스트림 패러다임으로 꾸민 몬테카를로 방법

Monte Carlo Methods in Stream Paradigm, Python Edition

김재우

2022-02-17

목차

---

1 계산 절차를 간추리는 방법[1]

프로그램을 잘 짜려면 얼개가 튼튼해야 합니다. 아래는 프로그램 짜는 법을 처음 배우기 시작할 때 누구나 흔히 풀어보는 문제입니다.[2] 하지만 프로그램 짜는 기본기 곧 컴퓨터로 계산하는 절차를 간추리는 연습을 꾸준히 하는데 이보다 알맞은 문제도 드뭅니다. 제대로 간추리는 방법을 알면 단순한 코드의 품질도 몰라보게 달라집니다.

프로그램 짜는 기본기를 잘 갖추고 있는지 점검하기 위해서 아래 세 함수의 같고 다른 점을 찾아서 계산 절차를 간추려 보기로 합니다.

[1] 2022년 아카데미 경남 캠퍼스 1기 기술 면접 1번 문제. 프로그램 짜는 틀이 잡혔는지 알아보려고 낸 문제다. 정답을 맞힌 사람이 없어서 한 번 더 풀어보라고 다시 문제지의 1번 문제로 내놓았다. 이 번에는 의도를 알아차리기 쉽도록 코드 틀을 마련하여 채우는 방식으로 꾸몄다. 이 문제는 문제지가 끌고 가려는 이야기 흐름에서 조금 벗어난 시작이다. 하지만 이 문제지 자체가 2022년 경남 아카데미 1기의 학기 시작 전 공부 거리로 나가는 것이라 정답지를 대신하는 것이라 봐도 좋다.

[2] Harold Abelson, Gerald Jay Sussman, Julie Sussman, “Structure and Interpretation of Computer Programs”, Chapter 2.2.1,3 (2nd ed.) 1996

1. a에서 b까지 정수를 모두 더하는 함수
2. a에서 b까지 정수를 모두 곱하는 함수
3. 마구잡이 수를 모두 곱하거나 더할 수 있는 함수

값을 차례대로 늘어놓은 순열(sequence) xs의 값을 이항 연산자 glue로 누적하는 함수 fold를 정의합니다. 인자 identity는 glue 연산자의 항등원(identity element)이거나 초기 값일 수 있습니다. [3]

```
def fold( identity, glue, xs):
    result = identity
    for x in xs:
        result = glue( result, x )
    return result
```

1번 문제의 답이 되는 함수를 sum이라고 합시다. sum과 fold의 계산 절차가 거의 같다는 것을 알아차리는 것이 중요합니다. 계산 절차를 간추려낸 fold가 있기 때문에 다른 점만 인자로 건네주면 순열 xs로부터 차례대로 열거되는 모든 수의 합을 구하는 함수 sum을 정의할 수 있습니다. sum 함수를 def 키워드로 정의하지 않고 lambda 연산자의 값으로 정의할 수 있다는 점도 눈여겨 봅시다.

```
sum = lambda xs: fold( 0, lambda x, y: x + y, xs )
```

sum으로 1번 문제를 푸는 sum_integers를 정의합니다.

```
sum_integers = ...
print( sum_integers( 1, 10 ) )
```

```
55
```

마찬가지로 fold를 써서 순열 xs의 모든 수를 곱하는 함수 product와 2번 문제를 푸는 product_integers을 정의합니다.

```
product = ...
product_integers = ...
print( product_integers( 1, 10 ) )
```

```
3628800
```

fold는 마구잡이 수열을 모두 더하거나 곱하는 3번 문제에도 그대로 쓸 수 있습니다.

```
from random import random

n = 20
print( sum( random() for _ in range( 0, n ) ) )
print( ... )
```

```
10.00683497691481
5.031436054307361e-12
```

fold처럼 되쓰임새가 높은 함수를 정의할 수 있는 이유는 순열 xs가 데이터를 만드는 코드(계산 절차)와 데이터를 쓰는 코드를 서로 떼어 놓았기 때문입니다. 여러 프로그래밍 언어에서 이런 얼개로 코드를 짜맞추는 방식을 두고 iterator 패턴이라고 하고 xs를 iterator (enumerator, list)라고 합니다.

