Initiation_Python_Jupyter_TP2

October 11, 2021

In [2]:
```
random()
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-2-347a394b3b57> in <module>
----> 1 random()


NameError: name 'random' is not defined
```

In [3]:
```
import numpy
print(numpy.pi)
print(numpy.e)
print(numpy.euler_gamma)
```

```
3.141592653589793
2.718281828459045
0.5772156649015329
```

In [4]:
```
print(numpy.random.rand())
print(numpy.random.randint(1,10))
```

```
0.7644154963495916
7
```

In [5]:
```
print(numpy.random.binomial(100,0.25,10))
print()
help(numpy.random.binomial)
```

```
[28 25 19 28 21 36 20 24 31 28]

Help on built-in function binomial:
```

```
binomial(...) method of mtrand.RandomState instance
    binomial(n, p, size=None)

    Draw samples from a binomial distribution.

    Samples are drawn from a binomial distribution with specified
    parameters, n trials and p probability of success where
    n an integer >= 0 and p is in the interval [0,1]. (n may be
    input as a float, but it is truncated to an integer in use)

    Parameters
    ----------
    n : int or array_like of ints
        Parameter of the distribution, >= 0. Floats are also accepted,
        but they will be truncated to integers.
    p : float or array_like of floats
        Parameter of the distribution, >= 0 and <=1.
    size : int or tuple of ints, optional
        Output shape.  If the given shape is, e.g., ``(m, n, k)``, then
        ``m * n * k`` samples are drawn.  If size is ``None`` (default),
        a single value is returned if ``n`` and ``p`` are both scalars.
        Otherwise, ``np.broadcast(n, p).size`` samples are drawn.

    Returns
    -------
    out : ndarray or scalar
        Drawn samples from the parameterized binomial distribution, where
        each sample is equal to the number of successes over the n trials.

    See Also
    --------
    scipy.stats.binom : probability density function, distribution or
        cumulative density function, etc.

    Notes
    -----
    The probability density for the binomial distribution is

    .. math:: P(N) = \binom{n}{N}p^N(1-p)^{n-N},

    where :math:`n` is the number of trials, :math:`p` is the probability
    of success, and :math:`N` is the number of successes.

    When estimating the standard error of a proportion in a population by
    using a random sample, the normal distribution works well unless the
    product p*n <=5, where p = population proportion estimate, and n =
    number of samples, in which case the binomial distribution is used
```

```
instead. For example, a sample of 15 people shows 4 who are left
handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4,
so the binomial distribution should be used in this case.

References
----------
.. [1] Dalgaard, Peter, "Introductory Statistics with R",
       Springer-Verlag, 2002.
.. [2] Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill,
       Fifth Edition, 2002.
.. [3] Lentner, Marvin, "Elementary Applied Statistics", Bogden
       and Quigley, 1972.
.. [4] Weisstein, Eric W. "Binomial Distribution." From MathWorld--A
       Wolfram Web Resource.
       http://mathworld.wolfram.com/BinomialDistribution.html
.. [5] Wikipedia, "Binomial distribution",
       https://en.wikipedia.org/wiki/Binomial_distribution

Examples
--------
Draw samples from the distribution:

>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.

A real world example. A company drills 9 wild-cat oil exploration
wells, each with an estimated probability of success of 0.1. All nine
wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that
generate zero positive results.

>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
```

In [6]:
```python
print(numpy.cos(numpy.e**2))
print(numpy.arccos(0.4))
print(numpy.e**2-2*numpy.pi)
```

```
0.44835624181873357
1.1592794807274085
1.1058707917510633
```

In [7]:
```python
from numpy import *
import numpy.random as npr
```

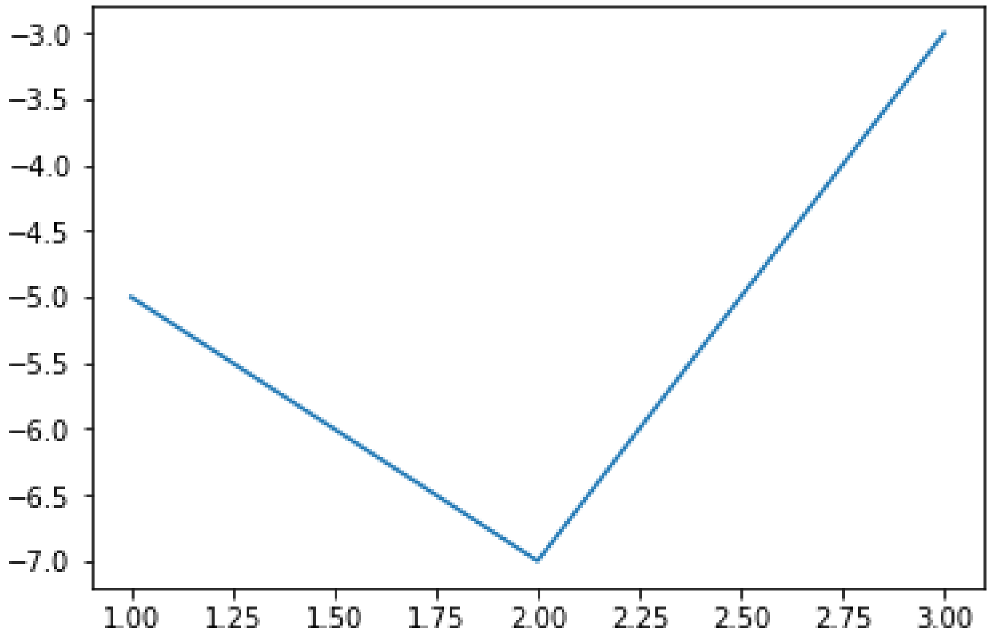
```
print(pi)
print(e)
print(random.rand())
print(npr.rand())
print(rand())
```

```
3.141592653589793
2.718281828459045
0.6228003968508204
0.9028626208717166
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-7-44c03736a0b2> in <module>
  6 print(random.rand())
  7 print(npr.rand())
----> 8 print(rand())


NameError: name 'rand' is not defined
```

In [8]:
```
import matplotlib.pyplot as plt
from scipy import *
from math import factorial as fact
```

