Number Representation

Integers

Integer numbers are represented by N bit words. Python3 allows you to store integers with practically **unlimited precision**, the only limitation comes from the (contiguous) space available in memory. In Python2, N depends on the PC architercture, N=64 in modern computers.

In [13]:
```
import sys

# size of an int "0"
a = 0
print(sys.getsizeof(a))

# size of an int "100"
a = 100
print(sys.getsizeof(a))

# size of an int "2**64"
a = 2**64
print(sys.getsizeof(a))
```

```
24
28
36
```

In [14]:
```
# Check the largest integer
print(sys.maxsize)

# Check also that corresponds to a 64-bit integer
print("Is your system a 64 bit one?", 2**63 - 1 == sys.maxsize)

# Python3 doesn't have a limit for integers
maxint = sys.maxsize+1
print(maxint)
```

```
9223372036854775807
Is your system a 64 bit one? True
9223372036854775808
```

Binary representation

The common assumption is that numbers (in Python as in all the other languages) are expressed as decimal numbers. Built-in functions allows explicitly to convert from one base to another.

In the binary representation, typically 1 bit (*j*) is used to specify the sign of the number, and the conversion between binary and decimal representation is:

$$d = (-1)^j \sum_{i=0}^{N-1} \alpha_i \, 2^i$$

where α_i are either 0 or 1. $b = \alpha_{N-1}\alpha_{N-2}\ldots\alpha_0$ is the binary representation of the number.

Example: an 8-bit integer in binary representation with one bit for the sign:

j	6	5	4	3	2	1	0
0	0	0	1	0	1	1	1

corresponds to:

Hexadecimal representation

When dealing with long binary numbers, it's convenient to group bits by groups of **four** and convert them to the hexadecimanl (hex) representation (base 16). In hex base, numbers are represented by a digit from 0-F.

hex	bin
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Example: colors are commonly saved using 24-bits, for example:

```
10001010 10111001 11100011
```

The three bytes (8-bits blocks) specify the red, green, and blue components (RGB). Each one is composed of two blocks of 4 bits, i.e. an hex digit. The same 24-bit word is thus equivalent to:

```
#8AB9E3
```

which is indeed referred to as "Hex color".

Switching between dec, bin and hex representations in Python is straightforward using the casting functions:

In [15]:
```
# an integer in decimal representation
a = 23

# its binary representation
a_bin = bin(a)
print('Binary representation of', a, ':', a_bin)

# its hexadecimal representation
a_hex = hex(a)
print('Hexadecimal representation of', a, ':', a_hex)

# converting back to integer
print('Decimal representation of', a_bin, ':', int(a_bin, 2))
print('Decimal representation of', a_hex, ':', int(a_hex, 16))
```

```
Binary representation of 23 : 0b10111
Hexadecimal representation of 23 : 0x17
Decimal representation of 0b10111 : 23
Decimal representation of 0x17 : 23
```

Bitwise operators

Logical operators

In [16]:
```
a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101

c = a & b         # 12 = 0000 1100
print("Bitwise AND ", c)

c = a | b         # 61 = 0011 1101
print("Bitwise OR  ", c)

c = a ^ b         # 49 = 0011 0001
print("Bitwise XOR ", c)
```

```
Bitwise AND  12
Bitwise OR   61
Bitwise XOR  49
```

Unary operator

The `~` operator returns the complement of a number, which is the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as `-x - 1`

In [17]:
```
c = ~a           # -61 = 1100 0011
print("Bitwise NOT ", c)
```

```
Bitwise NOT  -61
```

Shifts

In [18]:
```
c = a << 2       # 240 = 1111 0000
print("Left shift (towards most significant) of two positions ", c)

c = a >> 2       # 15 = 0000 1111
print("Right shift (towards least significant) of two positions ", c
```

```
Left shift (towards most significant) of two positions  240
Right shift (towards least significant) of two positions  15
```

Link to the Python documentation (https://realpython.com/python-bitwise-operators/).

Masking and shifting

Logical operators and shifts can be combined to "read" a bitstream, by filtering long binary words.

