Inferential Statistics Ia - Frequentism

Learning objectives

Welcome to the first Frequentist inference mini-project! Over the course of working on this mini-project and the next frequentist mini-project, you'll learn the fundamental concepts associated with frequentist inference. The following list includes the topics you will become familiar with as you work through these two mini-projects:

- the *z*-statistic
- the *t*-statistic
- the difference and relationship between the two
- the Central Limit Theorem, including its assumptions and consequences
- how to estimate the population mean and standard deviation from a sample
- the concept of a sampling distribution of a test statistic, particularly for the mean
- how to combine these concepts to calculate a confidence interval

Prerequisites

For working through this notebook, you are expected to have a very basic understanding of:

- what a random variable is
- what a probability density function (pdf) is
- what the cumulative density function is
- a high-level sense of what the Normal distribution

If these concepts are new to you, please take a few moments to Google these topics in order to get a sense of what they are and how you might use them.

While it's great if you have previous knowledge about sampling distributions, this assignment will introduce the concept and set you up to practice working using sampling distributions. This notebook was designed to bridge the gap between having a basic understanding of probability and random variables and being able to apply these concepts in Python. The second frequentist inference mini-project focuses on a real-world application of this type of inference to give you further practice using these concepts.

For this notebook, we will use data sampled from a known normal distribution. This allows us to compare our results with theoretical expectations.

I An introduction to sampling from the Normal distribution

First, let's explore the ways we can generate the Normal distribution. While there's a fair amount of interest in sklearn within the machine learning community, you're likely to have heard of scipy if you're coming from the sciences. For this assignment, you'll use scipy.stats to complete your work.

In [1]:
```
from scipy.stats import norm
from scipy.stats import t
import numpy as np
import pandas as pd
from numpy.random import seed
import matplotlib.pyplot as plt
```

Q: Call up the documentation for the `norm` function imported above. What is the second listed method?

In [2]:
```
print (norm.__doc__)
```

```
A normal continuous random variable.

    The location (``loc``) keyword specifies the mean.
    The scale (``scale``) keyword specifies the standard deviation.

    As an instance of the `rv_continuous` class, `norm` object inherits from it
    a collection of generic methods (see below for the full list),
    and completes them with details specific for this particular distribution.

    Methods
    -------
    rvs(loc=0, scale=1, size=1, random_state=None)
        Random variates.
    pdf(x, loc=0, scale=1)
        Probability density function.
    logpdf(x, loc=0, scale=1)
        Log of the probability density function.
    cdf(x, loc=0, scale=1)
        Cumulative distribution function.
    logcdf(x, loc=0, scale=1)
        Log of the cumulative distribution function.
    sf(x, loc=0, scale=1)
        Survival function  (also defined as ``1 - cdf``, but `sf` is sometimes more accurate).
    logsf(x, loc=0, scale=1)
        Log of the survival function.
    ppf(q, loc=0, scale=1)
        Percent point function (inverse of ``cdf`` --- percentiles).
    isf(q, loc=0, scale=1)
        Inverse survival function (inverse of ``sf``).
    moment(n, loc=0, scale=1)
        Non-central moment of order n
    stats(loc=0, scale=1, moments='mv')
        Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
    entropy(loc=0, scale=1)
        (Differential) entropy of the RV.
    fit(data, loc=0, scale=1)
        Parameter estimates for generic data.
    expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)
        Expected value of a function (of one argument) with respect to the distribution.
    median(loc=0, scale=1)
        Median of the distribution.
    mean(loc=0, scale=1)
        Mean of the distribution.
    var(loc=0, scale=1)
        Variance of the distribution.
    std(loc=0, scale=1)
        Standard deviation of the distribution.
    interval(alpha, loc=0, scale=1)
        Endpoints of the range that contains alpha percent of the distribution

    Notes
    -----
    The probability density function for `norm` is:

    .. math::

        f(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}}

    for a real number :math:`x`.

    The probability density above is defined in the "standardized" form. To shift
    and/or scale the distribution use the ``loc`` and ``scale`` parameters.
    Specifically, ``norm.pdf(x, loc, scale)`` is identically
    equivalent to ``norm.pdf(y) / scale`` with
    ``y = (x - loc) / scale``.

    Examples
    --------
    >>> from scipy.stats import norm
    >>> import matplotlib.pyplot as plt
    >>> fig, ax = plt.subplots(1, 1)

    Calculate a few first moments:


    >>> mean, var, skew, kurt = norm.stats(moments='mvsk')

    Display the probability density function (``pdf``):

    >>> x = np.linspace(norm.ppf(0.01),
    ...                 norm.ppf(0.99), 100)
    >>> ax.plot(x, norm.pdf(x),
    ...        'r-', lw=5, alpha=0.6, label='norm pdf')

    Alternatively, the distribution object can be called (as a function)
    to fix the shape, location and scale parameters. This returns a "frozen"
    RV object holding the given parameters fixed.

    Freeze the distribution and display the frozen ``pdf``:

    >>> rv = norm()
    >>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')

    Check accuracy of ``cdf`` and ``ppf``:

    >>> vals = norm.ppf([0.001, 0.5, 0.999])
    >>> np.allclose([0.001, 0.5, 0.999], norm.cdf(vals))
    True

    Generate random numbers:

    >>> r = norm.rvs(size=1000)

    And compare the histogram:

    >>> ax.hist(r, density=True, histtype='stepfilled', alpha=0.2)
    >>> ax.legend(loc='best', frameon=False)
    >>> plt.show()
```