[3] 항등원과 함께 이항 연산자 glue의 결합 법칙도 중요하다. 계산의 방향에 따라 결과가 다르기 때문이다. 덧셈, 곱셈 말고 뺄셈, 나눗셈이 들어가는 경우를 생각해보면 된다. 이 때문에 fold-left, fold-right로 어느 쪽으로 연산 결과를 쌓아가는지 구분지어 정의하는 편이 더 나을 수도 있다.

2 끝없는 계산

순열 곧 iterator를 써서 계산을 간추리면 얻을 수 있는 장점이 또 하나 있습니다. 데이터를 쓰는 코드에서 계산이 언제 끝나는지를 굳이 드러내지 않아도 됩니다. 데이터 열거의 시작과 끝을 모두 iterator 속에 간추릴 수 있기 때문입니다. 따라서 iterator를 쓰면 끝나는 계산과 끝없는 계산을 굳이 구분하지 않고 한 가지 계산으로 간추릴 수 있습니다.

하지만 계산 자원에는 분명히 끝이 있는데 끝없는 계산을 표현하는 것이 가능한 일일까요?

2.1 스트림

값 또는 데이터들이 언제 얼마나 쓰일지를 미리 알 수 없는 경우가 많습니다. 또, 이를테면 마우스로 클릭하는 좌표 값을 연이어 받아서 무언가를 하는 프로그램을 짜는 경우처럼 미리 데이터를 만들어 둔다는 게 아무런 의미가 없을 때도 있습니다. 이럴 때 데이터를 미리 만들어 저장해 두지 않고, (필요할 때 데이터를 만들어 낼 수 있도록) 계산하는 방법 그 자체만 따로 솎아낼 수 있다면 계산 자원을 선점하는 낭비를 줄이고 계산 방법 또는 문제를 푸는 방법을 적은 코드와 계산 또는 문제 풀이를 적은 코드를 분리하여 코드의 되쓰임새를 한층 더 끌어올릴 수 있습니다.

데이터 만드는 계산을 데이터를 쓰는 코드로부터 완전히 격리하기 위해서 널리 쓰는 방법 가운데 하나는 지연 계산법 (delayed evaluation)을 써서 순열을 표현하는 방법입니다. 스트림 (Stream)이라고 불리기도 합니다.[4]

먼저 계산을 미루는 연산이 필요합니다.

```
def delay( v ):
    return lambda: v
```

반대로 계산을 하도록 만드는 연산도 필요합니다.

```
def force( v ):
    return ...
```

새로운 스트림을 만들려면 빈 스트림을 표현하는 연산과 스트림에 새로운 원소를 덧붙이는 연산이 필요합니다. 이런 연산들을 아울러 constructor라고 합니다. xs를 delay 연산으로 감싸서 계산을 뒤로 미룹니다.

```
EmptyStream = None
def stream(x, y):
    return ( x, delay( y ) )
```

스트림의 부품을 머리와 꼬리로 구분 짓는 연산자도 필요합니다. 이런 연산들을 아울러 selector라고 합니다. 꼬리를 떼어낼 때 계산이 일어나도록 force 연산을 씁니다.

```
def head( stream ):
    (x, _) = stream
    return x


def tail( stream ):
    (_, xs) = stream
    return ...
```

스트림 연산으로 a에서 b에 이르는 정수의 순열을 표현합니다.

```
def integers( a, b ):
    if a > b: return EmptyStream
    return stream( a, ... )
```

스트림 원소마다 함수 f를 적용하는 절차를 간추려 foreach로 정의합니다.

