



Introduction

Functions allow programs to perform repeated tasks following a set of instructions contained within the function. They are isolated blocks of code capable of receiving values to work with (arguments) and can return a value if needed.

Often we need to perform a task repeatedly, but writing the code over and over again breaks a fundamental principle in programming known simply as the **DRY** principle:

DON'T REPEAT YOURSELF

Repeated sections of code that perform the same task should be 'housed' in a function that can be called when needed

There are various reasons for this:

- It makes the program easier to read and follow
- It speeds up the rate of development of programs
- Well-defined functions that are reliable allow a programmer to create more reliable programs
- Changes needed to the logic of a function can be made in one place, ensuring the change carries through the entire program
- Functions allow programmers to break complex applications or logic down into smaller and more manageable steps

First we consider declaring and calling a function in Python correctly and then explore their usage

Declaration

To declare a function we use the **def** (define) keyword followed by the name of the function. A function's name should be unique within its scope, concise and describe the task the function will perform. The function name must be followed with parenthesis (even if there are no arguments). Within the parenthesis, the parameters for the function are defined

```
def sum( number_1 , number_2 )
```

A function needs to perform a task, we can take our function above and give it the capability to perform addition and return the sum value

```
def sum(num1, num2):
    return num1 + num2
```

In Python, we declare inner blocks of execution with the use of indentation (standard is 4 spaces)

Consider a function that we want to use to just display a Menu:

```
def print_menu():
    print("""Welcome to this Calculator Application.

The following operations can be performed:
- Addition
- Subtraction
- Multiplication
- Division""")
```

Notice that this is a function with no parameters, but parenthesis are still needed

Calling a Function

Consider writing a simple program as shown below:

```
def print_menu():
    print("""Welcome to this Calculator Application.

The following operations can be performed:
- Addition
- Subtraction
- Multiplication
- Division""")

def add(num1, num2):
    return num1 + num2

def subtract(num1, num2):
    return num1 - num2

def multiply(num1, num2):
    return num1 * num2

def divide(num1, num2):
    return num1 / num2
```

NOTE:

The code has been kept simple for demonstration purposes, however there are problems with it:

- No docstrings
- Defensive Programming has not been used such as Type Checking, ensuring division by zero does not happen

If we leave the program as it is and then run the program, there will be no output. The reason is that we may have defined the functions, but to use them we have to **call** them.

To call a function, we simply enter its name and then pass in values for the parameters it may require. The parenthesis **must** be used in the call even if there are no arguments

Calling the `print_menu()` function:

```
# Print the Menu to the User
print_menu()
```



```
Welcome to this Calculator Application.

The following operations can be performed:
- Addition
- Subtraction
- Multiplication
- Division
```

Calling the `add()` function

```
# Perform an addition
sum = add(5, 10)
print(f"The Sum of 5 and 10 is: {sum}")
```



```
The Sum of 5 and 10 is: 15
```

In the above `add` function, the values 5 and 10 are provided for the required **parameters** `num1` and `num2` of the function `add()`. These values are used to perform addition (as is the logic of the function) and the value 15 is then **returned**.

Here we can see some of the logic for the execution of this program. As a programmer, it is important to be able to trace the execution steps that a program follows

1. The Main Program declares all the functions we had defined, allocating space in memory for them.
2. A function call is made (when we said `add(5, 10)`)
3. Our program 'moves into' the function. The variable 'sum' will not yet be instantiated with a value until the function call has been completed
4. We are now within the function call, the main program is waiting for a result. The summing of 5 and 10 is (and can be) performed and the result of this (15) is returned back to the main program
5. We are now back at the level of the main program. The value 15 is now assigned to the variable 'sum'
6. The `print()` function is called and the value for `sum` is printed to the console.
7. With nothing else to do, the program terminates.

A parameter of a function always needs a value when called. Failing to provide a value results in a `TypeError`

Correct

```
def divide(num1, num2):  
    return num1 / num2  
  
result = divide(4, 2)  
print(result)
```

TypeError - Missing an Argument

```
def divide(num1, num2):  
    return num1 / num2  
  
result = divide(4)  
print(result)
```

However, we can supply **default values** to function parameters that will be used if we do not provide an argument to the function.

