CS 61*A*

Structure and Interpretation of
Computer Programs

Discussion Fa23

CS 61A
Fall 2023

Control, Environment Diagrams

Discussion 1: August 30, 2023

Control Structures

Control structures direct the flow of a program using logical statements. For example, conditionals (`if-elif-else`) allow a program to skip sections of code, and iteration (`while`), allows a program to repeat a section.

Conditional Statements

Conditional statements let programs execute different lines of code depending on certain conditions. Let's review the `if-elif-else` syntax:

- The `elif` and `else` clauses are optional, and you can have any number of `elif` clauses.
- A **conditional expression** is an expression that evaluates to either a truthy value (`True`, a non-zero integer, etc.) or a falsy value (`False`, `0`, `None`, `""`, `[]`, etc.).
- Only the first `if`/`elif` expression that evaluates to a **truthy** value will have its corresponding indented suite be executed.
- If none of the conditional expressions evaluate to a true value, then the `else` suite is executed. There can only be one `else` clause in a conditional statement.

Here's the general form:

```
if <conditional expression>:
    <suite of statements>
elif <conditional expression>:
    <suite of statements>
else:
    <suite of statements>
```

Boolean Operators

Python also includes the **boolean operators** `and`, `or`, and `not`. These operators are used to combine and manipulate boolean values.

- `not` returns the opposite boolean value of the following expression, and will always return either `True` or `False`.
- `and` evaluates expressions in order and stops evaluating (short-circuits) once it reaches the first falsy value, and then returns it. If all values evaluate to a truthy value, the last value is returned.
- `or` evalutes expressions in order and short-circuits at the first truthy value and returns it. If all values evaluate to a falsy value, the last value is returned.

For example:

```
>>> not None
True
>>> not True
False
>>> -1 and 0 and 1
0
>>> False or 9999 or 1/0
9999
```

Q1: Case Conundrum

In this question, we will explore the difference between `if` and `elif`.

What is the result of evaluating the following code?

```
def special_case():
    x = 10
    if x > 0:
        x += 2
    elif x < 13:
        x += 3
    elif x % 2 == 1:
        x += 4
    return x

special_case()
```

What is the result of evaluating this piece of code?

```
def just_in_case():
    x = 10
    if x > 0:
        x += 2
    if x < 13:
        x += 3
    if x % 2 == 1:
        x += 4
    return x

just_in_case()
```

How about this piece of code?

```
def case_in_point():
    x = 10
    if x > 0:
        return x + 2
    if x < 13:
        return x + 3
    if x % 2 == 1:
        return x + 4
    return x

case_in_point()
```

Which of these code snippets result in the same output, and why? Based on your findings, when do you think using a series of `if` statements has the same effect as using both `if` and `elif` cases?

Q2: Jacket Weather?

Alfonso will only wear a jacket outside if it is below 60 degrees or it is raining.

Write a function that takes in the current temperature and a boolean value telling if it is raining. This function should return `True` if Alfonso will wear a jacket and `False` otherwise.

Try solving this problem using an `if` statement.

> **Note:** Since we'll either return `True` or `False` based on a single condition, whose truthiness value will also be either `True` or `False`. Knowing this, try to write this function using a single line.

```
def wears_jacket_with_if(temp, raining):
    """
    >>> wears_jacket_with_if(90, False)
    False
    >>> wears_jacket_with_if(40, False)
    True
    >>> wears_jacket_with_if(100, True)
    True
    """
    "*** YOUR CODE HERE ***"
```

Q3: Nearest Ten

Write a function that takes in a positive number n and rounds n to the nearest ten.

Solve this problem using an `if` statement.

> **Hint**: `x % 10` will get the units digit of `x`. For example, `1234 % 10` evaluates to `4`.

```
def nearest_ten(n):
    """
    >>> nearest_ten(0)
    0
    >>> nearest_ten(4)
    0
    >>> nearest_ten(5)
    10
    >>> nearest_ten(61)
    60
    >>> nearest_ten(2023)
    2020
    """
    "*** YOUR CODE HERE ***"
```

While Loops

To repeat the same statements multiple times in a program, we can use iteration. In Python, one way we can do this is with a **while loop**.

```
while <conditional clause>:
    <statements body>
```

As long as `<conditional clause>` evaluates to a true value, `<statements body>` will continue to be executed. The conditional clause gets evaluated each time the body finishes executing.

Q4: Square So Slow

What is the result of evaluating the following code?

```
def square(x):
    print("here!")
    return x * x

def so_slow(num):
    x = num
    while x > 0:
        x = x + 1
    return x / 0

square(so_slow(5))
```

Hint: What happens to `x` over time?

Q5: Fizzbuzz

Implement the classic *Fizz Buzz* sequence. The `fizzbuzz` function takes a positive integer `n` and prints out a *single line* for each integer from 1 to `n`. For each `i`:

- If `i` is divisible by both 3 and 5, print `fizzbuzz`.
- If `i` is divisible by 3 (but not 5), print `fizz`.
- If `i` is divisible by 5 (but not 3), print `buzz`.
- Otherwise, print the number `i`.

Try to make your implementation of `fizzbuzz` concise.

```
def fizzbuzz(n):
    """
    >>> result = fizzbuzz(16)
    1
    2
    fizz
    4
    buzz
    fizz
    7
    8
    fizz
    buzz
    11
    fizz
    13
    14
    fizzbuzz
    16
    >>> print(result)
    None
    """
    "*** YOUR CODE HERE ***"
```

Q6: Is Prime?

Write a function that returns `True` if a positive integer `n` is a prime number and `False` otherwise.

A prime number n is a number that is not divisible by any numbers other than 1 and n itself. For example, 13 is prime, since it is only divisible by 1 and 13, but 14 is not, since it is divisible by 1, 2, 7, and 14.

Hint: Use the `%` operator: `x % y` returns the remainder of `x` when divided by `y`.

```
def is_prime(n):
    """
    >>> is_prime(10)
    False
    >>> is_prime(7)
    True
    >>> is_prime(1) # one is not a prime number!!
    False
    """
    "*** YOUR CODE HERE ***"
```

Q7: Unique Digits

Write a function that returns the number of unique digits in a positive integer.

> **Hints:** You can use `//` and `%` to separate a positive integer into its one's digit and the rest of its digits.
>
> You may find it helpful to first define a function `has_digit(n, k)`, which determines whether a number `n` has digit `k`.

```
def unique_digits(n):
    """Return the number of unique digits in positive integer n.

    >>> unique_digits(8675309) # All are unique
    7
    >>> unique_digits(13173131) # 1, 3, and 7
    3
    >>> unique_digits(101) # 0 and 1
    2
    """
    "*** YOUR CODE HERE ***"









def has_digit(n, k):
    """Returns whether k is a digit in n.

    >>> has_digit(10, 1)
    True
    >>> has_digit(12, 7)
    False
    """
    assert k >= 0 and k < 10
    "*** YOUR CODE HERE ***"
```

Environment Diagrams

An **environment diagram** is a model we use to keep track of all the variables that have been defined and the values they are bound to. We will be using this tool throughout the course to understand complex programs involving several different assignments and function calls.

One key idea in environment diagrams is the **frame**. A frame helps us keep track of what variables have been defined in the current execution environment, and what values they hold. The frame we start off with when executing a program from scratch is what we call the **Global frame**. Later, we'll get into how new frames are created and how they may depend on their parent frame.

Here's a short program and its corresponding diagram (only visible on the online version of this worksheet):

See the web version of this resource for the environment diagram.

Remember that programs are mainly just a set of statements or instructions— so drawing diagrams that represent these programs also involves following sets of instructions! Let's dive in…

Assignment Statements

Assignment statements, such as `x = 3`, define variables in programs. To execute one in an environment diagram, record the variable name and the value:

1. Evaluate the expression on the right side of the `=` sign.
2. Write the variable name and the expression's value in the current frame.

Q8: Assignment Diagram

Use these rules to draw an environment diagram for the assignment statements below:

```
x = 11 % 4
y = x
x **= 2
```

def Statements

A `def` statement creates ("defines") a function object and binds it to a name. To diagram `def` statements, record the function name and bind the function object to the name. It's also important to write the **parent frame** of the function, which is where the function is defined.

A very important note: Assignments for `def` statements use pointers to functions, which can have different behavior than primitive assignments (such as variables bound to numbers).

1. Draw the function object to the right-hand-side of the frames, denoting the intrinsic name of the function, its parameters, and the parent frame (e.g. `func square(x) [parent = Global]`.
2. Write the function name in the current frame and draw an arrow from the name to the function object.

Q9: def Diagram

Use these rules for defining functions and the rules for assignment statements to draw a diagram for the code below.

```
def double(x):
    return x * 2

def triple(x):
    return x * 3

hat = double
double = triple
```

CS 61A Fall 2023

Environment Diagrams, Higher-Order Functions

Discussion 2: September 6, 2023

Last week, you learned about the **environment diagram**, which is a visual model for keeping track of the state of a computer program. We have already gone over rules for variable assignment and `def` statements. This week, we will go over several more rules for constructing environment diagrams.

Scope

Scope refers to the idea that names have meaning only within a specific context. For example, the following program produces an error because the scope of the variable `b` is **local**—limited to the call to `f()`. Outside of that function call, `b` has no meaning, so the line `print(b)` errors.

```
def f():
    b = 2
f()
print(b)
```

On the other hand, some variables are defined with a broader scope. The following code prints `2`. `b` is accessible within the call to `f()` because `b` is defined in the **global scope**, outside of any function.

```
b = 2
def f():
    print(b)
f()
```

Scope can also be relevant in disambiguating variables. In the following code, there are two variables named `b`: one is defined in the global scope and the other is defined in the local scope of the call to `f(3)`. The line `print(b)` prints `3` instead of `2` because the local variable takes precedence over the global variable.

```
b = 2
def f(b):
    print(b)
f(3)
```

Call Expressions

In Python, local variables are created whenever a function is called. To formalize this idea of scope, we have rules for how call expressions, such as `square(2)`, should be treated in environment diagrams. Specifically, we create a new frame in our diagram to keep track of local variables:

1. Evaluate the operator, which should evaluate to a function.
2. Evaluate the operands from left to right.
3. Draw a new frame, labelling it with the following:

- A unique index (`f1`, `f2`, `f3`, …).
- The **intrinsic name** of the function, which is the name of the function object itself as it was named in its `def` statement. For example, if the function object is `func square(x) [parent=Global]`, the intrinsic name is `square`.
- The parent frame (e.g. [`parent=Global`]).

4. In the new frame, bind the **formal parameters** to the argument values obtained in step 2 (e.g. bind `x` to 2).
5. Evaluate the body of the function in this new frame until a return value is obtained. Write down the return value in the frame.

If you never hit a `return` statement in your function, it implicitly returns `None`. In that case, the "Return value" box should contain `None`.

> **Note:** Since we do not know how built-in functions like `min(...)` or imported functions like `add(...)` are implemented, we do not draw a new frame when we call them because we would not be able to fill it out accurately.

When we want to look up the value of a variable, we follow these rules:

1. Search for the variable in the current frame.
2. If the variable is not bound in the current frame, look up to the parent frame. If the variable is not bound in that frame, look up to its parent frame, etc.
3. If you look up all the way to the global frame and still cannot find the variable, throw a `NameError`.

Q1: Call Diagram

Let's put it all together! Draw an environment diagram for the following code.

```
def double(x):
    return x * 2

hmmm = double
wow = double(3)
hmmm(wow)
```

Q2: Nested Calls Diagrams

Draw the environment diagram that results from executing the code below.

```
def f(x):
    return x

def g(x, y):
    if x(y):
        return not y
    return y

x = 3
x = g(f, x)
f = g(f, 0)
```

Lambda Expressions

A lambda expression evaluates to a function, called a lambda function. For example, `lambda y: x + y` is a lambda expression, and can be read as "a function that takes in one parameter `y` and returns `x + y`."

A lambda expression by itself evaluates to a function but does not bind it to a name. Also note that the return expression of this function is not evaluated until the lambda function is called. This is similar to how defining a new function using a def statement does not execute the function's body until it is later called.

```
>>> what = lambda x : x + 5
>>> what
<function <lambda> at 0xf3f490>
```

Unlike `def` statements, lambda expressions can be used as an operator or an operand to a call expression. This is because they are simply one-line expressions that evaluate to functions. In the example below, `(lambda y: y + 5)` is the operator and `4` is the operand.

```
>>> (lambda y: y + 5)(4)
9
>>> (lambda f, x: f(x))(lambda y: y + 1, 10)
11
```

Lambda functions **do not** have an intrinsic name, so we use the Greek letter lambda as a placeholder wherever we would usually use a function's intrinsic name.

Q3: Lambda the Environment Diagram

Draw the environment diagram for the following code and predict what Python will output.

```
a = lambda x: x * 2 + 1
def b(b, x):
    return b(x + a(x))
x = 3
x = b(a, x)
```

Higher Order Functions

A **higher order function** (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both. For example, the function `compose` below takes in two functions as arguments and returns a function that is the composition of the two arguments.

```
def composer(func1, func2):
    """Returns a function f, such that f(x) = func1(func2(x))."""
    def f(x):
        return func1(func2(x))
    return f
```

HOFs are powerful abstraction tools because they allow us to express certain general patterns (functions) as named concepts in our programs.

Environment diagrams can model more complex programs that utilize higher order functions.

```
def delete_num(x):
    """Returns a lambda function that takes in y and deletes x digits from y."""
    return lambda y: y // (10 ** x) # Note that ** means exponent (^) in Python

delete_two = delete_num(2)
delete_two(614)
```

See the web version of this resource for the environment diagram.

The parent of any function (including lambdas) is always the frame in which the function is defined. It is useful to include the parent in environment diagrams in order to find variables that are not defined in the current frame. In the previous example, when we call `delete_two` (which is really the lambda function), we need to know what `x` is in order to compute `y // (10 ** x)`. Since `x` is not in the frame `f2`, we look at the frame's parent, which is `f1`. There, we find `x` is bound to 2.

As illustrated above, higher order functions that return a function have their return value represented with a pointer to the function object.

Q4: Make Adder

Draw the environment diagram for the following code:

```
n = 9
def make_adder(n):
    return lambda k: k + n
add_ten = make_adder(n+1)
result = add_ten(n)
```

There are 3 frames total (including the Global frame). In addition, consider the following questions:

1. In the Global frame, the name `add_ten` points to a function object. What is the intrinsic name of that function object, and what frame is its parent?
2. What name is frame `f2` labeled with (`add_ten` or lambda)? Which frame is the parent of `f2`?
3. What value is the variable `result` bound to in the Global frame?

Q5: Make Keeper

Implement `make_keeper`, which takes a positive integer `n` and returns a function `f` that takes as its argument another one-argument function `cond`. When `f` is called on `cond`, it prints out the integers from 1 to `n` (including `n`) for which `cond` returns a true value when called on each of those integers. Each integer is printed on a separate line.

```
def make_keeper(n):
    """Returns a function that takes one parameter cond and prints
    out all integers 1..i..n where calling cond(i) returns True.

    >>> def is_even(x): # Even numbers have remainder 0 when divided by 2.
    ...     return x % 2 == 0
    >>> make_keeper(5)(is_even)
    2
    4
    >>> make_keeper(5)(lambda x: True)
    1
    2
    3
    4
    5
    >>> make_keeper(5)(lambda x: False)  # Nothing is printed
    """
    "*** YOUR CODE HERE ***"
```

Currying

One important application of HOFs is converting a function that takes multiple arguments into a chain of functions that each take a single argument. This is known as **currying**. For example, here is a curried version of the `pow` function:

```
>>> def curried_pow(x):
...     def h(y):
...         return pow(x, y)
...     return h

>>> curried_pow(2)(3)
8
>>> pow(2, 3) == curried_pow(2)(3)
True
```

This is useful if, say, you need to calculate a lot of powers of 2. Using the normal `pow` function, you would have to put in `2` as the first argument for every function call:

```
>>> pow(2, 3)
8
>>> pow(2, 4)
16
>>> pow(2, 10)
1024
```

With `curried_pow`, however, you can create a one-argument function specialized for taking powers of 2 one time, and then keep using that function for taking powers of 2:

```
>>> pow_2 = curried_pow(2)
>>> pow_2(3)
8
>>> pow_2(4)
16
>>> pow_2(10)
1024
```

This way, you don't have to put `2` in as an argument for every call! If instead you wanted to take powers of 3, you could quickly make a similar function specialized in taking powers of 3 using `curried_pow(3)`.

Q6: Currying

Write a function `curry` that will curry any two argument function.

```
def curry(func):
    """
    Returns a Curried version of a two-argument function FUNC.
    >>> from operator import add, mul, mod
    >>> curried_add = curry(add)
    >>> add_three = curried_add(3)
    >>> add_three(5)
    8
    >>> curried_mul = curry(mul)
    >>> mul_5 = curried_mul(5)
    >>> mul_5(42)
    210
    >>> curry(mod)(123)(10)
    3
    """
    "*** YOUR CODE HERE ***"
```

First, try implementing `curry` with `def` statements. Then attempt to implement `curry` in a single line using lambda expressions.

HOFs and Lambdas

Q7: Make Your Own Lambdas

For the following problem, first read the doctests for functions `f1`, `f2`, `f3`, and `f4`. Then, implement the functions to conform to the doctests without causing any errors. **Be sure to use lambdas in your function definition instead of nested `def` statements.** Each function should have a one line solution.

```
def f1():
    """
    >>> f1()
    3
    """
    "*** YOUR CODE HERE ***"



def f2():
    """
    >>> f2()()
    3
    """
    "*** YOUR CODE HERE ***"



def f3():
    """
    >>> f3()(3)
    3
    """
    "*** YOUR CODE HERE ***"



def f4():
    """
    >>> f4()()(3)()
    3
    """
    "*** YOUR CODE HERE ***"
```

Extra Practice

This question is particularly challenging, so it's recommended if you're feeling confident on the previous questions or are studying for the exam.

Q8: Match Maker

Implement `match_k`, which takes in an integer `k` and returns a function that takes in a variable `x` and returns `True` if all the digits in `x` that are `k` apart are the same.

For example, `match_k(2)` returns a one argument function that takes in `x` and checks if digits that are 2 away in `x` are the same.

`match_k(2)(1010)` has the value of `x = 1010` and digits 1, 0, 1, 0 going from left to right. `1 == 1` and `0 == 0`, so the `match_k(2)(1010)` results in `True`.

`match_k(2)(2010)` has the value of `x = 2010` and digits 2, 0, 1, 0 going from left to right. `2 != 1` and `0 == 0`, so the `match_k(2)(2010)` results in `False`.

Important: You may not use strings or indexing for this problem. You do not have to use all the lines; one staff solution does not use the line directly above the while loop.

Hint: Floor dividing by powers of 10 gets rid of the rightmost digits.

```
def match_k(k):
    """Returns a function that checks if digits k apart match.

    >>> match_k(2)(1010)
    True
    >>> match_k(2)(2010)
    False
    >>> match_k(1)(1010)
    False
    >>> match_k(1)(1)
    True
    >>> match_k(1)(2111111111111111)
    False
    >>> match_k(3)(123123)
    True
    >>> match_k(2)(123123)
    False
    """
    def ____________________________:
        ____________________________
        while ____________________________:
            if ____________________________:
                return ____________________________
            ____________________________
        ____________________________
    ____________________________
```

CS 61A
Fall 2023

Recursion

Discussion 3: September 13, 2023

Recursion

A *recursive* function is a function that is defined in terms of itself.

