

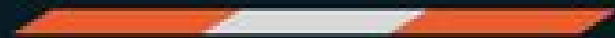


ACADGILD

Mastering Data
Science



Statistics



Student t Distribution, Chi-Squared
Distribution and F Distribution



Agenda

- 1 Student T Distribution
- 2 Determining Student t Values
- 3 Using the T table for all values
- 4 F Distribution
- 5 Determining Values of F
- 6 Check Requisite Conditions
- 7 Inference about Population Variance
- 8 Testing and Estimating Population Variance
- 9 Comparing Two Populations
- 10 Making Inference about $\mu_1 - \mu_2$
- 11 **Test Statistics for $\mu_1 - \mu_2$ (Equal Variances)**
- 12 Test Statistics for $\mu_1 - \mu_2$ (Non-equal Variances)
- 13 Test Estimate for $\mu_1 - \mu_2$ (Equal Variances)
- 14 Inference about the ratio of the Two Variables

Student t Distribution



Here the letter t is used to represent the random variable, hence the name. The density function for the Student t distribution is as follows,

$$f(t) = \frac{\Gamma[(\nu + 1)/2]}{\sqrt{\nu\pi}\Gamma(\nu/2)} \left[1 + \frac{t^2}{\nu} \right]^{-(\nu+1)/2}$$

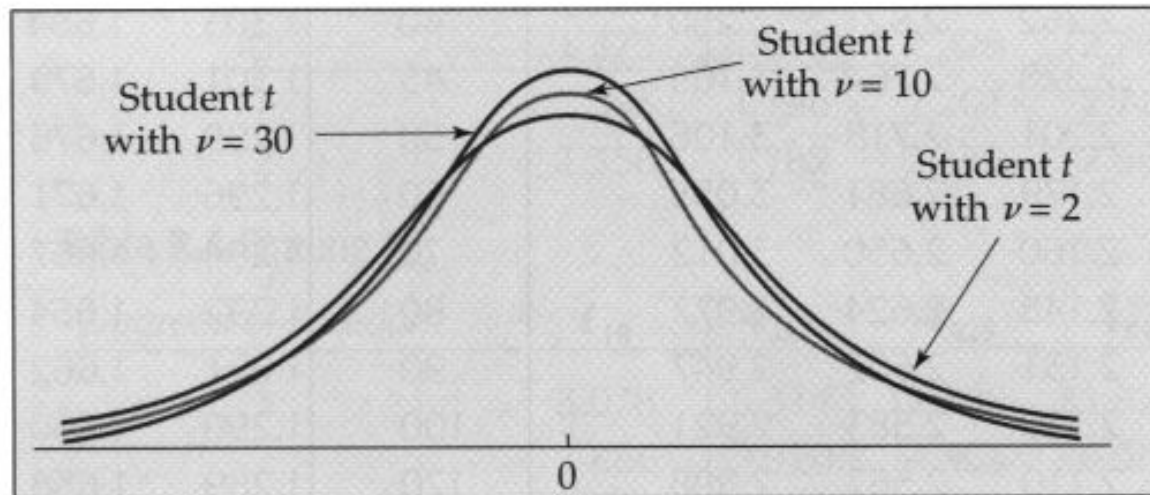
ν (nu) is called the degrees of freedom, and

Γ (Gamma function) is $\Gamma(k) = (k-1)(k-2)\dots(2)(1)$

Student t Distribution



In much the same way that μ and σ define the normal distribution, ν , the degrees of freedom, defines the student t distribution:



As the number of degrees of freedom increases, the t distribution approaches the standard normal distribution.

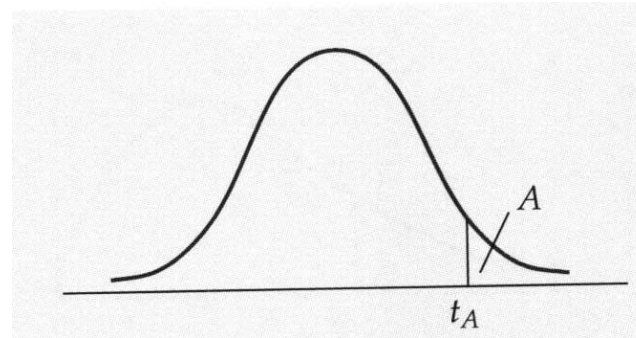


Determining Student t values



- The Student 't' distribution is used extensively in statistical inferences.
- That is, values of a Student '*t*' random variable with ν degrees of freedom such that:

$$P(t > t_{A,\nu}) = A$$



- The values for A are pre-determined “critical” values, typically in the 10%, 5%, 2.5%, 1% and 1/2% range.