We will modify the above `division()` function to include a third parameter called 'quotient_only'. This means we will give additional functionality to the function allowing it to return the decimal answer from the division or to just give the Whole Number component, but we will supply a default value to this parameter as seen below. The default value is specified within the parenthesis.

```
def divide(num1, num2, quotient_only = False):  
    if quotient_only:  
        return int(num1 / num2)  
    else:  
        return num1 / num2  
  
result1 = divide(5, 2)  
result2 = divide(5, 2, True)  
  
print(result1)  
print(result2)
```

The statement: 'if quotient_only' is equivalent to 'if quotient_only == True'

By using a Type Cast, we can cast the decimal to an integer which removes the decimal component

→ 2.5
2

With a default value, the function will use the default if a value is not supplied during the function's call. We can see this above with `result1`

The `print()` function already makes use of this.

In the above example, we can ask 'why were the two outputs printed on different lines?'

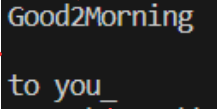
The reason is that the built-in `print()` function contains more parameters than we have used (or at least have specified values for). The `print()` function has a parameter called 'end' with a **default** value set to the character "\n". This special character is the newline character and is the reason each `print()` output to the console occurs on a newline

The cool thing about default values is that they can be changed. For this exercise we will change the default value for the 'end' parameter of the print() function

```
# set the end to be the number 2
print("Good", end = "2")

#set the end to two new line characters instead of one
print("Morning", end = "\n\n")

# set the end to be an underscore
print("to you", end = "_")
```



Good2Morning
to you_

Argument Order

Usually, the order in which the parameters are given in the function declaration is the same order in which we need to supply values for those parameters

Consider the function below that simply prints the values it is given

```
def print_message(first, second, third):
    print(first + " " + second + " " + third)

print_message("One", "Two", "Three")
```

→ One Two Three

Notice that the value 'One' was supplied to the parameter 'first', 'Two' was supplied to the parameter 'second' and 'Three' was supplied to the parameter 'third'

To prove this, we change the order of the values and notice how the function maintains the order of the first, second and third parameter

```
def print_message(first, second, third):
    print(first + " " + second + " " + third)

print_message("Three", "One", "Two")
```

→ Three One Two

What happens if we do not want the function to use the arguments (values) in the order they were supplied?

It is common to want to change the order of the arguments. Some functions take quite a few arguments and trying to remember the order for all of them is difficult; needing to constantly double check the function's declaration can also waste time


Python has functionality to allow us to change the order of the arguments

Note: The Interpreter cannot ‘just know’ when we want to change the order of the values we are supplying and therefore determine on its own which value is meant for which parameter.

To tell the interpreter which value is meant for which parameter, we specify the name of the parameter when we call the function. In doing so, we can supply values in the order that we want. We will also look at combination types where we want the order to be considered as well, but in a moment.

```
def print_message(first, second, third):
    print(first + " " + second + " " + third)

print_message(third="Three", first = "One", second="Two")
```



One Two Three

In the function call above, we can see that we are specifying which values we want to be passed to which parameters. This allows us to control the order of the arguments passed to the function.

Now we want to consider using the order in which values are given as well as specifying values for specific arguments:

```
def print_message(first, second, third):
    print(first + " " + second + " " + third)

print_message(third="Three", "One", "Two")
```


Above, the goal was to specify a value for the ‘third’ parameter and then have the remaining values ‘One’ and ‘Two’ be given to the remaining arguments in the order in which they appear. We can do this but it has not been implemented correctly above and results in a `SyntaxError`

To correctly use the order of parameters **and** allow our program to specify values for a specific parameter, we have to follow a correct order when doing so



1. arguments to be used in order must appear first
2. values to be given to a specific parameter must come after values without a specified parameter

```
def print_message(first, second, third):
    print(first + " " + second + " " + third)

print_message("One", "Two", third="Three",)
```



One Two Three

 non-specified
  specified

Why is this useful?