In [9]:
```
plt.plot([1,2,3],[-5,-7,-3])
plt.show()
```




```
In [10]: help(numpy.random.chisquare)
```

```
Help on built-in function chisquare:

chisquare(...) method of mtrand.RandomState instance
    chisquare(df, size=None)

    Draw samples from a chi-square distribution.

    When `df` independent random variables, each with standard normal
    distributions (mean 0, variance 1), are squared and summed, the
    resulting distribution is chi-square (see Notes).  This distribution
    is often used in hypothesis testing.

    Parameters
    ----------
    df : float or array_like of floats
         Number of degrees of freedom, should be > 0.
    size : int or tuple of ints, optional
        Output shape.  If the given shape is, e.g., ``(m, n, k)``, then
        ``m * n * k`` samples are drawn.  If size is ``None`` (default),
        a single value is returned if ``df`` is a scalar.  Otherwise,
        ``np.array(df).size`` samples are drawn.

    Returns
    -------
```

```
    out : ndarray or scalar
        Drawn samples from the parameterized chi-square distribution.

    Raises
    ------
    ValueError
        When `df` <= 0 or when an inappropriate `size` (e.g. ``size=-1``)
        is given.

    Notes
    -----
    The variable obtained by summing the squares of `df` independent,
    standard normally distributed random variables:

    .. math:: Q = \sum_{i=0}^{\mathtt{df}} X^2_i

    is chi-square distributed, denoted

    .. math:: Q \sim \chi^2_k.

    The probability density function of the chi-squared distribution is

    .. math:: p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)}
                     x^{k/2 - 1} e^{-x/2},

    where :math:`\Gamma` is the gamma function,

    .. math:: \Gamma(x) = \int_0^{-\infty} t^{x - 1} e^{-t} dt.

    References
    ----------
    .. [1] NIST "Engineering Statistics Handbook"
           https://www.itl.nist.gov/div898/handbook/eda/section3/eda3666.htm

    Examples
    --------
    >>> np.random.chisquare(2,4)
    array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

```
In [11]: L = [2,3,5,7,11,13,17,19]
         print(L)
         print(type(L))
         print(L[0])
         print(L[3])
         print(L[-1])
         print(len(L))
```

```
         print(L[:4])
         print(L[4:])
         print(L[1:6])
```

```
[2, 3, 5, 7, 11, 13, 17, 19]
<class 'list'>
2
7
19
8
[2, 3, 5, 7]
[11, 13, 17, 19]
[3, 5, 7, 11, 13]
```

```
In [12]: L.append(23)
         print(L)
         L.append(29)
         print(L)
         L.append("N'importe quoi")
         print(L)
         L.append([1,1,2,3,5,8,13,21])
         print(L)
         L[-1].append(34)
         print(L)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, "N'importe quoi"]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, "N'importe quoi", [1, 1, 2, 3, 5, 8, 13, 21]]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, "N'importe quoi", [1, 1, 2, 3, 5, 8, 13, 21, 34]]
```

```
In [13]: L.insert(0,1)
         print(L)
         L.insert(11,31)
         print(L)
         L[-1].insert(0,0)
         print(L)
```

```
[1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, "N'importe quoi", [1, 1, 2, 3, 5, 8, 13, 21, 34]]
[1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, "N'importe quoi", [1, 1, 2, 3, 5, 8, 13, 21, 34]]
[1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, "N'importe quoi", [0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

```
In [14]: M = [1,2,3]
         N = [4,5,6]
         print(M,N)
         K = M+N
```

```
        print(K)
        print(M,N)
```

```
[1, 2, 3] [4, 5, 6]
[1, 2, 3, 4, 5, 6]
[1, 2, 3] [4, 5, 6]
```

```
In [15]: # L = []
         # for k in range(100):
         #     L.append(npr.rand())

         L = [npr.rand() for k in range(100)]
         print(L[:20])
         L.sort()
         print(L[:20])
```

```
[0.6904718987457737, 0.27602967675694545, 0.9031961023170058, 0.05292021204701536, 0.591587462
[0.005998683497628554, 0.010538130278089364, 0.02295180838868116, 0.04584621747558537, 0.05292
```

```
In [16]: L = [(i,j) for i in range(1,21) for j in range(1,21) if i-j >= 10]
         print(L)

         L = [npr.rand() for k in range(100)]
         M = [x for x in L if x > 1/2]
         print(len(M),min(M))
```

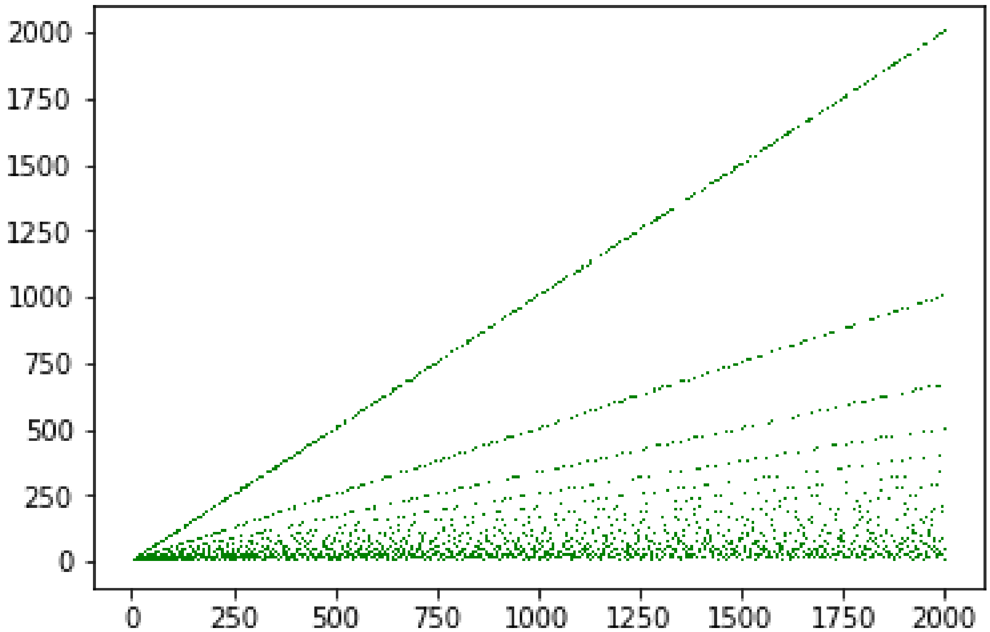
```
[(11, 1), (12, 1), (12, 2), (13, 1), (13, 2), (13, 3), (14, 1), (14, 2), (14, 3), (14, 4), (15
47 0.5036075807762839
```

```
In [17]: n = 3628800
         D = [d for d in range(1,n+1) if n%d == 0]
         print(len(D),D[100])
```

```
270 700
```