A: Probability density function

Q: Use the method that generates random variates to draw five samples from the standard normal distribution.

A: rvs is the method that generates random variables

In [3]:
```
seed(47)
# draw five samples here
rv = norm.rvs(size=5)
```

Q: What is the mean of this sample? Is it exactly equal to the value you expected? Hint: the sample was drawn from the standard normal distribution.

A: Because of small sample size i didn't got the mean value what i was expecting

In [ ]:

In [4]:
```
# Calculate and print the mean here, hint: use np.mean()
print(np.mean(rv))
```

```
0.19355593334131074
```

Q: What is the standard deviation of these numbers? Calculate this manually here as $\sqrt{\frac{\sum_i(x_i - \bar{x})^2}{n}}$. Hint: np.sqrt() and np.sum() will be useful here and remember that numpy supports broadcasting.

A:

In [5]:
```
np.sqrt(np.sum((rv-np.mean(rv))**2)/5)
```

Out[5]:
```
0.9606195639478641
```

Here we have calculated the actual standard deviation of a small (size 5) data set. But in this case, this small data set is actually a sample from our larger (infinite) population. In this case, the population is infinite because we could keep drawing our normal random variates until our computers die. In general, the sample mean we calculate will not be equal to the population mean (as we saw above). A consequence of this is that the sum of squares of the deviations from the *population* mean will be bigger than the sum of squares of the deviations from the *sample* mean. In other words, the sum of squares of the deviations from the *sample* mean is too small to give an unbiased estimate of the *population* variance. An example of this effect is given here. Scaling our estimate of the variance by the factor $n/(n-1)$ gives an unbiased estimator of the population variance. This factor is known as Bessel's correction. The consequence of this is that the n in the denominator is replaced by $n-1$.

Q: If all we had to go on was our five samples, what would be our best estimate of the population standard deviation? Use Bessel's correction ($n-1$ in the denominator), thus $\sqrt{\frac{\sum_i(x_i - \bar{x})^2}{n-1}}$.

A:

In [6]:
```
np.sqrt(np.sum((rv-np.mean(rv))**2)/4)
```

Out[6]:
```
1.0740053227518152
```

Q: Now use numpy's std function to calculate the standard deviation of our random samples. Which of the above standard deviations did it return?

A:

In [7]:
```
np.std(rv)
```

Out[7]:
```
0.9606195639478641
```

Q: Consult the documentation for np.std() to see how to apply the correction for estimating the population parameter and verify this produces the expected result.

