Concatenated class

Create a concatenated 1-bit minwise hasher of length *m* using an underlying *q*-bit hash function. The *seed* argument seeds the RNG so that multiple instances can be created with identical behaviour.

```
>>> import pseudo
>>> h = pseudo.Concatenated(1024, q=64, seed=6)
>>>
```

The returned object can be used to hash token sets via its *hash* method. We can use padded bigrams for tokenization.

```
>>> import tokenization
>>> tokens1 = tokenization.n_grams('thompson', 2, pad=True)
>>> tokens2 = tokenization.n_grams('timpson', 2, pad=True)
>>> tokens1
['_t', 'th', 'ho', 'om', 'mp', 'ps', 'so', 'on', 'n_']
>>> tokens2
['_t', 'ti', 'im', 'mp', 'ps', 'so', 'on', 'n_']
>>>
```

We can generate the true Jaccard score for these tokens sets.

```
>>> pseudo.Jaccard(tokens1, tokens2)
0.5454545454545454
>>>
```

We can securely tokenize the token sets using the *Concatenated* instance created earlier. In fact we can create a second instance using the same seed to simulate pseudonymization in distinct locations.

```
>>> hash1 = h.hash(tokens1)
>>> hash2 = pseudo.Concatenated(1024, q=64, seed=6).hash(tokens2)
```

Hashes print as long ints.

```
>>> hash1
171895235200038946890402970428569944779870460885573762488446273820393565865841485343645616161728342261092681201688570650744187204337759207659090485665310779799536599898998509149858262826292348966905322667620008405806604077273651383154347751639002796921479955285490743570934704296590419567354655328119900712436L
>>>
```

In fact, these *C_hash* instances are derived from Python's long integer type and are associated with two extra attributes - the number of significant bits and the XOR compression factor (which are both required for Jaccard score estimation).

Given two hashes we can estimated the Jaccard score.

```
>>> pseudo.J_hat_from_conc(hash1, hash2, 1024)
0.572265625
>>>
```

The above exploits the fact that concatenated 1-bit hashes are subclasses of *long*. The above call will also work with any Python *int* or *long* instances. It is the user's responsibility to ensure that hashes are comparable (same length, generated by the same 1-bit hashes etc.).

As we know the true score we can calculate the variance of estimation.

```
>>> pseudo.var_J_hat(0.5454545454545454, 1024, N=1)
0.0006860149793388431
>>>
```

C_hash class

This class contains a couple of methods for compressing hashes, and a property that returns the bitstring representation of the hash.

The *compressed* method returns a new hash with fewer significant bits. The returned value is simply the value we would have had if we had used a smaller number of minwise hash functions.

```
>>> comp1 = hash1.compressed(256)
>>> comp2 = hash2.compressed(256)
>>> comp1.m, comp1.N
(256, 1)
>>> pseudo.J_hat_from_conc(comp1, comp2, 256, N=1)
0.65625
>>> pseudo.var_J_hat(0.5454545454545454, 256, N=1)
0.0027440599173553723
>>>
```

The *XOR* method returns XOR compressed hashes with compression factor N.

```
>>> comp1 = hash1.XOR(N=4)
>>> comp2 = hash2.XOR(N=4)
>>> comp1.m, comp1.N
(256, 4)
>>> pseudo.J_hat_from_conc(comp1, comp2, 256, N=4)
```

```
0.5149417859767794
>>> pseudo.var_J_hat(0.5454545454545454, 256, N=4)
0.009197553658364058
>>>
```

The variance is greater than for the 'bitstring slice' form of compression for this Jaccard score, but would be lower for sufficiently high Jaccard scores.

We can print the binary representation of a hash.

```
>>> comp1.digits
'01101111101101111110011000101001001110001010100101011111011001000001000001 1
10001100010110011010000110010001110001001110111110011000010110111110110010 1
00111101001101111111101011101100100101011000101110110100100100101001111101 1
01100000111001000011000001000'
>>>
```

This shows that compression still produces what is (for all practical purposes) a random integer.

BloomFilter class

We need to generate a suitable list of hash functions, supplying the number of hash functions, the bit length of the Bloom filter and an optional seed.

```
>>> funcs = set_like.k_hashes(4, 1024, seed=6)
>>>
```

For the Bloom filter we need to supply the length and the hash functions.

```
>>> bf1 = set_like.BloomFilter(1024, funcs)
>>>
```

We can generate a second Bloom filter with distinct, but identically behaving, hash functions. This simulates the generation of comparable Bloom filters in distinct locations.

```
>>> bf2 = set_like.BloomFilter(1024, set_like.k_hashes(4, 1024, seed=6))
>>>
```

We can add our tokens (or we could have supplied them as an optional keyword argument 'items' when creating the Bloom filters).

```
>>> for token in tokens1:
```

```
      bf1.add(token)


>>> for token in tokens2:
      bf2.add(token)
```

We can use the standard Jaccard score estimator or the bias-corrected estimator due to Swamidass and Baldi (2007)[1]. Note the extra argument required for the corrected estimator.

```
>>> pseudo.J_hat_from_bf(bf1.bits, bf2.bits)
0.5348837209302325
>>> pseudo.J_hat_from_bf_corrected(bf1.bits, bf2.bits, 1024)
0.5272637368168046
>>>
```

Of course, we can simply use the Bloom filter as a probabilistic set data structure.

```
>>> 'th' in bf1
True
>>>
```

Note: we would generally use a larger number of hash functions for this purpose (to minimize the false positive rate). For a 1024 bit Bloom filter and 9 items we might use 79 hash functions.

```
>>> set_like.opt_k(1024, 9)
78.86474587704267
```

Both concatenated hashes and Bloom filters can be (partially) serialized using their *hex* or *digits* methods. Only the *digits* method automatically pads the returned string to convey the number of significant bits. The builtins *str* and *repr* can also be used.

Requirements

numpy

1Swamidass, S. J. and Baldi, P. (2007), *Mathematical correction for fingerprint similarity measures to improve chemical retrieval*, Journal of chemical information and modeling (ACS Publications) **47** (3): pp.952-964