Programming for Data Analysis - Assignment

Generation of random numbers using the `numpy.random` package

Table of Contents

1. Introduction

Overall purpose of the package

Random numbers are used widely in several computational science areas, including in:

- statistical sampling: for example, in a quality control test, it is definitely more feasible to test a subset (randomly picked) rather than the entire lot of products.
- image analysis to ensure a particular algorithm is able to distinguish between features and noise
- cryptography so that a secret message, such as your credit card details, can be transmitted securely without anyone else being able to read it
- gaming and gambling:
- bioinformatics - cluster analysis, bootstrapping tests and stochastic simulations of biosystems (http://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf (http://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf))
- and more (https://en.wikipedia.org/wiki/Applications_of_randomness)

Notes: Equations are rendered here using Tex/MathJax (1 (https://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/Typesetting%20Equations.html#Motivating-Examples) and 2 (https://math.meta.stackexchange.com/questions/5020/mathjax-basic-tutorial-and-quick-reference)), with assistance from Detexify (http://detexify.kirelabs.org/classify.html) and Emre Sermutlu's compilation (https://pic.plover.com/MISC/symbols.pdf).

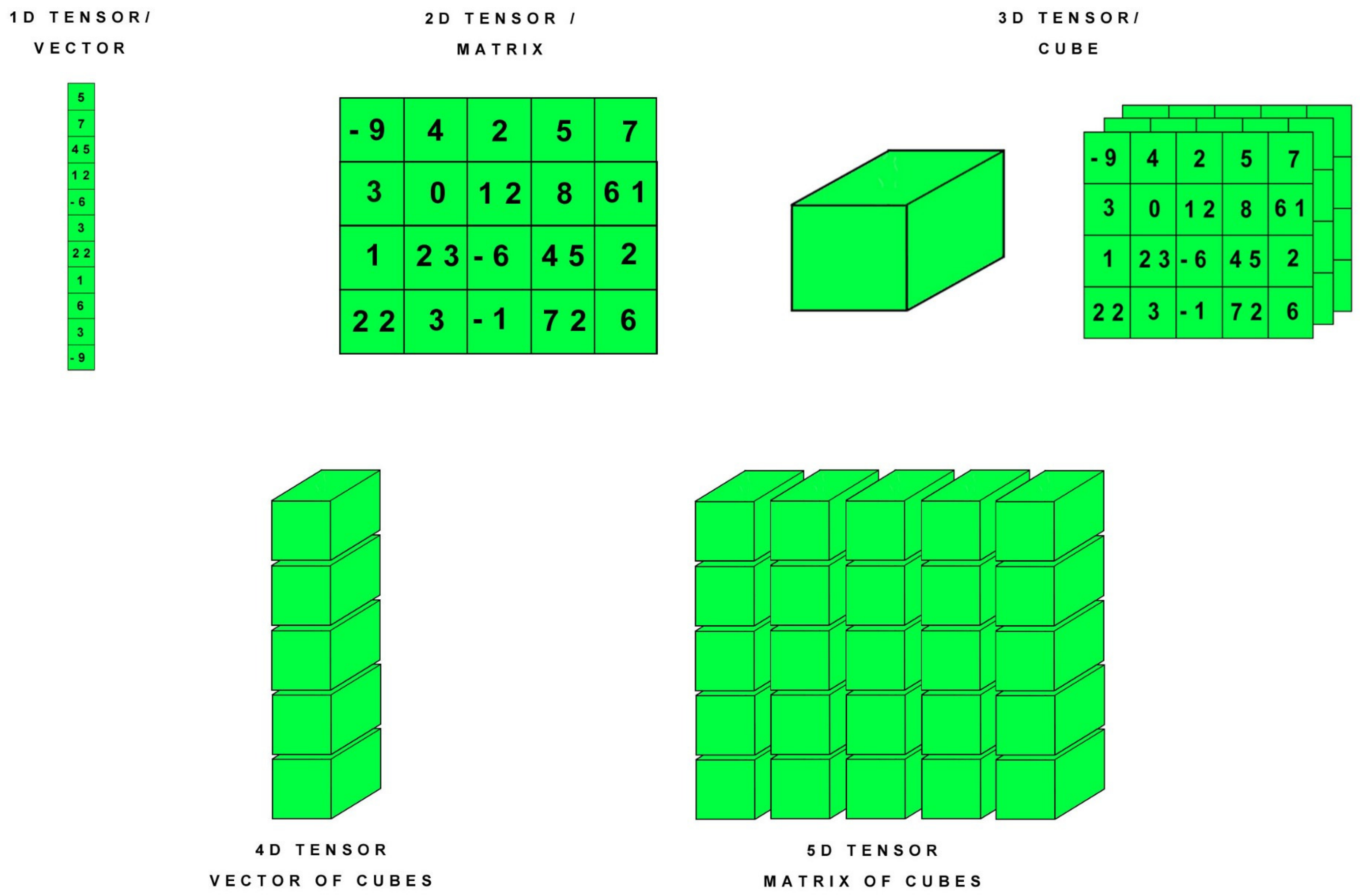
How to generate random numbers? What is the difference between a true random number generator and a pseudo-random generator?

In [1]:

```python
import numpy as np # we are using the numpy.random package for this notebook

# to make interactive plots with plotly
import plotly.graph_objs as go
from plotly.offline import download_plotlyjs, init_notebook_mode, iplot
init_notebook_mode(connected=True)
import plotly.figure_factory as ff
```

Dimensionality of arrays

This graphics below is an excellent way to visualize simple and multidimensional arrays.



Taken from [Hacker Noon (https://hackernoon.com/learning-ai-if-you-suck-at-math-p4-tensors-illustrated-with-cats-27f0002c9b32)](https://hackernoon.com/learning-ai-if-you-suck-at-math-p4-tensors-illustrated-with-cats-27f0002c9b32)

At the very basic level, a 1D array is simply a collection of numbers in a column (or a row). By adding another dimension, *i.e.* a row, we now have a collection of numbers arranged in a column AND a row (think of it as a rectangle). This 2D array is a matrix.

This rectangle can be converted into a cube in the third dimension - the best analogy for this is a collection of rectangles arranged like sheets in a book.

We can also stack a few books together on a bookshelf to create a fourth dimension (4D array). As you will see in a library, our books will be arranged on multiple bookshelves (fifth dimension - 5D array).

This visualization technique allows us to construct even more complex arrays. Actually, if you notice carefully, the output of the `rand()` function is already formatted with a white space between two matrices. (https://hackernoon.com/learning-ai-if-you-suck-at-math-p4-tensors-illustrated-with-cats-27f0002c9b32)

1D array of a single row containing three numbers (https://hackernoon.com/learning-ai-if-you-suck-at-math-p4-tensors-illustrated-with-cats-27f0002c9b32)

In [2]:
```
example_1D_array = np.random.rand(3)
example_1D_array
```

Out[2]:
```
array([0.34000761, 0.32661723, 0.12744482])
```

2D array of 2 rows and 3 columns

In [3]:
```
example_2D_array = np.random.rand(2,3)
example_2D_array
```

Out[3]:
```
array([[0.17763548, 0.29347119, 0.8845309 ],
       [0.28038058, 0.28116757, 0.79259403]])
```

3D array of 2 rows and 3 columns arranged as 4 sheets, making up a cube

Note that the sheets are separated by horizontal white spaces - see the example for 4D array below with an annotated image

In [4]:
```
example_3D_array = np.random.rand(4,2,3)
example_3D_array
```

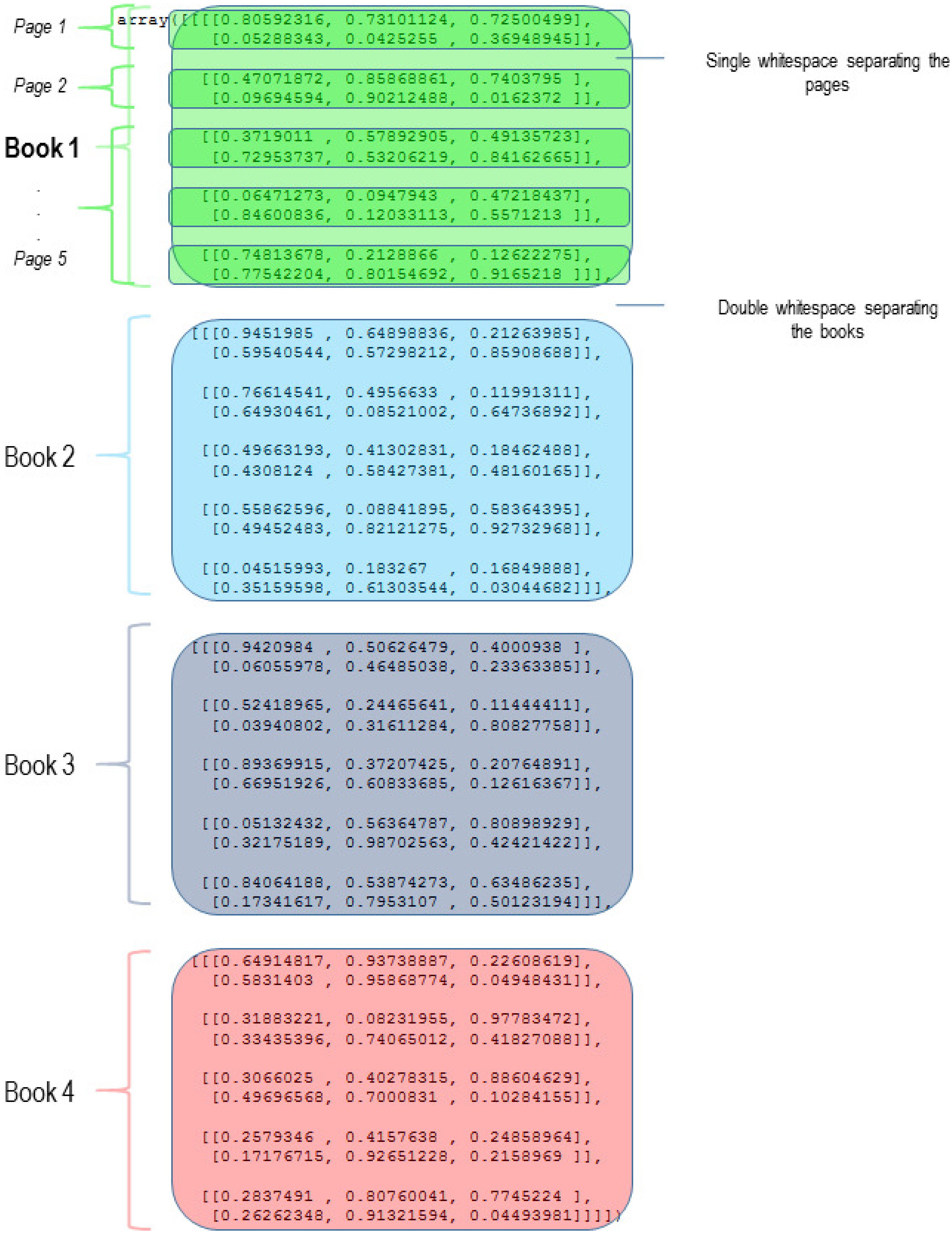
Out[4]:
```
array([[[0.02604053, 0.34671744, 0.99976493],
        [0.80846109, 0.16919613, 0.09458895]],

       [[0.08681357, 0.01850597, 0.38542285],
        [0.40249465, 0.73822171, 0.50395193]],

       [[0.68570057, 0.55777161, 0.14631179],
        [0.35160425, 0.81152279, 0.26291255]],

       [[0.95060043, 0.12119726, 0.06114748],
        [0.58514643, 0.07019437, 0.47641714]]])
```

