2.3 Sequences

A sequence is an ordered collection of values. The sequence is a powerful, fundamental abstraction in computer science. Sequences are not instances of a particular built-in type or abstract data representation, but instead a collection of behaviors that are shared among several different types of data. That is, there are many kinds of sequences, but they all share common behavior. In particular,

Length. A sequence has a finite length. An empty sequence has length 0.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Python includes several native data types that are sequences, the most important of which is the `list`.

2.3.1 Lists

A `list` value is a sequence that can have arbitrary length. Lists have a large set of built-in behaviors, along with specific syntax to express those behaviors. We have already seen the list literal, which evaluates to a `list` instance, as well as an element selection expression that evaluates to a value in the list. The built-in `len` function returns the length of a sequence. Below, `digits` is a list with four elements. The element at index 3 is 8.

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
```

Additionally, lists can be added together and multiplied by integers. For sequences, addition and multiplication do not add or multiply elements, but instead combine and replicate the sequences themselves. That is, the `add` function in the `operator` module (and the `+` operator) yields a list that is the concatenation of the added arguments. The `mul` function in `operator` (and the `*` operator) can take a list and an integer `k` to return the list that consists of `k` repetitions of the original list.

```
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

Any values can be included in a list, including another list. Element selection can be applied multiple times in order to select a deeply nested element in a list containing lists.

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

2.3.2 Sequence Iteration

In many cases, we would like to iterate over the elements of a sequence and perform some computation for each element in turn. This pattern is so common that Python has an additional control statement to process sequential data: the `for` statement.

Consider the problem of counting how many times a value appears in a sequence. We can implement a function to compute this count using a `while` loop.

```
>>> def count(s, value):
        """Count the number of occurrences of value in sequence s."""
        total, index = 0, 0
        while index < len(s):
            if s[index] == value:
                total = total + 1
            index = index + 1
        return total

>>> count(digits, 8)
2
```

The Python `for` statement can simplify this function body by iterating over the element values directly without introducing the name `index` at all.

```
>>> def count(s, value):
        """Count the number of occurrences of value in sequence s."""
        total = 0
        for elem in s:
            if elem == value:
                total = total + 1
        return total

>>> count(digits, 8)
2
```

A `for` statement consists of a single clause with the form:

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

A `for` statement is executed by the following procedure:

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

This execution procedure refers to *iterable values*. Lists are a type of sequence, and sequences are iterable values. Their elements are considered in their sequential order. Python includes other iterable types, but we will focus on sequences for now; the general definition of the term "iterable" appears in the section on iterators in Chapter 4.

An important consequence of this evaluation procedure is that `<name>` will be bound to the last element of the sequence after the `for` statement is executed. The `for` loop introduces yet another way in which the environment can be updated by a statement.

Sequence unpacking. A common pattern in programs is to have a sequence of elements that are themselves sequences, but all of a fixed length. A `for` statement may include multiple names in its header to "unpack" each element sequence into its respective elements. For example, we may have a list of two-element lists.

```
>>> pairs = [[1, 2], [2, 2], [2, 3], [4, 4]]
```

and wish to find the number of these pairs that have the same first and second element.

```
>>> same_count = 0
```

The following `for` statement with two names in its header will bind each name `x` and `y` to the first and second elements in each pair, respectively.

```
>>> for x, y in pairs:
        if x == y:
            same_count = same_count + 1

>>> same_count
2
```

This pattern of binding multiple names to multiple values in a fixed-length sequence is called *sequence unpacking*; it is the same pattern that we see in assignment statements that bind multiple names to multiple values.

Ranges. A `range` is another built-in type of sequence in Python, which represents a range of integers. Ranges are created with `range`, which takes two integer arguments: the first number and one beyond the last number in the desired range.

```
>>> range(1, 10)  # Includes 1, but not 10
range(1, 10)
```

Calling the `list` constructor on a range evaluates to a list with the same elements as the range, so that the elements can be easily inspected.

```
>>> list(range(5, 8))
[5, 6, 7]
```

If only one argument is given, it is interpreted as one beyond the last value for a range that starts at 0.

```
>>> list(range(4))
[0, 1, 2, 3]
```

Ranges commonly appear as the expression in a `for` header to specify the number of times that the suite should be executed: A common convention is to use a single underscore character for the name in the `for` header if the name is unused in the suite:

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

Go Bears!
Go Bears!
Go Bears!
```

This underscore is just another name in the environment as far as the interpreter is concerned, but has a conventional meaning among programmers that indicates the name will not appear in any future expressions.

2.3.3 Sequence Processing

Sequences are such a common form of compound data that whole programs are often organized around this single abstraction. Modular components that have sequences as both inputs and outputs can be mixed and matched to perform data processing. Complex components can be defined by chaining together a pipeline of sequence processing operations, each of which is simple and focused.

List Comprehensions. Many sequence processing operations can be expressed by evaluating a fixed expression for each element in a sequence and collecting the resulting values in a result sequence. In Python, a list comprehension is an expression that performs such a computation.

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

The `for` keyword above is not part of a `for` statement, but instead part of a list comprehension because it is contained within square brackets. The sub-expression `x+1` is evaluated with `x` bound to each element of `odds` in turn, and the resulting values are collected into a list.

Another common sequence processing operation is to select a subset of values that satisfy some condition. List comprehensions can also express this pattern, for instance selecting all elements of `odds` that evenly divide `25`.

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

The general form of a list comprehension is:

```
[<map expression> for <name> in <sequence expression> if <filter expression>]
```

To evaluate a list comprehension, Python evaluates the `<sequence expression>`, which must return an iterable value. Then, for each element in order, the element value is bound to `<name>`, the filter expression is evaluated, and if it yields a true value, the map expression is evaluated. The values of the map expression are collected into a list.

Aggregation. A third common pattern in sequence processing is to aggregate all values in a sequence into a single value. The built-in functions `sum`, `min`, and `max` are all examples of aggregation functions.

By combining the patterns of evaluating an expression for each element, selecting a subset of elements, and aggregating elements, we can solve problems using a sequence processing approach.

A perfect number is a positive integer that is equal to the sum of its divisors. The divisors of `n` are positive integers less than `n` that divide evenly into `n`. Listing the divisors of `n` can be expressed with a list comprehension.

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

>>> divisors(4)
[1, 2]
>>> divisors(12)
[1, 2, 3, 4, 6]
```

Using `divisors`, we can compute all perfect numbers from 1 to 1000 with another list comprehension. (1 is typically considered to be a perfect number as well, but it does not qualify under our definition of `divisors`.)

```
>>> [n for n in range(1, 1000) if sum(divisors(n)) == n]
[6, 28, 496]
```

We can reuse our definition of `divisors` to solve another problem, finding the minimum perimeter of a rectangle with integer side lengths, given its area. The area of a rectangle is its height times its width. Therefore, given the area and height, we can compute the width. We can assert that both the width and height evenly divide the area to ensure that the side lengths are integers.

```
>>> def width(area, height):
        assert area % height == 0
        return area // height
```

The perimeter of a rectangle is the sum of its side lengths.

```
>>> def perimeter(width, height):
        return 2 * width + 2 * height
```

The height of a rectangle with integer side lengths must be a divisor of its area. We can compute the minimum perimeter by considering all heights.

```
>>> def minimum_perimeter(area):
        heights = divisors(area)
        perimeters = [perimeter(width(area, h), h) for h in heights]
        return min(perimeters)

>>> area = 80
>>> width(area, 5)
16
>>> perimeter(16, 5)
42
>>> perimeter(10, 8)
36
>>> minimum_perimeter(area)
36
>>> [minimum_perimeter(n) for n in range(1, 10)]
[4, 6, 8, 8, 12, 10, 16, 12, 12]
```

Higher-Order Functions. The common patterns we have observed in sequence processing can be

Higher-Order Functions. The common patterns we have observed in sequence processing can be expressed using higher-order functions. First, evaluating an expression for each element in a sequence can be expressed by applying a function to each element.

```
>>> def apply_to_all(map_fn, s):
        return [map_fn(x) for x in s]
```

Selecting only elements for which some expression is true can be expressed by applying a function to each element.

```
>>> def keep_if(filter_fn, s):
        return [x for x in s if filter_fn(x)]
```

Finally, many forms of aggregation can be expressed as repeatedly applying a two-argument function to the `reduced` value so far and each element in turn.

```
>>> def reduce(reduce_fn, s, initial):
        reduced = initial
        for x in s:
            reduced = reduce_fn(reduced, x)
        return reduced
```

For example, `reduce` can be used to multiply together all elements of a sequence. Using `mul` as the `reduce_fn` and 1 as the `initial` value, `reduce` can be used to multiply together a sequence of numbers.

```
>>> reduce(mul, [2, 4, 8], 1)
64
```

We can find perfect numbers using these higher-order functions as well.

```
>>> def divisors_of(n):
        divides_n = lambda x: n % x == 0
        return [1] + keep_if(divides_n, range(2, n))

>>> divisors_of(12)
[1, 2, 3, 4, 6]
>>> from operator import add
>>> def sum_of_divisors(n):
        return reduce(add, divisors_of(n), 0)

>>> def perfect(n):
        return sum_of_divisors(n) == n

>>> keep_if(perfect, range(1, 1000))
[1, 6, 28, 496]
```

