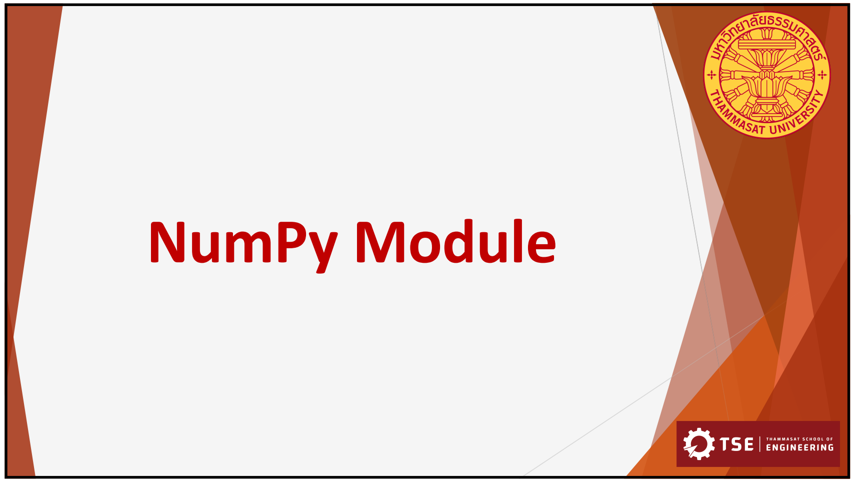
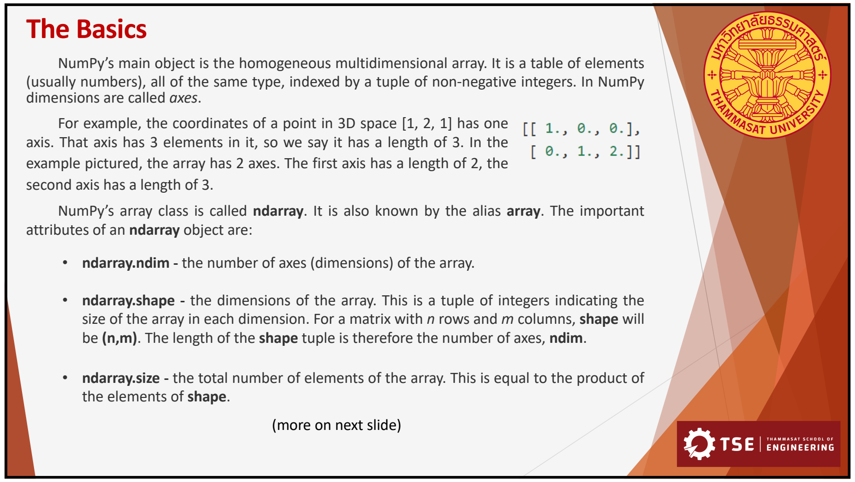


NumPy Module



The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers. In NumPy dimensions are called *axes*.

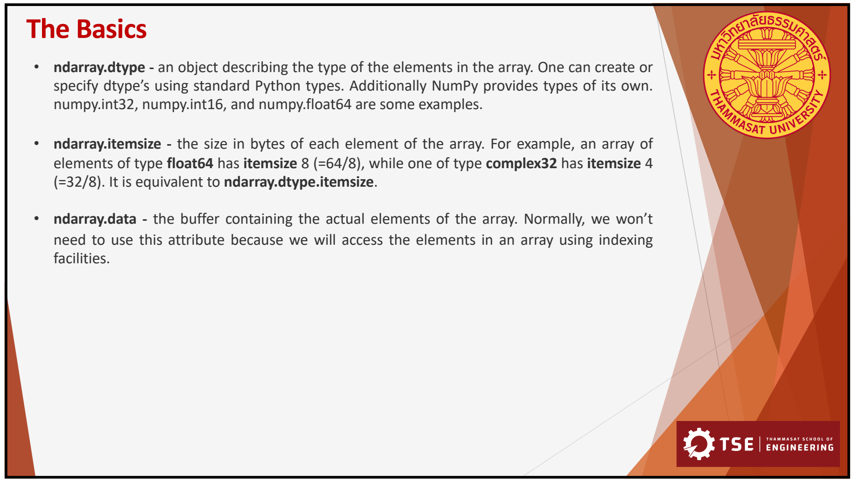
For example, the coordinates of a point in 3D space [1, 2, 1] has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

NumPy's array class is called **ndarray**. It is also known by the alias **array**. The important attributes of an **ndarray** object are:

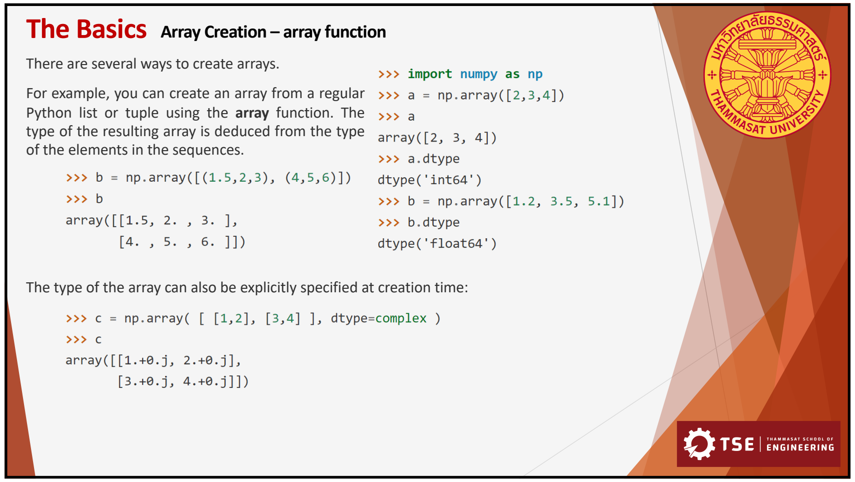
- **ndarray.ndim** - the number of axes (dimensions) of the array.
- **ndarray.shape** - the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, **shape** will be **(n,m)**. The length of the **shape** tuple is therefore the number of axes, **ndim**.
- **ndarray.size** - the total number of elements of the array. This is equal to the product of the elements of **shape**.