4D array of 4 books each consisting of 5 sheets of 2 rows and 3 columns

In [5]:
```
example_4D_array = np.random.rand(4,5,2,3)
example_4D_array
```

Out[5]:
```
array([[[[0.57302658, 0.28261316, 0.10238532],
         [0.9694148 , 0.66717577, 0.53597438]],

        [[0.64145337, 0.74941695, 0.27781248],
         [0.62704188, 0.10819654, 0.31763465]],

        [[0.39803217, 0.75878451, 0.68557088],
         [0.45824308, 0.49428422, 0.78832806]],

        [[0.00319669, 0.20639942, 0.65678991],
         [0.3580754 , 0.47060556, 0.47207424]],

        [[0.90967188, 0.81246908, 0.62971708],
         [0.97243127, 0.47488327, 0.02837893]]],


       [[[0.64930916, 0.47567769, 0.44613796],
         [0.28574788, 0.00448234, 0.25784596]],

        [[0.46317817, 0.90183573, 0.09133356],
         [0.27220998, 0.65628914, 0.57876357]],

        [[0.51401481, 0.56586348, 0.10564608],
         [0.48843714, 0.28496641, 0.1814057 ]],

        [[0.25820618, 0.08888796, 0.74410946],
         [0.51671458, 0.95328972, 0.59868112]],

        [[0.1157813 , 0.24456825, 0.21628425],
         [0.66583488, 0.1254239 , 0.33975527]]],


       [[[0.21775932, 0.15191005, 0.13306913],
         [0.39675463, 0.08877802, 0.98629711]],

        [[0.05328281, 0.81029   , 0.5911412 ],
         [0.74054291, 0.34700538, 0.42104074]],

        [[0.85752519, 0.85800118, 0.23788793],
         [0.82661306, 0.05992136, 0.90487446]],

        [[0.32178208, 0.90467704, 0.28452954],
         [0.24537113, 0.22359458, 0.97785228]],

        [[0.41017429, 0.52625642, 0.41599619],
         [0.93221091, 0.80742603, 0.74771081]]],


       [[[0.58553227, 0.64505606, 0.99687412],
         [0.12752665, 0.66894909, 0.9457063 ]],

        [[0.97807582, 0.68100821, 0.79715105],
         [0.71299623, 0.74233523, 0.77232349]],

        [[0.12759139, 0.05586584, 0.11696648],
         [0.47278819, 0.87529799, 0.43073372]],

        [[0.90314153, 0.83307505, 0.3159542 ],
         [0.04155481, 0.32155868, 0.77705146]],

        [[0.06947494, 0.46103015, 0.90844143],
         [0.89515959, 0.18760615, 0.13678829]]]])
```

The array above can be visualized as follows:



Simple Random Data and Permutations functions

Simple Random Data

The corresponding documentations are here: [https://docs.scipy.org/doc/numpy/reference/routines.random.html (https://docs.scipy.org/doc/numpy/reference/routines.random.html)](https://docs.scipy.org/doc/numpy/reference/routines.random.html)

Function	Official documentation	Own interpretation
rand(d0, d1, …, dn)	Random values in a given shape	The values are taken from a uniform distribution over [0, 1). By providing more than one argument, multidimensional arrays can be obtained.
randn(d0, d1, …, dn)	Return a sample (or samples) from the “standard normal” distribution	Similar to the `rand()` function but from a normal distribution
randint(low[, high, size, dtype])	Return random integers from low (inclusive) to high (exclusive)	Similar to the `rand()` function but from a discrete uniform distribution, because this will return only integers (discrete data)
random_integers(low[, high, size])	Random integers of type np.int between low and high, inclusive	This function is almost the same as `randint()` except that both limits are inclusive (closed interval). When calling this function, Python will return `This function is deprecated. Please call randint(10, 20000 + 1) instead`
random_sample([size])	Return random floats in the half-open interval [0.0, 1.0).	Returns random floating-point numbers similar to `rand()` but only takes one argument indicating the number of values to be return. Therefore, it cannot output multidimensional arrays.
random([size])	Return random floats in the half-open interval [0.0, 1.0).	Same as `random_sample()` - see Note1 below
ranf([size])	Return random floats in the half-open interval [0.0, 1.0).	Same as `random_sample()`
sample([size])	Return random floats in the half-open interval [0.0, 1.0).	Same as `random_sample()`
choice(a[, size, replace, p])	Generates a random sample from a given 1-D array	This is an interesting function, as it allows a random selection of a pre-defined number of values from a 1D array. Alternatively, if an integer is passed to the `a` argument, the function will first create an evenly-spaced Numpy array from the `range()` function before randomly selecting those values. Additionally, the `replace` argument can be used as a form of permutation.
bytes(length)	Return random bytes	Self-explanatory

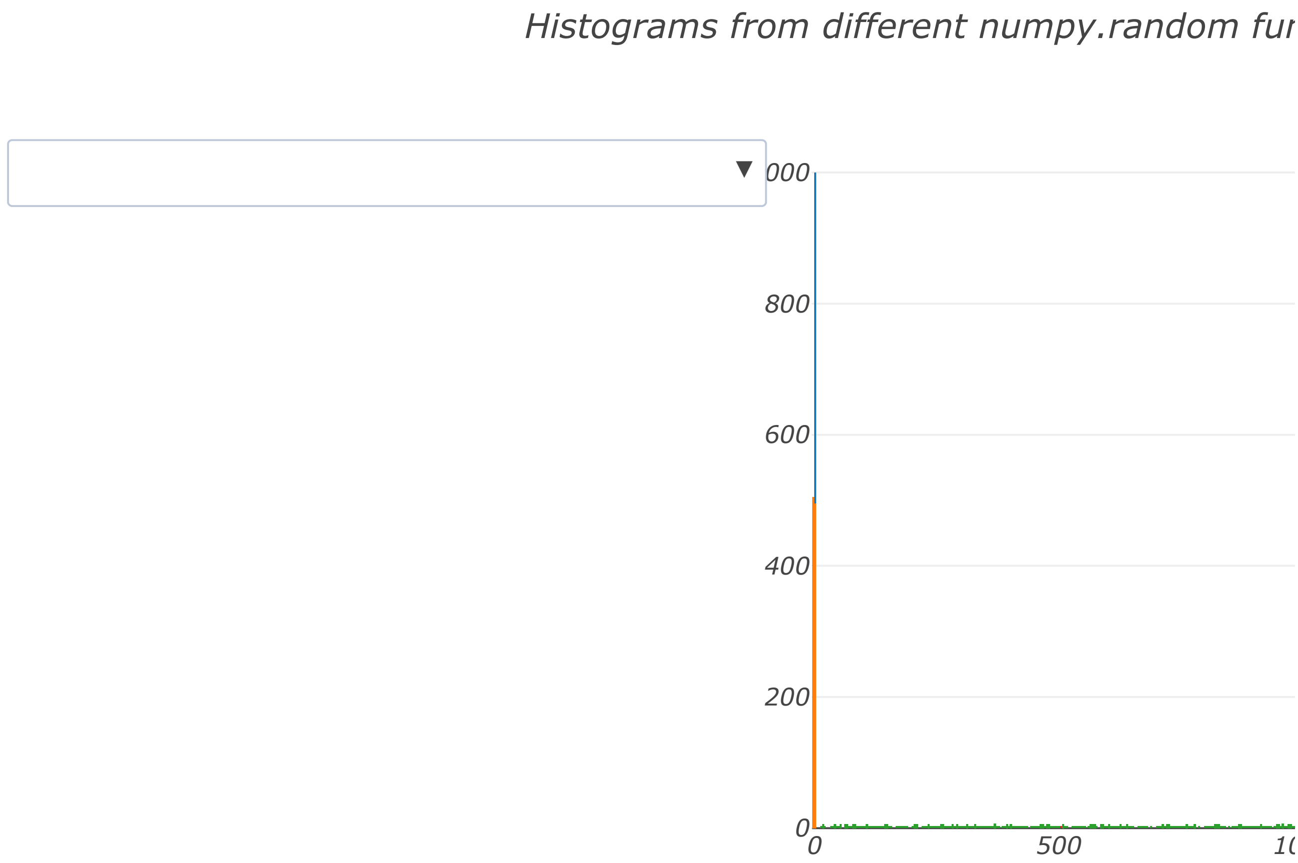
Note1: According to the source code for [`numpy.random` (https://github.com/numpy/numpy/blob/master/numpy/random/__init__.py)](https://github.com/numpy/numpy/blob/master/numpy/random/__init__.py):

> ***Some aliases:***
>
> *ranf = random = sample = random_sample*
> ***all**.extend(['ranf', 'random', 'sample']) </blockquote></i>*

In order to visualize the outputs of the different functions, the code below will create a few variables that can then be examined by plotting the values.