Conventional Names. In the computer science community, the more common name for `apply_to_all` is `map` and the more common name for `keep_if` is `filter`. In Python, the built-in `map` and `filter` are generalizations of these functions that do not return lists. These functions are discussed in Chapter 4. The definitions above are equivalent to applying the `list` constructor to the result of built-in `map` and `filter` calls.

```
>>> apply_to_all = lambda map_fn, s: list(map(map_fn, s))
>>> keep_if = lambda filter_fn, s: list(filter(filter_fn, s))
```

The `reduce` function is built into the `functools` module of the Python standard library. In this version, the `initial` argument is optional.

```
>>> from functools import reduce
>>> from operator import mul
>>> def product(s):
        return reduce(mul, s)

>>> product([1, 2, 3, 4, 5])
120
```

In Python programs, it is more common to use list comprehensions directly rather than higher-order functions, but both approaches to sequence processing are widely used.

2.3.4 Sequence Abstraction

We have introduced two native data types that satisfy the sequence abstraction: lists and ranges. Both satisfy the conditions with which we began this section: length and element selection. Python includes two more behaviors of sequence types that extend the sequence abstraction.

Membership. A value can be tested for membership in a sequence. Python has two operators `in` and `not in` that evaluate to `True` or `False` depending on whether an element appears in a sequence.

```
>>> digits
[1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing. Sequences contain smaller sequences within them. A *slice* of a sequence is any contiguous span of the original sequence, designated by a pair of integers. As with the `range` constructor, the first integer

indicates the starting index of the slice and the second indicates one beyond the ending index.

In Python, sequence slicing is expressed similarly to element selection, using square brackets. A colon separates the starting and ending indices. Any bound that is omitted is assumed to be an extreme value: 0 for the starting index, and the length of the sequence for the ending index.

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Enumerating these additional behaviors of the Python sequence abstraction gives us an opportunity to reflect upon what constitutes a useful data abstraction in general. The richness of an abstraction (that is, how many behaviors it includes) has consequences. For users of an abstraction, additional behaviors can be helpful. On the other hand, satisfying the requirements of a rich abstraction with a new data type can be challenging. Another negative consequence of rich abstractions is that they take longer for users to learn.

Sequences have a rich abstraction because they are so ubiquitous in computing that learning a few complex behaviors is justified. In general, most user-defined abstractions should be kept as simple as possible.

Further reading. Slice notation admits a variety of special cases, such as negative starting values, ending values, and step sizes. A complete description appears in the subsection called slicing a list in Dive Into Python 3. In this chapter, we will only use the basic features described above.

2.3.5 Strings

Text values are perhaps more fundamental to computer science than even numbers. As a case in point, Python programs are written and stored as text. The native data type for text in Python is called a string, and corresponds to the constructor `str`.

There are many details of how strings are represented, expressed, and manipulated in Python. Strings are another example of a rich abstraction, one that requires a substantial commitment on the part of the programmer to master. This section serves as a condensed introduction to essential string behaviors.

String literals can express arbitrary text, surrounded by either single or double quotation marks.

```
>>> 'I am string!'
'I am string!'
>>> "I've got an apostrophe"
"I've got an apostrophe"
>>> '您好'
'您好'
```

We have seen strings already in our code, as docstrings, in calls to `print`, and as error messages in `assert` statements.

Strings satisfy the two basic conditions of a sequence that we introduced at the beginning of this section: they have a length and they support element selection.

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

The elements of a string are themselves strings that have only a single character. A character is any single letter of the alphabet, punctuation mark, or other symbol. Unlike many other programming languages, Python does not have a separate character type; any text is a string, and strings that represent single characters have a length of 1.

Like lists, strings can also be combined via addition and multiplication.

```
>>> 'Berkeley' + ', CA'
'Berkeley, CA'
>>> 'Shabu ' * 2
'Shabu Shabu '
```

Membership. The behavior of strings diverges from other sequence types in Python. The string abstraction does not conform to the full sequence abstraction that we described for lists and ranges. In particular, the membership operator `in` applies to strings, but has an entirely different behavior than when it is applied to sequences. It matches substrings rather than elements.

```
>>> 'here' in "Where's Waldo?"
True
```

Multiline Literals. Strings aren't limited to a single line. Triple quotes delimit string literals that span multiple lines. We have used this triple quoting extensively already for docstrings.

