Randomly Sampling and Shuffling 1-D lists in Python and R

Thomas Walker

September 2022

1 Introduction

In this report, we will investigate how the programming languages go about sampling from a list of elements. We will cover multiple cases, including sampling with and without replacement as well as generating samples of different sizes. Furthermore, we will consider how programming languages shuffle the elements of a list. Some of these cases may overlap, for example, if we take a sample without replacement that is the same in size as the size of the list then we are effectively permuting the elements of the list. However, we ought to be careful as different there may be different functions to perform these operations within the programming and they may yield different results.

Throughout the report, we will be sampling/shuffling a list running from 0 to $n-1$. We do this for generality as for any applications we can use the outputted list to sample as an index list for an alias list and generate our sample in that way.

We reference various functions not defined in this report, refer to the primary and secondary report for the definitions of some of these functions.

2 Python

Let's first consider how python shuffles a list. For this we consider each element, in turn, sample from the set of indexes before this element uniformly and then simply swap the element we are at with the element at the index we have sampled. In code it will look something like this:

```python
def python_random_interval(max, state):
    if max==0:
        return 0
    mask = max
    mask|=mask>>1
    mask|=mask>>2
    mask|=mask>>4
    mask|=mask>>8
    mask|=mask>>16
    mask|=mask>>32
    if max<=0xffffffff:
        integer, state = mt19937_next(state)
        value = integer&mask
        while value>max:
            integer, state = mt19937_next(state)
            value = integer&mask
    else:
        integer, state = python_next_64(state)
        value = integer&mask
        while value>max:
            integer, state = python_next_64(state)
            value = integer&mask
    return value, state

def python_shuffle(x, state):
    n = len(x)
    for i in reversed(range(1,n)):
        j, state = python_random_interval(i, state)
        buf = x[j]
        x[j] = x[i]
        x[i] = buf
    return x, state
```

```
state = python_state_from_seed(1)
x = [0,1,2,3,4,5,6,7,8]
print(python_shuffle(x, state)[0])
rng.seed(1)
x = [0,1,2,3,4,5,6,7,8]
rng.shuffle(x)
print(x)
```

```
[8, 2, 6, 7, 1, 0, 4, 3, 5]
[8, 2, 6, 7, 1, 0, 4, 3, 5]
```