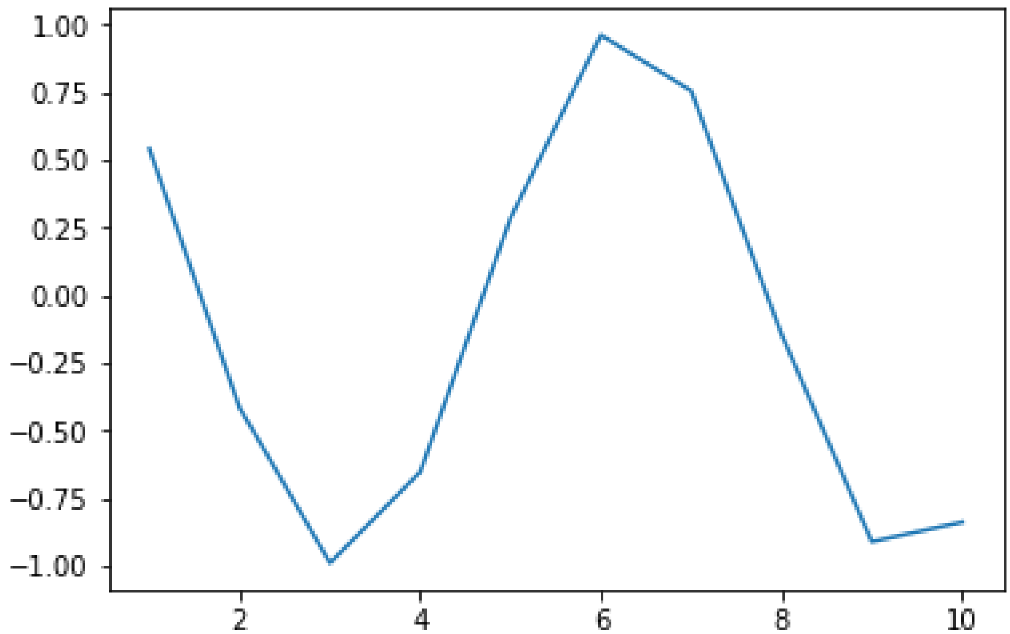
```
In [76]: def nombre_de_diviseurs(n):
             c = 0
             for d in range(1,n+1):
                 if n%d == 0:
                     c += 1
             return c

         def est_premier(p):
             return nombre_de_diviseurs(p) == 2
```

```
def plus_grand_divseur_premier(n):
    for d in range(n,1,-1):
        if n%d == 0:
            if est_premier(d):
                return d

N = [n for n in range(2,2001)]
P = [plus_grand_divseur_premier(n) for n in N]
plt.plot(N,P,'g,')
plt.show()
```




```
In [19]: A = numpy.array([1,2,3,4,5,6,7,8,9,10])
         print(A)
         B = numpy.cos(A)
         print(B)
         plt.plot(A,B)
         plt.show()
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[ 0.54030231 -0.41614684 -0.9899925  -0.65364362  0.28366219  0.96017029
  0.75390225 -0.14550003 -0.91113026 -0.83907153]
```



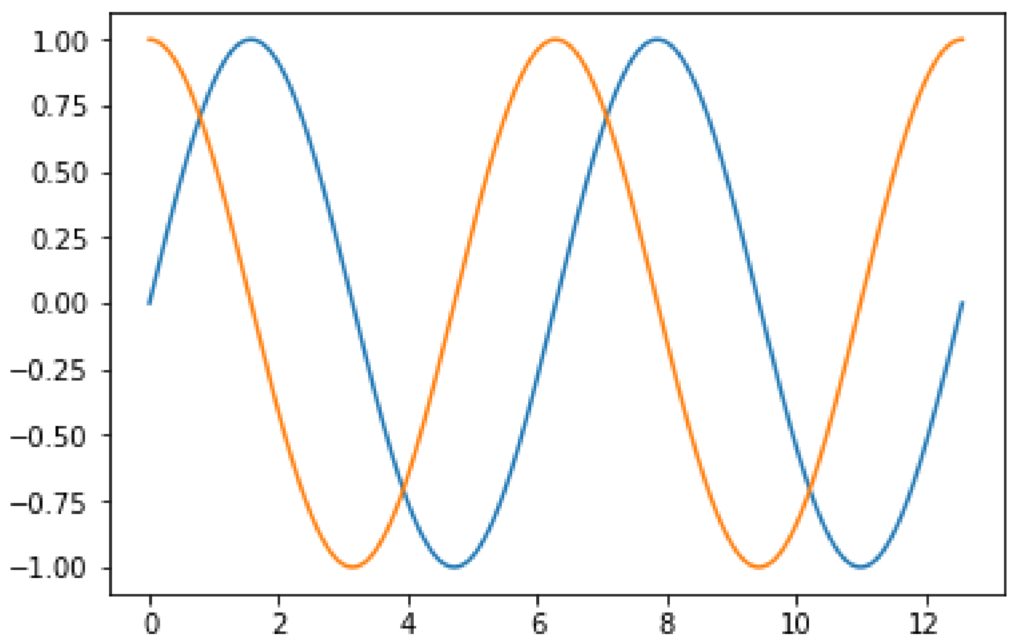
In [20]: help(numpy.linspace)

```
Help on function linspace in module numpy:

linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)
    Return evenly spaced numbers over a specified interval.

    Returns `num` evenly spaced samples, calculated over the
    interval [`start`, `stop`].

    The endpoint of the interval can optionally be excluded.

    .. versionchanged:: 1.16.0
        Non-scalar `start` and `stop` are now supported.

    Parameters
    ----------
    start : array_like
        The starting value of the sequence.
    stop : array_like
        The end value of the sequence, unless `endpoint` is set to False.
        In that case, the sequence consists of all but the last of ``num + 1``
        evenly spaced samples, so that `stop` is excluded.  Note that the step
        size changes when `endpoint` is False.
    num : int, optional
        Number of samples to generate. Default is 50. Must be non-negative.
```

```
endpoint : bool, optional
    If True, `stop` is the last sample. Otherwise, it is not included.
    Default is True.
retstep : bool, optional
    If True, return (`samples`, `step`), where `step` is the spacing
    between samples.
dtype : dtype, optional
    The type of the output array.  If `dtype` is not given, infer the data
    type from the other input arguments.

    .. versionadded:: 1.9.0

axis : int, optional
    The axis in the result to store the samples.  Relevant only if start
    or stop are array-like.  By default (0), the samples will be along a
    new axis inserted at the beginning. Use -1 to get an axis at the end.

    .. versionadded:: 1.16.0

Returns
-------
samples : ndarray
    There are `num` equally spaced samples in the closed interval
    ``[start, stop]`` or the half-open interval ``[start, stop)``
    (depending on whether `endpoint` is True or False).
step : float, optional
    Only returned if `retstep` is True

    Size of spacing between samples.


See Also
--------
arange : Similar to `linspace`, but uses a step size (instead of the
         number of samples).
geomspace : Similar to `linspace`, but with numbers spaced evenly on a log
            scale (a geometric progression).
logspace : Similar to `geomspace`, but with the end points specified as
           logarithms.

Examples
--------
>>> np.linspace(2.0, 3.0, num=5)
array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([ 2. ,  2.2,  2.4,  2.6,  2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
```