Example: a 8-bit word is used to store 3 integer values. Starting from the leftmost bit:

- the first number uses 1 bit
- the second number uses 4 bits
- the third number the remaining 3 bits. We want to extract these 3 values from the initial word.

bit number	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
first integer	X	|	|	|				
second integer	| X	X	X	X	|			
third integer	|	|	| X	X	X			
binary word (206)	1	1	0	0	1	1	1	0

- first number (1 bit): first, we create a mask (an appropriate binary word) that filters only the information of the desired bit, and sets to 0 everything else

binary word (206)	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**0**
mask (128)	1	0	0	0	0	0	0	0
masked word (128)	1	0	0	0	0	0	0	0

then, we shift the resulting word by the appropriate amount of bits, in this case:

masked word (128)	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
masked and shifted (1)	0	0	0	0	0	0	0	1

The Python code to perform these operations is:

In [19]:
```
word = 206
first_mask = 128
first_shift = 7
first_number = (word & first_mask) >> first_shift
print(first_number)
```

```
1
```

- second number (4 bits):

  | bit number | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |---|---|---|---|---|---|---|---|---| | binary word (206) | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | | mask (120)| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | | masked word (64) | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | | masked and shifted (9) | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |

In [20]:
```
second_mask = 120
second_shift = 3
second_number = (word & second_mask) >> second_shift
print(second_number)
```

```
9
```

Floating point numbers

Non-integer number **cannot be represented with infinite precision** on a computer. Single precision (also known as *float*) and double precision numbers use 32 and 64 bits respectively. Note that all floating point numbers in Python are double precision (64 bits).

A standard has been developed by IEEE (IEEE-754) to represent floating point numbers in hardware such that the relative precision (see later) is the same across the whole validity range.

The 32 or 64 bits are divided among 3 quantities uniquely characterizing the number:

$$x_{float} = (-1)^s \times 1.f \times 2^{e-bias}$$

where:

- *s* is the sign
- *f* the fractional part of the mantissa
- *e* the exponent.

In addition, in order to get numbers smaller than 1, a constant *bias* term is added to the exponent. Such *bias* is typically equal to **half of the max value of *e***.

The mantissa is defined as:

$$\text{mantissa} = 1.f = 1 + \frac{m_{n-1}}{2^1} + \frac{m_{n-2}}{2^2} + . . + \frac{m_0}{2^n}$$

where n is the number of bits dedicated to *f* (see below) and m_i are the binary coefficients.

Numbers exceeding the maximum allowed value are *overflows* and the calculations involving them provide incorrect answers. Numbers smaller in absolute value than the minimum allowed value are *underflows* and simply set to zero, also in this case incorrect

results are yielded.

Single precision

For single precision floating point numbers, $0 \leq e \leq 255$ and $bias = 127$. Bits are arranged as follows:

	*s** / **e*	*f*
Number of bits	1	8	23
Bit position	31	30-23	22-0

An example is given below:

In [21]:
```
from IPython.display import Image
Image(url='http://www.dspguide.com/graphics/F_4_2.gif')
```

Out[21]:

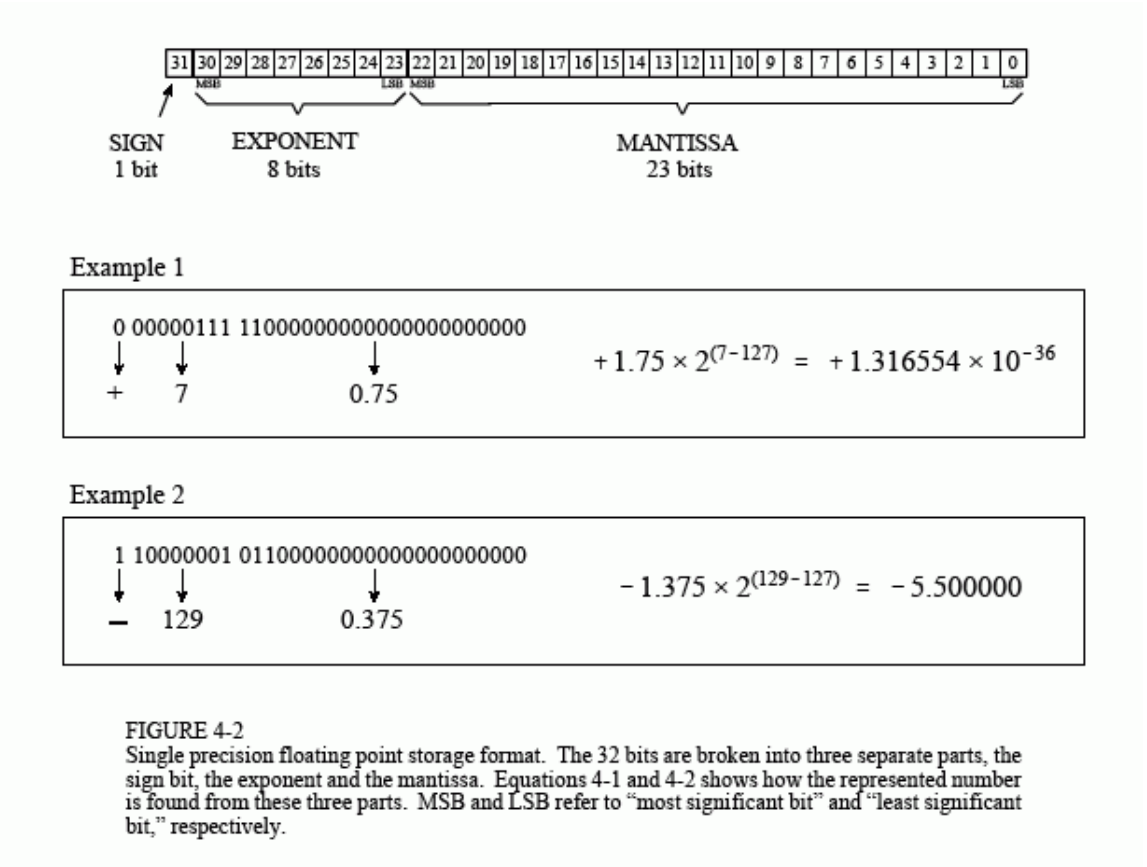

FIGURE 4-2
Single precision floating point storage format. The 32 bits are broken into three separate parts, the sign bit, the exponent and the mantissa. Equations 4-1 and 4-2 shows how the represented number is found from these three parts. MSB and LSB refer to "most significant bit" and "least significant bit," respectively.

Special values are also possibiles. N.B.: those are not numbers that can be used in the mathematical sense!

special value	conditions	value
$+\infty$	s=0, e=255, f=0	+INF
$-\infty$	s=1, e=255, f=0	-INF
not a number	e=255, f>0	NaN

The largest value is obtained for $f \sim 2$ and $e = 254$, i.e. $2 \times 2^{127} \sim 3.4 \times 10^{38}$.

The value closest to zero is obtained instead for $f = 2^{-23}$ and $e = 0$, i.e. $2^{-149} \sim 1.4 \times 10^{-45}$.

Double precision

For double precision floating point numbers, $0 \leq e \leq 2047$ and $bias = 1023$. Bits are arranged as follows:

	*s** / **e*	*f*
Number of bits	1	11	52
Bit position	63	62-52	51-0

Special values are also possibiles. N.B.: those are not numbers that can be used in the mathematical sense!

special value	conditions	value
$+\infty$	s=0, e=2047, f=0	+INF
$-\infty$	s=1, e=2047, f=0	-INF
not a number	e=2047, f>0	NaN

The validity range for double numbers is from 2.2×10^{-308} up to 1.8×10^{308}

Serious scientific calculations almost always requires at least double precision floating point numbers.

Floating point numbers on your system

Information about the floating point reresentation on your system can be obtained from `sys.float_info` . Definitions of the stored values are given on the Python doc page (https://docs.python.org/3/library/sys.html#sys.float_info).