```
>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, "Readability counts."\nRead more: import this.'
```

In the printed result above, the `\n` (pronounced "*backslash en*") is a single element that represents a new line. Although it appears as two characters (backslash and "n"), it is considered a single character for the

purposes of length and element selection.

String Coercion. A string can be created from any object in Python by calling the `str` constructor function with an object value as its argument. This feature of strings is useful for constructing descriptive strings from objects of various types.

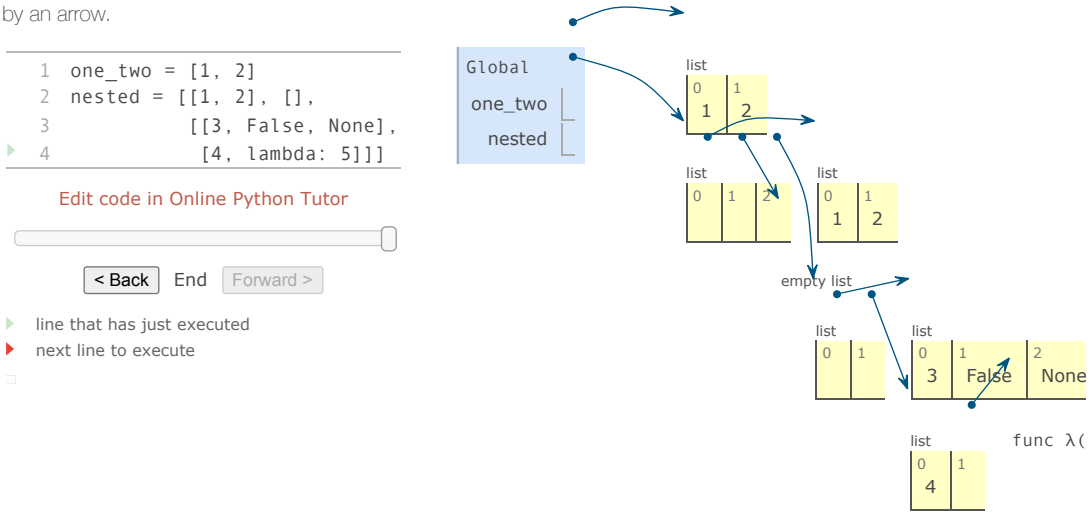
```
>>> str(2) + ' is an element of ' + str(digits)
'2 is an element of [1, 8, 2, 8]'
```

Further reading. Encoding text in computers is a complex topic. In this chapter, we will abstract away the details of how strings are represented. However, for many applications, the particular details of how strings are encoded by computers is essential knowledge. The strings chapter of Dive Into Python 3 provides a description of character encodings and Unicode.

2.3.6 Trees

Our ability to use lists as the elements of other lists provides a new means of combination in our programming language. This ability is called a *closure property* of a data type. In general, a method for combining data values has a closure property if the result of combination can itself be combined using the same method. Closure is the key to power in any means of combination because it permits us to create hierarchical structures — structures made up of parts, which themselves are made up of parts, and so on.

We can visualize lists in environment diagrams using *box-and-pointer* notation. A list is depicted as adjacent boxes that contain the elements of the list. Primitive values such as numbers, strings, boolean values, and `None` appear within an element box. Composite values, such as function values and other lists, are indicated by an arrow.



Nesting lists within lists can introduce complexity. The *tree* is a fundamental data abstraction that imposes regularity on how hierarchical values are structured and manipulated.

A tree has a root label and a sequence of branches. Each branch of a tree is a tree. A tree with no branches is called a leaf. Any tree contained within a tree is called a sub-tree of that tree (such as a branch of a branch). The root of each sub-tree of a tree is called a node in that tree.

The data abstraction for a tree consists of the constructor `tree` and the selectors `label` and `branches`. We begin with a simplified version.

```
>>> def tree(root_label, branches=[]):
        for branch in branches:
            assert is_tree(branch), 'branches must be trees'
        return [root_label] + list(branches)

>>> def label(tree):
        return tree[0]

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

A tree is well-formed only if it has a root label and all branches are also trees. The `is_tree` function is applied in the `tree` constructor to verify that all branches are well-formed.

