### Evaluation rule for call expressions:
1. Evaluate the operator and operand subexpressions.
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

### Applying user-defined functions:
1. Create a new local frame with the same parent as the function that was applied.
2. Bind the arguments to the function’s formal parameter names in that frame.
3. Execute the body of the function in the environment beginning at that frame.

### Execution rule for def statements:
1. Create a new function value with the specified name, formal parameters, and function body.
2. Its parent is the first frame of the current environment.
3. Bind the name of the function to the function value in the first frame of the current environment.

### Execution rule for assignment statements:
1. Each clause is considered in order.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

### Execution rule for or expressions:
1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

### Execution rule for and expressions:
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

### Execution rule for not expressions:
1. Evaluate exp; The value is True if the result is a false value, and False otherwise.

### Execution rule for while statements:
1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.
3. If it is a false value, execute the entire loop, then return to step 1.

### Non-Pure Functions
- abs(number):
  - 2
- pow(x, y):
  - 1024

### Pure Functions
- print(x):
  - None

### Higher-order function: A function that takes a function as an argument.
- Function of a single argument (not called term)
- Function of a single argument (called term)
- Function of multiple arguments

### Nested-def statements: Functions defined within other function bodies are bound to names in the local frame.
square = lambda x,y: x * y

A function

with formal parameters x and y

that returns the value of “x * y”

Must be a single expression

def make_adder(k):
    """Return a function that takes one argument k and returns k + n."
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    """Return a function h that composes f and g."
    def h(x):
        return f(g(x))
    return h

def count_partitions(n, m):
    """Finds the number of partitions of n into at most m parts.

    >>> count_partitions(10, 5)
    42
    >>> count_partitions(10, 2)
    1
    >>> count_partitions(10, 1)
    1
    >>> count_partitions(10, 0)
    0
    >>> count_partitions(10, 10)
    1
    >>> count_partitions(10, 11)
    0
    """
    if n < m:
        return 1
    if n < 0:
        return 0
    return count_partitions(n, m-1) + count_partitions(n-m, m)

from operator import floordiv, mod

def divide_exact(n, d):
    """Return the quotient and remainder of dividing N by D.
    >>> (q, r) = divide_exact(2012, 10)
    >>> q
    201
    >>> r
    2
    """
    return (floordiv(n, d), mod(n, d))