111-PROBA-Ejercicios1

April 23, 2018

0.1 1 PDF-Cap 11-Ej 1.1

Construyamos una función que tire una monera 1000 veces y que cuente las veces que sale cara, definiendo p como la probabilidad de que salga cara (si ponemos p=5/10 será una moneda perfecta y si ponemos otro número estará "trucada").

```
In [2]: def moneda(p):
            vecescara=0
            for tirada in srange(1000):
                x=random()
                if x<=p:
                    vecescara=vecescara+1
            return vecescara
```

```
In [3]: moneda(5/10)
```

```
Out[3]: 516
```

Como hemos puesto 5/10 sale un número muy cercano a 500, es decir, la mitad de las veces.

```
In [4]: moneda(1/10)
```

```
Out[4]: 109
```

Ahora, una vez construida la función vayamos a lo que nos pide el ejercicio.

```
In [5]: def pmenosde100(p):
            vecesquesalemenosde100=0
            for tirada in srange(10000):
                if tirada%1000 == 0:
                    print "Mil más procesados"
                if moneda(p)<=100:
                    vecesquesalemenosde100=vecesquesalemenosde100+1
            return (vecesquesalemenosde100/10000).n()
```

```
In [5]: [pmenosde100(k/10) for k in srange(1,10)]
```

```
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
```

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

KeyboardInterrupt                         Traceback (most recent call last)

<ipython-input-5-948a7786420e> in <module>()
----> 1 [pmenosde100(k/Integer(10)) for k in srange(Integer(1),Integer(10))]


<ipython-input-4-fa726f14eeb2> in pmenosde100(p)
  4         if tirada%Integer(1000) == Integer(0):
  5             print "Mil más procesados"
----> 6         if moneda(p)<=Integer(100):
  7             vecesquesalemenosde100=vecesquesalemenosde100+Integer(1)
  8     return (vecesquesalemenosde100/Integer(10000)).n()


<ipython-input-1-704b1b27396b> in moneda(p)
  2     vecescara=Integer(0)
  3     for tirada in srange(Integer(1000)):
----> 4         x=random()
  5         if x<=p:
  6             vecescara=vecescara+Integer(1)


src/cysignals/signals.pyx in cysignals.signals.python_check_interrupt (build/src/cysig


src/cysignals/signals.pyx in cysignals.signals.sig_raise_exception (build/src/cysignals


KeyboardInterrupt:
```

Con 10000 repeticiones del experimento de lanzar la moneda 1000 veces va a tardar demasiado. Hay que bajar ese número de repeticiones por ejemplo a 1000. De momento, dejamos $p = 0.1$.

```
In [6]: pmenosde100(1/10)

Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados
Mil más procesados


Out[6]: 0.529500000000000
```

Veamos la explicación de este resultado:

Calculemos la probabilidad "teórica" de obtener exactamente 100 caras en mil lanzamientos. La sucesión de mil lanzamientos se puede representar como una cadena de longitud 1000 de ceros y unos (cero para cara y uno para cruz). En cada lanzamiento de la moneda la probabilidad de obtener cara es 1/10 y lanzamientos sucesivos son "independientes", el resultado de los anteriores no influye en los que siguen. En estas condiciones, las probabilidades "se multiplican", es decir, la probabilidad de obtener dos caras en las primeras dos tiradas es 1/100.

La probabilidad de obtener 100 caras en las primeras 100 tiradas es $(1/10)^{100}$, y la probabilidad de que el resto de las 1000 tiradas sean cruces es $((1/10)^{100})((9/10)^{900})$. Para cada uno de las posibles cadenas de ceros y unos que contienen 100 ceros y 900 unos la probabilidad de aparición de esa cadena particular es la misma , e igual a $((1/10)^{100})((9/10)^{900})$.

£Cuántas cadenas hay que contengan 100 ceros y 900 unos? Basta elegir las 100 posiciones en las que colocamos ceros, ya que el resto van a ser unos necesariamente. Una ordenación tiene todos los ceros al comienzo, y hay 1000! reordenaciones de esta primera, pero muchas son iguales, por ejemplo, si intercambiamos el primer cero con el segundo queda la misma cadena. £Cuántas reordenaciones de la cadena que tiene los cien ceros al principio sigue teniendo los cien ceros al principio? Todas las que permuten los cien primeros ceros entre sí o los 900 unos entre sí. El número de esas reordenaciones es $100! \times 900!$, de forma que el número de cadenas con cien ceros y novecientos unos es

$$\binom{1000}{100} := \frac{1000!}{100! \times 900!}.$$

Entonces la probabilidad de obtener exactamente 100 caras es

```
In [7]: (binomial(1000,100)*(1/10)^100*(9/10)^(1000-100)).n()

Out[7]: 0.0420167908610854
```

La probabilidad de obtener "a lo más 100 caras" se obtiene sumando la de obtener 0 caras con la de obtener 1 cara, y así hasta la de obtener exactamente 100 caras:

```
In [8]: sum([(binomial(1000,n)*(1/10)^n*(9/10)^(1000-n)).n() for n in srange(0,101)])
```

