## 1.7 Recursive Functions

**Video:** Show Hide

A function is called *recursive* if the body of the function calls the function itself, either directly or indirectly. That is, the process of executing the body of a recursive function may in turn require applying that function again. Recursive functions do not use any special syntax in Python, but they do require some effort to understand and create.

We'll begin with an example problem: write a function that sums the digits of a natural number. When designing recursive functions, we look for ways in which a problem can be broken down into simpler problems. In this case, the operators `%` and `//` can be used to separate a number into two parts: its last digit and all but its last digit.

```
>>> 18117 % 10
7
>>> 18117 // 10
1811
```

The sum of the digits of 18117 is `1+8+1+1+7 = 18`. Just as we can separate the number, we can separate this sum into the last digit, 7, and the sum of all but the last digit, `1+8+1+1 = 11`. This separation gives us an algorithm: to sum the digits of a number `n`, add its last digit `n % 10` to the sum of the digits of `n // 10`. There's one special case: if a number has only one digit, then the sum of its digits is itself. This algorithm can be implemented as a recursive function.

```
>>> def sum_digits(n):
        """Return the sum of the digits of positive integer n."""
        if n < 10:
            return n
        else:
            all_but_last, last = n // 10, n % 10
            return sum_digits(all_but_last) + last
```

This definition of `sum_digits` is both complete and correct, even though the `sum_digits` function is called within its own body. The problem of summing the digits of a number is broken down into two steps: summing all but the last digit, then adding the last digit. Both of these steps are simpler than the original problem. The function is recursive because the first step is the same kind of problem as the original problem. That is, `sum_digits` is exactly the function we need in order to implement `sum_digits`.

```
>>> sum_digits(9)
9
>>> sum_digits(18117)
18
>>> sum_digits(9437184)
36
>>> sum_digits(11408855402054064613470328848384)
126
```

We can understand precisely how this recursive function applies successfully using our environment model of computation. No new rules are required.

```
1  def sum_digits(n):

2      if n < 10:

3          return n

4      else:

5          all_but_last, last = n // 10, n % 10

6          return sum_digits(all_but_last) + last

7

8  sum_digits(738)
```

**Edit code in Online Python Tutor**

< Back    End    Forward >

| Global | |
| --- | --- |
| sum_digits | |

| f1: sum_digits [parent=Global] | |
| --- | --- |
| n | 738 |
| all_but_last | 73 |
| last | 8 |
| Return value | 18 |

| f2: sum_digits [parent=Global] | |
| --- | --- |
| n | 73 |
| all_but_last | 7 |
| last | 3 |
| Return value | 10 |

| f3: sum_digits [parent=Global] | |
| --- | --- |
| n | 7 |
| Return value | 7 |

line that has just executed

next line to execute

When the `def` statement is executed, the name `sum_digits` is bound to a new function, but the body of that function is not yet executed. Therefore, the circular nature of `sum_digits` is not a problem yet. Then, `sum_digits` is called on 738:

1. A local frame for `sum_digits` with `n` bound to 738 is created, and the body of `sum_digits` is executed in the environment that starts with that frame.
2. Since 738 is not less than 10, the assignment statement on line 4 is executed, splitting 738 into 73 and 8.
3. In the following return statement, `sum_digits` is called on 73, the value of `all_but_last` in the current environment.
3. Another local frame for `sum_digits` is created, this time with `n` bound to 73. The body of `sum_digits` is again executed in the new environment that starts with this frame.
4. Since 73 is also not less than 10, 73 is split into 7 and 3 and `sum_digits` is called on 7, the value of `all_but_last` evaluated in this frame.
5. A third local frame for `sum_digits` is created, with `n` bound to 7.
6. In the environment starting with this frame, it is true that `n < 10`, and therefore 7 is returned.
7. In the second local frame, this return value 7 is summed with 3, the value of `last`, to return 10.
8. In the first local frame, this return value 10 is summed with 8, the value of `last`, to return 18.

