

CS61a

Lec 1: Expressions

- Call Expressions in Python In terminal window of python, you can perform simple calculation and calling several functions as follows

```
>>> from operator import add, mul
>>> add(2, 3)
5
>>> mul(2, 3)
6
>>> mul(add(2, mul(4, 6)), add(3, 5))
208
```

- Anatomy of a Call Expression **OPERATOR(OPERAND, OPERAND)**
- Evaluating Nested Expressions

Lec 2: Functions

```
>>> f = max
>>> f(1, 2, 3)
3
```

– Defining Functions

```
>>> def square(x):
    return mul(x, x)
>>> square(11)
121
```

Binds names to values

```
>>> def <name>(<formal parameter>)
    return <return expression>
```

Function's signature

– Calling User-Defined Functions

Looking up names in environments

- Procedure for calling/applying user-defined functions:
 - i. Add a local frame, forming a new environment
 - ii. Bind the function's formal parameters to its arguments in that frame
 - iii. Execute the body of the function in that new environment

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

– Environment Diagrams

- Visualize the interpreter's program
- Different frames: Global frame and local frame
- Within a frame, a name cannot be repeated
- Assignment statements change the value bound to the name within the certain frame
- Execution rule:
 - i. Evaluate all expressions to the right of `=` from left to right.
 - ii. Bind all names to the left of `=` to those resulting values in the current frame

– Print and none

None is a special value that indicates that nothing is returned

```
>>> print(print(1), print(2))
1
2
None None
```

- Pure functions: just return values
- Non-pure functions: have side effects (anything that happens as a consequence of calling a function)

Lec 3: Control

Multiple Environments in One Diagram

– Operators

- Division: `/` (`truediv`)
- Integer division: `//` (`floordiv`) e.g. `2013 // 10 = 201` , `2013 / 10 = 201.3`
- Mod: `%` (`mod`)

– Multiple return values

```
>>> def divide_exact(n, d):
        return n // d, n % d
>>> quotient, remainder = divide_exact(2013, 10)
>>> quotient
201
>>> remainder
3
```

To run python file in an interactive way, \$ python3 -i ex.py In the python file,

```
"""Our first Python source file."""
from operator import floordiv, mod
def divide_exact(n, d):
    """Return the quotient and remainder of dividing N by D
    >>> q, r = divide_exact(2013, 10)
    >>> q
    201
    >>> r
    3
    """
    return floordiv(n, d), mod (n, d)
```

Then execute in the terminal window python3 -m doctest ex.py Or python3 -m doctest -v ex.py (with clear documentation test outputs) And you can bind default value to the formal parameters def divide_exact(n, d = 10)

– Conditionals

```
def absolute_value(x):
    """Return the absolute value of x"""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

- Boolean contexts
 - False values: False, 0, ' ', None
 - True values: Anything else

Lec 4: Higher–Order Functions

– Iteration

To implement the famous Fibonacci Sequence,

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1"""
    pred, curr = 0, 1
    k = 1
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

– If Call Expression

This is somehow different from the traditional conditionals with if–else statement

```
def if_(c, t, f):
    if c:
        return t
    else:
        return f

from math import sqrt

def real_sqrt(x):
    """Return the real part of the square root of x"""
    return if_(x >= 0, sqrt(x), 0)
```

```
$ python3 -i ex.py
>>> real_sqrt(-16)
ValueError: math domain error
```

Why is there an error? When executing `return if_(x >= 0, sqrt(x), 0)`, it will get the value of `sqrt(x)`, which is erroneous

– Logical Operators

- To evaluate the expression `<left> and <right>` :
 - i. Evaluate the subexpression `<left>` .
 - ii. If the result is a false value `v`, then the expression evaluates to `v`.
 - iii. Otherwise, the expression evaluates to the value of the subexpression `<right>` .
- To evaluate the expression `<left> or <right>` :

- i. Evaluate the subexpression `<left>` .
 - ii. If the result is a true value `v`, then the expression evaluates to `v`.
 - iii. Otherwise, the expression evaluates to the value of the subexpression `<right>` .
- Short-circuit evaluation.
 - `or` operator: If the first operand is considered `True` in a boolean context (e.g., **non-zero numbers, non-empty sequences, non-empty strings, etc.**), it immediately returns that operand, and subsequent operands are not evaluated.
 - For example, when evaluating `-1 or 5` , since `-1` is non-zero and thus `True` , the result of the expression is `-1` without evaluating `5` .
 - `and` operator: If the first operand is considered `False` in a boolean context (e.g., `0` , `False` , `None` , empty sequences like `[]` , `{}` , or `""`), it immediately returns that operand, and subsequent operands are not evaluated. However, if the first operand is `True` , it continues to evaluate the next operand, and the final result is the value of the last operand if all operands are `True` , or the first `False` operand it encounters.
 - For example, when evaluating `0 and 5` , since `0` is considered `False` , the result of the expression is `0` without evaluating `5` . On the other hand, when evaluating `5 and 10` , since `5` is `True` , it then evaluates `10` , and the result is `10` .
 - For more practice:

```
$ cd lab02
$ python3 ok -q short-circuit -u
```

e.g.

```
from math import sqrt

def has_big_sqrt(x):
    return x > 0 and sqrt(x) > 10

def reasonable(n):
    return n == 0 or 1 / n != 0

>>> has_big_sqrt(1000)
True
>>> has_big_sqrt(-1000)
False
>>> reasonable(10 ** 10000)
False
>>> reasonable(0)
True
```

– Higher-Order Functions

1. It could be a function that takes another function as an argument

- Assertion

```
>>> assert 3 > 2, 'Math is broken'
>>> assert 2 > 3, 'That is false'
AssertionError: That is false
```

```
def area_square(r):
    assert r > 0, 'A length must be positive'
    return r * r
```

2. It could also be a function that returns a function as return value

```
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
```

Another example for Homework 2:

```
def make_repeater(f, n):
    """Returns the function that computes the nth application of f.
    >>> add_three = make_repeater(increment, 3)
    >>> add_three(5)
    8
    >>> make_repeater(triple, 5)(1) # 3 * (3 * (3 * (3 * (3 * 1))))
    243
    >>> make_repeater(square, 2)(5) # square(square(5))
    625
    >>> make_repeater(square, 3)(5) # square(square(square(5)))
    390625
    """
    def repeater(x):
        i = 0
        res = x
        while i < n:
            res = f(res)
            i = i + 1
        return res
    return repeater
```

Lec 5: Environments

– Environment for higher-order functions

```
def apply_twice(f, x):  
    return f(f(x))  
def square(x):  
    return x * x  
  
>>> apply_twice(square, 3)  
81
```

Applying a user-defined function

- Create a new frame
- Bind formal parameters (*f* & *x*) to arguments
- Execute the body: `return f(f(x))`

– Environment for nested definitions

- When a function is defined:
 - Create a function value `func <name>(<formal parameters>) [parent = current_frame]`
 - Bind `<name>` to the function value in the current frame
- When a function is called
 - Add a local frame, titled with the `<name>` of the function being called
 - Copy the parent of the function to the local frame: `[parent = <label>]`
 - Bind the `<formal parameter>` to the arguments in the local frame
 - Execute the body of the function in the environment that starts with the local frame

– Local Names

- Local names are not visible to other (non-nested) functions

```
def f(x, y):  
    return g(x)  
def g(a):  
    return a + y  
result = f(1, 2)  
  
NameError: global name 'y' is not defined
```

When executing, firstly, try to find `y` in the frame of `g`, but failed. The function `g` belongs to the global frame, so again search for `y` in the global frame, but failed again.

– Function Composition

```
def square(x):  
    return x * x  
  
def triple(x):  
    return 3 * x  
  
def compose1(f, g):  
    def h(x):  
        return f(g(x))  
    return h
```

– Lambda Expressions

```
>>> x = 10  
>>> square = x * x  
>>> square  
100  
>>> x = 20  
>>> square  
100  
>>> square = lambda x: x * x  
>>> square(4)  
16  
>>> (lambda x: x * x)(3)  
9
```

Expressions with the keyword `lambda` evaluates to a function, e.g. `lambda x: x * x` is a function with formal parameter `x` and returns the value of `x * x`. Not very common in Python, and cannot contain statements, only `return` statement allowed

Only the `def` statement gives the function an intrinsic name

```
>>> square = lambda x: x * x  
>>> square  
<function <lambda> at 0x1003c1bf8>  
>>> def square(x):  
    return x * x  
>>> square  
<function square at 0x10293e730>
```

– Currying

- Transforming a multi-argument function into a single-argument, higher-order function


```
def curry2(f):
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
$ python3 -i ex.py
>>> from operator import add
>>> add(2, 3)
5
>>> m = curry2(add)
>>> add_three = m(3)
>>> add_three(2)
5
```

or

```
curry2 = lambda f: lambda x: lambda y: f(x, y)
```

Lec 6: Sounds (Optional)

Lec 7: Functional Abstractions

– Lambda Function Environment

A lambda function's parent is the current frame in which the lambda expression is evaluated

– Return Statements

A return statement completes the evaluation of a call expression and provide its value Inverse functions:

```
def search(f):
    x = 0
    while not f(x):
        if f(x):
            return x
        x += 1

def inverse(f):
    """Return g(y) such that g(f(x)) -> x."""
    return lambda y: search(lambda x: f(x) == y)

>>> sqrt = inverse(square)
>>> square(16)
```

```
256
>>> sqrt(256)
16
```

– Abstractions

- Choosing names
 - Names should convey the meaning or purpose of the values to which they are bound
 - Typically convey their effect
 - Names can be long if they help document your code
- Which values deserve a name?
 - Repeated compound expressions
 - Meaningful parts of complex expressions

– Errors & Tracebacks

Lec 8: Function Examples

– What would Python display?

```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

>>> horse(mask)
```

– Decorators

```
def trace1(fn):
    """Returns a version of fn that first prints out how it is called.

    fn - a function of 1 argument
    """
    def traced(x):
        print('Calling', fn, 'on argument', x)
        return fn(x)
    return traced

@trace1
def square(x):
    return x * x
```

```
def sum_squares_up_to(n):
    k, total = 1, 0
    while k <= n:
        total, k = total + square(k), k + 1
    return total

>>> sum_squares_up_to(5)
Calling <function square at 0x7f385941c7c0> on argument 1
Calling <function square at 0x7f385941c7c0> on argument 2
Calling <function square at 0x7f385941c7c0> on argument 3
Calling <function square at 0x7f385941c7c0> on argument 4
Calling <function square at 0x7f385941c7c0> on argument 5
55
```

or instead of `@trace1`, you could try `square = trace1(square)`

A decorator in Python is a function that takes another function as input and returns a modified version of it.

Basic syntax:

```
def decorator_function(func):
    def wrapper():
        # code to run before func
        result = func()
        # code to run after func
        return result
    return wrapper

@decorator_function
def original_function():
    pass
# just a placeholder in Python
```

Lec 9: Recursion

TLDR: Too long; Didn't Read

– Self-Reference

```
def print_all(x):
    print(x)
    return print_all
```

```
print_all(1)(3)(5)
```

```
$ python ex.py
```

```
1
```

3
5

– Recursive Functions

- A function is called recursive if the body of that function calls itself, either directly or indirectly

e.g. Digit Sums

```
def split(n):
    """Split positive n into (1) all but its last digit and (2) its last digit."""
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    # Base cases:
    if n < 10:
        return n
    # Recursive cases:
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

– Recursion in Environment Diagrams

- The same function is called multiple times.
- Different frames keep track of the different arguments in each call
- What n evaluates to depends upon which is the current environment

– Verifying the Correctness of Recursion

1. Verify the base case
2. Treat f as a functional abstraction!
3. Assume that $f(n - 1)$ is correct
4. Verify that $f(n)$ is correct, assuming that $f(n - 1)$ is correct

– Mutual Recursion

e.g. The Luhn Algorithm

- Used to verify credit card numbers
- Rule:

- i. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., $7 * 2 = 14$), then sum the digits of the products (e.g., 10: $1 + 0 = 1$; 14: $1 + 4 = 5$)
 - ii. Take the sum of all the digits (the Luhn sum)
- The Luhn sum of a valid credit card is a multiple of 10

```
def split(n):
    return n // 10, n % 10

def sum_digits(n):
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last

def luhn_sum(n):
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return luhn_sum_double(all_but_last) + last

def luhn_sum_double(n):
    all_but_last, last = split(n)
    luhn_digit = sum_digits(2 * last)
    if n < 10:
        return luhn_digit
    else:
        return luhn_sum(all_but_last) + luhn_digit
```

– Recursion and Iteration

- Iteration is a special case of recursion

Converting Recursion to Iteration

- Idea: Figure out what state must be maintained by the iterative function

```
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```

Converting Iteration to Recursion

- More formulaic because iteration is a special case of recursion

```
digit_sum = 0
def sum_digits_rec(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
```

Lec 10: Tree Recursion

– Order of Recursive Calls

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n // 10)
        print(n)
```

```
>>> cascade(12345)
12345
1234
123
12
1
12
123
1234
12345
```

Each `cascade` frame is from a different call to `cascade`. Until the Return value appears, that call has not completed

Another version of `cascade` :

```
def cascade(n):
    print(n)
    if n >= 10:
        cascade(n // 10)
    print(n)
```

This version, shorter though it seems, is less clear somehow, didn't clearly put the base case first

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)

grow = lambda n: f_then_g(grow, print, n // 10)
shrink = lambda n: f_then_g(print, shrink, n // 10)

>>> inverse_cascade(5)
1
12
123
1234
12345
1234
123
12
1
```

– Tree Recursion

- Tree-shaped processes arise whenever executing the body of a recursive function makes more than one call to that function.
- e.g. Fibonacci Sequence

```
>>> fib(35)
(After a long, long time) 9227465
```


- This process is highly repetitive; fib is called on the same argument multiple times

– Example: Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order
e.g. For `count_partitions(6, 4)`:

$$2 + 4 = 6 \quad 1 + 1 + 4 = 6 \quad 3 + 3 = 6 \quad 1 + 2 + 3 = 6 \quad 1 + 1 + 1 + 3 = 6 \quad 2 + 2 + 2 = 6 \quad 1 + 1 + 2 + 2 = 6$$

$$6 \quad 1 + 1 + 1 + 1 + 2 = 6 \quad 1 + 1 + 1 + 1 + 1 + 1 = 6$$

 example of calculating `count_partitions(6, 4)`

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0

    else:
        with_m = count_partitions(n - m, m)
        without_m = count_partitions(n, m - 1)
        return with_m + without_m
```

– Example: Count Dollars

See in `cs61a/hw03/hw03.py`

```
def next_smaller_dollar(bill):
    """Returns the next smaller bill in order."""
    if bill == 100:
        return 50
    if bill == 50:
        return 20
    if bill == 20:
        return 10
    elif bill == 10:
        return 5
    elif bill == 5:
        return 1

def count_dollars(total):
    """Return the number of ways to make change.
    >>> count_dollars(15) # 15 $1 bills, 10 $1 & 1 $5 bills, ... 1 $5 & 1 $10 bills
    6
    >>> count_dollars(10) # 10 $1 bills, 5 $1 & 1 $5 bills, 2 $5 bills, 10 $1 bills
    4
    >>> count_dollars(20) # 20 $1 bills, 15 $1 & $5 bills, ... 1 $20 bill
    10
    >>> count_dollars(45) # How many ways to make change for 45 dollars?
    44
    >>> count_dollars(100) # How many ways to make change for 100 dollars?
    344
    """
    def helper(total, bill):
        if total == 0 or total == 1:
            return 1
        elif total < 0:
            return 0
        if bill == None:
            return 0
        use_bill = helper(total - bill, bill)
```



```
    not_use_bill = helper(total, next_smaller_dollar(bill))
    return use_bill + not_use_bill
return helper(total, 100)
```

Lec 11: Sequences

– Lists

```
>>> odds = [41, 43, 47, 49] # list literal
>>> len(odds)
4
>>> odds[0] # zero-based index
41
>>> getitem(odds, 0)
41
```

Concatenation and repetition:

```
>>> digits = [1, 8, 2, 8]
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
# or
>>> add([2, 7], mul(digits, 2))
```

Nested lists:

```
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```

```
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 5 not in digits
True
>>> '1' == 1
False
>>> '1' in digits
False
>>> [1, 8] in digits
False
```

To swap two elements with index i and j in a list:

```
s[i], s[j] = s[j], s[i]
```

– For Statements

```
def count(s, value):  
    """Count the number of times that value appears in sequence s."""  
    total = 0  
    for element in s:  
        if element == v:  
            total = total + 1  
    return total
```

Execution Procedure:

```
for <name> in <expression>:  
    <suite>
```

1. Evaluate the header `<expression>` , which must yield an iterable value
2. For each element in that sequence, in order: (1). Bind `<name>` to that element in the current frame (2). Execute the `<suite>`

Sequence Unpacking in For Statements

```
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]  
    # A sequence of fixed-length sequences  
>>> same_count = 0  
  
>>> for x, y in pairs:  
    if x == y:  
        same_count += 1  
>>> same_count  
2
```

- If you iterate over a list, but change the contents of that list at the same time, you may not visit all the elements. This can be prevented by making a copy of the list. You can either use a list slice (`list[:]`), or use the built-in `list` function to make sure the original list is not affected.

– Ranges

- A range is a sequence of consecutive (normally) integers The range type

- Look at the figure above! Just remember: the little arrow always points at the left space of the value inside the parenthesis, i.e., including the first value and excluding the final value
- Length = ending value – starting value Zero-based index
- Use list constructors to build range

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]
>>> list(range(4))
# Range with a 0 starting value
[0, 1, 2, 3]
```

```
def cheer():
    for _ in range(3):
        print('Go Bears!')
```

– List Comprehensions

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o', 'p']
>>> [letters[i] for i in [3, 4, 6, 8]]
['d', 'e', 'm', 'o']
```

```
>>> odds = [1, 3, 5, 7, 9]
>>> [x + 1 for x in odds]
[2, 4, 6, 8, 10]
```

```
>>> [x for x in odds if 25 % x == 0]
[1, 5]
```


This way we can make some functions way much easier

```
def divisors(n):
    return [1] + [x for x in range(2, n) if n % x == 0]
>>> divisors(18)
[1, 2, 3, 6, 9]
```

Lec 12: Containers

– Box-and-Pointer Notation

- The closure property of Data Types
 - A method for combining data values satisfies the closure property if the result of combination can itself be combined using the same method

- Closure is powerful because it permits us to create hierarchical structures
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on
- Lists are represented as a row of index-labeled adjacent boxes, one per element. Each box either contains a primitive value or points to a compound value  box-and-pointer notation

Slicing

- Syntax: `sequence[start:stop:step]`

```
>>> odds = [3, 5, 7, 9, 11]
>>> list(range(1, 3))
[1, 2]
```

```
>>> [odds[i] for i in range(1, 3)]
[5, 7]
```

```
>>> odds[1:3]
[5, 7]
>>> odds[:3]
[3, 5, 7]
>>> odds[1:]
[5, 7, 9, 11]
>>> odds[:]
[3, 5, 7, 9, 11]
```

- Slicing always creates new values

– Processing Container Values

Sequence Aggregation

- To aggregate means to combine separate elements together
- Several built-in functions take iterable arguments and aggregate them into a value
 - `sum(iterable[, start])`
 - It returns the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When iterable is empty, return start.

```
>>> sum([2, 3, 4])
9
>>> sum([2, 3, 4], 5)
14
>>> sum([[2, 3], [4]]) # the 'start' parameter defaults to 0, which cannot be added to a sequence
TypeError
```

```
>>> sum([[2, 3], [4]], [])
[2, 3, 4]
```

- `max(iterable[, key=func])` OR `max(a, b, c, ...[, key=func])`

```
>>> max(range(5))
4
>>> max(range(10), key=lambda x: x**2 + x - 1)
9
```

- `all(iterable) -> bool`
 - Return True if `bool(x)` is True for all values `x` in the iterable
 - `bool(x)` returns whether `x` evaluates to True or False in the Boolean context

```
>>> [x < 5 for x in range(5)]
[True, True, True, True, True]
>>> all([x < 5 for x in range(5)])
True
>>> all(range(5))
False
```

– String

- Strings are an abstraction

```
>>> 'f = lambda x: x * x'
'f = lambda x: x * x'
>>> exec('f = lambda x: x * x')
>>> f
<function <lambda> at 0x7fe1661f0860>
```

- String literals could have 3 forms: Single Quotes, Double Quotes (when string includes a single quote), Triple Quotes (be it single or double; can span multiple lines)
- Escape Sequences: A backslash "escapes" the following character
 - Common escape sequences include: `\n` for newline; line feed `\t` for tab `\\` for backslash `\"` for double quote (inside a double-quoted string) `\'` for single quote (inside a single-quoted string)

```
>>> city = 'Beijing'
>>> len(city)
7
```

```
>>> city[3]
'j'
```

- Careful: An element of a string is itself a string, but with only one element!
- However, the `in` and `not in` operators match substrings

```
>>> 'here' in "Where's Waldo"
True
>>> [2, 3, 4] in [1, 2, 3, 4, 5]
False
```

– Dictionaries

- Dictionaries are collections of key–value pairs

```
>>> numerals = {'I': 1, 'V': 5, 'X': 10}
>>> numerals['X']
10
```

```
>>> numerals[10]
KeyError
```

```
>>> list(numerals)
['I', 'V', 'X']
```

```
>>> numerals.values()
dict_values([1, 5, 10])
>>> list(numerals.values())
[1, 5, 10]
```

```
>>> len(numerals)
3
```

```
>>> {1: 'first', 1: 'second'}
{1: 'second'}
```

- Two restrictions for keys:
 - The key cannot be a list or a dictionary

```
>>> {[1]: 'first'}
TypeError: unhashable type: 'list'
```

- Two keys cannot be equal; There can be at most one value for a given key
- Dictionary Comprehensions: `{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}`

- An expression that evaluates to a dictionary using this evaluation procedure
 - i. Add a new frame with the current frame as its parent
 - ii. Create an empty result dictionary that is the value of the expression
 - iii. For each element in the iterable value of `<iter exp>` : (1) Bind `<name>` to that element in the new frame from step 1 (2) If `<filter exp>` evaluates to a true value, then add to the result dictionary an entry that pairs the value of `<key exp>` to the value of `<value exp>`

Lec 13: Data Abstraction

– Data Abstraction

- Isolate two parts of any program that uses data
 - How data are represented (as parts)
 - How data are manipulated (as units)
- Data abstraction is a methodology by which functions enforce an abstraction barrier between **representation** and **use** e.g. Rational numbers

`$ fraction = \large\frac{numerator}{denominator} $`

- `rational(n, d)` returns a rational number `x`
- `numerator(x)` returns the numerator of `x`

1. Constructor: It's a special method, usually `__init__`, in a class for initializing object attributes when the object is created.
2. Selector: It's a method or function used to select or retrieve specific data from an object or data collection.

```
def mul_rational(x, y):
    return rational(numerator(x) * numerator(y), denominator(x) * denominator(y))
def add_rationals(x, y):
    nx, dx = numerator(x), denominator(x)
    ny, dy = numerator(y), denominator(y)
    return rational(nx * dy + ny * dx, dx * dy)
def equal_rational(x, y):
    return numerator(x) * denominator(y) == numerator(y) * denominator(x)
```

– Pairs

Representing pairs using lists A list literal: Comma-separated expressions in brackets

```
>>> pair = [1, 2]
>>> x, y = pair
```

```
>>> x # Unpacking a list
1
>>> pair[0]
1
>>> from operator import getitem
>>> getitem(pair, 0)
1
```

- Reducing to Lowest Terms

```
from fractions import gcd

def rational(n, d):
    g = gcd(n, d)
    return [n // g, d // g]
```

– Data Representation

- We need to guarantee that constructor and selector functions work together to **specify the right behavior**
- Behavior condition: If we construct rational number x from numerator n and denominator d , then $\text{numer}(x) / \text{denom}(x)$ must equal n / d
- Data abstractions uses selectors and constructors to define behavior
- If behavior conditions are met, then the representation is valid

...