[4] Harold Abelson, Gerald Jay Sussman, Julie Sussman, “Structure and Interpretation of Computer Programs”, Section 3.5.1 Streams Are Delayed Lists의 아이디어를 빌렸다.

```
def foreach( f, stream ):
    if stream == EmptyStream: return None
    else: ...
    return foreach( f, tail( stream ) )
```

스트림 원소를 보는데 print()를 쓰지는 못합니다. 스트림 원소를 차례대로 찍은 절차를 적는데 foreach를 써 봅니다.

```
def print_stream( stream ):
    foreach( ... print( ... , end=' ' ), stream )
    print()

print_stream( integers( 1, 10 ) )
```

```
1 2 3 4 5 6 7 8 9 10
```

끝나는 정수열은 잘 찍힙니만, 끝없는 정수열을 늘어놓는 데 스트림 연산을 쓸 수 있을까요?

```
def integers_from( n ):
    return stream( n, integers_from( n + 1 ) )

print_stream( integers_from( 1 ) )
```

스택이 넘쳤다고 투덜대면서 프로그램이 멈춥니다.

```
    RecursionError                            Traceback (most recent call last)
    ...
        [... skipping similar frames: integers_from at line 2 (2970 times)]

    RecursionError: maximum recursion depth exceeded
```

계산 방법에는 잘못이 없습니다. 하지만 계산 결과는 틀렸습니다.[5]

Python 같은 언어에서는 이런 아이디어를 그대로 가져다 쓸 수 없습니다. Java, Python 같은 언어에서는 list, array 같이 끝나는 순열(sequence)을 끝없는 순열과 구분하지 않고 하나로 간추려 낼 수 없습니다. 그 때문에 빚어지는 프로그램 설계 문제를 줄이고자 표현력의 결핍을 막는 방편으로 iterator라는 패턴 또는 비슷한 이름의 부품들을 씁니다.[6]

2.2 대안을 찾아서

언어마다 표현법이 다르지만 iterator로 끝없는 데이터 목록을 표현할 수 있습니다.

정수 n부터 끝없이 정수를 열거하는 방법을 표현합니다.

```
def integers_from( n ):
    while True:
        yield( n )
        n += 1
```

범위를 지정하여 열거하는 연산은 이미 있습니다.

[5] 인자에 대응하는 식의 계산부터 먼저 끝낸 다음에 함수 식을 펼쳐 계산하는 방식을 '인자먼저 계산법(applicative-order evaluation 또는 eager evaluation)'이라고 한다. 이와 달리, 함수를 정의하는 식에서 인자에 대응하는 자리에 인자로 건넨 식을 그대로 펼친 다음에 계산하는 방식을 두고 '정의대로 계산법(normal-order evalation 또는 delayed evaulation)'이라고 한다. Python 언어를 포함해서, 프로그래밍 언어 대부분이 인자를 먼저 계산하는 방법을 쓰는데 이런 언어에서는 끝없이 펼쳐지는 값을 함수로 정의하는 것이 불가능하기 때문에 다른 기법이 필요하다. 한편, 끝없는 계산을 표현하려면 식의 계산을 최대한 뒤로 미룰 수 있는 방법이 있어야 하는데 정의 대로 계산하는 법을 따르는 언어라면 특별한 기능이나 기법을 쓰지 않고 보통 함수를 정의하는 수단만 있으면 된다.

[6] Iterator를 만드는 데도 여러가지 기법이 있지만 여기서는 주로 (python 언어에서 generator라고하는) yield 연산으로 함수 내에서 값을 열거하는 방법을 쓰기로 한다.

```
print( range(1, 11) )
```

```
range(1, 11)
```

이로부터 iterator를 얻어내는 연산도 있습니다.

```
print( iter(range(1, 11)))
```

```
<range_iterator object at 0x7fb888fd5a80>
```

연산 결과를 찍어봐야 정수열을 볼 수는 없습니다. 두 연산 모두 계산을 하지 않기 때문입니다. 끝없는 계산의 순열을 표현하지만 때로는 계산을 하도록 만들어서 그 값을 찍는 연산도 필요합니다.

```
def print_iterator( xs ):
    for x in xs:
        print(x, end=' ')
    print()


print_iterator( iter(range(1, 11)))
```

```
1 2 3 4 5 6 7 8 9 10
```

하지만 끝없는 연산을 찍으려하면 그 또한 끝나지 않습니다. 끝없는 계산을 n번 계산으로 끝내는 연산이 필요합니다.

```
def finite( n, xs ):
    for i in range(1, n+1):
        yield( next(xs) )


print_iterator( finite( 100, integers_from( 1 ) ) )
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
↪  38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
↪  70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

finite는 여전히 계산을 미룹니다. 계산이 일어나도록 만드는 간편 함수가 있으면 편리합니다.
계산 결과를 목록(list)에 저장하면서 비로소 계산을 하게됩니다.