In [22]:
```
import sys
print(sys.float_info)
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Accuracy and the perils of calculations with floats

Floats can only have a limited number of meaningful decimal places, on the basis of how many bits are allocated for the fractional part of the mantissa: 6-7 decimal places for singles, 15-16 for doubles.

This means that calculations involving numbers with more than those decimal places involved do not yield the correct result, simply because the binary representation of those numbers does not allow to store them with sufficient accuracy.

In [23]:
```
# Addding an increasingly small number to 7
for e in [14, 15, 16]:
    print(7 + 1.0 * 10**-e)
```

```
7.00000000000001
7.000000000000001
7.0
```

There are ways to print floats (e.g. filling data into an output file) controlling the number of decimals:

In [24]:
```
print(format(math.pi, '.13f'))  # give 13 significant digits

print('%.14f' % (0.1 * 0.1 * 100)) # give <15 significant digits
print('%.17f' % (0.1 * 0.1 * 100)) # give >15 significant digits
```

```
---------------------------------------------------------------------
--------
NameError                                 Traceback (most recent ca
ll last)
<ipython-input-24-bc3d599324c8> in <module>
----> 1 print(format(math.pi, '.13f'))  # give 13 significant digit
s
      2
      3 print('%.14f' % (0.1 * 0.1 * 100)) # give <15 significant d
igits
      4 print('%.17f' % (0.1 * 0.1 * 100)) # give >15 significant d
igits

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

This is not a bug, but the direct consequence to the fact that the mantissa is represented by a limited amount of bits, therefore calculations can only make sense if an appropriate number of decimal digits are concerned:

In [25]:
```
# 23 bits are used for f in single precision floating point
print("Single precision:", 2**-23)

# 53 bits are used for f in double precision floating point
print("Double precision:", 2**-53)
```

```
Single precision: 1.1920928955078125e-07
Double precision: 1.1102230246251565e-16
```

Operations with floats

It should never been forgotten that computers store numbers in binary format. In the same way it is not possible to express the fraction 1/3 with a finite decimal places, analogously fractions that are well represented in the decimal base cannot be easily represented in binary, e.g. 1/10 is the infinitely repeating number:

$$0.0001100110011001100110011001100110011001100110011001100110011...$$

corresponding to $3602879701896397/2^{55}$ which is close to but not exactly equal to the true value of 1/10 (even though it is even printed to be like that!). Similarly 0.1 is not 1/10, and making calculations assuming that exactly typically yield to wrong results:

In [26]:
```
# sometimes, trivial (for humans) operations can yield unexpected re
print(0.1 + 0.1 == 0.2)

# does it work for 0.3, too?
print(0.1 + 0.1 + 0.1 == 0.3)
```

```
True
False
```

A lesson of paramount importance is that you must **never** compare floating point numbers with the "==" operator as *what is printed is not what is stored*!

The function `float.hex()` yield the exact value stored for a floating point number:

In [27]:
```
import math
x = math.pi
print("dec =", x)
print("hex =", x.hex())

# from the previous example: the two numbers are not the same, bit-w
print((0.1 + 0.1 + 0.1).hex(), (0.3).hex())
```

```
dec = 3.141592653589793
hex = 0x1.921fb54442d18p+1
0x1.3333333333334p-2 0x1.3333333333333p-2
```

For finite floating-point numbers, this representation will always include a leading `0x` and a trailing `p` and exponent. Since Python's floats are stored internally as binary numbers, converting a float to or from a decimal string usually involves a small rounding error.

In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

A typical case is subtraction of numbers very close by in value (e.g. when dealing with spectral frequencies). The same happens with functions evaluated near critical points (see later):

In [28]:
```
print(1 + 6.022e23 - 6.022e23)
```

```
0.0
```

In these cases, associative law does not necessarily hold:

In [29]:
```
print(6.022e23 - 6.022e23 + 1)
print(1 + 6.022e23 - 6.022e23)
```

```
1.0
0.0
```

Distributive law does not hold:

In [30]:
```
import math
a = math.exp(1)
b = math.pi
c = math.sin(1)
a*(b + c) == a*b + a*c
```

Out[30]:
```
False
```

Also identities after casting large numbers may not yield the expected result

In [31]:
```
x = 287475839859383374
print(x == int(float(x)))
```

```
False
```

From numbers to functions: conditioning and stability

Function conditioning

A mathematical function $f(x)$ is well-conditioned if $f(x + \epsilon) \simeq f(x)$ for all small perturbations ϵ.