Consider this recursive `factorial` function:

```
def factorial(n):
    """Return the factorial of N, a positive integer."""
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Inside of the body of `factorial`, we are able to call `factorial` itself, since the function body is not evaluated until the function is called.

When `n` is 1, we can directly return the factorial of 1, which is 1. This is known as the *base case* of this recursive function, which is the case where we can return from the function call directly without having to first recurse (i.e. call `factorial`) and then returning. The base case is what prevents `factorial` from recursing infinitely.

Since we know that our base case `factorial(1)` will return, we can compute `factorial(2)` in terms of `factorial(1)`, then `factorial(3)` in terms of `factorial(2)`, and so on.

There are three main steps in a recursive definition:

1. **Base case.** You can think of the base case as the case of the simplest function input, or as the stopping condition for the recursion.

   In our example, `factorial(1)` is our base case for the `factorial` function.

2. **Recursive call on a smaller problem.** You can think of this step as calling the function on a smaller problem that our current problem depends on. We assume that a recursive call on this smaller problem will give us the expected result; we call this idea the "recursive leap of faith".

   In our example, `factorial(n)` depends on the smaller problem of `factorial(n-1)`.

3. **Solve the larger problem.** In step 2, we found the result of a smaller problem. We want to now use that result to figure out what the result of our current problem should be, which is what we want to return from our current function call.

   In our example, we can compute `factorial(n)` by multiplying the result of our smaller problem `factorial(n-1)` (which represents `(n-1)!`) by `n` (the reasoning being that `n! = n * (n-1)!`).

Q1: Warm Up: Recursive Multiplication

These exercises are meant to help refresh your memory of the topics covered in lecture.

Write a function that takes two numbers `m` and `n` and returns their product. Assume `m` and `n` are positive integers. Use **recursion**, not `mul` or `*`.

Hint: `5 * 3 = 5 + (5 * 2) = 5 + 5 + (5 * 1)`.

For the base case, what is the simplest possible input for `multiply`?

For the recursive case, what does calling `multiply(m - 1, n)` do? What does calling `multiply(m, n - 1)` do? Do we prefer one over the other?

```
def multiply(m, n):
    """Takes two positive integers and returns their product using recursion.
    >>> multiply(5, 3)
    15
    """
    "*** YOUR CODE HERE ***"
```

Q2: Recursion Environment Diagram

Draw an environment diagram for the following code:

Note: If you can't move elements around, make sure you're logged in!

```
def rec(x, y):
    if y > 0:
        return x * rec(x, y - 1)
    return 1

rec(3, 2)
```

Note: This problem is meant to help you understand what really goes on when we make the "recursive leap of faith". However, when approaching or debugging recursive functions, you should avoid visualizing them in this way for large or complicated inputs, since the large number of frames can be quite unwieldy and confusing. Instead, think in terms of the three steps: base case, recursive call, and solving the full problem.

Q3: Find the Bug

Find the bug in this recursive function.

```
def skip_mul(n):
    """Return the product of n * (n - 2) * (n - 4) * ...

    >>> skip_mul(5) # 5 * 3 * 1
    15
    >>> skip_mul(8) # 8 * 6 * 4 * 2
    384
    """
    if n == 2:
        return 2
    else:
        return n * skip_mul(n - 2)
```

Q4: Is Prime

Write a function `is_prime` that takes a single argument `n` and returns `True` if `n` is a prime number and `False` otherwise. Assume `n > 1`. We implemented this in Discussion 1 iteratively, now time to do it recursively!

> *Hint*: You will need a helper function! Remember helper functions are nested functions that are useful if you need to keep track of more variables than the given parameters, or if you need to change the value of the input.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    "*** YOUR CODE HERE ***"
```

Q5: Recursive Hailstone

Recall the `hailstone` function from Homework 1. First, pick a positive integer `n` as the start. If `n` is even, divide it by 2. If `n` is odd, multiply it by 3 and add 1. Repeat this process until `n` is 1. Write a recursive version of `hailstone` that prints out the values of the sequence and returns the number of steps.

> *Hint*: When taking the recursive leap of faith, consider both the return value and side effect of this function.

```
def hailstone(n):
    """Print out the hailstone sequence starting at n, and return the number of elements
    in the sequence.
    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    >>> b = hailstone(1)
    1
    >>> b
    1
    """
    "*** YOUR CODE HERE ***"
```

Q6: Merge Numbers

Write a procedure `merge(n1, n2)`, which takes numbers with digits in decreasing order and returns a single number with all of the digits of the two in decreasing order. Any number merged with 0 will be that number (treat 0 as having no digits). Use recursion.

> *Hint*: If you can figure out which number has the smallest digit out of both, then we know that the resulting number will have that smallest digit, followed by the merge of the two numbers with the smallest digit removed.

```
def merge(n1, n2):
    """Merges two numbers by digit in decreasing order.
    >>> merge(31, 42)
    4321
    >>> merge(21, 0)
    21
    >>> merge (21, 31)
    3211
    """
    "*** YOUR CODE HERE ***"
```

CS 61A
Fall 2023

Tree Recursion

Discussion 4: September 20, 2023

Tree Recursion

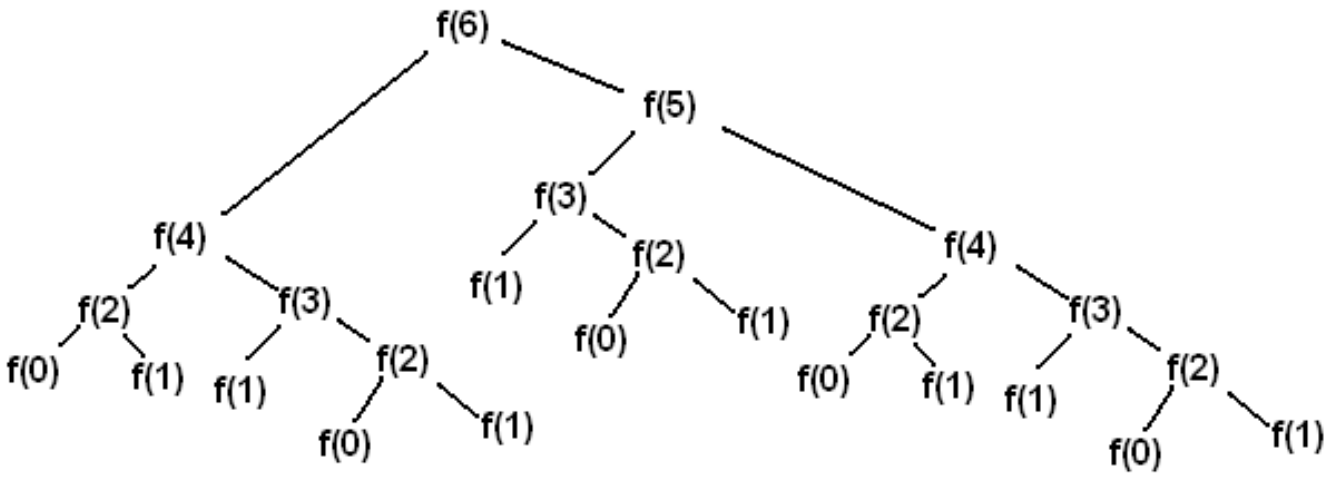
A tree recursive function is a recursive function that makes more than one call to itself, resulting in a tree-like series of calls.

For example, this is the Virahanka-Fibonacci sequence: `0, 1, 1, 2, 3, 5, 8, 13, ...`.

Each term is the sum of the previous two terms. This tree-recursive function calculates the `n`th Virahanka-Fibonacci number.

```
def virfib(n):
    if n == 0 or n == 1:
        return n
    return virfib(n - 1) + virfib(n - 2)
```

Calling `virfib(6)` results in a call structure that resembles an upside-down tree (where `f` is `virfib`):



Virahanka-Fibonacci tree.

Each recursive call `f(i)` makes a call to `f(i-1)` and a call to `f(i-2)`. Whenever we reach an `f(0)` or `f(1)` call, we can directly return `0` or `1` without making more recursive calls. These calls are our base cases.

A base case returns an answer without depending on the results of other calls. Once we reach a base case, we can go back and answer the recursive calls that led to the base case.

As we will see, tree recursion is often effective for problems with branching choices. In these problems, you make a recursive call for each branching choice.

Q1: Count Stair Ways

Imagine that you want to go up a flight of stairs that has `n` steps, where `n` is a positive integer. You can take either one or two steps each time you move. In how many ways can you go up the entire flight of stairs?

You'll write a function `count_stair_ways` to answer this question. Before you write any code, consider:

- How many ways are there to go up a flight of stairs with `n = 1` step? What about `n = 2` steps? Try writing or drawing out some other examples and see if you notice any patterns.
- What is the base case for this question? What is the smallest input?
- What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

Now, fill in the code for `count_stair_ways`:

```
def count_stair_ways(n):
    """Returns the number of ways to climb up a flight of
    n stairs, moving either one step or two steps at a time.
    >>> count_stair_ways(1)
    1
    >>> count_stair_ways(2)
    2
    >>> count_stair_ways(4)
    5
    """
    "*** YOUR CODE HERE ***"
```

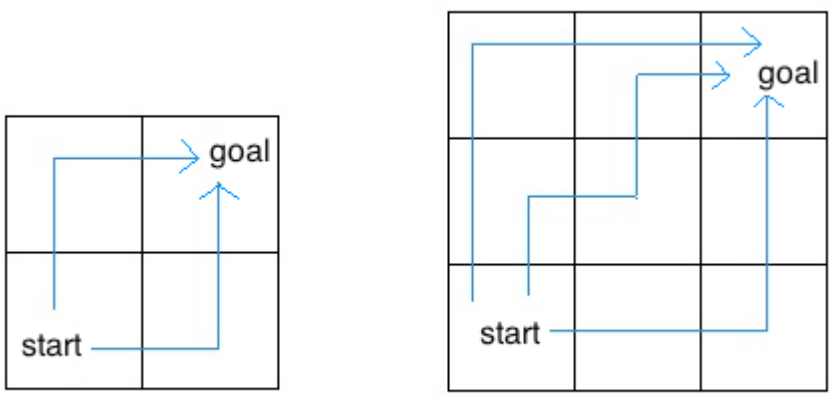
Q2: Count K

Consider a special version of the `count_stair_ways` problem where we can take up to `k` steps at a time. Write a function `count_k` that calculates the number of ways to go up an `n`-step staircase. Assume `n` and `k` are positive integers.

```
def count_k(n, k):
    """Counts the number of paths up a flight of n stairs
    when taking up to k steps at a time.
    >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
    4
    >>> count_k(4, 4)
    8
    >>> count_k(10, 3)
    274
    >>> count_k(300, 1) # Only one step at a time
    1
    """
    "*** YOUR CODE HERE ***"
```

Q3: Insect Combinatorics

An insect is inside an `m` by `n` grid. The insect starts at the bottom-left corner `(1, 1)` and wants to end up at the top-right corner `(m, n)`. The insect can only move up or to the right. Write a function `paths` that takes the height and width of a grid and returns the number of paths the insect can take from the start to the end. (There is a closed-form solution to this problem, but try to answer it with recursion.)



Insect grids.

In the `2` by `2` grid, the insect has two paths from the start to the end. In the `3` by `3` grid, the insect has six paths (only three are shown above).

Hint: What happens if the insect hits the upper or rightmost edge of the grid?

```
def paths(m, n):
    """Return the number of paths from one corner of an
    M by N grid to the opposite corner.

    >>> paths(2, 2)
    2
    >>> paths(5, 7)
    210
    >>> paths(117, 1)
    1
    >>> paths(1, 157)
    1
    """
    "*** YOUR CODE HERE ***"
```

Q4: Max Product

Write a function that takes in a list and returns the maximum product that can be formed using non-consecutive elements of the list. All numbers in the input list are greater than or equal to 1.

```
def max_product(s):
    """Return the maximum product that can be formed using
    non-consecutive elements of s.
    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """
```

Q5: Flatten

Write a function `flatten` that takes a list and returns a "flattened" version of it. The input list may be a "deep list" (a list that contains other lists).

In the following example, `[1, [[2], 3], 4, [5, 6]]` is a deep list because `[[2], 3]` and `[5, 6]` are lists. Note that `[[2], 3]` is itself a deep list.

```
>>> lst = [1, [[2], 3], 4, [5, 6]]
>>> flatten(lst)
[1, 2, 3, 4, 5, 6]
```

Hint: you can check if something in Python is a list with the built-in `type` function. For example:

```
>>> type(3) == list
False
>>> type([1, 2, 3]) == list
True
```

```
def flatten(s):
    """Returns a flattened version of list s.

    >>> flatten([1, 2, 3])
    [1, 2, 3]
    >>> deep = [1, [[2], 3], 4, [5, 6]]
    >>> flatten(deep)
    [1, 2, 3, 4, 5, 6]
    >>> deep                                # input list is unchanged
    [1, [[2], 3], 4, [5, 6]]
    >>> very_deep = [['m', ['i', ['n', ['m', 'e', ['w', 't', ['a'], 't', 'i', 'o'], 'n
    ']], 's']]]
    >>> flatten(very_deep)
    ['m', 'i', 'n', 'm', 'e', 'w', 't', 'a', 't', 'i', 'o', 'n', 's']
    """
    "*** YOUR CODE HERE ***"
```

CS 61A
Fall 2023

Trees, Data Abstraction

Discussion 5: September 27, 2023

Data Abstraction

So far, we have encountered several data types: numbers, strings, booleans, lists, tuples, and dictionaries. But what if we want to represent more complicated things — such as the objects we encounter in real life?

Enter the idea of *data abstraction*, which allows us to build and use objects in code that represent a compound value.

An *abstract data type* consists of two types of functions:

- **Constructors**: functions that build instances of the abstract data type.
- **Selectors**: functions that retrieve information from instances of the ADT.

An integral part of data abstraction is the concept of *abstraction*. Users of the abstract data type have access to an *interface* of defined actions (i.e. constructors and selectors) they can take with the ADT. The user of the ADT does *not* need to know how the constructors and selectors are implemented. The nature of *abstraction* allows whoever uses them to assume that the functions have been written correctly and work as described.

This is called the **abstraction barrier**!

In fact, interacting within an ADT outside of its interface of constructors and selectors is called "violating the abstraction barrier" and is strongly discouraged (even when it doesn't produce an error).

In this way, data abstraction mimics how we think about the world. For example, a car is a complicated machine, but it has a simple interface by which we interact with it: the wheel, the gas pedal, the gear shift. If you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of. It's easy to drive a car because of the suppression of complexity that abstraction provides us.

For the car, the principle of the abstraction barrier also applies. If we want to start the car, we should use the provided interface of turning the ignition key, rather than getting out, popping the hood, and messing around with the engine. While you could likely do that if you had enough knowledge, it is unnecessarily costly and highly dependent on how exactly the car was constructed. What are you going to do if the car is electric?

When you're writing code using ADTs that have been provided for you, make sure that you're never assuming anything about the ADT other than that it can be constructed with its constructor and its information can be accessed by selectors. Your code should never depend on the specific implementation of the ADT under the hood.

Rationals

Q1: Extending Rationals

First, fill in the following code to implement the rational ADT from lecture.

```
from math import gcd

def make_rat(num, den):
    """Creates a rational number, given a numerator and denominator.

    >>> a = make_rat(2, 4)
    >>> numer(a)
    1
    >>> denom(a)
    2
    """
    "*** YOUR CODE HERE ***"




def numer(rat):
    """Extracts the numerator from a rational number."""
    "*** YOUR CODE HERE ***"



def denom(rat):
    """Extracts the denominator from a rational number."""
    "*** YOUR CODE HERE ***"
```

Q2: Divide

Next, we'll be implementing two additional functions to handle operations between rational numbers.

First, implement `div_rat(x,y)`, which returns the result of dividing rational number `x` by rational number `y`.

```
def div_rat(x, y):
    """The quotient of rationals x/y.
    >>> a, b = make_rat(3, 4), make_rat(5, 3)
    >>> c = div_rat(a, b)
    >>> numer(c)
    9
    >>> denom(c)
    20
    """
    "*** YOUR CODE HERE ***"
```

Q3: Less Than

Finally, implement `lt_rat(x, y)`, which returns `True` if and only if rational number `x` is less than rational number `y`.

```
def lt_rat(x, y):
    """Returns True iff x < y as rational numbers; else False.
    >>> a, b = make_rat(6, 7), make_rat(12, 16)
    >>> lt_rat(a, b)
    False
    >>> lt_rat(b, a)
    True
    >>> lt_rat(a, b)
    False
    >>> a, b = make_rat(-6, 7), make_rat(-12, 16)
    >>> lt_rat(a, b)
    True
    >>> lt_rat(b, a)
    False
    >>> lt_rat(a, a)
    False
    """
    "*** YOUR CODE HERE ***"
```

Trees

In computer science, **trees** are recursive data structures that are widely used in various settings and can be implemented in many ways.

In this class, we consider a `tree` an *abstract data type* (ADT). This will act as our own custom type, much like an integer or string!

A tree has a root **label** and a sequence of **branches**, each of which is also a tree. A **leaf** is a tree with no branches.

- The arguments to the constructor `tree` are the label for the root node and an optional list of branches.
- If no branches parameter is provided, the default value `[]` is used.
- The selectors for these are `label` and `branches`.

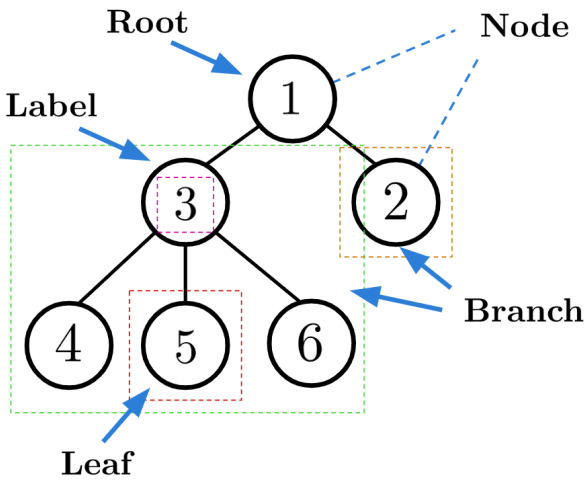
Remember `branches` returns a list of trees and not a tree directly. It's important to distinguish between working with a tree and working with a **sequence of** trees.

We have also provided a convenience predicate function, `is_leaf`, which returns whether a tree is a leaf.

Let's try to create the tree `t` from above:

```
t2 = tree(3,
          [tree(4),
           tree(5),
           tree(6)])

t1 = tree(1,
          [t2,
           tree(2)])
```

The following is an illustration of tree `t1`.



Example Tree

We note that there are many trees in this data structure. For example, `t1` contains the tree `t2`. So we may refer to a tree as a **node** of a larger structure. Thus, `t1` and `t2` are nodes (locations) in the tree `t1`, and `t1` is the **root node** of this tree.

In the tree `t1`, `t2` is a **child** of `t1`, and `t1` is the **parent** of `t2` since `t2` is a branch of `t1`.

A **path** in a tree is a sequence of nodes where tree `i+1` is the **child** of tree `i`. We say the path is from the first node in the sequence to the last node. The **height** of a tree is this the maximum length **path** to a **leaf** in the tree, where a **leaf** is a tree with no branches.

The **depth** of a node `n` in a tree `t` is the length of the path from the root node of `t` to `n`. For example, in the tree `t1`, the depth of node labeled `4` is 2.

