Continuous Random Variables, Expectations, Data, Statistics, Arrays and Tuples

Monte Carlo Methods

Random Variables

A **random variable** is a mapping from the sample space Ω to the set of real numbers $\mathbb{R}$. In other words, it is a numerical value determined by the outcome of the experiment.

Continuous random variable

When a random variable takes on values in the continuum we call it a continuous RV.

Examples

- Volume of water that fell on the Southern Alps yesterday (See video link below)
- Vertical position above sea level, in micrometers, since the original release of a pollen grain at the head waters of Waimakariri
- Resting position in degrees of a roulettet wheel after a brisk spin

Probability Density Function

A RV X with DF F is called continuous if there exists a piece-wise continuous function f, called the **probability density function** (PDF) f of X, such that, for any a, $b \in \mathbb{R}$ with $a < b$,

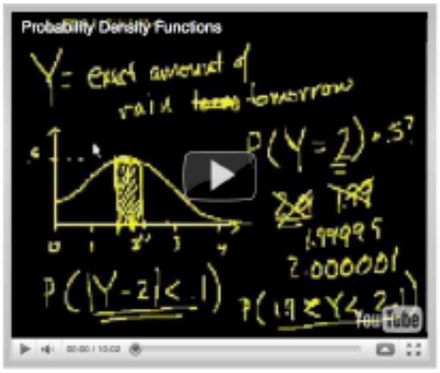
$$P(a < X \le b) = F(b) - F(a) = \int_a^b f(x)\, dx\ .$$

jsMath

The following hold for a continuous RV X with PDF f:

1. For any $x \in \mathbb{R}$, $P(X = x) = 0$.
2. Consequentially, for any $a, b \in \mathbb{R}$ with $a \leq b$
$$P(a < X < b) = P(a < X \leq b) = P(a \leq X \leq b) = P(a \leq X < b)$$
3. By the fundamental theorem of calculus, except possibly at finitely many points (where the continuous pieces come together in the piecewise-continuous f):
$$f(x) = \frac{d}{dx} F(x)$$
4. And of course f must satisfy:
$$\int_{-\infty}^{\infty} f(x)\, dx = P(-\infty < X < \infty) = 1$$

Let's Watch the Khan Academy movie about probability density functions to warm-up to the meaning behind the maths above. Consider the continuous random variable Y that measures the exact amount of rain tomorrow in inches. Think of the probability space $(\Omega, \mathcal{F}, P)$ underpinning this random variable $Y : \Omega \to \mathbf{Y}$. Here the sample space, range or support of the random variable Y denoted by $\mathbf{Y} = [0, \infty) = \{y : 0 \leq y < \infty\}$.


Probability Density Functions
Y = exact amount of rain tomorrow
P(Y=2) = .5?
1.99999
2.000001
P(|Y-2|<.1)
P(1.9≤Y≤2.1)
YouTube

The $Uniform(0,1)$ RV

The $Uniform(0,1)$ RV is a continuous RV with a probability density function (PDF) that takes the value 1 if $x \in [0,1]$ and 0 otherwise. Formally, this is written

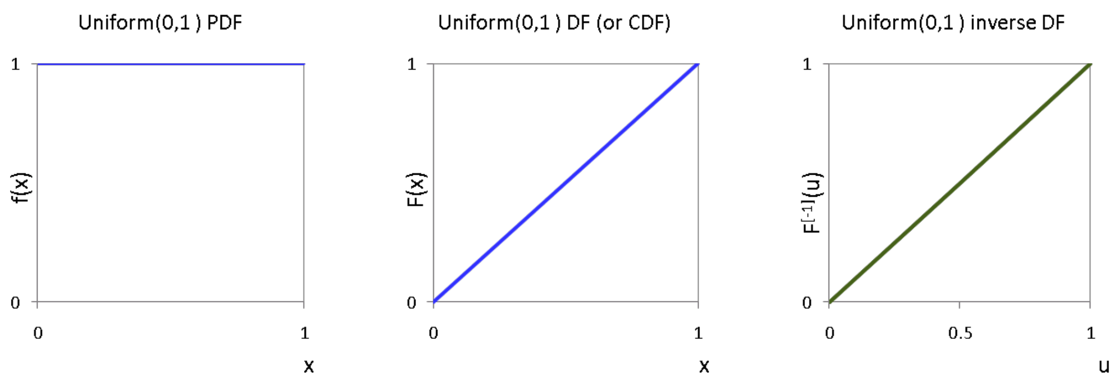
$$f(x) = \mathbf{1}_{[0,1]}(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1, \\ 0 & \text{otherwise} \end{cases}$$

and its distribution function (DF) or cumulative distribution function (CDF) is:

$$F(x) := \int_{-\infty}^{x} f(y)\, dy = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{if } 0 \leq x \leq 1, \\ 1 & \text{if } x > 1 \end{cases}$$

Note that the DF is the identity map in $[0,1]$.

The PDF, CDF and inverse CDF for a $Uniform(0,1)$ RV are shown below



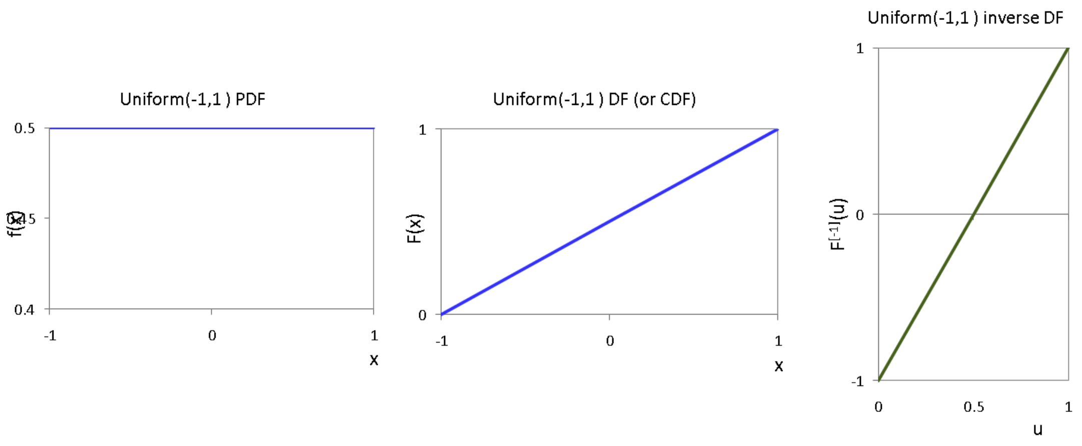
The $Uniform(0,1)$ is sometimes called the Fundamental Model.

The $Uniform(0,1)$ distribution comes from the $Uniform(a,b)$ family.

$$f(x) = \mathbf{1}_{[a,b]}(x) = \begin{cases} \frac{1}{(b-a)} & \text{if } a \leq x \leq b, \\ 0 & \text{otherwise} \end{cases}$$

This is saying that if X is a $Uniform(a,b)$ RV, then all $a \leq x \leq b$ are equally probable. The $Uniform(0,1)$ RV is the member of the family where $a=0, b=1$.

The PDF, CDF and inverse CDF for a $Uniform(-1,1)$ RV are shown below



Sage has a function for simulating samples from a $Uniform(a,b)$ distribution. We will learn more about this later in the course.

```
uniform(-1,1)
```

Expectations

The **expectation** of X is also known as the **population mean**, first moment, or expected value of X.

jsMath

$$E\left(X\right) := \int x \, dF(x) = \begin{cases} \sum_x x f(x) & \text{if } X \text{ is discrete} \\ \int x f(x) \, dx & \text{if } X \text{ is continuous} \end{cases}$$

Sometimes, we denote $E(X)$ by EX for brevity. Thus, the expectation is a single-number summary of the RV X and may be thought of as the average.

In general though, we can talk about the Expectation of a function g of a RV X.

