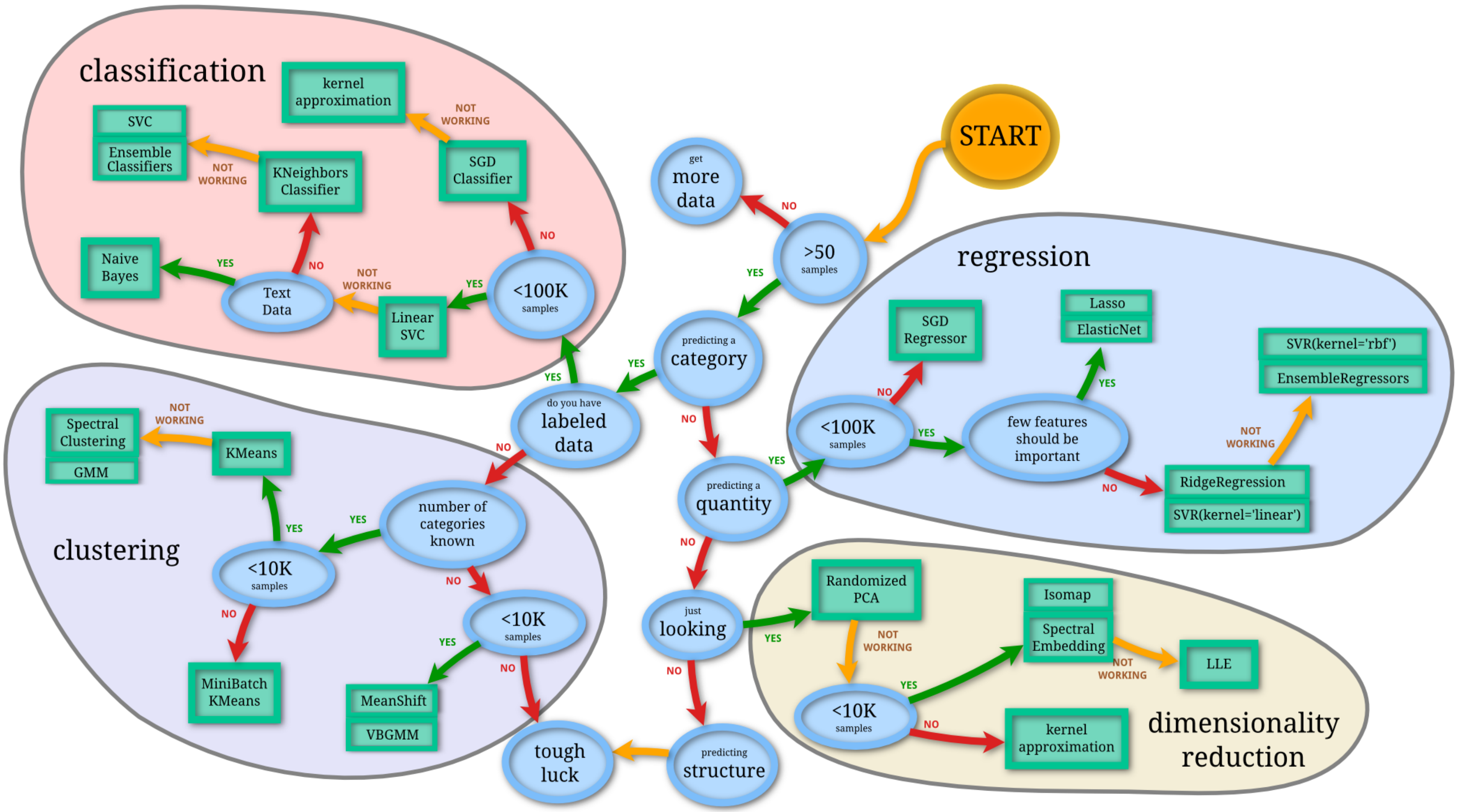
# Numpy & sklearn




*Last modified on 2022/03/30 by f.maire@qut.edu.au*

# Today's Menu

- Python modules
  - numpy
  - sklearn
- Classification examples
  - linear SVC
  - k-nearest neighbours

# What are Numpy and Numpy arrays?

## Numpy arrays

**Python objects**

- high-level number objects: integers, floating point
- containers: lists (costless insertion and append), dictionaries (fast lookup)

**Numpy provides**

- extension package to Python for multi-dimensional arrays
- closer to hardware (efficiency)
- designed for scientific computation (convenience)
- Also known as *array oriented computing*

```
>>> import numpy as np
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

# Why it is useful?

- Memory-efficient container that provides fast numerical operations.