Tree ADT Implementation

For your reference, we have provided our implementation of trees as a data abstraction. However, as with any data abstraction, we should only concern ourselves with what our functions do rather than their specific implementation!

```
def tree(label, branches=[]):
    """Construct a tree with the given label value and a list of branches."""
    return [label] + list(branches)

def label(tree):
    """Return the label value of a tree."""
    return tree[0]

def branches(tree):
    """Return the list of branches of the given tree."""
    return tree[1:]

def is_leaf(tree):
    """Returns True if the tree's list of branches is empty, and False otherwise."""
    return not branches(tree)
```

Q4: Tree Abstraction Barrier

Consider a tree `t` constructed by calling `tree(1, [tree(2), tree(4)])`. For each of the following expressions, answer these two questions:

- What does the expression evaluate to?
- Does the expression violate any abstraction barriers? If so, write an equivalent expression that does not violate abstraction barriers.

1. `label(t)`
2. `t[0]`
3. `label(branches(t)[0])`
4. `is_leaf(t[1:][1])`
5. `[label(b) for b in branches(t)]`
6. **Challenge:** `branches(tree(5, [t, tree(3)]))[0][0]`

Q5: Height

Write a function that returns the height of a tree. Recall that the height of a tree is the number of non-root nodes in the longest path from the root to a leaf.

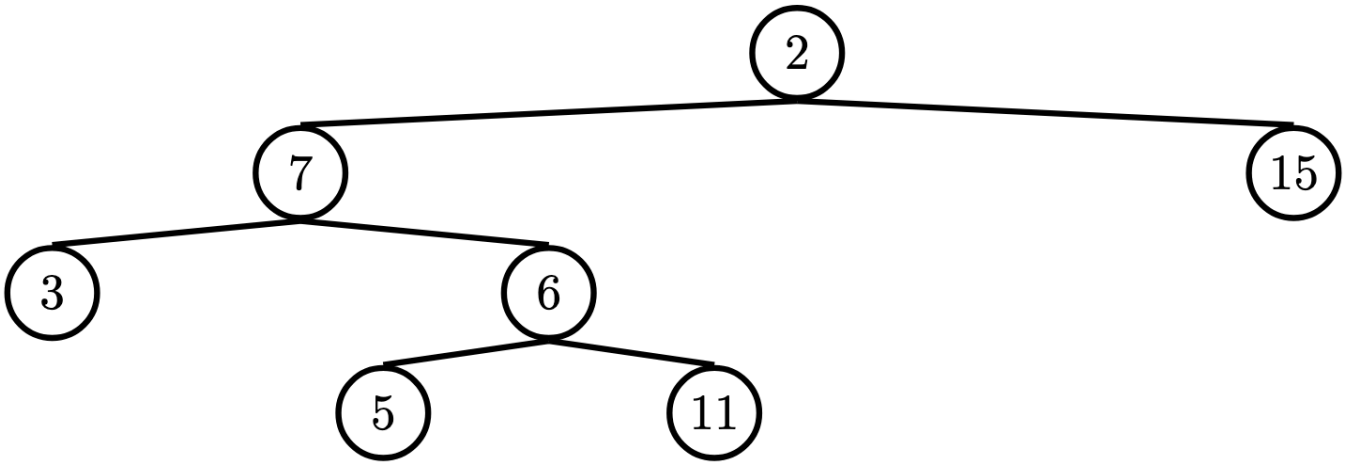
```
def height(t):
    """Return the height of a tree.
    >>> t = tree(3, [tree(5, [tree(1)]), tree(2)])
    >>> height(t)
    2
    >>> t = tree(3, [tree(1), tree(2, [tree(5, [tree(6)]), tree(1)])])
    >>> height(t)
    3
    """
    "*** YOUR CODE HERE ***"
```

Q6: Find Path

Write a function `find_path` that takes in a tree `t` with unique labels and a value `x`. It returns a list containing the labels of the nodes along the path from the root of `t` to the node labeled `x`.

If `x` is not a label in `t`, return `None`. Assume that the labels of `t` are unique.

For the following tree, `find_path(t, 5)` should return `[2, 7, 6, 5]`.



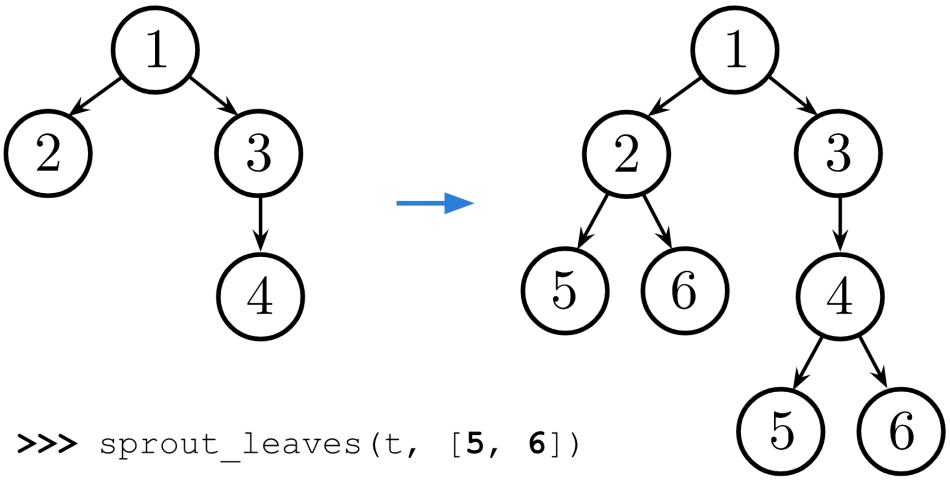
Example Tree

```
def find_path(t, x):
    """
    >>> t = tree(2, [tree(7, [tree(3), tree(6, [tree(5), tree(11)])] ), tree(15)])
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)  # returns None
    """
    if ______________________________:
        return ____________________________
    ____________________________:
        path = ________________________
        if ____________________________:
            return ____________________________
```

Q7: Sprout Leaves

Define a function `sprout_leaves` that takes in a tree, `t`, and a list of leaf labels, `leaves`. It produces a new tree that is identical to `t`, but where each old leaf node has new branches, for each label in `leaves`.

For example, suppose we have the tree `t = tree(1, [tree(2), tree(3, [tree(4)])])`. Calling `sprout_leaves(t, [5, 6])` produces the following tree:




```
>>> sprout_leaves(t, [5, 6])
```

Example Tree

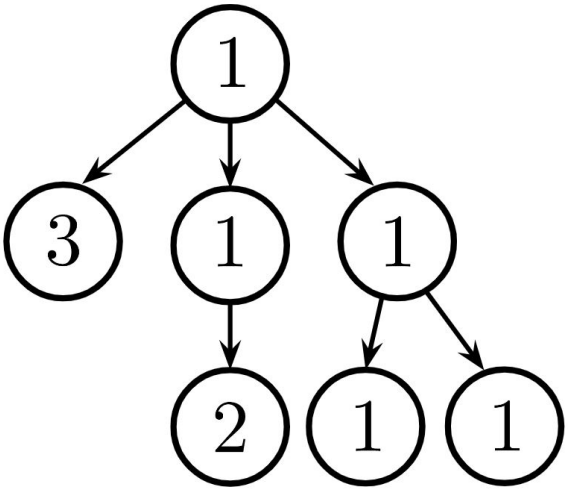
```
def sprout_leaves(t, leaves):
    """Sprout new leaves containing the data in leaves at each leaf in
    the original tree t and return the resulting tree.
    >>> t1 = tree(1, [tree(2), tree(3)])
    >>> print_tree(t1)
    1
      2
      3
    >>> new1 = sprout_leaves(t1, [4, 5])
    >>> print_tree(new1)
    1
      2
        4
        5
      3
        4
        5

    >>> t2 = tree(1, [tree(2, [tree(3)])])
    >>> print_tree(t2)
    1
      2
        3
    >>> new2 = sprout_leaves(t2, [6, 1, 2])
    >>> print_tree(new2)
    1
      2
        3
          6
          1
          2
    """
    "*** YOUR CODE HERE ***"
```

Additional Practice

Q8: Perfectly Balanced

Part A: Implement `sum_tree`, which returns the sum of all the labels in tree `t`.

Part B: Implement `balanced`, which returns whether every branch of `t` has the same total sum and that the branches themselves are also balanced.



Example Tree

- For example, the tree above is balanced because each branch has the same total sum, and each branch is also itself balanced.

  Hint: If we ever need to select a specific branch, we will need to break index into our branches list!

  Challenge: Solve both of these parts with just 1 line of code each.

```
def sum_tree(t):
    """
    Add all elements in a tree.
    >>> t = tree(4, [tree(2, [tree(3)]), tree(6)])
    >>> sum_tree(t)
    15
    """
    "*** YOUR CODE HERE ***"
```

```
def balanced(t):
    """
    Checks if each branch has same sum of all elements and
    if each branch is balanced.
    >>> t = tree(1, [tree(3), tree(1, [tree(2)]), tree(1, [tree(1), tree(1)])])
    >>> balanced(t)
    True
    >>> t = tree(1, [t, tree(1)])
    >>> balanced(t)
    False
    >>> t = tree(1, [tree(4), tree(1, [tree(2), tree(1)]), tree(1, [tree(3)])])
    >>> balanced(t)
    False
    """
    "*** YOUR CODE HERE ***"
```

CS 61A
Fall 2023

Iterators, Generators

Discussion 6: October 4, 2023

Iterators

An iterable is an object whose elements we can go through one at a time. Iterables can be used in for loops and list comprehensions. Iterables can also be converted into lists using the `list` function. Examples of iterables we have seen so far in Python include strings, lists, tuples, and ranges.

```
>>> for x in "cat":
...     print(x)
c
a
t
>>> [x*2 for x in (1, 2, 3)]
[2, 4, 6]
>>> list(range(4))
[0, 1, 2, 3]
```

Note the abstraction present here: given some iterable object, we have a set of defined actions that we can take with it. In this discussion, we will peak below the abstraction barrier and examine how iterables are implemented "under the hood".

In Python, **iterables** are formally implemented as objects that can be passed into the built-in `iter` function to produce an *iterator*. An **iterator** is another type of object that can produce elements one at a time with the `next` function.

- `iter(iterable)`: Returns an iterator over the elements of the given iterable.
- `next(iterator)`: Returns the next element in an iterator, or raises a `StopIteration` exception if there are no elements left.

For example, a list of numbers is an iterable, since `iter` gives us an iterator over the given sequence, which we can navigate using the `next` function:

```
>>> lst = [1, 2, 3]
>>> lst_iter = iter(lst)
>>> lst_iter
<list_iterator object ...>
>>> next(lst)
1
>>> next(lst)
2
>>> next(lst)
3
>>> next(lst)
StopIteration
```

Iterators are very simple. There is only a mechanism to get the next element in the iterator: `next`. There is no way to index into an iterator and there is no way to go backward. Once an iterator has produced an element, there is no way for us to get that element again unless we store it.

Note that iterators themselves are iterables: calling `iter` on an iterator simply returns the same iterator object.

For example, we can see what happens when we use `iter` and `next` with a list:

```
>>> lst = [1, 2, 3]
>>> next(lst)             # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> next(list_iter)       # Calling next on an iterator
1
>>> next(iter(list_iter)) # Calling iter on an iterator returns itself
2
>>> for e in list_iter:   # Exhausts remainder of list_iter
...     print(e)
3
>>> next(list_iter)       # No elements left!
StopIteration
>>> lst                   # Original iterable is unaffected
[1, 2, 3]
```

The `map` and `filter` functions we learned earlier in class return iterator objects.

Q1: WWPD: Iterators

What would Python display?

```
>>> s = "cs61a"
>>> s_iter = iter(s)
>>> next(s_iter)
```

```
>>> next(s_iter)
```

```
>>> list(s_iter)
```

```
>>> s = [[1, 2, 3, 4]]
>>> i = iter(s)
>>> j = iter(next(i))
>>> next(j)
```

```
>>> s.append(5)
>>> next(i)
```

```
>>> next(j)
```

```
>>> list(j)
```

```
>>> next(i)
```

Generators

We can define custom iterators by writing a *generator function*, which returns a special type of iterator called a **generator**.

A generator function looks like a normal Python function, except that it has at least one `yield` statement. When we call a generator function, a ***generator** object* is returned without evaluating the body of the generator function itself. (Note that this is different from ordinary Python functions. While generator functions and normal functions look the same, their evaluation rules are very different!)

When we first call `next` on the returned generator, we will begin evaluating the body of the generator function until an element is yielded or the function otherwise stops (such as if we `return`). The generator remembers where we stopped, and will continue evaluating from that stopping point on the next time we call `next`.

As with other iterators, if there are no more elements to be generated, then calling `next` on the generator will give us a `StopIteration`.

For example, here's a generator function:

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

To create a new generator object, we can call the generator function. Each returned generator object from a function call will separately keep track of where it is in terms of evaluating the body of the function. Like all other iterators, calling `iter` on an existing generator object returns the same generator object.

```
>>> c1, c2 = countdown(2), countdown(2)
>>> c1 is iter(c1)  # a generator is an iterator
True
>>> c1 is c2
False
>>> next(c1)
Beginning countdown!
2
>>> next(c2)
Beginning countdown!
2
```

In a generator function, we can also have a `yield from` statement, which will **yield** each element **from** an iterator or iterable.

```
>>> def gen_list(lst):
...     yield from lst
...
>>> g = gen_list([1, 2])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
StopIteration
```

Since generators are themselves iterators, this means we can use `yield from` to create recursive generators!

```
>>> def rec_countdown(n):
...     if n < 0:
...         print("Blastoff!")
...     else:
...         yield n
...         yield from rec_countdown(n-1)
...
>>> r = rec_countdown(2)
>>> next(r)
2
>>> next(r)
1
>>> next(r)
0
>>> next(r)
Blastoff!
StopIteration
```

Q2: WWPD: Generators

What would Python display? If the command errors, input the specific error.

```
>>> def infinite_generator(n):
...     yield n
...     while True:
...         n += 1
...         yield n
>>> next(infinite_generator)
```

```
>>> gen_obj = infinite_generator(1)
>>> next(gen_obj)
```

```
>>> next(gen_obj)
```

```
>>> list(gen_obj)
```

```
>>> def rev_str(s):
...     for i in range(len(s)):
...         yield from s[i::-1]
>>> hey = rev_str("hey")
>>> next(hey)
```

```
>>> next(hey)
```

```
>>> next(hey)
```

```
>>> list(hey)
```

```
>>> def add_prefix(s, pre):
...     if not pre:
...         return
...     yield pre[0] + s
...     yield from add_prefix(s, pre[1:])
>>> school = add_prefix("schooler", ["pre", "middle", "high"])
>>> next(school)
```

```
>>> list(map(lambda x: x[:-2], school))
```

Q3: Filter-Iter

Implement a generator function called `filter_iter(iterable, f)` that only yields elements of `iterable` for which `f` returns `True`.

Remember, `iterable` could be infinite!

```
def filter_iter(iterable, f):
    """
    Generates elements of iterable for which f returns True.
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter_iter(range(5), is_even)) # a list of the values yielded from the call
     to filter_iter
    [0, 2, 4]
    >>> all_odd = (2*y-1 for y in range(5))
    >>> list(filter_iter(all_odd, is_even))
    []
    >>> naturals = (n for n in range(1, 100))
    >>> s = filter_iter(naturals, is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    "*** YOUR CODE HERE ***"
```

Q4: What's the Difference?

Implement `differences`, a generator function that takes an iterable `it` whose elements are numbers. `differences` should produce a generator that yield the differences between successive terms of `it`. If `it` has less than 2 values, `differences` should yield nothing.

```
def differences(it):
    """
    Yields the differences between successive terms of iterable it.

    >>> d = differences(iter([5, 2, -100, 103]))
    >>> [next(d) for _ in range(3)]
    [-3, -102, 203]
    >>> list(differences([1]))
    []
    """
    "*** YOUR CODE HERE ***"
```

Q5: Primes Generator

Write a function `primes_gen` that takes a single argument `n` and yields all prime numbers less than or equal to `n` in decreasing order. Assume `n >= 1`. You may use the `is_prime` function included below, which we implemented in Discussion 3.

First approach this problem using a `for` loop and using `yield`.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i == 0:
            return False
        return helper(i + 1)
    return helper(2)

def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    "*** YOUR CODE HERE ***"
```

Now that you've done it using a `for` loop and `yield`, try using `yield from`!

Optional Challenge: Now rewrite the generator so that it also prints the primes in *ascending order*.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i == 0:
            return False
        return helper(i + 1)
    return helper(2)

def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    if ___________________________________:
        return
    if ___________________________________:
        yield ___________________________________
    yield from ___________________________________
```

Q6: Stair Ways

In discussion 4, we considered how many different ways there are to climb a flight of stairs with `n` steps if you are able to take 1 or 2 steps at a time. In this problem, you will write a generator function `stair_ways` that yields all the different ways you can climb such a staircase.

Each "way" of climbing a staircase is represented by a list of `1`s and `2`s, representing the sequence of step sizes a person should take to climb the flight.

For example, for a flight with 3 steps, there are three ways to climb it: * You can take one step each time: `[1, 1, 1]`. * You can take two steps then one step: `[2, 1]`. * You can take one step then two steps: `[1, 2]`..

Therefore, `stair_ways(3)` should yield `[1, 1, 1]`, `[2, 1]`, and `[1, 2]` in any order.

```
def stair_ways(n):
    """
    Yields all ways to climb a set of N stairs taking
    1 or 2 steps at a time.

    >>> list(stair_ways(0))
    [[]]
    >>> s_w = stair_ways(4)
    >>> sorted([next(s_w) for _ in range(5)])
    [[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2]]
    >>> list(s_w) # Ensure you're not yielding extra
    []
    """
    "*** YOUR CODE HERE ***"
```

CS 61A Fall 2023

OOP, String Representation

Discussion 7: October 11, 2023

Note: For formal explanations of the concepts on this discussion, feel free to look at the **Appendix** section on the back of the worksheet.

OOP

Here's a recap of the OOP vocab we've learned so far:

- **class**: a template for creating objects
- **instance**: a single object created from a class
- **instance variable**: a data attribute of an object, specific to an instance
- **class variable**: a data attribute of an object, shared by all instances of a class
- **method**: a bound function that may be called on all instances of a class

Instance variables, class variables, and methods are all considered **attributes** of an object.