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

The `is_leaf` function checks whether or not a tree has branches.

```
>>> def is_leaf(tree):
        return not branches(tree)
```

Trees can be constructed by nested expressions. The following tree `t` has root label 3 and two branches.

```
>>> t = tree(3, [tree(1), tree(2, [tree(1), tree(1)])])
>>> t
[3, [1], [2, [1], [1]]]
>>> label(t)
3
>>> branches(t)
[[1], [2, [1], [1]]]
>>> label(branches(t)[1])
2
>>> is_leaf(t)
False
>>> is_leaf(branches(t)[0])
True
```

Tree-recursive functions can be used to construct trees. For example, the nth Fibonacci tree has a root label of the nth Fibonacci number and, for `n > 1`, two branches that are also Fibonacci trees. A Fibonacci tree illustrates the tree-recursive computation of a Fibonacci number.

```
>>> def fib_tree(n):
        if n == 0 or n == 1:
            return tree(n)
        else:
            left, right = fib_tree(n-2), fib_tree(n-1)
            fib_n = label(left) + label(right)
            return tree(fib_n, [left, right])
>>> fib_tree(5)
[5, [2, [1], [1, [0], [1]]], [3, [1, [0], [1]], [2, [1], [1, [0], [1]]]]]
```

Tree-recursive functions are also used to process trees. For example, the `count_leaves` function counts the leaves of a tree.

```
>>> def count_leaves(tree):
      if is_leaf(tree):
          return 1
      else:
          branch_counts = [count_leaves(b) for b in branches(tree)]
          return sum(branch_counts)
>>> count_leaves(fib_tree(5))
8
```

Partition trees. Trees can also be used to represent the partitions of an integer. A partition tree for `n` using parts up to size `m` is a binary (two branch) tree that represents the choices taken during computation. In a non-leaf partition tree:

- the left (index 0) branch contains all ways of partitioning `n` using at least one `m`,
- the right (index 1) branch contains partitions using parts up to `m-1`, and
- the root label is `m`.

The labels at the leaves of a partition tree express whether the path from the root of the tree to the leaf represents a successful partition of `n`.

```
>>> def partition_tree(n, m):
        """Return a partition tree of n using parts of up to m."""
        if n == 0:
            return tree(True)
        elif n < 0 or m == 0:
            return tree(False)
        else:
            left = partition_tree(n-m, m)
            right = partition_tree(n, m-1)
            return tree(m, [left, right])

>>> partition_tree(2, 2)
[2, [True], [1, [1, [True], [False]], [False]]]
```

Printing the partitions from a partition tree is another tree-recursive process that traverses the tree, constructing each partition as a list. Whenever a `True` leaf is reached, the partition is printed.

```
>>> def print_parts(tree, partition=[]):
        if is_leaf(tree):
            if label(tree):
                print(' + '.join(partition))
        else:
            left, right = branches(tree)
            m = str(label(tree))
            print_parts(left, partition + [m])
            print_parts(right, partition)

>>> print_parts(partition_tree(6, 4))
4 + 2
4 + 1 + 1
3 + 3
3 + 2 + 1
3 + 1 + 1 + 1
2 + 2 + 2
2 + 2 + 1 + 1
2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1
```

Slicing can be used on the branches of a tree as well. For example, we may want to place a restriction on the number of branches in a tree. A binarized tree has at most two branches. A common tree transformation called *binarization* finds a binarized tree with the same labels as an original tree by grouping together branches.

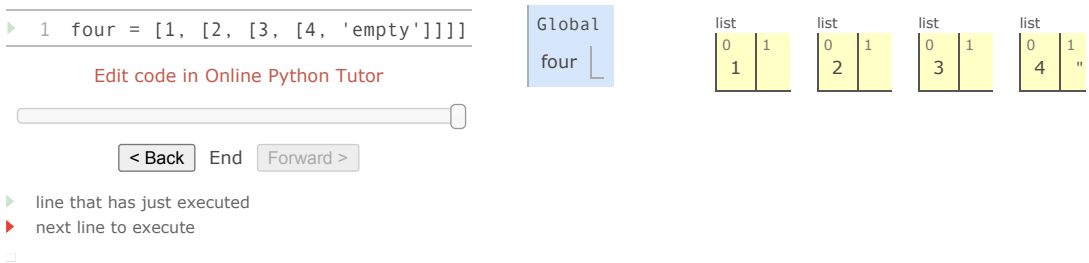
```
>>> def right_binarize(t):
        """Construct a right-branching binary tree."""
    return tree(label(t), binarize_branches(branches(t)))

>>> def binarize_branches(bs):
        """Binarize a list of branches."""
        if len(bs) > 2:
            first, rest = bs[0], bs[1:]
            return [right_binarize(first), binarize_branches(rest)]
        else:
            return [right_binarize(b) for b in bs]

>>> right_binarize(tree(0, [tree(x) for x in [1, 2, 3, 4, 5, 6, 7]]))
[0, [1], [[2], [[3], [[4], [[5], [[6], [7]]]]]]]
```

2.3.7 Linked Lists

So far, we have used only native types to represent sequences. However, we can also develop sequence representations that are not built into Python. A common representation of a sequence constructed from nested pairs is called a *linked list*. The environment diagram below illustrates the linked list representation of a four-element sequence containing 1, 2, 3, and 4.