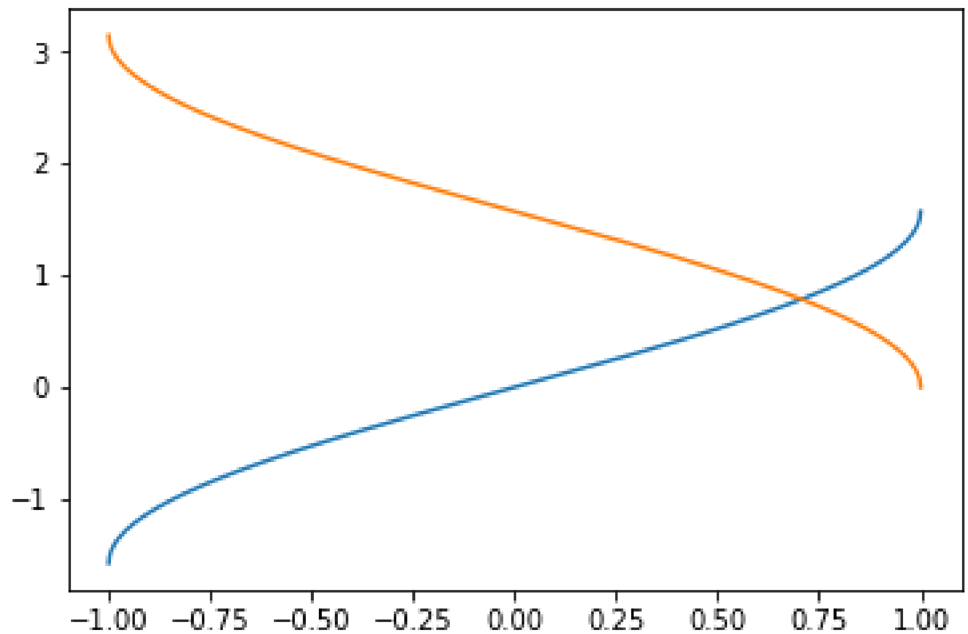
```
Graphical illustration:

>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> 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()
```

```
In [21]: L = numpy.linspace(0,4*numpy.pi,1000)
         M = numpy.sin(L)
         plt.plot(L,M)
         N = numpy.cos(L)
         plt.plot(L,N)
         plt.show()
```




```
In [22]: L = numpy.linspace(-1,1,1000)
         M = numpy.arcsin(L)
```

```
plt.plot(L,M)
N = numpy.arccos(L)
plt.plot(L,N)
plt.show()
```




```
In [23]: M = numpy.array([[1,2],[4,5],[7,8]])
         print(M)
```

```
[[1 2]
 [4 5]
 [7 8]]
```

```
In [24]: M = numpy.array([[2,1,3,1],[6,3,9,3],[2,1,3,1],[6,3,9,3]])
         print(M)
         print(M.dot(M))
```

```
[[2 1 3 1]
 [6 3 9 3]
 [2 1 3 1]
 [6 3 9 3]]
[[22 11 33 11]
 [66 33 99 33]
 [22 11 33 11]
 [66 33 99 33]]
```

```
In [25]: M = numpy.array([[npr.randint(1,10) for j in range(3)] for k in range(3)])
         print(M)
         print(M.dot(M))
```

```
[[7 1 3]
 [7 8 2]
 [7 1 5]]
[[ 77  18  38]
 [119  73  47]
 [ 91  20  48]]
```

```
In [26]: n, m = 4,6
         print(numpy.zeros((n,m)))
         print(numpy.ones((m,n)))
         print(numpy.eye(n))
```

```
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

```
In [27]: print(M)
         numpy.shape(M)
```

```
[[7 1 3]
 [7 8 2]
 [7 1 5]]
```

```
Out[27]: (3, 3)
```

```
In [28]: def is_vampire(M):
             n,m = numpy.shape(M)
             if not n == m:
                 return False
             for i in range(n):
                 for j in range(m):
```

```
                if M[i,j] < 0 or M[i,j] > 9:
                    return False
        return (M.dot(M) == 11*M).all()

    def list_of_matrices(n,m,S):
        L = []
        M = numpy.array([[S[0] for i in range(m)] for j in range(n)])
        L.append(numpy.array(M))
        for i in range(n):
            for j in range(m):
                new_L = []
                for M in L:
                    for s in S:
                        N = copy(M)
                        N[i,j] = s
                        new_L.append(N)
                L = list(new_L)
        return L