A:

In [8]:
```
print(np.std.__doc__)
```

```
    Compute the standard deviation along the specified axis.

    Returns the standard deviation, a measure of the spread of a distribution,
    of the array elements. The standard deviation is computed for the
    flattened array by default, otherwise over the specified axis.

    Parameters
    ----------
    a : array_like
        Calculate the standard deviation of these values.
    axis : None or int or tuple of ints, optional
        Axis or axes along which the standard deviation is computed. The
        default is to compute the standard deviation of the flattened array.

        .. versionadded:: 1.7.0

        If this is a tuple of ints, a standard deviation is performed over
        multiple axes, instead of a single axis or all the axes as before.
    dtype : dtype, optional
        Type to use in computing the standard deviation. For arrays of
        integer type the default is float64, for arrays of float types it is
        the same as the array type.
    out : ndarray, optional
        Alternative output array in which to place the result. It must have
        the same shape as the expected output but the type (of the calculated
        values) will be cast if necessary.
    ddof : int, optional
        Means Delta Degrees of Freedom.  The divisor used in calculations
        is ``N - ddof``, where ``N`` represents the number of elements.
        By default `ddof` is zero.
    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.

        If the default value is passed, then `keepdims` will not be
        passed through to the `std` method of sub-classes of
        `ndarray`, however any non-default value will be.  If the
        sub-class' method does not implement `keepdims` any
        exceptions will be raised.

    Returns
    -------
    standard_deviation : ndarray, see dtype parameter above.
        If `out` is None, return a new array containing the standard deviation,
        otherwise return a reference to the output array.

    See Also
    --------
    var, mean, nanmean, nanstd, nanvar
    numpy.doc.ufuncs : Section "Output arguments"

    Notes
    -----
    The standard deviation is the square root of the average of the squared
    deviations from the mean, i.e., ``std = sqrt(mean(abs(x - x.mean())**2))``.

    The average squared deviation is normally calculated as
    ``x.sum() / N``, where ``N = len(x)``.  If, however, `ddof` is specified,
    the divisor ``N - ddof`` is used instead. In standard statistical
    practice, ``ddof=1`` provides an unbiased estimator of the variance
    of the infinite population. ``ddof=0`` provides a maximum likelihood
    estimate of the variance for normally distributed variables. The
    standard deviation computed in this function is the square root of
    the estimated variance, so even with ``ddof=1``, it will not be an
    unbiased estimate of the standard deviation per se.

    Note that, for complex numbers, `std` takes the absolute
    value before squaring, so that the result is always real and nonnegative.

    For floating-point input, the *std* is computed using the same
    precision the input has. Depending on the input data, this can cause
    the results to be inaccurate, especially for float32 (see example below).
    Specifying a higher-accuracy accumulator using the `dtype` keyword can
    alleviate this issue.

    Examples
    --------
    >>> a = np.array([[1, 2], [3, 4]])
    >>> np.std(a)
    1.1180339887498949
    >>> np.std(a, axis=0)
    array([ 1.,  1.])
    >>> np.std(a, axis=1)
    array([ 0.5,  0.5])

    In single precision, std() can be inaccurate:

    >>> a = np.zeros((2, 512*512), dtype=np.float32)
    >>> a[0, :] = 1.0
    >>> a[1, :] = 0.1
    >>> np.std(a)
    0.45000005

    Computing the standard deviation in float64 is more accurate:

    >>> np.std(a, dtype=np.float64)
    0.44999999925494177
```

In [9]:
```
np.std(rv, ddof=1)
```

Out[9]:
```
1.0740053227518152
```

Summary of section

In this section, you've been introduced to the scipy.stats package and used it to draw a small sample from the standard normal distribution. You've calculated the average (the mean) of this sample and seen that this is not exactly equal to the expected population parameter (which we know because we're generating the random variates from a specific, known distribution). You've been introduced to two ways of calculating the standard deviation; one uses n in the denominator and the other uses $n-1$ (Bessel's correction). You've also seen which of these calculations np.std() performs by default and how to get it to generate the other.

You use n as the denominator if you want to calculate the standard deviation of a sequence of numbers. You use $n-1$ if you are using this sequence of numbers to estimate the population parameter. This brings us to some terminology that can be a little confusing.

The population parameter is traditionally written as σ and the sample statistic as s. Rather unhelpfully, s is also called the sample standard deviation (using $n-1$) whereas the standard deviation of the sample uses n. That's right, we have the sample standard deviation and the standard deviation of the sample and they're not the same thing!

The sample standard deviation
$$s = \sqrt{\frac{\sum_i(x_i - \bar{x})^2}{n-1}} \approx \sigma,$$
is our best (unbiased) estimate of the population parameter (σ).