(more on next slide)



The Basics

- **ndarray.dtype** - an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
- **ndarray.itemsize** - the size in bytes of each element of the array. For example, an array of elements of type **float64** has **itemsize** 8 (=64/8), while one of type **complex32** has **itemsize** 4 (=32/8). It is equivalent to **ndarray.dtype.itemsize**.
- **ndarray.data** - the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.



The Basics **Array Creation – array function**

There are several ways to create arrays.

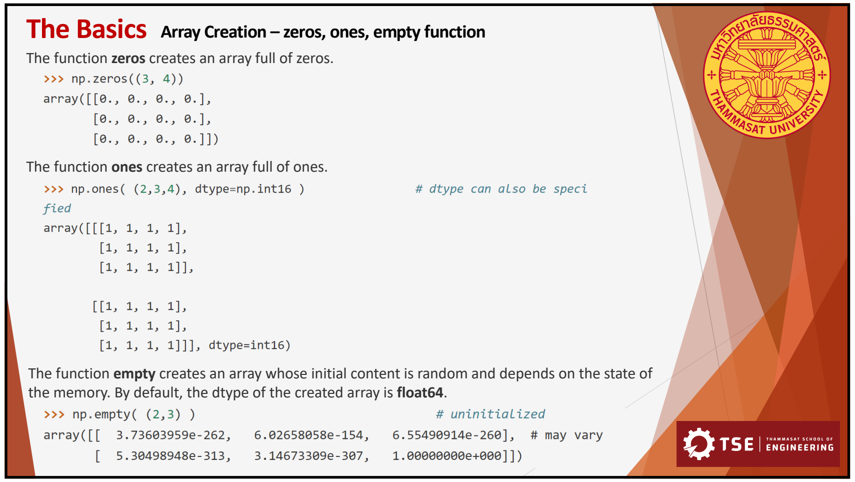
For example, you can create an array from a regular Python list or tuple using the **array** function. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

The type of the array can also be explicitly specified at creation time:

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```



The Basics Array Creation – zeros, ones, empty function

The function **zeros** creates an array full of zeros.

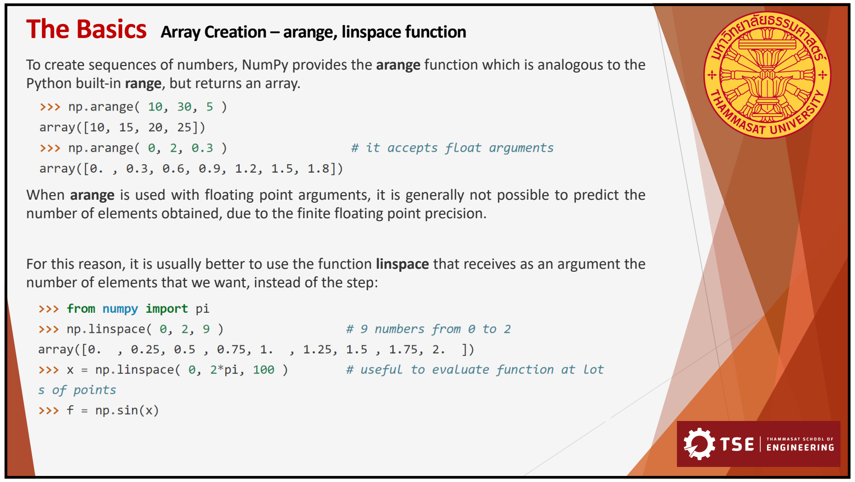
```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

The function **ones** creates an array full of ones.

```
>>> np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be speci
fied
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
```

The function **empty** creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is **float64**.

```
>>> np.empty( (2,3) )                                 # uninitialized
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],  # may vary
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```



The Basics Array Creation – arange, linspace function

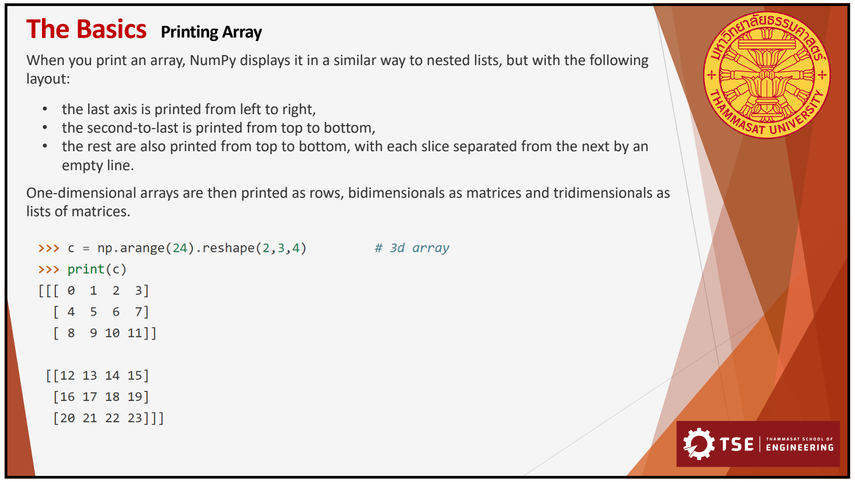
To create sequences of numbers, NumPy provides the **arange** function which is analogous to the Python built-in **range**, but returns an array.