The Expectation of a function g of a RV X with DF F is:

$$E\left(g(X)\right) := \int g(x) \, dF(x) = \begin{cases} \sum_x g(x) f(x) & \text{if } X \text{ is discrete} \\ \int g(x) f(x) \, dx & \text{if } X \text{ is continuous} \end{cases}$$

provided the sum or integral is well-defined. We say the expectation exists if

$$\int \left|g(x)\right| \, dF(x) < \infty \ .$$

When we are looking at the Expectation of X itself, we have $g(x) = x$

Thinking about the Expectations like this, can you see that the familiar Variance of X is in fact the Expection of $g(x) = (x - E(x))^2$?

The **variance** of X (a.k.a. second moment)

Let X be a RV with mean or expectation $E(X)$. The variance of X denoted by $V(X)$ or VX is

$$V(X) := E\left((X - E(X))^2\right) = \int (x - E(X))^2 \, dF(x)$$

provided this expectation exists. The **standard deviation** denoted by $\sigma(X) := \sqrt{V(X)}$.

Thus variance is a measure of ``spread'' of a distribution.

The **k-th moment** of a RV comes from the Expectation of $g(x) = x^k$.

We call

$$E(X^k) = \int x^k \, dF(x)$$

the k-th moment of the RV X and say that the k-th moment exists when $E(|X|^k) < \infty$.

Properties of Expectations

1. If the k-th moment exists and if $j < k$ then the j-th moment exists.
2. If $X_1, X_2, \ldots, X_n$ are RVs and $a_1, a_2, \ldots, a_n$ are constants, then

$$E\left(\sum_{i=1}^{n} a_i X_i\right) = \sum_{i=1}^{n} a_i E(X_i)$$

jsMath

3. Let $X_1, X_2, \dots, X_n$ be independent RVs, then

$$E\left(\prod_{i=1}^{n} X_i\right) = \prod_{i=1}^{n} E(X_i)$$

4. $V(X) = E(X^2) - (E(X))^2$
5. If a and b are constants, then:

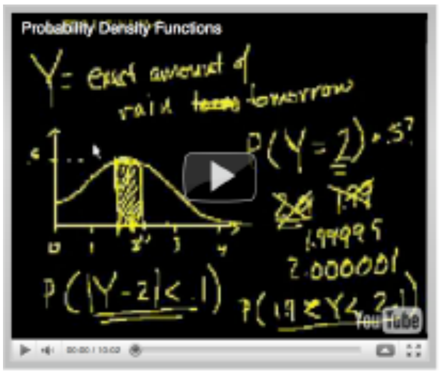
$$V(aX+b) = a^2 V(X)$$

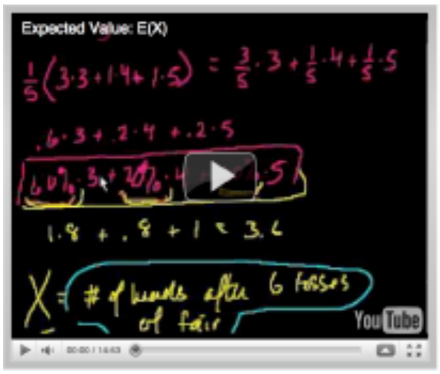
6. If $X_1, X_2, \dots, X_n$ are independent and $a_1, a_2, \dots, a_n$ are constants, then:

$$V\left(\sum_{i=1}^{n} a_i X_i\right) = \sum_{i=1}^{n} a_i^2 V(X_i)$$

You try at home

Please do this at home or with earphones! Watch the Khan Academy movie about probability density functions and expected value if you want to get another angle on the material.


Probability Density Functions
Y= exact amount of rain tomorrow
P(Y=2) ≈ .5?
1.99999
2.000001
P(|Y-2|<.1)
P(1.9≤Y≤2.1)



The population mean and variance of the $Bernoulli(\theta)$ RV

We have already met the discrete $Bernoulli(\theta)$ RV. Remember, that if we have an event A with $P(A) = \theta$, then a $Bernoulli(\theta)$ RV X takes the value 1 if "A occurs" with probability θ and 0 if "A does not occur" with probability $1-\theta$.

In other words, the indicator function $\mathbf{1}_A$ of "A occurs" with probability θ is the $Bernoulli(\theta)$ RV.

For example, flip a fair coin. Consider the event that it turns up heads. Since the coin is fair, the probability of this event θ is $\frac{1}{2}$. If we define an RV X that takes the value 1 if the coin turns up heads ("event coin turns up heads occurs") and 0 otherwise, then we have a $Bernoulli(\theta = \frac{1}{2})$ RV.

We all saw that given a parameter $\theta \in [0,1]$, the probability mass function (PMF) for the $Bernoulli(\theta)$ RV X is:

$$f(x;\theta) = \theta^x (1-\theta)^{1-x} \mathbf{1}_{\{0,1\}}(x) = \begin{cases} \theta & \text{if x=1,} \\ 1-\theta & \text{if x=0,} \\ 0 & \text{otherwise} \end{cases}$$

and its DF is:

jsMath

$$F(x;\theta) = \begin{cases} 1 & \text{if } 1 \leq x, \\ 1-\theta & \text{if } 0 \leq x < 1, \\ 0 & \text{otherwise} \end{cases}$$

Now let's look at some expectations: the **population mean** and **variance** of an RV $X \sim Bernoulli(\theta)$.

Because X is a discrete RV, our expectations use sums rather than integrals.

The first moment or expectation is:

$$\begin{aligned} E(X) &= \sum_{x=0}^{1} x f(x;\theta) \\ &= (0 \times (1-\theta)) + (1 \times \theta) \\ &= 0 + \theta \\ &= \theta \end{aligned}$$

The second moment is:

$$\begin{aligned} E(X^2) &= \sum_{x=0}^{1} x^2 f(x;\theta) \\ &= (0^2 \times (1-\theta)) + (1^2 \times \theta) \\ &= 0 + \theta \\ &= \theta \end{aligned}$$

The variance is:

$$\begin{aligned} V(X) &= E(X^2) - (E(X))^2 \\ &= \theta - \theta^2 \\ &= \theta(1-\theta) \end{aligned}$$

We can see that $E(X)$ and $V(X)$ will vary with the parameter θ. This is why we subscript E and V with θ, to emphasise that the values depend on the parameter.

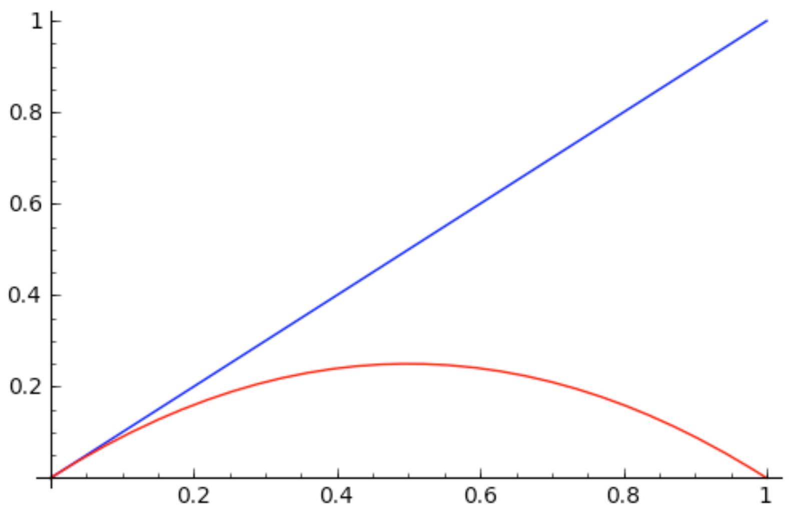
$$E_\theta(X) = \theta$$

$$V_\theta(X) = \theta(1-\theta)$$

We can use Sage to do a simple plot to see how $E_\theta(X)$ and $V_\theta(X)$ vary with θ.

```
def bernoulliPopMean(th):
    '''A function to find the population mean for an RV distributed
Bernoulli(theta).

    parameter th is the distribution parameter theta.'''

    return th

def bernoulliPopVariance(th):
    '''A function to find the population variance for an RV distributed
Bernoulli(theta).

    parameter th is the distribution parameter theta.'''

    return th*(1-th)

p = plot(bernoulliPopMean(x), xmin=0, xmax=1)
p += plot(bernoulliPopVariance(x), xmin=0, xmax=1, rgbcolor="red")
show(p)
```

jsMath



The population mean and variance of the $Uniform(0,1)$ RV

Now let's look at the the **population mean** and **variance** of a continuous RV $X \sim Uniform(0,1)$.

