Introduction

It's time we started how to learn about functions. What are functions? They are blocks of code that are created to do one specific job. When you create or define a function you will write it in such a way that it performs a particular task. When you want to use the function you *call* it. If you have a reason to perform that same task over and over again you just need to call the function created to handle the task, this call tells Python to run the code inside the function. Functions are going to make our coding lives much easier and more productive.

How to define a function

```
In [ ]: def welcome_message():
            """Display a welcome message to the user."""
            print("Good morning!")

        # Function call
        welcome_message()
```

This is the simplest form that a function can take. At line 1 we use the keyword *def* which tells Python that we are defining a function. This is the function definition, it tells Python what the name of the function is, and if necessary what kind of information the function needs to perform its task. This information if required will go inside the parentheses. In this example the name of the function is _welcome*message()*, and there is no additional information required, so the parentheses are empty. The definition ends with a colon (:).

After line 1 we have two indented lines, these two lines of code make up the body of the function. At line 2 we have what is called a *docstring*, which describes what the function does. Docstrings open and close with triple quotes. When Python generates documentation for your programs it looks for these docstrings.

The code at line 3 is the only actual code in the body of this function. As such this function has one job, print the message "Good morning!".

To use this function we call it. When you call a function you are telling Python to execute the code in the function. To call a function you write the name of the function. If needed add additional information in the parentheses. We call the function _welcome*message* at line 5.

Passing information to a function

Let's update the previous code example to print out a welcome message and a name. To accomplish this we will need to enter a name in the parentheses of the function's definition at _def welcome*message()*. By adding *name* here we allow the function to accept any value for name that we specify. The function will now expect a value for *name* each time it is called. Let's take a look:

In [ ]:
```
def welcome_message(name):
    """Display a welcome message to the user."""
    print(f"Good morning, {name.title()}")

welcome_message('Mary')
```

At line 5 we call the function _welcome*message* but this time we pass it the name 'tony' inside the parentheses. If we don't pass a name to the function it will generate a TypeError when it is called. We can call this function as many times as we want and pass it any name we like.

In [ ]:
```
def welcome_message(name):
    """Display a welcome message to the user."""
    print(f"Good morning, {name.title()}")

welcome_message('frank')
```

In [ ]:
```
def welcome_message(name):
    """Display a welcome message to the user."""
    print(f"Good morning, {name.title()}")

welcome_message('jane')
```

Arguments and Parameters

In previous code examples we defined the function _welcome*message()* which required a value for the variable *name*. When we called the function and gave it a name it executed correctly.

The variable *name* in the definition of _welcome*message* is called a *parameter*. A parameter is a piece of information that a function needs to do it job. When we entered a name at the function call such as 'tony' or 'jane' this is called an *argument*. An argument is a piece of information that is passed from a function call to a function.

Exercises

- Write a function that welcomes you home a by name. Call the function with your name and the name of someone else.
- Write a function called _favorite*color* that accepts one parameter, *color*.

Passing arguments

In the previous code example we passed a single parameter to the function definition. In Python a function definition can have multiple parameters. To facilitate this, a function call may need multiple arguments. There are several ways to pass arguments to a function. We can use *positional arguments*, which need to be in the same order the parameters are written. We could use _keyword*arguments*, where each argument consists of a variable name and a value, and lists and dictionaries of values. Let's take a look at *positional arguments* first.

Positional arguments

When a function is called Python has to match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values that match up in this way are called *positional arguments*. Let's take a look at an example where we have a function that describes a car:

In [ ]:
```python
def car_description(car_make, car_model):
    """Dislplay information about a car."""
    print(f"\nI drive a {car_make.title()}.")
    print(f"It's a {car_make.title()} {car_model.title()}.")

car_description('ford', 'focus')
```

In this example we have a function definition at line 1 with two parameters, _car*make*, and _car*model*. When we call the function, _car*description*, we need to provide two arguments, _car*make*, and _car*model*, in that order. We do just that at line 6, where we call the function and pass it two arguments. The first argument *'ford'* is assigned to the parameter _car*make*, and the second argument, *'focus'* is assigned to the parameter _car*model*. In the function body, we use the two parameters to display information to the user.

