CAPITULO III NÚMEROS ALEATORIOS

3.1 Introducción.

La simulación de cualquier sistema o proceso donde existen componentes aleatorios necesita de números al azar, un número aleatorio es un número de una secuencia cuya probabilidad de ocurrencia es igual a la de cualquier otro número de la secuencia. Disponer de un buen generador de números aleatorios es fundamental en cualquier simulación. Constituye también una parte fundamental en otras áreas, como la Informática (algoritmos aleatorizados, verificación de algoritmos, complejidad de algoritmos, criptografía, ...), la Estadística (métodos de muestreo y remuestreo, contrastes Montecarlo, Inferencia Bayesiana, ...), y, en general, en cualquier problema de cálculo científico que, de manera directa o indirecta, incluya algún elemento probabilístico incluyendo los juegos de video.

Actualmente existen múltiples generadores de números aleatorios en diferentes entornos y compiladores lo cual supondría para un usuario de la Simulación que no es necesario su estudio. Sin embargo, observaciones sobre algunos generadores comerciales sugieren que debemos actuar con cuidado con el uso de ellos. Incluso, el uso progresivo de modelos de simulación cada vez más detallados exige generadores de números aleatorios de mayor calidad. La forma de generar números aleatorios tiene mucha historia, desde los primeros métodos que se llevaron esencialmente a mano (Manuales) como lanzar dados, retirar cartas, agitar urnas y muchas de las loterías en la actualidad todavía son muy utilizadas. A medida que las computadoras han revolucionado tecnológicamente ha aumentado la atención en la generación de números aleatorios de forma numérica, aritmética, algorítmica de formas secuenciales, como el método de la raíz media que dio inicio a la gran cantidad de algoritmos y su utilización en los diferentes problemas de simulación. Algunos autores siguen discutiendo que realmente los generadores por algoritmos aritméticos no son realmente números aleatorios sino llamados pseudoaleatorios.

El inicio en la generación de números aleatorios con Métodos Manuales, los mismos que son laboriosos y no muy prácticos se utiliza en procesos tales como:

```
- Ruleta
- Baraja
- Dados
- Lanzamientos de moneda
**Las DESVENTAJAS:**
- No es reproducible
- Lento
- No se pueden usar en computadores
```

Luego continúa con Métodos generados por Equipos Eléctricos como:

```
- RAND(Rand Corporation (1955) utilizado para generar una tabla de un mi
llón de números aleatorios proveniente de las mediciones del ruido eléct
rico)
- ERNIE(Electronic Random Number Indicator Equipment, utilizado para gen
erar números aleatorios para la lotería)
```

Y por último los generados por Computador Digital (Algoritmos Matemáticos) que en la década del 1950 se convertiría en la época en la que se generara números aleatorios de forma aritmética, estos métodos son secuenciales con cada número nuevo obtenido por uno o varios de sus predecesores de acuerdo a una fórmula matemática fija. Uno de los primeros métodos propuestos por Von Neuman y Metropolis, con el famoso método de los **cuadrados medios** que resumimos a continuación, pero debemos señalar que los números aleatorios llamados propiamente pseudoaleatorios que deben cumplir con ciertas características:

- Deben ser uniformemente distribuidos u[0,1]
- Estadísticamente independiente (La correlación entre los números generados sea mínimos y se mide entre números consecutivos.
- Debe ser reproducible
- Generar a grandes velocidades
- Mínimo almacenamiento
- Sin repetición, con largos periodos

Método de los cuadrados medios

La primera técnica de generación de números pseudoaleatorios conocida como técnica del **cuadrado medio**. que consta de los siguientes pasos:

- Seleccione un valor inicial o "semilla" (r).
- Determinamos el número de dígitos
- Elévese al cuadrado el valor inicial.
- Tómense lo n dígitos centrales del cuadrado del número anterior
- Repítase el proceso cuantas veces sea necesario.

In [10]:

```
# Método de los cuadrados medios
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
n=100
#r=7182                                  # seleccionamos el valor inicial r
r=171                                   # seleccionamos el valor inicial r
l=len(str(r))                           # determinamos el número de dígitos
lista = []                              # almacenamos en una lista
lista2 = []
i=1
#while len(lista) == len(set(lista)):
while i < n:
    x=str(r*r)                          # Elevamos al cuadrado r
    if l % 2 == 0:
        x = x.zfill(l*2)
    else:
        x = x.zfill(l)
    y=(len(x)-l)/2
    y=int(y)
    r=int(x[y:y+l])
    lista.append(r)
    lista2.append(x)
    i=i+1

df = pd.DataFrame({'X2':lista2,'Xi':lista})
dfrac = df["Xi"]/10**l

df["ri"] = dfrac
#df.head()
df
```

Out[10]:

	X2	Xi	ri
0	29241	924	0.924
1	853776	537	0.537
2	288369	883	0.883
3	779689	796	0.796
4	633616	336	0.336
5	112896	128	0.128
6	16384	638	0.638
7	407044	70	0.070
8	4900	490	0.490
9	240100	401	0.401
10	160801	608	0.608
11	369664	696	0.696
12	484416	844	0.844
13	712336	123	0.123

	X2	Xi	ri
14	15129	512	0.512
15	262144	621	0.621
16	385641	856	0.856
17	732736	327	0.327
18	106929	69	0.069
19	4761	476	0.476
20	226576	265	0.265
21	70225	22	0.022
22	484	484	0.484
23	234256	342	0.342
24	116964	169	0.169
25	28561	856	0.856
26	732736	327	0.327
27	106929	69	0.069
28	4761	476	0.476
29	226576	265	0.265
...	...	...	...
69	116964	169	0.169
70	28561	856	0.856
71	732736	327	0.327
72	106929	69	0.069
73	4761	476	0.476
74	226576	265	0.265
75	70225	22	0.022
76	484	484	0.484
77	234256	342	0.342
78	116964	169	0.169
79	28561	856	0.856
80	732736	327	0.327
81	106929	69	0.069
82	4761	476	0.476
83	226576	265	0.265
84	70225	22	0.022
85	484	484	0.484
86	234256	342	0.342
87	116964	169	0.169

	X2	Xi	ri
88	28561	856	0.856
89	732736	327	0.327
90	106929	69	0.069
91	4761	476	0.476
92	226576	265	0.265
93	70225	22	0.022
94	484	484	0.484
95	234256	342	0.342
96	116964	169	0.169
97	28561	856	0.856
98	732736	327	0.327