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

When **arange** is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision.

For this reason, it is usually better to use the function **linspace** that receives as an argument the number of elements that we want, instead of the step:

```
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lot
s of points
>>> f = np.sin(x)
```



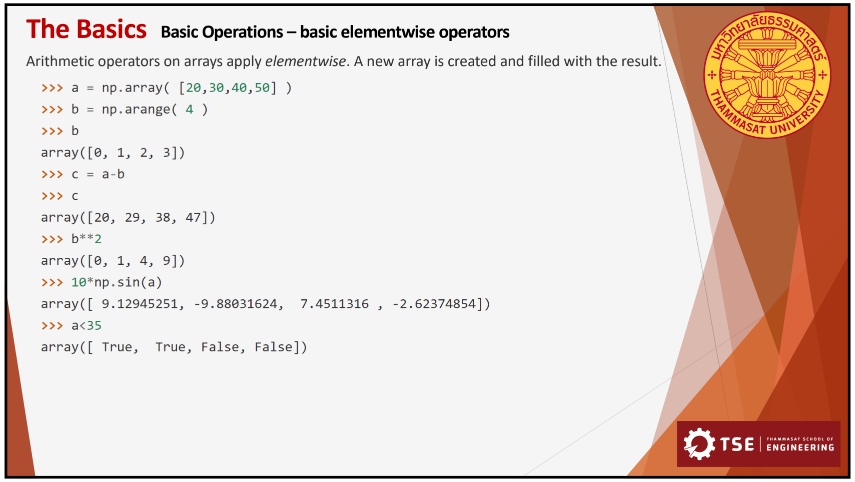
The Basics Printing Array

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

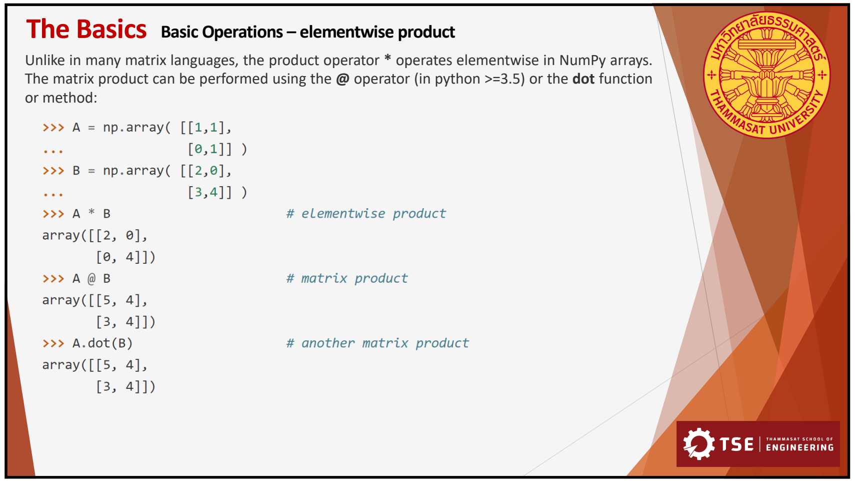
```
>>> c = np.arange(24).reshape(2,3,4)         # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```



The Basics Basic Operations – basic elementwise operators

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

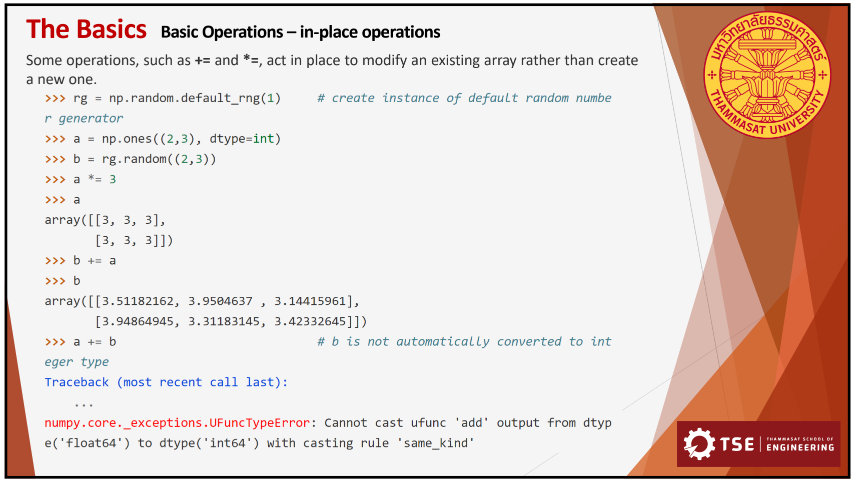
```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([ True,  True, False, False])
```



The Basics Basic Operations – elementwise product

Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the **@** operator (in python >=3.5) or the **dot** function or method:

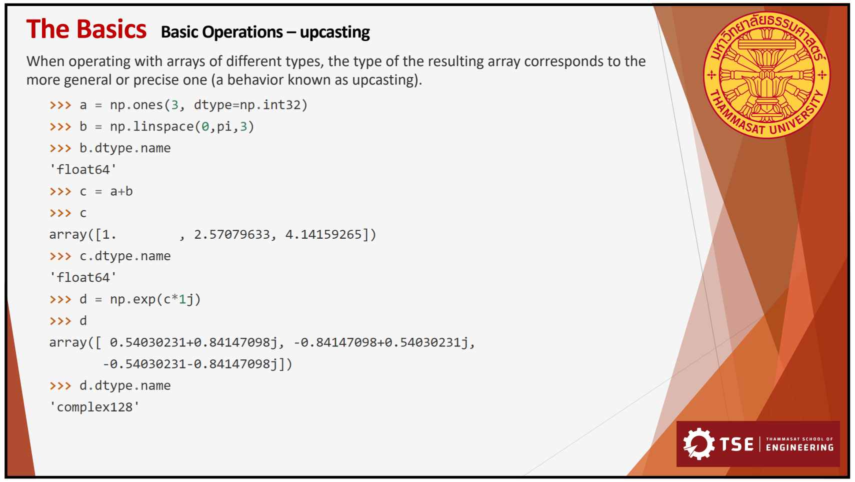
```
>>> A = np.array( [[1,1],
...                [0,1]] )
>>> B = np.array( [[2,0],
...                [3,4]] )
>>> A * B                       # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B                       # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)                    # another matrix product
array([[5, 4],
       [3, 4]])
```



The Basics Basic Operations – in-place operations

Some operations, such as **+=** and ***=**, act in place to modify an existing array rather than create a new one.

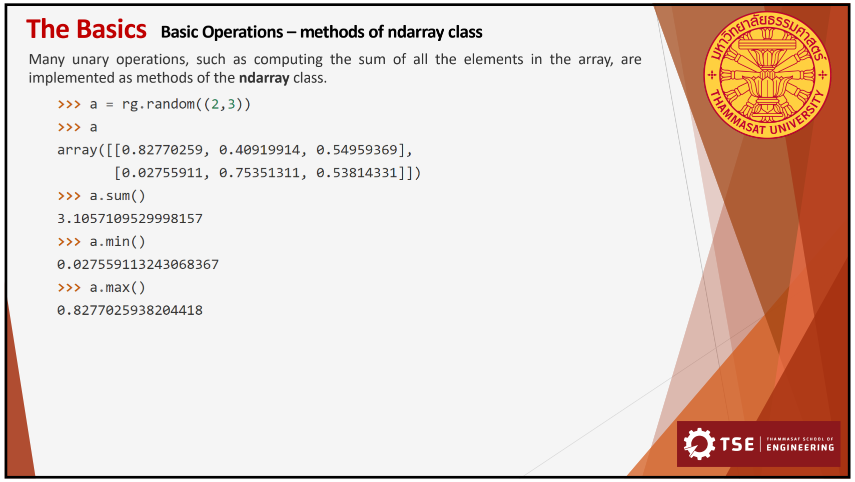
```
>>> rg = np.random.default_rng(1)     # create instance of default random numbe
r generator
>>> a = np.ones((2,3), dtype=int)
>>> b = rg.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[3.51182162, 3.9504637 , 3.14415961],