A linked list is a pair containing the first element of the sequence (in this case 1) and the rest of the sequence (in this case a representation of 2, 3, 4). The second element is also a linked list. The rest of the inner-most linked list containing only 4 is `'empty'`, a value that represents an empty linked list.

Linked lists have recursive structure: the rest of a linked list is a linked list or `'empty'`. We can define an abstract data representation to validate, construct, and select the components of linked lists.

```
>>> empty = 'empty'
>>> def is_link(s):
        """s is a linked list if it is empty or a (first, rest) pair."""
        return s == empty or (len(s) == 2 and is_link(s[1]))

>>> def link(first, rest):
        """Construct a linked list from its first element and the rest."""
        assert is_link(rest), "rest must be a linked list."
        return [first, rest]

>>> def first(s):
        """Return the first element of a linked list s."""
        assert is_link(s), "first only applies to linked lists."
        assert s != empty, "empty linked list has no first element."
        return s[0]

>>> def rest(s):
        """Return the rest of the elements of a linked list s."""
        assert is_link(s), "rest only applies to linked lists."
        assert s != empty, "empty linked list has no rest."
        return s[1]
```

Above, `link` is a constructor and `first` and `rest` are selectors for an abstract data representation of linked lists. The behavior condition for a linked list is that, like a pair, its constructor and selectors are inverse functions.

- If a linked list `s` was constructed from first element `f` and linked list `r`, then `first(s)` returns `f`, and `rest(s)` returns `r`.

We can use the constructor and selectors to manipulate linked lists.

```
>>> four = link(1, link(2, link(3, link(4, empty))))
>>> first(four)
1
>>> rest(four)
[2, [3, [4, 'empty']]]
```

Our implementation of this kind of abstract data uses pairs that are two-element `list` values. It is worth noting that we were also able to implement pairs using functions, and we can implement linked lists using any pairs, therefore we could implement linked lists using functions alone.

The linked list can store a sequence of values in order, but we have not yet shown that it satisfies the sequence abstraction. Using the abstract data representation we have defined, we can implement the two behaviors that characterize a sequence: length and element selection.

```
>>> def len_link(s):
        """Return the length of linked list s."""
        length = 0
        while s != empty:
            s, length = rest(s), length + 1
        return length

>>> def getitem_link(s, i):
        """Return the element at index i of linked list s."""
        while i > 0:
            s, i = rest(s), i - 1
        return first(s)
```

Now, we can manipulate a linked list as a sequence using these functions. (We cannot yet use the built-in `len` function, element selection syntax, or `for` statement, but we will soon.)

```
>>> len_link(four)
4
>>> getitem_link(four, 1)
2
```

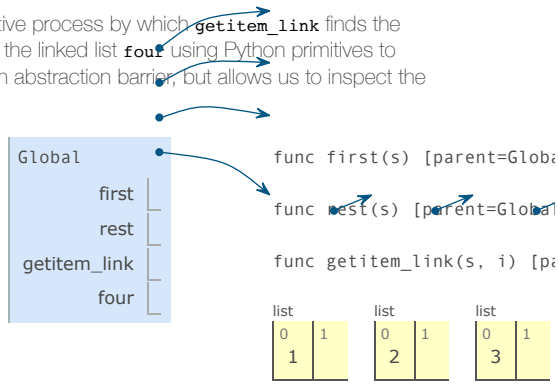
The series of environment diagrams below illustrate the iterative process by which `getitem_link` finds the element 2 at index 1 in a linked list. Below, we have defined the linked list `four` using Python primitives to simplify the diagrams. This implementation choice violates an abstraction barrier, but allows us to inspect the computational process more easily for this example.

```
1  def first(s):
2      return s[0]
3  def rest(s):
4      return s[1]
5
6  def getitem_link(s, i):
7      while i > 0:
8          s, i = rest(s), i - 1
9      return first(s)
10
11 four = [1, [2, [3, [4, 'empty']]]]
12 getitem_link(four, 1)
```