```
# Constructors and selectors
def rational(n, d):
    """Construct a rational number x that represents n / d"""
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select
>>> x, y = rational(1, 2), rational(3, 8)
>>> print_rational(mul_rational(x, y))
3 / 16
>>> x
<function rational.<locals>.select at 0x10293e6a8>
```

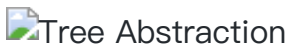
– Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using only...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational, rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals	two – element lists	list literals and element selection

Each new row is separated apart by abstraction barriers

Lec 14: Trees

– Tree Abstraction



Tree Abstraction

- Implementing the Tree Abstraction

```
# Constructor:
def tree(label, branches=[]):
    # By default, the branches is empty
    for branch in branches:
        assert is_tree(branch) 'branches must be trees' # Verifies the tree definition
    return [label] + list(branches)
    # Creates a list from a sequence of branches

# Selector:
def label(tree):
    # The label is the value of root for a tree
    return tree[0]

def branches(tree):
    return tree[1:]

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True

def is_leaf(tree):
```

```
return not branches(tree)
```

– Tree Processing

```
def fib_tree(n):  
    if n < 1:  
        return tree(n)  
    else:  
        left, right = fib_tree(n - 2), fib_tree(n - 1)  
        return tree(label(left) + label(right), [left, right])
```

– Tree Processing Uses Recursion

```
def count_leaves(t):  
    """Count the leaves of a tree"""  
    if is_leaf(t):  
        return 1  
    else:  
        branch_counts = [count_leaves(b) for b in branches(t)]  
        return sum(branch_counts)  
  
def increment_leaves(t):  
    """Return a tree like t but with leaf labels incremented."""  
    if is_leaf(t):  
        return tree(label(t) + 1)  
    else:  
        bs = [increment_leaves(b) for b in branches(t)]  
        return tree(label(t), bs)  
  
def increment(t):  
    """Return a tree like t but with all labels incremented"""  
    return tree(label(t) + 1, [increment(b) for b in branches(t)])
```

– Example: Summing paths

A basic example:

```
def fact_times(n, k):  
    """Return k * n * (n - 1) * ... * 1"""  
    if n == 0:  
        return k  
    else:  
        return fact_times(n - 1, k * n)
```

```
def print_sums(t, so_far):
    so_far = so_far + label(t)
    if is_leaf(t):
        print(so_far)
    else:
        for b in branches(t):
            print_sums(b, so_far)
```

Lec 15: Mutability

– Objects

```
>>> from datetime import date
>>> date
<class 'datetime.date'>

>>> today = date(2025, 1, 27)
>>> today
datetime.date(2025, 1, 27)
>>> str(today)
'2025-01-27'

>>> spring_festival = date(2025, 1, 29)
>>> spring_festival - today
datetime.timedelta(days=2)

>>> str(spring_festival - today)
'2 days, 0:00:00'

>>> today.year
2025
>>> today.month
1
>>> today.strftime('%A %B %d')
'Monday January 27'
```

- Objects represent information
- They consist of data and behavior, bundled together to create abstractions
- Objects can represent things, but also properties, interactions, & processes
- A type of object is called a class; classes are first-class values in Python
- Object-oriented Programming:
 - A metaphor for organizing large programs
 - Special syntax that can improve the composition of programs
- In Python, every value is an object

- All objects have attributes
- A lot of data manipulation happens through object methods
- Functions do one thing; objects do many related things

– Example: Strings

```
>>> s = 'Hello'
>>> s.upper()
'HELLO'
>>> s.swapcase()
'hELLO'

>>> a = 'A'
>>> ord(a)
65
>>> hex(ord(a))
'0x41' # 4th row, 1st column in the layout of ASCII table

>>> print('\a')
# Make a bell-like sound
```



ASCII

Beside ASCII, there is also the Unicode Standard

```
>>> from unicodedata import name, lookup
>>> name('A')
'LATIN CAPITAL LETTER A'
>>> lookup('WHITE SMILING FACE')
'😊'
>>> lookup('SNOWMAN')
'⛄'
```

– Mutation Operations

- Some objects can change

```
>>> suits = ['coin', 'string', 'myriad']
>>> original_suits = suits

>>> suits.pop()
'myriad'
>>> suits
['coin', 'string']

>>> suits.remove('string')
>>> suits
```

```

['coin']

>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
>>> suits
['coin', 'cup', 'sword', 'club']

>>> suits[2] = 'spade'
>>> suits[0:2] = ['heart', 'diamond']
>>> suits
['heart', 'diamond', 'spade', 'club']

>>> original_suits
['heart', 'diamond', 'spade', 'club']

```

It turns out that in Python, both `suits` and `original_suits` are references to the same list object. Think of them as two different names for the same thing.

This is the first example in the course of an object changing state. The same object can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation. Only objects of mutable types can change: lists & dictionaries. In a dictionary,

```

>>> numerals = {'I': 1, 'V': 5, 'X': 10}
>>> numerals
{'I': 1, 'V': 5, 'X': 10}

>>> numerals['X']
10
>>> numerals['X'] = 11
>>> numerals
{'I': 1, 'V': 5, 'X': 11}
>>> numerals['L'] = 50
>>> numerals
{'I': 1, 'V': 5, 'X': 11, 'L': 50}

>>> numerals.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pop expected at least 1 argument, got 0

>>> numerals.pop('X')
11
>>> numerals.remove('X')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'remove'

>>> numerals.get('X') # Nothing happens since there is no key 'X' anymore

```

```
>>> numerals
{'I': 1, 'V': 5, 'L': 50}
```

A function can change the value of any object in its scope

– Tuples

- Tuples are sequences, but **immutable**.
- It could be pronounced as 'toople' or 'tuhple'.

Guido van Rossum (the father of Python)

I pronounce tuple too–pull on Mon/Wed/Fri and tub–pull on Tue/Thu/Sat. On Sunday I don't talk about them.

```
>>> (3, 4, 5, 6)
(3, 4, 5, 6)
>>> 3, 4, 5, 6 # No parenthesis is OK
(3, 4, 5, 6)

>>> tuple([3, 4, 5, 6]) # Convert any sequence to a tuple
(3, 4, 5, 6)
>>> (2)
2
>>> (2,) # Create a tuple with only one element, with strange syntax
(2,)
>>> (3, 4) + (5, 6)
(3, 4, 5, 6)
>>> {(1, 2): 3} # Keys are OK to be a tuple, but not a list
{(1, 2): 3}
>>> {[1, 2]: 3}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

Immutable values are protected from mutation. An immutable sequence may still change if it contains a mutable value as an element, e.g.

```
>>> s = ([1, 2], 3)
>>> s[0] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> s[0] = [4, 2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
>>> s[0][0] = 4
>>> s
([4, 2], 3)
```

– Mutation

- Sameness and change
 - A list is still "the same" list even if we change its contents, just like a baby growing up but without changing his/her identity

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a == b
True
>>> a
[10, 20]
>>> b
[10, 20]
```

- Conversely, we can have two lists that happen to have the same contents, but are different

```
>>> a = [10]
>>> b = [10] # Didn't assign b = a
>>> a == b
True
>>> b.append(20)
>>> a == b
False
>>> a
[10]
>>> b
[10, 20]
```

- Identity Operators
 - `<exp0> is <exp1>` evaluates to True if both `<exp0>` and `<exp1>` evaluate to **the same object**
 - C.f. `<exp0> == <exp1>` evaluates to True if both `<exp0>` and `<exp1>` evaluate to **equal values**
 - Identical objects are always equal values, **but not the other way around**
- Mutable Default Arguments are Very Dangerous

```
>>> def f(s=[]):
...     s.append(5)
...     return len(s)
...
>>> f()
1
>>> f()
2
>>> f()
3
```

In Python, when you define a function with a default argument value, **the default value is evaluated only once at the time the function is defined**, not each time the function is called. This can lead to unexpected behavior when the default argument is a mutable object like a list, dictionary, or set.

– Mutable Functions

- A function with behavior that varies over time

```
# Let's model a bank account that has a balance of $100
>>> withdraw(25)
75
>>> withdraw(25)
50
>>> withdraw(60)
'Insufficient funds'
```

- Mutable Values & Persistent Local State  Mutable Values & Persistent Local State

- Extensive knowledge: the rule of LEGB, Local, Enclosing, Global, Built-in
- `nonlocal` keyword changes the "enclosing" part
- So we can also do this:

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        # We want to change the variable that belongs the outer scope
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Lec 16: Iterators

– Iterators

- A container can provide an iterator that provides access to its element in some order
 - `iter(iterable)` : Returns an iterator over the elements of an iterable value
 - `next(iterator)` : Return the next element in an iterator

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
```

```
>>> s = [1, 2, 3, 4, 5]
>>> t = iter(s)
>>> next(t)
1
>>> next(t)
2
>>> next(t)
3
>>> list(t)
[4, 5]
>>> next(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

– Dictionary Iteration

- All iterators are mutable
- A dictionary, its keys, its values, and its items are all iterable values
 - The order of items in a dictionary is the order in which they were added (Python 3.6+)

```
# Keys:
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```
# Values:
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
# Items:
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
>>> next(i)
>>>
```

– For Statements

```
>>> r = range(3, 6)
>>> ri = iter(r)
>>> next(ri)
3
>>> for i in ri:
...     print(i)
...
4
5
>>> for i in ri:
...     print(i)
...
# Nothing happens
```

– Built-In Iterator Functions

- Many built-in Python sequences operations return iterators that compute results lazily
 - `map(func, iterable)` : Iterate over `func(x)` for `x` in `iterable`
 - `filter(func, iterable)` : Iterate over `x` in `iterable` if `func(x)`
 - `zip(first_iter, second_iter)` : Iterate over co-indexed `(x, y)` pairs
 - `reversed(sequence)` : Iterate over `x` in a sequence in reverse order
- To view the contents of an iterator, place the resulting elements into a container such as `list()`, `tuple()` or `sorted()`

```
>>> bcd = ['b', 'c', 'd']
>>> [x.upper() for x in bcd]
['B', 'C', 'D']
>>> bcd_map = map(lambda x: x.upper(), bcd)
>>> bcd_map
<map object at 0x7f6b8ec2ffa0>
```

```
>>> list(bcd_map)
['B', 'C', 'D']

# Or, without calling list(bcd_map):
>>> bcd_map = map(lambda x: x.upper(), bcd)
>>> next(bcd_map)
'B'
>>> next(bcd_map)
'C'
>>> next(bcd_map)
'D'
```

- Lazy computation: Only when we ask for the next element is the function applied and the result computed

```
>>> def double(x):
...     print('**', x, '=>', 2 * x, '**')
...     return 2 * x
...
>>> map(double, [3, 5, 7])
<map object at 0x7f6b8eaa6650>
>>> m = map(double, [3, 5, 7])
>>> next(m)
** 3 => 6 **
6
>>> next(m)
** 5 => 10 **
10
>>> next(m)
** 7 => 14 **
14
```

```
>>> m = map(double, range(3, 7))
>>> f = lambda y: y >= 10
>>> t = filter(f, m)
>>> next(t)
** 3 => 6 **
** 4 => 8 **
** 5 => 10 **
10
```

– Zip

- The built-in zip function returns an iterator over co-indexed tuples

```
>>> list(zip([1, 2, 3], [4, 5, 6]))
[(1, 4), (2, 5), (3, 6)]
>>>
```

- If one iterable is longer than the other, zip only iterates over matches and skips extras

