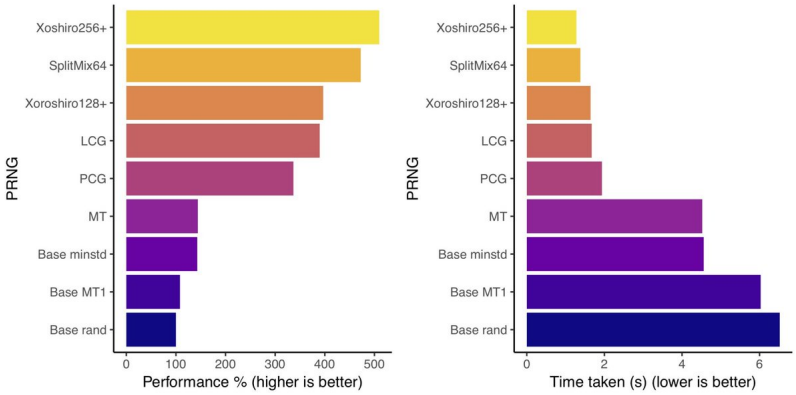
Mathis Hammel **@MathisHammel** *Tue Nov 16 13:29:48 +0000 2021*

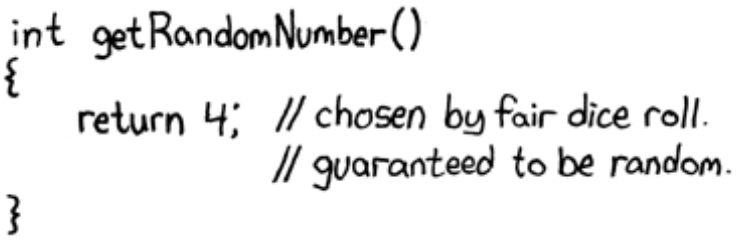
THREAD : J'ai trouvé un bug qui affecte toutes les versions récentes de Python, et il est interdit de le réparer !

Je vous explique pourquoi ■■ **https://t.co/WRPWkOjZQA**



Comme pour de très nombreux langages de programmation, le générateur de nombres aléatoires intégré à Python porte le nom de Mersenne Twister, ou MT19937.

C'est un RNG (Random Number Generator) très rapide et qui offre une entropie de très bonne qualité. **https://t.co/cOkcjRb4wq**



Mais le MT19937 n'est pas sécurisé contre les attaques cryptographiques : comme tous les générateurs de nombres aléatoires, il n'est en réalité que pseudo-aléatoire : après l'initialisation de ses variables internes, les bits en sortie sont produits de manière déterministe. **https://t.co/7Ha5CmtE5x**

```
>>> import random

>>> random.seed(1234)
>>> random.randint(1, 10000)
7221
>>> random.randint(1, 10000)
1915

>>> random.seed(1234)
>>> random.randint(1, 10000)
7221
>>> random.randint(1, 10000)
1915
```

Si un attaquant parvient à récupérer environ 20 kilobits consécutifs produits par une instance de Mersenne Twister, il peut reconstituer l'état de la mémoire interne du générateur, et peut ainsi prédire tous les nombres suivants qu'il produira.

Cette vulnérabilité est bien connue, et c'est pour ça qu'il faut absolument utiliser un RNG dit "cryptographique" si l'on souhaite avoir des nombres aléatoires vraiment imprévisibles (je suis d'ailleurs en train de préparer un autre thread au sujet des casinos en ligne! #teasing) **https://t.co/rLv8lPWHiw**

En 2019 alors que j'étais en pleins préparatifs pour le CTF INS'hAck (Capture The Flag = compétition de cybersécurité), je voulais créér un challenge de crypto sur lequel les participants devraient mettre en oeuvre une attaque contre le générateur MT19937 de Python.