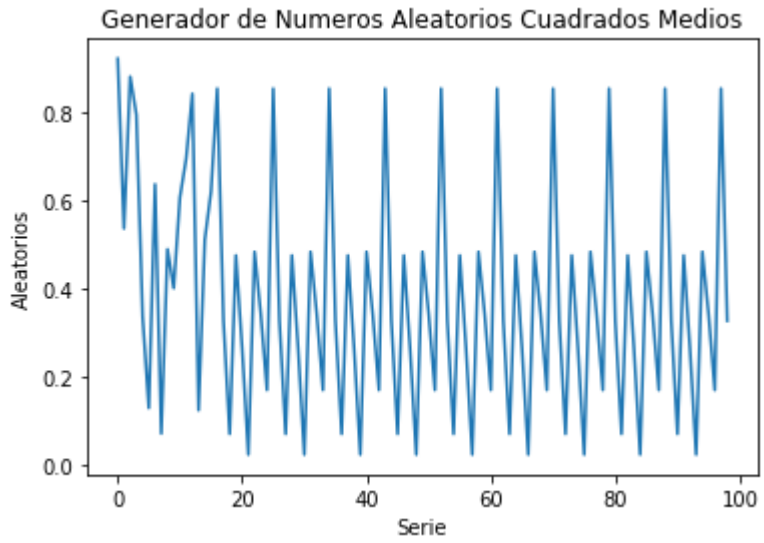
99 rows × 3 columns

In [11]:
```
1 df.tail()
2
```

Out[11]:

	X2	Xi	ri
94	484	484	0.484
95	234256	342	0.342
96	116964	169	0.169
97	28561	856	0.856
98	732736	327	0.327

In [12]:
```
### Graficando los numeros aleatorios generados
# seleccionamos del dataframe los nùmeros generados
x1=df['ri']
plt.plot(x1)
plt.title('Generador de Numeros Aleatorios Cuadrados Medios')
plt.xlabel('Serie')
plt.ylabel('Aleatorios')
plt.show()
```



Análisis del método de cuadrados medios ¶

Como podemos observar los Numeros Aleatorios generados por los cuadrados medios con muy pocos numeros generados no cumplen con los principios de los numeros aleatorios como: **Sin repetición, con largos periodos**, por lo tanto para utilizar este método hay que ser muy cuidadoso en la selección de la semilla.

en el código anterior podemos cambiar la cantidad de numeros **n** y/o la semilla **r**, donde observaremos facilmente que este método no favorece para la simulación.

3.2 Métodos congruenciales lineales

La mayoría de los generadores de números aleatorios que se utilizan hoy son los congruenciales lineales, Lehmer (1951) incorporo una secuencia de números enteros X1, X2,X3,X4, definida como una formula recursiva de la siguiente forma:

- Xn+1 = (aXn + c) (modulo m)
- Xn+2 = (aXn+1 + c) (modulo m)
    - Xn = Semilla
    - a = multiplicador
    - c = incremento
    - m = modulo
    - Que son enteros positivos

Propiedades:

- 0 < m , a < m, c<m
- ri = Xn+1 / m

Como el caso anterior de los **cuadrados medios** se debe seleccionar cuidadosamente a,c,m,Xn ya que de estas variables depende la calidad y el cumplimiento de los requisitos para ser considerado un buen generador de números aleatorios.

Ejemplo

m = 1000, a = 101, c = 457 , X0 = 4

In [8]:
```
# Generador de números aleatorios Congruencia lineal
n, m, a, x0, c  = 20, 1000, 101, 4, 457
x = [1] * n
r = [0.1] * n
print (" Método de Congruencia Lineal ")
print("-------------------------------")
print ("n=cantidad de números generados  : ", n)
print()
print ("m  : ", m)
print ("a  : ", a)
print ("c  : ", c)
print ("Xo : ", x0 )

for i in range(0, n):
        x[i] = ((a*x0)+c) % m
        x0 = x[i]
        r[i] = x0 / m
# llenamos nuestro DataFrame
d = {'Xn': x, 'ri': r }
df = pd.DataFrame(data=d)
df

```

```
 Método de Congruencia Lineal
-------------------------------
n=cantidad de números generados  :  20

m  :  1000
a  :  101
c  :  457
Xo :  4
```

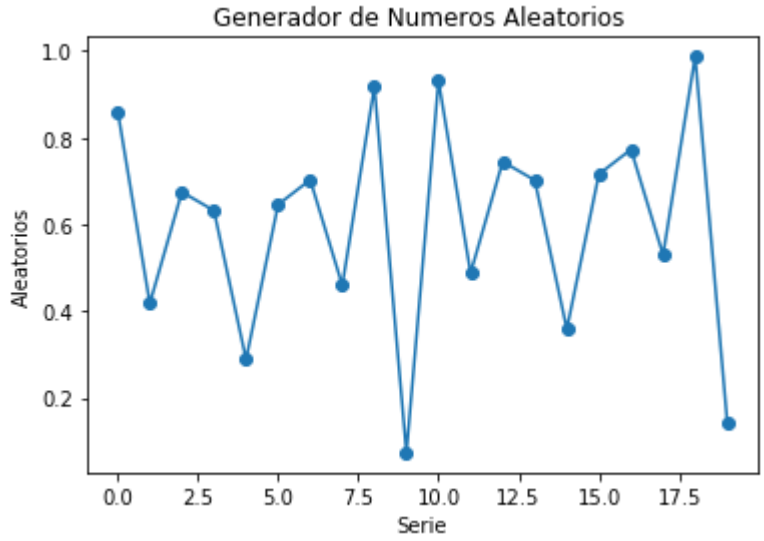
Out[8]:

	Xn	ri
0	861	0.861
1	418	0.418
2	675	0.675
3	632	0.632
4	289	0.289
5	646	0.646
6	703	0.703
7	460	0.460
8	917	0.917
9	74	0.074
10	931	0.931
11	488	0.488
12	745	0.745
13	702	0.702
14	359	0.359

	Xn	ri
15	716	0.716
16	773	0.773
17	530	0.530
18	987	0.987
19	144	0.144

In [9]:
```
# Graficamos los numeros generados
plt.plot(r,marker='o')
plt.title('Generador de Numeros Aleatorios ')
plt.xlabel('Serie')
plt.ylabel('Aleatorios')

```

Out[9]: Text(0, 0.5, 'Aleatorios')



Congruenciales multiplicativo

EL método multiplicativo congruencial es otro caso especial, donde la constante c=0 quedando la sucesión de la siguiente forma:

Xn+1 = (aXn) (mod m)