Python's NumPy library also has a function `numpy.random.choice` which allows us to sample elements from a 1-D array. We can specify whether this sampling is with or without replacement as well as the size of the sample to generate.

```
def python_random_choice(n, size, replace, state, p=None):
    if replace:
        if p is not None:
            cdf = p.cumsum()
            cdf /= cdf[-1]
            uniform_samples, state = python_random_uniform(state, size=size)
            idx = cdf.searchsorted(uniform_samples, side='right')
            idx = np.array(idx, copy=False).astype(int, casting='unsafe')
        else:
            idx = np.empty(size, dtype=int)
            for i in range(size):
                idx[i], state = python_random_interval(n-1, state)
    else:
        if p is not None:
            n_uniq=0
            p = p.copy()
            found = np.zeros(size, dtype=int)
            flat_found = found.ravel()
            while n_uniq<size:
                x, state = python_random_uniform(state=state,size=size-n_uniq)
                if n_uniq>0:
                    p[flat_found[0:n_uniq]]=0
                cdf = np.cumsum(p)
                cdf/=cdf[-1]
                new = cdf.searchsorted(x, side='right')
                _, unique_indices = np.unique(new, return_index=True)
                unique_indices.sort()
                new = new.take(unique_indices)
                flat_found[n_uniq:n_uniq + new.size] = new
                n_uniq += new.size
            idx = found
        else:
            shuffle_list, state = python_shuffle(np.arange(n, dtype=int), state)
            idx = shuffle_list[0:size]
    return idx, state
```

```
print('Sampling our list with replacement and not specifying the discrete␣
 ↪distribution to sample from.')
print('The output from our function:')
state = python_state_from_seed(1)
sample_list, state = python_random_choice(n=9, size=6, replace=True, state=state)
print(sample_list)
print("The output from NumPy's built-in choices function:")
x = [0,1,2,3,4,5,6,7,8]
rng.seed(1)
print(rng.choice(x, size=6, replace=True))
print('\nSampling our list with replacement and specifying the discrete␣
 ↪distribution to sample from.')
print('The output from our function:')
state = python_state_from_seed(1)
sample_list, state = python_random_choice(n=9, size=6, replace=True,␣
 ↪state=state, p=np.ones(9)/9)
print(sample_list)
print("The output from NumPy's built-in choices function:")
x = [0,1,2,3,4,5,6,7,8]
rng.seed(1)
print(rng.choice(x, size=6, replace=True, p=np.ones(9)/9))
print('\nSampling our list without replacement and not specifying the discrete␣
 ↪distribution to sample from.')
print('The output from our function:')
state = python_state_from_seed(1)
```

```
sample_list, state = python_random_choice(n=9, size=6, replace=False,␣
 ↪state=state)
print(sample_list)
print("The output from NumPy's built-in choices function:")
x = [0,1,2,3,4,5,6,7,8]
rng.seed(1)
print(rng.choice(x, size=6, replace=False))
print('\nSampling our list without replacement and specifying the discrete␣
 ↪distribution to sample from.')
print('The output from our function:')
state = python_state_from_seed(1)
sample_list, state = python_random_choice(n=9, size=6, replace=False,␣
 ↪state=state, p=np.ones(9)/9)
print(sample_list)
print("The output from NumPy's built-in choices function:")
x = [0,1,2,3,4,5,6,7,8]
rng.seed(1)
print(rng.choice(x, size=6, replace=False, p=np.ones(9)/9))
```

```
Sampling our list with replacement and not specifying the discrete distribution
to sample from.
The output from our function:
[5 8 5 0 0 1]
The output from NumPy's built-in choices function:
[5 8 5 0 0 1]

Sampling our list with replacement and specifying the discrete distribution
to sample from.
The output from our function:
[3 6 0 2 1 0]
The output from NumPy's built-in choices function:
[3 6 0 2 1 0]

Sampling our list without replacement and not specifying the discrete
distribution to sample from.
The output from our function:
[8 2 6 7 1 0]
The output from NumPy's built-in choices function:
[8 2 6 7 1 0]

Sampling our list without replacement and specifying the discrete
distribution to sample from.
The output from our function:
[3 6 0 2 1 4]
The output from NumPy's built-in choices function:
[3 6 0 2 1 4]
```

2.1 Remarks

Here we have shown that we can replicate the results of NumPy's choices function. We see that we get different samples by either specifying or not specifying the discrete distribution to sample from. Furthermore, we note that in this case generating a sample without replacement with size equal to the size of the list will give the same output as if we called our shuffle function directly (provided we do not specify the probabilities).