    L = list_of_matrices(2,3,[0,1,2,3,4])
    print(len(L))
    print(5**6)

    def all_vampires(n):
        Vampires = []
        S = [0,1,2,3,4,5,6,7,8,9]
        L = list_of_matrices(n,n,S)
        for M in L:
            if is_vampire(M):
                Vampires.append(M)
        return Vampires

    V = all_vampires(2)
    print(len(V))
    for M in V:
        print(M,M.dot(M))
    D = [numpy.linalg.det(M) for M in V]
    T = [M.trace() for M in V]
    print(D)
    print(T)
```

```
15625
15625
25
[[0 0]
 [0 0]] [[0 0]
 [0 0]]
[[2 2]
```

```
 [9 9]] [[22 22]
 [99 99]]
[[2 3]
 [6 9]] [[22 33]
 [66 99]]
[[2 6]
 [3 9]] [[22 66]
 [33 99]]
[[2 9]
 [2 9]] [[22 99]
 [22 99]]
[[3 3]
 [8 8]] [[33 33]
 [88 88]]
[[3 4]
 [6 8]] [[33 44]
 [66 88]]
[[3 6]
 [4 8]] [[33 66]
 [44 88]]
[[3 8]
 [3 8]] [[33 88]
 [33 88]]
[[4 4]
 [7 7]] [[44 44]
 [77 77]]
[[4 7]
 [4 7]] [[44 77]
 [44 77]]
[[5 5]
 [6 6]] [[55 55]
 [66 66]]
[[5 6]
 [5 6]] [[55 66]
 [55 66]]
[[6 5]
 [6 5]] [[66 55]
 [66 55]]
[[6 6]
 [5 5]] [[66 66]
 [55 55]]
[[7 4]
 [7 4]] [[77 44]
 [77 44]]
[[7 7]
 [4 4]] [[77 77]
 [44 44]]
[[8 3]
```

```
 [8 3]] [[88 33]
 [88 33]]
[[8 4]
 [6 3]] [[88 44]
 [66 33]]
[[8 6]
 [4 3]] [[88 66]
 [44 33]]
[[8 8]
 [3 3]] [[88 88]
 [33 33]]
[[9 2]
 [9 2]] [[99 22]
 [99 22]]
[[9 3]
 [6 2]] [[99 33]
 [66 22]]
[[9 6]
 [3 2]] [[99 66]
 [33 22]]
[[9 9]
 [2 2]] [[99 99]
 [22 22]]
[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, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11
```

In [29]:
```python
def all_vampires(n,k):
    Vampires = []
    S = list(range(k+1))
    L = list_of_matrices(n,n,S)
    for M in L:
        if is_vampire(M):
            Vampires.append(M)
    return Vampires

print(len(all_vampires(3,4)))
```

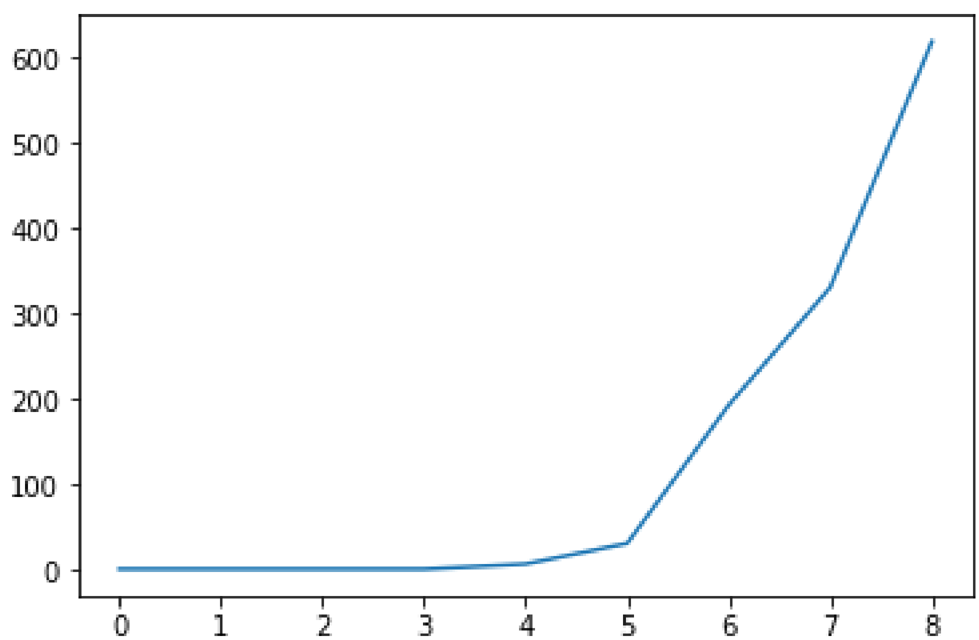
```
7
```

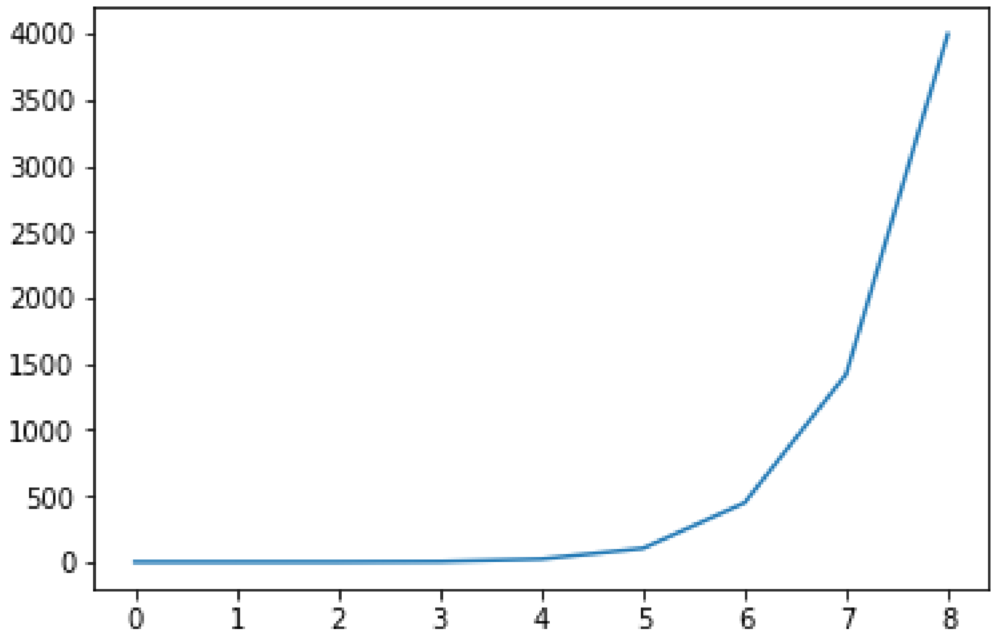
In [36]:
```python
from time import time

print(time())

t1 = time()
V = all_vampires(3,4)
t2 = time()
```

```
    print(len(V),t2-t1)