En général, les challenges qui demandent une attaque sur Mersenne Twister on les voit venir de loin : comme il faut récupérer 624 sorties consécutives de 32 bits pour pouvoir réaliser l'attaque, on voit très souvent "random.getrandbits(32)" dans le code source à exploiter. **https://t.co/dgeM1PhDtk**

```python
# Challenge du noxCTF 2018

import random

class ThreadedServer(object):
    def getKey(self, r):
        return str(r.getrandbits(32)).rjust(16, '0')

    def listenToClient(self, client, address):
        client_flag = self.flag
        r = random.Random()
        key = self.getKey(r)
        client_flag = self.encrypt(key, client_flag)
        while True:
            try:
                client.send('Please insert the decryption key:\n')
                key_guess = client.recv(16)
                if key_guess == key:
                    client.send('Correct! Your flag is: ' + self.decrypt(key, client_flag) + '\n')
                else:
                    client.send('Wrong! The key was: ' + key + '\n')
                    client_flag = self.decrypt(key, client_flag)
                    key = self.getKey(r)
                    client_flag = self.encrypt(key, client_flag)
```

Moi, j'avais envie de le déguiser un peu mieux que ça, pour que la porte dérobée soit moins évidente : il existe plein d'autres fonctions qui génèrent des bits aléatoires et qui sont bien plus souvent utilisées dans le monde réel : randint, shuffle, randrange, sample, etc.

Par exemple, saviez-vous que pour mélanger de manière parfaitement uniforme une liste de longueur N, il suffit de O(N) opérations ? On commence par choisir un index entre 0 et N-1, avec lequel on échange l'élément à l'indice N-1. Pareil pour N-2, et ainsi de suite jusqu'à 1. **https://t.co/fhyWcqhfH4**

```
def shuffle(self, x):
      """Shuffle list x in place, and return None."""

      randbelow = self._randbelow
      for i in reversed(range(1, len(x))):
          # pick an element in x[:i+1] with which to exchange x[i]
          j = randbelow(i + 1)
          x[i], x[j] = x[j], x[i]
```

C'est ce genre d'opération que je voulais utiliser dans mon code source vulnérable, les joueurs devraient alors étudier le fonctionnement interne de l'interpréteur Python pour comprendre comment retrouver les bits d'entropie au lieu de leur donner gratuitement avec getrandbits.

Un très grand nombre de fonctions du module random de Python font appel à la fonction cachée _randbelow, qui gère les appels au générateur de bits aléatoires du Mersenne Twister. Sa seule utilité est, comme son nom l'indique, de renvoyer un nombre entier compris entre 0 et n-1. **https://t.co/8uHPfmRnvN**

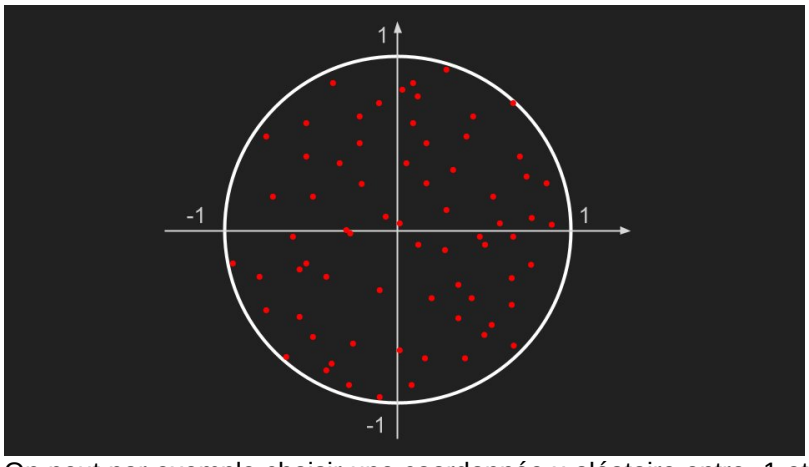
```
def randrange(self, start, stop=None, step=_ONE):
    # This code is a bit messy to make it fast for the
    # common case while still doing adequate error checking.
    istart = _index(start)
    if stop is None:
        # We don't check for "step != 1" because it hasn't been
        # type checked and converted to an integer yet.
        if step is not _ONE:
            raise TypeError('Missing a non-None stop argument')
        if istart > 0:
            return self._randbelow(istart)
        raise ValueError("empty range for randrange()")
```

```
def choice(self, seq):
    """Choose a random element from a non-empty sequence."""
    # raises IndexError if seq is empty
    return seq[self._randbelow(len(seq))]
```