       [3.94864945, 3.31183145, 3.42332645]])
>>> a += b                            # b is not automatically converted to int
eger type
Traceback (most recent call last):
    ...
numpy.core._exceptions.UFuncTypeError: Cannot cast ufunc 'add' output from dtyp
e('float64') to dtype('int64') with casting rule 'same_kind'
```



The Basics Basic Operations – upcasting

When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([1.        , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```



The Basics Basic Operations – methods of ndarray class

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the **ndarray** class.

```
>>> a = rg.random((2,3))
>>> a
array([[0.82770259, 0.40919914, 0.54959369],
       [0.02755911, 0.75351311, 0.53814331]])
>>> a.sum()
3.1057109529998157
>>> a.min()
0.027559113243068367
>>> a.max()
0.8277025938204418
```

The Basics Basic Operations – methods of ndarray class (cont.)

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the **axis** parameter you can apply an operation along the specified axis of an array:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                            # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                            # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                         # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

The Basics Universal Functions

NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called “universal functions”(**ufunc**). Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([1.        , 2.71828183, 7.3890561 ])
>>> np.sqrt(B)
array([0.        , 1.        , 1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([2., 0., 6.])
```

The Basics Indexing, Slicing and Iterating

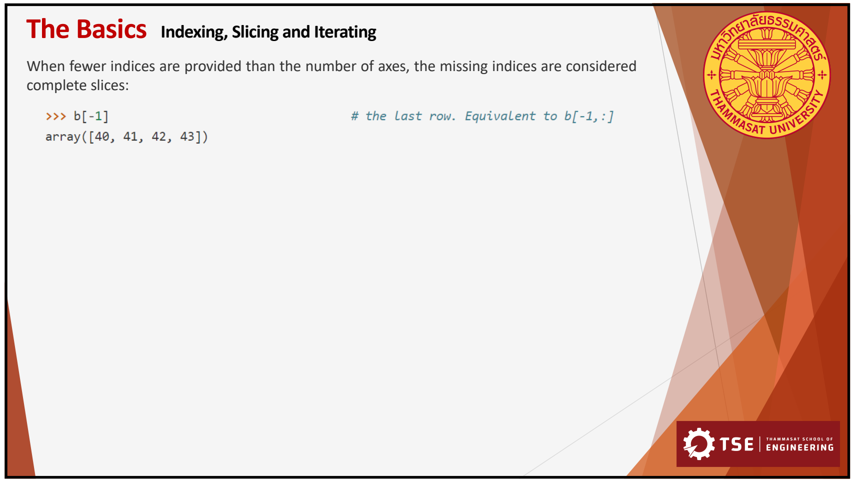
One-dimensional arrays can be indexed, sliced and iterated over, much like list and other Python sequences.