Multiple function calls

Functions can be called as many times as needed. Take a look:

In [ ]:
```python
def car_description(car_make, car_model):
    """Dislplay information about a car."""
    print(f"\nI drive a {car_make.title()}.")
    print(f"It's a {car_make.title()} {car_model.title()}.")

# Function call
car_description('ford', 'focus')
car_description('porsche', '911')
```

In this example we have a second function call at line 8. As with the previous example, the two arguments passed to the function, *porsche*, and *'911'* are matched to the parameters, _car*make*, and _car*model* in the function definition.

Writing functions and then calling them as we have just done is a very efficient way to work. We create a function once which displays information about our car when passed the correct arguments. Anytime we want to describe a new car we simply call the function again with the new car information. No matter how big the function _car*description* may become we can still call it using just one line of code.

When creating your own functions you can use as many positional arguments as you need. Don't forget that when using positional arguments you need to supply them to Python in the same order as the parameters in the function definition. If you start to get unexpected results from your functions double check your arguments and parameters and the order that you supply them in.

Keyword arguments

In Python, a _keyword*argument*, is a name-value pair that you pass to a function. You associate the name and value within the argument, which means that when you pass the argument to the function there's no confusion. Using keyword arguments stops you from worrying about correctly ordering your arguments in the function call, and they also help clarify the role of each value in a function call. Take a look:

In [ ]:
```
def car_description(car_make, car_model):
    """Dislplay information about a car."""
    print(f"\nI drive a {car_make.title()}.")
    print(f"It's a {car_make.title()} {car_model.title()}.")

# Function call
car_description(car_make='ford', car_model='focus')
car_description('porsche', '911')
```

The function _car*description* hasn't changed. But in this example when we call the function at line 7, we are explicitly telling Python which parameter each argument should be matched with. Now, when Python processes the function call it knows to assign the argument *'ford'* to the parameter _car*make*, and the argument *'focus'* to the parameter _car*model*.

When using keyword arguments in Python the order that you supply them in the function call does not matter. For example:

In [ ]:
```
def car_description(car_make, car_model):
    """Dislplay information about a car."""
    print(f"\nI drive a {car_make.title()}.")
    print(f"It's a {car_make.title()} {car_model.title()}.")

# Function call
car_description(car_model='focus', car_make='ford')
car_description('porsche', '911')
```

Default values

Each time you create a function you have the option of defining a *default value* for each parameter. If you provide an argument for a parameter in the function call, Python uses the argument value. If not, Python will use the parameter's default value. All of this means that when you define a default value for a parameter, you do not need to include the corresponding argument that you would normally write in the function call. Let's take a look at our _car*description* function again.

There may be times when you know that all of the makes of cars will be Fords. If you know this, then you can set the default value of _car*make* to 'ford'. Let's see it in action

In [ ]:
```
def car_description(car_make='ford', car_model):
    """Display information about a car."""
    print(f"\nI drive a {car_make}.")
    print(f"It's a {car_make.title()} {car_model.title()}.")

# Function call
car_description(car_model='focus')
```

What's happening here? We're getting a syntax error which tells us that "non-default argument follows default argument". What we need to do is change the order of the parameters in the function definition. But why? As we have set a default value for _car*make* it is unnecessary to specify an argument for this parameter. The only argument that we need to specify in the function call is _car*model*. Python still interprets this as a positional argument, so if the function is called with just the car model, this argument will be assigned to the first parameter in the function definition. To solve this problem we reorder the parameters in the function definition.

In [ ]:
```python
def car_description(car_model, car_make='ford'):
    """Display information about a car."""
    print(f"\nI drive a {car_make}.")
    print(f"It's a {car_make.title()} {car_model.title()}.")

# Function call
car_description(car_model='focus')
```

New when the function _car*description* is called with no car make argument included, it knows to use the value *'ford'*.

What happens if we include an argument for _car*make* in the function call?

In [ ]:
```python
def car_description(car_model, car_make='ford'):
    """Display information about a car."""
    print(f"\nI drive a {car_make}.")
    print(f"It's a {car_make.title()} {car_model.title()}.")

# Function call
car_description(car_model='focus', car_make='bmw')
```

When using default values, any parameter with a default value needs to be listed after all the parameters that don't have default values. This will allow Python to continue interpreting positional arguments correctly.

