



It is important to reuse code.

Use Python Built-in Functions

Suppose you need to find the maximum of two numbers in your program. You might decide that it makes sense to write a function to do this, especially if you are going to need this capability repeatedly in your program. This is simple enough to do, so you write such a function:

```
1 def maximum(num1, num2):
2     """
3     Inputs:
4         num1 - number
5         num2 - number
6
7     Output:
8         Returns the larger of num1 or num2.
9     """
10    if num1 > num2:
11        return num1
12    else:
13        return num2
```


Now, think about the limitations of this function. It will only work if you need the maximum of two numbers. If you later determine that you need to find the maximum of 3, 4, or even more numbers, you will have to either call the function repeatedly with pairs of numbers, rewrite it, or worse write multiple versions of maximum. But what if you had looked in the Python documentation first? There is already a built-in **max** function! From the Python documentation:

```
1 max(iterable, *[, key, default])
2 max(arg1, arg2, *args[, key])
3
4     Return the largest item in an iterable or the largest of two or more
5     arguments.
6
7     If one positional argument is provided, it should be an iterable. The
8     largest item in the iterable is returned. If two or more positional
9     arguments are provided, the largest of the positional arguments is returned.
10
11     There are two optional keyword-only arguments. The key argument specifies a
12     one-argument ordering function like that used for list.sort(). The default
13     argument specifies an object to return if the provided iterable is empty. If
14     the iterable is empty and default is not provided, a ValueError is raised.
15
16     If multiple items are maximal, the function returns the first one
17     encountered. This is consistent with other sort-stability preserving tools
18     such as sorted(iterable, key=keyfunc, reverse=True)[0] and heapq.nlargest(1,
19     iterable, key=keyfunc).
```

Not only would you not have had to write your maximum function at all, this function is already more versatile. It can accept any number of arguments and will return the maximum value (the second form) or it can even take an iterable (like a list or tuple) and return the maximum value within that iterable. Note that it has even more capabilities, such as the ability to take a key function which will be called on each argument and the item that yields the maximum value from the key function will be returned. This key function operates in much the same way as the key function does in **list.sort**. While you could replicate that functionality, the simple **maximum** function you wrote initially would no longer be so simple.

The "info" Dictionaries Used in this Specialization

In previous courses in this specialization, we have used "info" dictionaries to describe information about a particular CSV file. We will do the same in the projects in this class. Why do we do this? Again, this allows our code to be more general and reusable. While this is not the same idea as using a built-in function when you can, it makes it so that the functions you write are more flexible, much as the built-in **max** function is much more flexible than the **maximum** function above. Consider one of the info structures we will use in the projects in this class:



```

1 info = {
2     "gdpfile": "isp_gdp.csv",
3     "separator": ",",
4     "quote": "'",
5     "min_year": 1960,
6     "max_year": 2016,
7     "country_field": "Country Name",
8     "country_code": "Country Code"
9 }
```

This dictionary encapsulates all of the information you will need to know about a CSV file that contains GDP data for different countries around the world. It gives the file name and the format (specifically, which character is used as the field separator and which character is used as the quote character within the CSV file). It also gives further information about the contents of the file. It tells us that it contains data from years between 1960 and 2016 and it tells us that the country names are stored in a column named **Country Name** and that the country codes are stored in a column named **Country Code**.

By passing this info dictionary to functions that must process this CSV file (or the data contained within it), those functions do not have to hard code information about the CSV file. If you later get new data that spans different years or is formatted differently, you will not have to rewrite your code! Instead, you would just need to create a new info dictionary containing a description of the file and then would be able to reuse all of the code you have already written without having to modify it. Foresight and planning like this makes your code more usable in the long run and does not take much extra time when writing it.

Generalizing Your Own Functions

When you write functions, you should be thinking about code reuse. Ask yourself several questions:

- How might you want to use this function in the future?
- Is there an existing function that already does what I want?
- How easy/hard would it be to make this function more general?

Now, you can easily spend forever making every function you write extremely general, so there needs to be limits. Often, programmers first write the simple function they need right now (like **maximum** above. But, if you do that, then the first time you realize that you could reuse that function only if it were more general, you should step back and think about how to redesign it to be more useful not just in the new case, but in a variety of cases.

Don't neglect the second question. If a function already exists, use it! This not only applies to built-in functions, but also to functions you may have already written yourself (for the current program you are writing or maybe even in the past for a different project).

Often just a little bit of thought will allow you to easily make your functions more reusable. Return to our **maximum** function above. We already know we should just use the built-in function **max**, but what if that didn't exist? If we just made it take a list of numbers instead of two numbers, then we could easily reuse it no matter how many numbers we have:

```

1 def maximum(numbers):
2     """
3     Inputs:
4         numbers - List of numbers
5
6     Output:
7         Returns the largest number in numbers
8     """
9     largest = numbers[0]
10    for num in numbers:
11        if num > largest:
12            largest = num
13    return largest
14
15    print(maximum([3, 2]))
16    print(maximum([1, 3, 1, 2, 5]))
```

A little planning goes a long way and can save you a lot of time! Notice, however, that this function is still not perfect. What if the input list is empty, for example? The point is not that this is the ideal implementation, but rather that it is often not that difficult to make your functions more general and flexible.