If your data set *is* your entire population, you simply want to calculate the population parameter, σ, via
$$\sigma = \sqrt{\frac{\sum_i(x_i - \bar{x})^2}{n}}$$
as you have complete, full knowledge of your population. In other words, your sample *is* your population. It's worth noting at this point if your sample is your population then you know absolutely everything about your population, there are no probabilities really to calculate and no inference to be done.

If, however, you have sampled *from* your population, you only have partial knowledge of the state of your population and the standard deviation of your sample is not an unbiased estimate of the standard deviation of the population, in which case you seek to estimate that population parameter via the sample standard deviation, which uses the $n-1$ denominator.

You're now firmly in frequentist theory territory. Great work so far! Now let's dive deeper.

II Sampling distributions

So far we've been dealing with the concept of taking a sample from a population to infer the population parameters. One statistic we calculated for a sample was the mean. As our samples will be expected to vary from one draw to another, so will our sample statistics. If we were to perform repeat draws of size n and calculate the mean of each, we would expect to obtain a distribution of values. This is the sampling distribution of the mean. The Central Limit Theorem (CLT) tells us that such a distribution will approach a normal distribution as n increases. For the sampling distribution of the mean, the standard deviation of this distribution is given by
$$\sigma_{mean} = \frac{\sigma}{\sqrt n}$$

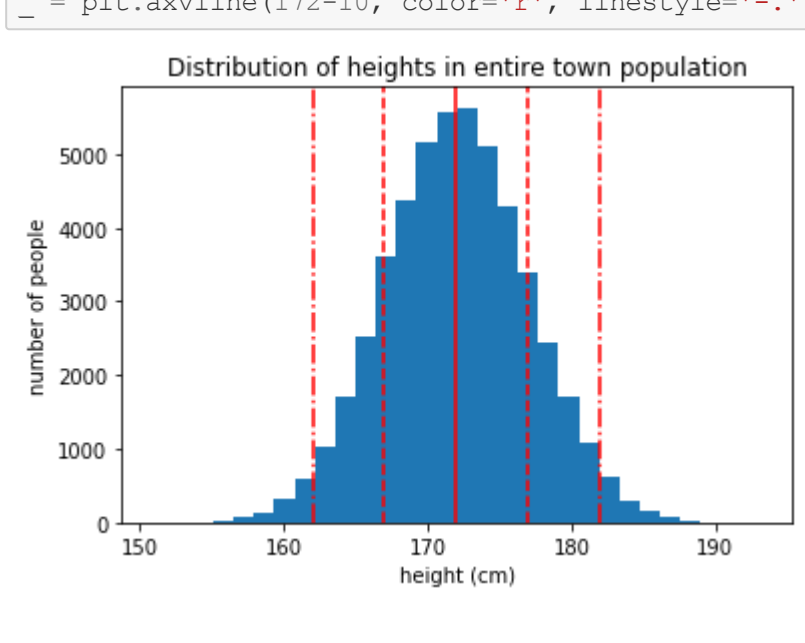
where σ_{mean} is the standard deviation of the sampling distribution of the mean and σ is the standard deviation of the population (the population parameter).

This is important because typically we are dealing with samples from populations and all we know about the population is what we see in the sample. From this sample, we want to make inferences about the population. We may do this, for example, by looking at the histogram of the values and by calculating the mean and standard deviation (as estimates of the population parameters), and so we are intrinsically interested in how these quantities vary across samples. In other words, now that we've taken one sample of size n and made some claims about the general population, what if we were to take another sample of size n? Would we get the same result? Would we make the same claims about the general population? This brings us to a fundamental question: *when we make some inference about a population based on our sample, how confident can we be that we've got it 'right'?*

Let's give our normal distribution a little flavor. Also, for didactic purposes, the standard normal distribution, with its variance equal to its standard deviation of one, would not be a great illustration of a key point. Let us imagine we live in a town of 50000 people and we know the height of everyone in this town. We will have 50000 numbers that tell us everything about our population. We'll simulate these numbers now and put ourselves in one particular town, called 'town 47', where the population mean height is 172 cm and population standard deviation is 5 cm.