Se ha demostrado que este método se comporta aceptablemente desde el punto de vista estadístico. Con éste método el periodo máximo es m/4 y para asegurar esta propiedad cuando se utilizan computadoras binarias se consideran los siguientes criterios en la selección de los parámetros.

```
- La elección de *m* debe hacerse mediante la relación:
    - m = 2^b , (b>2) para cumputadora binaria
    - Xn o sea el valor inicial o semilla deberá ser numero primo.
- En cuanto al valor de la constante *a* su valor podrá ser de Los númer
os primos y además a<m,
```

para el ejemplo vamos a utilizar los datos de un generador del la hoja electrónica Excel-03, Excel-07 *Wichman y Hill (1982) :

- Xn+1 = (171Xn) (mod 30264)
- Xn+1 = (172Xn) (mod 30307)
- Xn+1 = (170Xn) (mod 30323)

Ejemplo 1 : Xn+1 = (171Xn) (mod 30264)

In [10]:
```
# generador excel-03
# - Xn+1 = (171Xn ) (mod 30264)
n, m, a, x0 = 20, 1000, 747, 123
x = [1] * n
r = [0.1] * n
print (" Generador Congruencial multiplicativo")
print ("-------------------------------------")
for i in range(0, n):
        x[i] = (a*x0) % m
        x0 = x[i]
        r[i] = x0 / m
d = {'Xn': x, 'ri': r }
df = pd.DataFrame(data=d)
df

```

```
 Generador Congruencial multiplicativo
-------------------------------------
```

Out[10]:

	Xn	ri
0	881	0.881
1	107	0.107
2	929	0.929
3	963	0.963
4	361	0.361
5	667	0.667
6	249	0.249
7	3	0.003
8	241	0.241
9	27	0.027
10	169	0.169
11	243	0.243
12	521	0.521
13	187	0.187
14	689	0.689
15	683	0.683
16	201	0.201
17	147	0.147
18	809	0.809
19	323	0.323

Ejemplo 2 : Xn+1 = (172Xn) (mod 30307)

In [11]:
```
# generador excel-07
# - Xn+1 = (172Xn ) (mod 30307)

n, m, a, x0 = 20, 30307, 172, 172
x = [1] * n
r = [0.1] * n
print (" Generador Congruencial multiplicativo")
print ("--------------------------------------")
for i in range(0, n):
        x[i] = (a*x0) % m
        x0 = x[i]
        r[i] = x0 / m
d = {'Xn': x, 'ri': r }
df1 = pd.DataFrame(data=d)
df1.head()

```

```
 Generador Congruencial multiplicativo
--------------------------------------
```

Out[11]:

	Xn	ri
0	29584	0.976144
1	27179	0.896790
2	7510	0.247798
3	18826	0.621177
4	25530	0.842380

Ejemplo 3 : Xn+1 = (170Xn) (mod 30323)

In [12]:
```
# generador excel-07
# - Xn+1 = (170Xn ) (mod 30323)

n, m, a, x0 = 20, 30323, 170, 170
x = [1] * n
r = [0.1] * n
print (" Generador Congruencial multiplicativo")
print ("---------------------------------------")
for i in range(0, n):
        x[i] = (a*x0) % m
        x0 = x[i]
        r[i] = x0 / m
d = {'Xn': x, 'ri': r }
df2 = pd.DataFrame(data=d)
df2.head()

```

```
 Generador Congruencial multiplicativo
---------------------------------------
```

Out[12]:

	Xn	ri
0	28900	0.953072
1	674	0.022227
2	23611	0.778650
3	11234	0.370478
4	29754	0.981235

In [13]:
```
df2.tail()

```

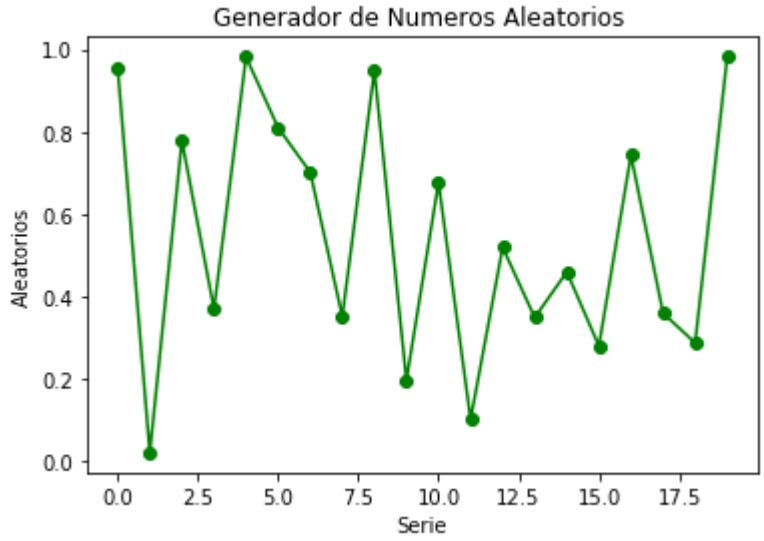
Out[13]:

	Xn	ri
15	8516	0.280843
16	22539	0.743297
17	10932	0.360518
18	8737	0.288131
19	29786	0.982291

Gráficamos los números aleatorios generados

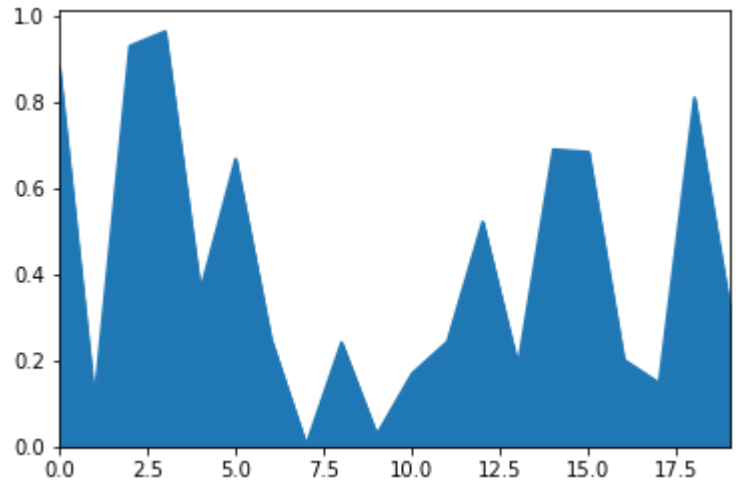
In [14]:
```
plt.plot(r,'g-', marker='o',)
plt.title('Generador de Numeros Aleatorios ')
plt.xlabel('Serie')
plt.ylabel('Aleatorios')

