A Brief Compilation

of Guides, Analyses, and Problems

Structure and Interpretation of Computer Programs

at the University of California, Berkeley

Alvin Wan

Contents

0.1 Purpose

This compendium is (unofficially) written for the Fall 2015 CS61A: Structure and Interpretation of Computer Programs class taught by Professor John DeNero at UC Berkeley. It's primary purpose is to offer additional practice problems and walkthroughs, as a supplement to Professor DeNero's textbook Composing Programs (composingprograms.com).

0.1.1 Structure

Each chapter is structured so that this book can be read on its own (with sufficient Python knowledge). A minimal guide at the beginning of each section covers essential materials and misconceptions but does not provide a comprehensive overview. Each guide is then followed by at least 5 practice problems, first provided as just problem statements, then provided as templates, as they are on CS61A exams.

0.1.2 Breakdown

For the most part, guides are simply summaries of select chapters from Composing Programs.
For more difficult parts of the course, guides may be accompanied by breakdowns and analyses of problem types that were otherwise not introduced in the course. These additional "Analysis" sections will attempt to provide a more regimented approach to solving complex problems.
Except for the first problem of each section, problems are written at about exam-level, err'ing on the side of difficulty where needed. All problems have have accompanying, working scripts at `github.com/alvinwan/abcSICP` and have been tested using UniExpect, a doctest-esque module for testing in languages like Scheme and SQL (`github.com/alvinwan/UniExpect`).

0.1.3 Resources

Additional resources, including 3 practice exams, 10 quizzes, and other random worksheets and problems are posted online at `alvinwan.com/cs61a`.

Chapter 1

Higher-Order Functions

1.1 Guide

Higher-order functions manifest themselves in three ways: functions as arguments, nested definitions, and functions as return values.

For this guide's original, full text, see Composing Programs 1.6.

1.1.1 Functions as Arguments

Using functions as arguments allows us to abstract functionality. In the following example, an `evaluator` accepts the function representation of a polynomial, and evaluates that polynomial for each x in a list.

```
def evaluator(polynomial, xs=(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)):
  """Evaluates a polynomial at all xs"""
  return [(x, polynomial(x)) for x in xs]
```

This allows us to grab the coordinates of any polynomial written as a function of x.

```
y1 = lambda x: 0.5*x + 3 # line
y2 = lambda x: (x - 3)**2 + 1 # parabola
y3 = lambda x: x**4 + 2*x**2 - 3 # random polynomial
```

We now have a simple way of evaluating any polynomial at an arbitrary n points, where `n=len(xs)`.

1.1.2 Nested Definitions

Functions defined within functions allow us to abstract away a function's inner workings, hiding it from the global frame. In the next chapter, we will see that nested definitions additionally allow us to take advantage of recursion without sacrificing code cleanliness (i.e., defining too many functions in the global frame).

The following is a naïve implementation of selection sort.

```
def selection_sort(lst):
  """Sorts a list of comparables in-place."""

  def min_index(lst, start):
    """Returns index of minimum value after the index 'start'."""
    assert 0 <= start < len(lst), 'Invalid index'
    j, biggest, k = start, lst[start], start
    while k < len(lst):
      if lst[k] > biggest:
        j, biggest = k, lst[k]
      k += 1
    return k

  for i, n in enumerate(lst):
    j = min_index(lst, i)
    lst[i], lst[j] = lst[j], lst[i]
```

By defining `min_index` inside of `selection_sort`, we avoid globally defining a function that is specific to `selection_sort`. More importantly, by working in a smaller namespace (within the scope of `selection_sort`), we lessen the risk of overriding an existing function with different functionality.