In [10]:
```
seed(47)
pop_heights = norm.rvs(172, 5, size=50000)
```

In [11]:
```
_ = plt.hist(pop_heights, bins=30)
_ = plt.xlabel('height (cm)')
_ = plt.ylabel('number of people')
_ = plt.title('Distribution of heights in entire town population')
_ = plt.axvline(172, color='r')
_ = plt.axvline(172+5, color='r', linestyle='--')
_ = plt.axvline(172-5, color='r', linestyle='--')
_ = plt.axvline(172+10, color='r', linestyle='-.')
_ = plt.axvline(172-10, color='r', linestyle='-.')
```



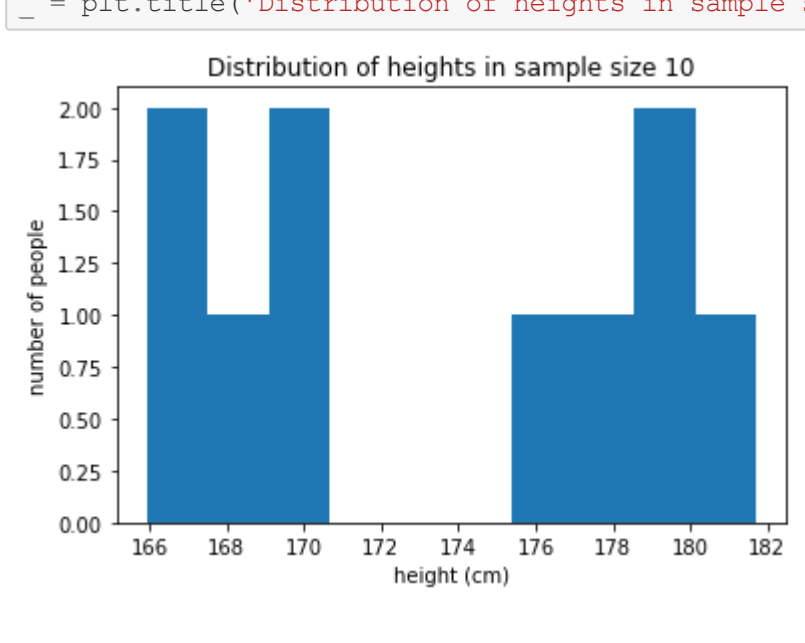
Now, 50000 people is rather a lot to chase after with a tape measure. If all you want to know is the average height of the townsfolk, then can you just go out and measure a sample to get a pretty good estimate of the average height?

In [12]:
```
def townsfolk_sampler(n):
    return np.random.choice(pop_heights, n)
```

Let's say you go out one day and randomly sample 10 people to measure.

In [13]:
```
seed(47)
daily_sample1 = townsfolk_sampler(10)
```

In [14]:
```
_ = plt.hist(daily_sample1, bins=10)
_ = plt.xlabel('height (cm)')
_ = plt.ylabel('number of people')
_ = plt.title('Distribution of heights in sample size 10')
```



The sample distribution doesn't look much like what we know (but wouldn't know in real-life) the population distribution looks like. What do we get for the mean?

In [15]:
```
np.mean(daily_sample1)
```

Out[15]:
```
173.47911444163503
```

And if we went out and repeated this experiment?