Using the t table for all values

Example

- If we want the value of t with 10 degrees of freedom such that the area under the Student 't' curve is 0.05:

Area under the curve value (t_A) : COLUMN

$t_{.05, 10}$

$t_{.05, 10} = 1.812$

Degrees of Freedom : ROW

DEGREES OF FREEDOM	$t_{.100}$	$t_{.050}$	$t_{.025}$	$t_{.010}$	$t_{.005}$
1	3.078	6.314	12.706	31.821	63.657
2	1.886	2.920	4.303	6.965	9.925
3	1.638	2.353	3.182	4.541	5.841
4	1.533	2.132	2.776	3.747	4.604
5	1.476	2.015	2.571	3.365	4.032
6	1.440	1.943	2.447	3.143	3.707
7	1.415	1.895	2.365	2.998	3.499
8	1.397	1.860	2.306	2.896	3.355
9	1.383	1.833	2.262	2.821	3.250
10	1.372	1.812	2.228	2.764	3.169
11	1.363	1.796	2.201	2.718	3.106

F Distribution



F Density Function

The F Density Function is given by:

$$f(F) = \frac{\Gamma\left(\frac{\nu_1 + \nu_2}{2}\right)}{\Gamma\left(\frac{\nu_1}{2}\right)\Gamma\left(\frac{\nu_2}{2}\right)} \left(\frac{\nu_1}{\nu_2}\right)^{\frac{\nu_1}{2}} \frac{F^{\frac{\nu_1-2}{2}}}{\left(1 + \frac{\nu_1 F}{\nu_2}\right)^{\frac{\nu_1 + \nu_2}{2}}}$$

$F > 0$. Two parameters define this distribution, and like we've already seen these are again **degrees of freedom**.

is the " ν_1 numerator" degrees of freedom and

is the " ν_2 denominator" degrees of freedom.

Determining Values of F

Example

What is the value of F for 5% of the area under the right hand “tail” of the curve, with a numerator degree of freedom of 3 and a denominator degree of freedom of 7?

Solution: Use the F look-up

There are different tables for different values of A. Make sure you start with the **correct table!!**

F **.05**, **3**,

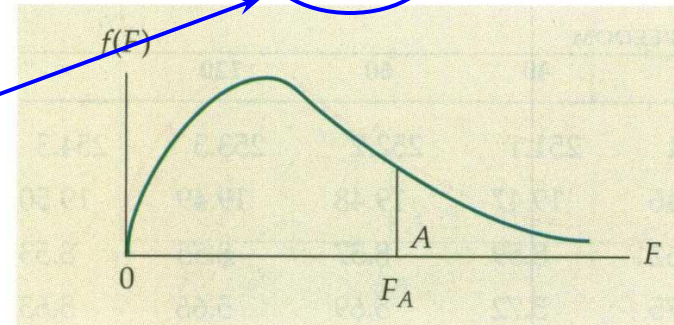
7

Denominator Degrees of Freedom : ROW

Numerator Degrees of Freedom : COLUMN

Table 6(a)

Critical Values of F, $A = .05$



Determining Values of F



For areas under the curve on the left hand side of the curve, we can leverage the following relationship:

$$F_{1-A, \nu_1, \nu_2} = \frac{1}{F_{A, \nu_2, \nu_1}}$$
A diagram illustrating the relationship between the two F-statistic terms. A blue arc connects the ν_1 term in the numerator of the left-hand side to the ν_1 term in the denominator of the right-hand side. A red arc connects the ν_2 term in the numerator of the left-hand side to the ν_2 term in the denominator of the right-hand side.

Pay close attention to the order of the terms!

Check Requisite Conditions



- The Student 't' distribution is robust, which means that if the population is non-normal, the results of the t-test and confidence interval estimate are still valid provided that the population is not extremely non-normal.
- To check this requirement, draw a histogram of the data and see how bell shaped the resulting figure is. If a histogram is extremely skewed (say in that case of an exponential distribution), that could be considered “extremely non-normal” and hence t-statistics would not be valid in this case.

Inference About Population Variation

- If we are interested in drawing inferences about a population's variability, the parameter we need to investigate is the population variance: σ^2
- The sample variance (s^2) is an unbiased, consistent and efficient point estimator for σ^2 .
- Moreover, the statistic, $\chi^2 = \frac{(n-1)s^2}{\sigma^2}$, has a chi-squared distribution, with $n-1$ degrees of freedom.

Testing and Estimating Population Variance

➤ Combining this statistic: $\chi^2 = \frac{(n-1)s^2}{\sigma^2}$

➤ With the probability statement: $P(\chi_{1-\alpha/2}^2 < \chi^2 < \chi_{\alpha/2}^2) = 1 - \alpha$

Yields the confidence interval estimator for σ^2

$$\begin{array}{cc}
 \underbrace{\hspace{15em}} \\
 LCL = \frac{(n-1)s^2}{\chi_{\alpha/2}^2} & UCL = \frac{(n-1)s^2}{\chi_{1-\alpha/2}^2} \\
 \textit{lower confidence limit} & \textit{upper confidence limit}
 \end{array}$$

Comparing Two Populations



- Previously we looked at techniques to estimate and test parameters for one population:

Population Mean μ , Population Variance σ^2

We still consider these parameters when we are looking at two populations, however our interest will now be:

- The difference between two means
- The ratio of two variances

Difference between Two Means



- In order to test and estimate the difference between two population means, we draw random samples from each of two populations. Initially we will consider independent samples, that is samples that are completely unrelated to one another.

Because we compare two population means, we use the statistic: $\bar{x}_1 - \bar{x}_2$

Difference between Two Means



- In order to test and estimate the difference between two population means, we draw random samples from each of two populations. Initially we will consider independent samples, that is samples that are completely unrelated to one another.