Because X is a continuous RV, our expectations use integrals.

$$\begin{aligned} E(X) &= \int_{x=0}^{1} x f(x)\, dx \\ &= \int_{x=0}^{1} x\, 1\, dx \\ &= \tfrac{1}{2}\left(x^2\right]_{x=0}^{x=1} \\ &= \tfrac{1}{2}(1-0) \\ &= \tfrac{1}{2} \end{aligned}$$

$$\begin{aligned} E(X^2) &= \int_{x=0}^{1} x^2 f(x)\, dx \\ &= \int_{x=0}^{1} x^2\, 1\, dx \\ &= \tfrac{1}{3}\left(x^3\right]_{x=0}^{x=1} \\ &= \tfrac{1}{3}(1-0) \\ &= \tfrac{1}{3} \end{aligned}$$

$$\begin{aligned} V(X) &= E(X^2) - (E(X))^2 \\ &= \tfrac{1}{3} - \left(\tfrac{1}{2}\right)^2 \\ &= \tfrac{1}{3} - \tfrac{1}{4} \\ &= \tfrac{1}{12} \end{aligned}$$

Winnings on Average

Think about playing a game where we draw $x \sim X$ and I pay you $r(x)$ (r is some reward function, a function of x that says what your reward is when x is drawn). Then, your average winnings from the game is the sum (or integral), over all the possible values of x, of $r(x)\times$ the chance that $X = x$.

Put formally, if $Y = r(X)$, then

$$E(Y) = E(r(X)) = \int r(x)\, dF(x)$$

jsMath

Probability is an Expectation

Remember when we first talked about the probability of some event A, we talked about the idea of the probability of A as the long term relative frequency of A?

Now consider some event A and a reward function $r(x) = \mathbf{1}_A(x)$.

Recall that $\mathbf{1}_A(x) = 1$ if $x \in A$ and $\mathbf{1}_A(x) = 0$ if $x \notin A$: the reward is 1 if $x \in A$ and 0 otherwise.

$$\begin{array}{lcl}
\text{If } X \text{ is continuous } E(\mathbf{1}_A(X)) & = & \int \mathbf{1}_A(x)\, dF(x) \\
 & = & \int_A f(x)\, dx \\
 & = & P(X \in A) = P(A) \\
\text{If } X \text{ is discrete } E(\mathbf{1}_A(X)) & = & \mathbf{1}_A(x)\, dF(x) \\
 & = & \sum_{x \in A} f(x) \\
 & = & P(X \in A) = P(A)
\end{array}$$

This says that probability is a special case of expectation: the probability of A is the expectation that A will occur.

Take a $Uniform(0,1)$ RV X. What would you say the probability that an observation of this random variable is ≤ 0.5 is, ie what is $P(X \leq 0.5)$?

Let's use Sage to simulate some observations for us and look at the relative frequency of observations ≤ 0.5:

```
uniform(0,1)
```

```
countObOfInterest = 0    # variable to count observations of interest
numberOfObs = 1000        # variable to control how many observations we
simulate
obOfInterest = 0.5        # variable for observation of interest
for i in range(numberOfObs): # loop to simulate observations
    if uniform(0,1) <= obOfInterest:    # conditional statement to check
observation
        countObOfInterest += 1    # accumulate count of observation of
interest

print "The relative frequency of x <=", obOfInterest.n(digits=2), " was",
```

```
randint(0,1)
```

Or, we could look at a similar simulation for a discrete RV, say a $Bernoulli(\frac{1}{2})$ RV. This could be modelling the event that we get a head when we throw a fair coin. For this we'll use the `randint(0,1)` function to simulate the observed value of our RV X.

jsMath

```
countObOfInterest = 0    # variable to count observations of interest
numberOfObs = 1000        # variable to control how many observations we
simulate
obOfInterest = 1        # variable for observation of interest
for i in range(numberOfObs): # loop to simulate observations
    if randint(0,1) == obOfInterest:    # conditional statement to check
observation
        countObOfInterest += 1    # accumulate count of observation of
interest

print "The relative frequency of x ==", obOfInterest, " was",
```

Another way of thinking about the $Bernoulli(\frac{1}{2})$ RV is that it has a discrete uniform distribution over $\{0,1\}$. It can take on a finite number of values (0 and 1 only) and the probabilities of observing either of these two values are are equal.

The $deMoivre(\frac{1}{k},\frac{1}{k},\ldots,\frac{1}{k})$ RV

We have seen that a $Bernoulli(\theta)$ RV has two outcomes (0, and 1). What if we are interested in modelling situations where there are more than two outcomes of interest? For example, we could use a $Bernoulli(\frac{1}{2})$ RV to model whether the outcome of the flip of a fair coin is a head, but we can't use it for modelling an RV which is the number we get when we toss a six-sided die.

So, now, we will consider a natural generalization of the $Bernoulli(\theta)$ RV with more than two outcomes. This is called the $deMoivre(\frac{1}{k},\frac{1}{k},\ldots,\frac{1}{k})$ RV (after Abraham de Moivre, 1667-1754), one of the first great analytical probabalists). A $deMoivre(\frac{1}{k},\frac{1}{k},\ldots,\frac{1}{k})$ RV X has a discrete uniform distribution over $\{1,2,...,k\}$: there are k possible equally probable ('equiprobable') values that the RV can take.

If we are rolling a die and X is the number on die, then $k=6$.

Or think of the New Zealand Lotto game. There are 40 balls in the machine, numbered $1,2,\ldots,40$. The number on the first ball out of the machine can be modelled as a $deMoivre(\frac{1}{40},\frac{1}{40},\ldots,\frac{1}{40})$ RV.

We say that an RV X is $deMoivre(\frac{1}{k},\frac{1}{k},\ldots,\frac{1}{k})$ distributed if its probability mass function PMF is:

$$f\left(x;\left(\frac{1}{k},\frac{1}{k},\ldots,\frac{1}{k}\right)\right)=\begin{cases}0 & \text{if } x\notin\{1,2,\ldots,k\}\\ \frac{1}{k} & \text{if } x\in\{1,2,\ldots,k\}\end{cases}$$

We can find the expectation:

$$\begin{aligned}E(X) &= \textstyle\sum_{x=1}^{k} xP(X=x)\\ &= (1\times\frac{1}{k})+(2\times\frac{1}{k})+\ldots+(k\times\frac{1}{k})\\ &= (1+2+\ldots+k)\frac{1}{k}\\ &= \frac{k(k+1)}{2}\frac{1}{k}\\ &= \frac{k+1}{2}\,,\end{aligned}$$

the second moment:

$$\begin{aligned}E(X^2) &= \textstyle\sum_{x=1}^{k} x^2P(X=x)\\ &= (1^2\times\frac{1}{k})+(2^2\times\frac{1}{k})+\ldots+(k^2\times\frac{1}{k})\\ &= (1^2+2^2+\ldots+k^2)\frac{1}{k}\\ &= \frac{k(k+1)(2k+1)}{6}\frac{1}{k}\\ &= \frac{2k^2+3k+1}{6}\,,\end{aligned}$$

and finally the variance:

jsMath

$$\begin{array}{rcl} V(X) & = & E(X^2) - (E(X))^2 \\ & = & \frac{2k^2+3k+1}{6} - \left(\frac{k+1}{2}\right)^2 \\ & = & \frac{2k^2+3k+1}{6} - \frac{k^2+2k+1}{4} \\ & = & \frac{4(2k^2+3k+1)-6(k^2+2k+1)}{24} \\ & = & \frac{8k^2+12k+4-6k^2-12k-6}{24} \\ & = & \frac{2k^2-2}{24} \\ & = & \frac{k^2-1}{12}. \end{array}$$

We coud use the Sage `randint` function to simulate the number on the first ball in a Lotto draw:

```
randint(1,40)
```

Product experiments

So far, we have talked about having one RV, or one observation of an RV. Often, our experiments involve a number of separate observations.

