

# 1 Basic Functions and Numerical Indexing

## 1.1 Mathematical Functions

`sum` and `cumsum`

`sum` sums elements in an array. By default, it will sum all elements in the array, and so the second argument is normally used to provide the axis to use

- 0 to sum down columns
- 1 to sum across rows

`cumsum` produces the cumulative sum of the values in the array, and is also usually used with the second argument to indicate the axis to use.

```
>>> x = randn(3,4)
>>> x
array([[0.08542071, 2.05598312, 2.1114733 , 0.7986635 ],
       [0.17576066, 0.83327885, 0.64064119 , 0.25631728],
       [0.38226593, 1.09519101, 0.29416551 , 0.03059909]])
>>>
>>> sum(x) # all elements
0.62339964288008698
>>>
>>> sum(x, 0) # Down rows, 4 elements
array([0.6434473, 2.31789529, 1.76499762, 0.57294532])
>>>
>>> sum(x, 1) # Across columns, 3 elements
array([ 0.76873297,
        0.23944028,
        1.15269233])
>>>
>>> cumsum(x,0) # Down rows
array([[0.08542071, 2.05598312, 2.1114733 , 0.7986635 ],
       [0.26118137, 1.22270427, 1.47083211, 0.54234622],
       [0.6434473 , 2.31789529, 1.76499762, 0.57294532]])
```

`sum` and `cumsum` can both be used as function or as methods. When used as methods, the first input is the axis so that `sum(x,0)` is the same as `x.sum(0)`.

**prod, cumprod**

**prod** and **cumprod** behave similarly to **sum** and **cumsum** except that the product and cumulative product are returned. As with **sum** and **cumsum**, **prod** and **cumprod** can be called as function or methods.

**diff**

**diff** computes the finite difference of a vector (also array) and returns ***n-1*** an element vector when used on an ***n*** element vector.

**diff** operates on the last axis by default, and so **diff(x)** operates across columns and returns **x[:,1:size(x,1)]x[:, : size(x,1)-1]** for a 2-dimensional array.

**diff** takes an optional keyword argument **axis** so that **diff(x, axis=0)** will operate across rows. **diff** can also be used to produce higher order differences (e.g. double difference).

```
>>> x= randn(3,4)
>>> x = randn(3,4)
>>> x
array([[0.08542071, 2.05598312, 2.1114733 , 0.7986635 ],
       [0.17576066, 0.83327885, 0.64064119 , 0.25631728],
       [0.38226593, 1.09519101, 0.29416551 , 0.03059909]])
>>>
>>> diff(x, axis=0)
array([[0.09033996, 2.88926197, 2.75211449, 1.05498078],
       [0.20650526, 1.92846986, 0.9348067 , 0.28691637]])
>>>
>>> diff(x, 2, axis=0) # Double difference, columnbycolumn
array([[0.11616531,
       4.81773183,
       3.68692119, 1.34189715]])
```

**exp**

**exp** returns the element-by-element exponential ( $e^x$ ) for an array.

**log**

**log** returns the element-by-element natural logarithm ( $\ln(x)$ ) for an array.

**log10**

**log10** returns the element-by-element base-10 logarithm ( $\log_{10}(x)$ ) for an array.

## 1.2 Rounding

`around`, `round`

`around` rounds to the nearest integer, or to a particular decimal place when called with two arguments.

```
>>> x = randn(3)
array([ 0.60675173, 0.3361189 , 0.56688485])
>>> around(x)
array([ 1., 0., 1.])
>>>
>>> around(x, 2)
array([ 0.61, 0.34, 0.57])
```

`around` can also be used as a method on an `ndarray` – except that the method is named `round`. For example, `x.round(2)` is identical to `around(x, 2)`.

The change of names is needed to avoid conflicting with the Python built-in function `round`.

### floor

floor rounds to the next smallest integer.

```
>>> x = randn(3)
array([ 0.60675173, 0.3361189, 0.56688485])
>>>
>>> floor(x)
array([ 0., 1., 1.] )
```

### ceil

ceil rounds to the next largest integer.

```
>>> x = randn(3)
array([ 0.60675173, 0.3361189 , 0.56688485])
>>>
>>> ceil(x)
array([ 1., 0., 0.] )
```

Note that the values returned are still floating points and so 0. is the same as 0..

## 1.3 Generating Arrays and Matrices

### linspace

`linspace(l,u,n)` generates a set of `n` points uniformly spaced between `l`, a lower bound (inclusive) and `u`, an upper bound (inclusive).

```
>>> x = linspace(0, 10, 11)
>>> x
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.] )
```

### logspace

`logspace(l,u,n)` produces a set of logarithmically spaced points between  $10^l$  and  $10^u$ . It is identical to `10**linspace(l,u,n)`.