In [16]:
```
daily_sample2 = townsfolk_sampler(10)
```

In [17]:
```
np.mean(daily_sample2)
```

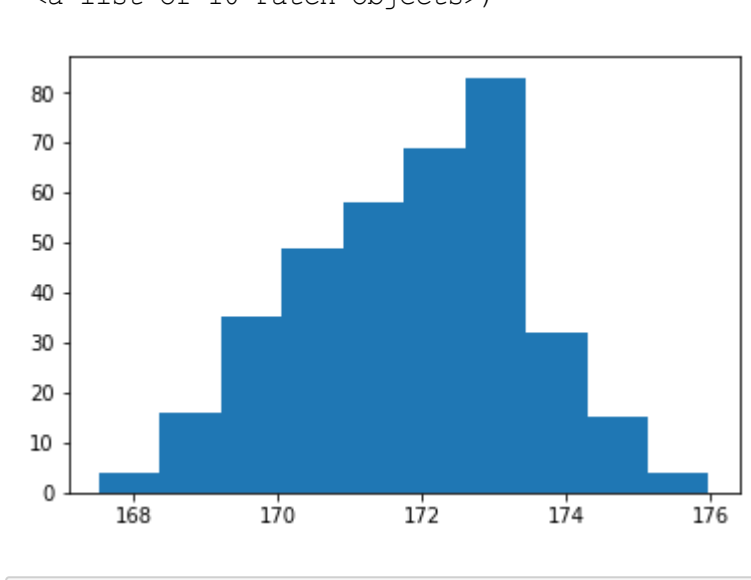
Out[17]:
```
173.7317666636263
```

Q: Simulate performing this random trial every day for a year, calculating the mean of each daily sample of 10, and plot the resultant sampling distribution of the mean.

A:

In [18]:
```
seed(47)
# take your samples here
arr=[]
for i in range (365):
    df=np.mean(townsfolk_sampler(10))
    arr.append(df)
plt.hist(arr, bins=10)
```

Out[18]:
```
(array([ 4., 16., 35., 49., 58., 69., 83., 32., 15.,  4.]),
 array([167.52865966, 168.37373915, 169.21881865, 170.06389814,
        170.90897763, 171.75405713, 172.59913662, 173.44421611,
        174.28929561, 175.1343751 , 175.9794546 ]),
 <a list of 10 Patch objects>)
```



In [19]:
```
np.std(pop_heights)/(10**0.5)
```

Out[19]:
```
1.5782419116544761
```

The above is the distribution of the means of samples of size 10 taken from our population. The Central Limit Theorem tells us the expected mean of this distribution will be equal to the population mean, and standard deviation will be $\sigma / \sqrt n$, which, in this case, should be approximately 1.58.

Q: Verify the above results from the CLT.

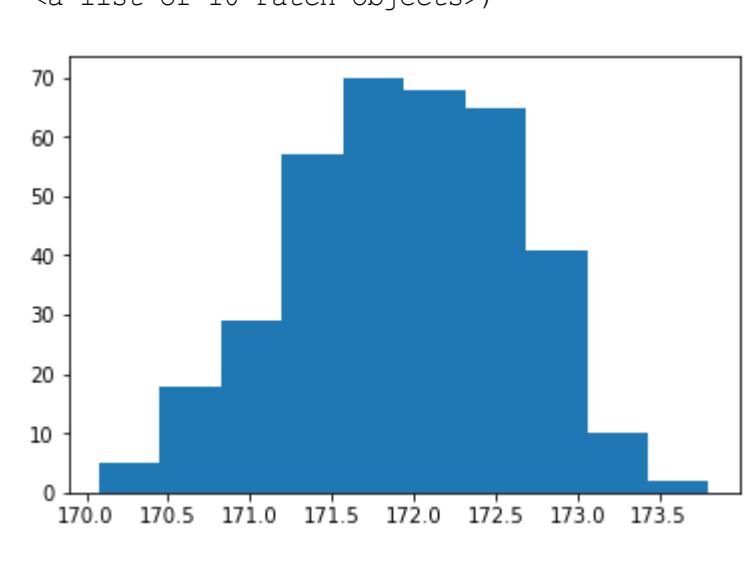
Remember, in this instance, we knew our population parameters, that the average height really is 172 cm and the standard deviation is 5 cm, and we see some of our daily estimates of the population mean were as low as around 168 and some as high as 176.

Q: Repeat the above year's worth of samples but for a sample size of 50 (perhaps you had a bigger budget for conducting surveys that year!) Would you expect your distribution of sample means to be wider (more variable) or narrower (more consistent)? Compare your resultant summary statistics to those predicted by the CLT.

A: I expect the distribution to be wider

In [20]:
```
seed(47)
# calculate daily means from the larger sample size here
arr1=[]
for i in range (365):
    df=np.mean(townsfolk_sampler(50))
    arr1.append(df)
plt.hist(arr1, bins=10)
```

Out[20]:
```
(array([ 5., 18., 29., 57., 70., 68., 65., 41., 10.,  2.]),
 array([170.07952859, 170.45187698, 170.82422537, 171.19657376,
        171.56892215, 171.94127054, 172.31361893, 172.68596732,
        173.05831571, 173.4306641 , 173.80301249]),
 <a list of 10 Patch objects>)
```



What we've seen so far, then, is that we can estimate population parameters from a sample from the population, and that samples have their own distributions. Furthermore, the larger the sample size, the narrower are those sampling distributions.