```

Out[14]: Text(0, 0.5, 'Aleatorios')



In [15]:
```
x1 = df["ri"]
x1.plot.area()
```

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x1d14a6cf4a8>

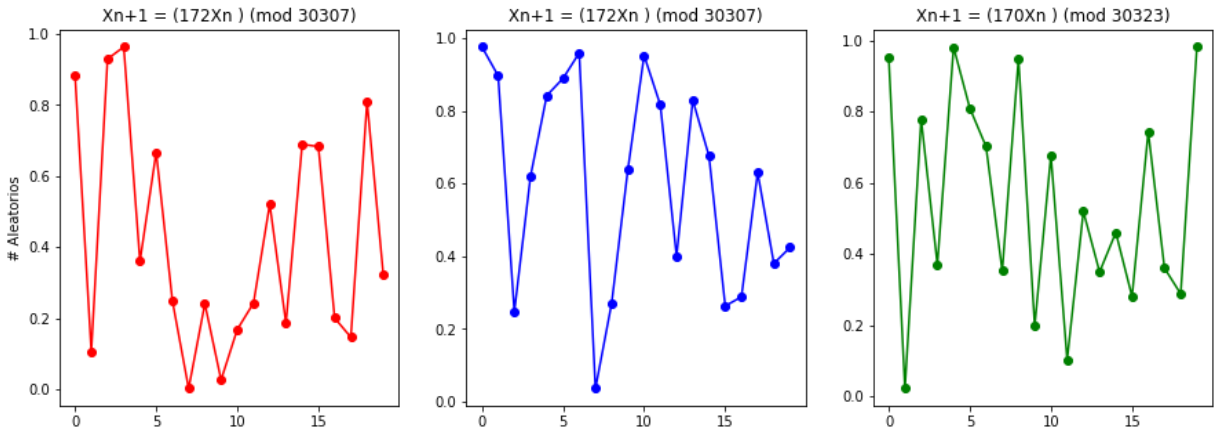


In [ ]:
```

```

Graficamos los tres modelos de generadores con sus diferentes constantes a,m,xo

In [16]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

x1 = df["ri"]
x2 = df1["ri"]
x3 = df2["ri"]
x = np.arange(n)

#fig = plt.figure()
plt.figure(figsize=(15,5))
# Gráfico 1
plt.subplot(131)
x1 = df["ri"]
p1, = plt.plot(x,x1,'r-',marker='o')
plt.ylabel('# Aleatorios')
plt.title(' Xn+1 = (172Xn ) (mod 30307) ')

# Gráfico 2
plt.subplot(132)
x2 = df1["ri"]
p1, = plt.plot(x,x2,'b-',marker='o')
plt.title(' Xn+1 = (172Xn ) (mod 30307) ')

# Gráfico 2
plt.subplot(133)
x3 = df2["ri"]
p1, = plt.plot(x,x3,'g-',marker='o')
plt.title(' Xn+1 = (170Xn ) (mod 30323) ')


```

Out[16]:
```
Text(0.5, 1.0, ' Xn+1 = (170Xn ) (mod 30323) ')
```



Conclusiones

En el gráfico podemos observar que las constantes a,m,xo son importantes en la generación de números aleatorios

3.3 Otros Generadores utilizados

David Ríos 2009, propone los siguientes ejercicios para analizar el comportamiento de estos generadores aleatorios:

- RANDU xi+1 = 65539xi mod 2^31
- Sinclair ZX81 xi+1=75xi mod(2^16 + 1)
- Numerical reciepes xi+1 = 1664525xi + 1013904223 mod 2^32
- Borland C/C++ xi+1=22695477xi + 1 mod 2^32

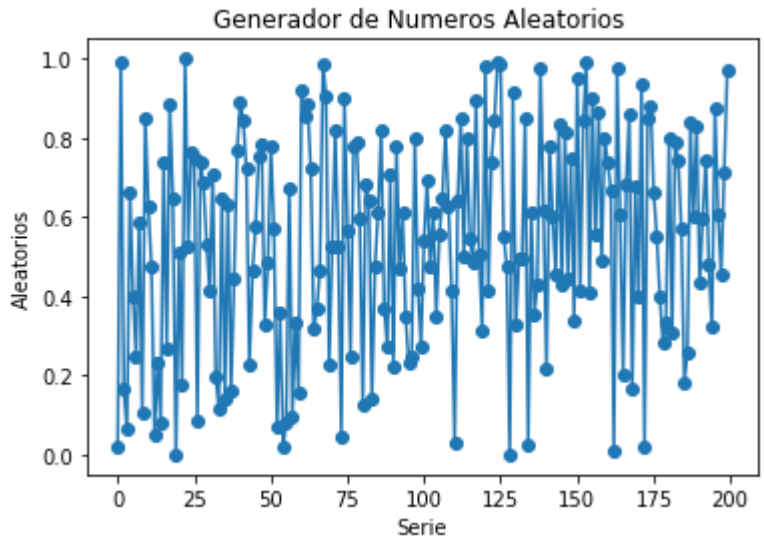
En el siguiente codigo generamos los números para observar su comportamiento,

In [17]:

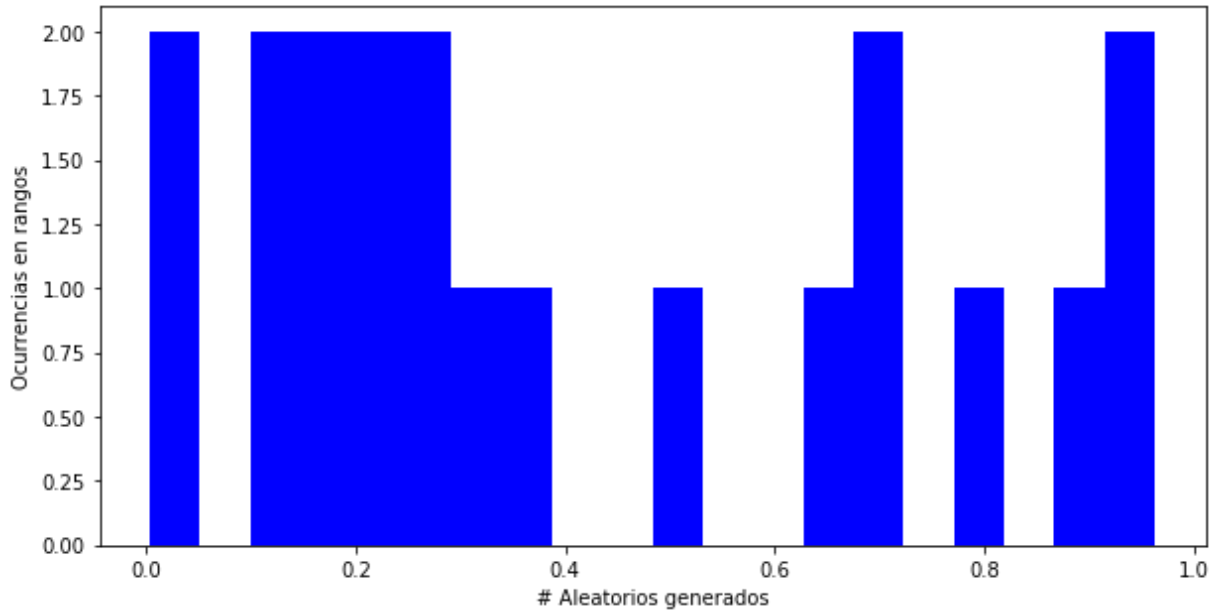
```
# Generador de números aleatorios Congruencia lineal
# Borland C/C++ xi+1=22695477xi + 1 mod 2^32