    L1V = []
    Ltime = []
    for k in range(9):
        t1 = time()
        V = all_vampires(3,k)
        t2 = time()
        L1V.append(len(V))
        Ltime.append(t2-t1)
        print(k,len(V),t2-t1)
```

```
1633895650.8378658
0 1 0.0005712509155273438
1 1 0.020522356033325195
2 1 0.2421119213104248
3 1 3.0042061805725098
4 7 22.243472576141357
5 31 103.05524635314941
6 193 448.24020886421204
7 331 1422.2847096920013
8 619 3996.1434228420258
```

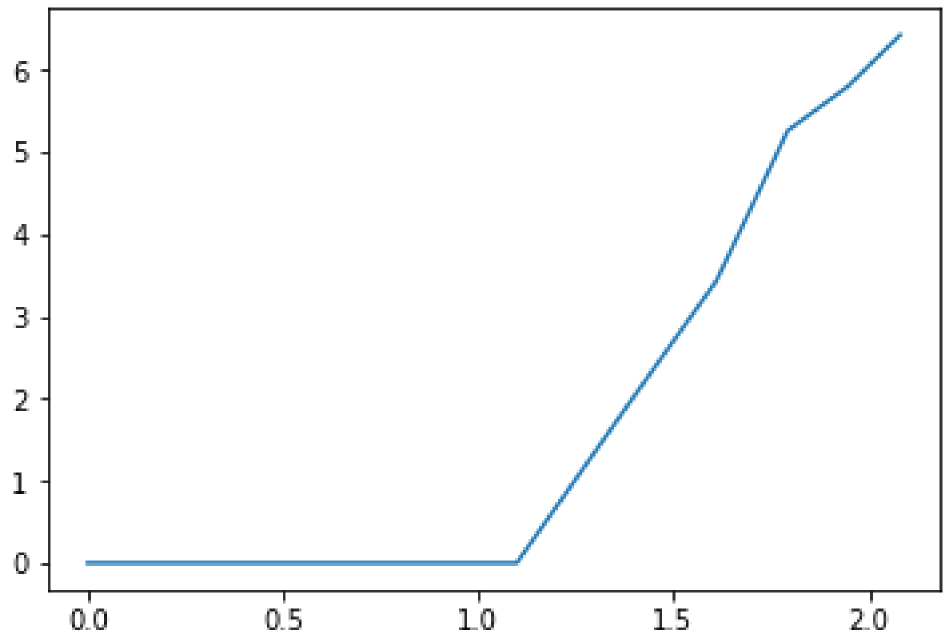
In [37]:
```
plt.plot(range(len(L1V)),L1V)
plt.show()
plt.plot(range(len(Ltime)),Ltime)
plt.show()
```



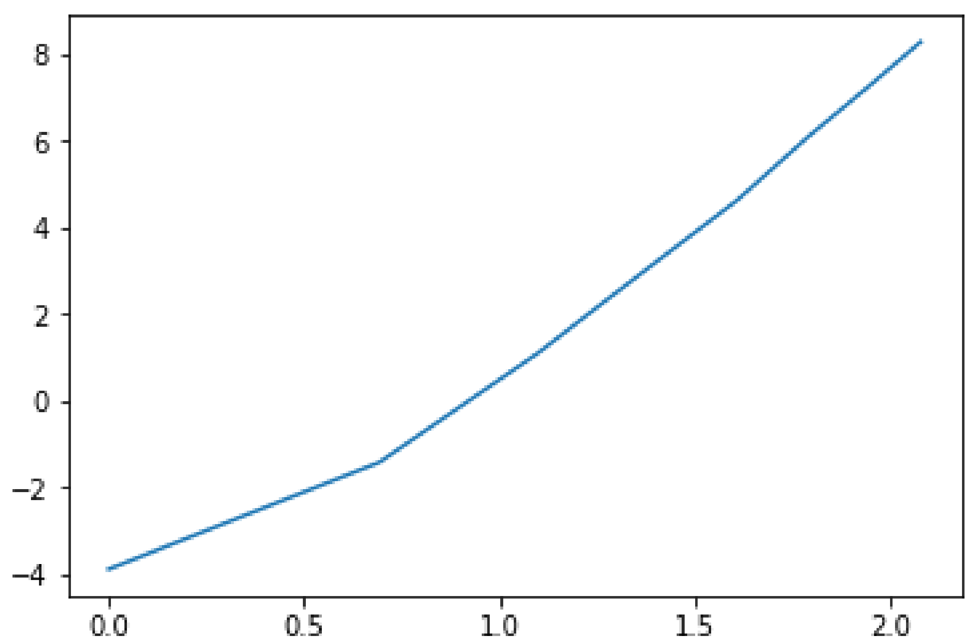