Say we flip two fair coins. We assume that the two flips are independent, and that for each coin we can define an RV which takes the value 1 if that coin lands heads up, 0 otherwise. We will label the RV for one coin as X_1 and the RV for the other coin as X_2

What we have is 2 **independent and identically distributed** (IID) $Bernoulli(\frac{1}{2})$ RVs X_1, X_2.

If we are interested in the results of these separate coin tosses, we can think of a **random vector** or $X = (X_1, X_2)$. There are four possible results of this product experiment: $(0, 0)$ if both coins land tails, $(1, 0)$ if the one associated with X_1 lands heads but the other tails, $(0, 1)$ if the one associated with X_2 lands heads but the other tails, and $(1, 1)$ if both land heads.

Because the two RVs are independent,

$P((X_1, X_2) = (0, 0)) = P(X_1 = 0)P(X_2 = 0) = \frac{1}{2}\frac{1}{2} = \frac{1}{4}$

$P((X_1, X_2) = (1, 0)) = P(X_1 = 1)P(X_2 = 0) = \frac{1}{2}\frac{1}{2} = \frac{1}{4}$

$P((X_1, X_2) = (0, 1)) = P(X_1 = 0)P(X_2 = 1) = \frac{1}{2}\frac{1}{2} = \frac{1}{4}$

$P((X_1, X_2) = (1, 1)) = P(X_1 = 1)P(X_2 = 1) = \frac{1}{2}\frac{1}{2} = \frac{1}{4}$

Similarly, we could have three IID RVs, or four, or five .....

In general, with n IID RVs we have a random vector $X = (X_1, X_2, \ldots, X_n)$

Example 1: New Zealand Lotto and n IID $deMoirve$ RVs

We showed that for New Zealand lotto (40 balls in the machine, numbered $1, 2, \ldots, 40$), the number on the first ball out of the machine can be modelled as a $deMoivre(\frac{1}{40}, \frac{1}{40}, \ldots, \frac{1}{40})$ RV.

There have been lots of Lotto draws. If we took the number on the first ball in n draws we would have

$X_1, X_2, \ldots, X_n \overset{IID}{\sim} deMoivre(\frac{1}{40}, \frac{1}{40}, \ldots, \frac{1}{40})$

As it happens, we have the New Zealand Lotto results from 1 August 1987 to 10 November 2008.

jsMath

This data was retrieved from the NZ lotto web site: http://lotto.nzpages.co.nz/previousresults.html

We have made a (hidden) function that enables us to get the ball one data in a list. Evaluate the cell below to get and show the data.

```
listBallOne = getLottoBallOneData()
listBallOne
```

Check how many values we have (this is n).

```
len(listBallOne)
```

```

```

Statistics

The official NZ Government site for statistics about New Zealand is http://www.stats.govt.nz/. Let's take a tour through it now!

Data and statistics

In general, given some probability triple $(\Omega, \mathcal{F}, P)$, let the function X measure the outcome ω from the sample space Ω.

$$X(\omega) : \Omega \rightarrow \mathbf{X}$$

X is called **data**.

$\mathbf{X}$ is called the data space (sample space of the data X).

$X(\omega) = x$ is the outcome ω measured by X and is called the observed data or the realisation of X.

Often the measurements made by X are real numbers or vectors of real numbers.

This is the link to the definition of a random variable we have already seen (a random variable X as a function or map from the sample space to the real line $\mathbb{R}$). We have also seen that X can in fact be a random vector $X = (X_1, X_2, \ldots, X_n)$, i.e. a vector of random variables.

When we talked about an experiment involving two IID $Bernoulli(\frac{1}{2})$ RVs above (tossing two coins), we listed the different results we might get as (0,0), (1,0), (0,1), (1,1).

Say we observe the outcome ω = (H, H) (two heads). Our $Bernoulli$ random vector X measures this as (1,1). Thus, (1,1) is the observed data or the realisation of X.

So what is a statistic?

A **statistic** is any measureable function of the data: $T(x) : \mathbf{X} \rightarrow \mathbf{T}$.

Thus, a statistic T is also an RV that takes values in the space $\mathbf{T}$.

When $x \in \mathbf{X}$ is the observed data, $T(x) = t$ is the observed statistic of the observed data x.

Example 2: New Zealand Lotto and 1114 IID $deMoirve$ RVs

Let's look again at our New Zealand lotto data.

```
listBallOne = getLottoBallOneData()
len(listBallOne)
```

jsMath

We can think of this list as x, the realisation of a random vector $X = (X_1, X_2, \ldots, X_{1114})$ where $X_1, X_2, \ldots, X_{1114} \overset{IID}{\sim} deMoivre(\frac{1}{40}, \frac{1}{40}, \ldots, \frac{1}{40})$

The data space is every possible sequence of ball numbers that we could have got in these 1114 draws. $\mathbf{X} = \{1, 2, \ldots, 40\}^{1114}$. There are $40^1 114$ possible sequences and

$P(X_1, X_2, \ldots, X_{1114}) = (x_1, x_2, \ldots, x_{1114}) = \frac{1}{40} \times \frac{1}{40} \times \ldots \times \frac{1}{40} = (\frac{1}{40})^{1114}$ if $(x_1, x_2, \ldots, x_{1114}) \in \mathbf{X} = \{1, 2, \ldots, 40\}^{1114}$

Our data is just one of the 40^{1114} possible points in this data space.

Some statistics

Sample mean

From a given sequence of RVs $X_1, X_2, \ldots, X_n$, or a random vector $X = (X_1, X_2, \ldots, X_n)$, we can obtain another RV called the **sample mean** (technically, the n-sample mean):

$$T_n((X_1, X_2, \ldots, X_n)) = \bar{X}_n((X_1, X_2, \ldots, X_n)) := \frac{1}{n}\sum_{i=1}^{n} X_i$$

We write $\bar{X}_n((X_1, X_2, \ldots, X_n))$ as $\bar{X}$,

and its realisation $\bar{X}_n((x_1, x_2, \ldots, x_n))$ as $\bar{x_n}$.

By the properties of expectations that we have seen before,

$$E(\bar{X}_n) = E\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \frac{1}{n}E\left(\sum_{i=1}^{n} X_i\right) = \frac{1}{n}\sum_{i=1}^{n} E(X_i)$$

And because every X_i is identically distributed with the same expectation, say $E(X_1)$, then

$$E(\bar{X}_n) = \frac{1}{n}\sum_{i=1}^{n} E(X_i) = \frac{1}{n} \times n \times E(X_1) = E(X_1)$$

Similarly, we can show that,

$$V(\bar{X}_n) = V\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \frac{1}{n^2}V\left(\sum_{i=1}^{n} X_i\right)$$

And because every X_i is independently and identically distributed with the same variance, say $V(X_1)$, then

$$V(\bar{X}_n) = \frac{1}{n^2}V\left(\sum_{i=1}^{n} X_i\right) = \frac{1}{n^2} \times n \times V(X_1) = \frac{1}{n}V(X_1)$$

Sample variance

Sample variance is given by

$$\frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2$$

Sometimes, we divide by $n-1$ instead of n. It is a measure of spread from the sample.

Similarly, we can look at a **sample standard deviation** = $\sqrt{\text{sample variance}}$

jsMath

Example 3: New Zealand Lotto and 1114 IID $deMoirve$ RVs

Sage has a nice module called pylab which provides some useful statistical capabilities that we can use on our Lotto data. To be able to use these capabilities, it is easiest to convert our Lotto Data into a type called a `pylab.array`. (You will be looking more closely at pylab and arrays in the You Try sections below).

```
from pylab import array, mean, var, std # make pylab stuff we need
accessible in Sage
listBallOne = getLottoBallOneData()    # make sure we have our list
arrayBallOne = pylab.array(listBallOne) # make the array out of the list
arrayBallOne     # disclose the list
```

Now we can get the some sample statistics for the lotto ball one data.