In [6]:
```
# this will return an array containing 1000 random numbers uniformly-distribute
d
output_function_rand = np.random.rand(1000)

# this will return an array containing 1000 random numbers normally-distributed
output_function_randn = np.random.randn(1000)

# returns 1000 random integers between 10 (inclusive) and 20,000 (exclusive)
output_function_randint = np.random.randint(10, 2000, 1000)

# select 5 values randomly from the output_function_randint array with replacem
ent
random_selection_from_output_function_randint = np.random.choice(output_functio
n_randint, 5)

#data = [go.Histogram(x=random_array,)]
#iplot(data)
```

In [7]:
```
# plots with dropdown menu

trace_uniform = go.Histogram(x=output_function_rand)
trace_normal = go.Histogram(x=output_function_randn)
trace_integer = go.Histogram(x=output_function_randint, xbins=dict(start=0, end
= 2000, size=5), autobinx=False )

trace_randomSelection = go.Histogram(x=random_selection_from_output_function_ra
ndint,
                                    xbins=dict(start=np.amin(random_selection_f
rom_output_function_randint),
                                              end = np.amax(random_selection_fr
om_output_function_randint),
                                              size = 5),
                                     autobinx=False)

data = [trace_uniform, trace_normal, trace_integer, trace_randomSelection]

updatemenus = list([
    dict(active=-1,
         buttons=list([
            dict(label = 'rand() function',
                 method = 'update',
                 args = [{'visible': [True, False, False, False]},
                         {'title': 'Random numbers from rand() are uniformly di
stributed'
                          }]),
            dict(label = 'randn() function',
                 method = 'update',
                 args = [{'visible': [False, True, False, False]},
                         {'title': 'Random numbers from randn() are normally di
stributed'
                          }]),
            dict(label = 'randint() function',
                 method = 'update',
                 args = [{'visible': [False, False, True, False]},
                         {'title': 'Random integers from randint() are uniforml
y distributed',
                          }]),
            dict(label = 'choice() function using values generated from randint
()',
                 method = 'update',
                 args = [{'visible': [False, False, False, True]},
                         {'title': 'Distribution of randomly-selected values fr
om a larger set of \
randomly-generated integers',
                          }])
        ]),
    )
])

layout = dict(title='Histograms from different numpy.random functions', showleg
end=False,
              updatemenus=updatemenus)

fig = dict(data=data, layout=layout)
iplot(fig)
```



Permutations

Given a set or an array of numbers, it is possible to arrange the elements within the array in several particular orders.

For instance, the array of [1, 2, 3, 4] can be rearranged as follows:

Note: the code below uses the Sympy library, a Python library for symbolic computing.

In [8]:
```python
from sympy.utilities.iterables import multiset_permutations

for possible_array in multiset_permutations([1,2,3,4]):
    print(possible_array)
```

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

As shown above, the array can be ordered into 24 sets without replacement. It is actually trivial to calculate the number of possible unique sets by considering the following:

1. *If we have an array of 4 elements and we want to select all 4 of them, there are 4 possible options to place in the first position (i.e. we can place 1, 2, 3 or 4 first). Let's assume we placed element '3' in the first position.*

   Resulting array: 3 _
2. *If we were to not replace the element back into the original array, then the number of available options now is only 3 (elements 1, 2, and 4). Assume that we place, in the second position, the element 1.*

   Resulting array: 3 1
3. *Now, we are left with only the elements 2 and 4 (2 options). Let's place '4' in the third position.*

   Resulting array: 3 1 4 _
4. *As we do not have any more elements except for '2', we obtained the result below.*

   Resulting array: 3 1 4 2

In essence, the possible options are 4 x 3 x 2 x 1 = 24, without replacements. In discrete mathematics term, this whole process is called permutation, a subject within combinatorics.

The `choice()` function demonstrated above in the Simple Random Data section, selects a subset of values, with or without replacement, from an array of values. Numpy also has a similar function, `permutation()` to perform permutations. In the `choice()` function, we can specify whether or not to replace the elements, how many elements to be sampled and the probability associated with each element. In contrast, the `permutation()` function is almost like a shuffling function in that it does not take any arguments except for the input sequence/array, as alluded by the official documentation for a particular [example (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.choice.html#numpy.random.choice)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.choice.html#numpy.random.choice):

> # ***This is equivalent to np.random.permutation...***

The official documentation is located [here (https://docs.scipy.org/doc/numpy/reference/routines.random.html#permutations)](https://docs.scipy.org/doc/numpy/reference/routines.random.html#permutations).

Function	***Official documentation***	***Own interpretation***
permutation(x)	*Randomly permute a sequence, or return a permuted range*	*Elements in an array are ordered randomly. If x is an integer, `permutation(x)` first creates an evenly-spaced array with `np.arange(x)` . This function is similar to `choice` - see explanation above*
shuffle(x)	*Modify a sequence in-place by shuffling its contents*	*Similar to `permutation()` , except the sequence is modified in-place whereas the former returns a copy of the original sequence. The input must be an array*

In [9]:
```
# reorder the random_arr array randomly
random_arr = [1, 2, 3, 4]
np.random.permutation(random_arr)
```
Out[9]:
```
array([1, 4, 3, 2])
```

In [10]:
```
# note that random_arr is not modified by the permutation() function
random_arr
```
Out[10]:
```
[1, 2, 3, 4]
```

```
In [11]: # Note, however, that the shuffle() function replaces the original copy of rand
         om_arr
         np.random.shuffle(random_arr)
         random_arr

Out[11]: [3, 1, 2, 4]
```

Use and purpose of selected “Distributions” functions

In this section, the use of beta, exponential, Laplace, Poisson and Rayleigh distributions will be demonstrated along with the practical utility. To this end, we will import the corresponding Numpy and Scipy libraries, with provide convenient functions to generate random numbers that follow different distributions.

In practical terms, modeling a given phenomenon requires fitting the experimental data with a set of known distribution, ideally performed both visually (through graphical plots) and numerically (through the calculation of different statistics, such as R^2, sum of squared errors or Bayesian information criterion). A primer on choosing the correct probability distributions is given in [the work of Jonathan Mun (https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119197096.app03)](https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119197096.app03).

Throughout the examples below, there will be some differences between the standard form the probability distribution function and the form used by Scipy. However, they are mathematically equivalent so long as the [location (loc) (https://en.wikipedia.org/wiki/Location_parameter)](https://en.wikipedia.org/wiki/Location_parameter) and [scale (https://en.wikipedia.org/wiki/Scale_parameter)](https://en.wikipedia.org/wiki/Scale_parameter) parameters are taken into account. The former parameter defines the x value corresponding to the peak of the curve, allowing for right- or left-shifting of the curve. The scale parameter, in contrast, controls the spread or width of the distribution. Note that not all functions have these parameters defined in them.

```
In [12]: from scipy.stats import beta, expon, laplace, poisson, rayleigh
         from ipywidgets import interactive, HBox, VBox
```

Beta distribution

Official documentations are available on the [Scipy (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.beta.html#scipy.stats.beta)](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.beta.html#scipy.stats.beta) and [Numpy (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.beta.html#numpy.random.beta)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.beta.html#numpy.random.beta) reference webpages.

The beta distribution takes two parameters, α and β (both > 0), that together determine the probability density function (PDF) (i.e. the y values / curve shape):

$$f(x, a, b) = \frac{\gamma(a+b) x^{a-1} (1-x)^{b-1}}{\gamma(a)\gamma(b)}$$

for $0 < x < 1, a > 0, b > 0, \gamma(z) = gamma function$*, where the gamma function is:*

$$\mathit{\Gamma}(z) = \int_0^\infty x^{z-1} e^{-x} dx = (z-1)!$$

The beta distribution can be used to model the following cases:

- *Bayesian analysis [Johnson (2016) (https://arxiv.org/pdf/1307.6437.pdf)](https://arxiv.org/pdf/1307.6437.pdf)*
- *population genetics*
- *soil science and bioassays [Gupta & Nadarajah (2004) (https://www.crcpress.com/Handbook-of-Beta-Distribution-and-Its-Applications/Gupta-Nadarajah/p/book/9780824753962)](https://www.crcpress.com/Handbook-of-Beta-Distribution-and-Its-Applications/Gupta-Nadarajah/p/book/9780824753962)*

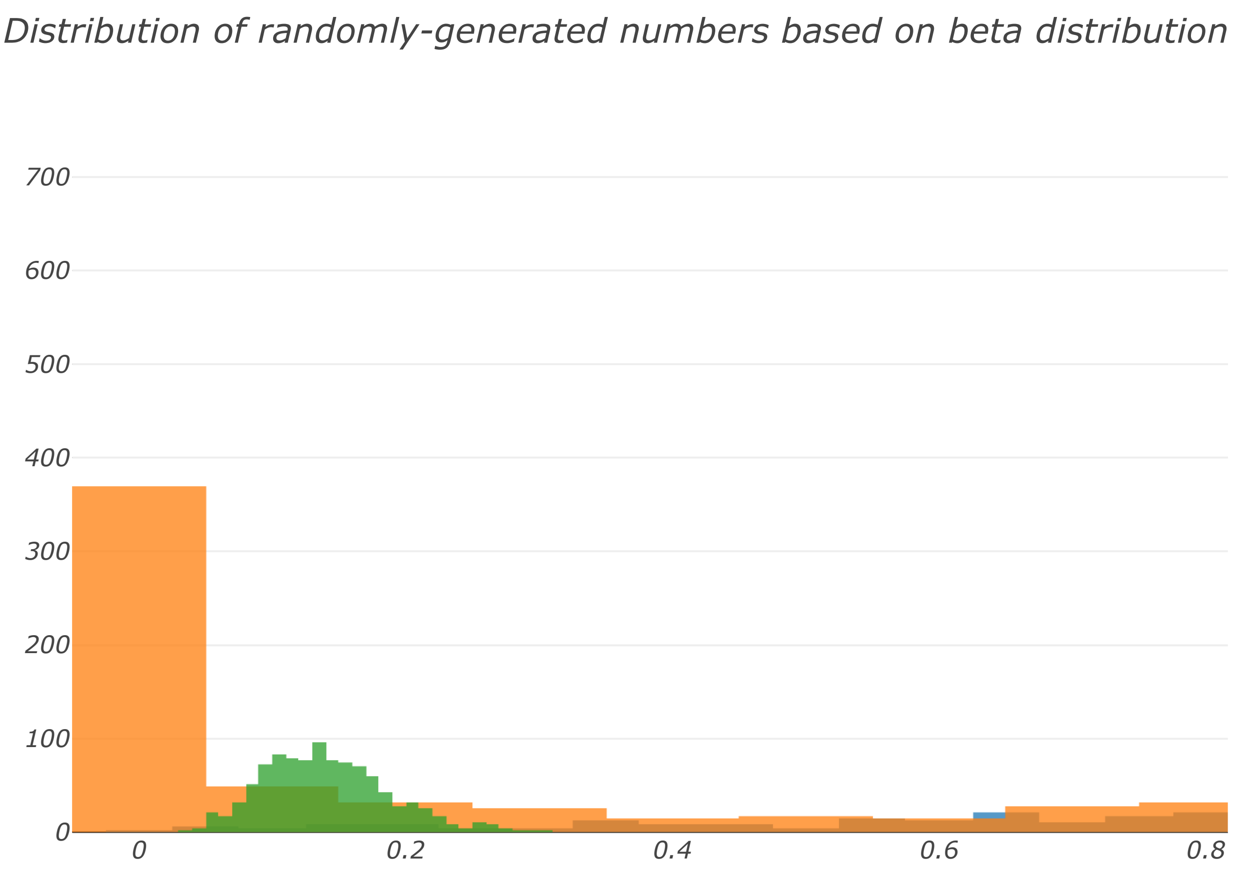
In this section, we will first look at how α and β values determine the shape of the curve, i.e. the PDF, based on the Scipy `beta()` *function. The code below provides an interactive way to explore the change in the PDF for different values of α and β.*

In [13]:
```
a= b= 1 # we need some a and b values for the initial y
x = np.arange(-0.5, 1, 0.01) # let's plot x from -0.5 to 1
y = beta.pdf(x,a,b) # calculate the beta probability density function using the
x, a and b values from above

layout = go.Layout(
    title = 'Beta probability density function',
    xaxis = dict(title='x'),
    yaxis = dict(title='PDF'),
)

# plot y over x as a scatter plot
f = go.FigureWidget(data=[go.Scatter(x=x, y=y)], layout = layout)

# we need to assign the slider to a function - this slider is actually from ipy
widgets and not from plotly
def beta_function(a,b):
    f.data[0].y = beta.pdf(x,a,b)

parameter_slider = interactive(beta_function, a=(0.1, 50, 0.1), b=(0.1, 50, 0.1
))
vb = VBox((f, parameter_slider))
vb.layout.align_items = 'center'
vb
```

Now we know how the PDF of beta distribution is dependent on α and β values, we can ask Numpy to generate a set of 1000 random values with a few different combinations of α and β, using the Numpy `beta` function.