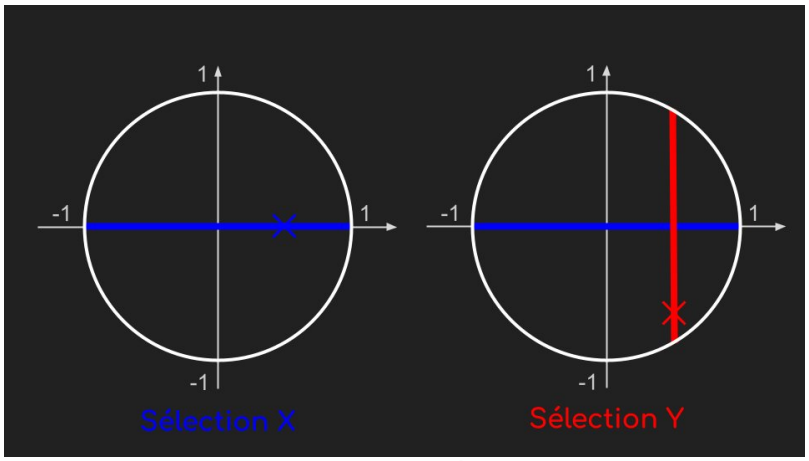
```python
def sample(self, population, k, *, counts=None):
    randbelow = self._randbelow
    if not 0 <= k <= n:
        raise ValueError("Sample larger than population or is negative")
    result = [None] * k
    setsize = 21        # size of a small set minus size of an empty list
    if k > 5:
        setsize += 4 ** _ceil(_log(k * 3, 4))  # table size for big sets
    if n <= setsize:
        # An n-length list is smaller than a k-length set.
        # Invariant:  non-selected at pool[0 : n-i]
        pool = list(population)
        for i in range(k):
            j = randbelow(n - i)
            result[i] = pool[j]
            pool[j] = pool[n - i - 1]  # move non-selected item into vacancy
    return result
```

Du coup, à partir d'un résultat donné d'une fonction shuffle/randint/sample on devrait pouvoir retrouver le résultat des différents appels à randbelow, et à partir de ceux-ci tous les bits qui ont été produits par getrandbits. C'est simple non ? Ben en pratique pas tant que ça ■

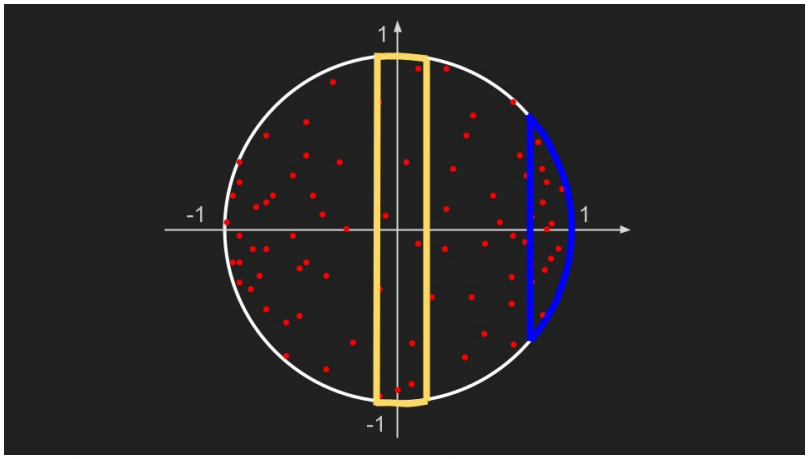
Pour comprendre comment fonctionne randbelow, il faut d'abord que je vous parle de la méthode de sélection/rejet qui est utilisée pour simuler des tirages sur n'importe quelle loi de probabilités. Admettons par exemple que vous souhaitiez choisir un point au hasard sur un disque: **https://t.co/j3FrCjdm2o**