Because we compare two population means, we use the statistic: $\bar{x}_1 - \bar{x}_2$

Sampling Distribution of Comparing two population means



- $\bar{x}_1 - \bar{x}_2$ is normally distributed if the original population is normal - or - approximately normal if the population are non-normal and the sample sizes are large ($n_1, n_2 > 30$)
- The expected value of $\bar{x}_1 - \bar{x}_2$ is $\mu_1 - \mu_2$
- The variance of $\bar{x}_1 - \bar{x}_2$ is $\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}$
- The Standard error is $\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$

Making Inferences about $\mu_1 - \mu_2$

- Since $\bar{x}_1 - \bar{x}_2$ is normally distributed, If the original population is normal/approximately normal if the population are non-normal and the sample sizes are large ($n_1, n_2 > 30$), then:

$$z = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

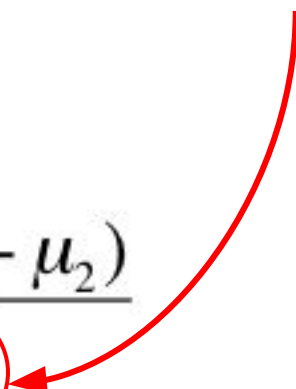
Z is a standard normal (or approximately normal) random variables. We could use this to build test statistics or confidence interval estimators for $\mu_1 - \mu_2$

Making Inferences about $\mu_1 - \mu_2$

- Except that, in practice, the z statistics is rarely used since the **population variances** are unknown.

$$z = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

??



Instead we use a t-statistic. We consider two cases for the unknown population variances: When we believe they are **equal** and conversely when they are **not equal**.

When are Variances Equal?



- How do we know when the population variances are equal?

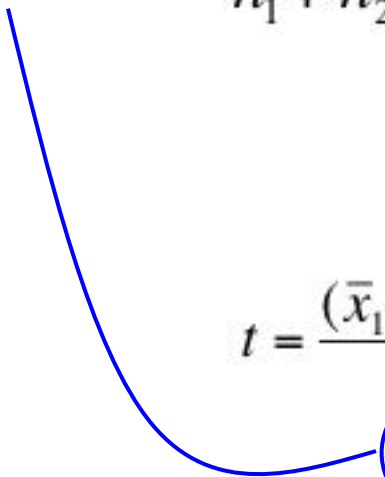
Since the Population variances are unknown, we can't know for certain whether they are equal, but we can **examine the sample variances** and **informally judge** their relative values to determine whether we can assume that the population variance are equal or not.

Test Statistics for $\mu_1 - \mu_2$ (Equal Variances)

1. Calculate s_p^2 - The Pooled variance estimator as,

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}$$

2. And use it here as,

$$t = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sqrt{s_p^2 \left(\frac{1}{n_1} + \frac{1}{n_2} \right)}}, \quad v = n_1 + n_2 - 2$$


↑
Degrees of Freedom

CI Estimator for $\mu_1 - \mu_2$ (Equal Variances)

1. The confidence interval estimator for $\mu_1 - \mu_2$ when the population variances are equal is given by:

$$(\bar{x}_1 - \bar{x}_2) \pm t_{\alpha/2} \sqrt{s_p^2 \left(\frac{1}{n_1} + \frac{1}{n_2} \right)}, \quad v = n_1 + n_2 - 2$$

Pooled Variance Estimator

Pooled Variance Estimator

Test Statistics for $\mu_1 - \mu_2$ (Unequal Variances)

- The test statistic for $\mu_1 - \mu_2$ when the population variances are unequal is given by:

$$t = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sqrt{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)}}, \quad v = \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{\frac{(s_1^2/n_1)^2}{n_1 - 1} + \frac{(s_2^2/n_2)^2}{n_2 - 1}}$$

- Likewise the confidence interval estimator is:

$$(\bar{x}_1 - \bar{x}_2) \pm t_{\alpha/2} \sqrt{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)}, \quad v = \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{\frac{(s_1^2/n_1)^2}{n_1 - 1} + \frac{(s_2^2/n_2)^2}{n_2 - 1}}$$

Degrees of Freedom



Inference about the ratio of two variances

- So far we have looked at comparing measures of central location, namely the mean of two populations.
- When looking at two population variances, we consider the ratio of the variances, i.e. the parameter of the interest to us is:

$$\sigma_1^2 / \sigma_2^2 \text{ or } \frac{\sigma_1^2}{\sigma_2^2}$$

- The sampling statistic: $\frac{s_1^2 / \sigma_1^2}{s_2^2 / \sigma_2^2}$ is F distributed with $\nu_1 = n_1 - 1$ and $\nu_2 = n_2 - 1$ degrees of freedom.

Inference about the ratio of two variances

- Our Null Hypothesis is always:

$$H_0: \sigma_1^2 / \sigma_2^2 = 1$$

That is the variances of the two populations will be equal, hence their ratio will be one.