Sample mean

```
arrayBallOne.mean()        # sample mean statistic; same as
pylab.mean(ArrayDataBallOne)
```

Sample variance

```
arrayBallOne.var()         # sample variance statistic; same as
pylab.var(ArrayDataBallOne)
```

Sample standard deviation

```
arrayBallOne.std()          # sample standard deviation statistic; same as
pylab.std(ArrayDataBallOne)
```

Double check that the sample standard deviation is indeed the square root of the sample variance

```
sqrt(arrayBallOne.var())   # just checking
```

Order statistics

The kth order statistic of a sample is the kth smallest value in the sample. Order statistics that may be of particular interest include the smallest (minimum) and largest (maximum) values in the sample.

```
arrayBallOne.min()         # sample minimum statistic
```

```
arrayBallOne.max()         # sample maximum statistic
```

```
arrayBallOneSorted = pylab.array(listBallOne) # make the array out of the
list for sorting next
arrayBallOneSorted.sort()        # sort the array to get the order statistic
```

jsMath

```
arrayBallOneSorted             # the original data array
```

```
arrayBallOne                   # the sorted data array
```

Frequencies

With lots of discrete data like this, we may be interested in the frequency of each value, or the number of times we get a particular value. This can be formalized as the following process of adding indicator functions of the data points $(X_1, X_2, \ldots, X_n)$:

$$\sum_{i=1}^{n} \mathbf{1}_{\{X_i\}}(x)$$

We can use the Sage dictionary to give us a mapping from ball numbers in the list to the count of the number of times each ball number comes up.

Although we are doing this with the Lotto data, mapping a list like this seems like a generally useful thing to be able to do, and so we write it as a function and then use the function on our Lotto ball one data:

```
%auto
def makeFreqDict(myDataList):
    '''Make a frequency mapping out of a list of data.

    Param myDataList, a list of data.
    Return a dictionary mapping each unique data value to its frequency
count.'''

    freqDict = {} # start with an empty dictionary

    for res in myDataList:

        if res in freqDict: # the data value already exists as a key
                freqDict[res] = freqDict[res] + 1 # add 1 to the count
        else: # the data value does not exist as a key value
            freqDict[res] = 1 # add a new key-value pair for this new data
value, frequency 1

    return freqDict # return the dictionary created
```

The kth order statistic of a sample is the kth smallest value in the sample. Order statistics that may be of particular interest include the smallest (minimum) and largest (maximum) values in the sample.

```
listBallOne = getLottoBallOneData()    # make sure we have our list
ballOneFreqs = makeFreqDict(listBallOne) # call the function
```

Thus, balls labelled by the number 1 come up 29 times in the first ball drawn (Ball One) out of 1,114 Lotto trials. Similarly, the number 2 comes up 28 times, ... 40 comes up 37 times. Of course, these numbers would be different if you have downloaded a more recent data file with additional trials!

So what? Well, we'd hope that the lotto is fair, i.e. that the probability of each ball coming up with any of the available numbers is the same for each number: the probability that Ball One is a 1 is the same as the probability that it is 2, is the same as the probability that it is 3,..... , is the same as the probability that it is 40. If the lotto is fair, the number that

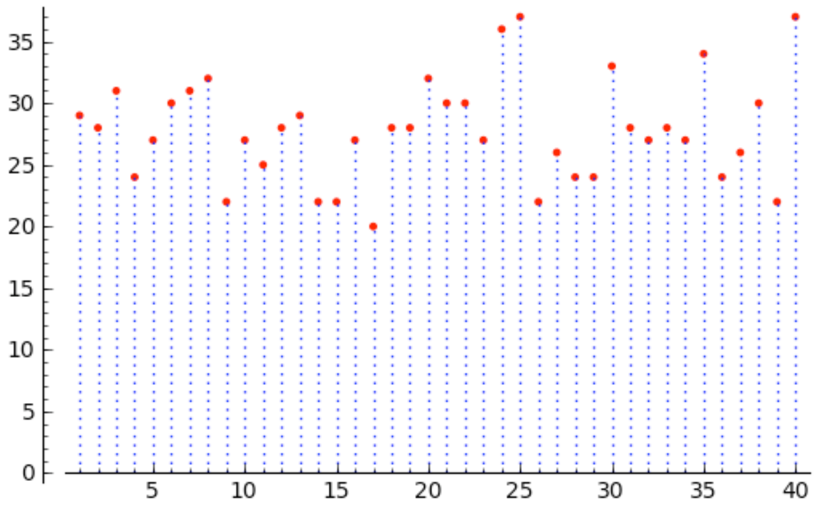
jsMath

comes up on each ball should be a discrete uniform random variable. More formally, we would expect it to be the $deMoivre(\frac{1}{40}, \frac{1}{40}, \ldots, \frac{1}{40})$ RV as lectured. Over the long term, we'd expect the number of times each number comes up on a given trial to be about the same as the number of times any other number comes up on that trial.

We have data from 1987 to 2008, and a first step to assessing the fairness of the lotto (for Ball One, anyway) could be to just visualise the data. We can use the list of points we created above and the SAGE plotting function `points` to plot a simple graphic like this.

Here we are plotting the frequency with which each number comes up against the numbers themselves, but it is a bit hard to see what number on the ball each red point relates to. To deal with this we add dotted lines going up from the number on the horizontal axis to the corresponding (number, frequency) tuple plotted as a red point.

```
lottoPlotCounts = points(ballOneFreqs.items(), rgbcolor="red")
for k in ballOneFreqs.keys():
    lottoPlotCounts += line([(k, 0),(k, ballOneFreqs[k])], rgbcolor="blue",
linestyle=":")
show(lottoPlotCounts)
```



Empirical mass function

What about plotting the height of a point as the *relative frequency* rather than the frequency? The relative frequency for a number is the count (frequency) for that number divided by the sample size, i.e., the total number of trials. This can be formalized as the following process of adding normalized indicator functions of the data points $(X_1, X_2, \ldots, X_n)$:

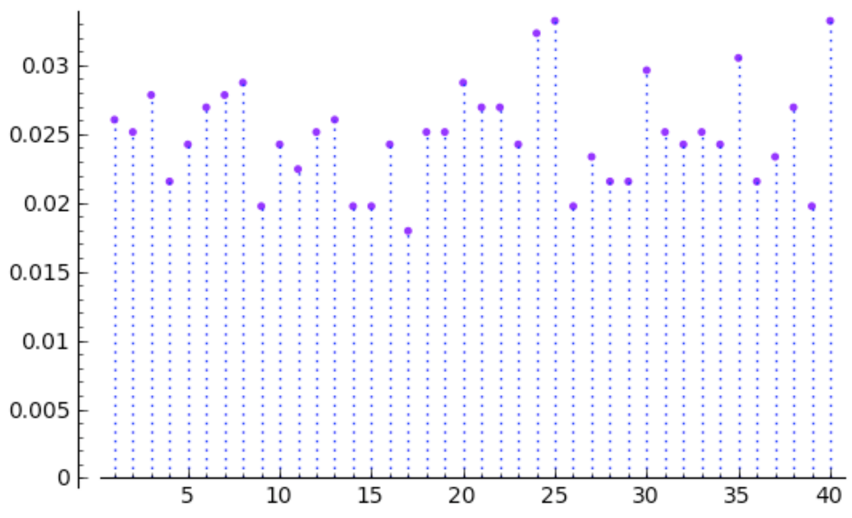
$$\frac{1}{n}\sum_{i=1}^{n}\mathbf{1}_{\{X_i\}}(x)$$

The You Try section below will show you some techniques we use in doing this. For now we have bundled them up into a hidden function called makeEMF so that you can just concentrate on the data for now.