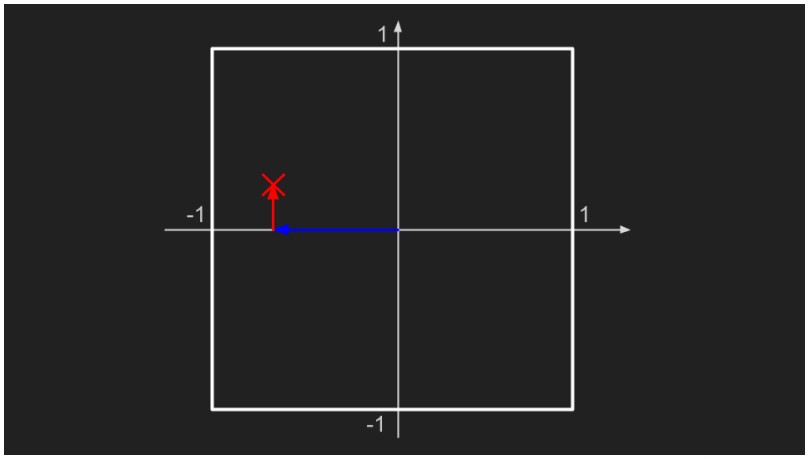
On peut par exemple choisir une coordonnée x aléatoire entre -1 et 1 puis une coordonnée y aléatoire entre -sqrt(1 - x²) et sqrt(1 - x²) qui nous garantit que le point tiré sera forcément dans le cercle. **https://t.co/a8BJIxi5jJ**



Cependant, on va rencontrer un problème : on a ici autant de chances de tomber dans la zone bleue que dans la zone jaune, alors que celles-ci ont des aires différentes : la densité du tirage n'est pas uniforme ! **https://t.co/0bcxt5hTWw**

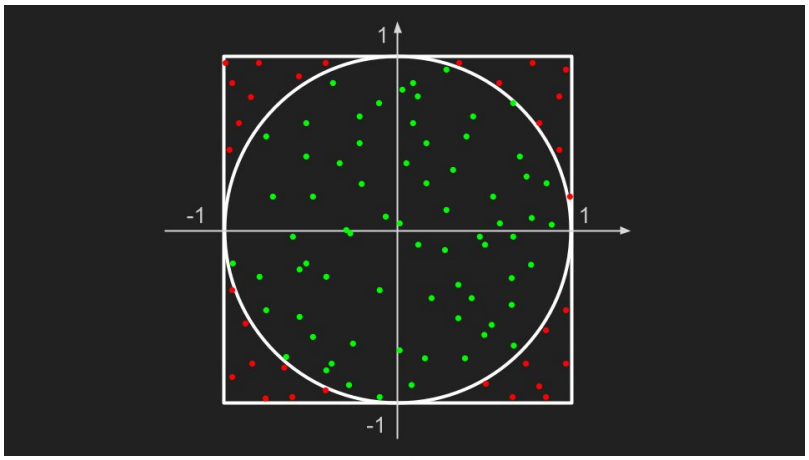


La méthode par rejet permet quant à elle de faire des tirages aléatoires compliqués en se basant un générateur de nombres aléatoires simple. On peut par exemple partir du carré, sur lequel il est facile de générer des points uniformes en tirant indépendamment X et Y dans [-1, 1]. **https://t.co/3cEgEoovAy**



On commence donc par tirer un point au hasard uniformément dans le carré, puis on regarde si ce point est dans le disque.