Equivalent function calls

So far we've used positional arguments, keyword arguments and default values. You might have already started to experiment with what you have learned and realized that they can all be used together. This is means that there can at times be several equivalent ways to call a function. Let's take a look at just a function definition:

> def car_description(car_model, car_make='ford'):

If we create a function using this definition then an argument always needs to be provided for _car*model*, and this value can be provided using positional or keyword arguments. If the car being described is not a 'ford', then an argument for _car*make* needs to be included in the function call. This argument can also be specified using positional or keyword arguments.

All of the following calls would work with this function:

- car_description('focus')
- car_description(car_model='focus')

- car_description('series 5', 'bmw')
- car_description(car_model='series 5', car_make='bmw')
- car_description(car_make='bmw', car_model='series 5')

Whether you use positional arguments, keyword arguments or default values the choice is yours. So long as you are getting the expected results from your code and your functions are easy to understand it doesn't matter.

Typical argument errors

Every programmer experiences errors in their functions no matter their level of expertise. If you start to get errors don't be too hard on yourself. The typical error when writing functions is an error about unmatched functions. This type of error occurs when you provide less or more arguments that a function requires to complete its task. Let's take a look:

In [ ]:
```python
def car_description(car_model, car_make='ford'):
    """Display information about a car."""
    print(f"\nI drive a {car_make}.")
    print(f"It's a {car_make.title()} {car_model.title()}.")

# Function call
car_description()
```

In this code example we have left out the necessary arguments in the function call. The error is a TypeError and directs us to line 7 of our program. Python also tells us, at the end if the error message that our code is missing 1 of the required positional arguments. Let's take a look at another example of TypeError:

In [ ]:
```python
def car_description(car_model, car_make):
    """Display information about a car."""
    print(f"\nI drive a {car_make}.")
    print(f"It's a {car_make.title()} {car_model.title()}.")

# Function call
car_description()
```

In this example at line 1 of our code we have removed the default value of _car*make*. We have also neglected to add the correct number of arguments to our function call at line 7. Now when we run our program, we still get a TypeError but this time Python tells use the names of the arguments that are missing.

Exercises

- Write a function called _car*order()* that accepts the color and transmission type (manual or automatic) for a car order. The function should print a sentence giving a summary of the car order.
- Using the _car*order()* function you have just created call it using positional arguments. Call it again using keyword arguments.
- Modify your _car*order()* function so that the transmission type is automatic by defualt. Print out a car order confirming that the transmission type is automatic.

Return values

Let's take a look at how to return a value from a function:

In [ ]:
```
def full_name(first_name, last_name):
    """Return a full name"""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

customer_name = full_name('tony', 'staunton')
print(customer_name)
```

How are return values different from functions that simply display their output? Functions don't always have to display their output, instead they can process some data and then return a value or set of values. The value the function returns is called a _return*value*.

In the example above the *return* statement at line 4 takes a value from inside the function and sends it back to the line that called the function. Let's take a closer look at the function above:

- The definition _full*name()* takes as parameters a first and last name, at line 1
- At line 3, the function combines these two parameters, add a whitespace between them, and assigns the result to _full*name*.
- The value of _full*name* is converted to title case, and then returned to the calling line, line 4

When you call a function that returns a value you need to provide a variable that the return value can be assigned to. In the example above the return value is assigned to the variable _customer*name* at line 6.

The output of our function _full*name* is a nicely formatted name consisting of a customers first and last name. I know what you're thinking, 'that seems like a lot of work just to output a name', and you're right but consider an application that collects the first and last name for a million customers and you want to output each name cleanly formatted. You could write:

In [ ]:
```
print("Name One")
print("Name Two")
```

or your could use the function we have just created.

Optional arguments

There will be times when creating functions that you will want to make arguments optional so that when your function is called a user can choose to provide additional information only if they want to. We can use default values to make an optional argument.

Let's take a look at an example of a function that handles addresses. As we are all familiar with when completing address forms online some fields are required and some are not:

```
In [ ]: def collect_address(address_line_1, address_line_2, address_line_3, address_
            """Return a full, formatted address"""
            full_address = f"{address_line_1}, {address_line_2}, {address_line_3}, {
            return full_address.title()

        address = collect_address('1', 'main street', 'dublin', 'ireland')
        print(address)
```

This function works well when it is provided with the four parts of an address. It takes in all four parts and then creates a string from them. The function also adds whitespace and commas where appropiate and converts them to title case.