1.1.3 Functions as Return Values

Functions as return values allow us to generate highly similar and marginally different functions. Take all fifth-degree polynomials, for example. Each naturally follows the same construct $ax^5+bx^4+cx^3+dx^2+ex+f$. Programming a series of lambda functions that each take nearly-identical forms would be tedious.

```
y1 = lambda x: 5*x**5 + 52*x**4 + 7*x**3 + 2*x**2 + 3*x + 7
y2 = lambda x: 10*x**5 + 7*x**4 + 3*x**3 + 11*x**2 + 2*x + 3
y3 = lambda x: 3*x**5 + 6*x**4 + 22*x**3 + 2*x**2 + 41*x + 2
...
```

Instead, we can define a `deg5_generator`, which takes 6 arguments: 1 for each coefficient and 1 additional for the extra constant.

```
def deg5poly(a, b, c, d, e, f):
  """Returns a polynomial as a function with one parameter x."""
  return lambda x: a*x**5 + b*x**4 + c*x**3 + d*x**2 + e*x + f
```

Now, we can shorten our polynomial definitions.

```
y1 = deg5poly(5, 52, 7, 2, 3, 7)
y2 = deg5poly(10, 7, 3, 11, 2, 3)
y3 = deg5poly(3, 6, 22, 2, 41, 2)
...
```

1.2 Problems

1. **Warm up** Without using for, while, or slicing, reverse a list. You should use the built-in `map` function.

2. Replace each digit in a number with its distance – in digits – from the first 1 to come **after** it. Any number that does not have a 1 after it will be replaced by a 0, and two digits side-by-side have a distance of 1. Assume there is no distance greater than 9.

3. Write a function `next_look_and_say(n)` which returns the next number of the "look-and-say" sequence given the previous term `n`. The first term is 1. To generate a term in the sequence, look at the previous term and read it. 1121 is "two 1s one 2 one 1", which translates into 211211.

4. Crypto is a puzzle game, where players receive a total and a series of numbers. Assume we can only use multiply, divide, add, and subtract, and that order of operations doesn't apply ('4+4/2'=4). Write `crypto_solver`, which finds all possible permutations of mathematical operations that will yield the total. For example, given `nums=[6, 2, 2]`, `total=10`, return `'6+2+2', '6*2-2'`. Order of the results does not matter.

5. Write a function that replaces each element of a list with the sum of the two largest numbers after it. Use -1 if there are not enough numbers.

6. Write a function `matrix_product` that multiplies two matrices `A` and `B` together. If matrix multiplication is impossible, raise an error. Recall that the number of columns in the first matrix must equal the number of rows in the second matrix.

Chapter 2

Recursion

2.1 Guide

Recursive functions are functions that invoke themselves, either indirectly or directly. Before writing any recursive function, ask yourself the following three questions:

1. What is the base case?
2. How does the recursive call break down the problem into simpler parts?
3. What is the recursive case?

For this guide's original, full text, see Composing Programs 1.7.

2.1.1 Mutual Recursion

Mutual recursion occurs when two functions recursively call each other. This can be useful for maintaining abstraction barriers inside of a program.
Consider a program that, given the amount of time in a class period, prints the interaction between a teacher and a student, up until the bell.

```
def ask(t):
  """Student takes 3 seconds to ask a question."""
  if t <= 0:
   print('Ask me after class!')
  print('Asking...')
  answer(t-3)

def answer(t):
  """Teacher takes 20 seconds to answer."""
  if t <= 0:
    print('Stay after for a full explanation!')
  print('Answering...')
  ask(t-20)
```

Notice that these function abstract away the notion of asking from the notion of answering.

2.1.2 Tree Recursion

Tree recursion occurs when a function calls itself more than once. This results in a "tree" of function calls, where each function call branches into a few more. We can benefit from this type recursion most when finding permutations or combinations.
Consider a staircase with `n` steps. We can choose to take 1 or 2 steps at a time. How many ways can we walk these stairs?

```
def stairs(n):
  """Number of ways to walk n steps, if you can choose to take 1 or 2
  steps at a time"""
  if n <= 2:
    return n
  return stairs(n-2) + stairs(n-1)
```

Let us break down this problem by answering the three questions we ask ourselves for any recursion problem.

1. **What is the base case?** We consider `n=0`, `n=1`, or `n=2` steps. If `n=0`, there are 0 ways we can walk the stairs. If `n=1`, we can only walk it one way: take the step. If `n=2`, we have two ways: take one step at a time, or take both at once. Combining all of these base cases, it just so happens that if `n<=2`, we have exactly `n` ways of walking these steps.

2. **How does the recursive call break down the problem into simpler parts?** Let us consider `n=3`. In this case, we know that our last move covers either 2 steps or 1. As a consequence, we know that all the ways to traverse 3 steps is all the ways to traverse 1 step and all the ways to traverse 2 steps (`3-2` and `3-1`, respectively). Our recursive case is the sum of all the ways to traverse `3-1` and `3-2` steps.