```
File Edit View Search Terminal Help
>>>
>>>
>>>
>>>
>>>
>>>
>>> import timeit
>>> timeit.timeit('[i**2 for i in range(10000)]',number=1000)
2.558639344999392
>>> timeit.timeit('a**2',setup='import numpy as np;a=np.arange(10000)',number=1000)
0.011102509997726884
>>>
```

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> a.shape
(4,)
>>> len(a)
4
```

```
>>> b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> len(b)     # returns the size of the first dimension
2
```

```
>>> a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> b = np.zeros((2, 2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> c = np.eye(3)
>>> c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> d = np.diag(np.array([1, 2, 3, 4]))
>>> d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

```
>>> a = np.random.rand(4)        # uniform in [0, 1]
>>> a
array([ 0.95799151,  0.14222247,  0.08777354,  0.51887998])

>>> b = np.random.randn(4)       # Gaussian
>>> b
array([ 0.37544699, -0.11425369, -0.47616538,  1.79664113])

>>> np.random.seed(1234)         # Setting the random seed
```

# Basic data types

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')

>>> b = np.array([1., 2., 3.])
>>> b.dtype
dtype('float64')
```

# Basic data types

You can explicitly specify which data-type you want:

```
>>> c = np.array([1, 2, 3], dtype=float)
>>> c.dtype
dtype('float64')
```

The **default** data type is floating point:

```
>>> a = np.ones((3, 3))
>>> a.dtype
dtype('float64')
```

```
int32
int64
unit32
unit64
```

```
>>> e = np.array([True, False, False, True])
>>> e.dtype
dtype('bool')
```

# Indexing and slicing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

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

The usual python idiom for reversing a sequence is supported:

```
>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

**Slicing**: Arrays, like other Python sequences can also be sliced:

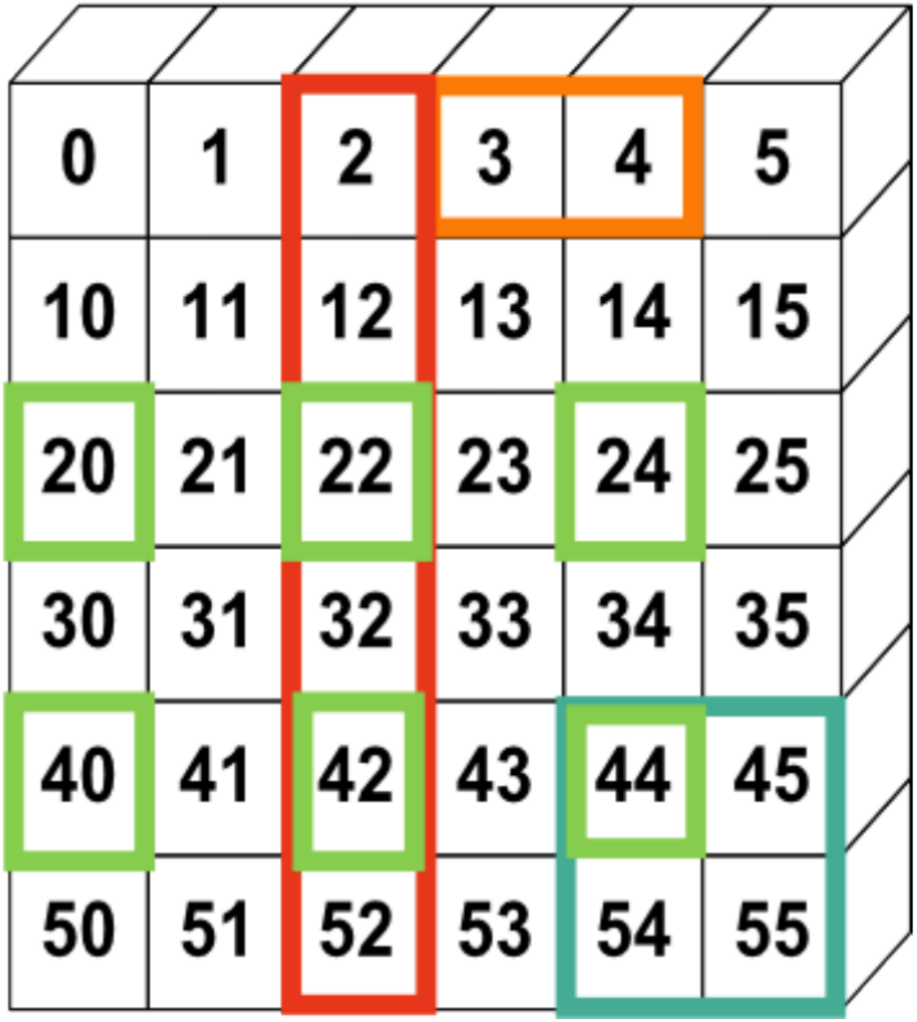
```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [start:end:step]
array([2, 5, 8])
```