```
>>> list(zip([1, 2], [3, 4, 5], [6, 7]))
[(1, 3, 6), (2, 4, 7)]
```

```
>>> s = [3, 1, 4, 1, 3]
>>> list(zip(s, reversed(s)))
[(3, 3), (1, 1), (4, 4), (1, 1), (3, 3)]
def palindrome(s):
    """Return whether s is the same backward and forward"""
    return all([a == b for a, b in zip(s, reversed(s))])
```

– Using Iterators

Reasons for Using Iterators

- Code that processes an iterator (via `next`) or iterable (via `for` or `iter`) makes few assumptions about the data itself.
 - Changing the data representation from a list to a tuple, map object, or dict_keys doesn't require rewriting code.
 - Others are more likely to be able to use your code on their data.
- An iterator bundles together a sequence and a position within that sequence as one object.
 - Passing that object to another function always retains the position.
 - Useful for ensuring that each element of a sequence is processed only once.
 - Limits the operations that can be performed on the sequence to only requesting next.

Lec 17: Generators

– Generators

- Generator is a special kind of iterator
- Use `yield` keyword

```
>>> def minus_plus(x):
...     yield -x
...     yield x
...
>>> t = minus_plus(3)
>>> next(t)
-3
>>> next(t)
3
```

```
>>> next(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> t
<generator object minus_plus at 0x7f503c9a1780>
```

- A generator function is a function that yields values instead of returning them
- A normal function returns once; a generator function can yield multiple times
- A generator is an iterator created automatically by calling a generator function
- When a generator function is called, it returns a generator that iterates over its yields

```
def evens(start, end):
    even = start + (start % 2) # Not sure if start is even
    while even < end:
        yield even
        even += 2

>>> list(evens(1, 10))
[2, 4, 6, 8]
```

– Generators & Iterators

- Generators can Yield from Iterators
- A `yield from` statement yields all values from an iterator or iterable (Python 3.3)

```
def a_then_b(a, b):
    for x in a:
        yield x
    for x in b:
        yield x

# Shorthand version
def a_then_b(a, b):
    yield from a
    yield from b
```

e.g.

```
def countdown(k):
    if k > 0:
        yield k
        # The old way of doing this:
        for x in countdown(k - 1):
            yield x
        # Or the simpler way with "yield from"
```

```

        yield from countdown(k - 1)
    else:
        yield 'Blast off!'

def prefixes(s):
    if s: # If s is not empty
        yield from prefixes(s[:-1]) # Get rid of the final character
        yield s

def substrings(s):
    if s:
        yield from prefixes(s)
        yield from substrings(s[1:])

```

– Example: Partitions

```

def count_partitions(n, m):
    if n < 0 or m == 0:
        return 0
    else:
        exact_match = 1 if n == m else 0
        with_m = count_partitions(n - m, m)
        without_m = count_partitions(n, m - 1)
        return exact_match + with_m + without_m

def list_partitions(n, m):
    if n < 0 or m == 0:
        return []
    else:
        exact_match = []
        if n == m:
            exact_match = [[m]]
        with_m = [p + [m] for p in list_partitions(n - m, m)]
        without_m = list_partitions(n, m - 1)
        return exact_match + with_m + without_m

>>> for p in list_partitions(6, 4):
...     print(p)

[2, 4]
[1, 1, 4]
[3, 3]
[1, 2, 3]
[1, 1, 1, 3]
[2, 2, 2]
[1, 1, 2, 2]
[1, 1, 1, 1, 2]
[1, 1, 1, 1, 1, 1]

```

```

def partitions(n, m):

```

```

"""Yield partitions"""
if n > 0 and m > 0:
    if n == m:
        yield str(m)
    for p in partitions(n - m, m):
        yield p + ' + ' + str(m)
    yield from partitions(n, m - 1)

```

Lec 18: Objects

– Object–Oriented Programming

- A method for organizing programs
 - Extends data abstraction
 - Bundles together information and related behavior
 - Each object has its own local state
 - Interact with an object using its **methods**
 - Several objects may be all instances of a common **class**
 - Different classes may relate to each other
- A class defines how objects of a particular type behave
- An object is an instance of a class; the class is its type
- A method is a function called on an object using a dot expression

– Class statements

- A class describes the behavior of its instances

```

class Account:
    # __init__ is a constructor method name for the function that constructs an Account instan
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    # self is the instance of the account class on which deposit was invoked using a dot expre
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if self.balance < amount:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance

```

– Creating Instances

Object Construction

```
>>> a = Account('Alan')
>>> a.balance
0
>>> a.balance = 12
>>> a.balance
12
>>> b = Account('Ada')
>>> b.balance
0
>>> b.balance = 20
>>> a.backup = b # A new attribute can be added at any time
>>> a.backup.balance
20
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('John')
>>> a is b
False
```

– Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's attributes

Lec 19: Attributes

– Class Attributes

The class statement:

```
class <name>:
    <suite>
```

Class attributes are "shared" accross all instances of a class because they are attributes of the class, not the instance


```
class Account:
    interest = 0.2
    def __init__(...):
        ...

>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
```

– Attribute Lookup

Both instances and classes have attributes that can be looked up by dot expressions

- To evaluate a dot expression `<expression>.<name>` :
 - i. Evaluate the `<expression>` to the left of the dot, which yields the object of the dot expression
 - ii. `<name>` is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
 - iii. If not, `<name>` is looked up in the class, which yields a class attribute value
 - iv. That value is returned unless it is a function, in which case a bound method is returned instead
- Using `getattr` , we can look up an attribute using a string

```
>>> getattr(tom_account, 'balance')
10
>>> hasattr(tom_account, 'deposit')
True
```

– Attribute Assignment

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

```
# Wrong: (Creating a new attribute in the object)
tom_account.interest = 0.04

# Correct
Account.interest = 0.04
>>> tom_account.interest
0.04
```



– Method Calls

- Methods are invoked using dot notation `<expression>.<name>`
- The `<expression>` can be any valid Python expression
- The `<name>` must be a simple name
- Evaluates to the value of the attribute looked up by `<name>` in the object that is the value of the `<expression>`

– Bound Methods

Bound methods are functions that are also class attributes, where the self argument has already been filled in with an instance of the class

- Terminology: Attributes, Functions and Methods Dot expressions evaluate to bound methods for class attributes that are functions Object + Function = Bound Method

```
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>

>>> Account.deposit(tom_account, 1001)
1001
>>> tom_account.deposit(1007)
2018
```

Lec 20: Inheritance

– Inheritance

- Inheritance is a method for relating classes together.
- The specialized class may have the same attributes as the general class, along with some special-case behavior

```
class <name>(<base class>):
    <suite>
```

- Conceptually, the new subclass "shares" attributes with its base class
- The subclass may override certain inherited attributes
- Using inheritance, we implement a subclass by specifying its differences from the base class

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)
20
>>> ch.withdraw(5) # Withdrawals incur a $1 fee
14
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

- Base class attributes aren't copied into subclasses!
- To look up a name in a class:
 - a. If it names an attribute in the class, return the attribute value
 - b. Otherwise, look up the name in the base class, if there is one

– Object–Oriented Design

- Don't repeat yourself; use existing implementations
- A is preferred to B: A is better than B
- Inheritance and Composition
 - Inheritance is best for representing *is-a* relationships
 - Composition is best for representing *has-a* relationships

```
class Bank:
    """A bank has accounts"""
    def __init__(self):
        self.accounts = []

    def open_account(self, holder, amount, kind=Account):
        account = kind(holder)
        account.deposit(amount)
        self.accounts.append(account)
        return account

    def pay_interest(self):
        for a in self.accounts:
            a.deposit(a.balance * a.interest)

    def too_big_to_fail(self):
        return len(self.accounts) > 1
```

– Multiple Inheritance:

A class may inherit from multiple base classes in Python:

```
class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        ....
```

Lec 21: Representations

– String Representations

- In Python, all objects produce two string representations:
 - The `str` is legible to humans
 - The `repr` is legible to the Python interpreter
- The `str` and `repr` strings are often the same, but not always
- For most object types, `eval(repr(object)) == object`

```
>>> min
<built-in function min>
>>> repr(min)
'<built-in function min>'
```

```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> half
Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
'1/2'
>>> print(half)
1/2
>>> eval(repr(half))
Fraction(1, 2)
>>> eval(str(half))
0.5
```

```
>>> s = "Hello, World"
>>> s
'Hello, World'
>>> print(repr(s))
'Hello, World'
>>> print(s)
Hello, World
>>> print(str(s))
```

```

Hello, World
>>> str(s)
'Hello, World'
>>> repr(s)
"'Hello, World'"
>>> eval(repr(s))
'Hello, World'
>>> repr(repr(repr(s)))
'\''\\\'Hello, World\\\'\"\\''
>>> eval(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'Hello' is not defined

```

– F-Strings

String interpolation involves evaluating a string literal that contains expressions Using string concatenation:

```

>>> from math import pi
>>> 'pi starts with' + str(pi) + '...'
'pi starts with3.141592653589793...'

```

Using string interpolation:

```

>>> f'pi starts with {pi}...'
'pi starts with 3.141592653589793...'

>>> f'2 + 2 = {2 + 2}'
'2 + 2 = 4'

```

- The result of evaluating an f-string literal contains the str string of the value of each sub-expression
- Sub-expressions are evaluated in the current environment

– Polymorphic Function

- A function that applies to many (poly) different forms (morph) of data
- `str` and `repr` are both polymorphic; they apply to any object
- `repr` invokes a zero-argument method `__repr__` on its argument; same for `str`

```

>>> half.__repr__()
'Fraction(1, 2)'

```

```
>>> half.__str__()
'1/2'
```

The behavior of `repr` is slightly more complicated than just invoking `__repr__` on its argument:

- An instance attribute called `__repr__` is ignored! Only class attributes are found

```
def repr(x):
    return type(x).__repr__(x)
# First, get its "class" using type(x)
# Then, use the __repr__ method in "class", with x as its "self" argument
```

The behavior of `str` is also complicated

- An instance attribute called `__str__` is ignored
- If no `__str__` attribute is found, uses the `repr` string
- Interfaces
 - Message passing: Objects interact by looking up attributes on each other
 - The attribute look-up rules allow different data types to respond to the same message
 - A shared message (attribute name) that elicits similar behavior from different object classes is a powerful method of abstraction
 - An interface is a set of shared messages, along with a specification of what they mean
 - E.g.: Classes that implement `__repr__` and `__str__` methods that return Python-interpretable and human-readable strings implement an interface for producing string representations

```
class Ratio:
    def __init__(self, n, d):
        self.numer = n
        self.denom = d
    def __repr__(self):
        return 'Ratio({0}, {1})'.format(self.numer, self.denom)
    def __str__(self):
        return '{0}/{1}'.format(self.numer, self.denom)
```

– Special Method Names

- Certain names are special because they have built-in behavior
- These names always start and end with two underscores

Special Method Name	Description
<code>__init__</code>	The method automatically invoked when an object is constructed

Special Method Name	Description
<code>__repr__</code>	The method invoked to display an object as a Python expression
<code>__add__</code>	The method invoked to add an object to another
<code>__bool__</code>	The method invoked to convert an object to True or False
<code>__float__</code>	The method invoked to convert an object to a float (real number)

```
>>> zero, one, two = 0, 1, 2
>>> one + two
3
>>> bool(zero), bool(one)
(False, True)
```

The same behavior using special methods:

```
>>> zero, one, two = 0, 1, 2
>>> one.__add__(two)
3
>>> zero.__bool__(), one.__bool__()
(False, True)
```

```
def gcd(n, d):
    # Using a special algorithm here
    while n != d:
        n, d = min(n, d), abs(n - d)
    return n

class Ratio:
    def __init__(self, n, d):
        ...
    def __add__(self, other):
        # Type dispatch
        if isinstance(other, int):
            n = self.numer + self.denom * other
            d = self.denom
        elif isinstance(other, Ratio):
            n = self.numer * other.denom + self.denom * other.numer
            d = self.denom * other.denom
        elif isinstance(other, float):
            return float(self) + other
        g = gcd(n, d) # Greatest common divisor
        return Ratio(n // g, d // g)