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
# equivalent to a[0:6:2] = 1000;
# from start to position 6, exclusive, set every 2nd element to 1000
>>> a[:6:2] = 1000
>>> a
array([1000,    1, 1000,   27, 1000,  125,  216,  343,  512,  729])
>>> a[ : :-1]                                 # reversed a
array([ 729,  512,  343,  216,  125, 1000,   27, 1000,    1, 1000])
>>> for i in a:
...     print(i**(1/3.))
...
9.999999999999998
1.0
9.999999999999998
3.0
9.999999999999998
4.999999999999999
5.999999999999999
6.999999999999999
7.999999999999999
8.999999999999998
```

The Basics Indexing, Slicing and Iterating

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

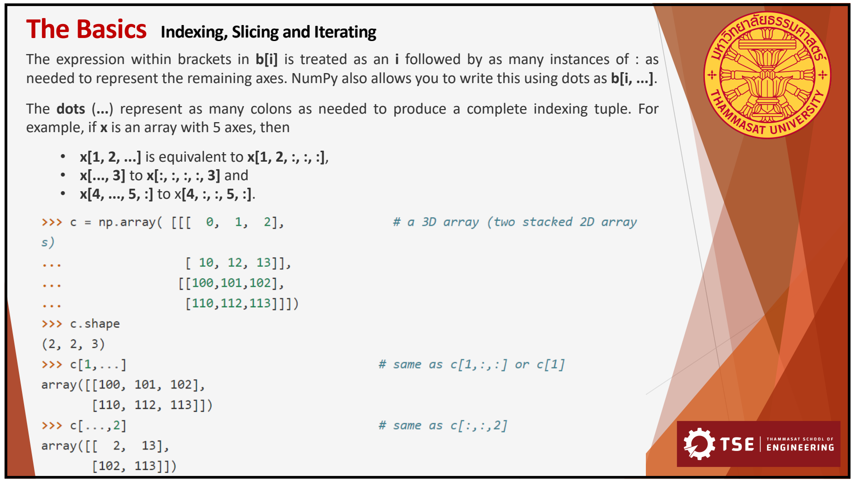
```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]                       # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                        # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                      # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```



The Basics Indexing, Slicing and Iterating

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

```
>>> b[-1]                                   # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```



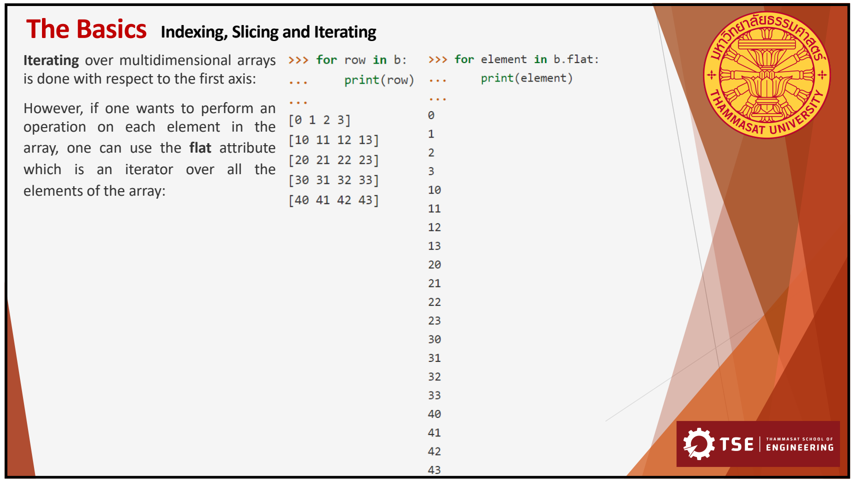
The Basics Indexing, Slicing and Iterating

The expression within brackets in **b[i]** is treated as an **i** followed by as many instances of : as needed to represent the remaining axes. NumPy also allows you to write this using dots as **b[i, ...]**.

The **dots** (...) represent as many colons as needed to produce a complete indexing tuple. For example, if **x** is an array with 5 axes, then

- **x[1, 2, ...]** is equivalent to **x[1, 2, :, :, :]**,
- **x[..., 3]** to **x[:, :, :, :, 3]** and
- **x[4, ..., 5, :]** to x**[4, :, :, 5, :]**.