Edit code in Online Python Tutor

< Back Step 5 of 14 Forward >

line that has just executed
next line to execute



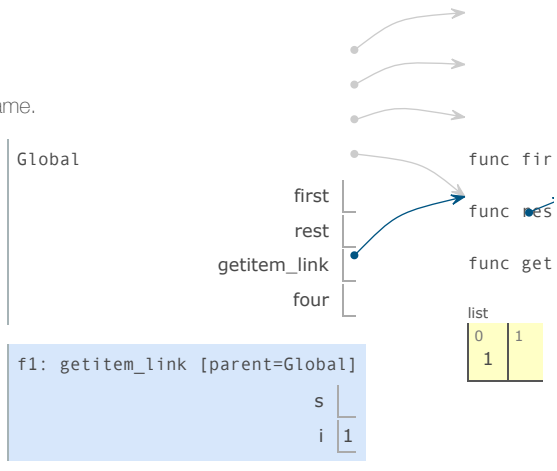
First, the function `getitem_link` is called, creating a local frame.

```
1  def first(s):
2      return s[0]
3  def rest(s):
4      return s[1]
5
6  def getitem_link(s, i):
7      while i > 0:
8          s, i = rest(s), i - 1
9      return first(s)
10
11 four = [1, [2, [3, [4, 'empty']]]]
12 getitem_link(four, 1)
```

Edit code in Online Python Tutor

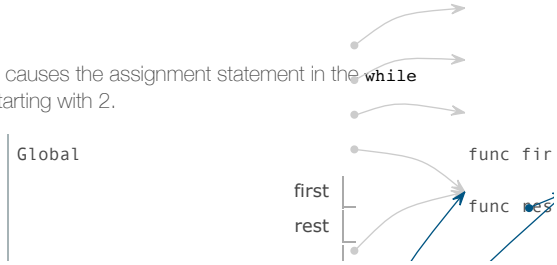
< Back Step 6 of 14 Forward >

line that has just executed
next line to execute



The expression in the `while` header evaluates to true, which causes the assignment statement in the `while` suite to be executed. The function `rest` returns the sublist starting with 2.

```
1  def first(s):
2      return s[0]
3  def rest(s):
4      return s[1]
```

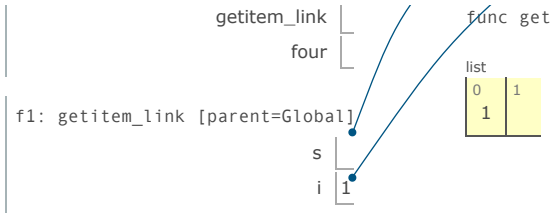


```
5
6  def getitem_link(s, i):
7      while i > 0:
8          s, i = rest(s), i - 1
9      return first(s)
10
11 four = [1, [2, [3, [4, 'empty']]]]
12
12 getitem_link(four, 1)
```

Edit code in Online Python Tutor

< Back Step 9 of 14 Forward >

line that has just executed
next line to execute



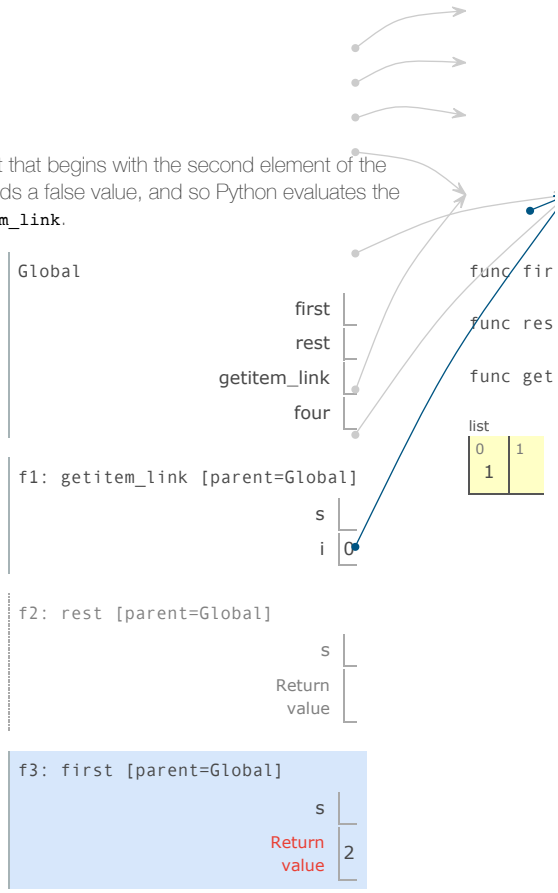
Next, the local name `s` will be updated to refer to the sub-list that begins with the second element of the original list. Evaluating the `while` header expression now yields a false value, and so Python evaluates the expression in the return statement on the final line of `getitem_link`.

```
1  def first(s):
2      return s[0]
3  def rest(s):
4      return s[1]
5
6  def getitem_link(s, i):
7      while i > 0:
8          s, i = rest(s), i - 1
9      return first(s)
10
11 four = [1, [2, [3, [4, 'empty']]]]
12 getitem_link(four, 1)
```

Edit code in Online Python Tutor

< Back Step 13 of 14 Forward >

line that has just executed
next line to execute



This final environment diagram shows the local frame for the call to `first`, which contains the name `s` bound to that same sub-list. The `first` function selects the value 2 and returns it, which will also be returned from `getitem_link`.