Q1: WWPD: Legally Blonde OOP

Below we have defined the classes `Student` and `Professor`. Remember that Python passes the `self` argument implicitly to methods when calling the method directly on an object.

```
class Student:

    extension_days = 3 # this is a class variable

    def __init__(self, name, staff):
        self.name = name # this is an instance variable
        self.understanding = 0
        staff.add_student(self)
        print("Added", self.name)

    def visit_office_hours(self, staff):
        staff.assist(self)
        print("Thanks, " + staff.name)

class Professor:

    def __init__(self, name):
        self.name = name
        self.students = {}

    def add_student(self, student):
        self.students[student.name] = student
```

```
    def assist(self, student):
        student.understanding += 1

    def grant_more_extension_days(self, student, days):
        student.extension_days = days
```

What will the following lines output?

```
>>> callahan = Professor("Callahan")
>>> elle = Student("Elle", callahan)
```

```
>>> elle.visit_office_hours(callahan)
```

```
>>> elle.visit_office_hours(Professor("Paulette"))
```

```
>>> elle.understanding
```

```
>>> [name for name in callahan.students]
```

```
>>> x = Student("Vivian", Professor("Stromwell")).name
```

```
>>> x
```

```
>>> elle.extension_days
```

```
>>> callahan.grant_more_extension_days(elle, 7)
>>> elle.extension_days
```

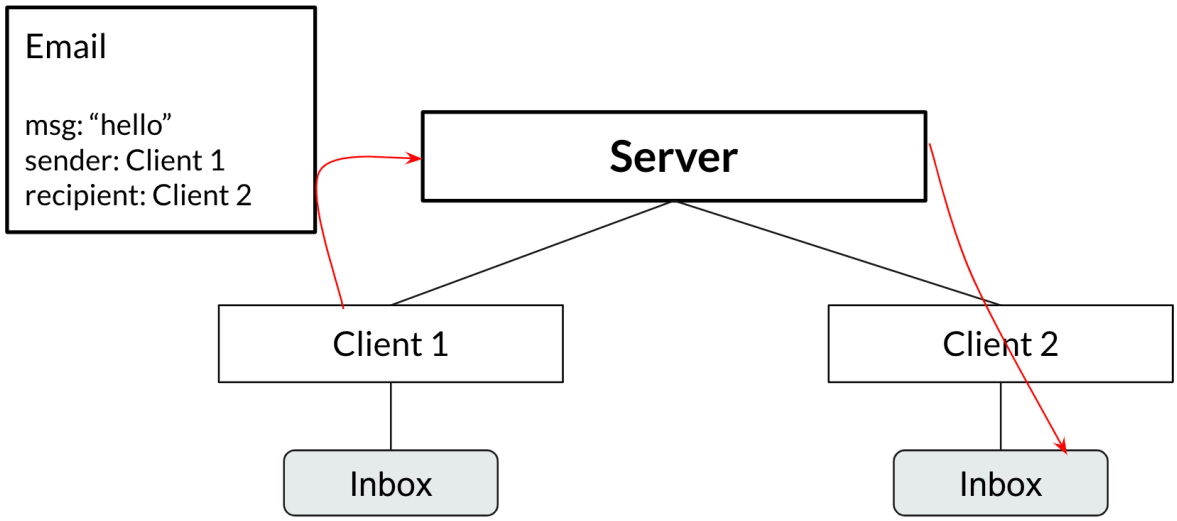
```
>>> Student.extension_days
```

Q2: Email

We would like to write three different classes (`Server`, `Client`, and `Email`) to simulate a system for sending and receiving emails. A `Server` has a dictionary mapping client names to `Client` objects, and can both send `Email`s to `Client`s in the `Server` and register new `Client`s. A `Client` can both compose emails (which first creates a new `Email` object and then sends it to the recipient client through the server) and receive an email (which places an email into the client's inbox).

Emails will only be sent/received within the same server, so clients will always use the server they're registered in to send emails to other clients that are registered in the same rerver.

An example flow: A `Client` object (Client 1) composes an `Email` object with message `"hello"` with recipient Client 2, which the `Server` routes to Client 2's inbox.



Email example

Fill in the definitions below to finish the implementation!

```
class Email:
    """
    Every email object has 3 instance attributes: the
    message, the sender name, and the recipient name.
    >>> email = Email('hello', 'Alice', 'Bob')
    >>> email.msg
    'hello'
    >>> email.sender_name
    'Alice'
    >>> email.recipient_name
    'Bob'
    """
    def __init__(self, msg, sender_name, recipient_name):
        "*** YOUR CODE HERE ***"
```

```
class Client:
    """
    Every Client has three instance attributes: name (which is
    used for addressing emails to the client), server
    (which is used to send emails out to other clients), and
    inbox (a list of all emails the client has received).

    >>> s = Server()
    >>> a = Client(s, 'Alice')
    >>> b = Client(s, 'Bob')
    >>> a.compose('Hello, World!', 'Bob')
    >>> b.inbox[0].msg
    'Hello, World!'
    >>> a.compose('CS 61A Rocks!', 'Bob')
    >>> len(b.inbox)
    2
    >>> b.inbox[1].msg
    'CS 61A Rocks!'
    """
    def __init__(self, server, name):
        self.inbox = []
        "*** YOUR CODE HERE ***"




    def compose(self, msg, recipient_name):
        """Send an email with the given message msg to the given recipient client."""
        "*** YOUR CODE HERE ***"




    def receive(self, email):
        """Take an email and add it to the inbox of this client."""
        "*** YOUR CODE HERE ***"
```

```
class Server:
    """
    Each Server has one instance attribute: clients (which
    is a dictionary that associates client names with
    client objects).
    """
    def __init__(self):
        self.clients = {}

    def send(self, email):
        """
        Take an email and put it in the inbox of the client
        it is addressed to.
        """
        "*** YOUR CODE HERE ***"



    def register_client(self, client, client_name):
        """
        Takes a client object and client_name and adds them
        to the clients instance attribute.
        """
        "*** YOUR CODE HERE ***"
```

Q3: Keyboard

Below is the definition of a `Button` class, which represents a button on a keyboard. It has three attributes: `pos` (numerical position of the button on the keyboard), `key` (the letter of the button), and `times_pressed` (the number of times the button is pressed).

```
class Button:
    def __init__(self, pos, key):
        self.pos = pos
        self.key = key
        self.times_pressed = 0
```

We'd like to create a `Keyboard` class that takes in an arbitrary number of `Button`s and stores these `Button`s in a dictionary. The keys in the dictionary will be `int`s that represent the position on the `Keyboard`, and the values will be the respective `Button`. Fill out the methods in the `Keyboard` class according to each description.

Important: Utilize the doctests as a reference for the behavior of a `Keyboard` instance.

- **Hint:** You can iterate over `*args` as if it were a list.

```
class Button:
    def __init__(self, pos, key):
        self.pos = pos
        self.key = key
        self.times_pressed = 0

class Keyboard:
    """A Keyboard stores an arbitrary number of Buttons in a dictionary.
    Each dictionary key is a Button's position, and each dictionary
    value is the corresponding Button.
    >>> b1, b2 = Button(5, "H"), Button(7, "I")
    >>> k = Keyboard(b1, b2)
    >>> k.buttons[5].key
    'H'
    >>> k.press(7)
    'I'
    >>> k.press(0) # No button at this position
    ''
    >>> k.typing([5, 7])
    'HI'
    >>> k.typing([7, 5])
    'IH'
    >>> b1.times_pressed
    2
    >>> b2.times_pressed
    3
    """
    def __init__(self, *args):
        ----------------
        for _________ in _________________:
            ----------------

    def press(self, pos):
        """Takes in a position of the button pressed, and
        returns that button's output."""
        if ____________________:
            ----------------
            ----------------
            ----------------
        ----------------

    def typing(self, typing_input):
        """Takes in a list of positions of buttons pressed, and
        returns the total output."""
        ----------------
        for ________ in ____________________:
            ----------------
        ----------------
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Inheritance

Recall that a *subclass* (child class) by default inherits all of the methods and class attributes of its *superclass* (parent class). The subclass can override methods and class attributes by redefining them. `super()` can be used to access the methods and class attributes of the parent class.

Q4: That's inheritance, init?

Let's say we want to create a class `Monarch` that inherits from another class, `Butterfly`. We've partially written an `__init__` method for `Monarch`. For each of the following options, state whether it would correctly complete the method so that every instance of `Monarch` has all of the instance attributes of a `Butterfly` instance. You may assume that a monarch butterfly has the default value of 2 wings.

```
class Butterfly():
    def __init__(self, wings=2):
        self.wings = wings

class Monarch(Butterfly):
    def __init__(self):
        ________________________________________
        self.colors = ['orange', 'black', 'white']
```

```
super.__init__()
```

```
super().__init__()
```

```
Butterfly.__init__()
```

```
Butterfly.__init__(self)
```

Some butterflies like the Owl Butterfly have adaptations that allow them to mimic other animals with their wing patterns. Let's write a class for these `MimicButterflies`. In addition to all of the instance variables of a regular `Butterfly` instance, these should also have an instance variable `mimic_animal` describing the name of the animal they mimic. Fill in the blanks in the lines below to create this class.

```
class MimicButterfly(_______________):
    def __init__(self, mimic_animal):
        ________________.__init__()
        _______________ = mimic_animal
```

Q5: Cat

Below is the implementation of a `Pet` class. Each pet has three instance attributes (`is_alive`, `name`, and `owner`), as well as two class methods (`eat` and `talk`).

```
class Pet():

    def __init__(self, name, owner):
        self.is_alive = True    # It's alive!!!
        self.name = name
        self.owner = owner

    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")

    def talk(self):
        print(self.name)
```

Implement the `Cat` class, which inherits from the `Pet` class seen above. To complete the implementation, override the `__init__` and `talk` methods and add a new `lose_life` method.

Hint: You can call the `__init__` method of `Pet` (the superclass of `Cat`) to set a cat's `name` and `owner`.

Hint: The `__init__` method can be called at any point and used just like any other method.

```
class Cat(Pet):

    def __init__(self, name, owner, lives=9):
        "*** YOUR CODE HERE ***"




    def talk(self):
        """Print out a cat's greeting.

        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """
        "*** YOUR CODE HERE ***"


    def lose_life(self):
        """Decrements a cat's life by 1. When lives reaches zero,
        is_alive becomes False. If this is called after lives has
        reached zero, print 'This cat has no more lives to lose.'
        """
        "*** YOUR CODE HERE ***"









    def revive(self):
        """Revives a cat from the dead. The cat should now have
        9 lives and is_alive should be true. Can only be called
        on a cat that is dead. If the cat isn't dead, print
        'This cat still has lives to lose.'
        """
        if not self.is_alive:
            ----------------------------
        else:
            ----------------------------
```

Q6: NoisyCat

More cats! Fill in this implementation of a class called `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot: in fact, it talks twice as much as a regular `Cat`! If you'd like to test your code, feel free to copy over your solution to the `Cat` class above.

```
class __________ # Fill me in!
    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?
        "*** YOUR CODE HERE ***"




    def talk(self):
        """Talks twice as much as a regular cat.
        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """
        "*** YOUR CODE HERE ***"
```

Representation: Repr, Str

Recall that, for any given object `obj`:

- `repr(obj)` is used to get a formal representation of `obj`, which is displayed when `obj` is evaluated directly in the interpreter. The `__repr__` method of `obj` defines the output of `repr(obj)`.
- `str(obj)` is used to get a human-readable representation of `obj`, which is displayed when `print(obj)` is evaluated directly in the interpreter. The `__str__` method of `obj` defines the output of `str(obj)`.

Q7: WWPD: Repr-esentation

Note: This is not the typical way `repr` is used, nor is this way of writing `repr` recommended, this problem is mainly just to make sure you understand how `repr` and `str` work.

```
class Car:
    def __init__(self, color):
        self.color = color

    def __repr__(self):
         return self.color

    def __str__(self):
         return self.color * 2

class Garage:
    def __init__(self):
         print('Vroom!')
         self.cars = []

    def add_car(self, car):
         self.cars.append(car)

    def __repr__(self):
         print(len(self.cars))
         ret = ''
         for car in self.cars:
             ret += str(car)
         return ret
```

Given the above class definitions, what will the following lines output?

```
>>> Car('red')
```

```
>>> print(Car('red'))
```

```
>>> repr(Car('blue'))
```

```
>>> g = Garage()
```

```
>>> g.add_car(Car('red'))
>>> g.add_car(Car('blue'))
>>> g
```

Q8: Cat Representation

Now let's implement the `__str__` and `__repr__` methods for the `Cat` class from earlier so that they exhibit the following behavior:

```
>>> cat = Cat("Felix", "Kevin")
>>> cat
Felix, 9 lives
>>> cat.lose_life()
>>> cat
Felix, 8 lives
>>> print(cat)
Felix
```

```
# (The rest of the Cat class is omitted here, but assume all methods from the Cat class
    above are implemented)
    def __repr__(self):
        "*** YOUR CODE HERE ***"


    def __str__(self):
        "*** YOUR CODE HERE ***"
```

Appendix: Explanation of Material

OOP

Object-oriented programming (OOP) is a programming paradigm that allows us to treat data as objects, like we do in real life.

For example, consider the **class** `Student`. Each of you as individuals is an **instance** of this class.

Details that all CS 61A students have, such as `name`, are called **instance variables**. Every student has these variables, but their values differ from student to student. A variable that is shared among all instances of `Student` is known as a **class variable**. For example, the `extension_days` attribute is a class variable as it is a property of all students.

All students are able to do homework, attend lecture, and go to office hours. When functions belong to a specific object, they are called **methods**. In this case, these actions would be methods of `Student` objects.

Here is a recap of what we discussed above:

- **class**: a template for creating objects
- **instance**: a single object created from a class
- **instance variable**: a data attribute of an object, specific to an instance
- **class variable**: a data attribute of an object, shared by all instances of a class
- **method**: a bound function that may be called on all instances of a class

Instance variables, class variables, and methods are all considered **attributes** of an object.

Inheritance

To avoid redefining attributes and methods for similar classes, we can write a single **base class** from which the similar classes **inherit**. For example, we can write a class called `Pet` and define `Dog` as a **subclass** of `Pet`:

```
class Pet:

    def __init__(self, name, owner):
        self.is_alive = True    # It's alive!!!
        self.name = name
        self.owner = owner

    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")

    def talk(self):
        print(self.name)

class Dog(Pet):

    def talk(self):
        super().talk()
        print('This Dog says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class **is a** more specific version of the other: a dog **is a** pet (We use **is a** to describe this sort of relationship in OOP languages, and not to refer to the Python `is` operator).

Since `Dog` inherits from `Pet`, the `Dog` class will also inherit the `Pet` class's methods, so we don't have to redefine `__init__` or `eat`. We do want each `Dog` to `talk` in a `Dog`-specific way, so we can **override** the `talk` method.

We can use `super()` to refer to the superclass of `self`, and access any superclass methods as if we were an instance of the superclass. For example, `super().talk()` in the `Dog` class will call the `talk()` method from the `Pet` class, but passing the `Dog` instance as the `self`.

This is a little bit of a simplification, and if you're interested you can read more in the Python documentation on `super`.

Representation: Repr, Str

There are two main ways to produce the "string" of an object in Python: `str()` and `repr()`. While the two are similar, they are used for different purposes.

`str()` is used to describe the object to the end user in a "Human-readable" form, while `repr()` can be thought of as a "Computer-readable" form mainly used for debugging and development.

When we define a class in Python, `__str__` and `__repr__` are both built-in methods for the class.

We can call those methods using the global built-in functions `str(obj)` or `repr(obj)` instead of dot notation, `obj.__repr__()` or `obj.__str__()`.

In addition, the `print()` function calls the `__str__` method of the object and displays the returned string **with the quotations removed**, while simply calling the object in interactive mode in the interpreter calls the `_repr__` method and displays the returned string **with the quotations removed**.

Here are some examples:

```
class Rational:

    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __str__(self):
        return str(self.numerator) + '/' + str(self.denominator)

    def __repr__(self):
        return 'Rational' + '(' + str(self.numerator) + ',' + str(self.denominator) + ')'

>>> a = Rational(1, 2)
>>> [str(a), repr(a)]
['1/2', 'Rational(1,2)']
>>> print(a)
1/2
>>> a
Rational(1,2)
```

CS 61A Fall 2023

Linked Lists, Efficiency

Discussion 8: October 18, 2023

Note: For formal explanations of the concepts on this discussion, feel free to look at the **Appendix** section on the back of the worksheet.

Linked Lists

A **linked list** is a recursive data structure that represents sequences. The `Link` class implements linked lists in Python. Each `Link` instance has a `first` attribute (which stores the first value of the linked list) and a `rest` attribute (which points to the rest of the linked list). An empty linked list is represented as `Link.empty`, and a non-empty linked list is represented as a `Link` instance.

Q1: WWPD: Linked Lists

What would Python display? Try drawing the box-and-pointer diagram if you get stuck!

```
>>> link = Link(1, Link(2, Link(3)))
>>> link.first
```

```
>>> link.rest.first
```

```
>>> link.rest.rest.rest is Link.empty
```

```
>>> link.rest = link.rest.rest
>>> link.rest.first
```

```
>>> link = Link(1)
>>> link.rest = link
>>> link.rest.rest.rest.rest.first
```

```
>>> link = Link(2, Link(3, Link(4)))
>>> link2 = Link(1, link)
>>> link2.rest.first
```

```
>>> link = Link(1000, 2000)
```

```
>>> link = Link(1000, Link())
```

```
>>> link = Link(Link("Hello"), Link(2))
>>> link.first
```

```
>>> link = Link(Link("Hello"), Link(2))
>>> link.first.rest is Link.Empty
```

Q2: Sum Nums

Write a function `sum_nums` that receives a linked list `s` and returns the sum of its elements. You may assume the elements of `s` are all integers. Try to implement `sum_nums` with recursion!

```
def sum_nums(s):
    """
    Returns the sum of the elements in s.

    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """
    "*** YOUR CODE HERE ***"
```

Q3: Remove All

Write a function `remove_all` that takes a linked list and a `value` as input. This function mutates the linked list by removing all nodes that store `value`.

You may assume the first element of the linked list is not equal to `value`. You should mutate the input linked list; `remove_all` does not return anything.

```
def remove_all(link, value):
    """Removes all nodes in link that contain value. The first element of
    link is never equal to value.

    >>> l1 = Link(0, Link(2, Link(2, Link(3, Link(1, Link(2, Link(3)))))))
    >>> print(l1)
    <0 2 2 3 1 2 3>
    >>> remove_all(l1, 2)
    >>> print(l1)
    <0 3 1 3>
    >>> remove_all(l1, 3)
    >>> print(l1)
    <0 1>
    >>> remove_all(l1, 3)
    >>> print(l1)
    <0 1>
    """
    "*** YOUR CODE HERE ***"
```

Q4: Flip Two

Write a recursive function `flip_two` that receives a linked list `s` and flips every pair of values in `s`.

```
def flip_two(s):
    """
    Flips every pair of values in s.

    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))
    """
    "*** YOUR CODE HERE ***"









"*** YOUR CODE HERE ***"
```

Q5: Make Circular

Write a function `make_circular` that takes in a non-circular, non-empty linked list `s` and mutates `s` so that it becomes circular.

```
def make_circular(s):
    """Mutates linked list s into a circular linked list.

    >>> lnk = Link(1, Link(2, Link(3)))
    >>> make_circular(lnk)
    >>> lnk.rest.first
    2
    >>> lnk.rest.rest.first
    3
    >>> lnk.rest.rest.rest.first
    1
    >>> lnk.rest.rest.rest.rest.first
    2
    """
    "*** YOUR CODE HERE ***"
```

Efficiency

A function's runtime complexity is a measure of how the runtime of the function changes as its input changes. A function `f(n)` has…

- constant runtime if the runtime of `f` does not depend on `n`. Its runtime is $\Theta(1)$.
- logarithmic runtime if the runtime of `f` is proportional to `log(n)`. Its runtime is $\Theta(\log(n))$.
- linear runtime if the runtime of `f` is proportional to `n`. Its runtime is $\Theta(n)$.
- quadratic runtime if the runtime of `f` is proportional to `n^2`. Its runtime is $\Theta(n^2)$.
- exponential runtime if the runtime of `f` is proportional to `b^n`, for some constant `b`. Its runtime is $\Theta(b^n)$.

Q6: WWPD: Orders of Growth

What is the *worst-case* runtime of `is_prime`?

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

What is the order of growth of the runtime of `bar(n)` with respect to `n`?

```
def bar(n):
    i, sum = 1, 0
    while i <= n:
        sum += biz(n)
        i += 1
    return sum

def biz(n):
    i, sum = 1, 0
    while i <= n:
        sum += i**3
        i += 1
    return sum
```

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

What is the order of growth of the runtime of `foo` in terms of `n`, where `n` is the length of `lst`? Assume that slicing a list and evaluating `len(lst)` take constant time.

