Chapter 13

Functions

Functions are useful for breaking up a large program to make it easier to read and maintain. They are also useful if find yourself writing the same code at several different points in your program. You can put that code in a function and call the function whenever you want to execute that code. You can also use functions to create your own utilities, math functions, etc.

13.1 Basics

Functions are defined with the `def` statement. The statement ends with a colon, and the code that is part of the function is indented below the `def` statement. Here we create a simple function that just prints something.

```
def print_hello():
    print('Hello!')

print_hello()
print('1234567')
print_hello()
```

```
Hello!
1234567
Hello!
```

The first two lines define the function. In the last three lines we call the function twice.

One use for functions is if you are are using the same code over and over again in various parts of your program, you can make your program shorter and easier to understand by putting the code in a function. For instance, suppose for some reason you need to print a box of stars like the one below at several points in your program.

```
***************
*             *
*             *
***************
```

Put the code into a function, and then whenever you need a box, just call the function rather than typing several lines of redundant code. Here is the function.

```
def draw_square():
    print('*' * 15)
    print('*', ' '*11, '*')
    print('*', ' '*11, '*')
    print('*' * 15)
```

One benefit of this is that if you decide to change the size of the box, you just have to modify the code in the function, whereas if you had copied and pasted the box-drawing code everywhere you needed it, you would have to change all of them.

13.2 Arguments

We can pass values to functions. Here is an example:

```
def print_hello(n):
    print('Hello ' * n)
    print()

print_hello(3)
print_hello(5)
times = 2
print_hello(times)
```

```
Hello Hello Hello
Hello Hello Hello Hello Hello
Hello Hello
```

When we call the `print_hello` function with the value 3, that value gets stored in the variable `n`. We can then refer to that variable `n` in our function's code.

You can pass more than one value to a function:

```
def multiple_print(string, n)
    print(string * n)
    print()

multiple_print('Hello', 5)
multiple_print('A', 10)
```

```
HelloHelloHelloHelloHello
AAAAAAAAAA
```

13.3 Returning values

We can write functions that perform calculations and return a result.

Example 1 Here is a simple function that converts temperatures from Celsius to Fahrenheit.

```
def convert(t):
    return t*9/5+32

print(convert(20))
```

```
68
```

The `return` statement is used to send the result of a function's calculations back to the caller.

Notice that the function itself does not do any printing. The printing is done outside of the function. That way, we can do math with the result, like below.

```
print(convert(20)+5)
```

If we had just printed the result in the function instead of returning it, the result would have been printed to the screen and forgotten about, and we would never be able to do anything with it.

Example 2 As another example, the Python `math` module contains trig functions, but they only work in radians. Let us write our own sine function that works in degrees.

```
from math import pi, sin

def deg_sin(x):
    return sin(pi*x/180)
```

Example 3 A function can return multiple values as a list.

Say we want to write a function that solves the system of equations $ax + by = e$ and $cx + dy = f$. It turns out that if there is a unique solution, then it is given by $x = (de - bf)/(ad - bc)$ and $y = (af - ce)/(ad - bc)$. We need our function to return both the x and y solutions.

```
def solve(a,b,c,d,e,f):
    x = (d*e-b*f)/(a*d-b*c)
    y = (a*f-c*e)/(a*d-b*c)
    return [x,y]
```

```
xsol, ysol = solve(2,3,4,1,2,5)
print('The solution is x = ', xsol, 'and y = ', ysol)
```

```
The solution is x =  1.3 and y =  -0.2
```

This method uses the shortcut for assigning to lists that was mentioned in Section 10.3.

Example 4 A `return` statement by itself can be used to end a function early.

```
def multiple_print(string, n, bad_words):
    if string in bad_words:
        return
    print(string * n)
    print()
```

The same effect can be achieved with an if/else statement, but in some cases, using `return` can make your code simpler and more readable.

13.4 Default arguments and keyword arguments

You can specify a default value for an argument. This makes it optional, and if the caller decides not to use it, then it takes the default value. Here is an example:

```
def multiple_print(string, n=1)
    print(string * n)
    print()

multiple_print('Hello', 5)
multiple_print('Hello')
```

```
HelloHelloHelloHelloHello
Hello
```

Default arguments need to come at the end of the function definition, after all of the non-default arguments.

Keyword arguments A related concept to default arguments is *keyword arguments*. Say we have the following function definition:

```
def fancy_print(text, color, background, style, justify):
```

Every time you call this function, you have to remember the correct order of the arguments. Fortunately, Python allows you to name the arguments when calling the function, as shown below:

```
fancy_print(text='Hi', color='yellow', background='black',
            style='bold', justify='left')
```

```
fancy_print(text='Hi', style='bold', justify='left',
            background='black', color='yellow')
```

As we can see, the order of the arguments does not matter when you use keyword arguments.