Si c'est le cas, on a notre point aléatoire sur le disque, et sinon on recommence. Normalement, il suffit de quelques itérations pour trouver un point valide. **https://t.co/IezTvAmSjI**



Pour le cercle il existe d'autres méthodes n'utilisant pas de rejet (avec des coordonnées polaires selon des distributions bien choisies notamment), mais le tirage sélection/rejet reste très souvent utilisé car très rapide (en moyenne) et peu susceptible aux erreurs de précision.

Le tirage de nombres entiers entre 0 et N-1 c'est un peu la même chose : on pourrait par exemple générer un nombre de 1000 bits et prendre son modulo N, mais ce n'est pas parfaitement équiprobable si le nombre n'est pas un multiple de N.

De même, on peut se dire que la fonction random (qui génère un nombre réel entre 0 et 1) peut nous aider : la partie entière de random()*N est théoriquement uniforme sur les entiers de 0 à N-1. Mais comme la représentation des flottants n'est pas continue en pratique, même souci.

Pour assurer un tirage le plus uniforme possible, Python va ainsi utiliser la méthode par rejet : à l'aide de la fonction getrandbits, il génère un entier codé sur autant de bits que notre nombre N. Tant que le nombre tiré n'est pas dans [0,

N-1], on en génère un nouveau. **https://t.co/TlGKyOw65b**

```python
def _randbelow_with_getrandbits(self, n):
    "Return a random int in the range [0,n).  Returns 0 if n==0."

    if not n:
        return 0
    getrandbits = self.getrandbits
    k = n.bit_length()  # don't use (n-1) here because n can be 1
    r = getrandbits(k)  # 0 <= r < 2**k
    while r >= n:
        r = getrandbits(k)
    return r
```

Tirage par sélection/rejet

Les tirages par rejet peuvent entraîner un "gâchis" d'entropie : si le nombre tiré ne respecte pas le critère de sélection, les bits utilisés pour sa génération sont perdus. Pour mon challenge de CTF, il fallait du zéro déchet pour que l'attaquant puisse récupérer tous les bits.

Il existe un seul cas dans lequel _randbelow serait garanti de ne faire qu'un seul tirage : si N est une puissance de 2, alors n'importe quel nombre tiré avec getrandbits() est dans [0, N-1] et il n'y a pas de rejet. Sauf que j'ai trouvé un bug dans l'implémentation de Python !

Pour calculer le nombre de bits à tirer aléatoirement, on va appeler la fonction n.bit_length(). Dans le cas où N est une puissance de deux, ce nombre de bits est incorrect : en prenant n=1024 (2^10), on ne devrait à avoir à tirer que 10 bits. Cependant, n.bit_length() vaut 11 ! **https://t.co/U6YlHoZER1**

```python
def _randbelow_with_getrandbits(self, n):
    "Return a random int in the range [0,n).  Returns 0 if n==0."

    if not n:
        return 0
    getrandbits = self.getrandbits
    k = n.bit_length()  # don't use (n-1) here because n can be 1
    r = getrandbits(k)  # 0 <= r < 2**k
    while r >= n:
        r = getrandbits(k)
    return r
```

Par conséquent, au lieu d'avoir un tirage parfait qui consomme exactement le bon nombre de bits et réussit du premier coup, on se retrouve avec des tirages qui n'ont que 50% de chances de réussir, et qui utilisent en moyenne 2x plus de temps et d'entropie que nécessaire. Du coup,

Je me suis mis en tête de proposer un patch du code source de Python. Comme pour tout projet titanesque, la procédure est assez lourde : il faut créer une issue détaillée sur le tracker externe, signer un accord de propriété intellectuelle, compiler et tester le fork, ... **https://t.co/1fItmLZrl3**

classification

Titre: _randbelow_with_getrandbits function inefficient with powers of two	
Type: performance	Stage: resolved
Components: Library (Lib)	Versions: Python 3.8

process

État: closed	Resolution: rejected
Dependencies:	Supplanté par:
Affecté à: rhettinger	Liste des curieux: Mathis Hammel, mark.dickinson, rhettinger, tim.peters
Priorité: low	Mots-clé: patch

Created on **2019-05-21 20:50** by **Mathis Hammel**, last changed **2019-05-23 15:47** by **mark.dickinson**. This issue is now **closed**.