```
>>> a[0,3:5]
array([3,4])
```

```
>>> a[4:,4:]
array([[44, 45],
       [54, 55]])
```

```
>>> a[:,2]
array([2,12,22,32,42,52])
```

```
>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



# Copies and views

- A slicing operation creates a view on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory.

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]
>>> b
array([0, 2, 4, 6, 8])
```

```
>>> b[0] = 12
>>> b
array([12,  2,  4,  6,  8])
>>> a   # (!)
array([12,  1,  2,  3,  4,  5,  6,  7,  8,  9])

>>> a = np.arange(10)
>>> c = a[::2].copy()  # force a copy
>>> c[0] = 12
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

# Using boolean masks

- Numpy arrays can be indexed with slices, but also with boolean or integer arrays (masks). It creates copies not views.

```
>>> np.random.seed(3)
>>> a = np.random.random_integers(0, 20, 15)
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,  0, 20, 12,  7, 14])
>>> (a % 3 == 0)
array([False,  True, False,  True, False, False, False,  True, False,
        True,  True, False,  True, False, False], dtype=bool)
>>> mask = (a % 3 == 0)
```

```
>>> extract_from_a = a[mask] # or,  a[a%3==0]
>>> extract_from_a           # extract a sub-array with the mask
array([ 3,  0,  9,  6,  0, 12])
```

Indexing with a mask can be very useful to assign a new value to a sub-array:

```
>>> a[a % 3 == 0] = -1
>>> a
array([10, -1,  8, -1, 19, 10, 11, -1, 10, -1, -1, 20, -1,  7, 14])
```

# Indexing with an array of integers

```
>>> a = np.arange(0, 100, 10)
>>> a
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Indexing can be done with an array of integers, where the same index is repeated several time:

```
>>> a[[2, 3, 2, 4, 2]]  # note: [2, 3, 2, 4, 2] is a Python list
array([20, 30, 20, 40, 20])
```

New values can be assigned with this kind of indexing:

```
>>> a[[9, 7]] = -100
>>> a
array([   0,   10,   20,   30,   40,   50,   60, -100,   80, -100])
```

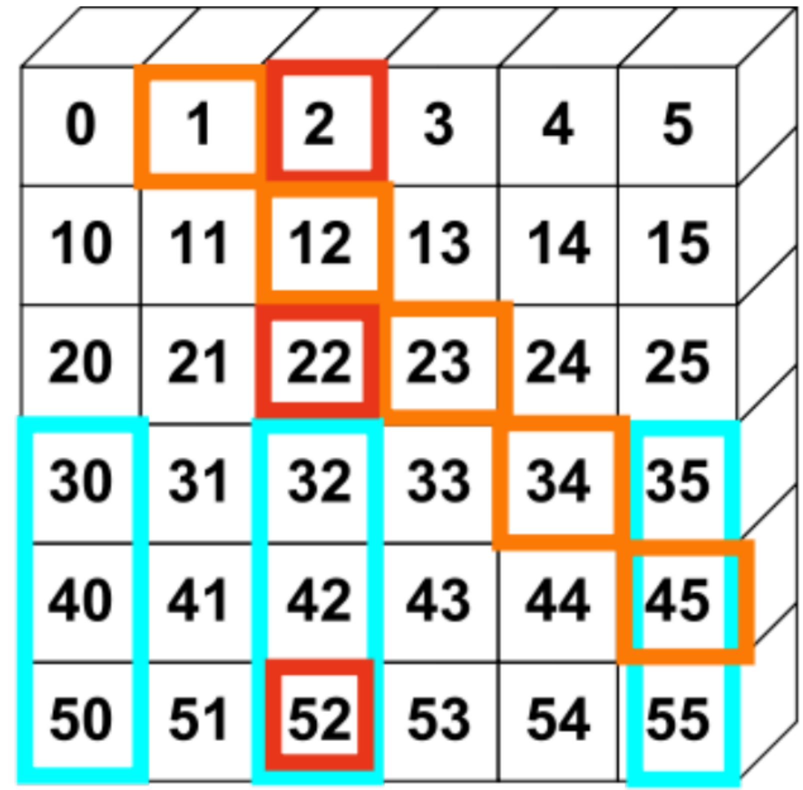
```
>>> a = np.arange(10)
>>> idx = np.array([[3, 4], [9, 7]])
>>> idx.shape
(2, 2)
>>> a[idx]
array([[3, 4],
       [9, 7]])
```

When a new array is created by indexing with an array of integers, the new array has the same shape than the array of integers

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



# Element-wise operations

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

# Warning: Array multiplication is not matrix multiplication

```
>>> c = np.ones((3, 3))
>>> c * c                       # NOT matrix multiplication!
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

```
>>> c.dot(c)
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
```

Matrix multiplication

# Comparisons

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False,  True, False,  True], dtype=bool)
>>> a > b
array([False, False,  True, False], dtype=bool)
```

```
>>> 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
```

Array-wise comparisons

# 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], dtype=bool)
>>> np.logical_and(a, b)
array([ True, False, False, False], dtype=bool)
```

# 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([  1.00000000e+00,   2.71828183e+00,   7.38905610e+00,
         2.00855369e+01,   5.45981500e+01])
```

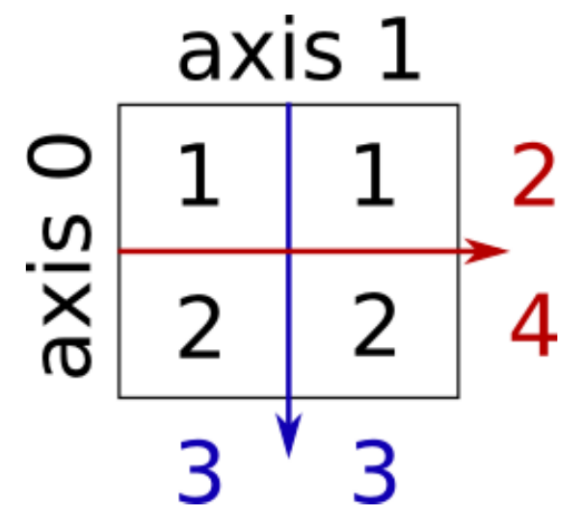
# Linear algebra

- The sub-module numpy.linalg implements basic linear algebra, such as
  - solving linear systems
  - singular value  decomposition
  - etc.
- However, it is not guaranteed to be compiled using efficient routines, and thus it is recommended touse of scipy.linalg

# Basic reductions

```
>>> x = np.array([1, 2, 3, 4])
>>> np.sum(x)
10
>>> x.sum()
10
```

```
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>> x.sum(axis=0)    # columns (first dimension)
array([3, 3])
>>> x[:, 0].sum(), x[:, 1].sum()
(3, 3)
>>> x.sum(axis=1)    # rows (second dimension)
array([2, 4])
>>> x[0, :].sum(), x[1, :].sum()
(2, 4)
```



# Extrema

