Arithmetical functions	
Function signature	**Description**
even(n) odd(n) remainder(n,m)	For any integer n, those expressions do what is expected of them: the value of even(n) is True if n is even and False if n is odd. The expression odd(n) works the other way around. Since these functions can be (and are) defined in terms of the remainder expression n % 2, the justification for including them is that they make code more readable. The same is true of remainder(n,m), which is defined as n % m
dlen(n) blen(n)	These expresions deliver the number of decimal and binary digits of the positive integer n.
convert(n,b) hohner(D,b)	If n is an integer ≥ 2, convert(n,b) returns the list of digits of the positive integer n relative to the base b. To get the integer n whose list of digits in the base b is D, we have hohner(D,b) (the name points to the well known generating rule used to find n).
factorial(n)	Computes n!
fib(N) fibonacci = fib	Delivers the Nth Fibonacci number.
gcd(*N) lcm(*N)	The expression gcd(*N) returns the greatest common divisor of a list of elements. The expression lcm(*N) returns the least common multipler of a list of elements.
igcd(m,n) ilcm(m,n)	The expression igcd(m,n) yields the greatest common divisor of the positive integers m and n. Similarly, their least common multiple is supplied by the expression ilcm(m,n). Both functions can be applied to a sequence of any number of integers. The names gcd and lcm can also be used, but they are resolved, when the parameters are integers, by calling igcd and ilcm.
extended_euclidean_algorithm(m, n) bezout	Given integers m and n, this function returns a triple (x,y,d) of integers such that d = igcd(m,n) and d = x m +y n. It also works for polynomials. Bezout is an alias of extended_euclidean_algorithm.
test_bezout(r) test_bezout()	It computes two random integers of r digits a and b, and then it executes the bezout(a,b). The function returns 'Success' if d==a*x+b*y. Otherwise, it returns 'Failure'. By default r = 10.
test_gcd(m,n) test_gcd()	It computes a random integer of m digits a and a random intiger of n digits b. Then, it computes d = gcd(a,b). The function returns 'Success' if gcd(a//d,b//d) == 1. Otherwise it returns 'Failure'.
rabin_miller(n, k=25) strong_probable_prime(n,a) strong_pseudo_prime(n,a)	This function implements the Rabin-Miller primality test on the integer n. If it returns False, then n is composite. Otherwise it is prime with very high probability (not less than p=1-1/4^k, which for the default iteration number k = 25 is p = 0.9999999999999991). The Rabin-Miller test uses the function strong_probable_prime(n,a) that tells wheter an odd integer n is a strong probable prime to the base a (cf. Crandall-Pomerance-2005, Algorithm 3.5.2), and strong_pseudo_prime(n,a) tests whether n is a strong probable prime base a but not prime.
is_prime(n) is_prime_power(n) is_perfect_power(n) is_square(n)	For a positive integer n, is_prime(n) is true if n is prime and false otherwise. By default, the test is based on the rabin_miller method. We can call is_prime(n,method='BPSW') or is_prime(n,method='miller'), which are equivalent to BPSW(n) and miller(n), respectively (these functions are described later in this section). Similarly, is_prime_power(n) is true if and only if n is a power of a prime number. Finally, for an odd integer n > 1, the expression is_perfect_power(n) produces a pair of integers (x,p) with the following properties: If n is not a perfect power, this pair is equal to (n,1), and otherwise p is prime and n = x^p. Although this function is very fast, its complete analysis is a bit involved and we refer to the note X180630 for details and references. An interesting exercise is to use it to produce a faster version of is_prime_power(n). Finally, the function is_square(n) decides whether the positive integer n is a perfect square.
primes_less_than(n)	Delivers the list of prime numbers that are less than the integer n.