Express your answer with Θ notation.

```
def foo(lst, i):
    mid = len(lst) // 2
    if mid == 0:
        return lst
    elif i > 0:
        return foo(lst[mid:], -1)
    else:
        return foo(lst[:mid], 1)
```

Appendix: Explanation of Material

Linked Lists

The `Link` class implements linked lists in Python. Each `Link` instance has a `first` attribute (which stores the first value of the linked list) and a `rest` attribute (which points to the rest of the linked list).

```
class Link:
    empty = ()

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

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

An empty linked list is represented as `Link.empty`, and a non-empty linked list is represented as a `Link` instance.

The `rest` attribute of a `Link` instance is always another linked list! When `Link` instances are linked via their `rest` attributes, a sequence is formed.

To check if a linked list is empty, compare it to the class attribute `Link.empty`.

Efficiency

Throughout this class, we have mainly focused on *correctness* — whether a program produces the correct output. However, computer scientists are also interested in creating *efficient* solutions to problems. One way to quantify efficiency is to determine how a function's *runtime* changes as its input changes. In this class, we measure a function's runtime by the number of operations it performs.

A function `f(n)` has…

- constant runtime if the runtime of `f` does not depend on `n`. Its runtime is $\Theta(1)$.
- logarithmic runtime if the runtime of `f` is proportional to `log(n)`. Its runtime is $\Theta(\log(n))$.
- linear runtime if the runtime of `f` is proportional to `n`. Its runtime is $\Theta(n)$.
- quadratic runtime if the runtime of `f` is proportional to `n^2`. Its runtime is $\Theta(n^2)$.
- exponential runtime if the runtime of `f` is proportional to `b^n`, for some constant `b`. Its runtime is $\Theta(b^n)$.

Example 1: It takes a single multiplication operation to compute `square(1)`, and it takes a single multiplication operation to compute `square(100)`. In general, calling `square(n)` results in a *constant* number of operations that does not vary according to `n`. We say `square` has a runtime complexity of $\Theta(1)$.

input	function call	return value	operations
1	`square(1)`	1*1	1
2	`square(2)`	2*2	1
…	…	…	…
100	`square(100)`	100*100	1
…	…	…	…
n	`square(n)`	n*n	1

Example 2: It takes a single multiplication operation to compute `factorial(1)`, and it takes 100 multiplication operations to compute `factorial(100)`. As `n` increases, the runtime of `factorial` increases *linearly*. We say `factorial` has a runtime complexity of $\Theta(n)$.

input	function call	return value	operations
1	`factorial(1)`	1*1	1
2	`factorial(2)`	2*1*1	2
…	…	…	…
100	`factorial(100)`	100*99*…*1*1	100
…	…	…	…
n	`factorial(n)`	n*(n-1)*…*1*1	n

Example 3: Consider the following function:

```
def bar(n):
    for a in range(n):
        for b in range(n):
            print(a,b)
```

Evaulating `bar(1)` results in a single `print` call, while evalulating `bar(100)` results in 10,000 `print` calls. As `n`

increases, the runtime of `bar` increases *quadratically*. We say `bar` has a runtime complexity of Θ(`n^2`).

input	function call	operations (prints)
1	`bar(1)`	1
2	`bar(2)`	4
...	...	...
100	`bar(100)`	10000
...	...	...
n	`bar(n)`	n^2

Example 4: Consider the following function:

```
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n - 1) + rec(n - 1)
```

Evaluating `rec(1)` results in a single addition operation. Evaluating `rec(4)` results in `2^4 - 1` = 15 addition operations, as shown by the diagram below.

During the evaulation of `rec(4)`, there are two calls to `rec(3)`, four calls to `rec(2)`, eight calls to `rec(1)`, and 16 calls to `rec(0)`.

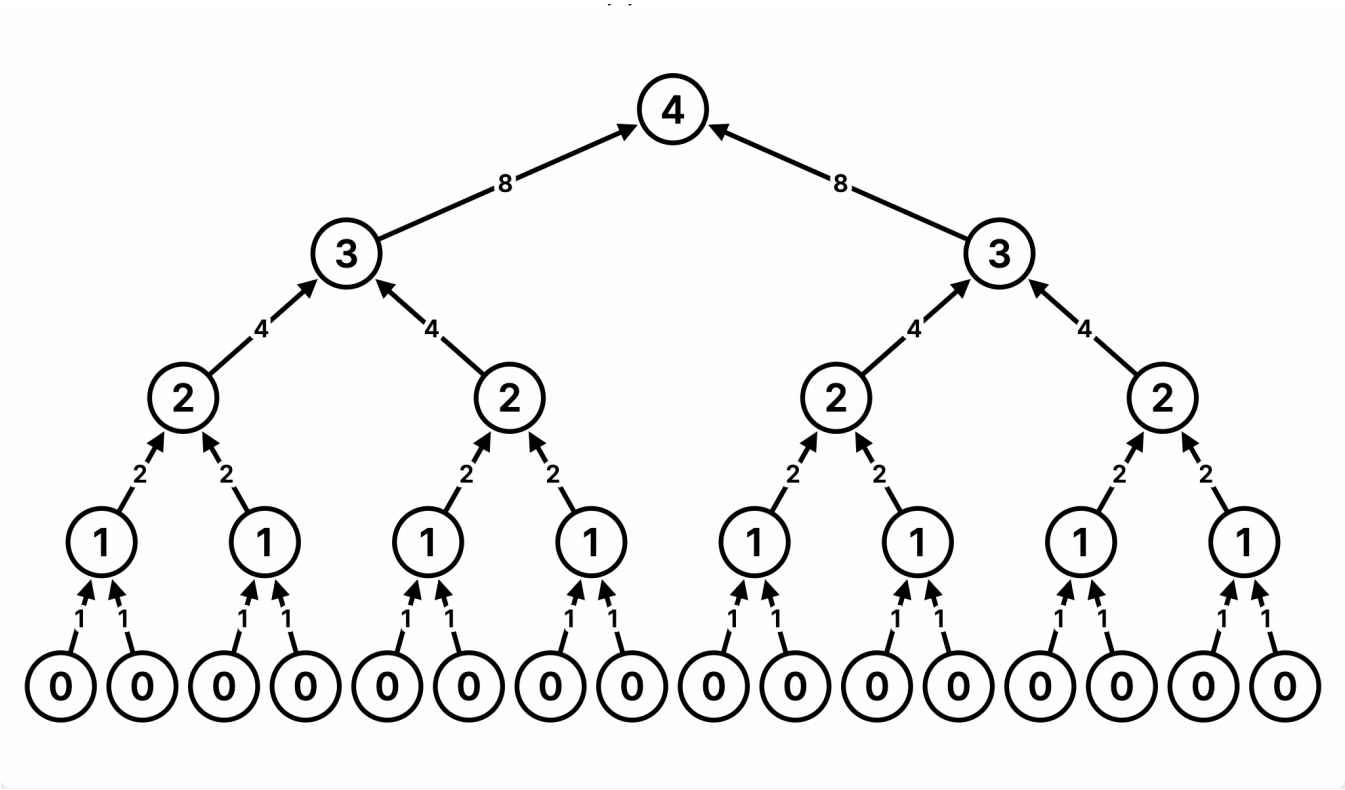
So we have eight instances of `rec(0) + rec(0)`, four instances of `rec(1) + rec(1)`, two instances of `rec(2) + rec(2)`, and a single instance of `rec(3) + rec(3)`, for a total of 1 + 2 + 4 + 8 = 15 addition operations.

As `n` increases, the runtime of `rec` increases *exponentially*. In particular, the runtime of `rec` approximately doubles when we increase `n` by 1. We say `rec` has a runtime complexity of Θ(`2^n`).

input	function call	return value	operations
1	`rec(1)`	2	1
2	`rec(2)`	4	3
...	...	...	...
10	`rec(10)`	1024	1023
...	...	...	...
n	`rec(n)`	2^n	2^n - 1

Tips for finding the order of growth of a function's runtime:

- If the function is recursive, determine the number of recursive calls and the runtime of each recursive call.
- If the function is iterative, determine the number of inner loops and the runtime of each loop.
- Ignore coefficients. A function that performs `n` operations and a function that performs `100 * n` operations are both linear.
- Choose the largest order of growth. If the first part of a function has a linear runtime and the second part has a quadratic runtime, the overall function has a quadratic runtime.
- In this course, we only consider constant, logarithmic, linear, quadratic, and exponential runtimes.



Above: Call structure of rec(4).

CS 61A Fall 2023

Midterm Review

Discussion 9: October 25, 2023

This discussion is optional: that means that attendance is not expected, and you will receive no credit for attending this discussion.

Your TA will not be able to get to all of the problems on this worksheet so feel free to work through the remaining problems on your own. Bring any questions you have to office hours or post them on Ed. Good luck on the midterm!

Fun

Q1: Around and Around

Berkeley students can juggle a lot of responsibilities. But can they juggle balls? Your TA has brought some tennis balls for you to use. Let's learn to juggle!

0. Get comfortable Get into pairs, and introduce yourselves if you don't already know each other. Take a ball and toss it from hand to hand in whatever way feels right — get a feel for things!

1. One ball Now that you're comfortable with the feel of the ball, let's practice our one ball toss. Take a single ball and toss it from one hand to the other. The ball should reach its apex above your opposite shoulder, and you should catch it slightly outside of that shoulder.

> Getting a solid, high throw is crucial here! Don't cut corners!

You should feel like you're "scooping" inward and upward as you do the throws. This will help you transition from catching to throwing.

After you do this for a minute, have one partner stop juggling to observe the other and provide guidance and feedback. Then, switch roles.

2. Two balls Let's double the number of balls we're using. To get the two-ball throw down, start by tossing one ball from hand to hand. When that ball reaches its apex, throw the second ball!

> If you're having trouble throwing the second ball, don't even worry about catching the first one. Your body will know how to catch it. The throw is the hardest part! One tip is to say "left right" or "right left" as you throw the two balls.

Get comfortable throwing two balls, and consider switching up which hand you start with. After you do this for a minute, have one partner stop juggling to observe the other and provide guidance and feedback. Then, switch roles.

3. Three balls Let's add the final ball. Start with two balls in your preferred hand and one ball in your other hand. Throw the two balls as before, but now toss the third ball when the second one reaches its apex!

> Again, the throw is the hardest part! Consider saying "left right left" or "right left right" as you throw to get it into your head.

After you do this for a minute, have one partner stop juggling to observe the other and provide guidance and feedback. Then, switch roles.

4. Cascade Now, all you have to do is keep the cycle going! If you're still having trouble with this, feel free to take some balls home and practice.

Recursion

Q2: Paths List

(Adapted from Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns a list of paths, where each path is a list containing steps to reach `y` from `x` by repeated incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9, so one path is [3, 4, 8, 9].

```
def paths(x, y):
    """Return a list of ways to reach y from x by repeated
    incrementing or doubling.
    >>> paths(3, 5)
    [[3, 4, 5]]
    >>> sorted(paths(3, 6))
    [[3, 4, 5, 6], [3, 6]]
    >>> sorted(paths(3, 9))
    [[3, 4, 5, 6, 7, 8, 9], [3, 4, 8, 9], [3, 6, 7, 8, 9]]
    >>> paths(3, 3) # No calls is a valid path
    [[3]]
    >>> paths(5, 3) # There is no valid path from x to y
    []
    """
    if ____________________________________
        return ____________________________________
    elif ____________________________________
        return ____________________________________
    else:
        a = ____________________________________
        b = ____________________________________
        return ____________________________________
```

Trees

Q3: Widest Level

Write a function that takes a `Tree` object and returns the elements at the depth with the most elements.

In this problem, you may find it helpful to use the second optional argument to `sum`, which provides a starting value. All items in the sequence to be summed will be concatenated to the starting value. By default, start will default to 0, which allows you to sum a sequence of numbers. We provide an example of sum starting with a list, which allows you to concatenate items in a list.

```
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...              Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    levels = []
    x = [t]
    while _____________:
        _____________
        __________ = sum(_______________________________, [])
    return max(levels, key=_________________________________)
```

Q4: Level Mutation Link

As a reminder, the depth of a node is how far away the node is from the root. We define this as the number of edges between the root to the node. As there are no edges between the root and itself, the root has depth 0.

Given a tree `t` and a linked list of one-argument functions `funcs`, write a function that will mutate the labels of `t` using the function from `funcs` at the corresponding depth. For example, the label at the root node (with a depth of 0) will be mutated using the function at `funcs.first`. Assume all of the functions in `funcs` will be able to take in a label value and return a valid label value.

If `t` is a leaf and there are more than 1 functions in `funcs`, all of the remaining functions should be applied in order to the label of `t`. (See the doctests for an example.) If `funcs` is empty, the tree should remain unmodified.

```
def level_mutation_link(t, funcs):
    """Mutates t using the functions in the linked list funcs.

    >>> t = Tree(1, [Tree(2, [Tree(3)])])
    >>> funcs = Link(lambda x: x + 1, Link(lambda y: y * 5, Link(lambda z: z ** 2)))
    >>> level_mutation_link(t, funcs)
    >>> t
    Tree(2, [Tree(10, [Tree(9)])])
    >>> t2 = Tree(1, [Tree(2), Tree(3, [Tree(4)])])
    >>> level_mutation_link(t2, funcs)
    >>> t2
    Tree(2, [Tree(100), Tree(15, [Tree(16)])])
    >>> t3 = Tree(1, [Tree(2)])
    >>> level_mutation_link(t3, funcs)
    >>> t3
    Tree(2, [Tree(100)])
    """
    if ______________________:
        return
    t.label = ______________________
    remaining = ______________________
    if ________________ and ________________:
        while ______________________:
            ______________________
            remaining = remaining.rest
    for b in t.branches:
        ______________________
```

Lists and Mutability

Q5: Shuffle

Define a function `shuffle` that takes a sequence with an even number of elements (cards) and creates a new list that interleaves the elements of the first half with the elements of the second half.

To interleave two sequences `s0` and `s1` is to create a new sequence such that the new sequence contains (in this order) the first element of `s0`, the first element of `s1`, the second element of `s0`, the second element of `s1`, and so on.

> **Note:** If you're running into an issue where the special heart / diamond / spades / clubs symbols are erroring in the doctests, feel free to copy paste the below doctests into your file as these don't use the special characters and should not give an "illegal multibyte sequence" error.

```
def card(n):
    """Return the playing card numeral as a string for a positive n <= 13."""
    assert type(n) == int and n > 0 and n <= 13, "Bad card n"
    specials = {1: 'A', 11: 'J', 12: 'Q', 13: 'K'}
    return specials.get(n, str(n))

def shuffle(cards):
    """Return a shuffled list that interleaves the two halves of cards.

    >>> shuffle(range(6))
    [0, 3, 1, 4, 2, 5]
    >>> suits = ['H', 'D', 'S', 'C']
    >>> cards = [card(n) + suit for n in range(1,14) for suit in suits]
    >>> cards[:12]
    ['AH', 'AD', 'AS', 'AC', '2H', '2D', '2S', '2C', '3H', '3D', '3S', '3C']
    >>> cards[26:30]
    ['7S', '7C', '8H', '8D']
    >>> shuffle(cards)[:12]
    ['AH', '7S', 'AD', '7C', 'AS', '8H', 'AC', '8D', '2H', '8S', '2D', '8C']
    >>> shuffle(shuffle(cards))[:12]
    ['AH', '4D', '7S', '10C', 'AD', '4S', '7C', 'JH', 'AS', '4C', '8H', 'JD']
    >>> cards[:12]  # Should not be changed
    ['AH', 'AD', 'AS', 'AC', '2H', '2D', '2S', '2C', '3H', '3D', '3S', '3C']
    """
    assert len(cards) % 2 == 0, 'len(cards) must be even'
    half = _____________
    shuffled = []
    for i in _____________:
        _________________
        _________________
    return shuffled
```

Efficiency

Q6: Bonk

Describe the order of growth of the function below.

```
def bonk(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

Q7: Pow

Write the following function so it runs in (log k) time.

> **Hint:** This can be done using a procedure called repeated squaring.

```
def lgk_pow(n,k):
    """Computes n^k.

    >>> lgk_pow(2, 3)
    8
    >>> lgk_pow(4, 2)
    16
    >>> a = lgk_pow(2, 100000) # make sure you have log time
    """
    "*** YOUR CODE HERE ***"
```

Generators

Q8: Yield, Fibonacci!

Implement `fibs`, a generator function that takes a one-argument pure function `f` and yields all Fibonacci numbers `x` for which `f(x)` returns a true value. The Fibonacci numbers begin with 0 and then 1. Each subsequent Fibonacci number is the sum of the previous two. Yield the Fibonacci numbers in order.

```
def fibs(f):
    """Yield all Fibonacci numbers x for which f(x) is a true value.
    >>> odds = fibs(lambda x: x % 2 == 1)
    >>> [next(odds) for i in range(10)]
    [1, 1, 3, 5, 13, 21, 55, 89, 233, 377]
    >>> bigs = fibs(lambda x: x > 20)
    >>> [next(bigs) for i in range(10)]
    [21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
    >>> evens = fibs(lambda x: x % 2 == 0)
    >>> [next(evens) for i in range(10)]
    [0, 2, 8, 34, 144, 610, 2584, 10946, 46368, 196418]
    """
    n, m = 0, 1
    while _________:
        if _________:
            _________
        n, m = _________, _________
```

Q9: Partitions

Tree-recursive generator functions have a similar structure to regular tree-recursive functions. They are useful for iterating over all possibilities. Instead of building a list of results and returning it, just `yield` each result.

You'll need to identify a *recursive decomposition*: how to express the answer in terms of recursive calls that are simpler. Ask yourself what will be yielded by a recursive call, then how to use those results.

Definition. For positive integers `n` and `m`, a *partition* of `n` using parts up to size `m` is an addition expression of positive integers up to `m` in non-decreasing order that sums to `n`.

Implement `partition_gen`, a generator functon that takes positive `n` and `m`. It yields the partitions of `n` using parts up to size `m` as strings.

Reminder: For the `partitions` function we studied in lecture (video), the recursive decomposition was to enumerate all ways of partitioning `n` using at least one `m` and then to enumerate all ways with no `m` (only `m-1` and lower).

```
def partition_gen(n, m):
    """Yield the partitions of n using parts up to size m.

    >>> for partition in sorted(partition_gen(6, 4)):
    ...     print(partition)
    1 + 1 + 1 + 1 + 1 + 1
    1 + 1 + 1 + 1 + 2
    1 + 1 + 1 + 3
    1 + 1 + 2 + 2
    1 + 1 + 4
    1 + 2 + 3
    2 + 2 + 2
    2 + 4
    3 + 3
    """
    assert n > 0 and m > 0
    if n == m:
        yield ____
    if n - m > 0:
        "*** YOUR CODE HERE ***"



    if m > 1:
        "*** YOUR CODE HERE ***"
```

OOP

Q10: Mint

A mint is a place where coins are made. In this question, you'll implement a `Mint` class that can output a `Coin` with the correct year and worth.