When defining the function, it would be a good idea to give defaults. For instance, most of the time, the caller would want left justification, a white background, etc. Using these values as defaults means the caller does not have to specify every single argument every time they call the function. Here is a example:

```
def fancy_print(text, color='black', background='white',
                style='normal', justify='left'):
    # function code goes here

fancy_print('Hi', style='bold')
fancy_print('Hi', color='yellow', background='black')
fancy_print('Hi')
```

Note We have actually seen default and keyword arguments before—the `sep`, `end` and `file` arguments of the `print` function.

13.5 Local variables

Let's say we have two functions like the ones below that each use a variable `i`:

```
def func1():
    for i in range(10):
        print(i)

def func2():
    i=100
    func1()
    print(i)
```

A problem that could arise here is that when we call `func1`, we might mess up the value of `i` in `func2`. In a large program it would be a nightmare trying to make sure that we don't repeat variable names in different functions, and, fortunately, we don't have to worry about this. When a variable is defined inside a function, it is *local* to that function, which means it essentially does not exist outside that function. This way each function can define its own variables and not have to worry about if those variable names are used in other functions.

Global variables On the other hand, sometimes you actually do want the same variable to be available to multiple functions. Such a variable is called a *global* variable. You have to be careful using global variables, especially in larger programs, but a few global variables used judiciously are fine in smaller programs. Here is a short example:

```
def reset():
    global time_left
    time_left = 0

def print_time():
    print(time_left)

time_left=30
```

In this program we have a variable `time_left` that we would like multiple functions to have access to. If a function wants to change the value of that variable, we need to tell the function that `time_left` is a global variable. We use a **`global`** statement in the function to do this. On the other hand, if we just want to use the value of the global variable, we do not need a **`global`** statement.

Arguments We finish the chapter with a bit of a technical detail. You can skip this section for the time being if you don't want to worry about details right now. Here are two simple functions:

```
def func1(x):
    x = x + 1

def func2(L):
    L = L + [1]

a=3
M=[1,2,3]
func1(a)
func2(M)
```

When we call `func1` with `a` and `func2` with `L`, a question arises: do the functions change the values of `a` and `L`? The answer may surprise you. The value of `a` is unchanged, but the value of `L` is changed. The reason has to do with a difference in the way that Python handles numbers and lists. Lists are said to be *mutable* objects, meaning they can be changed, whereas numbers and strings are *immutable*, meaning they cannot be changed. There is more on this in Section 19.1.

If we want to reverse the behavior of the above example so that `a` is modified and `L` is not, do the following:

```
def func1(x):
    x = x + 1
    return x

def func2(L):
    copy = L[:]
    copy = copy + [1]

a=3
M=[1,2,3]
a=func1(a)  # note change on this line
```

```
func2(M)
```

13.6 Exercises

1. Write a function called `rectangle` that takes two integers `m` and `n` as arguments and prints out an $m \times n$ box consisting of asterisks. Shown below is the output of `rectangle(2,4)`

   ```
   ****
   ****
   ```

2. (a) Write a function called `add_excitement` that takes a list of strings and adds an exclamation point (`!`) to the end of each string in the list. The program should modify the original list and not return anything.

   (b) Write the same function except that it should not modify the original list and should instead return a new list.

3. Write a function called `sum_digits` that is given an integer `num` and returns the sum of the digits of `num`.

4. The *digital root* of a number n is obtained as follows: Add up the digits n to get a new number. Add up the digits of that to get another new number. Keep doing this until you get a number that has only one digit. That number is the digital root.

   For example, if $n = 45893$, we add up the digits to get $4+5+8+9+3 = 29$. We then add up the digits of 29 to get $2+9 = 11$. We then add up the digits of 11 to get $1+1 = 2$. Since 2 has only one digit, 2 is our digital root.

   Write a function that returns the digital root of an integer n. [Note: there is a shortcut, where the digital root is equal to n mod 9, but do not use that here.]

5. Write a function called `first_diff` that is given two strings and returns the first location in which the strings differ. If the strings are identical, it should return -1.

6. Write a function called `binom` that takes two integers n and k and returns the binomial coefficient $\binom{n}{k}$. The definition is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

7. Write a function that takes an integer n and returns a random integer with exactly n digits. For instance, if n is 3, then 125 and 593 would be valid return values, but 093 would not because that is really 93, which is a two-digit number.