```
Out[8]: 0.526599081295165

In [9]: [(binomial(1000,n)*(1/10)^n*(9/10)^(1000-n)).n() for n in srange(0,101)]

Out[9]: [1.74787125172265e-46,
         1.94207916858072e-44,
         1.07785393856230e-42,
         3.98406752105621e-41,
         1.10336536624807e-39,
         2.44211534396239e-38,
         4.49982364304181e-37,
         7.09972174791041e-36,
         9.79169957732644e-35,
         1.19918098527257e-33,
         1.32043150711680e-32,
         1.32043150711680e-31,
         1.20917292642455e-30,
         1.02107936009184e-29,
         7.99845498738610e-29,
         5.84183453152792e-28,
         3.99597709274653e-27,
         2.56996173808012e-26,
         1.55942740032886e-25,
         8.95530822878914e-25,
         4.88064298469008e-24,
         2.53070376983930e-23,
         1.25129241953165e-22,
         5.91190331546840e-22,
         2.67404145333918e-21,
         1.15993975931513e-20,
         4.83308233047970e-20,
         1.93721077773137e-19,
         7.47978605846279e-19,
         2.78557549763442e-18,
         1.00177548451964e-17,
         3.48287534044462e-17,
         1.17184243225376e-16,
         3.81933829771596e-16,
         1.20696082806906e-15,
         3.70134653941180e-15,
         1.10240722547296e-14,
         3.19135304911691e-14,
         8.98617832251341e-14,
         2.46287850320738e-13,
         6.57451733772859e-13,
         1.71044353501882e-12,
         4.33945859810330e-12,
         1.07421223177854e-11,
```

```
2.59601289346481e-11,
6.12787241025274e-11,
1.41355510912835e-10,
3.18801790569372e-10,
7.03282653732898e-10,
1.51819747472499e-9,
3.20845732991880e-9,
6.64059795952694e-9,
1.34656569734852e-8,
2.67619346139705e-8,
5.21472265008850e-8,
9.96591439794691e-8,
1.86860894961505e-7,
3.43853186829747e-7,
6.21175393065999e-7,
1.10197216623008e-6,
1.92028853411575e-6,
3.28792572325829e-6,
5.53290726548304e-6,
9.15320461203367e-6,
0.0000148898484747839,
0.0000238237575596543,
0.0000375003591216781,
0.0000580851333659159,
0.0000885513552784307,
0.000132898330305149,
0.000196394199228720,
0.000285831933149780,
0.000409780657247138,
0.000578807381925943,
0.000805637301869893,
0.00110521502449114,
0.00149462558136595,
0.00199283410848793,
0.00262020780930820,
0.00339779409308321,
0.00434634494406894,
0.00548509924354379,
0.00683036071113380,
0.00839393725946563,
0.0101815350091666,
0.0121912236188191,
0.0144121054408521,
0.0168233261467929,
0.0193935565303307,
0.0220810531281667,
0.0248343696293332,
0.0275937440325925,
```

```
0.0302931320357809,
0.0328628003446703,
0.0352323403222411,
0.0373339185169011,
0.0391055512243003,
0.0404941790455527,
0.0414583261656849,
0.0419701573529155,
0.0420167908610854]
```

Vemos que las cantidades que sumamos sólo superan las centésimas desde 84 caras, y

```
In [10]: sum([(binomial(1000,n)*(1/10)^n*(9/10)^(1000-n)).n() for n in srange(84,101)])
```

```
Out[10]: 0.488278109405186
```

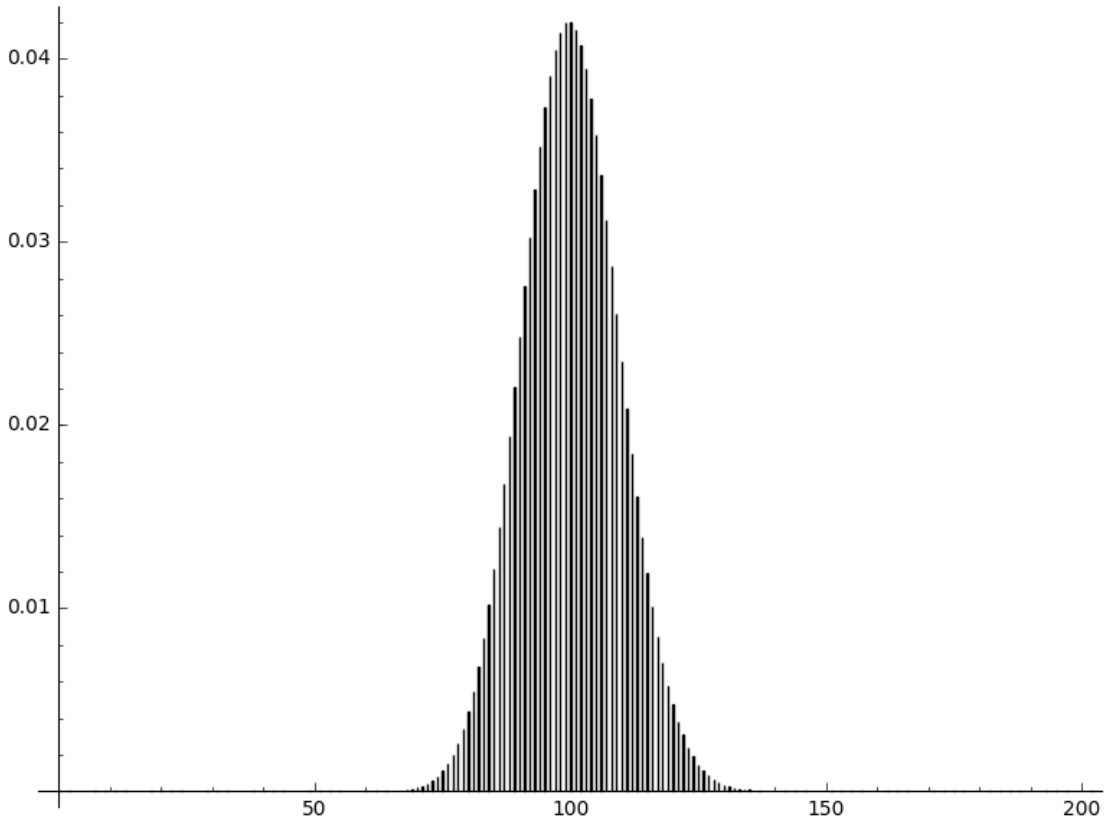
Aproximadamente, la operación que hacemos es $5(0.01 + 0.02 + 0.03 + 0.04) = 0.5$.