- Each `Mint` *instance* has a `year` stamp. The `update` method sets the `year` stamp of the instance to the `present_year` *class attribute* of the `Mint` *class*.
- The `create` method takes a subclass of `Coin` (*not* an instance!), then creates and returns an *instance* of that class stamped with the `mint`'s year (which may be different from `Mint.present_year` if it has not been updated.)
- A `Coin`'s `worth` method returns the `cents` value of the coin plus one extra cent for each year of age *beyond 50*. A coin's age can be determined by subtracting the coin's year from the `present_year` class attribute of the `Mint` class.

```
class Mint:
    """A mint creates coins by stamping on years.

    The update method sets the mint's stamp to Mint.present_year.

    >>> mint = Mint()
    >>> mint.year
    2023
    >>> dime = mint.create(Dime)
    >>> dime.year
    2023
    >>> Mint.present_year = 2103  # Time passes
    >>> nickel = mint.create(Nickel)
    >>> nickel.year     # The mint has not updated its stamp yet
    2023
    >>> nickel.worth()  # 5 cents + (80 - 50 years)
    35
    >>> mint.update()   # The mint's year is updated to 2102
    >>> Mint.present_year = 2178     # More time passes
    >>> mint.create(Dime).worth()    # 10 cents + (75 - 50 years)
    35
    >>> Mint().create(Dime).worth()  # A new mint has the current year
    10
    >>> dime.worth()     # 10 cents + (155 - 50 years)
    115
    >>> Dime.cents = 20  # Upgrade all dimes!
    >>> dime.worth()     # 20 cents + (155 - 50 years)
    125
    """
    present_year = 2023

    def __init__(self):
        self.update()

    def create(self, coin):
        "*** YOUR CODE HERE ***"



    def update(self):
        "*** YOUR CODE HERE ***"
```

```
class Coin:
    cents = None # will be provided by subclasses, but not by Coin itself

    def __init__(self, year):
        self.year = year

    def worth(self):
        "*** YOUR CODE HERE ***"



class Nickel(Coin):
    cents = 5

class Dime(Coin):
    cents = 10
```

Linked Lists

Q11: Every Other

Implement `every_other`, which takes a linked list `s`. It mutates `s` such that all of the odd-indexed elements (using 0-based indexing) are removed from the list. For example:

```
>>> s = Link('a', Link('b', Link('c', Link('d'))))
>>> every_other(s)
>>> s.first
'a'
>>> s.rest.first
'c'
>>> s.rest.rest is Link.empty
True
```

If `s` contains fewer than two elements, `s` remains unchanged.

Do not return anything! `every_other` should mutate the original list.

```
def every_other(s):
    """Mutates a linked list so that all the odd-indiced elements are removed
    (using 0-based indexing).

    >>> s = Link(1, Link(2, Link(3, Link(4))))
    >>> every_other(s)
    >>> s
    Link(1, Link(3))
    >>> odd_length = Link(5, Link(3, Link(1)))
    >>> every_other(odd_length)
    >>> odd_length
    Link(5, Link(1))
    >>> singleton = Link(4)
    >>> every_other(singleton)
    >>> singleton
    Link(4)
    """
    "*** YOUR CODE HERE ***"
```

Q12: Insert

Implement a function `insert` that takes a `Link`, a `value`, and an `index`, and inserts the `value` into the `Link` at the given `index`. You can assume the linked list already has at least one element. Do not return anything – `insert` should mutate the linked list.

Note: If the index is out of bounds, you should raise an `IndexError` with: raise IndexError('Out of bounds!')

```
def insert(link, value, index):
    """Insert a value into a Link at the given index.

    >>> link = Link(1, Link(2, Link(3)))
    >>> print(link)
    <1 2 3>
    >>> other_link = link
    >>> insert(link, 9001, 0)
    >>> print(link)
    <9001 1 2 3>
    >>> link is other_link # Make sure you are using mutation! Don't create a new linked
    list.
    True
    >>> insert(link, 100, 2)
    >>> print(link)
    <9001 1 100 2 3>
    >>> insert(link, 4, 5)
    Traceback (most recent call last):
        ...
    IndexError: Out of bounds!
    """
    "*** YOUR CODE HERE ***"
```

CS 61A Fall 2023

Scheme, Scheme Lists

Discussion 10: November 1, 2023

Note: For extended explanations of the concepts on this discussion, feel free to look at the **Appendix** section on the back of the worksheet.

Introduction

In the next part of the course, we will be working with the **Scheme** programming language. In addition to learning how to write Scheme programs, we will eventually write a Scheme interpreter in Project 4!

Scheme is a famous functional programming language from the 1970s. It is a dialect of Lisp (which stands for LISt Processing). The syntax of Scheme is very unique: it involves prefix notation and many nested parentheses (see http://xkcd.com/297/). Scheme features first-class functions and optimized tail-recursion, which were fairly new when Scheme was introduced.

Primitives and Defining Variables

Scheme *primitives* include numbers, symbols, and booleans. These primitives are said to be *atomic* because they are the simplest building blocks of the language that cannot be subdivided.

`(define <variable name> <expr>)` evaluates `<expr>` and binds its value to `<variable name>` in the current environment.

WWSD

```
scm> (define a 1)
```

```
scm> a
```

```
scm> (define b a)
```

```
scm> b
```

```
scm> (define c 'a)
```

```
scm> c
```

Call Expressions

To evaluate a call expression `(<operator> <operand1> <operand2> ...)` in Scheme:

1. Evaluate the operator to get a procedure.
2. Evaluate each of the operands from left to right.
3. Apply the value of the operator to the evaluated operands.

WWSD

What would Scheme display? As a reminder, the built-in `quotient` function performs floor division.

```
scm> (define a (+ 1 2))
```

```
scm> a
```

```
scm> (define b (- (+ (* 3 3) 2) 1))
```

```
scm> (+ a b)
```

```
scm> (= (modulo b a) (quotient 5 3))
```

Special Forms

Special forms look like call expressions, but they *do not* follow the same rules of evaluation. Scheme identifies a special form based on the presence of a keyword as the operator. These keywords include `define`, `if`, `cond`, `and`, `or`, `let`, `begin`, `lambda`. Each special form has its own special rules of evaluation.

For an explanation of the special forms, check out the appendix of this worksheet and the **Scheme Specification** (https://cs61a.org/articles/scheme-spec/#special-forms-2).

WWSD

What would Scheme display?

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))
```

```
scm> ((if (< 4 3) + -) 4 100)
```

```
scm> (cond
        ((and (- 4 4) (not #t)) 1)
        ((and (or (< 9 (/ 100 10)) (/ 1 0)) #t) -1)
        (else (/ 1 0))
    )
```

```
scm> (let (
            (a (- 3 2))
            (b (+ 5 7))
          )
        (* a b)
        (if (< (+ a b) b)
            (/ a b)
            (/ b a)
        )
    )
```

```
scm> (begin
        (if (even? (+ 2 4))
            (print (and 2 0 3))
            (/ 1 0)
        )
        (+ 2 2)
        (print 'lisp)
        (or 2 0 3)
    )
```

Defining Functions

Q1: Virahanka-Fibonacci

Write a function that returns the `n`-th Virahanka-Fibonacci number.

```
(define (vir-fib n)
    'YOUR-CODE-HERE




)

(expect (vir-fib 10) 55)
(expect (vir-fib 1) 1)
```

```
scm> (vir-fib 0)
0
scm> (vir-fib 1)
1
scm> (vir-fib 10)
55
```

Pairs and Lists

All lists in Scheme are linked lists. Scheme lists are composed of two element pairs. The empty list is `nil`. We use the following procedures to construct and select from lists:

- `(cons first rest)` constructs a list with the given first element and rest of the list. For now, if `rest` is not a pair or `nil` it will error.
- `(car lst)` gets the first item of the list
- `(cdr lst)` gets the rest of the list

Two other ways of creating lists are using the built-in `list` procedure or the `quote` special form.

WWSD

What would Scheme display?

```
scm> (cons 1 (cons 2 nil))
```

```
scm> (car (cons 1 (cons 2 nil)))
```

```
scm> (cdr (cons 1 (cons 2 nil)))
```

```
scm> (list 1 2 3)
```

```
scm> '(1 2 3)
```

```
scm> (cons 1 '(list 2 3))  ; Recall quoting
```

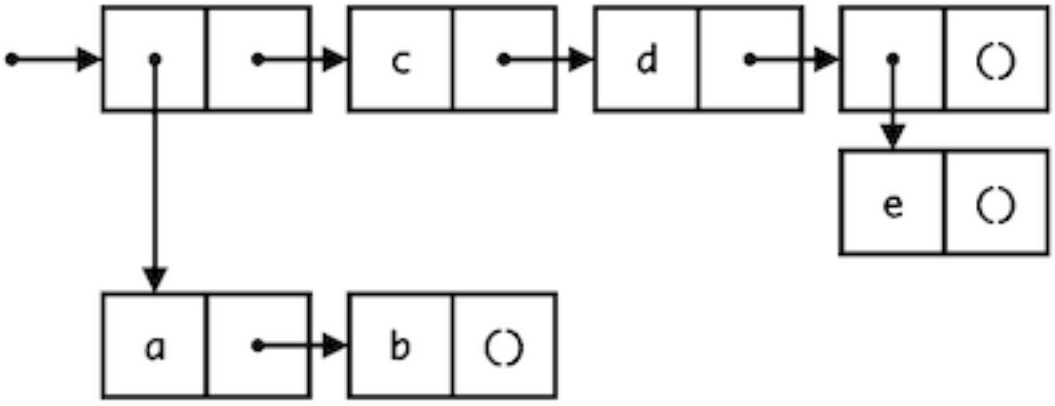
```
scm> '(cons 4 (cons (cons 6 8) ()))
```

```
scm> (cons 1 (list (cons 3 nil) 4 5))
```

Q2: List Making

Let's make some Scheme lists. We'll define the same list with `list`, `quote`, and `cons`.

The following list was visualized using the `draw` feature of code.cs61a.org.



First, use `list`:

```
(define with-list
    (list
        'YOUR-CODE-HERE


    )
)
(draw with-list)
```

Now use `quote`. What differences are there?

```
(define with-quote
    '(
        'YOUR-CODE-HERE


    )

)
(draw with-quote)
```

Now try with `cons`. For convenience, we've defined a `helpful-list` and `another-helpful-list`:

```
(define helpful-list
   (cons 'a (cons 'b nil)))
(draw helpful-list)

(define another-helpful-list
    (cons 'c (cons 'd (cons (cons 'e nil) nil))))
(draw another-helpful-list)

(define with-cons
    (cons
        'YOUR-CODE-HERE


    )
)
(draw with-cons)
```

Q3: List Concatenation

Write a function which takes two lists and concatenates them.

Notice that simply calling `(cons a b)` would not work because it will create a deep list. Do not call the builtin procedure `append`, since it does the same thing as `list-concat` should do.

```
(define (list-concat a b)
    'YOUR-CODE-HERE




)

(expect (list-concat '(1 2 3) '(2 3 4)) (1 2 3 2 3 4))
(expect (list-concat '(3) '(2 1 0)) (3 2 1 0))
```

```
scm> (list-concat '(1 2 3) '(2 3 4))
(1 2 3 2 3 4)
```

Q4: Map

Write a function that takes a procedure and applies it to every element in a given list using your own implementation *without* using the built-in `map` function.

```
(define (map-fn fn lst)
    'YOUR-CODE-HERE




)

(map-fn (lambda (x) (* x x)) '(1 2 3))
; expect (1 4 9)
```

```
scm> (map-fn (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
```

Q5: Remove

Implement a procedure `remove` that takes in a list and returns a new list with *all* instances of `item` removed from `lst`. You may assume the list will only consist of numbers and will not have nested lists.

> *Hint*: You might find the built-in `filter` procedure useful (though it is definitely possible to complete this question without it).
>
> You can find information about how to use `filter` in the 61A Scheme builtin specification!

```
(define (remove item lst)
  'YOUR-CODE-HERE




)

(expect (remove 3 nil) ())
(expect (remove 2 '(1 3 2)) (1 3))
(expect (remove 1 '(1 3 2)) (3 2))
(expect (remove 42 '(1 3 2)) (1 3 2))
(expect (remove 3 '(1 3 3 7)) (1 7))
```

Q6: List Duplicator

Write a Scheme function, `duplicate` that, when given a list, such as `(1 2 3 4)`, duplicates every element in the list (i.e. `(1 1 2 2 3 3 4 4)`).

```
(define (duplicate lst)
    'YOUR-CODE-HERE




)

(expect (duplicate '(1 2 3)) (1 1 2 2 3 3))
(expect (duplicate '(1 1)) (1 1 1 1))
```

Appendix: Explanation of Material

Primitives and Defining Variables

Scheme has a set of **atomic** primitive expressions. Atomic means that these expressions cannot be divided up.

```
scm> 123
123
scm> #t
True
scm> #f
False
```

Unlike in Python, the only primitive in Scheme that is a false value is `#f` and its equivalents, `false` and `False`. **This means that 0 is not false.**

In Scheme, we can use the `define` special form to bind values to symbols, which we can then use as variables. When a symbol is defined this way, the `define` special form returns the symbol.

- `(define <variable name> <expr>)`

Evaluates `<expr>` and binds its value to `<variable name>` in the current environment.

Call Expressions

Call expressions apply a procedure to some arguments.

```
(<operator> <operand1> <operand2> ...)
```

Call expressions in Scheme work exactly like they do in Python. To evaluate them:

1. Evaluate the operator to get a procedure.
2. Evaluate each of the operands from left to right.
3. Apply the value of the operator to the evaluated operands.

For example, consider the call expression `(+ 1 2)`. First, we evaluate the symbol `+` to get the built-in addition procedure. Then we evaluate the two operands `1` and `2` to get their corresponding atomic values. Finally, we apply the addition procedure to the values `1` and `2` to get the return value `3`.

Operators may be symbols, such as `+` and `*`, or more complex expressions, as long as they evaluate to procedure values.

Here is a reference for the Scheme Built-In Procedures.

```
scm> (- 1 1)                ; 1 - 1
0
scm> (* (+ 1 2) (+ 1 2))    ; (1 + 2) * (1 + 2)
9
```

=, eq?, equal?

We can use the `=`, `eq?`, and `equal?` procedures to check the equality of two operands:

- `(= <a> <b>)` returns true if `a` equals `b`. Both must be numbers.
- `(eq? <a> <b>)` returns true if `a` and `b` are equivalent primitive values. For two objects, `eq?` returns true if both refer to the same object in memory. Similar to checking identity between two objects using `is` in Python
- `(equal? <a> <b>)` returns true if `a` and `b` are *pairs* that have the same contents (`car`s and `cdr`s are equivalent). Similar to checking equality between two lists using `==` in Python. If `a` and `b` are not pairs, `equal?` behaves like `eq?`.

```
scm> (define a '(1 2 3))
a
scm> (= a a)
Error
scm> (equal? a '(1 2 3))
#t
scm> (eq? a '(1 2 3))
#f
```

Special Forms

Special form expressions contain a **special form** as the operator. Special form expressions *do not* follow the same rules of evaluation as call expressions. Each special form has its own rules of evaluation – that's what makes them special! Here's the Scheme Specification to reference the special forms we will cover in this class.

It is important to note that everything in Scheme is either an **atomic** or an **expression**, so although these special forms look and operate similarly to Python, they are evaluated differently.

Special forms like `if`, `cond`, `and`, `or` in Python direct the control flow of a program and allow you to evaluate specific expressions under some condition. In Scheme, however, these special forms are expressions that take in a set amount of parameters and return some value based on the condition passed in.

If Expression

An `if` expression looks like this:

`(if <predicate> <if-true> [if-false])`

`<predicate>` and `<if-true>` are required expressions and `[if-false]` is optional.

The rules for evaluation are as follows:

1. Evaluate `<predicate>`.
2. If `<predicate>` evaluates to a truth-y value, evaluate `<if-true>` and return its value. Otherwise, evaluate `[if-false]` if provided and return its value.

`if` is a special form as not all of its operands will be evaluated. The value of the first operand determines whether the second or the third operator is evaluated.

> **Important:** Only `#f` is a false-y value in Scheme; everything else is truth-y, including `0`.

```
scm> (if (< 4 5) 1 2)
1
scm> (if #f (/ 1 0) 42)
42
```

Cond Expression

A `cond` expression looks like this:

`(cond (<pred1> <if-pred1>) (<pred2> <if-pred2>) ... (<predn> <if-predn>) [(else <else-expression>)])`

Must have at least one `<predn>` and `<if-predn>` and `[(else <else-expression>)]` is optional.

The rules for evaluation are as follows:

1. Evaluate the predicates `<pred1>`, `<pred2>`, ..., `<predn>` in order until you reach one that evaluates to a truth-y value.
2. If you reach a predicate that evaluates to a truth-y value, evaluate and return the corresponding expression in the clause.
3. If none of the predicates are truth-y and there is an else clause, evaluate and return `<else-expression>`.

`cond` is a special form because it does not evaluate its operands in their entirety; the predicates are evaluated separately from their corresponding return expression. In addition, the expression short circuits upon reaching the first predicate that evaluates to a truth-y value, leaving the remaining predicates unevaluated.

```
scm> (cond
        ((< 4 5) 1)
        (else 2)
      )
1
scm> (cond
        (#f (/ 1 0))
        (else 42)
      )
42
```

Let Expressions

A `let` expression looks like this: `(let ([binding_1] ... [binding_n]) <body> ...)`

Each `binding` corresponds to expressions of the form `(<name> <expression>)`.

Scheme evaluates a `let` expression using the following steps:

1. Create a new local frame that extends the current environment (in other words, it creates a new child frame whose parent is the current frame).
2. For each `binding` provided, bind each `name` to its corresponding evaluated `expression`.
3. Finally, the `body` expressions are evaulated in order in this new frame, returning the result of evaluating the last expression.

Note that bindings are optional within a `let` statement, but we typically include them.

```
scm> (let (
            (x 5)
            (y 10)
          )
      (print x)
      (print y)
      (- x y)
      (+ x y)
     )
5
10
15
```

Note that `(- x y)` in the body of this `let` expression *does* get evaluated, but the result doesn't get returned by the `let` expression because only the value of the *last* expression in the body, `(+ x y)`, gets returned. Thus, the interpreter does *not* display `-5` (the result of `(- x y)`).

However, we see that `5` and `10` *are* displayed out by the interpreter. This is because printing `5` and printing `10` were *side effects* of evluating the expressions `(print x)` and `(print y)`, respectively. `5` and `10` are *not* the return values of `(print x)` and `(print y)`.

Begin Expressions

A `begin` expression looks like this: `(begin <body_1> ... <body_n>)`

Scheme evaluates a `begin` expression by evaluating each `body` in order in the current environment, returning the result of evaluating the last `body`.

```
scm> (begin
        (print (< 2 3))
        (print 'hello)
        (+ 1 2)
        (- 5 7)
     )
#t
hello
-2
```

Again, note that `(+ 1 2)` does get evaluted, but the result, `3`, does not get returned by the `begin` expression (and thus does not get displayed by the interpreter) because it is not the last body expression.

Boolean operators

Like Python, Scheme has the boolean operators `and`, `or`, and `not`. `and` and `or` are special forms because they are short-circuiting operators, while `not` is a builtin procedure.

- `and` takes in any amount of operands and evaluates these operands from left to right until one evaluates to a false-y value. It returns that first false-y value or the value of the last expression if there are no false-y values.

- `or` also evaluates any number of operands from left to right until one evaluates to a truth-y value. It returns that first truth-y value or the value of the last expression if there are no truth-y values.
- `not` takes in a single operand, evaluates it, and returns its opposite truthiness value.

```
scm> (and 25 32)
32
scm> (or 1 (/ 1 0))    ; Short-circuits
1
scm> (not (odd? 10))
#t
```

Defining Functions

All Scheme procedures are constructed as lambda procedures.

One way to create a procedure is to use the `lambda` special form.

`(lambda (<param1> <param2> ...) <body>)`

This expression creates a lambda function with the given parameters and body, but does not evaluate the body. As in Python, the body is not evaluated until the function is called and applied to some argument values. The fact that neither the parameters nor the body is evaluated is what makes `lambda` a special form.

We can also assign the value of an expression to a name with a `define` special form:

1. `(define (<name> <param> ...) <body> ...)`
2. `(define <name> (lambda (<param> ...) <body> ...))`

These two expressions are equivalent; the first is a concise version of the second.