```
def take( n, xs ):
    return list( ... )
```

피보나치 수 10개를 곧바로 찍을 수 있습니다.

```
def fibonacci_numbers():
    a, b = 0, 1
    while True:
        yield( a )
        ...


print( take( 10, fibonacci_numbers() ) )
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

끝없는 짝수열에서 10개만 찍는 코드를 씁니다. 그 과정에서 되쓰임새가 또렷한 계산 절차를 함수 세 개로 나누어 간추렸습니다.

```
def filter( istrue, xs ):
    ...


def isdivisible( x, y ):
```

```
    return x % y == 0

def iseven( x ):
    return isdivisible( x, 2 )

print_iterator( finite( 10, filter( iseven, integers_from( 1 ) ) ) )
```

n번째 계산 결과만 출력하는 연산이 있으면 쓸모가 많습니다. 이를 테면 fold, sum을 iterator 패턴에 맞게 다시 정의하고 10까지 더한 값을 구한다고 하면 계산 과정에서 얻은 값을 저장할 필요가 없습니다.

```
def index( n, xs ):
    ...

def fold( glue, identity, xs ):
    ...

def sum( xs ):
    return fold(  lambda x, y: x + y, 0, xs )

print( index( 10, sum( integers_from( 1 ) ) ) )
```

```
55
```

끝없는 계산은 끝없이 찍힙니다.

```
def map( f, xs ):
    ...
# 1 2 ... infinite!
print_iterator( map( lambda x: x + 1, integers_from(0)))
```

그런데 이런 연산을 어디다 쓸 수 있을까요?

3 몬테카를로 시뮬레이션

몬테카를로 방법은 여러 분야에서 시뮬레이션을 하는데 널리 쓰입니다. 일단 끝없는 순열은 쓰지 않고 계산 절차만 간추려 봅니다.

참 거짓을 답하는 함수(실험)를 n 번 시행하고 참이 나온 횟수를 n으로 나눕니다.

```
def monte_carlo( n_trials, truths ):
    return sum( [ truths() for i in range( n_trials+1 ) ] )  / n_trials
```

3.1 어림 값

몬테카를로 방식으로 π 값을 어림잡을 수 있습니다. 두 마구잡이 수가 서로 소인지 알아보는 함수를 만듭니다.

```
from math import gcd, sqrt
from random import randint

def dirichlet_test():
    return gcd( randint(1, 1000), randint(1, 1000) ) == 1
```

몬테카를로 방식으로 값을 구합니다.

```
def quess_pi( n_trials ):
    return sqrt( 6 / monte_carlo( n_trials, dirichlet_test ) )

print( quess_pi( 100_000 ))
```

```
3.139623287689948
```

십만 번 정도 시행하면 π 값에 다가갑니다.

3.2 끝없는 실험

몬테카를로 실험에 iterator 기법을 적용해 봅니다.

먼저 어떤 함수의 계산 값을 끝없이 늘어놓은 함수를 만듭니다.

```
def repeat( f ):
    ...

from numpy.random import randint, random, normal

print_iterator( take( 10, repeat( random ) ) )
```

```
0.9280550225904395 0.22280097961036138 0.10318922203641656 0.5838716457253749 0.03550028920191295
↪ 0.36444575313103367 0.3816809748705672 0.326481870159794 0.4301836750394781 0.24527212163771261
```