```
In [6]: L = [(binomial(1000,n)*(1/10)^n*(9/10)^(1000-n)).n() for n in srange(0,1001)]
```

```
In [7]: sum(L)
```

```
Out[7]: 1.00000000000000
```

```
In [8]: bar_chart(L,width=0.1).show(ymin=0,xmax=200)
```



El 100 queda, más o menos en el centro de esta "campana de Gauss" y la probabilidad de obtener a lo más 100 caras es la suma de los 101 primeros elements de la lista L, aproximadamente el área bajo la campana situada a la izquierda del cien.

0.2 2 PDF-cap. 11 Ej-1.2

Para generar un número aleatorio en el intervalo [0,1] solo tenemos que utilizar la función random().

```
In [14]: random()

Out[14]: 0.4829880359824984
```

Si multiplicamos random() por cualquier número (en particular será la longitud del intervalo, es decir, (b-a)) obtendremos un número entre 0 y ese número. Porque random() está entre 0 y 1.

Si ahora sumamos el comienzo del intervalo "a" desplazaremos el número al azar que estaba entre 0 y (b-a) adentro del intervalo [a,b] porque si x pertenece a (a,b) entonces x+t pertenece a (a+t,b+t).

```
In [15]: aleatorio(a,b)=a+(b-a)*random()

In [16]: aleatorio(3,7)

Out[16]: 3.193498189170919
```

0.3 3 PDF-Cap. 11-Ej 4

Queremos comprobar cómo al incrementar el número de dardos que lanzamos obtenemos una mejor aproximación a π.

```
In [1]: def f(n):
            dentro=[]
            fuera=[]
            for punto in srange(n):
                p=[random(),random()]
                if (p[0])^2+(p[1])^2<=1:
                    dentro.append(p)
                else:
                    fuera.append(p)
            return ((len(dentro))/n).n()

In [18]: f(10000)

Out[18]: 0.785500000000000

In [19]: def esbuenaaprox(n):
             L=[]
             k=1
```

```
            while k<=n:
                L.append(abs((f(k)-(pi/4)).n()))
                k=k*10
            return L
```

```
In [20]: esbuenaaprox(10^6)
```

```
Out[20]: [0.214601836602552,
          0.0146018366025518,
          0.0253981633974483,
          0.00860183660255176,
          0.00930183660255168,
          0.000918163397448324,
          0.000686836602551755]
```

Vemos que los errores a medida que avanza el k son menores luego cuantos más puntos elegimos más exacta es nuestra medición (lógicamente) y como además se reduce el error vemos que se aproxima a /4.

Ejercicio: adaptar este programa para usar los cuatro núcleos simultáneamente, y luego comprobar, usando la potencia de cálculo añadida, que la aproximación a π sigue mejorando cuando seguimos aumentando n.

0.4 4 PDF-Cap. 11-11.5.3

```
In [21]: def integrar(f,a,b,M):
             L=[]
             for k in srange(10000):
                 (x,y)=(a+(b-a)*random(),M*random())
                 if y<f(x):
                     L.append(x)
             return ((len(L))/10000)*(b-a)*M.n()
```

```
In [22]: integrar(x^3,1,2,8)
```

```
/usr/lib/sagemath/local/lib/python2.7/site-packages/sage/repl/ipython_kernel/__main__.py:1: Dep
See http://trac.sagemath.org/5930 for details.
  from ipykernel.kernelapp import IPKernelApp
```

```
Out[22]: 3.73760000000000
```

```
In [23]: integrate(x^3,x,1,2).n()
```

```
Out[23]: 3.75000000000000
```

```
In [ ]:
```