This recursive function applies correctly, despite its circular character, because it is applied twice, but with a different argument each time. Moreover, the second application was a simpler instance of the digit summing problem than the first. Generate the environment diagram for the call `sum_digits(18117)` to see that each successive call to `sum_digits` takes a smaller argument than the last, until eventually a single-digit input is reached.

This example also illustrates how functions with simple bodies can evolve complex computational processes by using recursion.

## 1.7.1  The Anatomy of Recursive Functions

**Video:** Show Hide

A common pattern can be found in the body of many recursive functions. The body begins with a *base case*, a conditional statement that defines the behavior of the function for the inputs that are simplest to process. In the case of `sum_digits`, the base case is any single-digit argument, and we simply return that argument. Some recursive functions will have multiple base cases.

The base cases are then followed by one or more *recursive calls*. Recursive calls always have a certain character: they simplify the original problem. Recursive functions express computation by simplifying problems incrementally. For example, summing the digits of 7 is simpler than summing the digits of 73, which in turn is simpler than summing the digits of 738. For each subsequent call, there is less work left to be done.

Recursive functions often solve problems in a different way than the iterative approaches that we have used previously. Consider a function `fact` to compute `n` factorial, where for example `fact(4)` computes $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ 4!=4·3·2·1=24.

A natural implementation using a `while` statement accumulates the total by multiplying together each positive integer up to `n`.

```
>>> def fact_iter(n):
        total, k = 1, 1
        while k <= n:
            total, k = total * k, k + 1
        return total
```

```
>>> fact_iter(4)
24
```

On the other hand, a recursive implementation of factorial can express `fact(n)` in terms of `fact(n-1)`, a simpler problem. The base case of the recursion is the simplest form of the problem: `fact(1)` is 1.

```
1  def fact(n):
2      if n == 1:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(4)
```

Edit code in Online Python Tutor

< Back    End    Forward >

line that has just executed

next line to execute



```
Global
            fact

f1: fact [parent=Global]
            n  4
         Return  24
         value

f2: fact [parent=Global]
            n  3
         Return  6
         value

f3: fact [parent=Global]
            n  2
         Return  2
         value

f4: fact [parent=Global]
            n  1
         Return  1
         value
```

func fact(n) [par(

These two factorial functions differ conceptually. The iterative function constructs the result from the base case of 1 to the final total by successively multiplying in each term. The recursive function, on the other hand, constructs the result directly from the final term, `n`, and the result of the simpler problem, `fact(n-1)`.

As the recursion "unwinds" through successive applications of the *fact* function to simpler and simpler problem instances, the result is eventually built starting from the base case. The recursion ends by passing the argument 1 to `fact`; the result of each call depends on the next until the base case is reached.

The correctness of this recursive function is easy to verify from the standard definition of the mathematical function for factorial:

$$(n-1)! = (n-1) \cdot (n-2) \cdot \cdots \cdot 1$$
$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 1$$
$$n! = n \cdot (n-1)!$$

(n–1)!=(n–1)·(n–2)·····1n!=n·(n–1)·(n–2)·····1n!=n·(n–1)!

While we can unwind the recursion using our model of computation, it is often clearer to think about recursive calls as functional abstractions. That is, we should not care about how `fact(n-1)` is implemented in the body of `fact`; we should simply trust that it computes the factorial of `n-1`. Treating a recursive call as a functional abstraction has been called a *recursive leap of faith*. We define a function in terms of itself, but simply trust that the simpler cases will work correctly when verifying the correctness of the function. In this example, we trust that `fact(n-1)` will correctly compute `(n-1)!`; we must only check that `n!` is computed correctly if this assumption holds. In this way, verifying the correctness of a recursive function is a form of proof by induction.