Now, we plot the points based on relative frequencies. What we have made is another statistic called the **empirical mass function**.

```
listBallOne = getLottoBallOneData()     # make sure we have our list

numRelFreqPairs = makeEMF(listBallOne) # make a list of unique data values
and their relative frequencies
lottoPlotEMF = point(numRelFreqPairs, rgbcolor = "purple")
for k in numRelFreqPairs:    # for each tuple in the list
    kkey, kheight = k     # unpack tuple
    lottoPlotEMF += line([(kkey, 0),(kkey, kheight)], rgbcolor="blue",
linestyle=":")
```

jsMath



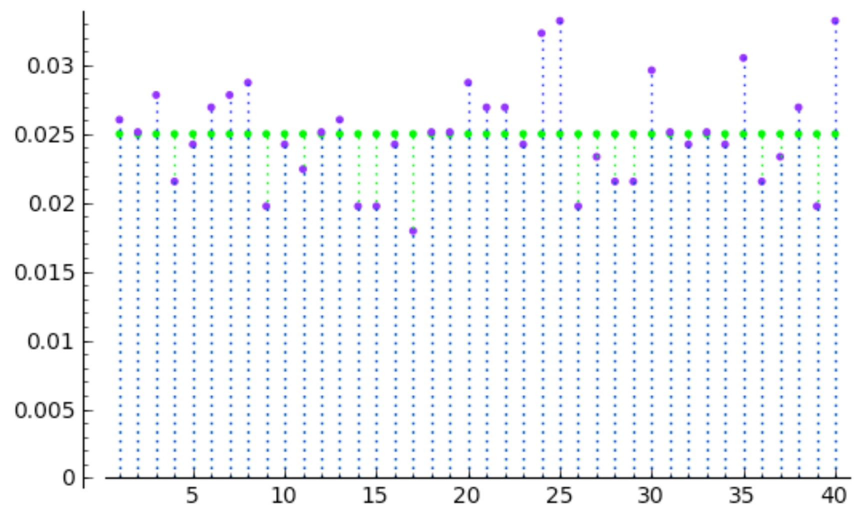
Let us plot the probability mass function (PMF) of the hypothesized probability model for a fair NZ Lotto, i.e., the PMF of the RV X ~ de Moivre (1/40,1/40, ... , 1/40) and overlay it on the empirical mass function of the observed data. We will meet list comprehension properly in a future worksheet.

```
# list comprehension by the constant 1/40 for f(x)=1/40, x=1,2,...,40
ballOneEqualProbs = [1/40 for x in range(1,41,1)]
ballOneEqualProbs
```

```
# make a list of (x,f(x)) tuples for the PDF using zip
numEqualProbsPairs = zip(ballOneFreqs.keys(), ballOneEqualProbs)
numEqualProbsPairs # disclose the list of tuple
```

```
listBallOne = getLottoBallOneData()    # make sure we have our list
numRelFreqPairs = makeEMF(listBallOne) # make a list of unique data values
and their relative frequencies

# make the EMF plot
lottoPlotEMF = point(numRelFreqPairs, rgbcolor = "purple")
for k in numRelFreqPairs:    # for each tuple in the list
    kkey, kheight = k     # unpack tuple
    lottoPlotEMF += line([(kkey, 0),(kkey, kheight)], rgbcolor="blue",
linestyle=":")

ballOneEqualProbs = [1/40 for x in range(1,41,1)]    # make sure we have the
equal probabilities
numEqualProbsPairs = zip(ballOneFreqs.keys(), ballOneEqualProbs) # and the
pairs

# make a plot of the equal probability pairs
equiProbabledeMoivre40PMF = point(numEqualProbsPairs, rgbcolor = "green")
for e in numEqualProbsPairs:    # for each tuple in list
```

jsMath



Empirical distribution function

Another extremely important statistics of the observed data is called the **empirical distribution function** (EDF). The EDF is the empirical or data-based distribution function (DF) just like the empirical mass function (EMF) is the empirical or data-based probability mass function (PMF). This can be formalized as the following process of adding indicator functions of the half-lines beginning at the data points $[X_1,+\infty),[X_2,+\infty),\ldots,[X_n,+\infty)$:

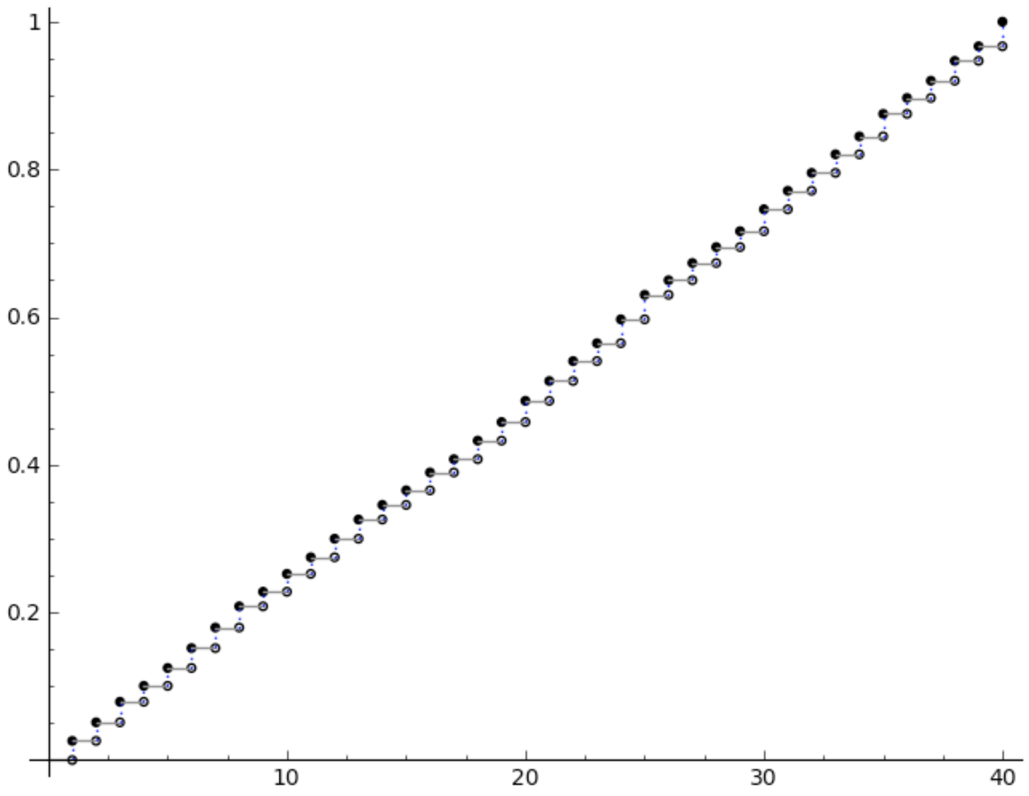
$$\widehat{F}_n(x)=\frac{1}{n}\sum_{i=1}^{n}\mathbf{1}_{[X_i,+\infty)}(x)$$

We have bundled up the techniques we use in a hidden function called makeEDF so that you can just concentrate on the data for now.

```
listBallOne = getLottoBallOneData()    # make sure we have our list
numCumFreqPairs = makeEDF(listBallOne)
lottoPlotEDF = points(numCumFreqPairs, rgbcolor = "black", faceted = true)
for k in range(len(numCumFreqPairs)):
    x, kheight = numCumFreqPairs[k]     # unpack tuple
    previous_x = 0
    previous_height = 0
    if k > 0:
        previous_x, previous_height = numCumFreqPairs[k-1] # unpack previous
tuple
    lottoPlotEDF += line([(previous_x, previous_height),(x,
previous_height)], rgbcolor="grey")
    lottoPlotEDF += points((x, previous_height),rgbcolor = "white", faceted
```

evaluate

jsMath



You try

Arrays

We have already talked about lists in SAGE. Lists are great but are also general: lists are designed to be able to cope with any sort of data but that also means they don't have some of the specific functionality we might like to be able to analyse numerical data sets. **Arrays** provide a way of representing and manipulating numerical data known an a matrix in maths lingo. Matrices are particularly useful for computational statistics.

The lists we have been dealing with were one-dimensional. An array and the matrix it represents in a computer can be multidimensional. The most common forms of array that we will meet are one-dimensional and two-dimensional. You can imagine a two-dimensional array as a grid, which has rows and columns. Take for instance a 3-by-6 array (i.e., 3 rows and 6 columns).

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

The array type is not available in the basic SAGE package, so we *import* a *library package* called PyLab which includes arrays. Make sure that you evaluate the next cell to import pylab into your worksheet. The capital letters in PyLab don't matter when we import. (The optional section below discusses importing library modules in a little more detail.)

```
import pylab          # this brings into our SAGE worksheet the capabilities
of the pylab library module
print 'pylab module imported'
```

Because the arrays are part of PyLab, not the basic Sage package, we don't just say array, but instead *qualify* arra jsMath the name of the module it is in. So, we say **`pylab.array`**.

