9.6. `random` — Gerador de números pseudo-aleatórios

Código fonte: Lib/random.py

Este módulo implementa geradores de números pseudo-aleatórios para várias distribuições.

Para inteiros , não é uniforme a partir de uma seleção de gama . Para as sequências, há seleção uniforme de um elemento aleatório, uma função para gerar uma permutação aleatória de uma lista no local, e uma função de amostragem aleatória sem reposição .

Na linha real, existem funções para calcular uniforme, normal (Gauss), lognormal , exponencial negativa, gama, beta e distribuições . Para gerar as distribuições de ângulos, a distribuição de von Mises está disponível .

Quase todas as funções do módulo depende da função básica random () , que gera uma flutuação aleatória uniforme no intervalo semi- aberto [0.0 , 1.0) . Python usa o Mersenne Twister como o gerador central . Ela produz carros alegóricos precisão de 53 bits e tem um período de 2 ** 19937-1 . A implementação subjacente em C é rápido e multitarefa. A Mersenne Twister é um dos geradores de números aleatórios mais extensivamente testados em existência. No entanto, sendo completamente determinista, ele não é adequado para todos os efeitos, e é totalmente inadequado para fins de criptografia .

As funções fornecidas por este módulo são realmente métodos de uma instância oculta da classe random.random vinculada. Você pode instanciar seus próprios casos de aleatório para obter geradores que não compartilham estado .

Classe Random também pode ser uma subclasse, se você quiser usar um gerador diferente básico de sua própria invenção: nesse caso, substituir o random (), seed (), GetState (), e setstate () métodos. Opcionalmente, um novo gerador pode fornecer um método ( ) getrandbits - isto permite randrange () para produzir seleções dentro de uma gama arbitrariamente grande .

O módulo random também fornece a classe SystemRandom que utiliza o sistema de os.urandom função ( ) para gerar números aleatórios a partir de fontes fornecidas pelo sistema operativo .

Aviso

Os geradores de pseudo - aleatório de este módulo não deve ser utilizado para fins de segurança . Use os.urandom () ou SystemRandom se você precisar de um gerador criptograficamente segura de números pseudo- aleatórios.

Funções da contabilidade :

`random.`**`seed`**(*a=None*, *version=2*)

Inicializa o gerador de números aleatórios.

Se a é omitido ou None, o estado atual do sistema de tempo é usado. Se fontes aleatórias são fornecidas pelo sistema operacional, eles são usados ao invés do sistema de tempo (veja a função os.urandom() para detalhes).

Se a é um inteiro, então é usado diretamente.

Com a versão dois (usada por definição), uma str, bytes, ou bytearray é convertido para inteiro e todos os seus bits são usados. Com a versão 1, o hash() de a é usado.

`random.`**`getstate`**()

Retorna um objeto que contêm o estado interno atual do gerador. O objeto retornado pode ser passado para setstate() para restaurar o estado.

`random.`**`setstate`**(*state*)

state deve ser obtido pela chamada de getstate(), e setstate() restaura o estado interno do gerador para o que era no momento em que getstate() foi chamado.

`random.`**`getrandbits`**(*k*)

Retorna um inteiro com um k aleatório de bits. Esse método pe usado com o gerador MersenneTwister e outros geradores podem também fornecer isso como uma parte opcional do API. Quando disponível, getrandbits() permite que randrange() lide com grandes ranges arbritários.

Funções para inteiros:

`random.`**`randrange`**(*stop*)

`random.`**`randrange`**(*start*, *stop*[, *step*])

Retorna aleatóriamente um element de range(começo, fim, passo). Isso é equivalente a choice(range(começo, fim, passo)), mas não constrói um objeto range.

`random.`**`randint`**(*a*, *b*)

Retorna um inteiro N de tal forma que a <= N <= b. Tal como randrange(a, b+1).

Funções para sequências:

`random.`**`choice`**(*seq*)

Retorna um elemento aleatória de uma sequência não-vazia. Se recebe uma sequência vazia devolve um erro do tipo IndexError.

`random.`**`shuffle`**(*x*[, *random*])

Embaralha uma sequência x. O argumento opcional random é 0-argumento da função retornando um número aleatório ente 0.0 e 1.0.

`random.`**`sample`**(*population*, *k*)

Retorna um lista de tamanho k com elementos únicos escolhidos da seuência population.

Retorna uma nova lista contend os elementos de population deixando a population original sem nenhuma troca.

As seguintes funções gerar distribuições específicas com valor de reais. Os parâmetros da função são nomeados após as variáveis correspondentes na equação de distribuição, como o usado na prática matemática comum; a maioria destas equações podem ser encontrados em qualquer texto de estatísticas.