몬테카를로 iterator를 정의합니다. 오로지 계산 방법 그 자체만을 있는 그대로 표현한 코드를 쓸 수 있습니다. 계산 시간(횟수)와 계산 공간 문제 곧 계산 자원 문제를 계산 방법으로부터 격리할 수 있습니다.

```
def monte_carlo( experiment ):
    n, sums = 0, sum( repeat( experiment ) )
    for s in sums:
        n = n + 1
        yield( s / n )
```

π 값으로 끝없이 수렴하는 수열을 표현할 수 있습니다.

```
from math import gcd, sqrt

pi = map( lambda x: 0 if x == 0 else sqrt( 6 / x ), monte_carlo( dirichlet_test ) )
```

끝없는 수열 pi에서 10만 번째 값만 뽑습니다.

```
print( index( 100_000, pi) )
```

```
3.1405936558730163
```

3.3 응용

Iterator 패턴과 몬테카를로 방법을 실제 어떻게 쓰는지 그 장단점을 또렷이 드러내는 본보기로 약재와 약물의 관계를 모의 실험하는 예제를 다룹니다. 특히 끝없는 이어지는 데이터를 순열(스트림)로 간추리는 기법은 시계열 데이터 시뮬레이션을 다루는 데 잘 들어맞습니다. 특히 이동 통신 환경에서 통신망 저편에서 흘러 들어오는 수백 개의 센서 데이터 스트림을 떠올려 보면 어렵지 않게 끝없는 데이터 순열을 떠올릴 수 있습니다.[7]

[7] 이 다음 단계인 시계열 데이터 시뮬레이션 곧 Quasi-Continous Simulation 문제에서 다루기로 합니다. 1기 자바 웹 서비스 개발자 과정에는 공식 교과로 포함되지 않습니다. 실무 개발자 과정에 앞선 기초 이수 과목 포함되어야 하는 내용이기 때문입니다.

3.3.1 확률 분포

시뮬레이션에서는 가짜(의사) 마구잡이 수(난수)를 정해진 분포에 따라 늘어 놓은 일이 필요합니다. repeat 함수로 널리 쓰이는 이항, 균등, 정상 분포 함수를 준비합니다.

```
def binomial_distribution( success_ratio ):
    if success_ratio < 0 or success_ratio > 1:
        raise ValueError( "binomial_distribution: "
                          + "success ratio is out of range ( 0 to 1 only )" )
    return repeat( lambda : random() ≤ success_ratio )
# binomial_distribution( 1.2 ) # error!
# print_iterator( take( 10, binomial_distribution( 0.5 ) ) )

def discrete_uniform_distribution( low, high ):
    return repeat( lambda: randint( low, high + 1 ) )
# print_iterator( take( 10, discrete_uniform_distribution( 1, 4 ) ) )

def normal_distribution( mean, standard_deviation ):
    return repeat( lambda: normal( mean, standard_deviation))
```

3.3.2 약효 시뮬레이션

확률 분포와 몬테카를로 방법으로 재밌는 실험을 할 수 있습니다. 돌림병을 치료하는데 꼭 필요한 약초가 있다고 합시다.[8]

좋은 약초를 발견할 확률이 herb_ratio일 때 마구잡이 수를 뽑아 이 값과 크기를 비교하면 참 거짓을 정할 수 있습니다. 이를 연속 시행하면 참 거짓의 이항 분포를 얻을 수 있습니다. 좋은 약초 발견 가능성을 이항 분포로 뽑아냅니다.

```
herb_ratio = 0.2
herb_availablities = binomial_distribution(herb_ratio)
```

약초의 품질을 넷으로 나눕니다.

```
Excellent, Good, Marginal, Poor = 0, 1, 2, 3
```

좋은 약초는 Excellent 품질. 좋은 약초가 없으면 다른 약초를 여럿 섞어서 대신 쓰는데 그 품질이 고르지 않습니다. 좋은 약초가 있느냐 없느냐에 따라 약초의 효과를 어림잡는 함수를 만듭니다.

```
def guess_quality( qualities ):
    return lambda isavailable: Excellent if isavailable else next( qualities )
```

guess_quality가 함수를 값으로 내놓고 있다는 걸 눈여겨 보아야 합니다. 이런 표현을 쓰면 간결한 코드를 쓸 수 있습니다. 하지만 코드를 읽기 어렵게 만들 수 있습니다.