The cell below makes a two-dimensional 3-by-6 (3 rows by 6 columns) array by specifying the rows, and each element in each row, individually.

```
array1 = pylab.array([[0,1,2,3,4,5],[6,7,8,9,10,11],[12,13,14,15,16,17]]) #
make an array the hard way
array1
```

We can use the array's **`shape`** method to find out its shape. The `shape` method for a two-dimensional array returns an *ordered pair* in the form (rows,columns). In this case it tells us that we have 3 rows and 6 columns.

```
array1.shape  # find out the shape of array1
```

Another way to make an array is to start with a one-dimensional array and *resize* it to give the shape we want, as we do in the two cells below.

We start by making a one-dimensional array using the **`range`** function we have met in previous labs.

```
array2 = pylab.array(range(18))     # make an one dimensional array
array2
```

The array's **`resize`** allows us to specify a new shape for the same array. We are going to make a two-dimensional array with 3 rows and 6 columns. Note that the shape you specify must be compatible with the number of elements in the array. In this case, array2 has 18 elements and we specify a new shape of 3 rows and 6 columns, which still makes 18 = 3*6 elements. If the new shape you specify won't work with the array, you'll get an error message.

```
array2.resize(3,6)      # use the array's resize method to change the shape
of the array 18=3*6
array2
```

The **`range`** function will only give us ranges of integers. The PyLab module includes a more flexible function **`arange`** (that's arange with one r -- think of it as something like **a**-for-array-**range** -- not 'arrange' with two r's). The **`arange`** function takes parameters specifying the start, stop and step values (similar to `range`), but is not restricted to integers and returns a one-dimensional array rather than a list.

Here we use **`arange`** to make an array of numbers from 0.0 (start_argument) going up in steps of 0.1 (step_argument) to 0.18 - 0.1 and just as with `range`, the last number we get in the array is stop_argument - step_argument = 0.18 - 0.01 = 0.17.

If you check the type for array3, you might be surprised to find that it is a `numpy.ndarray`. What happened to `pylab.array`? The PyLab module is actually made up of a number of other useful modules, one of which is NumPy. NumPy is an important Python language library for scientific computing (think 'Numerical' and 'Python' if you are wondering about the name). Python is the language underlying Sage. The array we are using is the NumPy N-dimensional array, or `numpy.ndarray`.

```
array3 = pylab.arange(0.0,0.18,0.01)   # the pylab.arange(start, stop, step)
array3
```

jsMath

If you check the type for array3, you might be surprised to find that it is a `numpy.ndarray`. What happened to `pylab.array`? The PyLab module is actually made up of a number of other useful modules, one of which is NumPy. NumPy is an important Python language library for scientific computing (think 'Numerical' and 'Python' if you are wondering about the name). Python is the language underlying SAGE. The array we are using is the NumPy N-dimensional array, or `numpy.ndarray`.

```
type(array3)
```

Check the shape of array3.

```
array3.shape
```

The `resize` method resizes the array it is applied to. The **`reshape`** method will leave the original array unchanged but return a new array, based on the original one, of the required new shape.

```
array3.resize(9,2)                       # which we can resize into a 9 by 2
array
```

```
array3.shape                              # try to see the shape of array3 now
```

```
array4.shape
```

The `resize` method resizes the array it is applied to. The `reshape` method will leave the original array unchanged but return a new array, based on the original one, of the required new shape.

```
array4 = array3.reshape(6,3)     # reshape makes a new array of the
specified size 6 by 3
array4
```

```
array3.shape
```

```
array4.shape
```

When we imported the Lotto data from a data file (in the bit of code we did not show you), we brought it in as an array, like this. You do not have to worry about the `getData` function. It is just a simple function we wrote to get data from a file.

```
myFilename = 'main_nzlotto1987_2008.csv'
lottoData = getData(myFilename,headerlines=1,sep=',') # get data from
myFilename, one headerline, csv
lottoData
```

jsMath

The array contains all the data (not just ball one). You can see the draw numbers, dates, and numbers on the balls in the summarised look at the array above.

Arrays: indexing, slicing and copying

Remember indexing into lists to find the elements at particular positions? We can do this with arrays, but we need to specify the index we want for each dimension of the array. For our two-dimensional arrays, we need to specify the row and column index, i.e. use a format like [row_index,column_index]. For exampe, `[0,0]` gives us the element in the first column of the first row (as with lists, array indices start from 0 for the first element).

```
array4 = pylab.arange(0.0,0.18,0.01)   # make sure we have array4
array4.resize(6,3)
```

```
array4
```

`[5,2]` gives us the element in the third column (index 2) of the sixth row (index 5).

```
array4[5,2]
```

We can use the colon to specify a range of columns or rows. For example, `[0:4,0]` gives us the elements in the first column (index 0) of rows with indices from 0 through 3. Note that the row index range `0:4` gives rows with indices starting from index 0 and ending at index 3=4-1, i.e., indices 0 through 3.

```
array4[0:4,0]
```

Similarly we could get all the elements in the second column (column index 1) of rows with indices 2 through 4.

```
array4[2:5,1]
```

The colon on its own gives everything, so a column index of `:` gives all the columns. Thus we can get, for example, all elements of a particular row -- in this case, the third row.

```
array4[2,:]
```

Or all the elements of a specified column, in this case the first column.

```
array4[:,0]
```

Or all the elements of a range of rows (think of this as like slicing the array horizontally to obtain row indices 1 through

jsMath

4=5-1).

```
array4[1:5,:]
```

Or all the elements of a range of columns (think of this as slicing the array vertically to obtain column indices 1 through 2).

```
array4[:,1:3]
```

```
array4[2:5,0:2]      # naturally, we can slice both horizontally and
vertically to obtain row indices 2 through 4 and column indices 0 through 1
```

Finally, `[:]` gives a copy of the whole array. This copy of the original array can be assigned to a new name for furter manipulation without affecting the original array.

```
CopyOfArray4 = array4[:]          # assign a copy of array4 to the new array
named CopyOfArray4
CopyOfArray4                      # disclose CopyOfArray4
```

```
CopyOfArray4.resize(9,2) # resize CopyOfArray4 from a 6-by-3 array to a
9-by-2 array
CopyOfArray4             # disclose CopyOfArray4 as it is now
```

```
array4               # note that our original array4 has remained unchanged
with size 6-by-3
```

We used the slicing technique to get the ball one data out of our big array of lotto data. You may have noticed that the data in the array was all strings. We therefore had to convert from a string to a number. In this case we used the ZZ function to convert to Sage.Rings.Integers. This was a bit rough-and-ready and would not work well if our data had had contained missing values or invalid values (non-numerical values where we expected numerical values), but in the case the data was good quality and it worked.

```
myFilename = 'main_nzlotto1987_2008.csv'
lottoData = getData(myFilename,headerlines=1,sep=',') # get data from
myFilename, one headerline, csv
listDataBallOne = [ZZ(x) for x in lottoData[:,2]] # convert to a list of
SageRingsIntegers
listDataBallOne
```

Useful arrays

PyLab (well, really NumPy) provides quick ways of making some useful kinds of arrays. Some of these are shown in the cells below.

An array of zeros of a particular shape. Here we ask for shape (2,3), i.e., 2 rows and 3 columns.

```
arrayOfZeros = pylab.zeros((2,3)) # get a 2-by-3 array of zeros
arrayOfZeros
```

jsMath

An array of ones of a particular shape.   Here we ask for shape (2,3), i.e., 2 rows and 3 columns.

```
arrayOfOnes = pylab.ones((2,3)) # get a 2-by-3 array of ones
arrayOfOnes
```

An array for the identity matrix, i.e., square (same number of elements on each dimension) matrix with 1's along the diagonal and 0's everywhere else.

```
iden = pylab.identity(3) # get a 3-by-3 identity matrix with 1's along the
diagonal and 0's elsewhere
iden
```

Tuples

In the labs we have done so far, you have seen a Sage type that we have not talked about directly -- a **tuple**.