3. **What is the recursive case?** This generalizes fairly well. For any `n` steps, we sum all the ways to traverse `n-2` and `n-1` steps. As it turns out, we have a function that tells us how many ways to traverse `n` steps: `stairs`, the very function we're writing. So, our recursive call is `stairs(n-1) + stairs(n-2)`.

2.1.3 Cascade

The cascade is a visualization of recursive calls.

```
def cascade(n, depth=0):
  """Visualization for cascade"""
  if n > 0:
    print(' ' * depth, 'cascade({})'.format(n))
    cascade(n-1, depth+1)
    print(' ' * depth, 'return')
```

Running `cascade(5)`, for example, will yield a cascade of depth 5. In the following Analysis section of chapter 3, we will discuss how to take advantage of this visualization.

2.2 Analysis

2.2.1 How do I even start?

Recursion questions are generally and initially daunting. The question is: *How do I even start?* First, answer the three questions specified in the introduction section of this chapter.

1. What is the base case?
2. How does the recursive call break down the problem into simpler parts?
3. What is the recursive case?

If the recursive case is particularly complex, try additionally asking yourself the following questions. These questions do not cover all possibilities with recursion but may get you started in the right direction.

1. **What data does each recursive call need?** Is the question asking for distance? The index of a particular value? Its depth? Determine what data each recursive call requires to complete its task. If the function is only responsible for changing values, think about what the function needs to compute that new value.
2. **Which leg of the cascade?** Think about which direction data is moving. This can be a particularly useful tool for visualizing recursion. See section 3.2.1 for more information about the cascade.
3. **What type of recursion is this?** Is this mutual or tree recursion? Is it "standard" recursion? Determine if you can break one recursive call into smaller subsets of the same problem. Consider if you're dealing with all combinations or permutations of a sequence. If either is true, the problem could benefit from tree recursion.

Make sure to take time addressing some if not all of these questions, before coding. Using your responses to the above questions, you should have some idea of how the problem works. If not, try guessing and checking. Ultimately, writing and testing code is preferable to sitting and staring.

2.3 Problems

1. **Warm up** Write `stairs(n)`, which gives the number of ways to take `n` steps, given that at each step, you can choose to take 1, 2, or 3 steps. `stairs(5) = 13, stairs(10) = 274`

2. Write `stairs(n, k)`, which gives the number of ways to take `n` steps, given that at each step, you can choose to take 1, 2, ... k-2, k-1 or k steps. `kstairs(5, 2) = 8, stairs(5, 5) = 16, stairs(10, 5) = 464`

3. Compute the determinant of a matrix `A` using co-factor expansion. You may not assume a fixed size for the matrix; `A` may be any n x n matrix where $n \in \mathbb{N}$ (i.e., where n is a natural number).

4. Write a function `permutations(lst)` that returns a list of all permutations of the provided `lst`.

5. Given a tuple of numbers, where each number represents the size of a slice of pie, distribute the slices among 2 people as evenly as possible. (i.e., minimizing the difference between the sums of two sets of values).

Chapter 3

Data Structures

3.1 Guide

This guide will only explore two data structures: Linked Lists and Trees. In here, we will use conventions from Composing Programs.

For this guide's original, full text, see Composing Programs 2.9.

3.1.1 Linked Lists

Linked lists are comprised of a value (called `first`) and a `rest` pointer. `rest` will point to either `Link.empty`, denoting the end of a linked list, or to the head of a linked list. Consider the following definition of a `Link`.

```
class Link:
  """Linked list with a value (first) and a next pointer (rest)"""
  empty = ()
  def __init__(self, first, rest=empty):
    self.first = first
    self.rest = rest
```

List construction is also recursive. Consider the following piece of code that generates a linked list of the first 5 numbers in the naturals. Each link is passed into the constructor of the previous link as `rest`: `Link(1, Link(2, Link(3, Link(4, Link(5)))))`
We can traverse this list either iteratively or recursively. The following is a commonly-used *iterative* approach to linked list traversal.

```
def print_list_iter(link):
  while link.rest is not Link.empty:
    print(link.first)
    naturals = link.rest
```