n, m, a, x0, c  = 200, 2**32, 22695477, 4, 1
x = [1] * n
r = [0.1] * n
print (" Método de Congruencia Lineal  Utilizado por:")
print("Borland C/C++ xi+1=22695477xi + 1 mod 2^32    ")
print ("n=cantidad de números generados  : ", n)
print()
print ("m  : ", m)
print ("a  : ", a)
print ("c  : ", c)
print ("Xo : ", x0 )

for i in range(0, n):
        x[i] = ((a*x0)+c) % m
        x0 = x[i]
        r[i] = x0 / m
# llenamos nuestro DataFrame
d = {'Xn': x, 'ri': r }
dfMCL = pd.DataFrame(data=d)
dfMCL

```

11	2046132792	0.476402
12	221316505	0.051529
13	1000654510	0.232983
14	349316615	0.081332
15	3163423092	0.736542
16	1151313157	0.268061
17	3793137674	0.883159
18	2782494227	0.647850
19	2024432	0.000471
20	2184728753	0.508672
21	762880678	0.177622
22	4293148767	0.999577
23	2252621228	0.524479

In [18]:
```
# Gráficos de dispersión
plt.title('Generador de Numeros Aleatorios ')
plt.xlabel('Serie')
plt.ylabel('Aleatorios')

#plt.scatter(df.index,df['ri'])
plt.plot(r,marker='o')
plt.show()
```



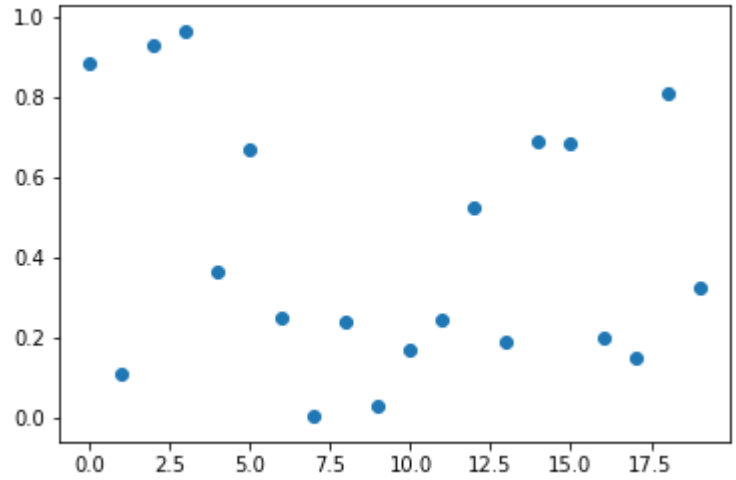
In [19]:
```
rS= df['ri']
#rS
plt.figure(figsize=(10,5))
plt.hist(rS,bins=20,color='blue')
plt.xlabel('# Aleatorios generados')
plt.ylabel('Ocurrencias en rangos')
plt.show()

#Aqui podemos observar en la grafica los números generados para lo cual se h
```



Gráficamos la dispersión del generador

In [20]:
```
# utilizamos scatter
plt.scatter(df.index,df['ri'])
```

Out[20]: <matplotlib.collections.PathCollection at 0x1d14ab08710>



Generador random de Python np.random.rand

Python cuenta con el módulo random para generación de números pseudo-aleatorios.

Contiene diversos métodos que corresponden a varias formas de generar este tipo de números. genera numeros aleatorios con una distribución uniforme [0,1] entre 0.0 excepto 1.0, la facilidad es que puede generar diferentes numeros aleatorios con diferentes intervalos y diferentes distribuciones que más adelante utilizaremos.

In [21]:
```
# para obtener la ayuda del módulo random debemos importar random
# y nos muestra la documentación completa del módulo random

import random
help(random)

```

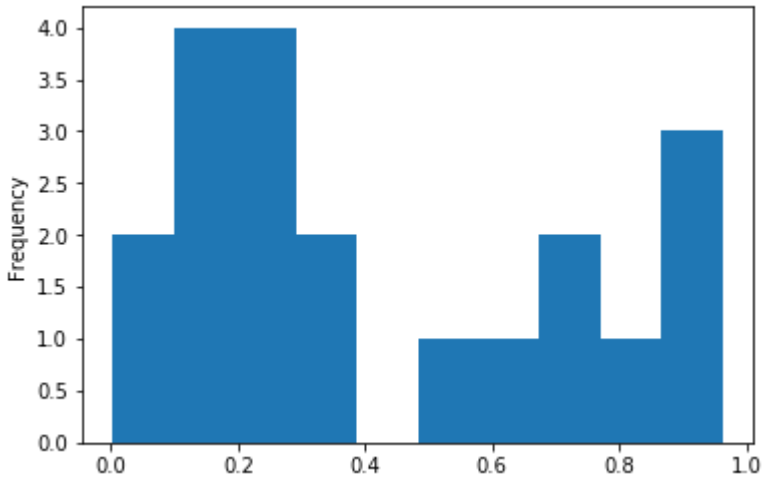
```
    Chooses k unique random elements from a population sequence or set.

    Returns a new list containing elements from the population while
    leaving the original population unchanged.  The resulting list is
    in selection order so that all sub-slices will also be valid random
    samples.  This allows raffle winners (the sample) to be partitioned
    into grand prize and second place winners (the subslices).