- Therefor, our statistic simplifies to: $F = s_1^2 / s_2^2$

- $df1 = n_1 - 1$
- $df2 = n_2 - 1$



Email us – support@acadgild.com

NumPy

Numpy introduction

The NumPy package (read as NUMerical PYthon) provides access to:

- Arrays, a new data structure which allow efficient vector and matrix operations.
- Also provides a number of linear algebra operations (such as solving of systems of linear equations, computation of Eigenvectors and Eigenvalues).

History

There are two other implementations that provide nearly the same functionality as NumPy i.e. “Numeric” and “Numarray”:

- Numeric was the first provision of a set of numerical methods (similar to Matlab) for Python. evolved from a PhD project.
- Numarray is a re-implementation of Numeric with certain improvements (Both Numeric and Numarray behave virtually identical for our purpose).
- Early in 2006, it was decided to merge the best aspects of Numeric and Numarray into the Scientific Python (``scipy``) package to provide (a hopefully “final”) ``array`` data type under the module name “NumPy”.

Going forward, we will implement “NumPy” package in the following materials as provided by new SciPy. If SciPy is too old then this doesn’t work for you. In this case, you will find that either “Numeric” or “Numarray” is installed and provided nearly the same capabilities.

Arrays

Array is a new data type (provided by NumPy) which *appears* to be very similar to a list but an array can keep only elements of the same data type whereas a list can mix different kinds of objects. Which means arrays are more efficient to store as we don’t need to store the type for every element. It also makes arrays the data structure of choice for numerical calculations to often deal with vectors and matrices.

Vectors and Matrices (with more than two indices) are called “arrays” in NumPy.

Vectors (1d-arrays)

Vector is the most in need data structure. Lets have a look at a few examples of how to generate Vectors:

Array Creation and Properties

Here are examples of how to create arrays.

Here we create an array using arange and then change its shape to be 3 rows and 5 columns.

```
In [1]: import numpy as np
```

```
In [2]: a = np.arange(15)
```

```
In [3]: a
```

```
Out[3]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
In [4]: a = np.arange(15).reshape(3,5)
```

```
print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

A NumPy array has a lot of meta-data associated with it describing its shape, datatype, etc.

```
In [5]: print(a.ndim)
print(a.shape)
print(a.size)
print(a.dtype)
print(a.itemsize)
print(type(a))
```

```
2
(3, 5)
15
int64
8
<class 'numpy.ndarray'>
```

In [6]: `help(a)`

```

order : {'C', 'F', 'A'}, optional
    If order is 'C' (False) then the result is contiguous (default).
    If order is 'Fortran' (True) then the result has fortran order.
    If order is 'Any' (None) then the result has fortran order
    only if the array already is in fortran order.

__deepcopy__(...)
    a.__deepcopy__() -> Deep copy of array.

    Used if copy.deepcopy is called on an array.

__delitem__(self, key, /)
    Delete self[key].

__divmod__(self, value, /)
    Return divmod(self, value).

__eq__(self, value, /)
    Return self==value.

```

we can create an array from a list

In [7]: `b = np.array([1.0, 2.0, 3.0, 4.0])`
`print(b)`
`print(b.dtype)`
`print(type(b))`

```

[ 1.  2.  3.  4.]
float64
<class 'numpy.ndarray'>

```

we can create a multi-dimensional array of a specified size initialized all to 0 easily. There is also an analogous `ones()` and `empty()` array routine.

Note that, here we explicitly set the datatype for the array.

Unlike lists in Python, all of the elements of a numpy array are of the same datatype.

In [8]: `c = np.eye(10, dtype=np.float64)`
`c`

Out[8]: `array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],`
 `[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],`
 `[0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],`
 `[0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],`
 `[0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],`
 `[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],`
 `[0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],`
 `[0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],`
 `[0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],`
 `[0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])`

linspace (and logspace) create arrays with evenly space (in log) numbers. For logspace, you specify the start and ending powers (base**start to base**stop)

```
In [9]: d = np.linspace(0, 1, 10, endpoint=False)
print(d)
```

```
[ 0.  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9]
```

```
In [10]: e = np.logspace(-1, 2, 15, endpoint=True, base=10)
print(e)
```

```
[ 0.1          0.16378937  0.26826958  0.43939706  0.71968567
 1.17876863   1.93069773  3.16227766  5.17947468  8.48342898
13.89495494  22.75845926  37.2759372  61.05402297 100.          ]
```

As always, as for help -- the numpy functions have very nice docstrings

```
In [11]: help(np.logspace)
```

Help on function logspace in module numpy.core.function_base:

```
logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)
    Return numbers spaced evenly on a log scale.
```

In linear space, the sequence starts at ``base ** start``
(`base` to the power of `start`) and ends with ``base ** stop``
(see `endpoint` below).

Parameters

start : float

``base ** start`` is the starting value of the sequence.

stop : float

``base ** stop`` is the final value of the sequence, unless `endpoint` is False. In that case, ``num + 1`` values are spaced over the interval in log-space, of which all but the last (a sequence of length `num`) are returned.

num : integer, optional

Number of samples to generate. Default is 50.

endpoint : boolean, optional

If true, `stop` is the last sample. Otherwise, it is not included. Default is True.

base : float, optional

The base of the log space. The step size between the elements in ``ln(samples) / ln(base)`` (or ``log_base(samples)``) is uniform. Default is 10.0.

dtype : dtype

The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.

Returns

samples : ndarray

`num` samples, equally spaced on a log scale.

See Also

arange : Similar to linspace, with the step size specified instead of the number of samples. Note that, when used with a float endpoint, the endpoint may or may not be included.

linspace : Similar to logspace, but with the samples uniformly distributed in linear space, instead of log space.

geomspace : Similar to logspace, but with endpoints specified directly.

Notes

Logspace is equivalent to the code