But as we all know when filling out address forms, sometimes the entire address is not needed, or you just don't want to fill out the entire address form, and this function as it is would not work if we left out any part of the address.

To make the last part of the address optional, we can give the argument _address_line*4* an empty default value, and ignore the argument unless a user provides a value. To make out function work without the fourth address line, we set the default value of _address_line*4* to an empty string and make sure that it is at the end of the list of parameters, which it already is, but watch out for this when creating your own optional arguments.

```
In [ ]: def collect_address(address_line_1, address_line_2, address_line_3, address_
            """Return a full, formatted address"""
            if address_line_4:
                full_address = f"{address_line_1}, {address_line_2}, {address_line_3
            else:
                full_address = f"{address_line_1}, {address_line_2}, {address_line_3
            return full_address.title()

        address = collect_address('1', 'main street', 'dublin')
        print(address)

        address = collect_address('1', 'main street', 'dublin', 'ireland')
        print(address)
```

Nice work! In this code example, the address is created from four possible parts. Because we always want to collect the first three parts of the address, these parameters are listed first in the function's definition. The fourth address line is optional and so it is listed last in the definition, and its default value is an empty string.

At line three of the function we check to see if a fourth address line has been provided. Python interprets non-empty strings as *True*, so, _if_address_line*4* evaluates to *True* ,if a fourth address line argument is in the function call. If a fourth address line is provided, it is combined with the three other address lines to create a full address. This address is then changed to title case and returned to the function call line where it's assigned to the variable *address* and then printed out. If no fourth address line is provided, the empty string fails the *if* test at line 3 and the *else* block runs at line 5. The full address is created with _address_line*1*, _address_line*2*, _address_line*3* and the formatted address is returned to the calling line where is assigned to the variable *address* and printed out.

Calling this function with 3 address lines is straight forward. If you are using a fourth address line, you have to make sure that _address_line*4* is the last argument passed so Python will match up the positional arguments correctly, line 12.

Returning a dictionary

As we have discussed in previous lessons Python has several types of data and dictionaries can return many of them including lists and dictionaries. Let's take a look at a function that returns a dictionary:

```
In [ ]: def build_character(first_name, last_name):
            """Return a dictionary that contains the name of a character."""
            character = {'first' : first_name, 'last' : last_name}
            return character

        hero = build_character('james', 'bond')
        print(hero)
```

In this code example we create a function called **build_character** at line 1. This function takes in a first and last name. At line 3 it places these values in a dictionary.

The value of **first_name** is saved with the key 'first', and the value of **last_name** is saved with the key 'last'. The dictionary, in full, is returned at line 4. This return value is printed out at lines 8 and 9.

The **build_character** function receives information in the form of first and last name which is just text information. It then puts this information into another data structure, a dictionary which we can further work with. Our function has labeled the strings 'james' and 'bond' as first name and last name. We could extend this function to accept additional values such as occupation. Take a look:

```
In [ ]: def build_character(first_name, last_name, occupation=None):
            """Return a dictionary that contains the name of a character."""
            character = {'first' : first_name, 'last' : last_name}
            if occupation:
                character['occupation'] = occupation
            return character

        hero = build_character('james', 'bond', occupation='super spy')
        print(hero)
```

Our function definition has a new parameter, **occupation**. It has been assigned the special value, **None**, which is used when there is no specific value assigned to a variable. **None** is a placeholder value which evaluates to **False** inn conditional tests.

If we call the **build_character** function with a value for occupation it will be stored in the dictionary.

Using a while loop with a function

Now, let's create a new function that works with a **while** loop. In the code example below I a going to reuse the **full_name** function which we created earlier.

In [ ]:
```python
def full_name(first_name, last_name):
    """Return a full name"""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# This will create an infinite loop
while True:
    print("\nWhat is your name: ")
    firstname = input("First name: ")
    lastname = input("Last name: ")

    formatted_name = full_name(firstname, lastname)
    print(f"\nHi, {formatted_name}")
```

In this code example the **while** loop asks a user to enter their name. Our program asks a user to enter their first and last names at line 8.