    __radd__ = __add__

    def __float__(self):
        return self.numer / self.denom
```

```

>>> Ratio(1, 3) + Ratio(1, 6)
Ratio(1, 2)
>>> Ratio(1, 3).__add__(Ratio(1, 6))
Ratio(1, 2)
>>> Ratio(1, 6).__radd__(Ratio(1, 3)) # Swap the order
Ratio(1, 2)

>>> Ratio(1, 3) + 1
Ratio(4, 3)
>>> 1 + Ratio(1, 3)
Ratio(4, 3)

```

Lec 22: Compositions

– Linked List

- A linked list is either empty or a first value and the rest of the linked list
- The first element is an attribute value
- The rest of the elements are stored in a linked list
- `Link.empty` is a class attribute that represents an empty linked list
- To create a linked list, use `Link(3, Link(4, Link(5, Link.empty)))`

```

class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

```

– Linked List Processing

```

def range_link(start, end):
    if start >= end:
        return Link.empty
    else:
        return Link(start, range_link(start + 1, end))

def map_link(f, s):
    if s is Link.empty:
        return s
    else:
        return Link(f(s.first), map_link(f, s.rest))

def filter_link(f, s):

```



```

if s is Link.empty:
    return s
filtered_rest = filter_link(f, s.rest)
if f(s.first):
    return Link(s.first, filtered_rest)
else:
    return filtered_rest

```

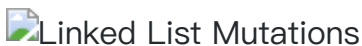
– Linked List Mutations

- Attribute assignment statements can change first and rest attributes of a link

```

>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2

```



– Example: Add to an Ordered List

```

def add(s, v):
    """Add v to an ordered list s with no repeats, returning modified s. (If v is already
    assert s is not List.empty
    if s.first > v:
        s.first, s.rest = v, Link(s.first, s.rest)
    elif s.first < v and empty(s.rest):
        s.rest = Link(v)
    elif s.first < v:
        add(s.rest, v)
    return s

```



– Tree Class

```

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
    def __repr__(self):
        if self.branches:

```

```

        branch_str = ',' + repr(self.branches)
    else:
        branch_str = ''
    return 'Tree({0}{1})'.format(repr(self.label), branch_str)
def __str__(self):
    return '\n'.join(self.indented())
def indented(self):
    lines = []
    for b in self.branches:
        for line in b.indented():
            lines.append(' ' + line)
    return [str(self.label)] + lines
def is_leaf(self):
    return not self.branches

def leaves(t):
    """Return a list of leaf labels in tree T"""
    if t.is_leaf():
        return [t.label]
    else:
        all_leaves = []
        for b in t.branches:
            all_leaves.extend(leaves(b))
        return all_leaves

def height(t):
    if t.is_leaf():
        return 0
    else:
        return 1 + max(height(branch) for branch in t.branches)

```

– Tree Mutation

- Removing subtrees from a tree is called *pruning*

```

def prune(t, n):
    """Prune all sub_trees whose label is n."""
    t.branches = [b for b in t.branches if b.label != n]
    for b in t.branches:
        prune(b, n)

```

Lec 23: Efficiency

– Measuring Efficiency

```

def fib(n):
    if n == 0 or n == 1:

```

```

        return n
    else:
        return fib(n - 2) + fib(n - 1)

# Decorator
def count(f):
    def counted(n):
        counted.call_count += 1
        return f(n)
    counted.call_count = 0
    return counted

>>> fib = count(fib)
>>> fib(5)
5
>>> fib.call_count
15

```

– Memoization

- Idea: Remember the results that have been computed before

```

def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized

>>> fib = memo(fib)
>>> fib(30)
832040

```

– Exponentiation

```

# Old version, less efficient:
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n - 1)

# New version, more efficient:
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:

```

```

        return square(exp_fast(b, n // 2))
    else:
        return b * exp_fast(b, n - 1)

def square(x):
    return x * x

```

- Jupyter Notebook is a common way that people use in order to execute Python code when the output is a graph or a chart
- The `exp(n)` takes linear time, while `exp_fast(n)` takes logarithmic time

– Orders of Growth

- Quadratic growth

```

def overlap(a, b):
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count

```

- Exponential growth
 - A tree recursion function
- Linear growth
- Logarithmic growth
- Constant growth
 - To find a value in the dictionary

– Order of Growth Notation

- Big Theta and Big O notation:
 - Big O describes the upper bound for the time
 - Big Θ describes the lower bound
 - $O(b^n)$: exponential
 - $O(n^2)$: quadratic
 - $O(n)$: linear
 - $O(\log(n))$: logarithmic
 - $O(1)$: constant

– Space

- Values and frames in active environments consume memory

- Memory that is used for other values and frames can be recycled
- Active environments:
 - Environments for any function calls currently being evaluated
 - Parent environments of functions named in active environments

```
def count_frames(f):
    def counted(n):
        counted.open_count += 1
        if counted.open_count > counted.max_count:
            counted.max_count = counted.open_count
        result = f(n)
        counted.open_count -= 1
        return result
    counted.open_count = 0
    counted.max_count = 0
    return counted
```

Lec 24: Decomposition

– Modular Design

Separation of concerns: isolate different parts of a program that address different concerns. A modular component can be developed and tested independently.

– Example: Restaurants

```
def search(query, ranking=lambda r: -r.stars):
    results = [r for r in all_restaurants if query in r.name]
    return sort(results, key=ranking)

def reviewed_both(r, s):
    return len([x for x in r.reviewers if x in s.reviewers])

class Restaurant:
    all = []
    def __init__(self, name, stars, reviewers):
        self.name, self.stars = name, stars
        self.reviewers = reviewers
        Restaurant.all.append(self)

    def similar(self, k, similarity=reviewed_both):
        """Return the K most similar restaurants to itself, using similarity function for comparison.
        others = list(Restaurant.all)
        others.remove(self)
        return sorted(others, key=lambda r: similarity(self, r))[:k]

    def __repr__(self):
        return '<' + self.name + '>'
```

```

import json

reviewers_for_restaurant = {}
for line in open('reviews.json'):
    r = json.loads(line)
    biz = r['business_id']
    if biz not in reviewers_for_restaurant:
        reviewers_for_restaurant[biz] = [r['user_id']]
    else:
        reviewers_for_restaurant[biz].append([r['user_id']])

for line in open('restaurants.json'):
    r = json.loads(line)
    reviewers = reviewers_for_restaurant[r['business_id']]
    Restaurant(r['name'], r['stars'], reviewers)

results = search('Thai')
for r in results:
    print(r, "shares reviewers with", r.similar(3))

```

– Set Intersections

Linear-Time Intersection of Sorted Lists: Given two sorted lists with no repeats, return the number of elements that appear in both.

```

def fast_overlap(s, t):
    """Return the overlap between sorted S and sorted T."""
    i, j, count = 0, 0, 0
    while i < len(s) and j < len(t):
        if s[i] == t[j]:
            count, i, j = count + 1, i + 1, j + 1
        elif s[i] < t[j]:
            i = i + 1
        else:
            j = j + 1
    return count

def reviewed_both(r, s):
    return fast_overlap(r.reviewers, s.reviewers)

```

Lec 25: Data Examples

– List

- addition & slicing create new lists containing existing elements



Lists in Environment Diagrams

More complicated examples

– Object

Instance attributes are found before class attributes

– Example: Iterables & Iterators

1. What are the indices of all elements in a list `s` that have the smallest absolute value? e.g. `[-4, -3, -2, 3, 2, 4] -> [2, 4]`

```
# Solved by myself
[i for i in range(len(s)) if abs(s[i]) == min([abs(x) for x in s])]
# Using map function
[i for i in range(len(s)) if abs(s[i]) == min(map(abs, s))]
# Using filter function
list(filter(lambda i: abs(s[i]) == min(map(abs, s)), range(len(s))))
```

2. What's the largest sum of two adjacent elements in a list `s`? (Assume `len(s) > 1`) e.g. `[-4, -3, -2, 3, 2, 4] -> 6`

```
# Solved by myself
max([s[i] + s[i + 1] for i in range(len(s) - 1)])
# Using zip function
max([a + b for a, b in zip(s[:-1], s[1:])])
```

3. Create a dictionary mapping each digit `d` to the lists of elements in `s` that end with `d` e.g. `[5, 8, 13, 21, 34, 55, 89] -> {1: [21], 3: [13], 4: [34], 5: [5, 55], ...}`

```
# Solved by myself
{x: [t for t in s if t % 10 == x] for x in range(10) if x in [m % 10 for m in s]}
# Using any function
{x: [t for t in s if t % 10 == x] for x in range(10) if any([m % 10 == x for m in s])}
```

4. Does every element equal some other element in `s`? e.g. `[-4, -3, -2, 3, 2, 4] -> False`

```
# Solved by myself
all([s[i] in s[i + 1:] for i in range(len(s) - 1)])
# Another way
min([sum([1 for y in s if y == x]) for x in s]) > 1
# Another way
min([s.count(x) for x in s]) > 1
```

– Example: Linked Lists

1. Is a linked list `s` ordered from least to greatest by absolute value (or a key function)?

```
def ordered(s, key=abs):
    if s is Link.empty or s.rest is Link.empty:
        return True
    elif key(s.first) > key(s.rest.first):
        return False
    else:
        return ordered(Link(s.rest.first, s.rest.rest), key)
```

2. Create a sorted Link containing all the elements of both sorted Links `s` & `t`

```
def merge(s, t):
    if s is Link.empty:
        return t
    elif t is Link.empty:
        return s
    elif s.first <= t.first:
        return Link(s.first, merge(s.rest, t))
    else:
        return Link(t.first, merge(s, t.rest))
```

3. Do the same thing, but never call `Link`

```
def merge(s, t):
    if s is Link.empty:
        return t
    elif t is Link.empty:
        return s
    elif s.first <= t.first:
        s.rest = merge(s.rest, t)
        return s
    else:
        t.rest = merge(s, t.rest)
        return t
```

Lec 26: Scheme

– Scheme

- Scheme is a dialect of Lisp
- Lisp is a beloved programming language, the beauty of which lies in its simplicity
- Scheme programs consists of expressions, which can be:
 - Primitive expressions: `2` , `3.3` , `true` , `+` , `quotient` , ...
 - Combinations: `(quotient 10 2)` , `(not true)`

- Numbers are self-evaluating; symbols are bound to values.
- Call expressions include an operator and 0 or more operands in parenthesis

```
> (quotient 10 2)
; Quotient is a Scheme's built-in integer division procedure
5
> (quotient (+ 8 7) 5)
3
; Combinations can span multiple lines
> (+ (* 3
      (+ (* 2 4)
          (+ 3 5)))
      (+ (- 10 7)
          6))
> (* 1 2 3 4)
24
> (number? 3)
#t
> (number? +)
#f
> (integer? 2.2)
#f
```

– Scheme Interpreters

It will be implemented in the 4th Project. (Somehow shocked!)

– Special Forms

A combination that is not a call expression is a special form:

- If expression (if <predicate> <consequent> <alternatives>)
- And and or (and <e1> <e2> ... <en>) (or <e1> <e2> ... <en>)
- Binding symbols: (define <symbol> <expression>)
- New procedures: (define (<symbol> <formal parameters>) <body>)

```
> (define pi 3.14)
> (* pi 2)
6.28
> (define (abs x)
  (if (< x 0)
      (-x)
      x))
> (abs -3)
3
```

```

> (define (square x) (* x x))
square
> (define (average x y) (/ (+ x y) 2))
> (define (sqrt x)
  (define (update guess)
    (if (= (square guess) x)
        guess
        (update (average guess (/ x guess))))))
  (update 1))

```

– Lambda Expressions

- Lambda expressions evaluate to anonymous procedures

```

> (lambda (<formal-parameters>) <body>)
; Two equivalent expressions:
> (define (plus4 x) (+ x 4))
> (define plus4 (lambda (x) (+ x 4)))

```

- An operator can be a call expression too:

```

> ((lambda (x y z) (+ x y (square z))) 1 2 3)
12

```

– More Special Forms

- `cond` : The `cond` special form that behaves like `if-elif-else` statements in Python (short for `condition`)