```
scm> ; Bind lambda function to square
scm> (define square (lambda (x) (* x x)))
square
scm> (define (square x) (* x x))          ; Same as above
square
scm> square
(lambda (x) (* x x))
scm> (square 4)
16
```

Pairs and Lists

All lists in Scheme are linked lists. Scheme lists are composed of two element pairs. We define a list as being either

- the empty list, `nil`
- a pair whose second element is a list

As in Python, linked lists are recursive data structures. The base case is the empty list.

We use the following procedures to construct and select from lists:

- `(cons first rest)` constructs a list with the given first element and rest of the list. For now, if `rest` is not a pair or `nil` it will error.
- `(car lst)` gets the first item of the list
- `(cdr lst)` gets the rest of the list

To visualize Scheme lists, you can use the `draw` function in code.cs61a.org.

```
scm> nil
()
scm> (define lst (cons 1 (cons 2 (cons 3 nil))))
lst
scm> lst
(1 2 3)
scm> (car lst)
1
scm> (cdr lst)
(2 3)
```

Scheme lists are displayed in a similar way to the Link class we defined in Python. Here is an example in 61A Code.

Two other ways of creating lists are using the built-in `list` procedure or the `quote` special form.

The `list` procedure has the syntax `(list <item> ...)`. It takes in an arbitrary number of operands and constructs a list with their values.

```
scm> (list 1 2 3)
(1 2 3)
```

The `quote` special form has the syntax `(quote <expression>)`. It returns the literal `expression` without evaluating it. A shorthand for the `quote` special form is `'<expression>`.

```
scm> (define a 61)
a
scm> a
61
scm> (quote a)
a
scm> 'a
a
```

We can use the `quote` form to create a list by passing in a combination as the `expression`:

```
scm> (quote (1 x 3))
(1 x 3)
scm> '(1 x 3)   ; Equivalent to the previous quote expression
(1 x 3)
```

An important difference between `list` (along with `cons`) and `quote` is that `list` and `cons` evaluate each of their operands before putting them into a list, while `quote` will return the list exactly as typed, without evaluating any of the individual elements.

```
scm> (define a 1)
a
scm> (define b 2)
b
scm> (list a b 3)
(1 2 3)
scm> '(a b 3)
(a b 3)
```

Note that if we wanted to create the list `(a b 3)` using the `list` procedure, we could quote the symbols `a` and `b` so that they are not evaluated when making the list:

```
scm> (list 'a 'b 3)
(a b 3)
```

CS 61A Fall 2023

Interpreters, More Scheme

Discussion 11: November 8, 2023

Note: For extended explanations of the concepts on this discussion, feel free to look at the **Appendix** section on the back of the worksheet.

Interpreters

Q1: From Pair to Expression

Recall that our Scheme interpreter is written in Python. The Python data structure that we use to represent Scheme lists is the `Pair` class, which is almost identical to the `Link` class you have already encountered.

Write out the Scheme expression with proper syntax that corresponds to the following `Pair` constructor calls.

```
>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
```

```
>>> Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```

```
>>> Pair('and', Pair(Pair('<', Pair(1, Pair(0, nil))), Pair(Pair('/', Pair(1, Pair(0, nil
    ))), nil)))
```

Scheme Eval/Apply

Scheme evaluates an expression by obeying the following evaluation rules:

- Call `scheme_eval` on the input expression
  - If the input expression is self-evaluating (e.g. number, boolean), output that value.
  - If the input expression is a special form, follow the evaluation rules for that special form.
  - Otherwise, the input expression is a call expression, and you follow the rules of evaluation for call expressions.

The tricky part of this is the evaluation rules for compound expressions (call expressions and special forms). As an example, let's look at the evaluation rules for call expressions.

Here's how you learned how to evaluate a Scheme call expression:

- Evaluate the operator to get a procedure.
- Evaluate each operand to get arguments.
- Apply the procedure to the arguments.

And here's how the Scheme interpreter handles it in Python:

- Call `scheme_eval` on the operator to get a procedure.
- Call `scheme_eval` on each operand to get arguments.
- Call `scheme_apply` to apply the procedure to the arguments.

Notice the similarities here? Every time we need to evaluate some subexpression in Scheme, the underlying Python implementation calls `scheme_eval` recursively on that call expression!

Let's see if you can extend this to other special forms.

Q2: Connecting Scheme to Python

Recall the rules for how you handle a `define` special form in Scheme:

> To evaluate `(define <symbol> <expr>)`
>
> - Evaluate `<expr>` to get a value.
> - Bind that value to `<symbol>` in the current frame.

Now write out how the Scheme interpreter handles `(define <symbol> <expr>)` using `scheme_eval` and/or `scheme_apply`.

Let's do `and` now. Write out the rules for how you handle an `and` special form in Scheme:

Now write out how the Scheme interpreter handles an `and` special form using `scheme_eval` and/or `scheme_apply`.

Q3: Counting Eval and Apply

How many calls to `scheme_eval` and `scheme_apply` would it take to evaluate each of the following expressions?

Take the following Scheme expression as an example: `(* 2 (+ 3 4))`

To evaluate this entire expression, we…

- Call `scheme_eval` on `(* 2 (+ 3 4))`
  - Call `scheme_eval` on `*`
  - Call `scheme_eval` on `2`
  - Call `scheme_eval` on `(+ 3 4)`
    - Call `scheme_eval` on `+`
    - Call `scheme_eval` on `3`
    - Call `scheme_eval` on `4`
    - Call `scheme_apply` to apply `+` to `(3 4)` → `(+ 3 4)` evaluates to `7`
  - Call `scheme_apply` to apply `*` to `(2 7)` → `(* 2 (+ 3 4))` evaluates to `14`

In total, there are 7 calls to `scheme_eval` and 2 calls to `scheme_apply` to get our final result of `14`. A visualization of these specific calls are shown below, where each underline is a call to `scheme_eval` and each overline is a call to `scheme_apply`:

```
scm> (+ 1 2)
```

For this particular prompt please list out the inputs to `scheme_eval` and `scheme_apply`.

```
scm> (+ 2 4 6 8)
```

```
scm> (+ 2 (* 4 (- 6 8)))
```

```
scm> (and 1 (+ 1 0) 0)
```

```
scm> (and (+ 1 0) (< 1 0) (/ 1 0))
```

More Scheme

Q4: Reverse

Write the procedure `reverse`, which takes in a list `lst` and outputs a reversed list.

Hint: you may find the built-in append procedure useful.

```
(define (reverse lst)
    'YOUR-CODE-HERE




)
```

Q5: Longest Increasing Subsequence

Write the procedure `longest-increasing-subsequence`, which takes in a list `lst` and returns the longest subsequence in which all the terms are increasing. *Note: the elements do not have to appear consecutively in the original list.* For example, the longest increasing subsequence of `(1 2 3 4 9 3 4 1 10 5)` is `(1 2 3 4 9 10)`. Assume that the longest increasing subsequence is unique.

Hint: The built-in procedures length and filter might be helpful to solving this problem.

```
; helper function
; returns the values of lst that are bigger than x
; e.g., (larger-values 3 '(1 2 3 4 5 1 2 3 4 5)) --> (4 5 4 5)
(define (larger-values x lst)
    _______________________________________________)

(define (longest-increasing-subsequence lst)
    (if (null? lst)
        nil
        (begin
            (define first (car lst))
            (define rest (cdr lst))
            (define large-values-rest
                (larger-values first rest))
            (define with-first
                _______________________________________________)
            (define without-first
                _______________________________________________)
            (if ___________________________________________
                with-first
                without-first))))

(expect (longest-increasing-subsequence '()) ())
(expect (longest-increasing-subsequence '(1)) (1))
(expect (longest-increasing-subsequence '(1 2 3)) (1 2 3))
(expect (longest-increasing-subsequence '(1 9 2 3)) (1 2 3))
(expect (longest-increasing-subsequence '(1 9 8 7 6 5 4 3 2 3)) (1 2 3))
(expect (longest-increasing-subsequence '(1 9 8 7 2 3 6 5 4 5)) (1 2 3 4 5))
(expect (longest-increasing-subsequence '(1 2 3 4 9 3 4 1 10 5)) (1 2 3 4 9 10))
```

Q6: Cons All

Implement `cons-all`, which takes in an element `first` and a list of lists `rests`, and adds `first` to the beginning of each list in `rests`:

```
scm> (cons-all 1 '((2 3) (2 4) (3 5)))
((1 2 3) (1 2 4) (1 3 5))
```

You may find it helpful to use the built-in map procedure.

```
(define (cons-all first rests)
  'YOUR-CODE-HERE


)
```

Q7: List Change

Implement the `list-change` procedure, which lists all of the ways to make change for a positive integer `total` amount of money, using a list of currency denominations, which is sorted in descending order. The resulting list of ways of making change should also be returned in descending order.

To make change for 10 with the denominations (25, 10, 5, 1), we get the possibilities:

```
10
5, 5
5, 1, 1, 1, 1, 1
1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

To make change for 5 with the denominations (4, 3, 2, 1), we get the possibilities:

```
4, 1
3, 2
3, 1, 1
2, 2, 1
2, 1, 1, 1
1, 1, 1, 1, 1
```

Therefore, your function should behave as follows for these two inputs

```
scm> (list-change 10 '(25 10 5 1))
((10) (5 5) (5 1 1 1 1 1) (1 1 1 1 1 1 1 1 1 1))
scm> (list-change 5 '(4 3 2 1))
((4 1) (3 2) (3 1 1) (2 2 1) (2 1 1 1) (1 1 1 1 1))
```

You may want to use `cons-all` in your solution.

You may also find the built-in append procedure useful.

```
;; List all ways to make change for TOTAL with DENOMS
(define (list-change total denoms)
  'YOUR-CODE-HERE









)
```

Appendix: Explanation of Material

Interpreters

An interpreter is a program that allows you to interact with the computer in a certain language. It understands the expressions that you type in through that language, and performs the corresponding actions in some way, usually using an underlying language.

In Project 4, you will use Python to implement an interpreter for Scheme. The Python interpreter that you've been using all semester is written (mostly) in the C programming language. The computer itself uses hardware to interpret machine code (a series of ones and zeros that represent basic operations like adding numbers, loading information from memory, etc).

When we talk about an interpreter, there are two languages at work:

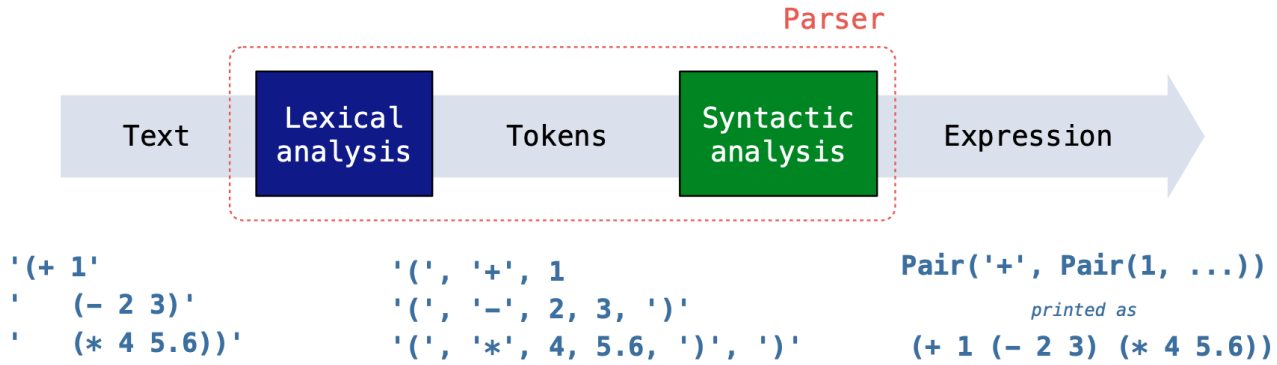
1. **The language being interpreted/implemented.** For Project 4, we are interpreting the Scheme language.
2. **The underlying implementation language.** For Project 4, we will implement an interpreter for Scheme using Python.

REPL

Many interpreters use a Read-Eval-Print Loop (REPL). This loop waits for user input, and then processes it in three steps:

- **Read:** The interpreter takes the user input (a string) and passes it through a parser. The parser processes the input in two steps:
  - The *lexical analysis* step turns the user input string into tokens that are like "words" of the implemented language. Tokens represent the **smallest** units of information.
  - The *syntactic analysis* step takes the tokens from the previous step and organizes them into a data structure that the underlying language can understand. For our Scheme interpreter, we create a `Pair` object (similar to a Linked List) from the tokens to represent the original call expression.
    - The first item in the `Pair` represents the operator of the call expression. The subsequent items are the operands of the call expression, or the arguments that the operator will be applied to. Note that operands themselves can also be nested call expressions.

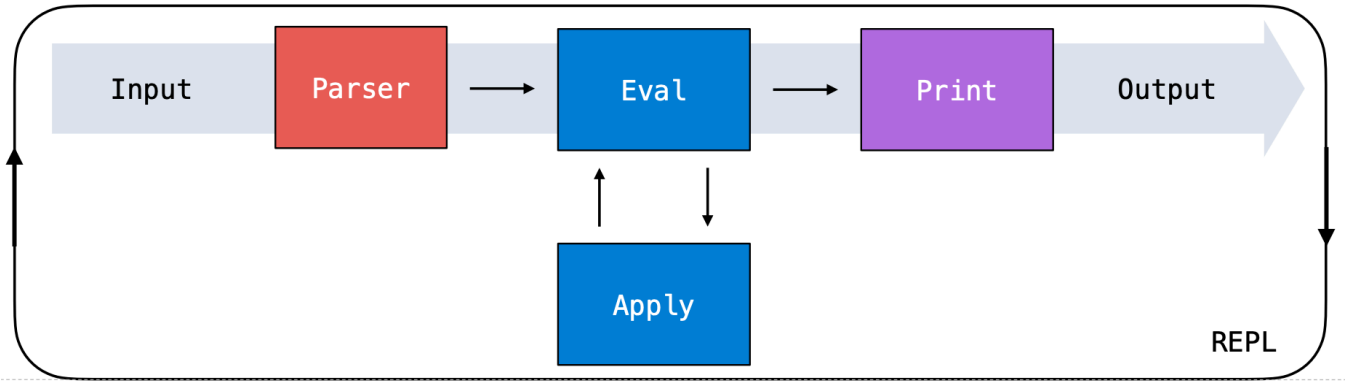
Below is a summary of the read process for a Scheme expression input:



- **Eval:** Mutual recursion between `eval` and `apply` evaluate the expression to obtain a value.
  - `eval` takes an expression and evaluates it according to the rules of the language. Evaluating a call expression involves calling `apply` to apply an evaluated operator to its evaluated operands.

  - `apply` takes an evaluated operator, i.e., a function, and applies it to the call expression's arguments. Apply may call `eval` to do more work in the body of the function, so `eval` and `apply` are *mutually recursive*.
- **Print:** Display the result of evaluating the user input.

Here's how all the pieces fit together:



CS 61A Fall 2023

Macros

Discussion 12: November 15, 2023

Note: For extended explanations of the concepts on this discussion, feel free to look at the **Appendix** section on the back of the worksheet.

Macros

- Evaluate the operator to get a macro.
- Apply the macro to the unevaluated operands. This involves the following steps.
  - Bind the unevaluated operands to the formal parameters in a new frame.
  - Evaluate each expression in the body of the macro using normal Scheme evaluation rules.
  - The value of the last expression is returned.
- Evaluate the expression produced by the macro in the frame it was called in.

Q1: Shapeshifting Expressions

When writing macros in Scheme, the goal is to create a list of symbols that represents a certain Scheme expression. In this question, we'll practice different methods of creating such Scheme lists.

We have executed the following code to define `x` and `y` in our current environment.

```
(define x '(+ 1 1))
(define y '(+ 2 3))
```

We want to use `x` and `y` to build a list that represents the following expression:

```
(begin (+ 1 1) (+ 2 3))
```

What would be the result of calling `eval` on a quoted version of the expression above?

```
(eval '(begin (+ 1 1) (+ 2 3)))
```

Now that we know what this expression should evaluate to, let's build our scheme list.

How would we construct the scheme list for the expression `(begin (+ 1 1) (+ 2 3))` using quasiquotation?

How would we construct this scheme list using the `list` procedure?

How would we construct this scheme list using the `cons` procedure?

Q2: WWSD: Macros

For each expression, write what the Scheme interpreter would output.

```
scm> (define-macro (f x) (car x))
```

```
scm> (f (2 3 4))
```

```
scm> (define x 2000)
```

```
scm> (f (x y z))
```

```
scm> (define-macro (g x) (+ x 2))
```

```
scm> (g 2)
```

```
scm> (g (+ 2 3))
```

```
scm> (define-macro (h x) (list '+ x 2))
```

```
scm> (h (+ 2 3))
```

```
scm> (define-macro (if-else-5 condition consequent) `(if ,condition ,consequent 5))
```

```
scm> (if-else-5 #f (/ 1 0))
```

```
scm> (if-else-5 (= 1 1) 2)
```

Q3: Mystery Macro

For this question, we'll consider the following macro:

```
(define-macro (mystery expr)
    `(let ((/ (lambda (a b) (if (= b 0) 1 (/ a b))))) ,expr))
```

What does this macro do?

Why can't we do the same thing with a regular procedure?

```
(define (mystery-proc expr)
    ... )
```

Professor Oppenheimer has written a procedure `letter-grade` to determine a student's letter grade on an assignment given a number of points `earned` and a number of points `possible`.

```
(define (letter-grade earned possible)
    (cond
        ((>= (/ earned possible) 0.9) 'A)
        ((>= (/ earned possible) 0.8) 'B)
        ((>= (/ earned possible) 0.7) 'C)
        ((>= (/ earned possible) 0.6) 'D)
        (else 'F)))
```

The procedure works well, but Professor Oppenheimer has noticed that some optional assignments have 0 points possible, which causes a division by zero error. Professor Oppenheimer wants to define `letter-grade` such that all students will receive an A for assignments with 0 points possible. Can you help him out using `mystery`?

```
(define-macro (mystery expr)
    `(let ((/ (lambda (a b) (if (= b 0) 1 (/ a b))))) ,expr))

(define (letter-grade earned possible)
    'YOUR-CODE-HERE









    )

; Tests
(expect (letter-grade 100 0) A)
(expect (letter-grade 95 100) A)
(expect (letter-grade 85 100) B)
(expect (letter-grade 75 100) C)
(expect (letter-grade 65 100) D)
(expect (letter-grade 55 100) F)
```

Q4: Max Macro

Define the macro `max`, which takes in two expressions `expr1` and `expr2` and returns the maximum of their values. If they have the same value, return the value of the first expression. **For this question, it's okay if your solution evaluates `expr1` and `expr2` more than once.** As an extra challenge, think about how you could use the `let` special form to ensure that `expr1` and `expr2` are evaluated only once.

```
scm> (max 5 10)
10
scm> (max 12 12)
12
scm> (max 100 99)
100
```

First, try using quasiquotation to implement this macro procedure.

```
(define-macro (max expr1 expr2)
    'YOUR-CODE-HERE


    )

; Test
(expect (max -3 (+ 1 2)) 3)
(expect (max 1 1) 1)
```

Now, try writing this macro using `list`.