The following is a commonly-used *recursive* approach to linked list traversal.

```
def print_list_rec(link):
```

```
  if link is Link.empty:
   return
  print(link.first)
  return print_list_rec(link.rest)
```

3.1.2 Trees

Trees are similarly comprised of a value (called `entry`) and a list of `branches`. `branches` is either empty, denoting a *leaf* (a tree without branches), or a list of trees. Consider the following definition of a `Tree`.

```
class Tree:
  """Tree with a value (entry) and a list of subtrees (branches)"""
  def __init__(self, entry, branches=()):
    self.entry = entry
    self.branches = branches
```

Just like list construction, tree construction is recursive. Consider the following piece of code that generates a tree containing the first 5 natural numbers: `Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3)])`
We can traverse this tree either iteratively or recursively. *Note that the two iterative traversals and their associated data structures discussed below are outside the scope of CS61A.* The iterative approach may take one of two forms: *depth-first traversal* or *breadth-first traversal*. With depth-first traversal, use a *stack*, which first removes the last item added to the stack. With breadth-first traversal, use a *queue*, which first removes the first item added to the stack.

```
def dfs(tree):
  """Traverses a tree depth-first, using a stack."""
  trees = [tree]
  while trees:
    tree = trees.pop() # pop from the end
    print(tree.entry)
    trees.extend(tree.branches) # tree.branches[::-1] to match dfs_rec

def bfs(tree):
  """Traverses a tree breadth-first, using a queue."""
  trees = [tree]
  while trees:
    tree = trees.pop(0) # pop from the beginning
    print(tree.entry)
    trees.extend(tree.branches)
```

The following is a commonly-used *recursive* approach to tree traversal. Notice that the recursive approach shown here is a depth-first traversal.

```
def dfs_rec(tree):
  print(tree.entry)
  return [dfs_rec(subtree) for subtree in tree.branches]
```

3.2 Analysis

3.2.1 Which leg of the cascade?

First, let us remind ourselves of the `cascade` function from section 2.1.3. This function will generate the cascading effect.

```
def cascade(n, depth=0):
  """Visualization for cascade"""
  if n > 0:
    print(' ' * depth, 'cascade({})'.format(n))
    cascade(n-1, depth+1)
    print(' ' * depth, 'return')
```

Then, let us *see* the cascade. This is the output of `cascade` using `cascade(3)`. Notice the sideways v-shape.

```
cascade(3) # deeper
 cascade(2) # deeper
  cascade(1) # deepest!
  return # rising
 return # rising
return # made it out!
```

You'll notice that on the *descending leg*, each function call brings us deeper into the stack of frames. On the *ascending leg*, each `return` brings us one step higher, decreasing depth. These are not formal terms that describe the recursion process; it is simply jargon we can use to discuss this effect.

In general, the descending leg is associated with progression deeper into a data structure, and the ascending leg is associated with the return back. For data structures that we've explored, this means the following:

- **Linked List** The *descending leg* progresses along a linked list, in the forward direction. We could take advantage of this leg if we were, for example, counting the distance of a node from the head of the linked list. The *ascending leg* progresses back left, in the reverse direction. We could take advantage of this leg if we were, for example, keeping track of the largest value *after* a node.

- **Tree** The *descending leg* progresses downward, deeper into the tree with each recursive call. We could use this leg if, for example, we were computing each node's depth. The *ascending leg* progresses upwards, getting shallower with each return. We could use this leg if, for example, we were calculating the closest, descendant leaf to a given parent node.

To take advantage of the descending leg, add parameters to your helper function to pass data forwards, or modify your data structure *before* the recursive call. To take advantage of the ascending leg, add return values to your helper function to pass data backwards, or modify your data structure *after* the recursive call.

3.2.2 Patterns for Recursion

Linked list and tree traversals actually reduce the problem of recursion by answering one of three questions for us: How does the recursive call break down the problem into simpler parts? With only a few exceptions, we can generally assume the following is true about each data structure.

1. For **linked lists**, we break down a problem into the current link and the rest of the linked list. Each recursive call processes only the current link (`first`) and *reduces the linked list by a link at a time*. The link's `rest` is left to the recursive call.