```
>>> x = np.array([1, 3, 2])
>>> x.min()
1
>>> x.max()
3

>>> x.argmin()   # index of minimum
0
>>> x.argmax()   # index of maximum
1
```

# Logical operations

```
>>> np.all([True, True, False])
False
```

```
>>> np.any([True, True, False])
True
```

```
>>> a = np.zeros((100, 100))
>>> np.any(a != 0)
False
>>> np.all(a == a)
True

>>> a = np.array([1, 2, 3, 2])
>>> b = np.array([2, 2, 3, 2])
>>> c = np.array([6, 4, 4, 5])
>>> ((a <= b) & (b <= c)).all()
True
```

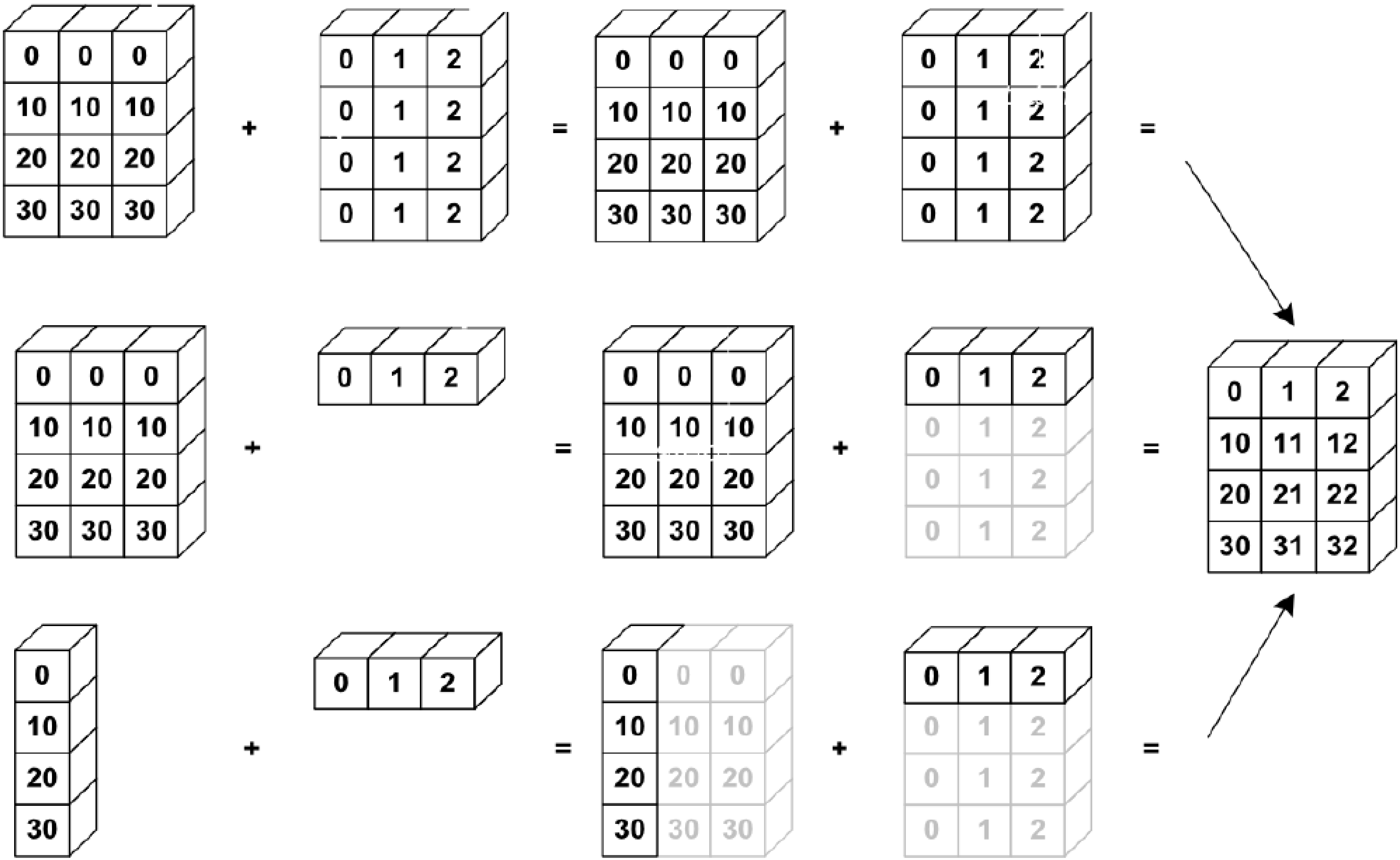
# Statistics

```
>>> x = np.array([1, 2, 3, 1])
>>> y = np.array([[1, 2, 3], [5, 6, 1]])
>>> x.mean()
1.75
>>> np.median(x)
1.5
>>> np.median(y, axis=-1) # last axis
array([ 2.,  5.])