```
>>> y = np.linspace(start, stop, num=num, endpoint=endpoint)
... # doctest: +SKIP
>>> power(base, y).astype(dtype)
... # doctest: +SKIP
```

Examples

```

-----
>>> np.logspace(2.0, 3.0, num=4)
array([ 100.          ,  215.443469   ,  464.15888336, 1000.          ])
>>> np.logspace(2.0, 3.0, num=4, endpoint=False)
array([ 100.          ,  177.827941   ,  316.22776602,  562.34132519])
>>> np.logspace(2.0, 3.0, num=4, base=2.0)
array([ 4.          ,  5.0396842   ,  6.34960421,  8.          ])

```

Graphical illustration:

```

>>> import matplotlib.pyplot as plt
>>> N = 10
>>> x1 = np.logspace(0.1, 1, N, endpoint=True)
>>> x2 = np.logspace(0.1, 1, N, endpoint=False)
>>> y = np.zeros(N)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()

```

we can also initialize an array based on a function:

```
In [12]: f = np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
         f
```

```
Out[12]: array([[0, 1, 2],
                [1, 2, 3],
                [2, 3, 4]])
```

Array Operations

Most operations (+, -, *, /) will work on an entire array at once, element-by-element.

Note that, the multiplication operator is not a matrix multiply (there is a new operator in python 3.5+, @, to do matrix multiplication).

Let's start by creating a simply array.

```
In [13]: a = np.arange(12).reshape(3,4)
         print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Multiplication by a scalar multiplies every element.


```
In [14]: a*2
```

```
Out[14]: array([[ 0,  2,  4,  6],
               [ 8, 10, 12, 14],
               [16, 18, 20, 22]])
```

Adding two arrays adds element-by-element

```
In [15]: a + a
```

```
Out[15]: array([[ 0,  2,  4,  6],
               [ 8, 10, 12, 14],
               [16, 18, 20, 22]])
```

Multiplying two arrays multiplies element-by-element

```
In [16]: a*a
```

```
Out[16]: array([[ 0,  1,  4,  9],
               [16, 25, 36, 49],
               [64, 81, 100, 121]])
```

We can think of our 2-d array 'a' as a 3 x 5 matrix (3 rows, 5 columns). We can take the transpose to get a 5 x 3 matrix, and then we can do a matrix multiplication.

```
In [17]: b = a.transpose()
b
```

```
Out[17]: array([[ 0,  4,  8],
               [ 1,  5,  9],
               [ 2,  6, 10],
               [ 3,  7, 11]])
```

```
In [18]: a @ b
```

```
Out[18]: array([[ 14,  38,  62],
               [ 38, 126, 214],
               [ 62, 214, 366]])
```

We can sum along axes or the entire array.

```
In [19]: a.sum(axis=0)
```

```
Out[19]: array([12, 15, 18, 21])
```

```
In [20]: a.sum()
```

```
Out[20]: 66
```

Also get the extrema

```
In [21]: print(a.min(), a.max())
```

```
0 11
```

Universal Functions

Till now, we have been discussing some of the basic nuts and bolts of NumPy. Now, we will dive into the reasons that NumPy is so important in the Python Data Science world. It provides an easy and flexible interface to optimized computation with arrays of data.

Computation on NumPy arrays can be very fast, or it can be very slow. The key to make it fast is to use **vectorized** operations, generally implemented through NumPy's **universal functions** (ufuncs).

This section tells you the need for NumPy's ufuncs to make repeated calculations on array elements in a much more efficient way. Then, introduces many of the most common and useful arithmetic ufuncs available in the NumPy package.

Universal functions work element-by-element. Let's create a new array scaled by pi

```
In [22]: b = a*np.pi/12.0
print(b)
```

```
[[ 0.          0.26179939  0.52359878  0.78539816]
 [ 1.04719755  1.30899694  1.57079633  1.83259571]
 [ 2.0943951   2.35619449  2.61799388  2.87979327]]
```

```
In [23]: c = np.cos(b)
print(c)
```

```
[[ 1.00000000e+00  9.65925826e-01  8.66025404e-01  7.07106781e-01]
 [ 5.00000000e-01  2.58819045e-01  6.12323400e-17 -2.58819045e-01]
 [-5.00000000e-01 -7.07106781e-01 -8.66025404e-01 -9.65925826e-01]]
```

```
In [24]: d = b + c
```

```
In [25]: print(d)
```

```
[[ 1.          1.22772521  1.38962418  1.49250494]
 [ 1.54719755  1.56781598  1.57079633  1.57377667]
 [ 1.5943951   1.64908771  1.75196847  1.91386744]]
```

Array Slicing: Accessing Sub-arrays

We can use square brackets to access individual array elements. We can also use them to access sub-arrays with the **slice** notation, marked by the colon (':') character.