```
>>> c = np.array( [[[  0,  1,  2],               # a 3D array (two stacked 2D array
s)
...                 [ 10, 12, 13]],
...                [[100,101,102],
...                 [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...]                                   # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                                   # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```



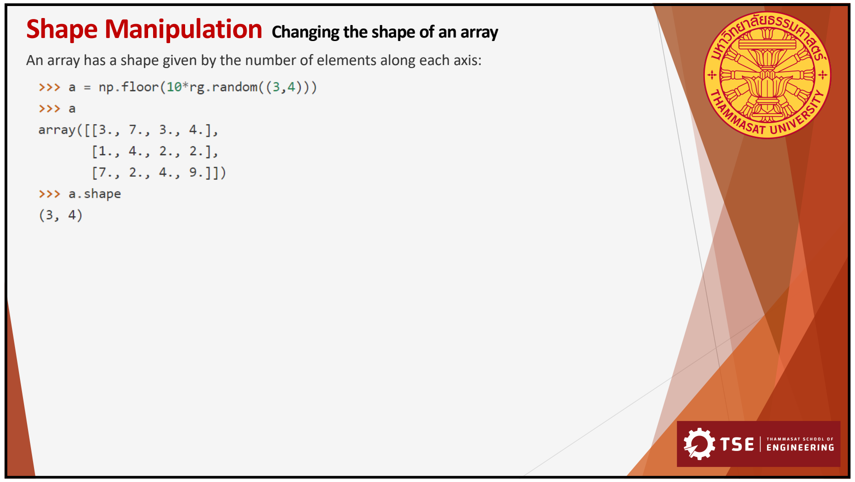
The Basics Indexing, Slicing and Iterating

Iterating over multidimensional arrays is done with respect to the first axis:

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

However, if one wants to perform an operation on each element in the array, one can use the **flat** attribute which is an iterator over all the elements of the array:

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43
```



Shape Manipulation Changing the shape of an array

An array has a shape given by the number of elements along each axis:

```
>>> a = np.floor(10*rg.random((3,4)))
>>> a
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
>>> a.shape
(3, 4)
```

Shape Manipulation Changing the shape of an array

The shape of an array can be changed with various commands. Note that the following three commands all return a modified array, but do not change the original array:

```
>>> a.ravel()  # returns the array, flattened
array([3., 7., 3., 4., 1., 4., 2., 2., 7., 2., 4., 9.])
>>> a.reshape(6,2)  # returns the array with a modified shape
array([[3., 7.],
       [3., 4.],
       [1., 4.],
       [2., 2.],
       [7., 2.],
       [4., 9.]])
>>> a.T  # returns the array, transposed
array([[3., 1., 7.],
       [7., 4., 2.],
       [3., 2., 4.],
       [4., 2., 9.]])
>>> a.T.shape
(4, 3)
>>> a.shape
(3, 4)
```

Shape Manipulation Changing the shape of an array

The **reshape** function returns its argument with a modified shape, whereas the **ndarray.resize** method modifies the array itself:

```
>>> a
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
>>> a.resize((2,6))
>>> a
array([[3., 7., 3., 4., 1., 4.],
       [2., 2., 7., 2., 4., 9.]])
```

If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated:

```
>>> a.reshape(3,-1)
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
```

Shape Manipulation Stacking together different arrays

Several arrays can be stacked together along different axes:

```
>>> a = np.floor(10*rg.random((2,2)))
>>> a
array([[9., 7.],
       [5., 2.]])
>>> b = np.floor(10*rg.random((2,2)))
>>> b
array([[1., 9.],
       [5., 1.]])
>>> np.vstack((a,b))
array([[9., 7.],
       [5., 2.],
       [1., 9.],
       [5., 1.]])
>>> np.hstack((a,b))
array([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
```

Shape Manipulation Stacking together different arrays

The function **column_stack** stacks 1D arrays as columns into a 2D array. It is equivalent to **hstack** only for 2D arrays:

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))     # with 2D arrays
array([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
>>> a = np.array([4.,2.])
>>> b = np.array([3.,8.])
>>> np.column_stack((a,b))     # returns a 2D array
array([[4., 3.],
       [2., 8.]])
>>> np.hstack((a,b))           # the result is different
array([4., 2., 3., 8.])
>>> a[:,newaxis]               # view `a` as a 2D column vector
array([[4.],
       [2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[4., 3.],
       [2., 8.]])
>>> np.hstack((a[:,newaxis],b[:,newaxis]))   # the result is the same
array([[4., 3.],
       [2., 8.]])
```

Shape Manipulation Stacking together different arrays

On the other hand, the function **row_stack** is equivalent to **vstack** for any input arrays. In fact, **row_stack** is an alias for **vstack**:

```
>>> np.column_stack is np.hstack
False
>>> np.row_stack is np.vstack
True
```

In general, for arrays with more than two dimensions, **hstack** stacks along their second axes, **vstack** stacks along their first axes, and **concatenate** allows for an optional arguments giving the number of the axis along which the concatenation should happen.

Shape Manipulation Stacking together different arrays

In complex cases, **r_** and **c_** are useful for creating arrays by stacking numbers along one axis. They allow the use of range literals (“:”)

```
>>> np.r_[1:4,0,4]
array([1, 2, 3, 0, 4])
```

When used with arrays as arguments, **r_** and **c_** are similar to **vstack** and **hstack** in their default behavior, but allow for an optional argument giving the number of the axis along which to concatenate.