    Members of the population need not be hashable or unique.  If the
    population contains repeats, then each occurrence is a possible
    selection in the sample.

    To choose a sample in a range of integers, use range as an argument.
    This is especially fast and space efficient for sampling from a
    large population:   sample(range(10000000), 60)

seed(a=None, version=2) method of Random instance
    Initialize internal state from hashable object.

    None or no argument seeds from current time or from an operating
```

In [22]:
```
# genera un número aleatorio con dsitribución uniforme
np.random.rand()

```

Out[22]:
```
0.6482488570713822
```

In [23]:
```
# Generamos otro número aleatorio con distribución normal
lista=np.random.randn(100)
lista
#lista.plot.area()
dfl =pd.DataFrame({'X1':lista})
dfl.head()
x1.plot.hist()

```

Out[23]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1d14ab33be0>
```



In [24]:
```
# generamos un arreglo de 5 números aleatorios distribución uniforme

np.random.rand(5)
```

Out[24]:
```
array([0.52480589, 0.47801364, 0.56811539, 0.3635772 , 0.28420988])
```

In [25]:
```
# generamos una matriz de números aleatorios de 2 filas y 4 columnas
np.random.rand(2,4)

```

Out[25]:
```
array([[0.70656049, 0.9610001 , 0.96418141, 0.38559339],
       [0.93244932, 0.6778016 , 0.62711745, 0.09130302]])
```

Inicialización del generador de números mediante el método *ramdom.seed()*.

Aún cuando no es obligatorio, se sugiere utilizar este método para inicializar el generador de números aleatorios.

Ejemplo:

In [26]:
```
import random
# primer número generado con la misma semilla
random.seed(10)
print(random.random())
print("Observamos como el nuevo numero generado va a ser igual")
# segundo número generado con la misma semilla
random.seed(10)
print(random.random())

```

```
0.5714025946899135
Observamos como el nuevo numero generado va a ser igual
0.5714025946899135
```

Igual forma utilizando Random de numpy

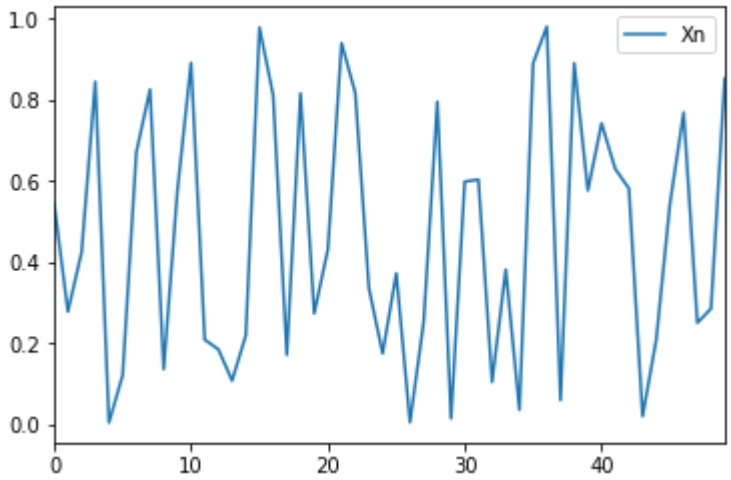
In [27]:
```
# podemos cambiar la semilla de la secuencia de # aleatorios generados
np.random.seed(100)
x = np.random.rand(50)
df = pd.DataFrame({'Xn': x})
df.head()

```

Out[27]:

	Xn
0	0.543405
1	0.278369
2	0.424518
3	0.844776
4	0.004719

In [28]:
```
df.plot()
```

Out[28]: `<matplotlib.axes._subplots.AxesSubplot at 0x1d14a2ebfd0>`



los dos secuencias de numeros aleatorios es la misma igual que el grafico

```
# Serie 2
np.random.seed(100)
y = np.random.rand(61)
print(y)
dfy = pd.DataFrame({'Yn': y})
dfy
```

In [29]:
```
# Graficamos la serie 2 que va a ser igua a la primera

dfy.plot()

```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-29-dd2c4b9847ee> in <module>
      1 # Graficamos la serie 2 que va a ser igua a la primera
      2
----> 3 dfy.plot()

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

Obtención de un número entero aleatorio dentro de un rango *random.randint()*.

Generará un número aleatorio que va dentro del rango ingresado.

In [ ]:
```
# generamos 10 números aleatorios con size=10 y los numeros entre 1 y 10
np.random.randint(10, size=10)

```

3.4 Test de números aleatorios

Una cuestión clave para un proyecto de simulación es saber si la secuencia de números aleatorios es suficientemente aleatoria. Como vemos en los generadores y gráficos anteriores que una secuencia aleatoria en función de su apariencia no es suficiente para determinar si un nùmero generado es **aleatorio"**, la realidad nos dice que antes de que realmente se use en simulación identifiquemos al generador, sus parámetros y que al menos se haya realizado alguna prueba que a continuación lo desarrollaremos.

```
 **"Ninguna cantidad de pruebas puede probar que un GNA dado sea perfect
o".**
```

Prueba de frecuencia en bloques

El enfoque de esta prueba es la proporción de ** 1 ** dentro de bloques de ** m ** bits. El proósito es determinar si una frecuencia de ** 1 ** dentro de un bloque es aproximadamente **m/2**, como se esperaría de la aleatoreidad.

Parámetros

```
- m longitud de cda bloque
- n longitud de la lista de datos
- vector lista con la secuencia de bits 1 y 0
```

Procedimiento

Procedimiento:

1. Partición de la secuencia en bloques de tamaño N, descartando bits no usados.

$$N = \left(\frac{n}{M}\right)$$

2. Determinar la proporción de unos con la fórmula:

$$\pi_i = \frac{\sum_{j=1}^{M} \varepsilon_{(i-1)M+j}}{M}, \text{ for } 1 \le i \le N.$$

3. Calcular la prueba estadística:

$$\chi^2(obs) = 4M \sum_{i=1}^{N} (\pi_i - 1/2)^2$$

4. Calcular P - value usando la fórmula:

$$\text{P-value} = chidist(\chi^2(obs), df)$$

Donde chidist es la distribución chi al cuadrado y df es el grado de libertad (número de bloques - 1)

5. Si el valor de P - value < 0.01, se concluye que la secuencia no es aleatoria. De otra forma la secuencia es aleatoria.