Arithmetical functions	
Function signature	**Description**
next_q (n) next_prime(n) next_p = next_prime	For any positive integer n, the first expression gives the first prime power ≥ n. Similarly, the second expression gives the first prime ≥ n. Since there are more prime powers than primes, we have next_q(x) ≤ next_p(x).
pollard(n)	If n is composite, this function attemps to find a non-trivial factor of n. It is the basis for the factoring function ifactor(n).
ifactor(n) prime_factors(n)	The function ifactor(n) computes the prime factorization p1^{e1}·p2^{e2}··· of a positive integer n in the form of a table: {p1:e1, p2:e2, …}. Note. Whereas in the table delivered by ifactor(n) the prime divisors of n do not appear in increasing order, they do so in the list produced by prime_factors(n)
divisors(n) tau(n) sigma(n)	For a positive integer n, the first expression supplies the list of positive divisors of n in increasing order. The number of such divisors is given by tau(n) and their sum by sigma(n).
phi_euler(n) lambda_carmichael(n)	Computes Euler's totient function of a positive integer n, which is the number of integers in 1,2, ..., n-1 that are coprime to n. For the Carmichael λ-function of n, see Yan-2002, Definition 1.4.7.
mu_moebius(n)	It returns 0 if n has a repeated prime factor, and otherwise 1 or -1 according to whether the number of prime factors is even or odd.
is_square_free_n(x)	For a positive integer x, it returns True if it is not divisible by any perfect square greater than 1. Equivalenty, if x is not divisible by the square of any prime, or also mu_moebius(x) ≠ 0. There is a similar function is_square_free(f) that is specific for univariate polynomials f. The separation of the two makes the computation of either one much more efficient.

Arithmetical functions

Function signature	Description

lucas_number(P,Q,x0,x1,N)
lucas_chain_V(P,Q,m,n)
lucas_U(P,Q,N)
lucas_V(P,Q,N)
lucas(N)
pell(N)
pell_lucas(N)
jacobsthal(N)
jacobsthal_lucas(N)
mersenne(N)
	The main function of this group is lucas_number(P,Q,x0,x1,N), where the parameters are integers, N non-negative. It returns the N-th Lucas number of the Lucas sequence associated to P, Q, x0, x1. In PyM it is defined as follows:
	
	def lucas_number(P,Q,x0,x1,N):
	    if N==0: return x0
	    if N==1: return x1
	    for _ in range(2,N+1):
	        x0, x1 = x1, P*x1-Q*x0
	    return x1
	
	The function lucas_chain_V(P,Q,m,n) is a variation of the function just described which is used in the BPSW test of primatlity and is implemented as follows (cf. Crandall-Pomerance-2005, Algorithm 3.6.7):
	
	def lucas_chain_V(P,Q,m,n):
	    mb = bin(m)[2:] # the binary string representing m
	    v0 = 2; v1 = P
	    j = 0
	    for b in mb:
	        Qj = power(Q,j,n)
	        if b == '1':
	            v0,v1 = (v0*v1)%n-(Qj*P)%n, (v1**2-2*Qj*Q)%n
	        else:
	            v0,v1 = (v0**2-2*Qj)%n,(v0*v1)%n-(Qj*P)%n
	        j = 2*j+int(b)
	    return (v0,v1)
	
	The other numbers are defined as follows:
	
	    lucas_U(P,Q,N) = lucas_number(P,Q,0,1,N)
	    pell(N) = lucas_U(2,-1,N)
	    jacobsthal(N) = lucas_U(1,-2,N)
	
	    lucas_V(P,Q,N) = lucas_number(P,Q,2,P,N)
	    lucas(N) = lucas_V(1,-1,N)
	    pell_lucas(N) = lucas_V(2,-1,N)
	    jacobsthal_lucas(N) = lucas_V(1,-2,N)
	
	The N-th Mersenne number is 2^N-1 coincides with lucas_U(3,2,N) and can be obtained with mersenne(N).

baillie-pomerance-selfridge-wagstaff(n) BPSW = baillie-pomerance-selfridge-wagstaff	This implements the primality test of Baillie, Pomerance, Selfridge, and Wagstaff, as explained in Crandal-Pomerance-2005, Algorithm 3.6.9.