III Normally testing times!

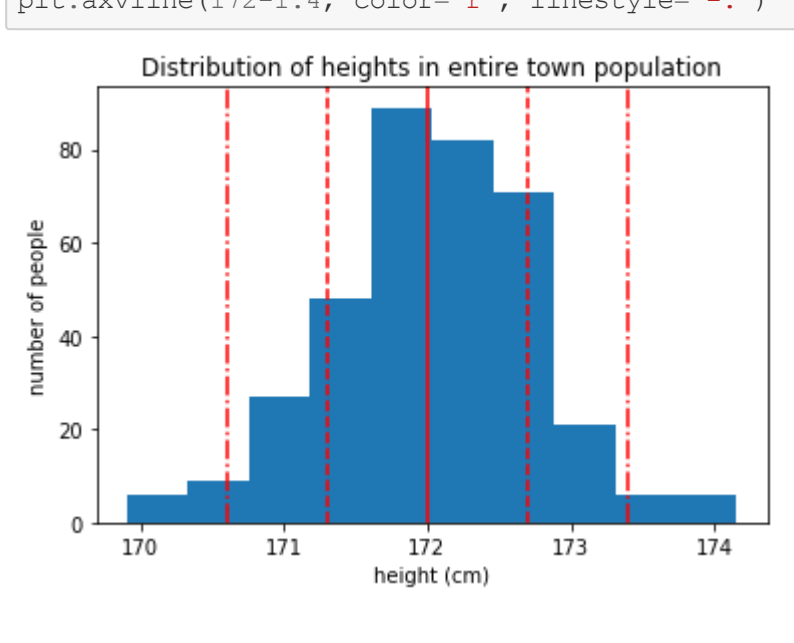
All of the above is well and good. We've been sampling from a population we know is normally distributed, we've come to understand when to use n and when to use $n-1$ in the denominator to calculate the spread of a distribution, and we've seen the Central Limit Theorem in action for a sampling distribution. All seems very well behaved in Frequentist land. But, well, why should we really care?

Remember, we rarely (if ever) actually know our population parameters but you still have to estimate them somehow. If we want to make inferences such as "is this observation unusual?" or "has my population mean changed?" then you need to have some idea of what the underlying distribution is so you can calculate relevant probabilities. In frequentist inference, you use the formulas above to deduce these population parameters. Take a moment in the next part of this assignment to refresh your understanding of how these probabilities work.

Recall some basic properties of the standard Normal distribution, such as about 68% of observations being within plus or minus 1 standard deviation of the mean.

Q: Using this fact, calculate the probability of observing the value 1 or less in a single observation from the standard normal distribution. Hint: you may find it helpful to sketch the standard normal distribution (the familiar bell shape) and mark the number of standard deviations from the mean on the x-axis and shade the regions of the curve that contain certain percentages of the population.

In [21]:
```
arr1=[]
for i in range (365):
    df=np.mean(townsfolk_sampler(50))
    arr1.append(df)
plt.hist(arr1, bins=10)
plt.xlabel('height (cm)')
plt.ylabel('number of people')
plt.title('Distribution of heights in entire town population')
plt.axvline(172, color='r')
plt.axvline(172+0.7, color='r', linestyle='--')
plt.axvline(172-0.7, color='r', linestyle='--')
plt.axvline(172+1.4, color='r', linestyle='-.')
plt.axvline(172-1.4, color='r', linestyle='-.')
```



Calculating this probability involved calculating the area under the pdf from the value of 1 and below. To put it another way, we need to integrate the pdf. We could just add together the known areas of chunks (from -Inf to 0 and then 0 to $+\sigma$ in the example above. One way to do this is using look up tables (literally). Fortunately, scipy has this functionality built in with the cdf() function.

Q: Use the cdf() function to answer the question above again and verify you get the same answer.