`arange`

- `arange(l,u,s)` produces a set of points spaced by stepsize  $s$  between  $l$ , a lower bound (inclusive) and  $u$ , an upper bound (exclusive).

$$(l, l + s, l + 2s, \dots, u - s)$$

```
>>> x = arange(4, 10, 1.25)
array([ 4. , 5.25, 6.5 , 7.75, 9. ])
```

- `arange` can be used with a single parameter, so that `arange(n)` is equivalent to `arange(0,n,1)`.

```
>>> x = arange(11.0)
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.] )
```

- Note that `arange` will return integer data type if all inputs are integer.

```
>>> x = arange(11)
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

### 1.3.1 meshgrid

`meshgrid` broadcasts two vectors to produce two 2-dimensional arrays, and is a useful function when plotting 3-dimensional functions.

```
>>> x = arange(5)
>>> y = arange(3)
>>> X,Y = meshgrid(x,y)
>>> X
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
>>> Y
```

## Data Analysis with Python

```
array([[0, 0, 0, 0, 0],  
       [1, 1, 1, 1, 1],  
       [2, 2, 2, 2, 2]])
```

### 1.3.2 `r_`

`r_` is a convenience function which generates 1-dimensional arrays from slice notation. While `r_` is highly flexible, the most common use is

```
r_[ start : end : stepOrCount ]
```

where `start` and `end` are the start and end points, and `stepOrCount` can be either a step size, if a real value, or a count, if complex.

```
>>> # arange equiv
>>> r_[0:10:1]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>
>>> # arange equiv
>>> r_[0:10:.5]
array([ 0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ,
        5.5, 6. , 6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
>>>
>>> # linspace equiv, includes end point
>>> r_[0:10:5j]
array([ 0. , 2.5, 5. , 7.5, 10. ])
```

`r_` can also be used to concatenate slices using commas to separate slice notation blocks.

```
>>> r_[0:2, 7:11, 1:4]
array([ 0, 1, 7, 8, 9, 10, 1, 2, 3])
```

Note that `r_` is not a function and that is used with `[]`.

### 1.3.3 `c_`

`c_` is virtually identical to `r_` except that column arrays are generated, which are 2-dimensional (second dimension has size 1).

```
>>> c_[0:5:2]
array([[0],[2],[4]])
>>>
```

```
>>> c_[1:5:4j]
array([[ 1. ],
       [ 2.33333333],
       [ 3.66666667],
       [ 5. ]])
```

`c_`, like `r_`, is not a function and is used with `[]`.

#### 1.3.4 `ix_`

`ix_(a,b)` constructs an n-dimensional open mesh from n 1-dimensional lists or arrays. The output of `ix_` is an n-element tuple containing 1-dimensional arrays. The primary use of `ix_` is to simplify selecting slabs inside a matrix. Slicing can also be used to select elements from an array as long as the slice pattern is regular.

`ix_` is particularly useful for selecting elements from an array using indices which are not regularly spaced, as in the final example.

```
>>> x = reshape(arange(25.0),(5,5))
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.],
       [10., 11., 12., 13., 14.],
       [15., 16., 17., 18., 19.],
       [20., 21., 22., 23., 24.]])
>>>
>>> x[ix_([2,3],[0,1,2])] # Rows 2 & 3, cols 0, 1 and 2
array([[10., 11., 12.],
       [15., 16., 17.]])
>>>
>>> x[2:4,:3] # Same, standard slice
array([[10., 11., 12.],
       [15., 16., 17.]])
>>>
>>> x[ix_([0,3],[0,1,4])] # No slice equiv
```



### 1.3.5 mgrid

`mgrid` is very similar to `meshgrid` but behaves like `r_` and `c_` in that it takes slices as input, and uses a real valued variable to denote step size and complex to denote number of values. The output is an  $n + 1$  dimensional vector where the first index of the output indexes the meshes.

```
>>> mgrid[0:3,0:2:.5]
>>>
>>> mgrid[0:3:3j,0:2:5j]
>>>
```

### 1.3.6 ogrid

`ogrid` is identical to `mgrid` except that the arrays returned are always 1-dimensional. `ogrid` output is generally more appropriate for looping code, while `mgrid` is usually more appropriate for vectorized code. When the size of the arrays is large, then `ogrid` uses much less memory.

```
>>> ogrid[0:3,0:2:.5]
[array([[ 0.],
[ 1.],
[ 2.]]) , array([[ 0. , 0.5, 1. , 1.5]])]
>>> ogrid[0:3:3j,0:2:5j]
[array([[ 0. ],
[ 1.5],
[ 3. ]]) ,
array([[ 0. , 0.5, 1. , 1.5, 2. ]])]
```