3 R

In R we have a similar function, `sample()`, which allows us to generate a sample from a list either with or without replacement. Furthermore, we can specify the size of the sample and the probability distribution against which we sample.

```
def r_rbits(bits, state):
    v=0
    n=0
    while n<=bits:
        u1, state = r_random_uniform(state)
        v1 = int(np.floor(u1[0]*65536))
        v = 65536*v+v1
        n+=16
    return v&((1 << bits)-1), state

def r_R_unif_index(dn, state):
    if dn<=0:
        return 0, state
    bits = int(np.ceil(np.log2(dn)))
    dv, state = r_rbits(bits, state)
    while dn<=dv:
        dv, state = r_rbits(bits, state)
    return dv, state

def r_random_sample_replace_prob(n, p, size, state):
    perm = np.empty(n)
    ans = np.empty(size, dtype=int)
    nm1 = n-1
    for i in range(n):
        perm[i]=i
    for i in range(1,n):
        p[i]+=p[i-1]
    for i in range(size):
        rU, state = r_random_uniform(state)
        for j in range(nm1):
            if rU[0]<=p[j]:
                break
        if rU[0]>p[j]:
            j=nm1
        ans[i]=(perm[j]+1)%n
    return ans, state

def r_random_sample_replace_noprob(n, size, state):
    y = np.empty(size, dtype=int)
    for i in range(size):
        j, state = r_R_unif_index(n, state)
        y[i]=j
    return y, state

def r_random_sample_noreplace_prob(n, size, p, state):
    perm = np.empty(n)
    ans = np.empty(size, dtype=int)
    for i in range(n):
        perm[i]=i
    totalmass=1
    n1=n-1
    for i in range(size):
        u, state = r_random_uniform(state)
        rT= totalmass * u[0]
        mass=0
        for j in range(n1):
            mass+=p[j]
            if rT<=mass:
                break
        if rT>mass:
            j=n1
        ans[i]=(perm[j]+1)%n
        totalmass-=p[j]
        for k in range(j,n1):
            p[k]=p[k+1]
            perm[k]=perm[k+1]
        n1-=1
    return ans, state

def r_random_sample_noreplace_noprob(n, size, state):
    x = np.empty(n)
    y = np.empty(size, dtype=int)
    for i in range(n):
        x[i] = i
    N=n
    for k in range(size):
        j, state = r_R_unif_index(N, state)
        y[k]=x[j]
        N-=1
        x[j] = x[N]
    return y, state
```

```
print('Sampling our list with replacement and specifying the discrete␣
 ↪distribution to sample from.')
print('The output from our function:')
state = (r_state_from_seed(1))
sample_list, state = r_random_sample_replace_prob(n=9, p=np.ones(9)/9, size=6,␣
 ↪state=state)
print(sample_list)
print("The output from R's built-in sample function:")
robjects.r('''
set.seed(1)
x<-c(0,1,2,3,4,5,6,7,8)
print(sample(x, replace=TRUE, prob=c(1,1,1,1,1,1,1,1,1), size=6))
''')
print('\nSampling our list with replacement and not specifying the discrete␣
 ↪distribution to sample from.')
print('The output from our function:')
state = r_state_from_seed(1)
sample_list, state = r_random_sample_replace_noprob(n=9, size=6, state=state)
print(sample_list)
print("The output from R's built-in sample function:")
robjects.r('''
set.seed(1)
x<-c(0,1,2,3,4,5,6,7,8)
print(sample(x, replace=TRUE, size=6))
''')
print('\nSampling our list without replacement and specifying the discrete␣
 ↪distribution to sample from.')
print('The output from our function:')
state = r_state_from_seed(1)
sample_list, state = r_random_sample_noreplace_prob(n=9, size=6, p=np.ones(9)/9,␣
 ↪state=state)
print(sample_list)
print("The output from R's built-in sample function:")
robjects.r('''
set.seed(1)
x<-c(0,1,2,3,4,5,6,7,8)
print(sample(x, replace=FALSE, prob=c(1,1,1,1,1,1,1,1,1), size=6))
''')
print('\nSampling our list without replacement and not specifying the discrete␣
 ↪distribution to sample from.')
print('The output from our function:')
state = (r_state_from_seed(1))
sample_list, state = r_random_sample_noreplace_noprob(n=9, size=6, state=state)
print(sample_list)
print("The output from R's built-in sample function:")
robjects.r('''
set.seed(1)
x<-c(0,1,2,3,4,5,6,7,8)
print(sample(x, replace=FALSE, size=6))
''')
```

```
Sampling our list with replacement and specifying the discrete distribution
to sample from.
The output from our function:
[3 4 6 0 2 0]
The output from R's built-in sample function:
[1] 3 4 6 0 2 0

Sampling our list with replacement and not specifying the discrete distribution
to sample from.
The output from our function:
[8 3 6 0 1 6]
The output from R's built-in sample function:
[1] 8 3 6 0 1 6

Sampling our list without replacement and specifying the discrete
distribution to sample from.
The output from our function:
[3 4 7 0 2 8]
The output from R's built-in sample function:
[1] 3 4 7 0 2 8

Sampling our list without replacement and not specifying the discrete
distribution to sample from.
The output from our function:
[8 3 6 0 1 5]
The output from R's built-in sample function:
[1] 8 3 6 0 1 5
```

3.1 Remarks

The sample function in R lets us set the probabilities for the discrete distribution we wish to sample from. We note that when we set the probabilities all equal to each other we get a different sample to the sample where we do not specify the probabilities. By default R still samples uniformly when the probabilities are not given, however, it uses a different algorithm to do so and yields different samples. We note that for the case where we are not specifying the probabilities, the samples obtained when we sample with or without replacement are the same up to the point where we sample a duplicate element in the list.