2. For **trees**, we break down a problem into the current node and its subtrees. Each recursive call processes only the current link (`entry`) and *reduces the tree by a node at a time*. The tree's branches are left to the recursive call.

Sometimes, our base case may also be straightforward. Given that we're traversing the data structure one node at a time and traversing only one level deeper each time, the base case is simply if we've reached the end the data structure.

1. For a **linked list**, our base case is `if link is Link.empty`.

2. For a **tree**, our base case is `if not tree.branches`.

There are other base cases we may need to consider, if we aren't progressing so simply through the data structure. For example, if we're traversing multiple nodes at once, we may need to add checks for `None`. For example, if we're jumping three nodes at a time, we will need to check that `node.next` and `node.next.next` are not `None`. If we do not add this check, attempting to access `node.next.next.next` may error.

3.3 Problems

3.3.1 Linked Lists

1. **Warm up** Compute the dot product of two vectors `u` and `v`. If the two linked lists, have different length, raise an error.

2. Write a function `sum_reverse(lst)`, which sets each node's value to be the sum of its value and that of all the nodes after it. For example, [1, 2, 3, 4, 5, 6] becomes [21, 20, 18, 15, 11, 6].

3. Rotate a linked list to the left `k` times. For example, `rotate_left([1, 2, 3, 4, 5], 2)` would give `[3, 4, 5, 1, 2]`. Assume that `k` is less than the length of the linked list.

4. Write `average(lst, k)`. To implement this: conceptually divide the linked list into groups of `k` until there are no more or not enough nodes. Update each node to become the average of the group it's in. For k = 3, for example, `[1, 2, 3, 4, 5]` becomes `[2, 2, 2, 4.5, 4.5]`.

5. Check if a linked list represents a palindrome. Each link in the linked list contains one character in the candidate palindrome. Recall that a palindrome is any word that is read the same forwards as it is backwards.

3.3.2 Trees

1. **Warm up** Write `map_tree(f, t)`, which applies a function f to every other level of a tree `t`. Apply `f` to the root, and do not modify the original.

2. **W**rite `filter_tree`, which returns a new tree containing only nodes that pass the filter. To simplify the problem, assume that `f` and `t` are constructed such that the root of the provided `t` always passes the filter `f`. The branches of a deleted node will be pushed up a level.

3. Write `min_leaf`, which creates a new tree, where each node's value is its distance to the closest child leaf.

4. Write a function `sum_reverse_tree(t)`, which sets each node's value to be the sum of its value and that of all its branches. Modify the original tree, in place.

5. **Hard** Solve a maze, given a stringified maze, an optional start(default `(1, 1)`), an optional end (default to the bottom right), and an optional function moves that returns all possible moves given an x, y (default up, down, right, left). Your solution does not need to be optimized, and return the original maze if no path exists.

Chapter 4

Object-Oriented Programming

4.1 Guide

Object-oriented programming is a paradigm for computer programming that allows us to abstract data from its use. More specifically, it allows us to model real-world objects.

For this guide's original, full text, see Composing Programs 2.5-2.7.

4.1.1 Classes

Classes are reusable and extensible templates. We use the following syntax to define a class:

```
class <name>:
  <body>
```

With this syntax in mind, consider the following example of a class, `No2Pencil`. By convention, our class name is camel cased, meaning every word is capitalized.

```
class No2Pencil:
  """a brand new pencil"""
  led_weight = 2 # class attribute
  def __init__(self, length):
    self.length = length # instance attribute
```

We can now create `instances` of this `No2Pencil` class. Note the distinction between a *class* (a template) and an *instance* (an object made from that template). Whereas there is only one class for all `No2Pencil`s, there are many `No2Pencil` instances, namely `pencil1` and `pencil2` in the code below.

```
pencil1 = No2Pencil(5)
pencil2 = No2Pencil(5)
```

4.1.2 Attributes

Consider the definition of `No2Pencil` on the last page. Each of these instances has *instance attributes*, or properties that are specific to that instance. An example would include `length`. We know that each pencil should have its own length, as using one pencil should not affect the length of another pencil. Just as instances have instance attributes, classes may have *class attributes*. These are properties that all instances of a class would most likely share, such as `led_weight`. Since the `No2Pencil` class is for all No. 2 pencils, it would make sense that `No2Pencil` has a class attribute `led_weight = 2`.
To set or get attributes, we use dot notation, for both classes and instances. To get an attribute, use `<expression> . <name>`. To assign an attribute, use `<expression> . <name> = <value>`.

```
# set attribute
No2Pencil.led_weight = 3
pencil1.weight -= 1

# get attribute
print(No2Pencil.led_weight) # 3
print(pencil1.length) # 4
```

Remember that all dot expressions are evaluated in the following three steps.

