Execution rule for while statements:

1. Evaluate the header expression.
2. If it is a true value, execute the whole suite, then return to step 1.

Evaluation rule for not expressions:

1. If it is a true value, evaluate the operand subexpression.
2. If it is a false value v, evaluate the operand subexpression.
3. If the result is a true value v, evaluate the operand subexpression.
4. Otherwise, the expression evaluates to the value of the operand subexpression.

Evaluation rule for and expressions:

1. Evaluate the left operand subexpression.
2. If the result is a true value v, evaluate the right operand subexpression.
3. Otherwise, the expression evaluates to the value of the left operand subexpression.

Evaluation rule for assignment statements:

1. From the right operand subexpression, evaluate the right-hand-side expression.
2. Simultaneously bind the names on the left to those values.
3. Evaluate the expression(s) on the right of the equal sign.

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. Simultaneously bind the names on the left to those values.

Evaluation rule for call expressions:

1. Evaluate the operand subexpression.
2. Apply the function that is the value of the operand subexpression to the arguments that are the values of the operand subexpressions.
3. Evaluate the expression(s) on the right of the equal sign.

Calling/Applying:

1. Create a new local frame with the same parent as the function value in the current environment.
2. Bind the name of the function to the function value in the current environment.
3. Evaluate the expression(s) on the right of the equal sign.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

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. Evaluate the expression(s) on the right of the equal sign.
2. Simultaneously bind the names on the left to those values, in the first frame of the current environment.

Execution rule for conditional statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.
3. If it is a false value, execute the suite, then skip the remaining clauses in the statement.

Evaluation rule for or expressions:

1. Evaluate the left operand subexpression.
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 right operand subexpression.

Evaluation rule for and expressions:

1. Evaluate the subexpression on the 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 on the right.

Evaluation rule for not expressions:

1. Evaluate exp; The value is True if the result is a false value, and False otherwise.
2. Evaluate the header's expression.
3. If it is a true value, execute the (whole) suite, then return to step 1.

Function of a single argument (not called term)

A formal parameter that will be bound to a function

Function that takes a function value as an argument

Higher-order function: A function that takes a function value as an argument

Nested def statements: Functions defined within other function bodies are bound to names in the local frame

Nested frame: Functions defined within other function bodies are bound to names in the local frame

Pure Functions

-2 \[ \text{abs(number):} \]

2, 10 \[ \text{pow(x, y):} \]

Non-Pure Functions

-2 \[ \text{print(...):} \]

A and B:
True if A is True and B is True
False if A is False or B is False

A or B:
True if A is True or B is True
False if A is False and B is False

An environment is a sequence of frames

A function is non-nested if it consists of one local frame, followed by the global frame

Program output:

-2 \[ \text{abs(-2):} \]

2, 10 \[ \text{pow(2, 5):} \]
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder

A function that returns a function
A local def statement
The name add_three is bound to a function
Can refer to names in the enclosing function

When a function is defined:
1. Create a function value: func <name>(<formal parameters>)
2. Its parent is the current frame.
3. Bind <name> to the function value in the current frame (which is the first frame of the current environment).

When a function is called:
1. Add a local frame, titled with the <name> of the function being called.
2. Copy the parent of the function to the local frame: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.

def search(f):
    """Return the smallest non-negative integer x for which f(x) is a true value.
    """
    x = 0
    while True:
        if f(x):
            return x
        x += 1

def is_three(x):
    """Return whether x is three.
    >>> search(is_three)
    3
    """
    return x == 3

def inverse(f):
    """Return a function g(y) that returns x such that f(x) == y.
    >>> sqrt = inverse(lambda x: x * x)
    >>> sqrt(16)
    4
    """
    return lambda y: search(lambda x: f(x)==y)

False values so far: 0, False, '', None
Anything value that's not false is true.

if 0:  >>> print('*')  
>>> if 1 and 0:  >>> print('*')
    >>> if 1:  >>> print('*')
        >>> print('*')
    >>> if abs:  >>> print('*')
        >>> print('*')
        
from math import sqrt

def isPrime(n):
    i = 2
    while i <= int(sqrt(n)):
        if n % i == 0:
            return False
        i = i + 1
    return True