to use the `np.vectorize` function that make `f` element-wise for `numpy` arrays:

In [25]:

```
vectorized_f = np.vectorize(f)
vectorized_f(arr1, arr2)
```

Out[25]:

```
array([ 6,  4,  6,  1,  4, 16, 16,  5])
```

Advanced: beware, `np.vectorize`:

- is a bad choice for performance
- define a function that makes hypotheses regarding data type on the first call

Multi-variables functions

Let's define a 3-variables function.

In [26]:

```
def f(a, b, c):
    return (a + b) ** c

f(2, 3, 4)
```

Out[26]:

```
625
```

The function must be evaluated on the following values:

In [27]:

```
A = (2, 3, 4)
B = (4, 5)
C = (5, 6)
```

Method 1

First choice is define several loops:

```
In [28]:
results = {}
for a in A:
    for b in B:
        for c in C:
            results[(a, b, c)] = f(a, b, c)
print(results)
```

```
{(2, 4, 5): 7776, (2, 4, 6): 46656, (2, 5, 5): 16807, (2, 5, 6): 117649, (3,
4, 5): 16807, (3, 4, 6): 117649, (3, 5, 5): 32768, (3, 5, 6): 262144, (4, 4,
5): 32768, (4, 4, 6): 262144, (4, 5, 5): 59049, (4, 5, 6): 531441}
```

This solution is **very slow**.

Method 2

Instead, let's use `np.meshgrid` to create some evaluation grid:

In [29]:

```
aa, bb, cc = np.meshgrid(A, B, C)
```

Then vectorize the `f` function and apply the vectorized version on the grid:

In [30]:

```
vectorized_f = np.vectorize(f)
results = vectorized_f(aa, bb, cc)
print(results)
```

```
[[[  7776  46656]
  [ 16807 117649]
  [ 32768 262144]]

 [[ 16807 117649]
  [ 32768 262144]
  [ 59049 531441]]]
```

Results are the same as with method 1 but presented in a different way:

- along axis 2 (`axis=2`), values change according to `C`, with `a` and `b` being constant
- along axis 1 (`axis=1`), values change according to `B`, with `a` and `c` being constant
- along axis 0 (`axis=0`), values change according to `A`, with `b` and `c` being constant

Let's understand what is performed by `np.meshgrid` using a simpler 2D example:

In [31]:

```
aa, bb = np.meshgrid(A, B)
print(aa)
print(bb)
```

```
[[2 3 4]
 [2 3 4]]
[[4 4 4]
 [5 5 5]]
```

`aa` and `bb` are two arrays of shape (`len(B)` , `len(A)`). `aa` contains the values of `A` , repeated as many times as needed (`len(B)` times). Same for `bb` .

Using values of both `aa` and `bb` gives an exhaustive grid to evaluate a 2D function.

In [32]:

```
np.stack([aa, bb], axis=-1)
```

Out[32]:

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

       [[2, 5],
        [3, 5],
        [4, 5]]])
```