```
(define-macro (max expr1 expr2)
    'YOUR-CODE-HERE


    )

; Test
(expect (max -3 (+ 1 2)) 3)
(expect (max 1 1) 1)
```

Finally, write this macro using `cons`.

```
(define-macro (max expr1 expr2)
    'YOUR-CODE-HERE









    )

; Test
(expect (max -3 (+ 1 2)) 3)
(expect (max 1 1) 1)
```

Reflect: was it necessary to use macros to do this? Or could we have done the same thing with a procedure?

Q5: Multiple Assignment

Recall that in Scheme, the expression returned by a macro procedure is evaluated in the environment that called the macro. This concept allows us to set variables in the calling environment using calls to macro procedures! This is not possible with regular scheme procedures because any `define` expressions would be evaluated in the procedure's environment (and thus bind a symbol in that environment rather than the calling environment). In this problem, we'll explore this idea in more detail.

In Python, we can bind two variables in one line as follows:

```
>>> x, y = 1, 2
>>> x
1
>>> y
2
>>> x, y = y, x # swap the values of x and y
>>> x
2
>>> y
1
```

The expressions on the right of the assignment are first evaluated, then assigned to the variables on the left. Let's try to implement a similar feature in Scheme using macros.

Write a macro `multi-assign` which takes in two symbols `sym1` and `sym2` as well as two expressions `expr1` and `expr2`. It should bind `sym1` to the value of `expr1` and `sym2` to the value of `expr2` in the environment from which the macro was called.

```
scm> (multi-assign x y 1 (- 3 1))
scm> x
1
scm> y
2
```

First, implement a version of `multi-assign` which evaluates `expr1` first, binds it to `sym1`, then evaluates `expr2` and binds it to `sym2` (the order here is important).

```
(define-macro (multi-assign sym1 sym2 expr1 expr2)
    `(_________ (define ____________ ____________) (define ____________ ____________)
    undefined)
)

; Tests
(multi-assign x y 1 2)
(expect (= x 1) #t)
(expect (= y 2) #t)
```

This solution is great, but it doesn't quite behave in quite the same way that it does in Python:

```
scm> (multi-assign x y 1 (+ 2 3))
scm> x
1
scm> y
5
scm> (multi-assign x y y x)
scm> x
5
scm> y
5
```

Notice that `x` and `y` were not swapped like we wanted. This is because of the order of evaluation and bindings: first, the value of `y` is bound to `x`. Afterwards, `x` is evaluated and bound to `y`, but at this point, `x` no longer has its old value, it is actually the value of `y`!

Now, try writing a version of `multi-assign` which matches the behavior in Python, i.e. `expr1` and `expr2` should be both evaluated before being assigned to `sym1` and `sym2`.

```
scm> (multi-assign x y 5 6)
scm> x
5
scm> y
6
scm> (multi-assign x y y x)
scm> x
6
scm> y
5
```

```
(define-macro (multi-assign sym1 sym2 expr1 expr2)
    `(_________ (define ,sym2 (list ______________________________________))
            (define __________ ______________________________________________)
            (define __________ ______________________________________________)
            undefined)
)

; Tests
(multi-assign x y 1 2)
(expect (= x 1) #t)
(expect (= y 2) #t)
(multi-assign x y y x)
(expect (= x 2) #t)
(expect (= y 1) #t)
```

Q6: Replace

Write the macro `replace`, which takes in a Scheme expression `expr`, a Scheme symbol or number `old`, and a Scheme expression `new`. The macro replaces all instances of `old` with `new` before running the modified code.

```
(define (replace-helper e o n)
  (if ___________________________________
      ____________________________________________________________________
      ____________________________________________________________________))
(define-macro (replace expr old new)
    (replace-helper expr old new))

; Tests
(expect (replace (define x 2) x y) y)
(expect (= y 2) #t)
(expect (replace (+ 1 2 (or 2 3)) 2 0) 1)
```

Appendix: Explanation of Material

Macros

So far we've been able to define our own procedures in Scheme using the `define` special form. This doesn't allow us to do anything, however. Consider what happens when we try to write a function that evaluates a given expression twice:

```
scm> (define (twice expr) (begin expr expr))
twice
scm> (twice (print 'woof))
woof
```

Since `twice` is a procedure, we evaluate its call expression by first evaluating the operator and then each operand. That means that `woof` was printed when we evaluated the operand `(print 'woof)`. Inside the body of `twice`, the name `expr` is bound to the value `undefined`, so the expression `(begin expr expr)` does nothing at all!

The problem here is clear: we need to prevent the given expression from evaluating until we're inside the body of the procedure. Wouldn't it be cool if we could define our own special forms where we could avoid such the pitfalls of call expressions? This is where the `define-macro` special form, which has identical syntax to the regular `define` form, comes in:

```
scm> (define-macro (twice expr) (list 'begin expr expr))
twice
```

`define-macro` allows us to define what's known as a `macro`, which is simply a way for us to process unevaluated input expressions together into another expression. The rules for evaluating a macro expression are:

- Evaluate the operator to get a macro.
- Apply the macro to the unevaluated operands. This involves the following steps.
  - Bind the unevaluated operands to the formal parameters in a new frame.
  - Evaluate each expression in the body of the macro using normal Scheme evaluation rules.
  - The value of the last expression is returned.
- Evaluate the expression produced by the macro in the frame it was called in.

Note the key differences here between macros and regular procedures: the operands are not evaluated before being passed in to the macro. Additionally, the output of the body of the macro is evaluated after.

To evaluate `(twice (print 'woof))`:

- We evaluate `twice`, which evaluates to a macro procedure.
- We apply that macro procedure to the unevaluated operands:
  - Bind `(print 'woof)` to `expr` in a new frame.
  - Evaluate the body of the macro: `(list 'begin expr expr)` evaluates to `(begin (print 'woof) (print 'woof))`.
  - Return `(begin (print 'woof) (print 'woof))`
- Evaluate `(begin (print 'woof) (print 'woof))` in the current frame. `(print 'woof)` is evaluated twice, and woof is printed twice.

CS 61A Fall 2023

SQL, Final Review

Discussion 13: November 29, 2023

Note: For extended explanations of the concepts on this discussion, feel free to look at the **Appendix** section at the end of the worksheet.

SQL

SQL is a declarative programming language. Statements do not describe computations directly, but instead describe the desired result of some computation. It is the role of the query interpreter of the database system to plan and perform a computational process to produce such a result.

> For this discussion, you can test out your code at sql.cs61a.org. The `records` table should already be loaded in.

Select Statements

The following questions involve the `records` table:

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
...	...	...	...	...
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

To see the entire `records` table, go to sql.cs61a.org and enter `SELECT * FROM records;`. Note that you can answer the questions without seeing the entire table.

Q1: Oliver Employees

Write a query that outputs the names of employees that Oliver Warbucks directly supervises.

```
SELECT "YOUR CODE HERE"
```

Q2: Self Supervisor

Write a query that outputs all information about employees that supervise themselves.

```
SELECT "YOUR CODE HERE"
```

Q3: Rich Employees

Write a query that outputs the names of all employees with salary greater than 50,000 in alphabetical order.

```
SELECT "YOUR CODE HERE"
```

Joins

The following questions involve the `records` and `meetings` tables:

records

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
…	…	…	…	…
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

meetings

Division	**Day**	**Time**
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

Q4: Oliver Employee Meetings

Write a query that outputs the meeting days and times of all employees directly supervised by Oliver Warbucks.

```
SELECT "YOUR CODE HERE"
```

Q5: Different Division

Write a query that outputs the names of employees whose supervisor is in a different division.

```
SELECT "YOUR CODE HERE"
```

Q6: Middle Manager

A middle manager is a person who is both supervising someone and is supervised by someone different. Write a query that outputs the names of all middle managers.

```
SELECT "YOUR CODE HERE"
```

Aggregation

The following questions involve the `records` and `meetings` tables:

records

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
…	…	…	…	…
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

meetings

Division	**Day**	**Time**
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

Q7: Supervisor Sum Salary

Write a query that outputs each supervisor and the sum of salaries of all the employees they supervise.

```
SELECT "YOUR CODE HERE"
```

Q8: Num Meetings

Write a query that outputs the days of the week for which fewer than 5 employees have a meeting. You may assume no department has more than one meeting on a given day.

```
SELECT "YOUR CODE HERE"
```

Q9: Rich Pairs

Write a query that outputs all divisions for which there is more than one employee, and all pairs of employees within that division that have a combined salary less than 100,000.

```
SELECT "YOUR CODE HERE"
```

Final Review
Recursion, Sequences

Q10: Subsequences

A subsequence of a sequence `S` is a subset of elements from `S`, in the same order they appear in `S`. Consider the list `[1, 2, 3]`. Here are a few of its subsequences `[]`, `[1, 3]`, `[2]`, and `[1, 2, 3]`.

Write a function that takes in a list and returns all possible subsequences of that list. The subsequences should be returned as a list of lists, where each nested list is a subsequence of the original input.

In order to accomplish this, you might first want to write a function `insert_into_all` that takes an item and a list of lists, adds the item to the beginning of each nested list, and returns the resulting list.

```
def insert_into_all(item, nested_list):
    """Return a new list consisting of all the lists in nested_list,
    but with item added to the front of each. You can assume that
     nested_list is a list of lists.

    >>> nl = [[], [1, 2], [3]]
    >>> insert_into_all(0, nl)
    [[0], [0, 1, 2], [0, 3]]
    """
    "*** YOUR CODE HERE ***"



def subseqs(s):
    """Return a nested list (a list of lists) of all subsequences of S.
    The subsequences can appear in any order. You can assume S is a list.

    >>> seqs = subseqs([1, 2, 3])
    >>> sorted(seqs)
    [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
    >>> subseqs([])
    [[]]
    """
    if _________________:
        _________________
    else:
        _________________
        _________________
```

Trees

Q11: Long Paths

Implement `long_paths`, which returns a list of all *paths* in a tree with length at least `n`. A path in a tree is a linked list of node values that starts with the root and ends at a leaf. Each subsequent element must be from a child of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are listed in order from left to right. See the doctests for some examples.

```
def long_paths(tree, n):
    """Return a list of all paths in tree with length at least n.

    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])])])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    <0 1 2>
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 6 9>
    <0 11 12 13 14>
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 11 12 13 14>
    >>> long_paths(whole, 4)
    [Link(0, Link(11, Link(12, Link(13, Link(14)))))]
    """
    "*** YOUR CODE HERE ***"
```

Linked Lists

Q12: Deep Linked List Length

A linked list that contains one or more linked lists as elements is called a *deep* linked list. Write a function `deep_len` that takes in a (possibly deep) linked list and returns the *deep length* of that linked list. The deep length of a linked list is the total number of non-link elements in the list plus the total number of elements contained in all contained lists. See the function's doctests for examples of the deep length of linked lists.

Hint: Use `isinstance` to check if something is an instance of an object.

```
def deep_len(lnk):
    """ Returns the deep length of a possibly deep linked list.

    >>> deep_len(Link(1, Link(2, Link(3))))
    3
    >>> deep_len(Link(Link(1, Link(2)), Link(3, Link(4))))
    4
    >>> levels = Link(Link(Link(1, Link(2)), \
            Link(3)), Link(Link(4), Link(5)))
    >>> print(levels)
    <<<1 2> 3> <4> 5>
    >>> deep_len(levels)
    5
    """
    if _______________:
        return 0
    elif _______________:
        return 1
    else:
        return ____________________________
```

Generators

Q13: Powers

Implement `powers`, a generator function that takes positive integers `n` and `k`. It yields all integers `m` that are both powers of `k` and whose digits appear in order in `n`.

Assume the following functions are defined:

- `is_power(base, s)`: `is_power` takes in a positive integer `base` and a non-negative integer `s` and returns `True` if there is some number `n` where `pow(base, n) = s`
- `curry2`: `curry2 = lambda f: lambda x: lambda y: f(x, y)`

Hint: `filter(func, seq)` returns an iterator that yields all the values `x` in `seq` where `func(x)` is truthy.

```
def powers(n, k):
    """Yield all powers of k whose digits appear in order in n.

    >>> sorted(powers(12345, 5))
    [1, 5, 25, 125]
    >>> sorted(powers(54321, 5))  # 25 and 125 are not in order
    [1, 5]
    >>> sorted(powers(2493, 3))
    [3, 9, 243]
    >>> sorted(powers(2493, 2))
    [2, 4]
    >>> sorted(powers(164352, 2))
    [1, 2, 4, 16, 32, 64]
    """
    def build(seed):
        """Yield all non-negative integers whose digits appear in order in seed.
        0 is yielded because 0 has no digits, so all its digits are in seed.
        """
        if seed == 0:
            yield 0
        else:
            for x in _________________:
                ----------------
                ----------------
    yield from filter(curry2(_________)(_________), build(n))
```

Scheme

Q14: Group by Non-Decreasing

Define a function `nondecreaselist` that takes in a scheme list of numbers and outputs a list of lists, which overall has the same numbers in the same order, but grouped into lists that are non-decreasing.

For example, if the input is a stream containing elements

```
(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1)
```

the output should contain elements

```
((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1))
```

Note: The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
(define (nondecreaselist s)

    (if _______________________________
        _______________________________
        (let ((rest _______________________________ ))
            (if _______________________________
                (cons _______________________________ rest)
                (cons _______________________________ (cdr rest))
            )
        )
    )
)

(expect (nondecreaselist '(1 2 3 1 2 3)) ((1 2 3) (1 2 3)))

(expect (nondecreaselist '(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1))
        ((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1)))
```

Programs as Data

Q15: Or with Multiple Args

Recall `make-or` from Homework 9. Implement `make-long-or`, which returns, as a list, a program that takes in any number of expressions and `or`'s them together (applying short-circuiting rules). This procedure should do this without using the `or` special form. Unlike the `make-or` procedure from Homework 9, the arguments will be passed in as a list named `args`.

The behavior of the `or` procedure is specified by the following doctests:

```
scm> (define or-program (make-long-or '((print 'hello) (/ 1 0) 3 #f)))
or-program
scm> (eval or-program)
hello
scm> (eval (make-long-or '((= 1 0) #f (+ 1 2) (print 'goodbye))))
3
scm> (eval (make-long-or '((> 3 1))))
#t
scm> (eval (make-long-or '()))
#f
```

```
(define (make-long-or args)
    'YOUR-CODE-HERE









)
```

Appendix: Explanation of Material

Select Statements

A `SELECT` statement creates a table. The following statement creates a single-row table with columns named `first` and `last`.

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last;
Ben|Bitdiddle
```

A `UNION` statement creates a table that consists of the rows of two tables with the same number of columns.

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last UNION
...> SELECT "Louis", "Reasoner";
Ben|Bitdiddle
Louis|Reasoner
```

A `FROM` clause specifies an existing table from which information can be drawn. The following statement creates a table that consists of the `name` and `division` columns from an existing table `records`.

```
sqlite> SELECT name, division FROM records;
Alyssa P Hacker|Computer
...
Robert Cratchet|Accounting
```

The special syntax `SELECT *` selects all columns from an existing table. It's an easy way to display the contents of a table.

```
sqlite> SELECT * FROM records;
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
...
Robert Cratchet|Accounting|Scrivener|18000|Eben Scrooge
```

The general syntax of a `SELECT` statement is as follows:

```
SELECT [columns] FROM [tables]
WHERE [condition] ORDER BY [columns] LIMIT [limit];
```

- `SELECT [columns]`
  - Specifies the columns of our output table; `[columns]` is a comma-separated list of column names, and `*` selects all columns
- `FROM [tables]`
  - Specifies the existing tables from which we select columns; `[tables]` is a comma-separated list of table names
- `WHERE [condition]`
  - Filters the output table to include only rows which satisfy the `[condition]`, a boolean expression
- `ORDER BY [columns]`
  - Orders the rows of the output table by the given comma-separated list of columns
- `LIMIT [limit]`
  - Limits the number of rows in the output table to the integer `[limit]`

The following `SELECT` statement lists all information about employees with the "Programmer" title.

```
sqlite> SELECT * FROM records WHERE title = "Programmer";
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
Cy D Fect|Computer|Programmer|35000|Ben Bitdiddle
```

The following `SELECT` statement lists the names and salaries of each employee in the accounting division, sorted in descending order by their salaries.

```
sqlite> SELECT name, salary FROM records
...> WHERE division = "Accounting" ORDER BY salary DESC;
Eben Scrooge|75000
Robert Cratchet|18000
```

All valid SQL statements are terminated by a semicolon.

An SQL statement may have any number of line breaks and any amount of whitespace. But keep in mind that consistent line-breaking and indentation make your code a lot easier to read!

Joins

Joining tables is a fundamental database operation.

So far, we've been able to examine and filter through the information in individual rows. But what if we want to reveal relationships between rows of the same table or with information in a different table? The tool we use for this is called a join, which involves considering every possible combination of rows from multiple tables. In SQL, a join is specified by a comma-separated list of input tables in the `FROM` clause of a `SELECT` statement.

For example, suppose we have a `meetings` table that records divisional meetings.

Division	Day	Time

Division	Day	Time
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

The following statement joins the `records` table and the `meetings` table:

```
sqlite> SELECT * FROM records, meetings;
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle|Accounting|Monday|9am
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle|Computer|Wednesday|4pm
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle|Administration|Monday|11am
...
Robert Cratchet|Accounting|Scrivener|18000|Eben Scrooge|Administration|Monday|11am
Robert Cratchet|Accounting|Scrivener|18000|Eben Scrooge|Administration|Wednesday|4pm
```

There is one row in the joined table for each possible pair of a row from `records` and a row from `meetings`. Because `records` has 5 columns and `meetings` has 3 columns, the joined table has $5 + 3 = 8$ columns. The columns are named `Name`, `Division`, `Title`, `Salary`, `Supervisor`, `Division`, `Day`, `Time`. Because `records` has 9 rows and `meetings` has 4 rows, there are $9 * 4 = 36$ possible pairs of rows between the two tables; uncoincidentally, the joined table has 36 rows.

Sometimes, we join tables with overlapping column names. When this happens, we need to be able to disambiguate column names. The `AS` keyword gives an alias to a table listed in a `FROM` clause. Then we can use a dot expression to refer to a column in that table.

The following statement finds the name and title of Louis Reasoner's supervisor.

```
sqlite> SELECT b.name, b.title FROM records AS a, records AS b
...> WHERE a.name = "Louis Reasoner" AND
...> a.supervisor = b.name;
Alyssa P Hacker|Programmer
```

Aggregation

An aggregate function condenses information from multiple rows of a table into a single row. Some aggregate functions are `MAX`, `MIN`, `COUNT`, and `SUM`.

If we wanted to find the name and salary of the employee who makes the most money, we might type:

```
sqlite> SELECT name, MAX(salary) FROM records;
Lana Lambda|610000
```

The special syntax `COUNT(*)` counts the number of rows in a table. We can count the number of rows in `records` (which is the number of employees at our company).

```
sqlite> SELECT COUNT(*) FROM records;
9
```

The `GROUP BY [column name]` clause groups together all rows that have the same value in `[column name]`. Then aggregation is performed on each group so that there is exactly one row in the output table for each group.

The following statement finds the minimum salary earned in each division of our company.

```
sqlite> SELECT division, MIN(salary) FROM records GROUP BY division;
Computer|25000
Administration|25000
Accounting|18000
```

The `HAVING [condition]` clause filters the output table to include only the groups that satisfy the `[condition]`.

If we wanted to find all titles that are held by more than one person, we might type:

```
sqlite> SELECT title FROM records GROUP BY title HAVING COUNT(*) > 1;
Programmer
```