The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array ```x```.

Use this:

```
``` python
x[start:stop:step]
```
```

If any of these are unspecified, they default to the values ``start=0``, ``stop=``*``size of dimension``, ``step=1``.

Now, We'll take a look at accessing sub-arrays in one dimension and also in multiple dimensions.

```
In [26]: a = np.arange(9)
a
```

```
Out[26]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

Now look at accessing a single element Vs. a range (using slicing)

Giving a single (0-based) index just references a single value.

```
In [27]: a[2]
```

```
Out[27]: 2
```

```
In [28]: print(a[2:3])

[2]
```

```
In [29]: a[2:4]
```

```
Out[29]: array([2, 3])
```

```
In [30]: a[:]
```

```
Out[30]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

Multidimensional Arrays

Multidimensional arrays are stored in a contiguous space in memory which means that the columns/rows need to be unraveled (i.e. flattened) to seem like single one-dimensional array. This can be done via different conventions in different programming languages:

Storage Order:

- * Python/C use **row-major** storage i.e. rows are stored one after the other.
- * Fortran/matlab use **column-major** storage i.e. columns are stored one after another.

The ordering matters when,

- * Passing arrays between languages (we'll talk about this later this semester)

* Looping over arrays -- you want to access elements that are next to one-another in memory

* e.g, in Fortran:

```
<pre>
double precision :: A(M,N)
do j = 1, N
  do i = 1, M
    A(i,j) = ...
  enddo
enddo
</pre>
```

* in C

```
<pre>
double A[M][N];
for (i = 0; i < M; i++) {
  for (j = 0; j < N; j++) {
    A[i][j] = ...
  }
}
</pre>
```

We will try to avoid explicit loops over elements as much as possible implementing NumPy in Python. Let's look at multidimensional arrays:

```
In [31]: a = np.arange(15).reshape(3,5)
a
```

```
Out[31]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])
```

Note that, the output of 'a' shows the row-major storage. The rows are grouped together in the inner `[...]`

Giving a single index (0-based) for each dimension just references a single value in the array.

```
In [32]: a[1,1]
```

```
Out[32]: 6
```

A range of elements will be accessed by slicing. Think of the start and stop in the slice as referencing the left-edge of the slots in the array.

```
In [33]: a[0:2,0:2]
```

```
Out[33]: array([[0, 1],
               [5, 6]])
```

Access a specific column

```
In [34]: a[:,1]
```

```
Out[34]: array([ 1,  6, 11])
```

Sometimes we want a one-dimensional view into the array. Here we see the memory layout (i.e the row-major layout) more explicitly.

```
In [35]: a.flatten()
```

```
Out[35]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

We can also iterate over the first axis (rows).

```
In [36]: for row in a:
          print(row)
```

```
[0 1 2 3 4]
```

```
[5 6 7 8 9]
```

```
[10 11 12 13 14]
```

or element by element

```
In [37]: for e in a.flat:
          print(e)
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

In [38]: `help(a.flatten())`

Help on ndarray object:

```
class ndarray(builtins.object)
|   ndarray(shape, dtype=float, buffer=None, offset=0,
|           strides=None, order=None)
|
|   An array object represents a multidimensional, homogeneous array
|   of fixed-size items. An associated data-type object describes the
|   format of each element in the array (its byte-order, how many bytes it
|   occupies in memory, whether it is an integer, a floating point number,
|   or something else, etc.)
|
|   Arrays should be constructed using `array`, `zeros` or `empty` (refer
|   to the See Also section below). The parameters given here refer to
|   a low-level method (`ndarray(...)`) for instantiating an array.
|
|   For more information, refer to the `numpy` module and examine the
|   methods and attributes of an array.
|
|   Parameters
```

Copying Arrays

Simply using "=" does not make a copy, but much like with lists, you will just have multiple names pointing to the same ndarray object.

Therefore, we need to understand if two arrays, ``A`` and ``B`` point to:

- * The same array, including shape and data/memory space
- * The same data/memory space, but perhaps different shapes (a `_view_`)
- * A separate copy of the data (i.e. stored completely separately in memory)

All of these are possible:

```
* `B = A`
```

This is an `_assignment_` and no copy is made. ``A`` and ``B`` point to the same data in memory and share the same shape, etc. They are just two different labels for the same object in memory.

```
* `B = A[:>`
```

This is a `_view_` or `_shallow copy_`. The shape information for A and B are stored independently, but both point to the same memory location for data.

```
* `B = A.copy()`
```

This is a `_deep_ copy_`. A completely separate object will be created in memory, with a completely separate memory location.

Let's look at a few examples.

```
In [39]: a = np.arange(10)
         print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

Here is an assignment -- we can just use the `is` operator to test for equality.

```
In [40]: b = a
         b is a
```

```
Out[40]: True
```

Since b and a are the same, changes to the shape of one are reflected in the other -- no copy is made.