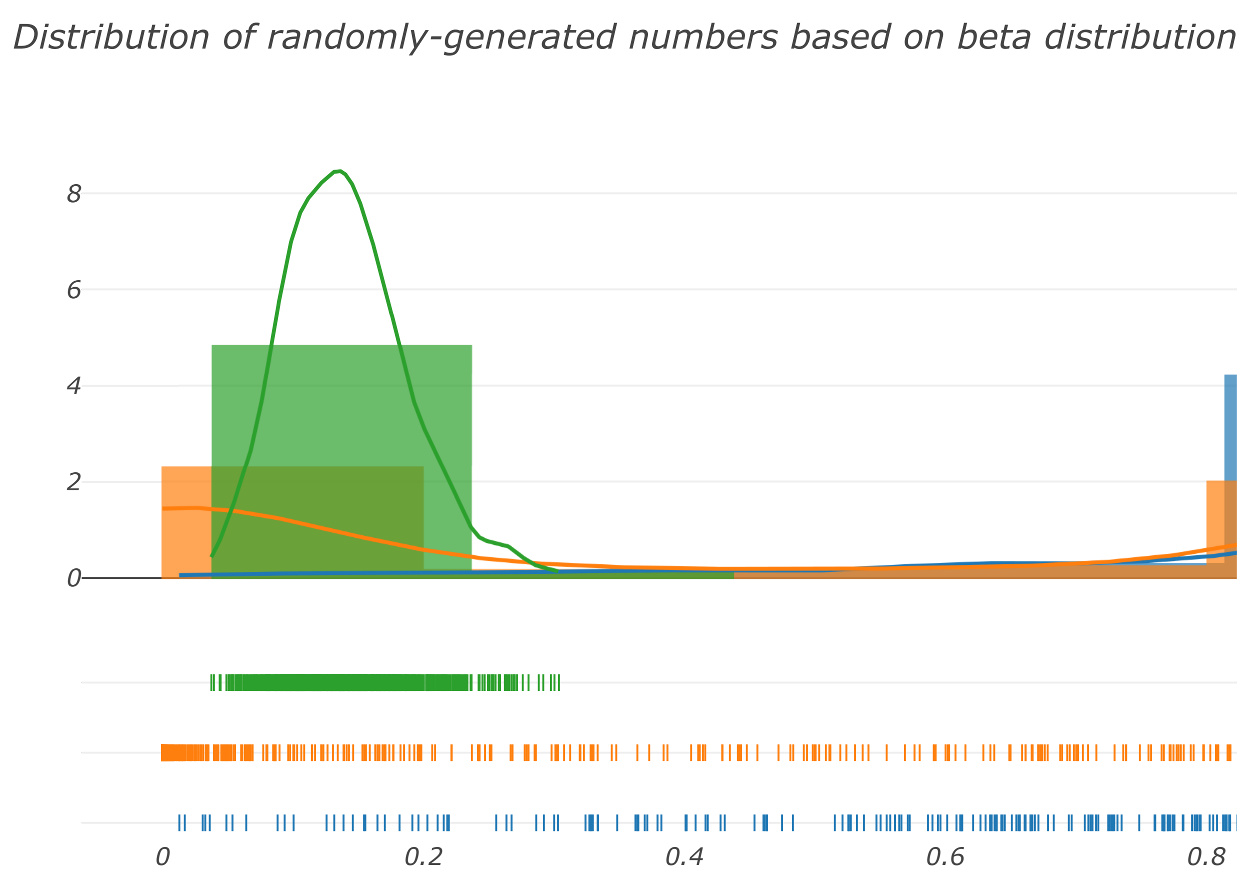
Note that the Scipy `beta()` function above returns the PDF whereas the `numpy.random.beta()` function generates random numbers.

In [14]:
```
# generate 1000 random values with different α and β values
betaValues_alpha11_beta01 = np.random.beta(1.1,0.1,1000)
betaValues_alpha01_beta01 = np.random.beta(0.1,0.1,1000)
betaValues_alpha80_beta48 = np.random.beta(8.0,48,1000)

# plot all the variables - reference: https://plot.ly/python/histograms/
data = [
    go.Histogram(x=betaValues_alpha11_beta01, opacity=0.75, name = 'α = 1.1, β
= 0.1'),
    go.Histogram(x=betaValues_alpha01_beta01, opacity=0.75, name = 'α = 0.1, β
= 0.1'),
    go.Histogram(x=betaValues_alpha80_beta48, opacity=0.75, name = 'α = 8.0, β
= 48')
]

layout = go.Layout(
    barmode='overlay',
    title = 'Distribution of randomly-generated numbers based on beta distribut
ions with different input parameters')

fig = go.Figure(data=data, layout=layout)
iplot(fig)
```



The Plotly library also comes with the ability to make distplots where the data distribution is shown along with a histogram and a fitted line.

In [15]:
```
hist_data = [betaValues_alpha11_beta01, betaValues_alpha01_beta01, betaValues_a
lpha80_beta48]
group_labels = ['α = 1.1, β = 0.1', 'α = 0.1, β = 0.1', 'α = 8.0, β = 48']

# figure_factory functions work slightly differently from standard plotly funct
ions
fig = ff.create_distplot(hist_data, group_labels, bin_size=0.2)
fig['layout'].update(
    title='Distribution of randomly-generated numbers based on beta distributio
ns with different input parameters')

iplot(fig)
```



Exponential distribution

The official documentations for the corresponding libraries can be found on the [Scipy (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.expon.html#scipy.stats.expon)](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.expon.html#scipy.stats.expon) and [Numpy (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.exponential.html#numpy.random.exponential)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.exponential.html#numpy.random.exponential) pages.

The exponential distribution function, also known as the negative exponential distribution, has a really simple form:

$$f(x) = \lambda e^{-\lambda x}$$

This function is only defined for all positive numbers [0, ∞], unlike the Laplace distribution, and is related to the Poisson distribution. However, the Poisson distribution has a discrete function - it concerns only events that happen over a fixed interval of time or space. In contrast, the exponential distribution models events over a continuous interval [(reference) (https://www.statisticshowto.datasciencecentral.com/exponential-distribution/)](https://www.statisticshowto.datasciencecentral.com/exponential-distribution/).

Although the main use of this function is to explain the time between events in a Poisson point process, it can also be used to [model other scenarios (https://www.hindawi.com/journals/jps/2017/2106748/)](https://www.hindawi.com/journals/jps/2017/2106748/):

- *bout (meal) criteria for analysis of animal behaviour Yeates et al., 2001 (https://www.sciencedirect.com/science/article/pii/S0022519301924257?via%3Dihub)*
- *rainfall estimation and analysis Madi & Raqab, 2007 (https://onlinelibrary.wiley.com/doi/abs/10.1002/env.826)*
- *episode peak and duration for ecohydroclimatic applications Biondi et al., 2008 (https://www.sciencedirect.com/science/article/pii/S0304380007004838?via%3Dihub)*
- *mean life of power system equipment with limited end-of-life failure data Cota-Felix et al., 2009 (https://ieeexplore.ieee.org/document/5281863)*

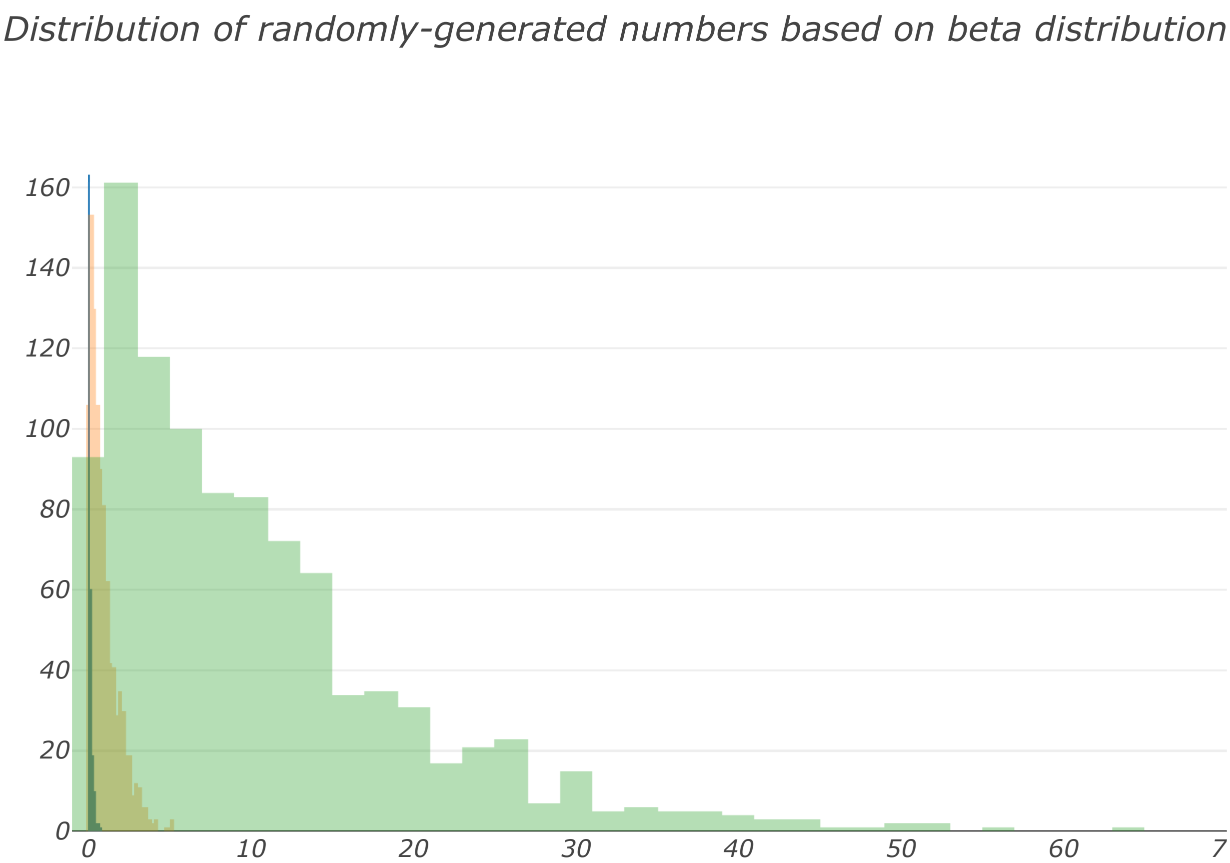
Using the Scipy `expon.pdf()` *function, we can generate illustrative probability density function curves with different parameters to determine their effect on the distribution. The figure below is fully interactive, so the loc and scale parameters are tunable.*