8. Write a function called `number_of_factors` that takes an integer and returns how many factors the number has.

9. Write a function called `factors` that takes an integer and returns a list of its factors.

10. Write a function called `closest` that takes a list of numbers `L` and a number `n` and returns the largest element in `L` that is not larger than `n`. For instance, if `L=[1,6,3,9,11]` and `n=8`, then the function should return 6, because 6 is the closest thing in `L` to 8 that is not larger than 8. Don't worry about if all of the things in `L` are smaller than `n`.

11. Write a function called `matches` that takes two strings as arguments and returns how many matches there are between the strings. A match is where the two strings have the same character at the same index. For instance, `'python'` and `'path'` match in the first, third, and fourth characters, so the function should return 3.

12. Recall that if `s` is a string, then `s.find('a')` will find the location of the *first* *a* in `s`. The problem is that it does not find the location of every *a*. Write a function called `findall` that given a string and a single character, returns a list containing all of the locations of that character in the string. It should return an empty list if there are no occurrences of the character in the string.

13. Write a function called `change_case` that given a string, returns a string with each upper case letter replaced by a lower case letter and vice-versa.

14. Write a function called `is_sorted` that is given a list and returns **`True`** if the list is sorted and **`False`** otherwise.

15. Write a function called `root` that is given a number x and an integer n and returns $x^{1/n}$. In the function definition, set the default value of n to 2.

16. Write a function called `one_away` that takes two strings and returns **`True`** if the strings are of the same length and differ in exactly one letter, like `bike/hike` or `water/wafer`.

17. (a) Write a function called `primes` that is given a number n and returns a list of the first n primes. Let the default value of n be 100.

    (b) Modify the function above so that there is an optional argument called `start` that allows the list to start at a value other than 2. The function should return the first n primes that are greater than or equal to `start`. The default value of `start` should be 2.

18. Our number system is called *base 10* because we have ten digits: 0, 1, ..., 9. Some cultures, including the Mayans and Celts, used a base 20 system. In one version of this system, the 20 digits are represented by the letters *A* through *T*. Here is a table showing a few conversions:

10	**20**	**10**	**20**	**10**	**20**	**10**	**20**
0	A	8	I	16	Q	39	BT
1	B	9	J	17	R	40	CA
2	C	10	K	18	S	41	CB
3	D	11	L	19	T	60	DA
4	E	12	M	20	BA	399	TT
5	F	13	N	21	BB	400	BAA
6	G	14	O	22	BC	401	BAB
7	H	15	P	23	BD	402	BAC

    Write a function called `base20` that converts a base 10 number to base 20. It should return the result as a string of base 20 digits. One way to convert is to find the remainder when the number is divided by 20, then divide the number by 20, and repeat the process until the number is 0. The remainders are the base 20 digits in reverse order, though you have to convert them into their letter equivalents.

19. Write a function called `verbose` that, given an integer less than 10^{15}, returns the name of the integer in English. As an example, `verbose(123456)` should return `one hundred twenty-three thousand, four hundred fifty-six`.

20. Write a function called `merge` that takes two already sorted lists of possibly different lengths, and merges them into a single sorted list.

    (a) Do this using the `sort` method.

    (b) Do this without using the `sort` method.

21. In Chapter 12, the way we checked to see if a word `w` was a real word was:

    ```
    if w in words:
    ```

    where `words` was the list of words generated from a wordlist. This is unfortunately slow, but there is a faster way, called a *binary search*. To implement a binary search in a function, start by comparing `w` with the middle entry in `words`. If they are equal, then you are done and the function should return **`True`**. On the other hand, if `w` comes before the middle entry, then search the first half of the list. If it comes after the middle entry, then search the second half of the list. Then repeat the process on the appropriate half of the list and continue until the word is found or there is nothing left to search, in which case the function short return **`False`**. The < and > operators can be used to alphabetically compare two strings.

22. A Tic-tac-toe board can be represented be a 3×3 two-dimensional list, where zeroes stand for empty cells, ones stand for X's and twos stand for O's.

    (a) Write a function that is given such a list and randomly chooses a spot in which to place a 2. The spot chosen must currently be a 0 and a spot must be chosen.

    (b) Write a function that is given such a list and checks to see if someone has won. Return **`True`** if there is a winner and **`False`** otherwise.

23. Write a function that is given a 9×9 potentially solved Sudoku and returns **`True`** if it is solved correctly and **`False`** if there is a mistake. The Sudoku is correctly solved if there are no repeated numbers in any row or any column or in any of the nine "blocks."