1. Evaluate the `expression`, left of the dot.
2. Look for an instance attribute `name`. If found, return the associated value.
3. Else, look for a class attribute `name`. If found, return the associated value.

4.1.3 Inheritance

We can define a *subclass*, by using the following syntax.

```
class <Name>(<Parent>):
  <body>
```

A subclass will inherit all class attributes and methods from the parent class, unless those attributes or methods are overridden in the subclass.

4.1.4 Nonlocal

An alternative to OOP to maintain state, `nonlocal` statements allow nested functions to re-assign variables in the parent function. Note that `nonlocal` is only needed for assignment. For impure functions that perform in-place modifications, such as `append` or `remove`, `nonlocal` is *not* needed. Remember that `nonlocal` and OOP are *two different ways to maintain state*.

```
def parent(n):
  def child():
    nonlocal n
    n = 5
```

4.2 Problems

All of the following questions pertain to `nonlocal` *statements. OOP is more so a paradigm to be comfortable with and less a type of programming tool to be tested.*

1. **Warm up** Write `hailstone_chicken(i)`, which returns a function `hailstone(start)` that prints the hailstone sequence for a given `start`. For every `i`th number `n` of the hailstone sequence, print a string with `'chicken'` repeated `n` times *instead* of `n` itself. For a hailstone sequence, if the number is even, divide it by 2. If the number is odd, multiply by 3 and add 1. For `h(8)` where `h=hailstone_chicken(3)`, print `8`, `4`, `chickenchicken`, and `1`.

2. Write `fib_range(x, y)`, which returns `fib`. In turn, `fib` returns the next number in a series of fibonacci numbers, starting from x and ending with y, excluding y. Raise a `StopIteration` exception if there are no more numbers in the sequence. For example, if `fib = fib_range(2, 21)`, calling `fib()` 5 times would return 2, 3, 5, 8, and 13. A 5th function call would raise a `StopIteration` exception. Note that 21 is not included.

3. Without using nonlocal, write `create_complex`, which returns a function `complex(a=None, b=None)`, and initializes the complex number to 0. `complex(a, b)` adds a complex number `a + bi` to the complex number, and `complex()` returns a stringified form of the complex.

4. Write `call_depth()`, which returns two function `f` and `g`. `g` returns the number of times (`n`), that `f` was called, multiplied by the maximum "depth" `d` of a call expression. For `g()`, d=1 and `f` wasn't called so n=0. That makes d*n=0. For `g(f(f()))`, d=3 and n=2, so d*n=6. For `g(f(f(), f(f())))`, d=4, n=4, so d*n=16. (Notice that `f()` has d=1, so `f(f(), f(f()))` is `f(1, 2)` and thus, d=max(1, 2) + 1= 3.)

5. Given a dictionary `rules` mapping integers `i` to strings `s`, `fizzbuzzes(rules)` returns a function `fizzbuzz(sequence)` that prints the `sequence` with a few modifications: apply all rules from the provided `rules`, where each `i`th term is replaced with the corresponding `s`. Higher values of `i` take precedence over lower values of `i`.

Chapter 5

Implicit Sequences

5.1 Guide

Implicit sequences allow us to take advantage of lazy computation and model infinite sequences. In this guide, we will discuss the three implicit sequences covered in Composing Programs: iterators, generators, and streams. The first two are written in Python, and the last is writte in Scheme.

For this guide's original, full text, see Composing Programs 4.2.