Procedimiento de: Emanuel Garcia (Modelado y Simulación)

In [ ]:
```
1
```

Type *Markdown* and LaTeX: α^2

In [30]:

```
# Prueba de Frecuencias en bloques

import math
from scipy.stats import chi2
rS
lista1=np.array(rS)
vector = []
# llenamos de 0 y 1
for i in range (0,len(lista1)-1):
    if lista1[i] < lista1[i+1]:
        vector.append(0)
    else:
        vector.append(1)
#vector
m = 10                          # numero de bloques
n = len(vector)                 # longuitud de la lista
N = int(n/m)                    # número de elementos de cada bloque

blocks = []
i = 0
while(len(blocks) != N):
    blocks.append(vector[i:(i+m)])
    i = i + m
sumas = []
for i in range(len(blocks)):
    sumas.append(0)
    for j in range(len(blocks[i])):
        sumas[i] = sumas[i] + blocks[i][j]/float(m)
x_2 = 0
for i in range(N):
    x_2 = x_2 + math.pow((sumas[i] - .5),2)
x_2 = x_2 * 4*m
df = m - 1
p_value = chi2.cdf(x_2, df)
if p_value < 0.01:
    print ("Test de frecuencia en Bloques: Not Pasa")
else:
     print ("Test de frecuencia en Bloques : Aprobado")

print ("Valor de P_value es %3f y el de Chi cuadrado es %3f " % (p_value, x_
print ("La proporción del bloque de sumas es:")

titulo = []
n=20
for i in range(n):
    titulo.append(float(i))

print(titulo)
print("Sumas")
print (sumas)

```

```
Test de frecuencia en Bloques: Not Pasa
Valor de P_value es 0.000000 y el de Chi cuadrado es 0.000000
La proporción del bloque de sumas es:
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 1
```

```
4.0, 15.0, 16.0, 17.0, 18.0, 19.0]
Sumas
[0.5]
```

3.5 Variables Aleatorias

Una variable aleatoria puede concebirse como un valor numérico que está afectado por el azar. Dada una variable aleatoria no es posible conocer con certeza el valor que tomará esta al ser medida o determinada, aunque sí se conoce que existe una distribución de probabilidad asociada al conjunto de valores posibles. (Wikipedia) Se emplea para nombrar a una función de valor real definida sobre un espacio muestral, es una función cuyo valor numérico depende del resultado de un experimento de azar.

Para definir las variables aleatorias Se utiliza letra mayúscula para la variable aleatoria y minúscula para la función de densidad:

F(x) Función de distribución acumulada de x

f(x) función de densidad de probabilidad (f.d.p) “función de densidad”.

Existen metodos para obtener variables aleatorias:

- Método de la transformada inversa
- Metodo del rechazo para proposito de este libro vamos a revisar el **método de la transformada inversa**

Método de la transformada inversa

Consiste en generar un numero aleatorio comprendido entre 0 y 1, luego identificarlo con la función de distribución acumulativa F(x), a través de la función inversa obtener el valor de x

1. Sea x el valor de la variable aleatoria y f(x) la función de densidad de probabilidad
2. Obtener la función acumulativa F(x)

$$F(x) = \int_0^x f(t)dt$$

$F(x)$ nos indica el valor del área bajo la curva

Algoritmo

1. Conocer f(x)
2. calcular

$$F(x) = \int_0^x f(t)dt$$

3. Generar un número aleatorio con distribución uniforme U(0,1)
4. Identificarlo y aplicar la inversa $F(x) = r$ o $r = F(x)$
5. $X = F^-1r$ entre $0 <= F(x) <= 1$

```
F(x) La función acumulativa indica la probabilidad que la variable aleat
oria tome un valor de x
```

Ejemplo 1

Generar los valores aleatorios de x con una f.d.p $f(x) = 2x$ entre $0 <= x <= 1$

1. $f(x) = 2x$
2. $F(x) = \int_0^x f(t)dt$
3. $F(x) = \int_{-\infty}^0 f(t)dt + \int_0^x f(t)dt$
4. $F(x) = t^2 | \ entre 0 - x$
5. $F(x) = x^2$
6. Generamos un número aleatorio r
7. $F(x) = r$ por lo tanto $x^2 = r$
8. calculamos la inversa $x = \sqrt{r}$

In [31]:
```
# Ejecución del Algoritmo
# Generamos numeros aleatorios por cualquiera de los metodos presentados, pa
# de Borland C/C++ xi+1=22695477xi + 1 mod 2^32
# en una DataFrame dfMCL
# Anexamos la columna de la transformada inversa de $x = √r$,  llanadole inv
# tomamos los datos del DataFrame
```

In [32]:
```
# probamos que este correcto

inv_xp = dfMCL[:1]
inv_xp

print(0.021137**1/2)
# resultado ok

```

```
0.0105685
```

In [33]:
```
# Graficamos los numeros

t1 = dfMCL["ri"]
t2 =inv_xp ##  dfMCL["inv_xp"]

x=range(200)
plt.figure(figsize=(50,15))

plt.subplot(131)
p1, = plt.plot(x,t1,'r-')
plt.xlabel('Secuencia x de números aleatorios')
plt.ylabel('ALEATORIOS')
plt.title('# ALEATORIOS POR METODO CONGRUENCIAL y TRANSFORMADA INVERSA')
#print(t2)
#plt.subplot(132)
#p2, = plt.plot(x,t2,'b-')
#plt.ylabel('ALEATORIOS')
#plt.title('# ALEATORIOS POR TRANSFORMADA INVERSA')
#plt.show()
t2.plot(marker="o")

```

```
C:\ProgramData\Anaconda3\lib\site-packages\pandas\plotting\_core.py:1001: Use
rWarning: Attempting to set identical left==right results
in singular transformations; automatically expanding.
left=0.0, right=0.0
  ax.set_xlim(left, right)
```