```
In [41]: b.shape = (2, 5)
         print(b)
         a.shape
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

```
Out[41]: (2, 5)
```

```
In [42]: b is a
```

```
Out[42]: True
```

```
In [43]: print(a)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

A shallow copy creates a new **view** into the array where the data is same, but the array properties can be different.

```
In [44]: a = np.arange(12)
         c = a[:]
         a.shape = (3,4)
```

```
print(a)
print(c)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Since the underlying data has the same memory, changing an element of one is reflected in the other element.

```
In [45]: c[1] = -1
print(a)

[[ 0 -1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Even slices into an array are just views, still pointing to the same memory.

```
In [46]: d = c[3:8]
print(d)

[3 4 5 6 7]
```

```
In [47]: d[:] = 0
```

```
In [48]: print(a)
print(c)
print(d)

[[ 0 -1  2  0]
 [ 0  0  0  0]
 [ 8  9 10 11]]
[ 0 -1  2  0  0  0  0  0  8  9 10 11]
[0 0 0 0 0]
```

There are lots of ways to inquire if two arrays are the same, views, own their own data, etc.

```
In [49]: print(c is a)
print(c.base is a)
print(c.flags.owndata)
print(a.flags.owndata)

False
True
False
True
```

Deep copy helps you to deal independently of the original to make a copy of the array data by implementing a deep copy.


```
In [50]: d = a.copy()
d[:, :] = 0.0

print(a)
print(d)

[[ 0 -1  2  0]
 [ 0  0  0  0]
 [ 8  9 10 11]]
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

Boolean Indexing

There are lots of fun ways to index arrays to access only those elements that meet certain conditions.

```
In [51]: a = np.arange(12).reshape(3,4)
a
```

```
Out[51]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

Here, we set all the elements in the array that are greater than 4 to zero

```
In [52]: a[a > 4] = 0
a
```

```
Out[52]: array([[0, 1, 2, 3],
               [4, 0, 0, 0],
               [0, 0, 0, 0]])
```

and now, all the zeros to -1

```
In [53]: a[a == 0] = -1
a
```

```
Out[53]: array([[ -1,  1,  2,  3],
               [ 4, -1, -1, -1],
               [-1, -1, -1, -1]])
```

```
In [54]: a == -1
```

```
Out[54]: array([[ True, False, False, False],
               [False,  True,  True,  True],
               [ True,  True,  True,  True]], dtype=bool)
```

if we have 2 tests, we need to use `logical_and()` or `logical_or()`.

```
In [55]: a = np.arange(12).reshape(3,4)
a[np.logical_and(a > 3, a <= 9)] = 0.0
a
```

```
Out[55]: array([[ 0,  1,  2,  3],
                [ 0,  0,  0,  0],
                [ 0,  0, 10, 11]])
```

Our test that we index the array with returns a boolean array of the same shape:

```
In [56]: a > 4
```

```
Out[56]: array([[False, False, False, False],
                [False, False, False, False],
                [False, False,  True,  True]], dtype=bool)
```

Avoiding Loops

CPython, Python's default implementation does some operations very slowly. This is in part due to the dynamic, interpreted nature of the language: the fact that types are flexible, so that sequences of operations cannot be compiled down to efficient machine code as in languages like C and Fortran.

Recently there have been various attempts to address this weakness: well-known examples are the [PyPy \(http://pypy.org/\)](http://pypy.org/) project, a just-in-time compiled implementation of Python; the [Cython \(http://cython.org\)](http://cython.org) project, which converts Python code to compilable C code; and the [Numba \(http://numba.pydata.org/\)](http://numba.pydata.org/) project, which converts snippets of Python code to fast LLVM bytecode. Each of these has its strengths and weaknesses, but it is safe to say that none of the three approaches has yet surpassed the reach and popularity of the standard CPython engine.

The relative sluggishness of Python generally manifests itself in situations where many small operations are being repeated – for instance looping over arrays to operate on each element.

In general, you want to avoid loops over elements on an array.

Here, let's create 1-d x and y coordinates and then try to fill some larger array

```
In [57]: M = 32
N = 64
xmin = ymin = 0.0
xmax = ymax = 1.0

x = np.linspace(xmin, xmax, M, endpoint=False)
y = np.linspace(ymin, ymax, N, endpoint=False)

print(x.shape)
print(y.shape)
```

```
(32,)
(64,)
```

we'll time out code

```
In [58]: import time
```

```
In [59]: t0 = time.time()

g = np.zeros((M, N))

for i in range(M):
    for j in range(N):
        g[i,j] = np.sin(2.0*np.pi*x[i]*y[j])

t1 = time.time()
print("time elapsed: {} s".format(t1-t0))

time elapsed: 0.0037317276000976562 s
```

Now let's instead do this using all array syntax. First will extend our 1-d coordinate arrays to be 2-d.