In [16]:
```
loc = scale = 1 # we need some loc and scale values for the initial y
x = np.arange(0, 400, 0.1) # let's plot x from 0 to 400
y = expon.pdf(x,loc,scale) # calculate the expon PDF using the x, loc and scale values from above

layout = go.Layout(
    title = 'Exponential probability density function',
    xaxis = dict(title='x'),
    yaxis = dict(title='PDF'),
)

# plot y over x as a scatter plot
f = go.FigureWidget(data=[go.Scatter(x=x, y=y)], layout = layout)

# we need to assign the slider to a function - this slider is actually from ipywidgets and not from plotly
def exponential_function(loc,scale):
    f.data[0].y = expon.pdf(x,loc,scale)

parameter_slider = interactive(exponential_function, loc=(0, 50.1, 0.1), scale=(0.01, 50.1, 0.1))
vb = VBox((f, parameter_slider))
vb.layout.align_items = 'center'
vb
```

The loc parameter determines the x value corresponding to the peak PDF while scale determines the curve steepness, especially for the x values beyond that of the peak PDF.

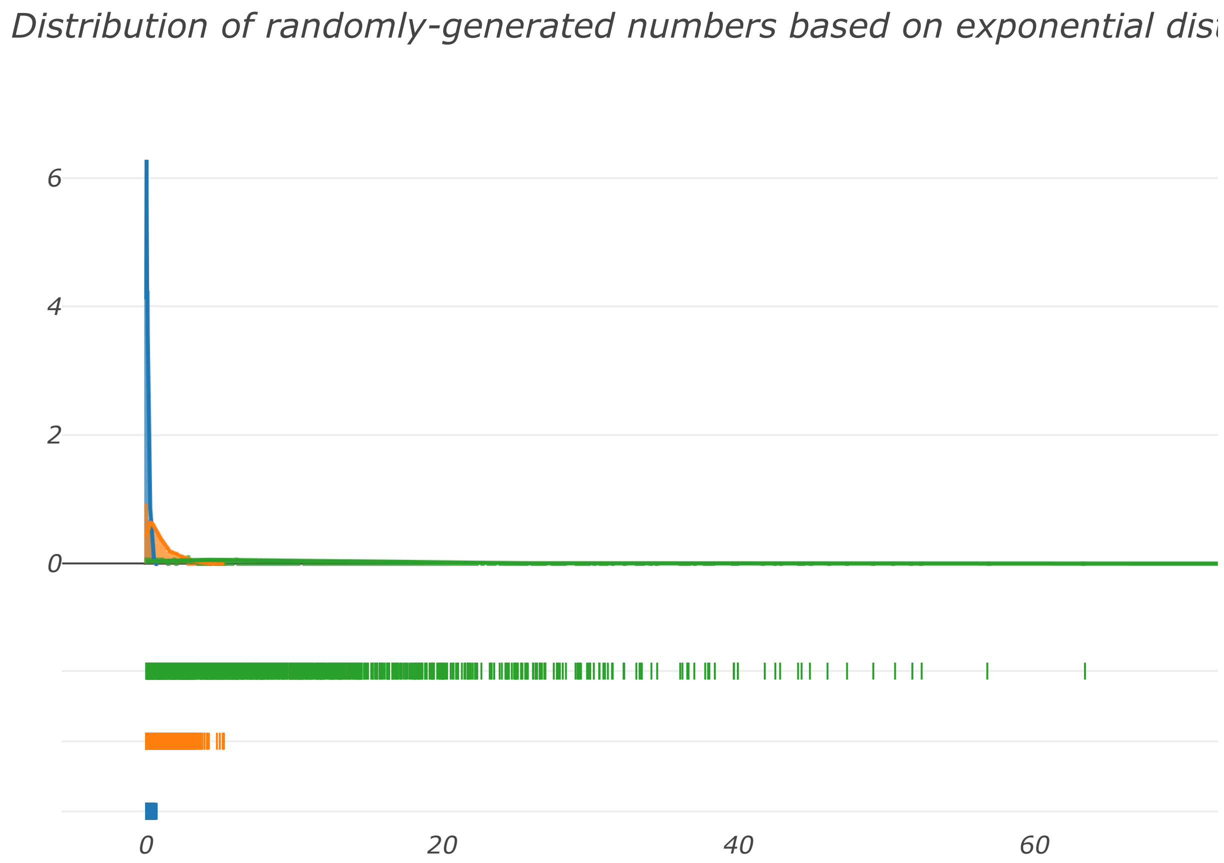
Based on the influences of the loc and scale parameters, we can now generate and visualize the Numpy-generated random numbers that follow am exponential distribution. The numpy `exponential()` *only takes the scale value as an argument, not the loc.*

In [17]:
```
# generate 1000 random values with different scale values
exponValues_scale01 = np.random.exponential(0.1,1000)
exponValues_scale1 = np.random.exponential(1,1000)
exponValues_scale10 = np.random.exponential(10,1000)

# plot all the variables - reference: https://plot.ly/python/histograms/
data = [
    go.Histogram(x=exponValues_scale01, opacity=0.95, name = 'scale = 0.1'), #
opacity = 0.95 to better see the curve
    go.Histogram(x=exponValues_scale1, opacity=0.35, name = 'scale = 1'),
    go.Histogram(x=exponValues_scale10, opacity=0.35, name = 'scale = 10')
]

layout = go.Layout(
    barmode='overlay',
    title = 'Distribution of randomly-generated numbers based on beta distribut
ions with different input parameters')

fig = go.Figure(data=data, layout=layout)
iplot(fig)
```



Note the extremely narrow range of the random numbers generated with a scale of 0.1. In other words, the distribution falls on rapidly. This is better visualized with a distplot. You can play around with the range slider at the bottom of the plot.

In [18]:
```python
hist_data = [exponValues_scale01, exponValues_scale1, exponValues_scale10]

group_labels = ['scale = 0.1', 'scale = 1', 'scale = 10']

# figure_factory functions work slightly differently from standard plotly funct
ions
fig = ff.create_distplot(hist_data, group_labels, bin_size=0.2)
fig['layout'].update(
    title='Distribution of randomly-generated numbers based on exponential dist
ributions with scale parameters')

iplot(fig)
```



Laplace distribution

The official documentations for the corresponding libraries can be found on the [Scipy (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.laplace.html#scipy.stats.laplace)] and [Numpy (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.laplace.html#numpy.random.laplace)] pages.

The Laplace distribution, also called the double exponential distribution (law of the difference between two exponential random variables) [Aryal (2006) (https://scholarcommons.usf.edu/etd/2444/)], is named after the French polymath, [Pierre-Simon Laplace (https://www.britannica.com/biography/Pierre-Simon-marquis-de-Laplace)]. The function for this distribution is defined for all real numbers (unlike the exponential distribution), and is therefore a continuous distribution function:

Standard form:

$$f(x) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}$$

Scipy form:

where $b = 1, \mu = 0$

$$f(x) = \frac{1}{2} e^{-|x|}$$

Examples of scenarios that can be explained according to the Laplace distribution include:

- *[daily sunspot numbers (https://arxiv.org/abs/1210.3119)]*
- *[flow cytometry for bacterial size measurements (https://www.idescat.cat/sort/sort322/32.2.2.julia-etal.pdf)]*
- *[encoding and decoding (codec) of analog signals (https://www.springer.com/us/book/9780817641665)] (Kotz et al., 2001)*
- *share market return models (same reference as above)*

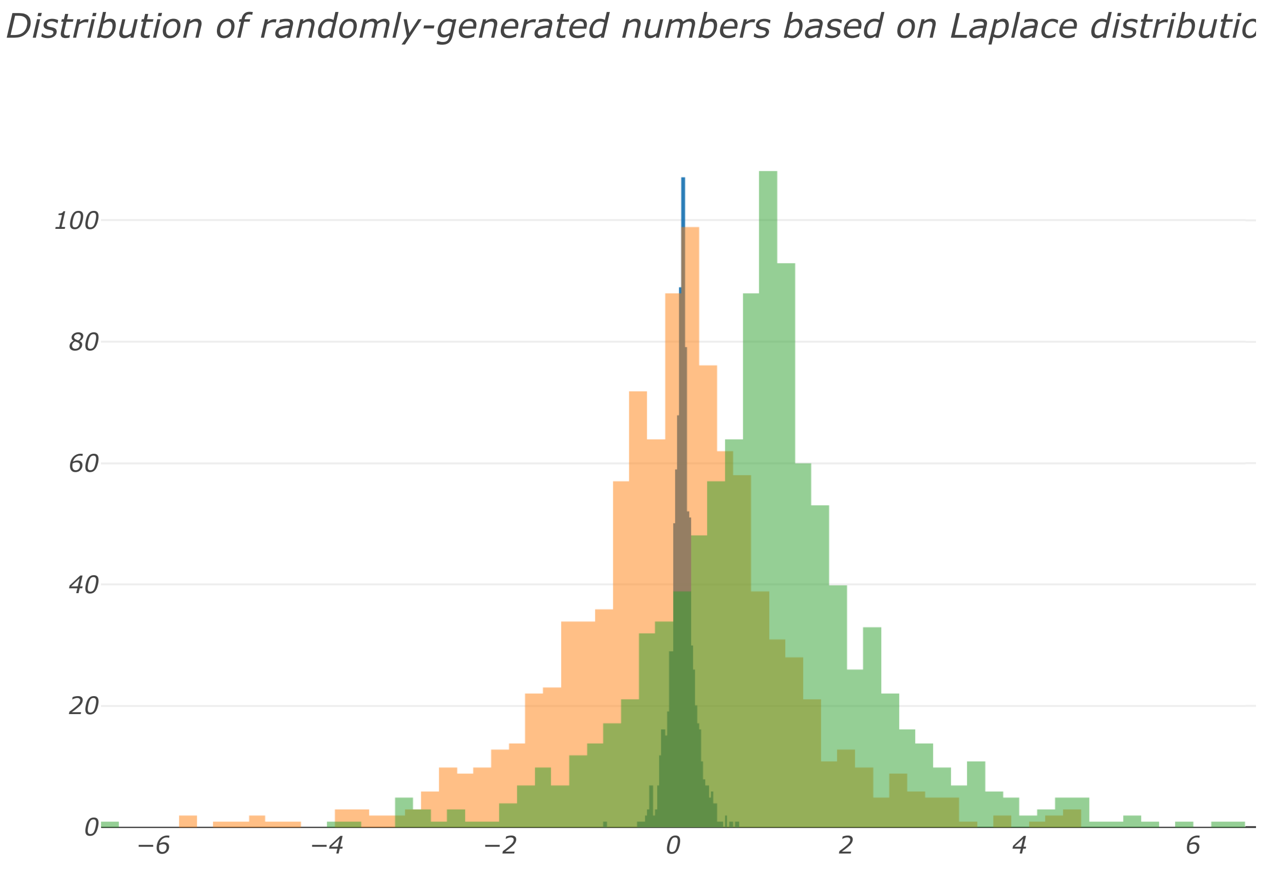
Using the Scipy `laplace.pdf()` *function, we can generate illustrative probability density function curves with different parameters to determine their effect on the distribution. The figure below is fully interactive, so the loc and scale parameters are tunable.*