A:

In [21]:
```
norm.cdf(1)
```

Out[21]:
```
0.8413447460685429
```

Q: Using our knowledge of the population parameters for our townsfolk's heights, what is the probability of selecting one person at random and their height being 177 cm or less? Calculate this using both of the approaches given above.

A:

In [22]:
```
norm.cdf(177, loc=172, scale=5)
```

Out[22]:
```
0.8413447460685429
```

Q: Turning this question around. Let's say we randomly pick one person and measure their height and find they are 2.00 m tall? How surprised should we be at this result, given what we know about the population distribution? In other words, how likely would it be to obtain a value at least as extreme as this? Express this as a probability.

A: Probability of occuring this height is around 0.001

In [23]:
```
norm.cdf(199, loc=172, scale=5)
```

Out[23]:
```
0.9999999666795515
```

We could calculate this probability by virtue of knowing the population parameters. We were then able to use the known properties of the relevant normal distribution to calculate the probability of observing a value at least as extreme as our test value. We have essentially just performed a z-test (albeit without having prespecified a threshold for our "level of surprise")!

We're about to come to a pinch, though here. We've said a couple of times that we rarely, if ever, know the true population parameters; we have to estimate them from our sample and we cannot even begin to estimate the standard deviation from a single observation. This is very true and usually we have sample sizes larger than one. This means we can calculate the mean of the sample as our best estimate of the population mean and the standard deviation as our best estimate of the population standard deviation. In other words, we are now coming to deal with the sampling distributions we mentioned above as we are generally concerned with the properties of the sample means we obtain.

Above, we highlighted one result from the CLT, whereby the sampling distribution (of the mean) becomes narrower and narrower with the square root of the sample size. We remind ourselves that another result from the CLT is that *even if the underlying population distribution is not normal, the sampling distribution will tend to become normal with sufficiently large sample size*. This is the key driver for us 'requiring' a certain sample size, for example you may frequently see a minimum sample size of 30 stated in many places. In reality this is simply a rule of thumb; if the underlying distribution is approximately normal then your sampling distribution will already be pretty normal, but if the underlying distribution is heavily skewed then you'd want to increase your sample size.

Q: Let's now start from the position of knowing nothing about the heights of people in our town.

- Use our favorite random seed of 47, to randomly sample the heights of 50 townsfolk
- Estimate the population mean using np.mean
- Estimate the population standard deviation using np.std (remember which denominator to use!)
- Calculate the (95%) margin of error (use the exact critial z value to 2 decimal places - look this up or use norm.ppf())
- Calculate the 95% Confidence Interval of the mean
- Does this interval include the true population mean?

A:

In [24]:
```
seed(47)
# take your sample now
sample=townsfolk_sampler(50)
```

In [31]:
```
pop_mean = np.mean(pop_heights)
pop_mean
```

Out[31]:
```
172.0192602425845
```

In [32]:
```
pop_std = np.std(pop_heights)
pop_std
```

Out[32]:
```
4.990839139566387
```

In [34]:
```
# Margin of error = Critical value x Standard deviation
MOE = round((1.96 * pop_std), 2)
MOE
```

Out[34]:
```
9.78
```

In [37]:
```
CI = round((MOE / np.sqrt(50)), 2)
print(CI)
CI_high = pop_mean + CI
CI_low = pop_mean - CI
print(CI_high)
print(CI_low)
```

```
1.38
173.3992602425845
170.6392602425845
```

In [38]:
```
# Critical value from z table: 1.96
z = 1.9
p = 0.975 # two tailed
z2 = norm.ppf(p)
print(z , z2)
```

```
1.9 1.959963984540054
```

In [ ]:

In [ ]:

In [ ]:

Q: Above we calculated the confidence interval using the critical z value. What is the problem with this? What requirement, or requirements, are we (strictly) failing?

A:The critical value of z is linked to the area under the standard normal model. We're assuming we have a standard normal model. We may not, especially with a "small sample".

Q: Calculate the 95% confidence interval for the mean using the *t* distribution. Is this wider or narrower than that based on the normal distribution above? If you're unsure, you may find this resource useful. For calculating the critical value, remember how you could calculate this for the normal distribution using norm.ppf().

A:

In [39]:
```
dof = 49
alpha = (1-0.95)/2
x = np.std(sample)/(50**0.5)
```