```
In [78]: plt.plot(numpy.log(numpy.array(range(len(L1V)))),numpy.log(numpy.array(L1V)))
         plt.show()
         plt.plot(numpy.log(numpy.array(range(len(Ltime)))),numpy.log(numpy.array(Ltime)))
         plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: RuntimeWarning: divide by zero
  """Entry point for launching an IPython kernel.
```



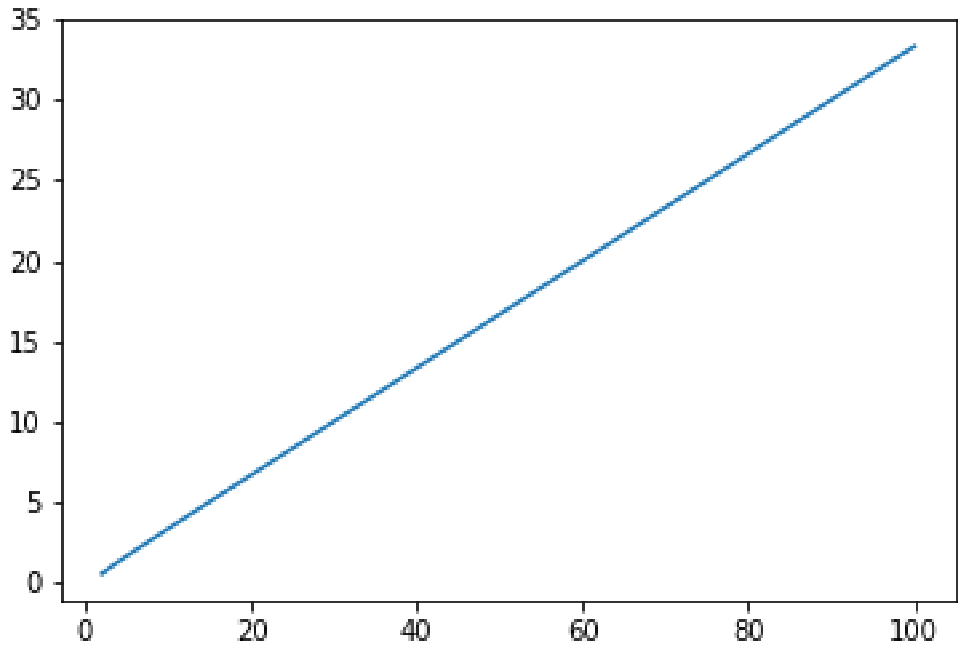

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by zero
  This is separate from the ipykernel package so we can avoid doing imports until
```

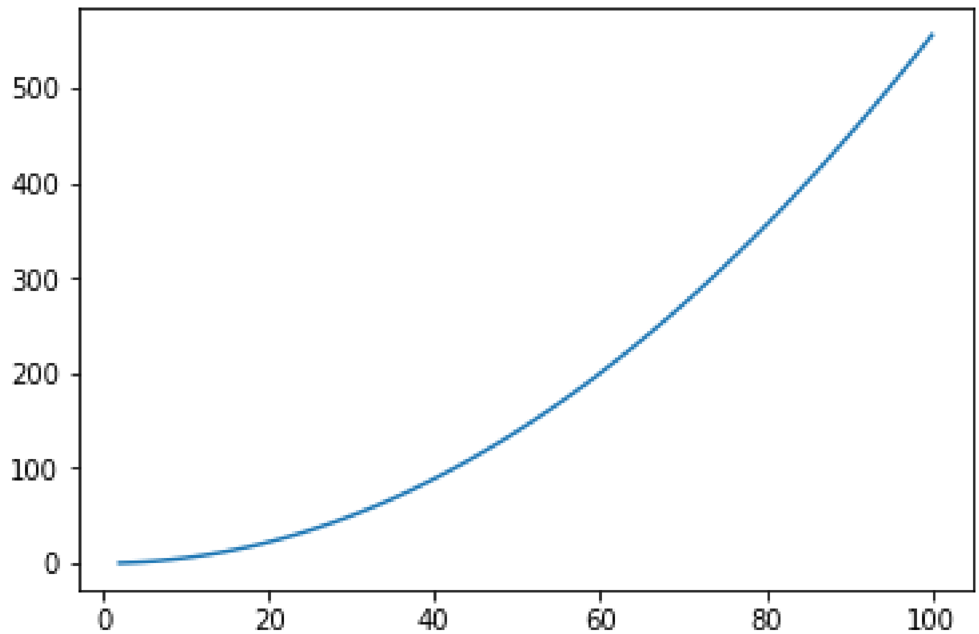


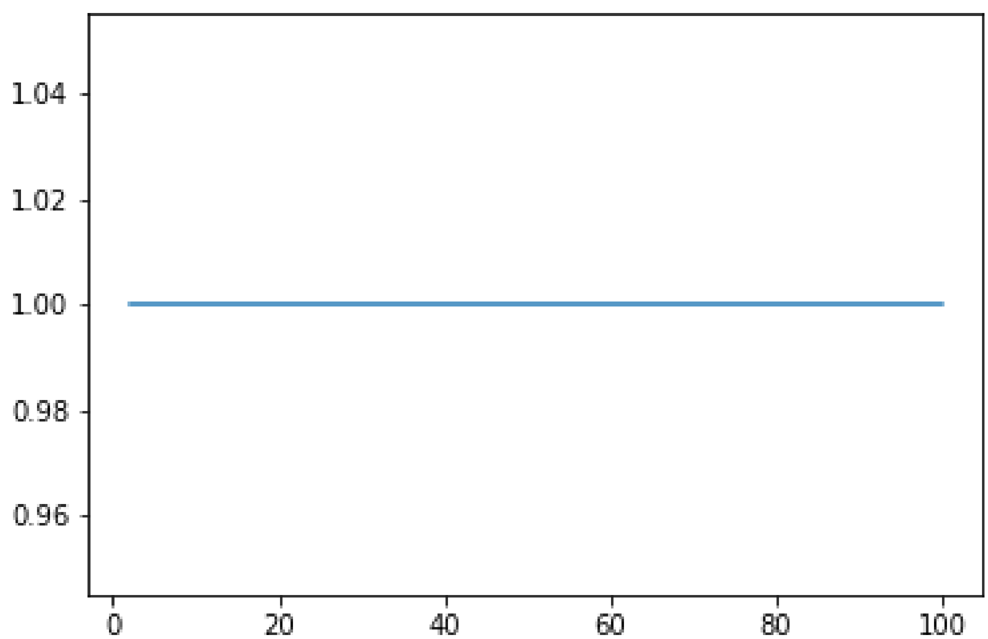
```
In [77]: D = {k : [(i,j) for i in range(1,7) for j in range(1,7) if abs(i-j) == k] for k in ra
         print(D)

         def loi_de_la_difference(n):
             D = {k : 0 for k in range(n)}
             for i in range(1,n+1):
                 for j in range(1,n+1):
                     D[abs(i-j)] += 1
             return D

         def somme(L):
             s = 0
             for a in L.values():
                 s += a
             return s

         def esperance(L):
             e = 0
             for x in L.keys():
                 e += x*L[x]
             return e/somme(L)

         def variance(L):
             e = 0
             for x in L.keys():
                 e += (x**2)*L[x]
             return e/somme(L) - esperance(L)**2

         def mode(L):
             y, m = 0, 0
             for x in L.keys():
                 if L[x] >= m:
                     y = copy(x)
                     m = L[x]
             return y

         N = list(range(2,101))
         E, V, M = [], [], []
         for n in N:
             L = loi_de_la_difference(n)
             E.append(esperance(L))
             V.append(variance(L))
             M.append(mode(L))

         plt.plot(N,E)
         plt.show()
         plt.plot(N,V)
         plt.show()
```