In [19]:
```
loc = scale = 1 # we need some loc and scale values for the initial y
x = np.arange(-250, 250, 0.1) # let's plot x from -250 to 250
y = laplace.pdf(x,loc,scale) # calculate the expon PDF using the x, loc and sca
le values from above

layout = go.Layout(
    title = 'Laplace probability density function',
    xaxis = dict(title='x'),
    yaxis = dict(title='PDF'),
)

# plot y over x as a scatter plot
f = go.FigureWidget(data=[go.Scatter(x=x, y=y)], layout = layout)

# we need to assign the slider to a function - this slider is actually from ipy
widgets and not from plotly
def laplace_function(loc,scale):
    f.data[0].y = laplace.pdf(x,loc,scale)

parameter_slider = interactive(laplace_function, loc=(-50, 50.1, 0.1), scale=(0
.01, 50.1, 0.1))
vb = VBox((f, parameter_slider))
vb.layout.align_items = 'center'
vb
```

Thanks to the interactive plot above, it is evident that the loc parameter shifts the value of x corresponding to the maximum PDF value while scale changes the curve width. We can also generate random numbers according to the Laplace distribution using the Numpy `laplace()` function, which takes both the scale and loc values as arguments.

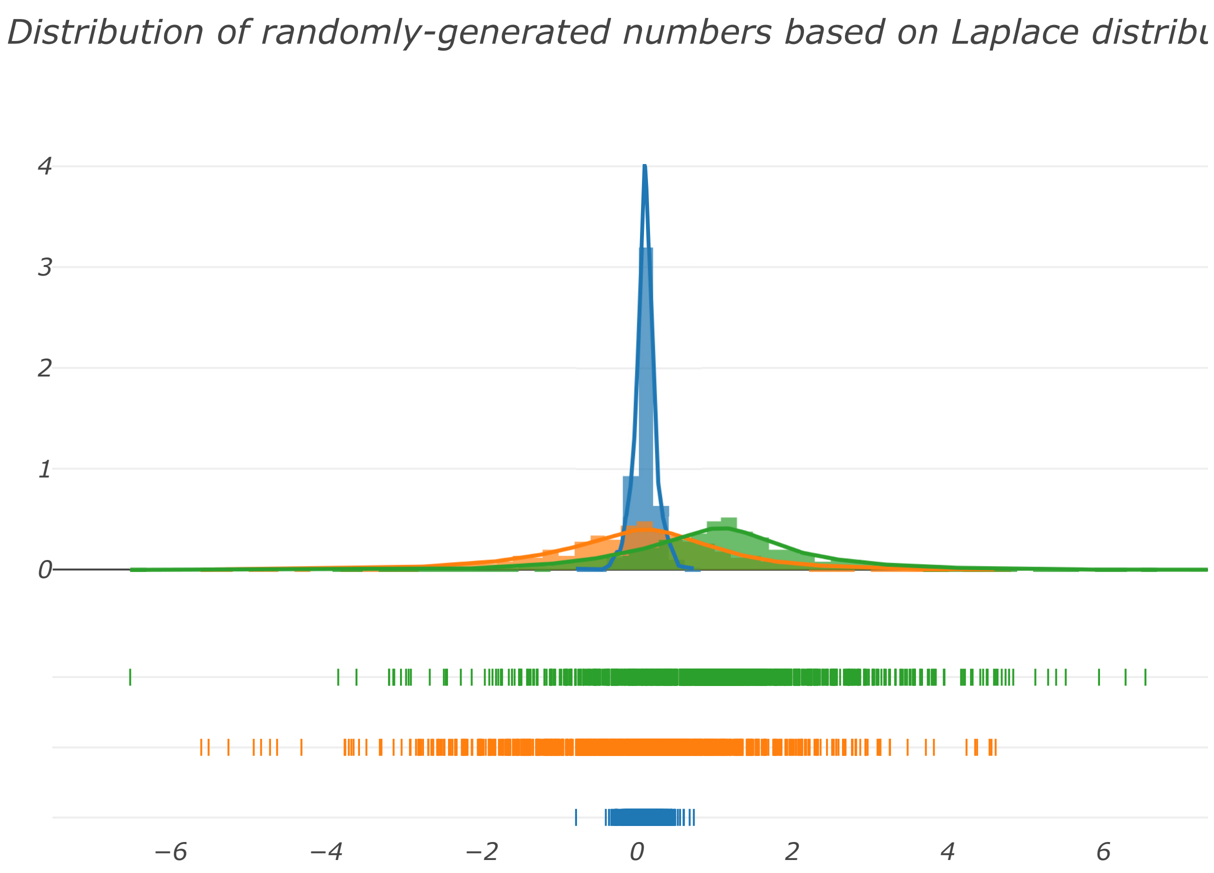
In [20]:

```
# generate 1000 random values with different scale values

laplaceValues_loc01_scale01 = np.random.laplace(0.1,0.1,1000)
laplaceValues_loc01_scale1 = np.random.laplace(0.1, 1, 1000)
laplaceValues_loc1_scale1 = np.random.laplace(1,1,1000)

# plot all the variables - reference: https://plot.ly/python/histograms/
data = [
    go.Histogram(x=laplaceValues_loc01_scale01, opacity=0.95, name = 'loc = 0.1
, scale = 0.1'), # opacity = 0.95 to better see the curve
    go.Histogram(x=laplaceValues_loc01_scale1, opacity=0.5, name = 'loc = 0.1,
scale = 1'),
    go.Histogram(x=laplaceValues_loc1_scale1, opacity=0.5, name = 'loc = 1, sca
le = 1')
]
layout = go.Layout(
    barmode='overlay',
    title = 'Distribution of randomly-generated numbers based on Laplace distri
butions with different input parameters')

fig = go.Figure(data=data, layout=layout)
iplot(fig)
```



With a scale of 0.1, the random numbers are generated with an extremely narrow range. This is better visualized with a distplot, which comes with an interactive range slider at the bottom of the plot.

In [21]:
```
hist_data = [laplaceValues_loc01_scale01, laplaceValues_loc01_scale1, laplaceValues_loc1_scale1]

group_labels = ['loc = 0.1, scale = 0.1', 'loc = 0.1, scale = 1', 'loc = 1, scale = 1']

# figure_factory functions work slightly differently from standard plotly functions
fig = ff.create_distplot(hist_data, group_labels, bin_size=0.2)
fig['layout'].update(
    title='Distribution of randomly-generated numbers based on Laplace distributions with different parameters')

iplot(fig)
```



Poisson distribution

The official documentations for the corresponding libraries can be found on the [Scipy (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.poisson.html#scipy.stats.poisson)](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.poisson.html#scipy.stats.poisson) and [Numpy (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.poisson.html#numpy.random.poisson)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.poisson.html#numpy.random.poisson) pages.

The Poisson distribution is [named after Baron Siméon Denis Poisson, a French mathematician, engineer and physicist (https://www.britannica.com/biography/Simeon-Denis-Poisson)](https://www.britannica.com/biography/Simeon-Denis-Poisson). Unlike the distributions above, the Poisson probability mass function is a discrete distribution, meaning it is only defined for positive integer values (k).

$$f(k) = e^{-\mu}\frac{\mu^k}{k!}$$

This distribution is usually used to model events within a predefined period of time or space that occur with a constant rate. Furthermore, the events occur independently, i.e, without influence of the prior event. The Poisson distribution is a special limiting case of a binomial distribution where the number of samples/trials is very large and the probability of a particular event occurring is very small. A few examples of Poisson-type events are:

- *the number of deaths by accidental horse kicking in the Prussian army - [Pandit (2015) (https://onlinelibrary.wiley.com/doi/full/10.1111/anae.13261)](https://onlinelibrary.wiley.com/doi/full/10.1111/anae.13261)*
- *number of typographical errors on a page*
- *radioactive decay (although see this paper: [Sitek et al., 2015 (https://www.sciencedirect.com/science/article/pii/S1120179715009011)](https://www.sciencedirect.com/science/article/pii/S1120179715009011)*

Reference: [Interactive Math (https://www.intmath.com/counting-probability/13-poisson-probability-distribution.php)](https://www.intmath.com/counting-probability/13-poisson-probability-distribution.php)