```

(cond(> x 10) (print 'big))
      (> x 5) (print 'medium))
      (else   (print 'small))

```

; Or:

```

(print
 (cond ((> x 10) 'big)
       ((> x 5)  'medium)
       (else     'small)))

```

- `begin` : The `begin` special form combines multiple expressions into one expression

```

if x > 10:
    print('big')
    print('guy')
else:

```

```
print('small')
print('fry')
```

```
(cond ((> x 10) (begin (print 'big) (print 'guy)))
      (else      (begin (print 'small) (print 'fry))))
```

- `let` : The `let` special form binds symbols to values temporarily; just for one expression

```
a = 3
b = 2 + 2
c = math.sqrt(a * a + b * b)
# a and b are still bound down here
```

```
(define c (let ((a 3)
                (b (+ 2 2)))
            (sqrt (+ (* a a) (* b b)))))
; a and b are not bound down here
```

– Sierpinski's Triangle

- Built-in drawing functions in the CS 61A Scheme Interpreter

```
> (fd 100) # forward
> (rt 90) # right turn
> (fd 40)
> (bk 100) # backward
> (lt 90)
> (define (twice fn) (fn) (fn))
> (twice line)
; GUI is generated
```

```
(define (line) (fd 50))
(define (twice fn) (fn) (fn))
```

```
(define (repeat k fn)
  (fn)
  (if (> k 1) (repeat (-k 1) fn)))
```

```
(define (tri fn)
  (repeat 3 (lambda() (fn) (lt 120)))))
```

```
(define (sier d k)
  (tri (lambda () (if (= d 1) (fd k) (leg d k)))))
```

```
(define (leg d k)
```

```
(sier (-d 1) (/ k 2))  
(penup) (fd k) (pendown))
```

Lec 27 Scheme Lists

– Lists

In the late 1950s, computer scientists used confusing names

- `cons` : Two-argument procedure that creates a linked list
- `car` : Procedure that returns the first element of a list
- `cdr` : Procedure that returns the rest of a list
- `nil` : The empty list Scheme lists are written in parentheses with elements separated by spaces

```
>>> Linked(1, Linked(2, Linked.empty))
```

```
> (cons 1 (cons 2 nil))  
(1 2)  
> (define x (cons 1 (cons 2 nil)))  
> x  
(1, 2)  
> (car x)  
1  
> (cdr x)  
(2)  
> (list? x)  
#t  
> (list? nil)  
#f  
> (null? nil)  
#t  
  
> (list 1 2 3 4)  
(1 2 3 4)  
> (cdr (list 1 2 3 4))  
(2 3 4)
```

– Symbolic Programming

Symbols normally refer to values! To refer to symbols:

```
> (define a 1)  
> (define b 2)
```

```
> (list a b)
(1 2)
```

Quotation is used to refer to symbols directly in Lisp

```
> (list 'a 'b)
(a b)
> (list (quote a) (quote b))
(a b)
```

– List Processing

(append s t) : list the element of s and t; append can be called on more than two lists

```
> (define s (cons 1 (cons 2 nil)))
> s
(1 2)
> (append s s)
(1 2 1 2)
> (append s s s s)
(1 2 1 2 1 2 1 2)
> (list s s s s)
((1 2) (1 2) (1 2) (1 2))
```

(map f s) : call a procedure f on each element of a list s and list the results

```
> (map even? s)
(#f #t)
> (map (lambda (x) (* 2 x)) s)
(2 4)
```

(filter f s) : call a procedure f on each element of a list s and list the elements for which a true value is the result

```
> (filter even? '(5 6 7 8 9))
(6 8)
```

(apply f s) : call a procedure f with the elements of a list as its arguments

```
> (apply + '(1 2 3 4))
10
```

– Example: Even Subsets

```
> (even-subsets '(3 4 5 7))
((5 7) (4 5 7) (4) (3 7) (3 5) (3 4 7) (3 4 5))
; Sums of these subsets are even
```

A recursive approach: The even subsets of s include...

- all the subsets of the rest of s
- the first element of s followed by an (even/odd) subset of the rest
- just the first element of s if it is even

```
(define (even-subsets s)
  (if (null? s) nil
      (append (even-subsets (cdr s))
              (map (lambda (t) (cons (car s) t))
                   (if (even? (car s))
                       (even-subsets (cdr s))
                       (odd-subsets (cdr s))))
              (if (even? (car s)) (list (list (car s))) nil))))

(define (odd-subsets s)
  (if (null? s) nil
      (append (odd-subsets (cdr s))
              (map (lambda (t) (cons (car s) t))
                   (if (odd? (car s))
                       (even-subsets (cdr s))
                       (odd-subsets (cdr s))))
              (if (odd? (car s)) (list (list (car s))) nil))))
```

Or another approach using `filter` procedure:

```
(define (nonempty-subsets s)
  (if (null? s)
      nil
      (let ((rest (nonempty-subsets (cdr s))))
        (append rest
                (map (lambda (t) (cons (car s) t)) rest)
                (list (list (car s)))))))

(define (even-subsets s)
  (filter (lambda (s) (even? (apply + s))) (nonempty-subsets s)))
```

Lec 28: Calculators

– Exceptions

– Raise

- Python exceptions are raised with a raise statement.
- `raise <exception>`
- `<expression>` must evaluate to a subclass of `BaseException` or an instance of one
- Exceptions are constructed like any other object. E.g., `TypeError('Bad argument')`
- `TypeError`, `NameError`, `KeyError`, `RecursionError`

```
def double(x):
    if isinstance(x, str):
        raise TypeError('double takes only numbers')
    return 2 * x
```

– Try Statements

- Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

Execution rule: The `<try suite>` is executed first. If, during the course of executing the `<try suite>`, an exception is raised that is not handled otherwise, and if the class of the exception inherits from `<exception class>`, then the `<exception suite>` is executed, with `<name>` bound to the exception.

```
def invert(x):
    result = 1 / x
    print('Never printed if x is 0')
    return result
```

```
def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)
```

```
>>> b = invert_safe(0)
>>> b
'division by zero'
```

– Example: Reduce

```

def divide_all(n, ds):
    try:
        return reduce(truediv, ds, n)
    except ZeroDivisionError:
        return float('inf')

def reduce(f, s, initial):
    """Combine elements of s pairwise using f, starting from initial
    f - a two-argument function
    s - a sequence of values that can be the second argument
    initial - a valule that can be the first argument

    >>> reduce(mul, [2, 4, 8], 1)
    64
    # ((1 * 2) * 4) * 8
    """
    for x in s:
        initial = f(initial, x)
    return initial

# OR:
if not s:
    return initial
else:
    first, rest = s[0], s[1:]
    return reduce(f, rest, f(initial, first))

```

– Programming Languages

- Machine languages: statements are interpreted by the hardware itself
- High-level languages: statements & expressions are interpreted by another program or compiled (translated) into another language
 - Provide means of abstraction such as naming, function definition, and objects

```

>>> def square(x):
...     return x * x
...
>>> from dis import dis # short for disassemble
>>> dis(square)
1           0 RESUME                0

2           2 LOAD_FAST              0 (x)
           4 LOAD_FAST              0 (x)
           6 BINARY_OP              5 (*)
          10 RETURN_VALUE

```


- **Metalinguistic Abstraction:** A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain
 - Type of application: **Erlang** was designed for concurrent programs. It is used to implement chat servers with many simultaneous connections
 - Problem domain: The MediaWiki mark-up language was designed for generating static web pages. It is used to create Wikipedia pages
- A programming language has: syntax(the legal statements and expressions in the language), semantics(execution and evaluation rules).
- To create a new programming language, you either need a:
 - Specification: A document describe the precise syntax and semantics of the language
 - Canonical Implementation: An interpreter or compiler for the language

– Parsing

- Reading scheme lists: A scheme list is written as elements in parenthesis: (<e0> <e1> ... <en>)
- The task of parsing a language involves coercing a string representation of an expression to the expression itself

```
class Pair:
    def __init__(self, first, rest):
        self.first = first
        self.rest = rest

    def __repr__(self):
        return 'Pair({0} {1})'.format(...)

    def __str__(self):
        ...

class nil:
    ...
```

- A parser takes text and returns an expression
- Text → Lexical analysis → Tokens → Syntactic analysis → Expression

– Scheme–Syntax Calculator

Expressions are represented as Scheme Lists (Pair instances) that encode tree structures

Calculator semantics:

- Primitive: A number evaluates to itself

- Call: A call expression evaluates to its argument values combined by an operator
 - + : sum of the arguments
 - * : product of the arguments
 - - : if one argument, negate it; if more than one, subtract the rest from the first
 - / : if one argument, invert it; if more than one, divide the rest from the first

– Evaluation

– The Eval Function

The eval function computes the value of an expression, which is always a number. It is a generic function that dispatches on the type of the expression (primitive or call).

```
def calc_eval(exp):
    if type(exp) in (int, float):
        # Primitive: A number evaluates to itself
        return exp
    elif isinstance(exp, Pair):
        # Call: Its argument values combined by an operator
        arguments = exp.second.map(calc_eval) # Get the argument values evaluated recursively
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

– Applying Built-in Operators

The apply function applies some operation to a (scheme) list of argument values. In calculator, all operations are named by built-in operators: +, -, *, /.

```
def calc_apply(operator, args):
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        ...
    ...
    else:
        raise TypeError
```

– Interactive Interpreter

Read-Eval-Print Loop The user interface for many programming languages is an interactive interpreter.

1. Print a prompt
2. **Read** text input from the user

3. Parse the text input into an expression
 4. **Evaluate** the expression
 5. If any errors occur, report those errors, otherwise
 6. **Print** the value of the expression and repeat
- Exceptions are raised within lexical analysis, eval, and apply:
 - Lexical analysis: The token 2.3.4 raises `ValueError("invalid numeral")`
 - Syntactic analysis: An extra `)` raises `SyntaxError("unexpected token")`
 - Eval: An empty combination raises `TypeError("() is not a number or call expression")`
 - Apply: No arguments to `-` raises `TypeError("- requires at least 1 argument")`
 - Handling Exceptions: An interactive interpreter prints information about each error

Lec 29: Interpreters

– Special Forms

The `scheme_eval` function choose behavior based on expression form:

- Symbols are looked up in the current environment
- Self-evaluating expressions are returned as values
- All other legal expressions are represented as Scheme lists, called combinations Special forms are identified with the first list element `if` , `lambda` , `define`

– Quotation

The quote special form evaluates to the quoted expression, which is not evaluated (`quote <expression>`) \rightarrow `<expression>` e.g. (`quote (+ 1 2)`) \rightarrow `(+ 1 2)` The `<expression>` itself is the value of the expression `'<expression>` is shorthand for (`quote <expression>`) The `scheme_read` parser (which just reads expression, even in multiple lines, without evaluating it) converts shorthand to a combination.

– Logical Forms

Logical forms may only evaluate some sub-expressions

1. If expression (`if <predicate> <consequent> <alternative>`)
2. And and or (`and <e1> ... <en>`) , (`or <e1> ... <en>`)
3. Cond expr'n (`cond (<p1> <e1>) ... (<pn> <en>) (else <e>)`)

The value of an if expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: `<consequent>` or `<alternative>` .

- Evaluate that sub-expression in place of the whole expression.

– Lambda Expressions

Lambda expressions evaluate to user-defined procedures: `(lambda (<formal parameters>) <body>)`

```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals # A list of symbols
        self.body = body # A scheme expression
        self.env = env # A Frame instance
```


Frames and Environments

A frame represents an environment by having a parent frame. Frame are Python instances with methods `lookup` and `define`

– Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope). Lexical scope: The parent of a frame is the environment in which a procedure was defined Dynamic scope: The parent of a frame is the environment in which a procedure was called

– Interpreting Scheme

The structure of an Interpreter  The structure of an Interpreter The apply function would call eval, and the eval function would call apply.