Out[33]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1d14bf35518>
```

Ejemplo 2 con la función exponencial

El tiempo entre llegadas **t** a una instalación se representa con una distribución exponencial con media

$$E(t) = 1/\lambda$$

unidades de tiempo; es decir,

$$f(t) = \lambda e^{-}\lambda t, t > 0$$

Determinar una muestra aleatoria de t a partir de f(t)

La función Acumulada es la siguiente:

$$F(t) = \int_0^x \lambda e - \lambda t dx = 1 - e - \lambda t, t > 0; 0at$$

- $R = F(t)$ despejar t
- $t = -(1/\lambda) ln(1 - R)$
- $(1 - R)$ es el complemento de **R** "remplazar $ln(1 - R)$ por $ln(R)$"
- $t = -(1/\lambda) ln(R)$
- **Ejm. λ = 4 clientes x por hora R = 0,9**
- **T1 = - (1/4) ln (1 – 0,9) = 0,577 ***
- ** t1 = -(1/4) ln (0.9) = 0.0263

tomado Ejemplo Taha, ejercicio 18.3-2

In [35]:
```
# comprobación del caso del libro de taha
import math
x = (1.0  - 0.9)
x1 = 0.9

lan=-1/4

print(math.log(x)*lan)
print(math.log(x1)*lan)


```

```
0.5756462732485115
0.02634012891445657
```

In [36]:
```
#  Calculamos para los números aleatorios del ejemplo anterior
# Landa = 4 clientes por hora
landa=4
dfexp = dfMCL['ri']
# calculamos a todos los elementos la inversa
exp_x = dfexp.values*(-1/landa)*np.log(dfexp)
# anexamos al Dataframa dfMCL
dfMCL["exp_x"] = exp_x
# Mostramos el resultado
dfMCL.head()

```

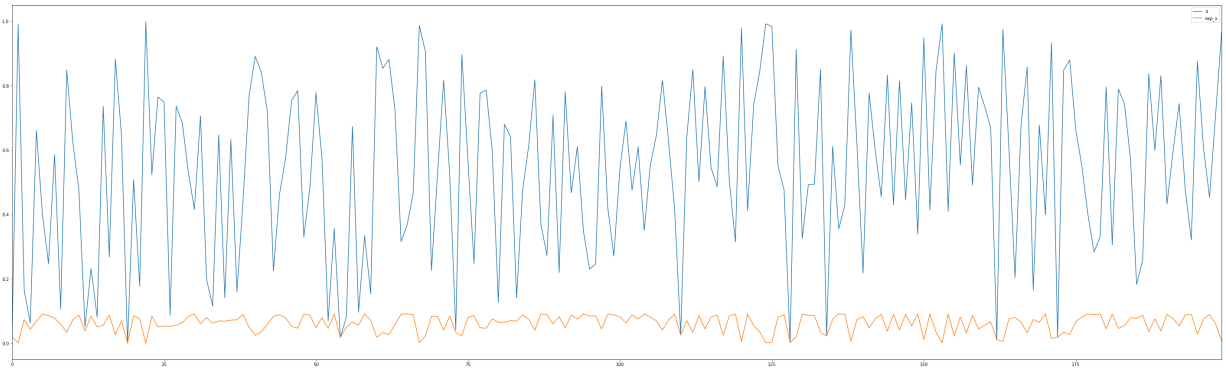
Out[36]:

	Xn	ri	exp_x
0	90781909	0.021137	0.020380
1	4261128730	0.992121	0.001962
2	705831779	0.164339	0.074192
3	274169216	0.063835	0.043910
4	2841246593	0.661529	0.068336

Type *Markdown* and LaTeX: α^2

In [37]:
```
dfgrafico = dfMCL.filter(items=['ri','inv_x','exp_x'])

dfgrafico.plot(figsize=(50,15))


```

Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x1d14bf7c8d0>



In [38]:
```
import math
x = 0.1
lan=-1/4

print(math.log(x)*lan)

```

```
0.5756462732485114
```

3.6 Distribuciones de Frecuencia

Las dos distribuciones importantes para simulación más utilizadas son la Exponencial y la de Poisson

Distribución de Poisson

la distribución de Poisson es una distribución discreta usadas en simulación donde el área de oportunidad para la ocurrencia de un evento es grande, si se conoce el número promedio "éxitos" que ocurren. se utiliza en muchos sistemas de la vida real:

- Numeros de defectos de un determinado material
- Las inperfecciones de una placa de vidrio
- Número de accidentes automovilisticos producidos en una intersección de vehiculos
- Número de multas que detecta un radar por día
- La llegada de clientes a un sistemas de líneas de espera

Distribución de Poisson

Variable aleatoria discreta, la cual con frecuencia resulta útil cuando tratamos con el número de ocurrencias de un evento durante un intervalo específico de tiempo o espacio.

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

λ = media o número medio de ocurrencias en un intervalo, Cantidad promedio de "éxitos"

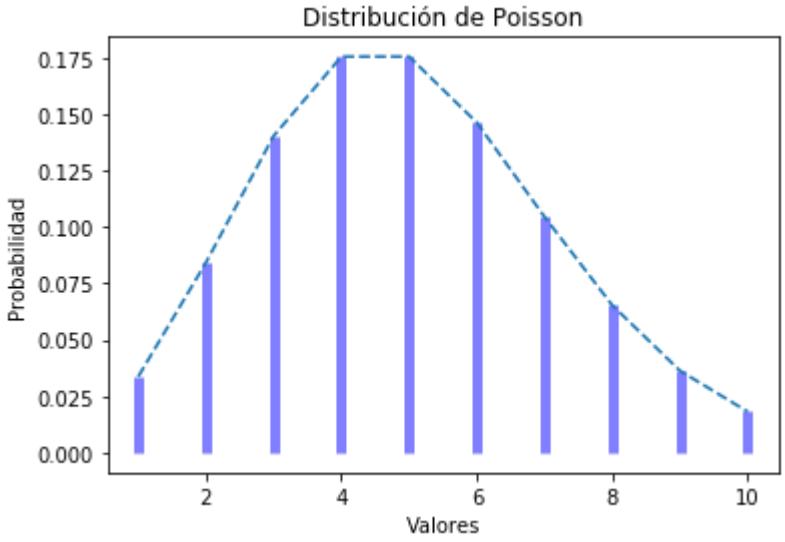
$e = 2.71828..$

$x = número de ocurrencias en el intervalo 0, 1, 2, 3, 4, 5 ... valores posibles de x$

$f(x)$ = probabilidad de x ocurrencias en el intervalo

Ejemplo

In [39]:
```
import numpy as np
import scipy.stats as stats
from scipy.stats import poisson
import matplotlib.pyplot as plt

landa = 5
poisson = stats.poisson(landa)
x = np.arange(poisson.ppf(0.01), poisson.ppf(0.99))

fmp = poisson.pmf(x)
plt.plot(x,fmp,'--')
plt.vlines(x,0,fmp,colors='b',lw=5,alpha=0.5)
plt.title('Distribución de Poisson')
plt.ylabel('Probabilidad')
plt.xlabel('Valores')
plt.show()

```



In [40]:
```
1 fmp
2
```

Out[40]:
```
array([0.03368973, 0.08422434, 0.1403739 , 0.17546737, 0.17546737,
       0.14622281, 0.10444486, 0.06527804, 0.03626558, 0.01813279])
```

In [41]:
```
1 x
```

Out[41]:
```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```