If you have run this code then you will have noticed there is an issue with it. There is no quit condition and it keeps going. Our program should allow a user to quit. Let's use a **break** statement which will allow a user to quit.

In [ ]:
```python
def full_name(first_name, last_name):
    """Return a full name"""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# This will create an infinite loop
while True:
    print("\nWhat is your name: ")
    print("Enter 'q' to quit")

    firstname = input("First name: ")
    if firstname == 'q':
        break

    lastname = input("Last name: ")
    if lastname == 'q':
        break

    formatted_name = full_name(firstname, lastname)
    print(f"\nHi, {formatted_name}")
```

In this example we have given the user a message telling them how to quit the program. If the user enters 'q' then we break out of the loop.

Exercises

- Write a function take can receive members of your family and their role in your family, for example brother, sister, cousin. Your function should return a string similar to "Frank Smith, cousin".
- Write a function called my_books(). This function should build a dictionary which contains information about your books (or books you like). The function should receive a book name and the name of its author and it should return a dictionary containing these two pieces of information. Make sure that you print out the return value.

- Update the function **my_books()** to use **None** as an optional parameter. In this parameter store the book genre such as business.
- Using the function **my_books()** add a **while** loop that will ask users to enter a book and author name. Once you have received the information call **my_books()** with the users input and print out the dictionary that is created. Don't forget the quit option!

Passing a list to a function

As you work on different Python projects you will start to work with all kinds of lists, phone numbers, products, and much more. To make your work easier and more productive it is good to know how to pass a list to a function. When you pass a list to a function, that function can access the contents of the list. Let's get started.

Let's go back to our list of books and use this list to print out a message telling us if a book fro the list is in stock or not.

In [ ]:
```
# Book stock function
def book_stock(books):
    """Print a message to let us know if a book is in stock"""
    for book in books:
        status = f"{book.title()}, is currently in stock"
        print(status)

book_catalog = ['elon musk', 'steve jobs', 'getting things done']
book_stock(book_catalog)
```

In this code example we first define the function **book_stock()**. This function expects a list of books which it assigns the parameter **books**. The function loops through the list it receives and then prints a messages telling us if a book is in stock. You can see at line 8 that we create or define a list of books and then pass the list **book_catalog** to **book_stock()** in our function call.

Changing or Modifying a list in a function

After you pass a list to a function, the function can modify that list. Changes made to the list inside a function are permanent.

Let's stay with our books list but this time consider the scenerio where we have two lists. The first list is all of the books in a customers shopping cart, and after the items have been paid for they are moved to a second list. We will first do this without using functions:

In [ ]:
```
# A list of books in a customers shopping cart
shopping_cart = ['elon musk', 'steve jobs', 'getting things done']
orders_complete = []

# Simulate ordering each item, until none are left
# move each purchased item to the list order_complete after purchase
while shopping_cart:
    current_order = shopping_cart.pop()
    print(f"Processing payment for: {current_order}")
    orders_complete.append(current_order)
```

```python
# Display a completed order
print("\nThe following items have been successfully ordered:")
for order_complete in orders_complete:
    print(order_complete) #Add title() method
```

This great little program does a lot. Its starts with a list of books that are in a customers shopping cart that have yet to be purchased. We then have an empty list called **orders_complete** which will receive each book after it has been purchased. As long as books remain in **shopping_cart**, the **while** loop at line 7 simulates a customer purchase by removing a book from the end of the shopping_cart list and storing it in the variable**current_order**. We then display a message telling a customer that the our online store is processing payment for a particular item. Then, our program adds the book the list **orders_complete**. When the loops has finished running (which it does when there are no more items in the shopping_cart list, a list of books that have been ordered is printed out.