The functions *fact_iter* and *fact* also differ because the former must introduce two additional names, `total` and `k`, that are not required in the recursive implementation. In general, iterative functions must maintain some local state that changes throughout the course of computation. At any point in the iteration, that state characterizes the result of completed work and the amount of work remaining. For example, when `k` is 3 and total is 2, there are still two terms remaining to be processed, 3 and 4. On the other hand, *fact* is characterized by its single argument `n`. The state of the computation is entirely contained within the structure of the environment, which has return values that take the role of `total`, and binds `n` to different values in different frames rather than explicitly tracking `k`.

Recursive functions leverage the rules of evaluating call expressions to bind names to values, often avoiding the nuisance of correctly assigning local names during iteration. For this reason, recursive functions can be easier to define correctly. However, learning to recognize the computational processes evolved by recursive functions certainly requires practice.

### 1.7.2   Mutual Recursion

**Video:** Show Hide

When a recursive procedure is divided among two functions that call each other, the functions are said to be *mutually recursive*. As an example, consider the following definition of even and odd for non-negative integers:

- a number is even if it is one more than an odd number
- a number is odd if it is one more than an even number
- 0 is even

Using this definition, we can implement mutually recursive functions to determine whether a number is even or odd:

```python
1   def is_even(n):
2       if n == 0:
3           return True
4       else:
5           return is_odd(n-1)
6
7   def is_odd(n):
8       if n == 0:
9           return False
10      else:
11          return is_even(n-1)
12
13  result = is_even(4)
```

Edit code in Online Python Tutor
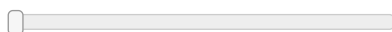
< Back     Step 1 of 18     Forward >

line that has just executed

next line to execute

Mutually recursive functions can be turned into a single recursive function by breaking the abstraction boundary between the two functions. In this example, the body of `is_odd` can be incorporated into that of `is_even`, making sure to replace `n` with `n-1` in the body of `is_odd` to reflect the argument passed into it:

```python
>>> def is_even(n):
        if n == 0:
            return True
```

```
    else:
        if (n-1) == 0:
            return False
        else:
            return is_even((n-1)-1)
```

As such, mutual recursion is no more mysterious or powerful than simple recursion, and it provides a mechanism for maintaining abstraction within a complicated recursive program.

### 1.7.3  Printing in Recursive Functions

**Video:** Show Hide

The computational process evolved by a recursive function can often be visualized using calls to `print`. As an example, we will implement a function `cascade` that prints all prefixes of a number from largest to smallest to largest.

```
>>> def cascade(n):
        """Print a cascade of prefixes of n."""
        if n < 10:
            print(n)
        else:
            print(n)
            cascade(n//10)
            print(n)
```

```
>>> cascade(2013)
2013
201
20
2
20
201
2013
```

In this recursive function, the base case is a single-digit number, which is printed. Otherwise, a recursive call is placed between two calls to `print`.

**Video:** Show Hide

It is not a rigid requirement that base cases be expressed before recursive calls. In fact, this function can be expressed more compactly by observing that `print(n)` is repeated in both clauses of the conditional statement, and therefore can precede it.

```
>>> def cascade(n):
        """Print a cascade of prefixes of n."""
        print(n)
        if n >= 10:
            cascade(n//10)
            print(n)
```

As another example of mutual recursion, consider a two-player game in which there are `n` initial pebbles on a table. The players take turns, removing either one or two pebbles from the table, and the player who removes the final pebble wins. Suppose that Alice and Bob play this game, each using a simple strategy:

- Alice always removes a single pebble
- Bob removes two pebbles if an even number of pebbles is on the table, and one otherwise

Given `n` initial pebbles and Alice starting, who wins the game?

A natural decomposition of this problem is to encapsulate each strategy in its own function. This allows us to modify one strategy without affecting the other, maintaining the abstraction barrier between the two. In order to incorporate the turn-by-turn nature of the game, these two functions call each other at the end of each turn.

```
>>> def play_alice(n):
        if n == 0:
            print("Bob wins!")
        else:
            play_bob(n-1)
```

```
>>> def play_bob(n):
        if n == 0:
            print("Alice wins!")
        elif is_even(n):
            play_alice(n-2)
        else:
            play_alice(n-1)
```

```
>>> play_alice(20)
Bob wins!
```

In `play_bob`, we see that multiple recursive calls may appear in the body of a function. However, in this example, each call to `play_bob` calls `play_alice` at most once. In the next section, we consider what happens when a single function call makes multiple direct recursive calls.