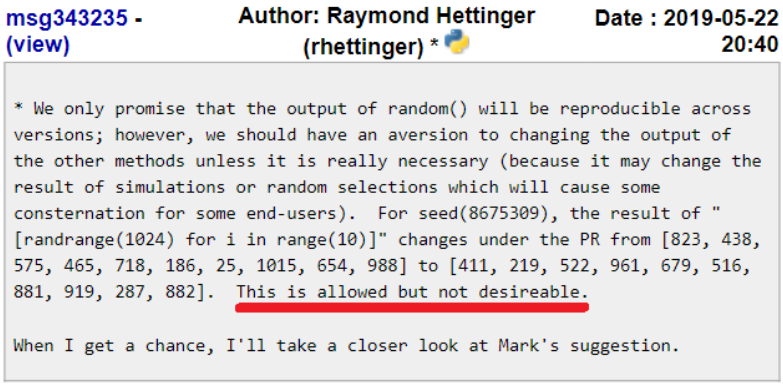
Pull Requests

URL	Status	Linked
PR 13485	closed	Mathis Hammel, 2019-05-22 06:20

Messages (6)

Avec une modification de seulement 4 lignes de code, mon fork fonctionne bien et on observe une accélération effective de 2x sur les appels à _randbelow sur les puissances de 2. Victoire !
Mais petit souci : y a des dizaines de tests unitaires qui ne passent pas ! **https://t.co/7MH2haA3mq**

En effet, le module random de Python doit respecter le reproductibilité : pour une même seed, le RNG devrait produire exactement les mêmes résultats entre toutes les versions de Python. Mais ce n'est pas le cas ici parce qu'on a réparé les fuites d'entropie ! **https://t.co/SHdvzxlRiJ**


msg343235 - (view) Author: Raymond Hettinger (rhettinger) * Date : 2019-05-22 20:40

```
* We only promise that the output of random() will be reproducible across
versions; however, we should have an aversion to changing the output of
the other methods unless it is really necessary (because it may change the
result of simulations or random selections which will cause some
consternation for some end-users).  For seed(8675309), the result of "
[randrange(1024) for i in range(10)]" changes under the PR from [823, 438,
575, 465, 718, 186, 25, 1015, 654, 988] to [411, 219, 522, 961, 679, 516,
881, 919, 287, 882].  This is allowed but not desireable.

When I get a chance, I'll take a closer look at Mark's suggestion.
```

Pour conclure, ce bug assez spécifique ne permet pas de justifier un tel changement, et le bug devra donc rester éternellement dans Python. Mais ça n'a pas été peine perdue, le bug que j'ai ouvert a engendré un refactoring et un nouveau fonctionnement interne à getrandbits :)

Voilà, j'espère que ce thread vous a plu et que vous saurez briller en société avec vos nouvelles connaissances sur les tirages par sélection/rejet et sur les internals de Python ;)

Si mes threads techniques vous intéressent, je vous invite également à jeter un oeil à **@Guardia_School**: c'est une école que je parraine et qui propose un cursus post-bac en 3/5 ans pour apprendre les métiers de la cybersécurité, ouverture en octobre 2022 ! **https://t.co/kSipHyOKfU**

@Guardia_School Et comme d'habitude, si vous aimez ce que je fais, la chose la plus sympa que vous puissiez faire pour donner de la visibilité à mes contenus c'est de les partager : que ce soit via un retweet ou un message sur le Slack de votre boîte, pensez-y ;)