– Define Expressions

`define` binds a symbol to a value in the first frame of the current environment `(define <name> <expression>)` Procedure definition is shorthand of `define` with a lambda expression `(define (<name> <formal parameters>) <body>)` `(define <name> (lambda (<formal parameters>) <body>))`

Lec 30: Tail Calls

– Tail Recursions

- Functional Programming All functions are pure functions. No re-assignment and no mutable data types. Name-value bindings are permanent.

Advantages of functional programming

- The value of an expression is independent of the order in which sub-expressions are evaluated.
- Sub-expressions can safely be evaluated in parallel or on demand (lazily).
- **Referential transparency:** The value of an expression does not change when we substitute one of its sub-expression with the value of that sub-expression.

But... no `for` / `while` statements! Can we make basic iteration efficient? Yes! With Tail Recursion!

Recursion would use, for example, $O(n)$ space (creating several new frames when executed), while iteration might only use constant space ($O(1)$)

– Tail Calls

A procedure call that has not yet returned is *active*. Some procedure calls are *tail calls*. A scheme interpreter should support an *unbounded (as many as we want) number* of active tail calls using only a constant amount of space. A tail call is a call expression in a tail context:

- The last body sub-expression in a lambda expression
- Sub-expressions 2 & 3 in a tail context if expression (consequent and alternative part) Less importantly,
- All non-predicate sub-expressions in a tail context `cond`
- The last sub-expression in a tail context `and` `or` `or`
- The last sub-expression in a tail context `begin`

Example: Length of a List

- Not Tail Recursion

```
(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s)))))
```

- Tail Recursion

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n))))
  (length-iter s 0))
```

– Tail Recursion Examples

– Map and Reduce

– General Computing Machine

Programs specify the logic of a computational device Interpreters are General Computing Machine

Lec 31: Programs as Data

– Programs as Data

A scheme expression is a scheme list

```
scm> (list 'quotient 10 2)
(quotient 10 2)
scm> (eval (list 'quotient 10 2))
5
```

– Generating Code

- Quasiquotation:
 - ```(a b)` => (a b)`
 - Parts of a quasiquotated expression can be unquoted with comma

```
scm> (define b 4)
scm> `(a ,(+ b 1))
(a 5)
```

- While statements

Lec 32: Macros

– Expressions

```
(define (square-expr term) `(* ,term ,term))
`(+ ,(square-expr 'a) ,(square-expr 'b))
```

– Macros

- Macros perform code transformations
- A macro is an operation performed on the source code of a program before evaluation

```
(define-macro (twice expr)
(list 'begin expr expr)) ; the expr doesn't get evaluated
```

```
scm> (twice(print(2))) ; (begin (print 2) (print 2))
2
2
```

```
(define-macro (check expr) (list 'if expr 'passed (list 'quote (list 'failed: expr))))
```

– For Macro

```
(define-macro (for sym vals expr) (list 'map (list 'lambda (list sym) expr) vals))
scm> (for x '(2, 3, 4, 5) (* 5 5))
(4 9 16 25)
```

– Trace

- Tracing recursive calls

Lec 33: SQL

– Databases

- Database Management Systems (DBSS)
 - A table is a collection of records. A column has a name and a type. A row has a value for each column.
- The structured query language (SQL) is perhaps the most widely used programming language.
- SQL is a declarative language.
- In *declarative* languages such as SQL & Prolog:
 - A program is a description of the desired result
 - The interpreter figures out how to generate the result
- In *imperative* languages such as Python & Scheme:
 - A program is a description of computational processes
 - The interpreter carries out execution/evaluation rules

```
create table cities as
select 38 as latitude, 122 as longitude, "Berkeley" as name union
```

```
select 42, 71, "Cambridge" union
select 45, 93, "Minneapolis";
```

latitude	longitude	name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

```
select "west coast" as region, name from cities where longitude >= 115 union
select "other", name from cities where longitude < 115;
```

region	name
other	Cambridge
other	Minneapolis
west coast	Berkeley

– Structured Query Language (SQL)

- The SQL language is an ANSI and ISO standard, but DBMS’s implement custom variants
 - A `select` statement creates a new table, either from scratch or by projecting a file
 - A `create table` statement gives a global name to a table
 - Lots of other statements exist: `analyze` , `delete` , `explain` , `insert` , `update` , `replace` , etc.
 - Most of the important action is in the `select` statement
 - The code for executing select statements fits on a single sheet of paper (next lecture)
- Selecting Value Laterals
 - A `select` statement always includes a comma–seperated list of column descriptions
 - A column description is an expression, optionally followed by `as` and a column name

```
select [expression] as [name], [expression] as [name], ...;
```

- Selecting literals creates a one–row table
- The `union` of two `select` statements is a table containing the rows of **both** of their results

```
select "abraham" as parent, "barack" as child;
```


parent	child
abraham	barack

```
sqlite> create table parents as
...> select "abraham" as parent, "barack" as child union
...> select "abraham"          , "clinton"          union
...> select "delano"           , "herbert"           union
...> select "fillmore"         , "abraham"         union
...> select "fillmore"         , "delano"         union
...> select "fillmore"         , "grover"         union
...> select "eisenhower"      , "fillmore";
```

```
sqlite> select * from parents;
```

```
+-----+-----+
| parent | child |
+-----+-----+
| abraham | barack |
| abraham | clinton |
| delano  | herbert |
| eisenhower | fillmore |
| fillmore | abraham |
| fillmore | delano  |
| fillmore | grover  |
+-----+-----+
```

SQL is often used as an interactive language The result of a select statement is displayed to the user, but not stored. A `create table` statement gives the result a name:

```
create table [name] as [select statement];
```

– Projecting Tables

A `select` statement can specify an input table using a `from` clause A subset of the rows of the input table can be selected using a `where` clause An ordering over the remaining rows can be declared using an `order by` clause

```
select [columns] from [table] where [condition] order by [order];
```

```
sqlite> select child from parents where parent = "abraham";
```

```
+-----+
| child |
+-----+
| barack |
| clinton |
+-----+
```

– Arithmetic

In a select expression, column names evaluate to row values Arithmetic expressions can combine row values and constants

```
sqlite> select chair, single + 2 * couple as total from lift;
+-----+-----+
| chair | total |
+-----+-----+
| 101   | 6     |
| 102   | 6     |
| 103   | 6     |
+-----+-----+
```

Lec 34: Tables

– Joining Tables

Two tables A & B are joined by a comma to yield all combos of a row from A & a row from B.

```
create table dogs as
select "abraham" as name, "long" as fur union
select "barack", "short" union
select "clinton", "long" union
select "delano", "short" union
select "eisenhower", "curly" union
select "fillmore", "curly" union
select "grover", "short" union
select "herbert", "curly";

create table parents as
select "abraham" as parent, "barack" as child union
select "abraham", "clinton" union
select "delano", "herbert" union
select "fillmore", "abraham" union
select "fillmore", "delano" union
select "fillmore", "grover" union
select "eisenhower", "fillmore";
```

```
sqlite> select * from parents, dogs
...>          where child = name;
```

```
+-----+-----+-----+-----+
| parent | child | name  | fur  |
+-----+-----+-----+-----+
| abraham | barack | barack | short |
| abraham | clinton | clinton | long  |
```

delano	herbert	herbert	curly
eisenhower	fillmore	fillmore	curly
fillmore	abraham	abraham	long
fillmore	delano	delano	short
fillmore	grover	grover	short

```
sqlite> select * from parents, dogs
...>         where child = name and fur = "curly";
```

parent	child	name	fur
eisenhower	fillmore	fillmore	curly
delano	herbert	herbert	curly

– Aliases and Dot Expressions

- Two tables may share a column name; dot expressions and aliases disambiguate column values

```
select [columns] from [table] where [condition] order by [order];
```

[table] is a comma-separated list of table names with optional aliases

For example, to select all the siblings

```
sqlite> select a.child as first, b.child as second
...>   from parents as a, parents as b
...>   where a.parent = b.parent and a.child < b.child;
```

first	second
barack	clinton
abraham	delano
abraham	grover
delano	grover

Multiple tables can be joined to yield all combinations of rows from each. For example, to select all grandparents with the same fur as their grandchildren

```
sqlite> create table grandparents as
...>   select a.parent as granddog, b.child as granpup
...>   from parents as a, parents as b
...>   -- Joining two tables here
```

```

...>         where b.parent = a.child;
sqlite> select granddog from grandparents, dogs as c, dogs as d
           -- Joining three tables here
...>         where granddog = c.name and
...>         granpup = d.name and
...>         c.fur = d.fur;
+-----+
| granddog |
+-----+
| fillmore |
+-----+

```

– Numerical Expressions

- Expressions can contain function calls and arithmetic operators
 - Combine values: +, -, *, /, %, and, or
 - Transform values: abs, round, not, -
 - Compare values: <, <=, >, >=, <>, !=, =
- No double equal since there is no notion of assignment in SQL

```

create table cities as
  select 42 as latitude, 122 as longitude, "Berkeley" as name union
  select 44, 93, "Minneapolis" union
  select 33, 117, "Los Angeles" union
  select 26, 80, "Miami" union
  select 90, 0, "North Pole";

```

```

create table cold as
  select name from cities where latitude >= 43;

```

```

create table distances as
  select a.name as first, b.name as second,
         60 * (b.latitude - a.latitude) as distance
from cities as a, cities as b;

```

```

sqlite> select second from distances
...>         where first = "Minneapolis"
...>         order by distance;

```

```

+-----+
| second |
+-----+
| Minneapolis |
| Berkeley    |
| Los Angeles |
| Miami       |
| North Pole  |
+-----+

```

– String Expressions

- String values can be combined to form longer strings

```
sqlite> select "Hello, " || "world";
+-----+
| "Hello, " || "world" |
+-----+
| Hello, world          |
+-----+
```

- Basic string manipulation is built into SQL, but differs from Python

```
sqlite> create table phrase as select "hello, world" as s;
sqlite> select substr(s, 4, 2) || substr(s, instr(s, " ")+1, 1) from phrase;
low
```

- Strings can be used to represent structured values, but doing so is rarely a good idea

```
sqlite> create table lists as select "one" as car, "two,three,four" as cdr;
sqlite> select substr(cdr, 1, instr(cdr, ",")-1) as cadr from lists;
two
```

```
sqlite> create table nouns as
...>   select "dog" as phrase union
...>   select "cat" union
...>   select "bird";
```

```
sqlite> select subject.phrase || " chased " || object.phrase
...>   from nouns as subject, nouns as object
...>   where subject.phrase <> object.phrase;
```

```
+-----+
| subject.phrase || " chased " || object.phrase |
+-----+
| bird chased cat                               |
| bird chased dog                               |
| cat chased bird                               |
| cat chased dog                               |
| dog chased bird                               |
| dog chased cat                               |
+-----+
```

Lec 35: Aggregation

– Aggregation

So far, all SQL expressions have referred to the values in a single row at a time. An aggregate function in the columns clause computes a value from a group of rows

```
create table animals as
  select "dog" as kind, 4 as legs, 20 as weight union
  select "cat", 4, 10 union
  select "ferret", 4, 10 union
  select "parrot", 2, 6 union
  select "penguin", 2, 10 union
  select "t-rex", 2, 12000;
```

```
sqlite> select max(legs) from animals;
```

```
+-----+
| max(legs) |
+-----+
| 4         |
+-----+
```

```
sqlite> select sum(weight) from animals;
```

```
+-----+
| sum(weight) |
+-----+
| 12056       |
+-----+
```

```
sqlite> select max(legs - weight) + 5 from animals;
```

```
+-----+
| max(legs - weight) + 5 |
+-----+
| 1                       |
+-----+
```

```
sqlite> select max(legs), min(weight) from animals;
```

```
+-----+-----+
| max(legs) | min(weight) |
+-----+-----+
| 4         | 6          |
+-----+-----+
```

```
sqlite> select min(legs), max(weight) from animals
```

```
...> where kind <> "t-rex";
```

```
+-----+-----+
| min(legs) | max(weight) |
+-----+-----+
| 2         | 20         |
+-----+-----+
```

```
sqlite> select avg(legs) from animals;
```

```
+-----+
| avg(legs) |
+-----+
| 3.0       |
+-----+
```

```
sqlite> select count(*) from animals;
```

```
+-----+
| count(*) |
+-----+
```

```
| 6          |
+-----+
sqlite> select count(distinct legs) from animals;
+-----+
| count(distinct legs) |
+-----+
| 2                    |
+-----+
```

An aggregate function also selects a row in the table

```
sqlite> select max(weight), kind from animals;
+-----+-----+
| max(weight) | kind |
+-----+-----+
| 12000       | t-rex |
+-----+-----+
```

-- Some arbitrary value:

```
sqlite> select avg(weight), kind from animals;
+-----+-----+
| avg(weight) | kind |
+-----+-----+
| 2009.33333333333 | cat |
+-----+-----+
```

```
sqlite> select max(legs), kind from animals;
+-----+-----+
| max(legs) | kind |
+-----+-----+
| 4         | cat |
+-----+-----+
```

-- Although cats, dogs and ferrets all have 4 legs

– Groups

Rows in a table can be grouped, and aggregation is performed on each group

```
select [columns] from [table] group by [expression] having [expression];
```

```
sqlite> select legs, max(weight) from animals group by legs;
+-----+-----+
| legs | max(weight) |
+-----+-----+
| 2    | 12000       |
+-----+-----+
```

```
| 4 | 20 |
+-----+
```

1. **group by legs :**

- This clause groups the rows in the `animals` table based on the value of the `legs` column. So, all rows with the same number of legs are put into the same group. In this case, there are two groups: one for animals with 2 legs and one for animals with 4 legs.