This example demonstrates a common pattern of computation with linked lists, where each step in an iteration operates on an increasingly shorter suffix of the original list. This incremental processing to find the length and elements of a linked list does take some time to compute. Python's built-in sequence types are implemented in a different way that does not have a large cost for computing the length of a sequence or retrieving its elements. The details of that representation are beyond the scope of this text.

Recursive manipulation. Both `len_link` and `getitem_link` are iterative. They peel away each layer of nested pair until the end of the list (in `len_link`) or the desired element (in `getitem_link`) is reached. We can also implement length and element selection using recursion.

```
>>> def len_link_recursive(s):
        """Return the length of a linked list s."""
        if s == empty:
            return 0
        return 1 + len_link_recursive(rest(s))

>>> def getitem_link_recursive(s, i):
        """Return the element at index i of linked list s."""
        if i == 0:
            return first(s)
        return getitem_link_recursive(rest(s), i - 1)

>>> len_link_recursive(four)
4
>>> getitem_link_recursive(four, 1)
2
```

These recursive implementations follow the chain of pairs until the end of the list (in `len_link_recursive`) or the desired element (in `getitem_link_recursive`) is reached.

Recursion is also useful for transforming and combining linked lists.

```
>>> def extend_link(s, t):
        """Return a list with the elements of s followed by those of t."""
        assert is_link(s) and is_link(t)
        if s == empty:
            return t
        else:
            return link(first(s), extend_link(rest(s), t))

>>> extend_link(four, four)
[1, [2, [3, [4, [1, [2, [3, [4, 'empty']]]]]]]]

>>> def apply_to_all_link(f, s):
        """Apply f to each element of s."""
        assert is_link(s)
        if s == empty:
            return s
        else:
            return link(f(first(s)), apply_to_all_link(f, rest(s)))

>>> apply_to_all_link(lambda x: x*x, four)
[1, [4, [9, [16, 'empty']]]]

>>> def keep_if_link(f, s):
        """Return a list with elements of s for which f(e) is true."""
        assert is_link(s)
        if s == empty:
            return s
        else:
            kept = keep_if_link(f, rest(s))
            if f(first(s)):
                return link(first(s), kept)
            else:
                return kept

>>> keep_if_link(lambda x: x%2 == 0, four)
[2, [4, 'empty']]

>>> def join_link(s, separator):
        """Return a string of all elements in s separated by separator."""
        if s == empty:
            return ""
        elif rest(s) == empty:
            return str(first(s))
        else:
            return str(first(s)) + separator + join_link(rest(s), separator)

>>> join_link(four, ", ")
'1, 2, 3, 4'
```

Recursive Construction. Linked lists are particularly useful when constructing sequences incrementally, a situation that arises often in recursive computations.

The `count_partitions` function from Chapter 1 counted the number of ways to partition an integer `n` using parts up to size `m` via a tree-recursive process. With sequences, we can also enumerate these partitions explicitly using a similar process.

We follow the same recursive analysis of the problem as we did while counting: partitioning `n` using integers up to `m` involves either

1. partitioning `n-m` using integers up to `m`, or
2. partitioning `n` using integers up to `m-1`.

For base cases, we find that 0 has an empty partition, while partitioning a negative integer or using parts smaller than 1 is impossible.

```
>>> def partitions(n, m):
        """Return a linked list of partitions of n using parts of up to m.
        Each partition is represented as a linked list.
        """
        if n == 0:
            return link(empty, empty) # A list containing the empty partition
        elif n < 0 or m == 0:
            return empty
        else:
            using_m = partitions(n-m, m)
            with_m = apply_to_all_link(lambda s: link(m, s), using_m)
            without_m = partitions(n, m-1)
            return extend_link(with_m, without_m)
```

In the recursive case, we construct two sublists of partitions. The first uses `m`, and so we prepend `m` to each element of the result `using_m` to form `with_m`.

The result of `partitions` is highly nested: a linked list of linked lists, and each linked list is represented as nested pairs that are `list` values. Using `join_link` with appropriate separators, we can display the partitions in a human-readable manner.

```
>>> def print_partitions(n, m):
        lists = partitions(n, m)
        strings = apply_to_all_link(lambda s: join_link(s, " + "), lists)
        print(join_link(strings, "\n"))

>>> print_partitions(6, 4)
4 + 2
4 + 1 + 1
3 + 3
3 + 2 + 1
3 + 1 + 1 + 1
2 + 2 + 2
2 + 2 + 1 + 1
2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1
```

Continue: 2.4 Mutable Data