```
In [60]: x2d, y2d = np.meshgrid(x, y, indexing="ij")

print(x2d[:,0])
print(x2d[0,:])

print(y2d[:,0])
print(y2d[0,:])

[ 0.      0.03125  0.0625  0.09375  0.125    0.15625  0.1875  0.21875
 0.25    0.28125  0.3125  0.34375  0.375    0.40625  0.4375  0.46875
 0.5     0.53125  0.5625  0.59375  0.625    0.65625  0.6875  0.71875
 0.75    0.78125  0.8125  0.84375  0.875    0.90625  0.9375  0.96875]

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

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

[ 0.      0.015625  0.03125  0.046875  0.0625    0.078125  0.09375
 0.109375  0.125    0.140625  0.15625  0.171875  0.1875  0.203125
 0.21875  0.234375  0.25    0.265625  0.28125  0.296875  0.3125
 0.328125  0.34375  0.359375  0.375    0.390625  0.40625  0.421875
 0.4375  0.453125  0.46875  0.484375  0.5     0.515625  0.53125
 0.546875  0.5625  0.578125  0.59375  0.609375  0.625    0.640625
 0.65625  0.671875  0.6875  0.703125  0.71875  0.734375  0.75
 0.765625  0.78125  0.796875  0.8125  0.828125  0.84375  0.859375
 0.875    0.890625  0.90625  0.921875  0.9375  0.953125  0.96875
 0.984375]
```

```
In [61]: t0 = time.time()
g2 = np.sin(2.0*np.pi*x2d*y2d)
t1 = time.time()
print("time elapsed: {} s".format(t1-t0))

time elapsed: 0.00035834312438964844 s
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in the following table. Note that, when constructing an array, they can be specified using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

| Data type | Description |
|------------|---|
| bool_ | Boolean (True or False) stored as a byte |
| int_ | Default integer type (same as C long; normally either int64 or int32) |
| intc | Identical to C int (normally int32 or int64) |
| intp | Integer used for indexing (same as C ssize_t; normally either int32 or int64) |
| int8 | Byte (-128 to 127) |
| int16 | Integer (-32768 to 32767) |
| int32 | Integer (-2147483648 to 2147483647) |
| int64 | Integer (-9223372036854775808 to 9223372036854775807) |
| uint8 | Unsigned integer (0 to 255) |
| uint16 | Unsigned integer (0 to 65535) |
| uint32 | Unsigned integer (0 to 4294967295) |
| uint64 | Unsigned integer (0 to 18446744073709551615) |
| float_ | Shorthand for float64. |
| float16 | Half precision float: sign bit, 5 bits exponent, 10 bits mantissa |
| float32 | Single precision float: sign bit, 8 bits exponent, 23 bits mantissa |
| float64 | Double precision float: sign bit, 11 bits exponent, 52 bits mantissa |
| complex_ | Shorthand for complex128. |
| complex64 | Complex number, represented by two 32-bit floats |
| complex128 | Complex number, represented by two 64-bit floats |

More advanced type specification is possible, such as specifying big or little endian numbers; for more information, refer to the [NumPy documentation \(http://numpy.org/\)](http://numpy.org/).

NumPy also supports compound data types, which will be covered in [Structured Data: NumPy's Structured Arrays \(02.09-Structured-Data-NumPy.ipynb\)](#).

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. Here are a few examples of those operations.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`.

`np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

```
In [62]: x = np.array([1, 2, 3, 4])
        y = np.array([4, 3, 2, 1])
        np.concatenate([x, y])
```

```
Out[62]: array([1, 2, 3, 4, 4, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
In [63]: z = [9, 4, 6]
        print(np.concatenate([x, y, z]))

[1 2 3 4 4 3 2 1 9 4 6]
```

It can also be used for two-dimensional arrays:

```
In [64]: grid = np.array([[10, 20, 30],
                        [40, 50, 60]])
```

```
In [65]: # concatenate along the first axis
        np.concatenate([grid, grid])
```

```
Out[65]: array([[10, 20, 30],
                [40, 50, 60],
                [10, 20, 30],
                [40, 50, 60]])
```

```
In [66]: # concatenate along the second axis (zero-indexed)
        np.concatenate([grid, grid], axis=1)
```

```
Out[66]: array([[10, 20, 30, 10, 20, 30],
                [40, 50, 60, 40, 50, 60]])
```

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

```
In [67]: x = [1, 2, 3, 4, 4, 3, 2, 1, 9, 4, 6]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

```
[1 2 3] [4 4] [3 2 1 9 4 6]
```

Notice that N split-points, leads to $N + 1$ subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

```
In [68]: grid = np.arange(25).reshape((5, 5))
grid
```

```
Out[68]: 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]])
```

```
In [69]: left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 5  6]
 [10 11]
 [15 16]
 [20 21]]
[[ 2  3  4]
 [ 7  8  9]
 [12 13 14]
 [17 18 19]
 [22 23 24]]
```

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to reduce an array with a particular operation, we can use the `reduce` method of any ufunc. A `reduce` repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

```
In [70]: x = np.arange(1, 9)
np.add.reduce(x)
```

```
Out[70]: 36
```

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

```
In [71]: np.multiply.reduce(x)
```

```
Out[71]: 40320
```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate`:

```
In [72]: np.add.accumulate(x)
```

```
Out[72]: array([ 1,  3,  6, 10, 15, 21, 28, 36])
```

```
In [73]: np.multiply.accumulate(x)
```

```
Out[73]: array([ 1,  2,  6, 24, 120, 720, 5040, 40320])
```

```
In [ ]:
```