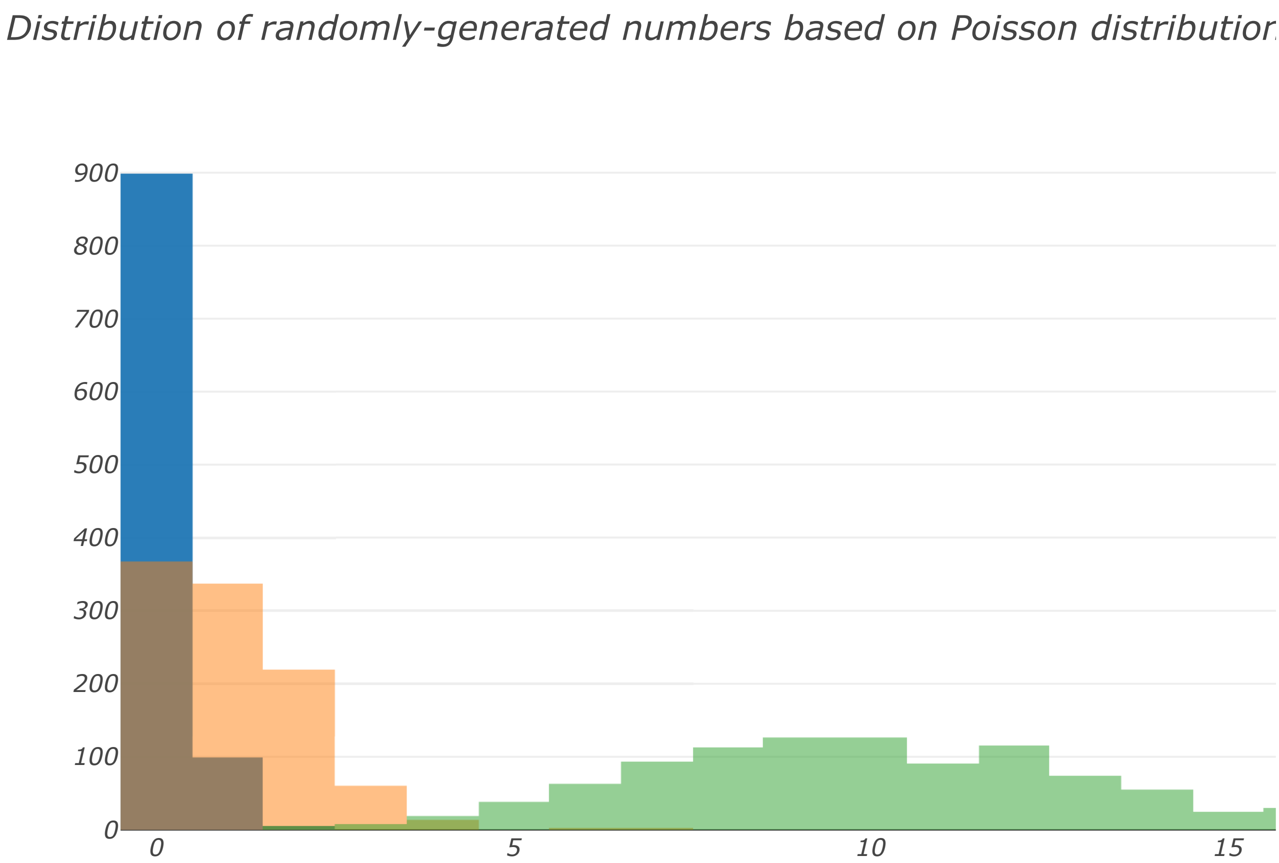
Using the Scipy `poisson.pmf()` function, we can generate illustrative probability mass function curves with different parameters to determine their effect on the distribution. The figure below is fully interactive, so you can tune the loc and scale parameters.

In [22]:
```
mu = loc = 1 # we need some loc and scale values for the initial y
x = np.arange(0, 150, 0.01) # let's plot x from 0 to 150 in a very fine interva
l to make visible sharp discrete peaks
y = poisson.pmf(x,mu,loc) # calculate the Poisson PMF using the x, mu and loc v
alues from above

layout = go.Layout(
    title = 'Poisson probability mass function',
    xaxis = dict(title='x'),
    yaxis = dict(title='PMF'),
)

# plot y over x as a scatter plot
f = go.FigureWidget(data=[go.Scatter(x=x, y=y)], layout = layout)

# we need to assign the slider to a function - this slider is actually from ipy
widgets and not from plotly
def poisson_function(mu,loc):
    f.data[0].y = poisson.pmf(x,mu,loc)

parameter_slider = interactive(poisson_function, mu=(0, 50, 1), loc=(0, 50, 1))
vb = VBox((f, parameter_slider))
vb.layout.align_items = 'center'
vb
```

Let's now generate 1000 random numbers according to the Poisson distribution. Note from above that the loc value determines the x value corresponding to the highest (peak) value of the PMF curve, while mu influences the curve width.

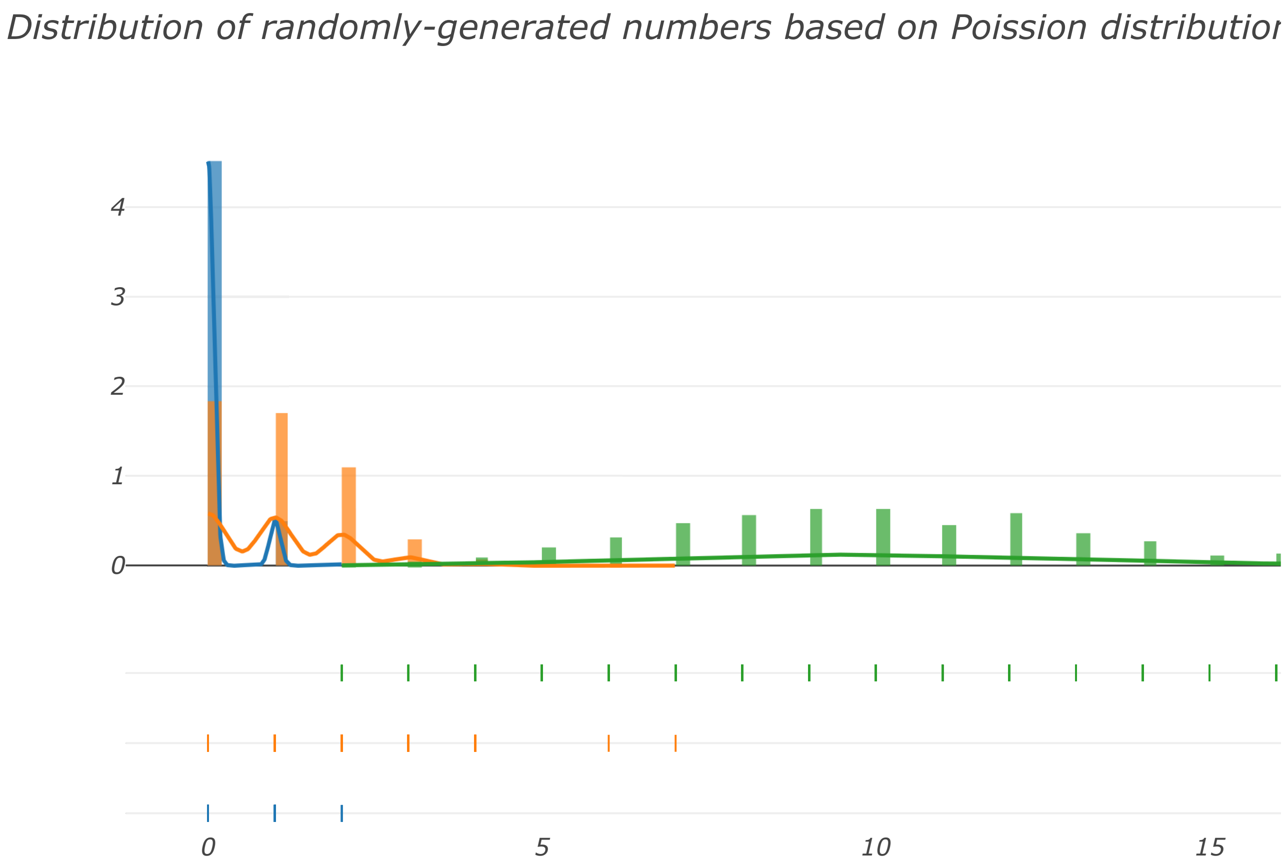
To generate the random numbers, we use the Numpy `poisson()` *function while passing different takes only the lambda/mu as an argument (Scipy uses the term mu while Numpy uses lambda - both are the same).*

In [23]:
```
# generate 1000 random values with different lambda values
poissonValues_lambda01 = np.random.poisson(0.1,1000)
poissonValues_lambda1 = np.random.poisson(1, 1000)
poissonValues_lambda10 = np.random.poisson(10,1000)

data = [
    # plot all the variables - reference: https://plot.ly/python/histograms/
    go.Histogram(x=poissonValues_lambda01, opacity=0.95, name = 'lambda = 0.1')
, # opacity = 0.95 to better see the curve
    go.Histogram(x=poissonValues_lambda1, opacity=0.5, name = 'lambda = 1'),
    go.Histogram(x=poissonValues_lambda10, opacity=0.5, name = 'lambda = 10')
    ]

layout = go.Layout(
    barmode='overlay',
    title = 'Distribution of randomly-generated numbers based on Poisson distri
butions with different lambda parameters')

fig = go.Figure(data=data, layout=layout)
iplot(fig)
```



The distplot below is similar to the histogram above, but also comes with a great distribution slider and plots the curve automatically.

In [24]:
```
hist_data = [poissonValues_lambda01, poissonValues_lambda1, poissonValues_lambd
a10]

group_labels = ['lambda = 0.1', 'lambda = 1', 'lambda = 10']

# figure_factory functions work slightly differently from standard plotly funct
ions
fig = ff.create_distplot(hist_data, group_labels, bin_size=0.2)
fig['layout'].update(
    title='Distribution of randomly-generated numbers based on Poission distrib
utions with different lambda parameters')

iplot(fig)
```



Rayleigh distribution

The official documentations for the corresponding libraries can be found on the [Scipy (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rayleigh.html#scipy.stats.rayleigh)](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rayleigh.html#scipy.stats.rayleigh) and [Numpy (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rayleigh.html#numpy.random.rayleigh)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rayleigh.html#numpy.random.rayleigh) pages.

The Rayleigh probability distribution function is a continuous function for positive random values, named after Lord Rayleigh [[1 (https://en.wikipedia.org/wiki/John_William_Strutt,_3rd_Baron_Rayleigh)](https://en.wikipedia.org/wiki/John_William_Strutt,_3rd_Baron_Rayleigh) and [2 (http://www.randomservices.org/random/special/Rayleigh.html)](http://www.randomservices.org/random/special/Rayleigh.html)]. The Rayleigh distribution is special cases of Weibull and Rice distributions.