### 1.7.4  Tree Recursion

**Video:** Show Hide

Another common pattern of computation is called tree recursion, in which a function calls itself more than once. As an example, consider computing the sequence of Fibonacci numbers, in which each number is the sum of the preceding two.

```
1  def fib(n):
2      if n == 1:
3          return 0
4      if n == 2:
5          return 1
```

```
6    else:
7        return fib(n-2) + fib(n-1)
8


9  result = fib(6)
```

Edit code in Online Python Tutor

[ < Back ]   Step 1 of 59   [ Forward > ]

line that has just executed

next line to execute

This recursive definition is tremendously appealing relative to our previous attempts: it exactly mirrors the familiar definition of Fibonacci numbers. A function with multiple recursive calls is said to be *tree recursive* because each call branches into multiple smaller calls, each of which branches into yet smaller calls, just as the branches of a tree become smaller but more numerous as they extend from the trunk.

We were already able to define a function to compute Fibonacci numbers without tree recursion. In fact, our previous attempts were more efficient, a topic discussed later in the text. Next, we consider a problem for which the tree recursive solution is substantially simpler than any iterative alternative.

## 1.7.5  Example: Partitions

**Video:** Show Hide

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order. For example, the number of partitions of 6 using parts up to 4 is 9.

1. `6 = 2 + 4`
2. `6 = 1 + 1 + 4`
3. `6 = 3 + 3`
4. `6 = 1 + 2 + 3`
5. `6 = 1 + 1 + 1 + 3`
6. `6 = 2 + 2 + 2`
7. `6 = 1 + 1 + 2 + 2`
8. `6 = 1 + 1 + 1 + 1 + 2`
9. `6 = 1 + 1 + 1 + 1 + 1 + 1`

We will define a function `count_partitions(n, m)` that returns the number of different partitions of $n$ using parts up to $m$. This function has a simple solution as a tree-recursive function, based on the following observation:

The number of ways to partition $n$ using integers up to $m$ equals

1. the number of ways to partition $n-m$ using integers up to $m$, and
2. the number of ways to partition $n$ using integers up to $m-1$.

To see why this is true, observe that all the ways of partitioning $n$ can be divided into two groups: those that include at least one $m$ and those that do not. Moreover, each partition in the first group is a partition of $n-m$, followed by $m$ added at the end. In the example above, the first two partitions contain 4, and the rest do not.

Therefore, we can recursively reduce the problem of partitioning $n$ using integers up to $m$ into two simpler problems: (1) partition a smaller number $n-m$, and (2) partition with smaller components up to $m-1$.

To complete the implementation, we need to specify the following base cases:

1. There is one way to partition 0: include no parts.
2. There are 0 ways to partition a negative $n$.
3. There are 0 ways to partition any $n$ greater than 0 using parts of size 0 or less.

```
>>> def count_partitions(n, m):
        """Count the ways to partition n using parts up to m."""
        if n == 0:
            return 1
        elif n < 0:
            return 0
        elif m == 0:
```

```
            return 0
        else:
            return count_partitions(n-m, m) + count_partitions(n, m-1)
```

```
>>> count_partitions(6, 4)
9
>>> count_partitions(5, 5)
7
>>> count_partitions(10, 10)
42
>>> count_partitions(15, 15)
176
>>> count_partitions(20, 20)
627
```

We can think of a tree-recursive function as exploring different possibilities. In this case, we explore the possibility that we use a part of size `m` and the possibility that we do not. The first and second recursive calls correspond to these possibilities.

Implementing this function without recursion would be substantially more involved. Interested readers are encouraged to try.