A tuple is another of the *sequence* types (like lists and sets and strings). The values in a tuple are enclosed in curved parentheses ( ) and the values are separated by commas.

Remember when we resized and reshaped arrays?   You've already used tuples when you told Sage what shape to make your array.   The answers Sage gave you when you asked about shape were also tuples.

```
import pylab
myArray = pylab.arange( 3.0, 4.0, 0.1).reshape(2, 5)
myArray.shape
```

Most often, we want to specify our own tuples.

```
myTuple1 = (1, 2)
myTuple1
```

```
type(myTuple1)
```

```
myTuple2 = (10, 11, 13)
myTuple2
```

Tuples are *immutable*.   In programming, an 'immutable' object is an object whose state cannot be modified after it has been created (Etymology: 'mutable comes from the Latin verb mutare, or 'to change' -- the same root we get 'mutate' from. So in-mutable, or immutable, means not capable of or susceptible to change).   This means that although we can access the element at a particular position in a tuple by indexing ...

```
myTuple1[0] # disclose what is in the first position in the tuple
```

... we can't change what is in that particular position in the tuple.

jsMath

```
myTuple1[0] = 10     # try to assign a different value to the first position
in the tuple
```

```
myTuple1[0]    # the first element in the tuple is immutably 1
```

Useful things you can do with tuples

Sage has a very useful `zip` function.   Zip can be used to 'zip' sequences together, making a list of tuples out of the values at corresponding index positions in each list. Consider a simple example: Note that in general `zip` works with sequences, so it can be used to zip tuples as well as lists.

```
zip(['x', 'y', 'z'], [1, 2, 3])
```

```
zip(('x', 'y', 'z'), (1, 2, 3))
```

This gives us a quick way to make a dictionary if we have separate lists or tuples which contain our keys and values. Note that the ordering in the key and value sequences has to be consistent   -- the first key will be mapped to the first value, etc., etc.

```
myKeys = ('x', 'y', 'z')
myValues = (1, 2, 3)
simpleDict = dict(zip(myKeys, myValues))
simpleDict
```

There are also some useful methods of dictionaries that allow you to 'dissect' the dictionary and extract just the keys or just the values.

```
simpleDict.keys()
```

```
simpleDict.values()
```

And there is also an `items()` method that gives you back your your (key, value) pairs again. You will see that is it a list of **tuples**.

```
simpleDict.items()
```

When we were exploring the lotto data, we made a dictionary that mapped the different possible balls labelled by numbers from 1 through 40 (the keys in the dictionary) to the count of how many times that number ball came up as the the first ball over 1114 draws.   When we plotted the counts against the frequencies, we used the `items()` method to get a list of the key value pairs, as tuples, so that we could plot it.

You'll recognise that what we have here is a list of tuples.   The tuples are the key-value pairs from the dictionary: The number 1 came up 29 times, giving tuple (1, 29); 2 came up 28 times, giving tuple (2, 28) etc., etc.

jsMath

```
listBallOne = getLottoBallOneData()    # make sure we have our list
ballOneFreqs = makeFreqDict(listBallOne) # call the function to make the
dictionary
lottoPairs = ballOneFreqs.items()
lottoPairs
```

You have seen the plot before.

```
listBallOne = getLottoBallOneData()        # make sure we have our list
ballOneFreqs = makeFreqDict(listBallOne)   # call the function to make the
dictionary

lottoPlotCounts = points(ballOneFreqs.items(), rgbcolor="red")
for k in ballOneFreqs.keys():
    lottoPlotCounts += line([(k, 0),(k, ballOneFreqs[k])], rgbcolor="blue",
linestyle=":")
```

Useful functions for sequences

Now we are going to demonstrate some useful methods of sequences. A list is a sequence, and so is a tuple. There are differences between them (tuples are immutable, for instance) but in many cases we can use the same methods on them because they are both sequences. We will demonstrate with a list and a tuple containing the same data values. The data could be the results of three IID $Bernoulli$ trials..

```
obsDataList = [0, 1, 1]
obsDataTuple = tuple(obsDataList)
obsDataTuple
```

```
obsDataList
```

A useful operation we can perform on a tuple is to count the number of times a particular element, say 0 or 1 in our `ObsDataTuple`, occurs. This is a **statistic** of our data called the **sample frequency** of the element of interest.

```
obsDataTuple.count(0)        # frequency of 0 using the tuple
```

```
obsDataList.count(1)        # frequency of 1 using the list
```

Another useful operation we can perform on a tuple is to sum all elements in our **`obsDataTuple`** or further scale the sum by the sample size. These are also **statistics** of our data called the **sample sum** and the **sample mean**, respectively.   Try the same things with **`obsDataList`**.

jsMath

```
sum(obsDataTuple)            # sample sum
```

```
sum(obsDataTuple)/3          # sample mean
```

```
obsDataTuple.count(1) / 3    # alternative expression for sample mean in IID
Bernoulli trials
```

We used a lot of the techniques we have seen so far when we wanted to get the relative frequency associated with each ball in the lotto data.

```
listBallOne = getLottoBallOneData()        # make sure we have our list
ballOneFreqs = makeFreqDict(listBallOne)   # call the function to make the
dictionary
totalCounts = sum(ballOneFreqs.values())
relFreqs = []
for k in ballOneFreqs.keys():
    relFreqs.append(k/totalCounts)
numRelFreqPairs = zip(ballOneFreqs.keys(), relFreqs) # zip the keys and
relative frequencies together
numRelFreqPairs
```

If you are interested and have time you can explore more information about tuples and sequences in general.

Optional exploration

Working through this section of the lab will give you a bit more insight and more programming skills but you should only start it when you are happy with the essential material above.

More on tuples and sequences in general

You can make tuples out almost any object.   It's as easy as putting the things you want in ( ) parentheses and separating them by commas.   Actually, you don't even need the ( ).   If you present Sage (Python) with a sequence of object separated by commas, the default result is a tuple.

```
myTuple2 = 60, 400    # assign a tuple to the variable named myTuple2
type(myTuple2)
```

jsMath

A statement like the one in the cell above is known as *tuple packing* (more generally, *sequence packing*).

When we work with tuples we also often use the opposite of packing - Python's very useful *sequence unpacking* capability.   This is, literally, taking a sequence and unpacking or extracting its elements.

```
x, y = myTuple2     # tuple unpacking
print x
print y
x * y
```

The statement below is an example of *multiple assignment*, which you can see now is really just a combination of tuple packing followed by unpacking.

```
x, y = 600.0, 4000.0
print x
print y
x/y
```

Try a for loop with a tuple -- it will work just like the for loop with a list.

There are a couple more things to note with tuples.   First, we can make an empty tuple like this:

```
empty = ()
len(empty)
```

Secondly, if we want a tuple with only one element we have to use a trailing comma.   If you think about it from the computer's point of view, if it can't see the trailing comma, how is it to tell that you want a tuple not just a single number?

```
singleton = (2.25,)
type(singleton)
```

Provided we use the trailing comma, we don't have to use the ( ) parentheses, just as when we created longer tuples above.

```
singleton = 2.25,
print "singleton is a ", type(singleton), "of length", len(singleton)
```

```
notASingletonTuple = (2.25)
type(notASingletonTuple)
```

jsMath

We have taken the empty and singleton tuples exercises above from http://docs.python.org/tutorial/datastructures.html. This is a very useful site to look for user-friendly information about Python in general (it can't help you with anything that Sage overlays on top of Python though, or with many of the specialised modules that you may want to import into your Sage notebook).

Mutable and immutable sequences

When we talked about tuples, we said that tuples are immutable sequences. Lists, on the other hand, are mutable. Try making yourself a list and then using indexing to change the value at a particular position in the list.

You'll remember that we can't do this with tuples. We can access elements but we can't change them. Tuples are immutable. Try it below with a tuple if you want to remind yourself.

Strings are also immutable sequences. Have a look at the cells below.

```
str = 'mystring'
str[0]
```

```
str[0] = 'b'
```

```
%hide
```

```
%hide
```

```
%hide
```

```
%hide
```

jsMath