The code above can be refactored (to make better) by writing two functions, each of which will perform a specific task. The first function we create will handle payment processing, and the second will provide our customers with an order summary:

In [ ]:
```python
def shopping_cart(unordered_books, orders_complete):
    """
    Simulate purchasing each book until there are none left to purchase
    Move each book ordered to orders_complete for printing.
    """
    while unordered_books:
        current_order = unordered_books.pop()
        print(f"Processing payment for: {current_order}")
        orders_complete.append(current_order)

def display_completed_orders(orders_complete):
    """Show all books that have been purchased."""
    print("\nThe following books have been successfully purchased:")
    for order_complete in orders_complete:
        print(order_complete)

unordered_books = ['elon musk', 'steve jobs', 'getting things done']
orders_complete = []

shopping_cart(unordered_books, orders_complete)
display_completed_orders(orders_complete)
```

In this code example we start off at line 1 by defining a function called **shopping_cart()**, which has two parameters, a list of unordered books and a list of orders that are complete. When the function is passed these two lists it simulates processing the payment for each book by emptying the list of unordered books and filling up the list of completed orders.

Our second function which is defined at line 11, is called **display_completed_orders()** and has one parameter, the list of orders complete. When provided with this list, the function **display_completed_orders** displays a message informing a customer that their item(s) have been successfully purchased.

As you can see this new program has the same output as the previous version without functions, however this code is better organized and easier to understand. If you start to share your code with other developers they will find this version much easier to understand and maintain. If you come back to this code after several months or even years you will also find it easier to understand and remember why you wrote it in such a way.

At line 17 we created a list of unordered books. Then at line 18 we created an empty list called orders_complete. Because we have already defined our two functions all we have to do is call them and then pass the right arguments. First we call **shopping_cart()** and pass to it the two lists that it needs. The shopping cart function simulates payments processing. Next, we call the **display_completed_orders()** function and pass to it the list of completed orders so that it can show a summary of the books that have been purchased.

This code is easier to maintain than the previous version. If we need to update a piece of code, the payments process message for example, we can change it once and the change will work anywhere that the function is called. We can also use our functions in other parts of the program. If we were to extend our program to save a list of unordered books for purchasing at a later date.

You may not have noticed, but each of our functions performs just one task. This is a coding best practice. A function should only have one specific job.

Stopping a function from changing a list

There will be times when you do not want a function to change a list. In our previous example we moved items that had been ordered to a list of completed orders after payment had been processed. But what about a feature that shows a customer their shopping cart history? To display this list you need to keep the original list of unordered items. But because we moved all the ordered items out of the **shopping_cart** list this list is now empty. And the empty version is all we have, the original is gone.

To solve this problem we can pass a function a copy of a list, not the original. This way, any changes that are made will only affect the copy and leave the original untouched. Let's take a look.

To send a copy of a list to a function we can use the code

In [ ]:
```
function_name(list_name[:])
```

In this line of code we are using slice notation to make a copy of the list to pass to the function. If we didn't want to empty the list of unordered books in the previous example, we could call **shopping_cart()** like this:

In [ ]:
```
shopping_cart(unordered_books[:], orders_complete)
```

By using this method the function **shopping_cart()** can still do its work because it is still receiving the names off all unordered books, only this time it is using a copy.

This method works and can be useful but it is more efficient for a function to work with an existing list which will avoid using the time and memory needed to make separate copy. This is relevant when you start to work with large lists and datasets.

Exercises

- Create a list of products. Pass the products list to a function called **in_stock()**. Have this function print out a message to a customer informing them that the products in your list are in stock.
- Build on the previous exercise and add a function called **items_dispatched()**. Use this function to receive items from the **in_stock()** and print a message informing the customer that their order products have been shipped and are in transit.
- Build on the previous exercise and call the function **items_dispatched()** using a copy of the list passed to it.After calling the function print out both lists to show that the original list has retained its items.

How to pass an arbitrary number of arguments

You won't always know how many arguments a function needs to accept. Python allows you to define functions that can receive an arbitrary number of arguments from the calling statement.

Consider a website that allows customer to customize a car before they purchase it. This website will need a function that builds a car and accepts a number of extras such as leather seats. The function won't know in advance what extras a customer may want to add. The function we are about to create will have one parameter, *extras. This single parameter can collect as many arguments as the calling statement provides. Have a look:

In [ ]:
```python
def assemble_car(*extras):
    """Print a list of extras that a customer has ordered."""
    print(*extras)

assemble_car('base line model')
assemble_car('leather seats', 'sat-nav')
```

In this code example when we define our function at line 1, we define it with one parameter ***extras**. This single parameter tells Python to create an empty tuple called **extras** and add whatever value(s) it receives from the calling statement into this tuple. The print statement in the body of the function shows that the function **assemble_car** can handle a function call with a single extra or a call with several extras.