5.1.1 Iterator

An *iterator* allows us to sequentially access a series. In Python, iterators must implement a `__next__` method, which returns the next element of a sequence when invoked. Below is a sample iterator that returns the hailstone sequence, given a start value `n`.

```
def HailstoneIter:
  """Generates the hailstone sequence, starting from start."""
  def __iter__(self, n):
    self.n = n

  def __next__(self):
    n = self.n
    if n == 1:
      raise StopIteration # terminate iterator if reached end
    self.n = 3*n+1 if n%2 else n//2
    return n
```

To use an iterator, instantiate an iterator and use Python's built-in `next(<iterator>)`.

```
hailstone = HailstoneIter(4)
next(hailstone) # 4
next(hailstone) # 2
next(hailstone) # 1
next(hailstone) # StopIteration error
```

An *iterable* is any object that can be iterated over. This means that a single iterable can have many associated *iterators*, with each iterator representing an iteration through the iterable. Iterables must implement the `__iter__(<iterator>)` method, which returns an iterable. A hailstone iterable would resemble the following.

```
def HailstoneIter:
  ...
  def __iter__(self):
    return self
```

To use an iterable, instantiate the iterable and use Python's built-in `iter(<iterable>)`.

```
hailstone = iter(Hailstone(4)) # hailstone becomes iterator
next(hailstone) # 4, just like iterator
...
```

In the above code, `__iter__` just returns `self`, since `HailstoneIter` is itself an iterator. Since `for` loops act on `iterables`, we can now use `for` loops with custom classes.

```
for n in Hailstone(4):
  print(n)
```

5.1.2 Generator

A generator is a special type of iterator, returned by a generator function. Using the `yield` statement anywhere inside of a function will automatically make it a generator when called. `yield` statements will put a function on pause, and return that value.

```
def hailstone_gen(n):
  while n != 1:
    yield n
    n = 3*n+1 if n%2 else n//2
```

5.1.3 Stream

Scheme does not support iterators or generators. Instead, scheme uses streams, which can "only" represent infinite sequences. Write the stream definition as though you were constructing a normal list, then substitute `cons` for `cons-stream`. When accessing the stream, use `cdr-stream` instead of `cdr`. All other properties of the scheme list remain the same. The following is a stream of the positive, even numbers.

```
(define (stream n)
  (cons-stream n (+ n 2))
(define pos_evens (stream 2))
```

5.2 Problems

5.2.1 Iterators and Generators

1. **Warm up** Write an iterable that iterates over every other of another sequence (iterator or generator).

2. Write `alt_sequences(seq1, seq2, seq3, order)`, which takes three sequences (generators or iterators) and a list indicating the order with which the sequences are interpolated. If `order` is `[1, 3, 2]`, `seq1` is the naturals, `seq2` is the naturals starting from 2, and `seq3` is the naturals starting from 3, the first three elements of the `alt_sequences` generator would be 1, 3, 2.

3. Write `combine`, which takes two sequences (generators or iterators) and a combiner function `f`. The ith term of the `combine` generator is the ith terms of both seq0 and seq1 combined by function f. The length of `combiner` should be the *longer* of the two sequences. The shorter sequence is padded with 0s.

4. Write a function `kdeepgen(k, lst)` which takes an integer `k` and a list of values to yield `lst`. `kdeepgen` returns a generator that has `k` 'depth' and `yield`s all elements of the provided `lst`. We define the standard generator function, which requires only one `for` loop to iterate over, to have depth 1. A generator with depth 2 would require two `for` loops, with one nested inside the other. A generator with depth k would require k `for` loops.

5. Write a function `flattens(gen)` that flattens a generator with 'depth', returning a generator with depth 1. See the previous question for an explanation of depth. You may use `isinstance(types.GeneratorType)` to test if a variable is a generator. (`import types` first)

5.2.2 Streams

1. **Warm up** Write a stream that contains over every other term of another stream.

2. For two streams `stream1` and `stream2`, combine the first term of `stream1` with the second term of `stream2` using a combiner function `f`. Calling `(combiner naturals naturals (lambda (x) (+ x x)))` should then give `3` (1+2), `6` (2+4), `9` (3+6) etc.

3. Write `alt_sequences(stream1, stream2, stream3, order)`, which takes three streams and a list indicating the order with which the streams are interpolated. If `order` is `(1 3 2)`, `stream1` is the naturals, `stream2` is the naturals starting from 2, and `stream3` is the naturals starting from 3, the first three elements of the `alt-sequences` stream would be 1, 3, 2.