네 가지 품질 가운데 하나가 고르게 뽑히도록 이산 균등 분포를 씁니다.

```
herb_qualities = map(  guess_quality(
                                  discrete_uniform_distribution(
                                      Excellent, Poor ) ),
                       herb_availablities )
```

약초 품질에 따라 약물의 효과가 갈리지만 약초 품질이 고르지 않으므로 약물의 효과도 편차가 있는 것이 자연스럽습니다. 약초의 품질이 떨어질 수록 약효의 편차도 큽니다. 실감나는 시뮬레이션을 위해서 약효를 네 가지로 나누고 약효에 따른 편차는 정상 분포를 따르는 마구잡이 수로 표현합니다.

[8] Wolfgang Kreutzer, “System Simulation Programming Styles and Languages,” 1986, pp. 17-29

```
effect_distributions_by_quality = [
    normal_distribution( mean, standard_deviation )
    for ( mean, standard_deviation ) in [
        ( 90, 10 ),
        ( 80, 20 ),
        ( 50, 30 ),
        ( 30, 40 )
    ]]
```

약초의 품질에 따른 약물 효과를 마구잡이로 뽑는 함수를 만듭니다. 약물 효과는 0에서 100사이 값이므로 이 범위를 넘는 값을 잘라냅니다.

```
def quess_effect_from( quality ):
   ...
   return guess
```

약초의 품질로부터 약효를 끝없이 계산하는 순열 effects를 정의합니다.

```
effects = map( quess_effect_from, herb_qualities )
```

3.3.2.1 계산

계산이 시작되려면 시행 횟수를 정해서 약물 효과 추정치를 뽑아내고 계산 결과를 목록에 저장해야 합니다. 계산하는 방법만 적었지 실제 계산을 하지는 않았기 때문에 계산 방식을 저장해둔 계산 환경 자원을 빼면 계산하는 과정에서 필요한 계산 자원이 조금도 소비되지 않았다는 사실을 정확히 인식하고 이해하는 것이 중요합니다. 계산하는 방법과 계산을 나누면 어떤 방식으로 프로그램을 설계할 수 있는지가 잘 드러나 있습니다.

take로 7000번 계산을 합니다. 끝없는 계산 순열 effect가 7000번으로 끝나도록 만든 뒤에 계산 결과를 순서대로 하나씩 목록에 저장합니다. 계산 순열 effects에는 다른 계산 순열 herb_qualities가 연결되어 있고 이는 다시 herb_avaliablities로 연결되어 있기 때문에 한 계산이 다른 계산으로 이어집니다.

```
number_of_trials = 7000
neffects = take( number_of_trials, effects )
```

약물 효과 시뮬레이션한 결과를 막대 그래프로 표현할 수 있습니다.

```
import matplotlib.pyplot as plot

plot.hist( neffects, rwidth=0.9 )
plot.xlabel("Potion effect")
plot.ylabel("# Samples")
plot.grid()
plot.show()
```

3.4 상태가 필요해

약효 모의 실험은 잘 되지만 실험은 결과 만큼 과정에 대한 정보도 중요합니다. 계산 과정에 쓴 데이터들 곧 약초와 품질, 그에 따른 약물의 품질, 그리고 약효에 대한 데이터가 될 수 있는 한 상세하게 보고서로 나와야 쓸모있는 실험이 됩니다. Iterator를 쓰는 코드 얼개를 조금도 망가뜨리지 않고 필요한 정보만 저장했다가 꺼내보는 방법이 필요합니다.

계산과 저장을 가능한 하지 않고 미루는 방식과 계산 과정에서 얻은 정보를 적절히 저장하는 두 가지 다른 방식을 한 데 엮어 쓰는 본보기가 됩니다. 문제는 그에 맞는 풀이 방법이 제 각기 다를 수 있습니다. 굳이 한 가지 방법만을 고집할 필요가 없습니다.