Now let's replace the print() statement with a loop that will run through the list of extras and print them to screen for customer confirmation.

In [ ]:
```python
def assemble_car(*extras):
    """Print a summary of extras ordered by the customer."""
    print("\nOrder a car with the following extras:")
    for extra in extras:
        print(f" - {extra}")

assemble_car('fog lamps')
assemble_car('leather seats', 'sat-nav')
```

How to mix positional arguments and arbitrary arguments

It is possible for a function to accept different types of arguments. The parameter that accepts an arbitrary number of arguments must be placed last in the function definition. Python will match positional and keyword arguments first and then collects any remaining arguments in the final parameter.

Let's take a look at an example which will make this easier to understand:

In [ ]:
```
def assemble_car(body_type, *extras):
    """Print a summary of the car we are about to assemble."""
    print(f"\nA {body_type} car with the extras outlined below is moving to
    for extra in extras:
        print(f" - {extra}")

assemble_car('four door', 'leather seats')
assemble_car('two door', 'sat-nav', 'fog lamps')
```

In this example, in the function definition, Python assigns the first value it receives to the parameter **body_type**. Every other value that comes after this are stored in the tuple **extras**.

How to work with arbitrary keyword arguments

So far everything that we have done has seemed very structured. We have known the position of arguments and their number. But there will be times when you are creating programs when you will need to be able to accepts an arbitrary number of arguments. In these instances you probably won't know what kind of information will be passed to your function. When you need this kind of flexibility, Python allows you to write functions that can accept as many key-value pairs as are in the calling statement.

Imagine you are hiring a new employee and will be creating a new profile for that person, you know that you are going to have information about the employee, you just don't know what kind of information you will receive. Let's take a look:

In [ ]:
```
def employee_profile(firstname, lastname, **employee_info):
    """Create a dictionary of information about a new employee."""
    employee_info['first_name'] = firstname
    employee_info['last_name'] = lastname
    return employee_info

new_employee = employee_profile('tony', 'staunton', area='project management

print(new_employee)
```

In this example the function **employee_profile()** takes a first and last name and it accepts an arbitrary number of keyword arguments. The double asterisks before the parameter ****employee_info** forces Python to create an empty dictionary called **employee_info**. Python will place whatever key-value pairs it receives into this dictionary. Inside the function you can access the key-value pairs in **employee_info** just like a normal dictionary.

In the body of the function **employee_profile()** we add the first and last name to the **employee_info** dictionary because we'll always receive these two pieces of information from a new employee at line 3, and at this point they have not yet been added to the employee dictionary. At line 5 we return the **employee_info** dictionary to the function call line.

At line 7 we call the function **employee_profile()**, passing it information about a new employee. The returned profile is assigned to **new_employee** and then printed out.

This function will work regardless of how many key-value pairs are provided to it in the function call.

You can mix positional, keyword, and arbitrary values in many different ways when creating your functions. As you explore more Python you will begin to see how they are mixed. It can take practice to mix and correctly use different types of arguments so don't be too hard on yourself if you struggle in the beginning. Remember, the simplest approach is always the best.

Slow is smooth, and smooth is fast.

Storing function in modules

As you start to learn more about functions and come to use them more often you will begin to see just how helpful they can be. Functions can help to make your code easier to understand, read and maintain. There will come a point along your programming journey when you can store your functions in a separate file called a **module**. When you want to use a module in your program you **import** into it your main program. The **import** statement is what Python uses to make your module available in the main or current program.

Moving your functions into a separate file will allow you to create separations in your code that will help you write better programs. If you have a function that prints a statement then you don't have to worry about that code in your main program. You just import it and know that it works. This will free you up to focus on making your main program do its job. Not only will it free up your programs from unnecessary code bloat but you will be able to use your functions in other programs. In time you may even want to share your programs with other developers.

Let's take a look at the ways of importing a module into your main program.