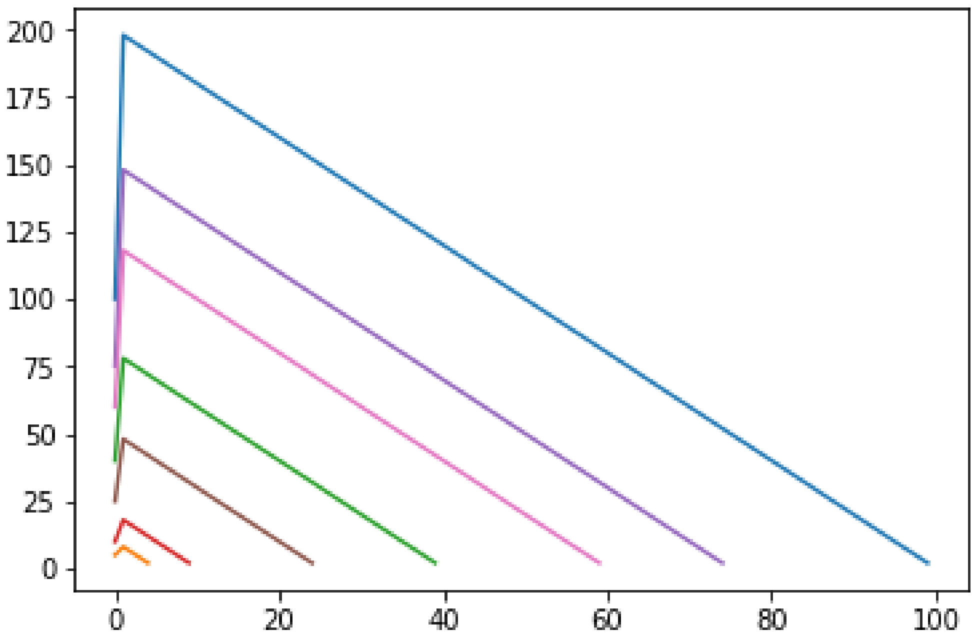
```
plt.plot(N,M)
plt.show()
for n in {5,10,25,40,60,75,100}:
    L = loi_de_la_difference(n)
    plt.plot(L.keys(),L.values())
plt.show()
```

```
{0: [(1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)], 1: [(1, 2), (2, 1), (2, 3), (3, 2), (3, 4
```

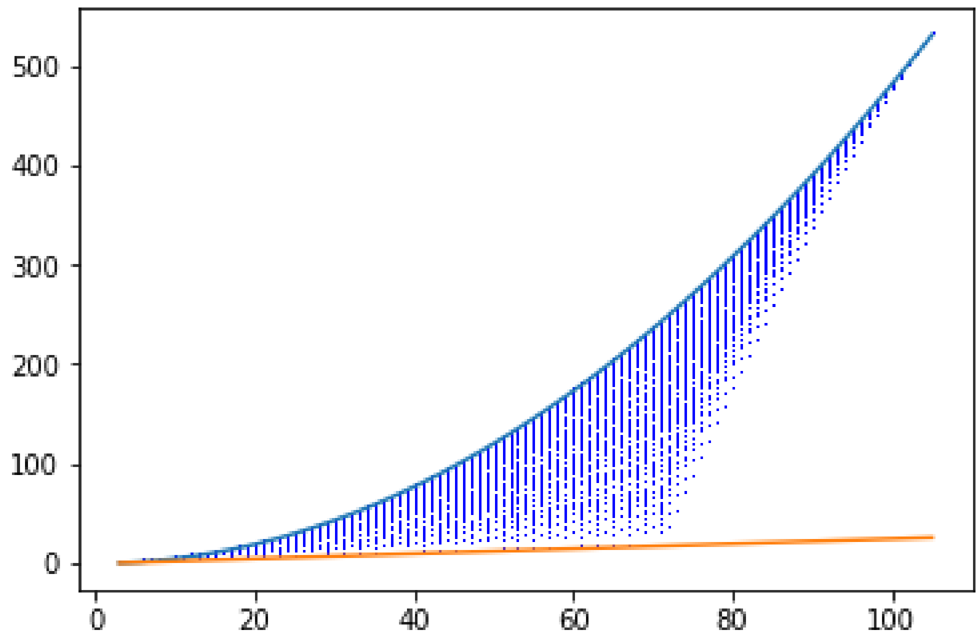



500
400
300
200
100
0
0
20
40
60
80
100


1.04
1.02
1.00
0.98
0.96
0
20
40
60
80
100



In [75]:
```
N = 35
Peri = []
Aire = []
for a in range(1,N+1):
    for b in range(1,N+1):
        for c in range(1,N+1):
            if 2*max(a,b,c) < a+b+c:
                Peri.append(a+b+c)
                s = (a+b+c)/2
                Aire.append(sqrt(s*(s-a)*(s-b)*(s-c)))

plt.plot(Peri,Aire,'b,')
plt.plot()
L = linspace(3,3*N,500)
plt.plot(L,numpy.sqrt(3)/36 * L**2)
plt.plot(L,1/4*numpy.sqrt((L-1)**2-1))
plt.show()
```


500
400
300
200
100
0
0
20
40
60
80
100