Real-life events that follow the Rayleigh distribution are generally related to those with age-dependent lifetime, such as:

- *wave heights*
- *sound and light radiation*
- *radio signals*
- *wind power - wind speeds at wind turbine sites*
- *ultrasound image models*
- *lifetime of electric and electronic components*

References:

[Aslam et al., 2015 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4439070/)](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4439070/)

[Bhattacharya & Tyagi, 1990 (https://link.springer.com/article/10.1007%2FBF02863540)](https://link.springer.com/article/10.1007%2FBF02863540)

Using the Scipy `rayleigh.pdf()` *function, we can generate illustrative curves with different parameters to determine their effect on the distribution. The figure below is fully interactive, so you can tune the loc and scale parameters.*

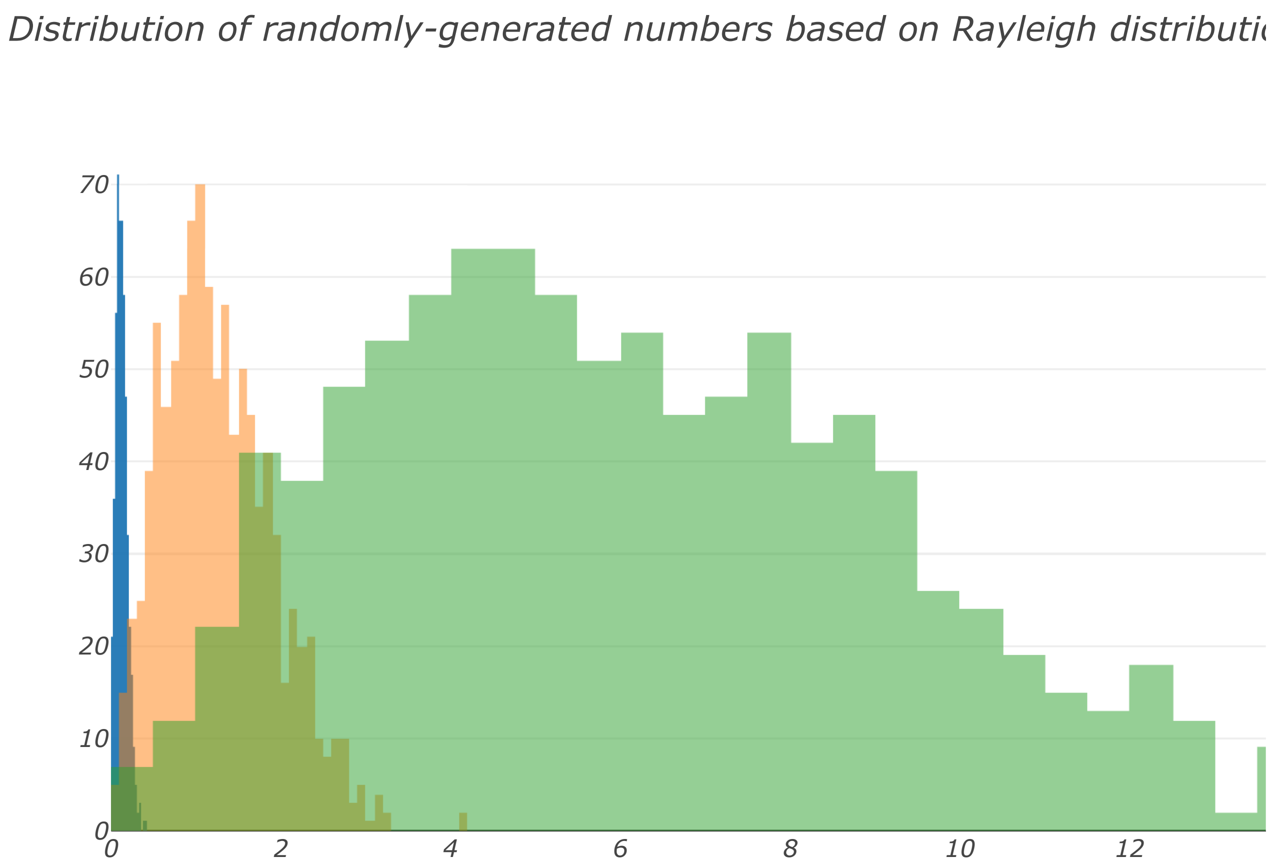
In [25]:

```
loc = scale = 1 # we need some initial loc and scale values to calculate the in
itial y
x = np.arange(0, 200, 0.1) # let's plot x from 0 to 200
y = rayleigh.pdf(x,loc,scale) # calculate the rayleigh PDF using the x, loc and
scale values from above

layout = go.Layout(
    title = 'Rayleigh probability density function',
    xaxis = dict(title='x'),
    yaxis = dict(title='PDF'),
)

f = go.FigureWidget(data=[go.Scatter(x=x, y=y)], layout = layout)

# we need to assign the slider to a function - this slider is actually from ipy
widgets and not from plotly
def rayleigh_function(loc,scale):
    f.data[0].y = rayleigh.pdf(x, loc, scale)

parameter_slider = interactive(rayleigh_function, loc=(0, 50, 1), scale=(0.1, 5
0, 0.1))
vb = VBox((f, parameter_slider))
vb.layout.align_items = 'center'
vb
```

Note from above that the loc value determines the x value corresponding to the highest (peak) value of the PDF curve while the scale parameter changes the width of the curve.

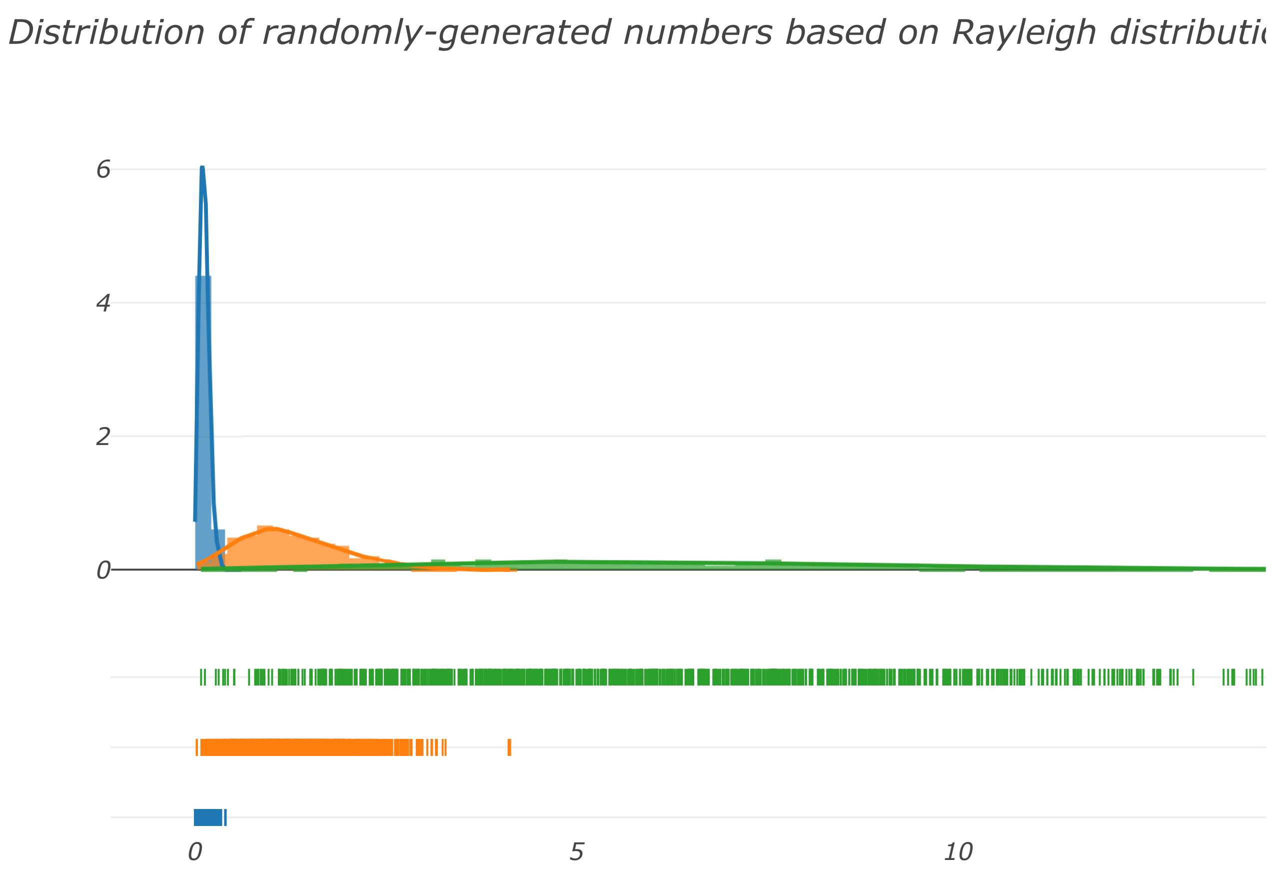
Based on the interactive plot above, we can now generate 1000 random numbers according to the Rayleigh distribution with 3 sets of scale values. The numpy `rayleigh()` *takes only the scale as an argument, although the peak can be shifted by multiplying these values by a constant (similar to the loc parameter).*

In [26]:
```
# generate 1000 random values with different scale values
rayleighValues_scale01 = np.random.rayleigh(0.1,1000)
rayleighValues_scale1 = np.random.rayleigh(1, 1000)
rayleighValues_scale5 = np.random.rayleigh(5,1000)

# plot all the variables - reference: https://plot.ly/python/histograms/
data =[
        go.Histogram(x=rayleighValues_scale01, opacity=0.95, name = 'scale = 0.
1'), # opacity = 0.95 for clearer curve
        go.Histogram(x=rayleighValues_scale1, opacity=0.5, name = 'scale = 1'),
        go.Histogram(x=rayleighValues_scale5, opacity=0.5, name = 'scale = 5')
]

layout = go.Layout(
    barmode='overlay',
    title = 'Distribution of randomly-generated numbers based on Rayleigh distr
ibutions with different scale parameters',
    )

fig = go.Figure(data=data, layout=layout)
iplot(fig)
```



Using the `distplot()` function, an alternative form the plot above can be obtained.

In [27]:
```
hist_data = [rayleighValues_scale01, rayleighValues_scale1, rayleighValues_scal
e5]

group_labels = ['scale = 0.1', 'scale = 1', 'scale = 5']

# figure_factory functions work slightly differently from standard plotly funct
ions
fig = ff.create_distplot(hist_data, group_labels, bin_size=0.2)
fig['layout'].update(
    title='Distribution of randomly-generated numbers based on Rayleigh distrib
utions with different scale parameters'
)

iplot(fig)
```



Conclusion

Using an interactive plotting technique, this notebook illustrates the functions provided by the Scipy and Numpy libraries. Although slightly more complicated, the use of the Plotly library, instead of Matplotlib, is justified by the high interactivity provided. We have also looked at the beta, exponential, Laplace, Poisson and Rayleigh distributions and their application to real-life problems. By having handy functions, it is possible to simulate these problems and develop codes for debugging and noise generation/reduction for predictive modeling, among others.

Overall, the Jupyter notebook is a great method to highlight different code snippets and to compare and contrast various alternative approaches. In this case, a Jupyter notebook has been created to simulate different Scipy and Numpy functions related to random number generation. The use of the Plotly library can be considered almost critical for good visualization and 'storytelling' especially for the non-technical colleagues or managers.