Arithmetical functions	
Function signature	**Description**
miller(n)	For an odd number n > 1, this function implements Miller's primality test (Crandall-Pomerance-2005, Algoritm 3.5.13). If it returns False, then n is composite. Otherwise, n is prime or the GRH is false.
lucas_lehmer(n) LL = lucas_lehmer	An implementation of the Lucas-Lehmer test (Crandall-Pomerance-2005, Algorithm 4.2.98697) to decide whether the n-th Mersenne number 2^n-1 is prime or not.
dec2bin(x, nb = 58) bin2dec(xb)	Let x be a real number, x ϵ [0,1]. The function dec2bin(x, nb) returns the binary expansion of x up to nb bits. The default value of nb is 58, which encompasses the machine precesion of Python floats. Conversely, bin2dec(xb) converts a string of bits into a real number in the interval [0,1] by interpreting that the kth bit b contributes with b/2^k.
floor(x) qfloor(x) frac(x)	If x is a real number, floor(x) or qfloor(x) is the greatest integer ≤ x. In mathematics, it is often denoted ⌊x⌋.
ceiling(x) qceiling(n,m) ceil = ceiling	If x is a real number, ceiling(x) is the least integer ≥ x. Since it relies on the Pyhton double precesion floats, it is not indicated when higher precesion is needed. An alternative is qceiling(n,m), n and m integers, m ≠ 0, which returns the (exact) ceiling of the rational number n/m.
continuous_fraction(x,n)	If x is a positive real number, this funtion returns the n-term continuous fraction of x in the form of a list [f0, f1, ..., fn-1], so that setting x0 = x we have f0 = floor(x0) and then, for j = 1, ..., n-1, fj = floor(xj) with xj = 1/(xj-1 - fj-1). Si xj-1 - fj-1 = 0, la iteració s'atura i la funció retorna [f0, f1, ..., fj-1]. Example: continuous_fraction(π,4) ⇒ [3,7,15,1].
continuous_fraction_value(F)	If F is a list of positive integers, this function computes the rational number corresponding to F regarded as a continuous fraction. Example: continuous_fraction_value([3,7,15,1]) ⇒ 355/113 :: Q. Note that 355/113 ⇒ 3.1415929 ..., whereas π = 3.1415926 ....
power_check(n,x,k)	fast check whether n==x**k
is_power(n,k)	Checks whether an odd integer n is a k-th power. Ex.: is_power(125.3) ⇒ 5
quo_rem(r0,r1)	It returns the quocient and residus of the division r0/r1
bit_product(a,b,r) bit_product(a,b) bdot = bit_product	Computes the scalar product of the first r-digits of the binary representation of a and b. By default r is the mininum between the binary digits of a and b.
cycle_factors(J)	Given a list of integers, it returns the cycles in the list.
Modular arithmetic functions	
order(k, n)	If gcd(k,n) = 1, the order of k in Zn*. Otherwise 'Error'
inverse(k,n)	If gcd(k,n) = 1, inverse(k,n) computes a positive integer k' such that k' < n and k'k ≡ 1 mod n. Otherwise, "Error"

Arithmetical functions	
Function signature	**Description**
mult(n,k,b) bpow(n,k,b) quot(n,k,b) power(n,k,m)	The value of these expressions is n · k mod 2^b, n^k mod 2^b and (m / k) mod 2^b, respectively. In the latter case, k has to be odd. The function bpow(n,k,b) coincides with power(n,k,2^b), as power(n,k,m) computes n^k mod m. These functions are used, for example, in the definition of the next two.
jacobi(a,n) legendre(a,n)	Computes the Jacobi symbol (a/n) of two integers a and n, which must be odd and positive. The PyM implementation is based on Algorithm 2.3.5 of Crandall-Pomerance-2005. If n is prime, it coincides with the Legendre symbol (a/n), which is 0 if a is divisible by n and otherwise it is +1 or -1 according to whether a is or is not a quadratic residue mod n. It coincides with legendre(a,Z_n)
nroot(n,k,b) nsqroot(n,b) sqroot(a,F)	If n is an odd integer and k is either odd or 2, nroot(n,k,b) computes an integer r < 2^b such that r^kn ≡ 1 mod 2^b, which is a kth root of n^{-1} mod 2^b. The function nsqroot(n,b) is equivalent to nroot(n,2,b). These funtions are used in the definition of is_perfect_power (n). sqroot(a,F) returns an element b in the domain F, such that b*b = a in the domain F.
cyclotomic_class (k, n, q) cyclotomic_class (k, n)	Assuming that q and n are positive integers and that gcd(q,n)=1, the call cyclotomic_class(k,n,q) supplies the q-cyclotomic class of k mod n, which by definition is the list [k, q·k,q^2·k,...,q^{r−1·k}], where the operations are done mod n and r is the least positive integer such that q^r·k=1.
cycloctomic_classes (n, q) cycloctomic_classes (n)	Assuming that q and n are positive integers and that gcd(q,n)=1, the function cycloctomic_classes(n,q) furnishes the list of all the q-cyclotomic classes mod n. Finally, cycloctomic_classes(n) is defined as cycloctomic_classes(n,2).
product(X)	Given a list X, it computes the product of all the elements in X.