In other words, the function $f(x)$ is **well-conditioned** if the solution varies gradually as the input varies. For a well-conditioned function, small pertubations in the input result in small effects in the output. However, a poorly-conditioned problem only needs some small perturbation to have large effects. For example, inverting a nearly singluar matrix (a matrix whose determinant is close to zero) is a poorly conditioned problem.

Algorithm stability

Suppose we have a computer algorithm $g(x)$ that implements the mathematical function $f(x)$. $g(x)$ is **numerically stable** if $g(x + \epsilon) \simeq f(x)$ and it is called **unstable** if large changes in the output are produced.

Analyzing an algorithm for stability is more complicated than determining the condition of an expression, even if the algorithm simply evaluates the expression. This is because an algorithm consists of many basic calculations and each one must be analyzed and, due to roundoff error, we must consider the possibility of small errors being introduced in every computed value.

Numerically unstable algorithms tend to amplify approximation errors due to computer arithmetic over time. If we used an infinite precision numerical system, stable and unstable alorithms would have the same accuracy. However, as we see below (e.g. variance calculation), when using floating point numbers, algebrically equivalent algorithms can give different results.

In general, we need both a well-conditioned problem and an algorihtm with sufficient numerical stabilty to obtain reliably accurate answers. In this case, we can be sure that $g(x) \simeq f(x)$.

In most of the cases, the solution to stability issues is solved by properly redefining the function as in the example above and below.

In summary:

- Well-/ill-conditioned refers to the problem; Stable/Unstable refers to an algorithm or numerical process.
- If the problem is well-conditioned, then there is a stable way to solve it.
- If the problem is ill-conditioned, then there is no reliable way to solve it in a stable way.
- Mixing roundoff-error with an unstable process is a recipe for disaster.
- With exact arithmetic (no roundoff-error), stability is not a concern.

1. Example of a poorly conditioned function: the tangent of an angle

In [32]:
```
import math
# Define two numbers x and x + epsilon very close to pi/2
x1 = 1.57078
x2 = 1.57079
# Calculate the tangent of the x1 and x2 angles
t1 = math.tan(x1)
t2 = math.tan(x2)

print ('tan(x1) =', t1)
print ('tan(x2) =', t2)
print ('% change in x =', 100.0*(x2-x1)/x1, '%')
print ('% change in tan(x) =', (100.0*(t2-t1)/t1), '%')
```

```
tan(x1) = 61249.008531503045
tan(x2) = 158057.9134162482
% change in x = 0.0006366263894271296 %
% change in tan(x) = 158.05791343536947 %
```

2. Example of a numerically unstable algorithm: the limit $\lim_{x \to 0} \frac{1-\cos(x)}{x^2}$

In [33]:
```
# Catastrophic cancellation occurs when subtracitng
# two numbers that are very close to one another

# We'll see numpy and matplotlib in the next lectures: forget about
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return (1 - np.cos(x))/(x*x)

#x = np.linspace(-4e-1, 4e-1, 1000) # uncomment to zoom out
x = np.linspace(-4e-8, 4e-8, 1000)
plt.plot(x, f(x))
plt.axvline(1.1e-8, color='red')
```

```
---------------------------------------------------------------------
--------
ModuleNotFoundError                       Traceback (most recent ca
ll last)
<ipython-input-33-6a8c1861ad73> in <module>
      3
      4 # We'll see numpy and matplotlib in the next lectures: forg
et about the technical details, for now
----> 5 import numpy as np
      6 import matplotlib.pyplot as plt
      7

ModuleNotFoundError: No module named 'numpy'
```