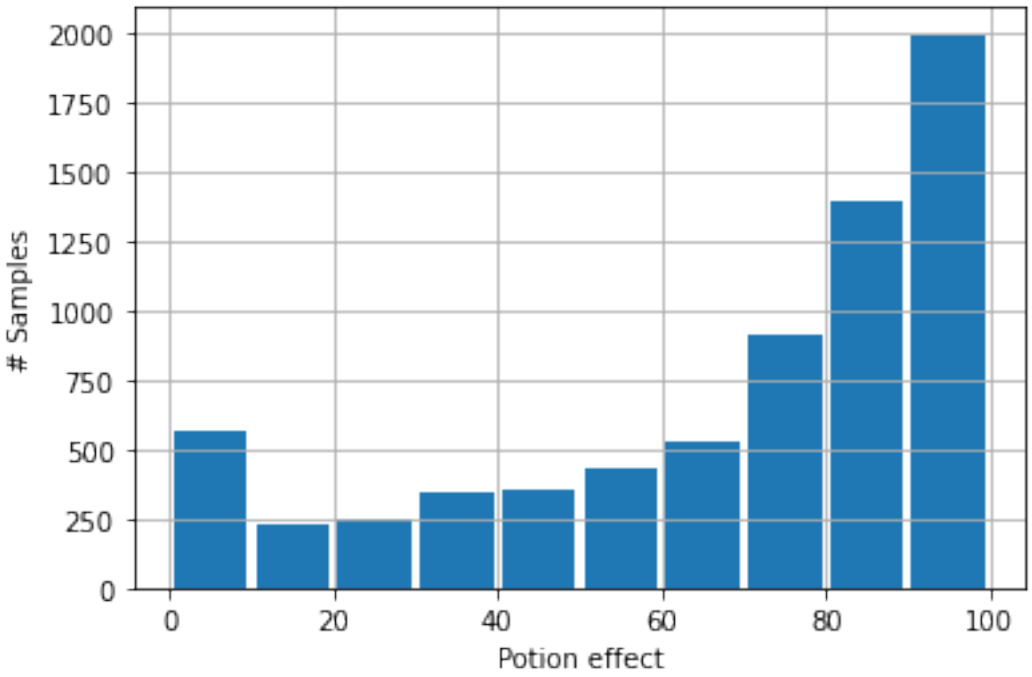

Figure 1: png

3.4.1 몬테카를로 클래스

Iterator로 동작하는 모듈을 만들 수 있습니다. **iter**, **next** 메소드를 구현하면 됩니다. 모듈을 만드는 ___init___에서 모의 실험과 분포의 이름을 받습니다. 이렇게 만들면 인자로 받은 순열 experiments와 다를 바 없이 iterator처럼 값을 끝없이 늘어놓으면서도, ___next___할 때마다 계산 과정에서 얻은 데이터 곧 횟수와 합을 저장하기 때문에 필요할 때 꺼내 쓸 수 있습니다.

```
class MonteCarlo:
    def __init__( self, experiments, name_of_experiment = "",  name_of_distribution = "" ):
        self.name_of_experiment = name_of_experiment
        self.experiments = experiments
        self.name_of_distribution = name_of_distribution
        self.number_of_trials = 0
        self.sum = 0

    def __iter__( self ):
        self.number_of_trials = 0
        self.sum = 0;
        return self

    def __next__(self):
        outcome = next( self.experiments )
        ...
        return outcome

    def mean( self ):
        if self.number_of_trials == 0:
            raise  BaseException("0 trials")
        else:
```

```
            return ...

    def report( self ):
        print( self.number_of_trials, self.name_of_experiment,
                "samples were taken by", self.name_of_distribution )
        print( "The mean was", self.mean() )
```

함수로 만든 monte_carlo와 쓰임새가 같다는 것을 보이기 위해서 MonteCarlo 클래스로 π 값을 어림잡아 봅니다.

```
dirichlet_test_experiments = MonteCarlo( repeat( dirichlet_test ), "dirichlet_test" )
take( 100_1000, dirichlet_test_experiments )
print( sqrt( 6 / dirichlet_test_experiments.mean() ) )
```

```
3.1411359422489764
```