Importing an entire module

If we are going to import functions then we first need a module. Again, a module is a file that holds one or more of your functions that is separate from your main program. In Python a module ends with the file extension **.py**. Let's create a module that contains the function **new_employee()**.:

In [ ]:
```
#def new_employee(name, role, *qualifications):
#    """Summarize the role and qualifications of a new employee."""
#    print(f"\n{name} is joining as a {role}, with the following qualificati
#    for qualification in qualifications:
#        print(f" - {qualification}")
```

Once we have a our function created we need to save it in a separate file in the same directory as our main program. Let's save our function in a file called new_hire.py

In [2]:
```
import new_hire

new_hire.new_employee('tony', 'project manager', 'Python', 'Java')
```

```
tony is joining as a project manager, with the following qualifications:
 - Python
 - Java
```

In the code example above when Python sees the line *import new_hire* it opens the file *new_hire.py* and copies all the functions from it into the main program. All of this happens behind the scenes, you don't see anything, you just know that it has happened. You will know that the import has been successful because your code will be running as expected.

At line 3 we are calling a function from an imported module. We do this by entering the name of the module we imported, *new_hire*, followed by the name of the function, *new_employee*, separated by a dot. The output of this code is the same if the full function was typed directly into the main program.

The technique that we have just used to import a module makes every function from that module available to our program. The *import* statement as we have just used it imports the full module which includes all functions. We can import specific functions from a module into our programs.

Importing specific functions

from module_name import function_name

This syntax is how to import a specific function from a module. We can import as many functions as we need from a module by separating each function name with a comma.

from module_name import function_a, function_b, function_c

To import our new_hire function using this syntax we would do the following:

In [3]:
```
from new_hire import new_employee

new_employee('tony', 'PM', 'Python', 'C#')
```

```
tony is joining as a PM, with the following qualifications:
 - Python
 - C#
```

As you can see at line 3, we don't need to use the dot notation when we call the function. This is because we have explicitly imported the function *new_employee* in the import statement.

Providing a function with an alias

There will be times when the name of a function that you are importing might conflict with an existing name in your program. Or there may be times when a function name is too long and you don't want to type out the whole thing. In either of these cases we can use an *alias* which is a like a nickname for a function.

Using the code from before let's give our *new_employee* function an alias, *ne()*. We do this by importing *new_employee* as *ne*.

In [4]:
```
from new_hire import new_employee as ne

ne('tony', 'PM', 'Python', 'C#')
```

```
tony is joining as a PM, with the following qualifications:
 - Python
 - C#
```

In this code example we have renamed the function *new_employee()* to *ne()*. Now any time that we want to call *new_employee()* we can just use its alias, *ne()*.

from module_name import function_name as fn

Providing a module with an alias

Functions are not the only thing that you can provide an alias to, you can also give a module an alias. Looking back at the previous code example where we imported *new_hire*, we give give the module name an alias like e for employee. Taking this approach to module alias naming can allow you to call a module's functions more quickly.

In [5]:
```
import new_hire as e

e.new_employee('tony', 'PM', 'Python', 'Java')
```

```
tony is joining as a PM, with the following qualifications:
 - Python
 - Java
```

In this code example the *new_hire* module is given the alias *e*. It's important to point out that all of the function and their names within the module remain unchanged.

The syntax is: import module_name as mn

Importing all functions of a module

We can tell Python to import all the functions inside a module by using the asterisk symbol:

In [6]:
```
from new_hire import *

new_employee('tony', 'PM1', 'Python', 'Java')
```

```
tony is joining as a PM1, with the following qualifications:
 - Python
 - Java
```

In this code example the asterisk in the *import* statement tells Python to copy all functions inside the module *new_hire* and make them available in the program. As all functions are imported we can call a function by name without using dot notation. In general, and in particular when working with modules that you are unfamiliar with, it is better not to use this approach. Although I do sometimes use it when I'm feeling lazy. The reason why it's best to stay away from this approach is because if you are working with a module and it has function names that are the same as yours you can get unexpected results.

The best practice is to import the function(s) that you want or import an entire module and use dot notation. This will help reduce bugs in your code, make it more readable and easier to understand.

Syntax: from module import *

Exercises

- Create a function that provides a greeting to a user and then place that function in a separate file. Don't forget that your new file should end with *.py*
- Create a new program with an import statement to import your new module and function and then call your function
- Modify your program to import the module and function using the various methods we have discussed.