`random.`**`random`**()

Retorna um real entre 0.0 e 1.0.

`random.`**`uniform`**(*a*, *b*)

Retorna um real N tal que a <= N <= b ou b <= N <= a.

random.**triangular**(*low*, *high*, *mode*)

Retorna um real aleatório N tal que low<= N <= high e o mode especificado entre esses limites. O mode por definição é o ponto médio entre os valores.

random.**betavariate**(*alpha*, *beta*)

Distribuição beta. Alpha > 0 e beta > 0, e se retorna um valor entre 0 e 1.

random.**expovariate**(*lambd*)

Distribuição exponencial. Lambd é 1.0 dividido pelo média desejada. Deve ser diferente de zero. Os valores retornados vão de zero ao infinito positivo se lambd é maior que zero, e ao infinito negativo se lambd < 0.

andom.**gammavariate**(*alpha*, *beta*)

Distribuição gamma. Alpha > 0 e beta > 0.

A probabilidade de distribuição é:

```
          x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu*, *sigma*)

Distribuição gaussiana. Mu é a média, e sigma o desvio padrão. Isso é um pouco mais rápido que normalvariate().

random.**lognormvariate**(*mu*, *sigma*)

Log normal de distribuição. Se você tomar o logarítmo natural da distribuição, você vai receber uma distribuição normal com média mu e desvio padrão sigma.mu, tendo qualquer valor, e sigma > 0.

random.**normalvariate**(*mu*, *sigma*)

Distribuição normal. Mu é a média, e sigma o desvio padrão.

random.**vonmisesvariate**(*mu*, *kappa*)

mu é o ângulo médio, em radianos entre 0 e 2*pi, e kappa é o parâmetro de concentração, que deve ser >= 0.

random.**paretovariate**(*alpha*)

Distribuiçao de pareto. Alpha é o parâmetro da forma.

random.**weibullvariate**(*alpha*, *beta*)

Distribuição de Weibull. Alpha é o parâmetro escalar e beta o parâmetro de forma.

Gerador alternativo

class `random.`**`SystemRandom`**([*seed*])

Classe que usa a função os.urandom() para gerar números aleatórios por fonts fornecidas pelo sistema operacional. Não está disponível em todos os sistemas. Não depende do estado do software, e as sequências não são reprodutíveis.

9.6.1. Notes on Reproducibility

Às vezes é útil para ser capaz de reproduzir as seqüências dadas por um número pseudo-aleatório generator. Re-utilizando um valor de semente, a mesma seqüência deve ser reprodutível de execução em execução, desde que vários segmentos não estejam rodando.A maioria dos algoritmos do módulo e funções de semeadura estão sujeitas a alterações em versões do Python, mas dois aspectos são a garantia de não mudar:
Se um novo método de semeadura é adicionado, em seguida, uma semeadora compatível com versões anteriores serão oferecidos.
Método aleatório do gerador () continuará a produzir a mesma seqüência quando a semeadora compatível é dada a mesma semente.

9.6.2. Exemplos e Receitas

Uso básico:

```
>>>
>>> random.random()                    # Random float x, 0.0 <= x
< 1.0
0.37444887175646646
```

```
>>> random.uniform(1, 10)              # Random float x, 1.0 <= x
< 10.0
1.1800146073117523
>>> random.randrange(10)               # Integer from 0 to 9
7
>>> random.randrange(0, 101, 2)        # Even integer from 0 to
100
26
>>> random.choice('abcdefghij')        # Single random element
```

```
'c'
>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]
>>> random.sample([1, 2, 3, 4, 5],  3)    # Three samples without
replacement
[4, 1, 5]
```

Uma tarefa comum é fazer random.choice com probabilidades com pesos diferentes.

Se os pesos são menores que inteiros, uma técnica simples é construer uma population com repetições.

```
>>>
```

```
>>> weighted_choices = [('Red', 3), ('Blue', 2), ('Yellow', 1),
('Green', 4)]

>>> population = [val for val, cnt in weighted_choices for i in
range(cnt)]

>>> random.choice(population)

'Green'
```

Uma solução mais genérica é arrumar os pesos numa distribuição comulativa com itertools.accumulate(), e então localizar o valor aleatório com bisect.bisect():

```
>>>
```

```
>>> choices, weights = zip(*weighted_choices)
>>> cumdist = list(itertools.accumulate(weights))
>>> x = random.random() * cumdist[-1]
>>> choices[bisect.bisect(cumdist, x)]
'Blue'
```