3.4.2 변해도 되는 것과 변하지 않는 것

코드 구조가 조금도 바뀌지 않았습니다. MonteCarlo 클래스로 모듈을 만들어 필요할 때마다 iterator에 덧 씌우는 코드 말고는 달라진 부분이 거의 없습니다. 하지만 이 번에는 단계별 정보를 저장할 준비가 되어 있습니다.

```
herb_ratio = 0.2
herb_availablities = MonteCarlo( ... ,
                                "herb availabilties", "binomial distribution")

Excellent, Good, Marginal, Poor = 0, 1, 2, 3

def guess_quality( qualities ):
    return lambda isavailable: Excellent if isavailable else next( qualities )

herb_qualities = MonteCarlo(
    map( guess_quality( discrete_uniform_distribution( Excellent, Poor ) ),
        herb_availablities ),
    "herb quality", "discrete uniform distribution" )

effect_distributions_by_quality = [
    MonteCarlo( normal_distribution( mean, standard_deviation ),
                quality + " effect", "normal distribution")
    for ( quality, mean, standard_deviation ) in [
        ( "excellent", 90, 10 ),
        ( "good", 80, 20 ),
        ( "marginal", 50, 30 ),
        ( "poor", 30, 40 )
    ]]

def quess_effect_from( quality ):
    guess = ...
    if guess < 0: return 0
    elif guess > 100: return 100
    return guess

effects = map( quess_effect_from, herb_qualities )
```

```
number_of_trials = 7000
neffects = take( number_of_trials, effects )
```

3.4.3 모의 실험 보고서

모든 계산이 마무리되었으므로 계산 결과를 볼 수 있습니다. MonteCarlo 클래스가 없다면 계산 순열을 끝없이 늘어놓는 일과 계산 과정을 기록하는 일, 이 둘을 한 꾸러미로 묶어내기가 (언제나 그렇듯이 한 방법으로 다른 방법을 완전히 대체하는 할 수 있지만) 무척 번거롭습니다. 문제마다 알맞은 방법을 골라서 서로 잘 어울리도록 짜 맞추면 프로그램의 얼개가 아주 튼튼해집니다. 성능을 시험하고 고장난 곳을 찾아서 고치기에도 좋은 짜임새를 갖추게 됩니다.

좋은 약초를 얻을 확률은 얼마나 될까요?

```
print( "Herb availability: ")
herb_availablities.report()
print()
```

```
Herb availability:
7000 herb availabilties samples were taken by binomial distribution
The mean was 0.2012857142857143
```

좋은 약초가 있고 없고에 따른 약재의 품질은 어떤가요?

```
print( "Herb quality (Excellent = 0, Good = 1, Marginal = 2, Poor = 3): ")
herb_qualities.report()
print()
```

```
Herb quality (Excellent = 0, Good = 1, Marginal = 2, Poor = 3):
7000 potion quality samples were taken by discrete uniform distribution
The mean was 1.197857142857143
```

약재의 품질에 따른 약물의 효과는 어떻게 분포되나요?

```
print( "Potion effects by the 4 quality categories: ")
for quality in [ Excellent, Good, Marginal, Poor ]:
    ...
    print()
```

```
Potion effects by the 4 quality categories:
2814 excellent effect samples were taken by normal distribution
The mean was 89.90235117145805

1379 good effect samples were taken by normal distribution
The mean was 79.81417545995487

1415 marginal effect samples were taken by normal distribution
The mean was 49.968438883249995

1392 poor effect samples were taken by normal distribution
The mean was 29.315780545260804
```

마찬가지로 약재 약물 약효의 몬테카를로 시뮬레이션 결과를 히스토그램 막대그래프로 나타냅니다.

```
import matplotlib.pyplot as plot

plot.hist( neffects, rwidth=0.9 )
```

```
plot.xlabel("Potion effect")
plot.ylabel("# Samples")
plot.grid()
plot.show()
```