>>> x.std()          # full population standard dev.
0.82915619758884995
```

# Broadcasting

- Basic operations on numpy arrays (addition, etc.) are element-wise
  - This works on arrays of the same size.
  - Nevertheless, It’s also possible to do operations on arrays of different sizes if Numpy can transform these arrays so that they all have the same size: this conversion is called broadcasting.


0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 1 2
10 11 12
20 21 22
30 31 32
0
10
20
30

```
>>> a = np.tile(np.arange(0, 40, 10), (3, 1)).T
>>> a
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
>>> b = np.array([0, 1, 2])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

f.maire@qut.edu.au

```
>>> a = np.ones((4, 5))
>>> a[0] = 2  # we assign an array of dimension 0 to an array of dimension 1
>>> a
array([[ 2.,  2.,  2.,  2.,  2.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

```
>>> a = np.arange(0, 40, 10)
>>> a.shape
(4,)
>>> a = a[:, np.newaxis]  # adds a new axis -> 2D array
>>> a.shape
(4, 1)
>>> a
array([[ 0],
       [10],
       [20],
       [30]])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

# Flattening

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a.ravel()
array([1, 2, 3, 4, 5, 6])
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> a.T.ravel()
array([1, 4, 2, 5, 3, 6])
```

Higher dimensions:
last dimensions ravel out “first”

# Reshaping

```
>>> a.shape
(2, 3)
>>> b = a.ravel()
>>> b = b.reshape((2, 3))
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> a.reshape((2, -1))     # unspecified (-1) value is inferred
array([[1, 2, 3],
       [4, 5, 6]])
```

Warning: ndarray.reshape may return a view (cf help(np.reshape))), or copy

# Adding a dimension

- Indexing with the np.newaxis object allows us to add an axis to an array

```
>>> z = np.array([1, 2, 3])
>>> z
array([1, 2, 3])

>>> z[:, np.newaxis]
array([[1],
       [2],
       [3]])

>>> z[np.newaxis, :]
array([[1, 2, 3]])
```

# Dimension shuffling

```
>>> a = np.arange(4*3*2).reshape(4, 3, 2)
>>> a.shape
(4, 3, 2)
>>> a[0, 2, 1]
5
>>> b = a.transpose(1, 2, 0)
>>> b.shape
(3, 2, 4)
>>> b[2, 1, 0]
5
```

# Sorting data

```
>>> a = np.array([[4, 3, 5], [1, 2, 1]])
>>> b = np.sort(a, axis=1)
>>> b
array([[3, 4, 5],
       [1, 1, 2]])
```

In-place sort:

```
>>> a.sort(axis=1)
>>> a
array([[3, 4, 5],
       [1, 1, 2]])
```

Sorting with fancy indexing:

```
>>> a = np.array([4, 3, 1, 2])
>>> j = np.argsort(a)
>>> j
array([2, 3, 1, 0])
>>> a[j]
array([1, 2, 3, 4])
```

# Finding minima and maxima

```
>>> a = np.array([4, 3, 1, 2])
>>> j_max = np.argmax(a)
>>> j_min = np.argmin(a)
>>> j_max, j_min
(0, 2)
```

# Loading data files (text files)

Example: `populations.txt`:

```
# year  hare    lynx    carrot
1900    30e3    4e3     48300
1901    47.2e3  6.1e3   48200
1902    70.2e3  9.8e3   41500
1903    77.4e3  35.2e3  38200
```