Shape Manipulation Splitting one array into several smaller ones

Using **hsplit**, you can split an array along its horizontal axis, either by specifying the number of equally shaped arrays to return, or by specifying the columns after which the division should occur:

```
>>> a = np.floor(10*rg.random((2,12)))
>>> a
array([[6., 7., 6., 9., 0., 5., 4., 0., 6., 8., 5., 2.],
       [8., 5., 5., 7., 1., 8., 6., 7., 1., 8., 1., 0.]])
# Split a into 3
>>> np.hsplit(a,3)
[array([[6., 7., 6., 9.],
       [8., 5., 5., 7.]]), array([[0., 5., 4., 0.],
       [1., 8., 6., 7.]]), array([[6., 8., 5., 2.],
       [1., 8., 1., 0.]])]
# Split a after the third and the fourth column
>>> np.hsplit(a,(3,4))
[array([[6., 7., 6.],
       [8., 5., 5.]]), array([[9.],
       [7.]]), array([[0., 5., 4., 0., 6., 8., 5., 2.],
       [1., 8., 6., 7., 1., 8., 1., 0.]])]
```

vsplit splits along the vertical axis, and **array_split** allows one to specify along which axis to split.

Copies and Views No Copy at All

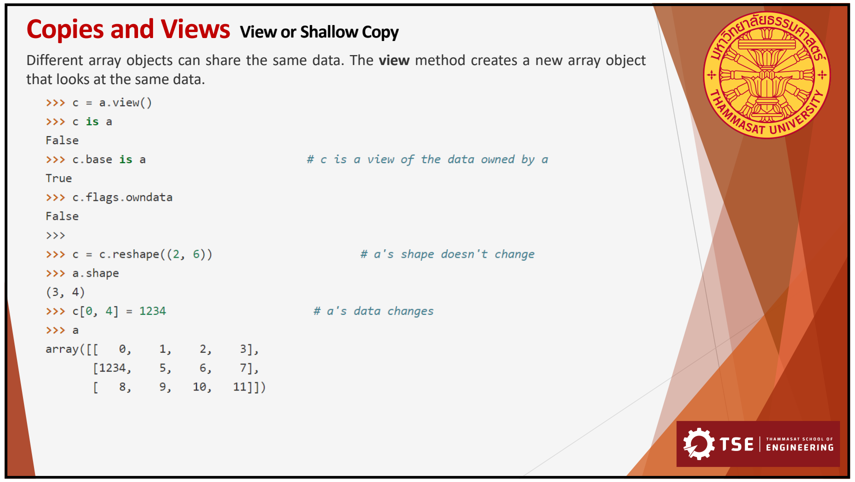
When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are **three cases**, so let’s look at No Copy at All case first.

Simple assignments make no copy of objects or their data.

```
>>> a = np.array([[ 0,  1,  2,  3],
...               [ 4,  5,  6,  7],
...               [ 8,  9, 10, 11]])
>>> b = a           # no new object is created
>>> b is a          # a and b are two names for the same ndarray object
True
```

Python passes mutable objects as references, so function calls make no copy.

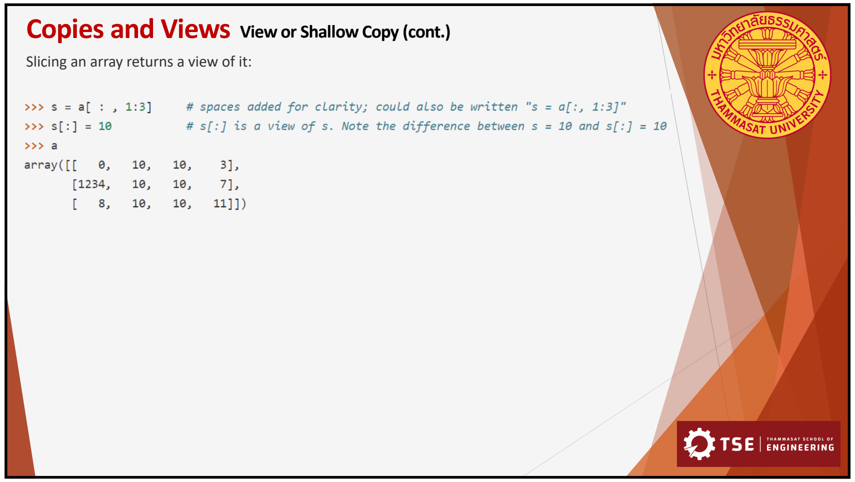
```
>>> def f(x):
...     print(id(x))
...
>>> id(a)                         # id is a unique identifier of an object
148293216  # may vary
>>> f(a)
148293216  # may vary
```



Copies and Views View or Shallow Copy

Different array objects can share the same data. The **view** method creates a new array object that looks at the same data.

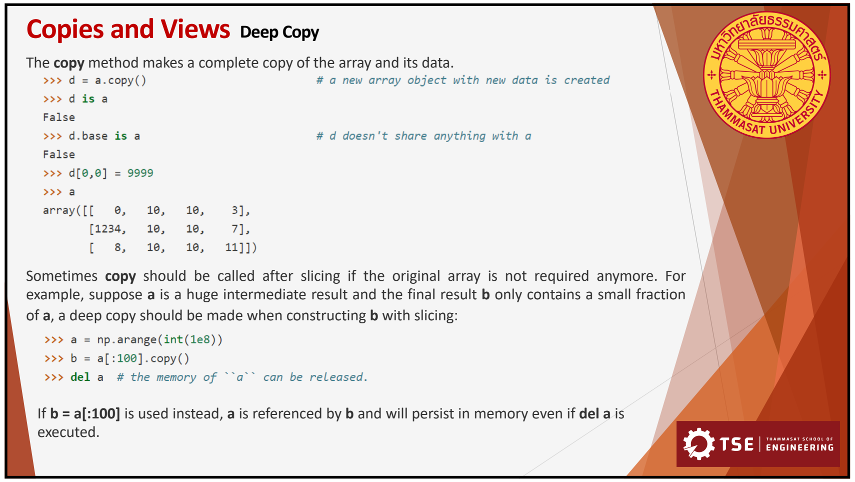
```
>>> c = a.view()
>>> c is a
False
>>> c.base is a                        # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c = c.reshape((2, 6))                      # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0, 4] = 1234                      # a's data changes
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```



Copies and Views View or Shallow Copy (cont.)

Slicing an array returns a view of it:

```
>>> s = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:, 1:3]"
>>> s[:] = 10           # s[:] is a view of s. Note the difference between s = 10 and s[:] = 10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```



Copies and Views Deep Copy

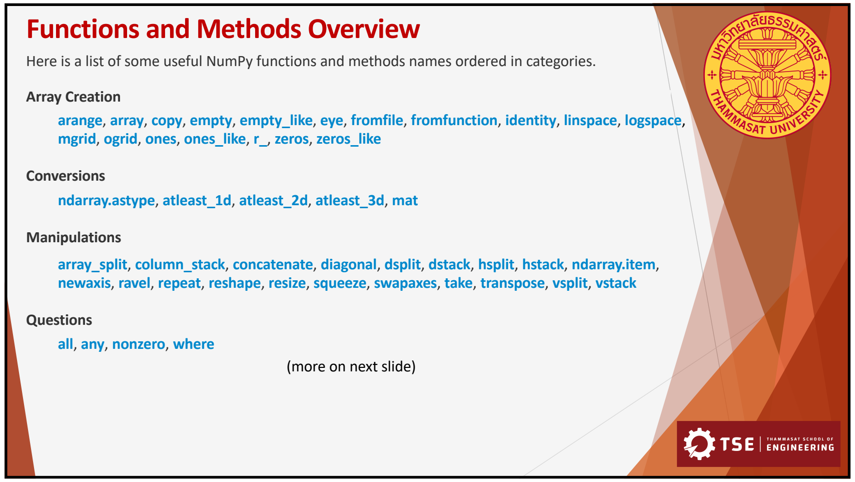
The **copy** method makes a complete copy of the array and its data.

```
>>> d = a.copy()                          # a new array object with new data is created
>>> d is a
False
>>> d.base is a                           # d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Sometimes **copy** should be called after slicing if the original array is not required anymore. For example, suppose **a** is a huge intermediate result and the final result **b** only contains a small fraction of **a**, a deep copy should be made when constructing **b** with slicing:

```
>>> a = np.arange(int(1e8))
>>> b = a[:100].copy()
>>> del a  # the memory of ``a`` can be released.
```

If **b = a[:100]** is used instead, **a** is referenced by **b** and will persist in memory even if **del a** is executed.



Functions and Methods Overview

Here is a list of some useful NumPy functions and methods names ordered in categories.

Array Creation

arange, **array**, **copy**, **empty**, **empty_like**, **eye**, **fromfile**, **fromfunction**, **identity**, **linspace**, **logspace**, **mgrid**, **ogrid**, **ones**, **ones_like**, **r_**, **zeros**, **zeros_like**

Conversions

ndarray.astype, **atleast_1d**, **atleast_2d**, **atleast_3d**, **mat**

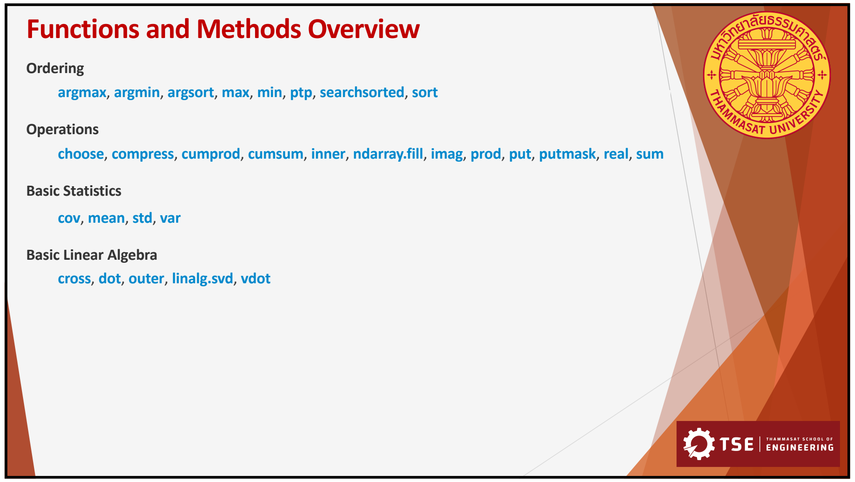
Manipulations

array_split, **column_stack**, **concatenate**, **diagonal**, **dsplit**, **dstack**, **hsplit**, **hstack**, **ndarray.item**, **newaxis**, **ravel**, **repeat**, **reshape**, **resize**, **squeeze**, **swapaxes**, **take**, **transpose**, **vsplit**, **vstack**

Questions

all, **any**, **nonzero**, **where**

(more on next slide)



Functions and Methods Overview

Ordering

argmax, argmin, argsort, max, min, ptp, searchsorted, sort

Operations

choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum

Basic Statistics

cov, mean, std, var

Basic Linear Algebra

cross, dot, outer, linalg.svd, vdot