In [34]:
```
# We know from L'Hopital's rule that the answer is 0.5 at 0
# and should be very close to 0.5 throughout this tiny interval
# but errors arisee due to catastrophic cancellation

print('%.30f' % np.cos(1.1e-8))
print('%.30f' % (1 - np.cos(1.1e-8))) # failure point: the exact ans
print('%2f' % ((1 - np.cos(1.1e-8))/(1.1e-8*1.1e-8)))
```

```
---------------------------------------------------------------------
--------
NameError                                 Traceback (most recent ca
ll last)
<ipython-input-34-859804d6edf0> in <module>
      3 # but errors arisee due to catastrophic cancellation
      4
----> 5 print('%.30f' % np.cos(1.1e-8))
      6 print('%.30f' % (1 - np.cos(1.1e-8))) # failure point: the
exact answer is 6.05e-17
      7 print('%2f' % ((1 - np.cos(1.1e-8))/(1.1e-8*1.1e-8)))

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

Solution: rewrite the function using $\sin$ instead of $\cos$: $1 - \cos(x) = 2\sin^2(\frac{x}{2})$

In [35]:
```
# Numerically stable version of funtion using simple trignometry

def f1(x):
    return 2*np.sin(x/2)**2/(x*x)

#x = np.linspace(-4e-1, 4e-1, 1000) # uncomment to zoom out
x = np.linspace(-4e-8, 4e-8, 1000)
plt.plot(x, f1(x))
plt.axvline(1.1e-8, color='red')
```

```
---------------------------------------------------------------------
--------
NameError                                 Traceback (most recent ca
ll last)
<ipython-input-35-c767742002f0> in <module>
      5
      6 #x = np.linspace(-4e-1, 4e-1, 1000) # uncomment to zoom out
----> 7 x = np.linspace(-4e-8, 4e-8, 1000)
      8 plt.plot(x, f1(x))
      9 plt.axvline(1.1e-8, color='red')

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

3. Another common example of a numerically unstable algorithm. The stable and unstable version of the variance:

$$s^2 = \frac{1}{n-1} \sum (x - \bar{x})^2$$

In [36]:
```
# check the result of the calculation of the variance of an array
# of randomly distributed data between [0, 1] around 1e12
x = 1e12 + np.random.uniform(0, 1, int(1e3))

# direct method
# squaring occuring after subtraction, element-wise
def direct_var(x):
    n = len(x)
    xbar = np.mean(x)
    return 1.0/(n-1)*np.sum((x - xbar)**2)

# sum of squares method (vectorized version)
# pay attention to the subtraction of two large numbers
def sum_of_squares_var(x):
    n = len(x)
    return (1.0/(n*(n-1))*(n*np.sum(x**2) - (np.sum(x))**2))

# Welford's method
# an optimized method
def welford_var(x):
    s = 0
    m = x[0]
    for i in range(1, len(x)):
        m += (x[i] - m) / i
        s += (x[i] - m)**2
    return s/(len(x) - 1)


# correct answer from a purpose-built function in numpy
print("Numpy:", np.var(x))
print("Direct:", direct_var(x))
print("Sum of squares:", sum_of_squares_var(x))
print("Welford's:", welford_var(x))
```

```
---------------------------------------------------------------------
--------
NameError                                 Traceback (most recent ca
ll last)
<ipython-input-36-e5d3de018765> in <module>
      1 # check the result of the calculation of the variance of an
array
      2 # of randomly distributed data between [0, 1] around 1e12
----> 3 x = 1e12 + np.random.uniform(0, 1, int(1e3))
      4
      5 # direct method

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

4. The example of the Likelihood: $\mathcal{L} = \prod_{i=0}^{N} Poisson(x, \mu)$

In [37]:
```
# loss of precision can be a problem when calculating Likelihoods
probs = np.random.random(1000) # Generating 1000 random numbers betw
#print(probs)
print("L =", np.prod(probs))

# when multiplying lots of small numbers, work in log space
print("log L =", np.sum(np.log(probs)))
```

```
---------------------------------------------------------------------
--------
NameError                                 Traceback (most recent ca
ll last)
<ipython-input-37-46a0ab63d2c4> in <module>
      1 # loss of precision can be a problem when calculating Likel
ihoods
----> 2 probs = np.random.random(1000) # Generating 1000 random num
bers between 0 and 1, as if they were probabilities
      3 #print(probs)
      4 print("L =", np.prod(probs))
      5

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

In [ ]:
```
```