```
>>> data = np.loadtxt('data/populations.txt')
>>> data
array([[  1900.,  30000.,   4000.,  48300.],
       [  1901.,  47200.,   6100.,  48200.],
       [  1902.,  70200.,   9800.,  41500.],
...
```

```
>>> np.savetxt('pop2.txt', data)
>>> data2 = np.loadtxt('pop2.txt')
```

# Numpy Summary

- What do you need to know to get started?
  - Know how to create arrays : array, arange, ones, zeros.
  - Know the shape of the array with array.shape,
  - Use slicing to obtain different views of the array array[::2]
  - Adjust the shape of the array using reshape or flatten it with ravel.
  - Obtain a subset of the elements of an array and/or modify their values with masks

# scikit-learn
# machine learning in Python

# Loading an example dataset

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
```

```
>>> iris.data.shape
(150, 4)
```

This data is stored in the **.data** member, which is a **(n_samples,n_features)** array.

Python 2.7.6 Shell

File Edit Shell Debug Options Windows Help

```
>>> print iris.DESCR
Iris Plants Database

Notes
-----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica
    :Summary Statistics:
    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
```

Ln: 402 Col: 4

Useful fields

- DESCR
- data
- feature_names
- target_names
- target

f.maire@qut.edu.au

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris.data.shape
(150, 4)
>>> print(iris.DESCR)
Iris Plants Database

Notes
-----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica
    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20  0.76     0.9565  (high!)
    ============== ==== ==== ======= ===== ====================
```

# Iris data set (cont.)

- The class of each observation is stored in the .target attribute of the dataset. This is an integer 1D array of length n_samples:

```
>>> iris.target.shape
(150,)
>>> import numpy as np
>>> np.unique(iris.target)
array([0, 1, 2])
```

# Learning and Predicting

- Given our data set, we would like to learn from it and predict the class of non-labelled input. In *scikit-learn*, we learn from existing data by creating an estimator and calling its fit(X, Y) method.

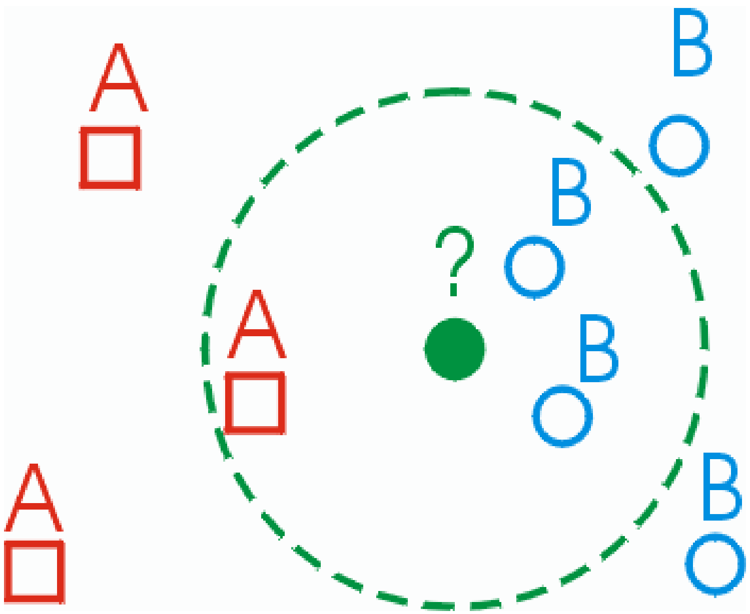
```
>>> from sklearn import svm
>>> clf = svm.LinearSVC()
>>> clf.fit(iris.data, iris.target) # learn from the data
LinearSVC(...)
```

- Once we have learned from the data, we can use our model to predict the most likely outcome on unseen data:

```
>>> clf.predict([[ 5.0,  3.6,  1.3,  0.25]])
array([0], dtype=int32)
```

# k-Nearest neighbors classifier

- The simplest possible classifier is the nearest neighbor classifier



# Summary - sklearn so far

- Neighbors-based classification
  - is a type of instance-based learning
  - Classification is computed from a simple majority vote of the nearest neighbors of each point
  - a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.
- Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection.
- A classifier object clf implements the classifier interface
  - clf.fit( ) ,  clf.predict( ) , clf.score( )