2. **max(weight) :**

- The `max` is an aggregate function. For each group created by the `group by` clause, it finds the maximum value in the `weight` column. For the group of animals with 2 legs (which includes the parrot, penguin, and t-rex), the maximum weight is 12000 (from the t-rex). For the group of animals with 4 legs (dog, cat, ferret), the maximum weight is 20 (from the dog).

3. The **select** statement:

- It selects two columns: the `legs` value (which represents the group) and the maximum `weight` value within each group. So, the result set shows the number of legs and the maximum weight for each distinct number of legs in the `animals` table.

```
sqlite> select legs, weight from animals group by legs, weight;
```

```
+-----+-----+
| legs | weight |
+-----+-----+
| 2    | 6      |
| 2    | 10     |
| 2    | 12000  |
| 4    | 10     |
| 4    | 20     |
+-----+-----+
```

• **group by legs, weight :**

- This `group by` clause groups the rows in the `animals` table based on both the `legs` and `weight` columns simultaneously. Each unique combination of the values in the `legs` and `weight` columns forms a separate group.
- For example, if there were multiple rows with the same number of legs and the same weight, they would be grouped together. In the `animals` table as initially defined:
 - The row with `legs = 4` and `weight = 20` (the dog) forms one group.
 - The rows with `legs = 4` and `weight = 10` (the cat and ferret) are grouped together because they have the same combination of `legs` and `weight` values.
 - The rows with `legs = 2` and `weight = 6` (the parrot), `legs = 2` and `weight = 10` (the penguin), and `legs = 2` and `weight = 12000` (the t – rex) each form their own groups since their `legs – weight` combinations are unique among themselves.

- A `having` clause filters the set of groups that are aggregated


```
sqlite> select weight/legs, count(*) from animals group by weight/legs having count(*) > 1
...> ;
```

weight/legs	count(*)
2	2
5	2

Lec 36: Databases

– Create Table and Drop Table

```
CREATE TABLE numbers (n UNIQUE, note DEFAULT "No comment")
```

```
DROP TABLE IF EXISTS [table-name]
```

– Modifying Tables

```
INSERT INTO [table-name](column-name) VALUES (values);
INSERT INTO [table-name](column-name) select-stmt;
```

```
sqlite> create table primes(n, prime);
sqlite> drop table if exists primes;
sqlite> select * from primes;
Parse error: no such table: primes
sqlite> create table primes(n UNIQUE, prime DEFAULT 1);
sqlite> select * from primes;
sqlite> INSERT INTO primes VALUES (2, 1), (3, 1);
sqlite> select * from primes;
```

n	prime
2	1
3	1

```
sqlite> INSERT INTO primes(n) VALUES (4), (5), (6), (7);
sqlite> select * from primes;
```

n	prime
2	1
3	1
4	1

5	1
6	1
7	1

```

+----+-----+
sqlite> INSERT INTO primes(n) SELECT n+6 FROM primes;
sqlite> select * from primes;
+----+-----+
| n | prime |
+----+-----+
| 2 | 1     |
| 3 | 1     |
| 4 | 1     |
| 5 | 1     |
| 6 | 1     |
| 7 | 1     |
| 8 | 1     |
| 9 | 1     |
| 10| 1     |
| 11| 1     |
| 12| 1     |
| 13| 1     |
+----+-----+

```

Update sets all entries in certain columns to new values, just for some subset of rows.

UPDATE qualified-table-name **SET** column-name = expr **WHERE** expr;

```

sqlite> UPDATE primes SET prime=0 WHERE n>2 AND n%2=0;
sqlite> select * from primes;
+----+-----+
| n | prime |
+----+-----+
| 2 | 1     |
| 3 | 1     |
| 4 | 0     |
| 5 | 1     |
| 6 | 0     |
| 7 | 1     |
| 8 | 0     |
| 9 | 1     |
| 10| 0     |
| 11| 1     |
| 12| 0     |
| 13| 1     |
| 14| 0     |
| 15| 1     |
| 16| 0     |
| 17| 1     |
| 18| 0     |
| 19| 1     |

```

20	0
21	1
22	0
23	1
24	0
25	1

```
sqlite> UPDATE primes SET prime=0 WHERE n>3 AND n%3=0;
sqlite> UPDATE primes SET prime=0 WHERE n>5 AND n%5=0;
```

Delete removes some or all rows from a table.

```
DELETE FROM qualified-table-name WHERE expr;
```

```
sqlite> DELETE FROM primes WHERE prime=0;
sqlite> select * from primes;
```

n	prime
2	1
3	1
5	1
7	1
11	1
13	1
17	1
19	1
23	1

– Python and SQL

```
import sqlite3

db = sqlite3.Connection("n.db")
db.execute("CREATE TABLE nums AS SELECT 2 UNION SELECT 3;")
db.execute("INSERT INTO nums VALUES (?), (?), (?);", range(4, 7))
print(db.execute("SELECT * FROM nums;").fetchall())

db.commit() # To commit the changes, much like Git
```

– Database Connections

Lec 27: Final Examples

– Trees

Trees are everywhere!

– Tree Processing

```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    def get_count(node, max_ancestor):
        if node.label > max_ancestor:
            return 1 + sum[get_count(branch, node.label) for branch in node.branches]
        else:
            return sum[get_count(branch, max_ancestor) for branch in node.branches]
    return get_count(t, t.label - 1)
```

– Recursive Accumulation

```
def bigs(t):
    n = [0]
    def get_count(node, max_ancestor):
        if node.label > max_ancestor:
            n[0] += 1
        for b in node.branches:
            get_count(b, max(node.label, max_ancestor))
    get_count(t, t.label - 1)
    return n[0]
```

– Designing Functions

1. From problem analysis to data definitions
2. Signature, Purpose statement, Header
3. Functional examples
4. Function template
5. Function definition
6. Testing

– Applying the Design Process

```
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."""
    result = []
    def process(t):
        """Find smallest label in t & maybe add t to result."""
        if t.is_leaf():
```

```

        return t.label
    else:
        smallest = min([process(branch) for branch in t.branches])
        if t.label < smallest:
            result.append(t)
        return min(smallest, t.label)
process(t)
return result

```

Congratulations, you have completed CS61a!

Extra Python: File I/O

Extra Python: Website Pachong

Extra Python: ???

Extra Python 1: Bitwise Operations

In Python, bitwise operations are operations that directly manipulate the binary bits of integers.

Bitwise AND Operator (&)

The bitwise AND operator compares the corresponding binary bits of two integers. If both corresponding binary bits are 1, the result of that bit is 1; otherwise, it is 0.

Example Code:

```

a = 5 # Binary representation: 0101
b = 3 # Binary representation: 0011
result = a & b # Binary operation: 0101 & 0011 = 0001
print(result) # Output: 1

```

Use-case:

- **Determine odd or even:** You can use `num & 1` to determine whether an integer `num` is odd or even. If the result is 1, `num` is odd; if the result is 0, `num` is even.

```

num = 7
if num & 1:
    print(f"{num} is odd")
else:
    print(f"{num} is even")

```

Bitwise OR Operator (|)

The bitwise OR operator compares the corresponding binary bits of two integers. If at least one of the two corresponding binary bits is 1, the result of that bit is 1; otherwise, it is 0.

Example Code:

```
a = 5 # Binary representation: 0101
b = 3 # Binary representation: 0011
result = a | b # Binary operation: 0101 | 0011 = 0111
print(result) # Output: 7
```

Use-case:

- **Set flag bits:** In some systems, binary bits are used to represent different flags. You can use the bitwise OR operator to set these flag bits. For example, if there are two flags `FLAG_A` and `FLAG_B`, you can combine them.

```
FLAG_A = 1 # Binary: 0001
FLAG_B = 2 # Binary: 0010
combined_flags = FLAG_A | FLAG_B # Binary: 0011
print(combined_flags) # Output: 3
```

Bitwise XOR Operator (^)

The bitwise XOR operator compares the corresponding binary bits of two integers. If the two corresponding binary bits are different, the result of that bit is 1; otherwise, it is 0.

Example Code:

```
a = 5 # Binary representation: 0101
b = 3 # Binary representation: 0011
result = a ^ b # Binary operation: 0101 ^ 0011 = 0110
print(result) # Output: 6
```

Use-case:

- **Swap two variables:** You can swap the values of two integers without using an extra variable.

```
x = 5
y = 3
x = x ^ y
y = x ^ y
```

```
x = x ^ y
print(f"x = {x}, y = {y}") # Output: x = 3, y = 5
```

Bitwise NOT Operator (~)

The bitwise NOT operator inverts all the binary bits of an integer, i.e., 0 becomes 1 and 1 becomes 0. In Python, the result of the bitwise NOT operation on a number n is $-n - 1$.

Example Code:

```
a = 5 # Binary representation: 0101
result = ~a # Binary operation: ~0101 = -0110 (result is -6 in Python)
print(result) # Output: -6
```

Use-case:

- **Generate masks:** In some cases, you can use the bitwise NOT operator to generate masks for masking certain bits.

Left Shift Operator (<<)

The left-shift operator shifts the binary bits of an integer to the left by a specified number of positions. The empty positions on the right are filled with 0s. Shifting left by one position is equivalent to multiplying the original number by 2.

Example Code:

```
a = 5 # Binary representation: 0101
result = a << 2 # Binary operation: 0101 shifted left 2 positions becomes 010100
print(result) # Output: 20
```

Use-case:

- **Fast multiplication:** When you need to multiply a number by a power of 2, you can use the left-shift operator, which is faster than the multiplication operation.

Right Shift Operator (>>)

The right-shift operator shifts the binary bits of an integer to the right by a specified number of positions. The empty positions on the left are filled according to the sign bit of the original number (0 for positive numbers and 1 for negative numbers). Shifting right by one position is equivalent to dividing the original number by 2 and rounding down.

Example Code:

```
a = 20 # Binary representation: 010100
result = a >> 2 # Binary operation: 010100 shifted right 2 positions becomes 0101
print(result) # Output: 5
```

Use-case:

- **Fast division:** When you need to divide a number by a power of 2, you can use the right-shift operator, which is faster than the division operation.

Summary

Bitwise operations are very useful for handling binary data, optimizing algorithm performance, and dealing with flag bits. However, it should be noted that the results of bitwise operations may produce unexpected results due to the binary representation of integers and the handling of sign bits. Use them with caution.