7

Consider being given a task to create a simple, probability function. Probability (the likelihood that something will happen) can be expressed as a decimal, fraction and percentage. It is common for different people to prefer different representations and we would like our function to give the probability value in a form the user prefers. It is also likely that different levels of precision in the number of decimal places may be desired by the user. The function below will take this into account

Please Note: The implementation of the solution below is just one of many ways to solve this

```
def determine_probability(desired_outcome, total_outcomes,
                          representation = "Percentage", precision = 2):
    """
    Function to calculate the probability of an event with specification for
    representation form and desired precision.

    :param desired_outcome: count of desired outcomes for an event
    :type desired_outcome: int
    :param total_outcomes: count of total, possible outcomes
    :type total_outcomes: int
    :param representation: desired form of answer ('Percentage'(default), 'Decimal')
    :type representation: string
    :param precision: desired decimal places (0 or positive int)
    :type precision: int

    :returns: probability result or error message
    :rtype: string
    """
    try:
        if int(precision) < 0:
            return "Invalid Precision"
        elif precision == 0:
            # round function requires precision as 'None' to remove all decimals
            precision = None

        # attempt to determine probability value
        result = int(desired_outcome)/int(total_outcomes)

        if representation == "Percentage":
            # round to desired number of decimal places after percentage conversion
            result = round(result * 100, precision)
            return str(result) + "%"

        elif representation == "Decimal":
            # round result to desired precision
            return str(round(result, precision))
        else:
            return "Invalid Answer Representation supplied"

    except (TypeError, ZeroDivisionError, ValueError):
        # caused by division by zero, invalid type given such as a letter
        # instead of number
        return "Invalid Input"
```


1. The function uses a try-except block to attempt to perform conversions and calculations. If invalid inputs have been received (such as a letter instead of a number) or a division by zero occurs, then an error is raised. The 'except' block can catch these errors and perform an action such as display a message to the user instead of the program terminating. This is known as **handling an error gracefully**.
2. The in-built **round()** function has been used to perform the rounding of decimals needed
3. The orange colored lines just below the function declaration are known as a **docstring**. These lines follow a specific format on the accepted manner in which to explain the purpose and actions of the function, the parameters it expects and what the function will return. Correct format and styling for Python Programs can be found from the **PEP-8** guide available from the Python website.

Using the function:

We will now attempt to perform probability calculations using the function we have created

1. Consider the Probability of getting a 'Tails' from a single, fair coin toss. We have one desired outcome (Tails) and a coin has two possible outcomes (Heads and Tails). The probability for this event is 1 out of 2 which is 50% as a percentage and 0.5 as a decimal

```
# retrieve answer as a Percentage
print(determine_probability(1, 2))

# retrieve answer as a Decimal
print(determine_probability(1, 2, representation = "Decimal"))
```

→ 50.0%
0.5

2. Now consider a person randomly choosing a whole number between 1 and 7. We want to determine the probability that the number this person has chosen is greater than or equal to 4 (four desired outcomes - 4, 5, 6, 7). We want the answer as a **decimal** and the number to be rounded to various decimal places (we will do multiple function calls)

```
# one decimal place
print(determine_probability(4, 7, representation="Decimal", precision = 1))

# two decimal places - the default precision value and does not need to be specified
print(determine_probability(4, 7, representation="Decimal"))

# three decimal places
print(determine_probability(4, 7, precision = 3, representation="Decimal"))
```

→ 0.6
0.57
0.571

Notice in the first function call:

The **representation** argument was given first and the **precision** argument was given second

Then notice in the third function call:

The **precision** argument was given first and the **representation** argument was given second

This demonstrates a potential use for the specification of parameters in a function call in that we are aware of **what** parameters a function has and **how** we can use them instead of having to also focus on ensuring we provide them in the correct order (but remember non-specified arguments must come first). In addition, this makes the function calls much clearer for a person to read and understand.

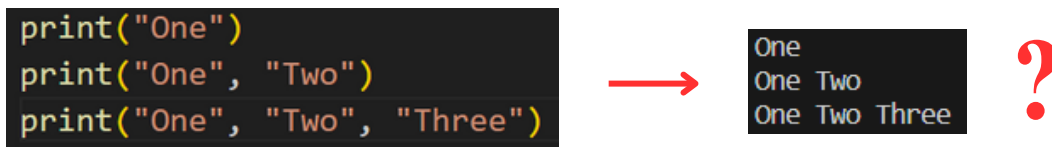
*args and **kwargs

We have completed multiple examples declaring and calling functions. We have seen the use of default values, the ability to specify which values belong to which parameter and have even taken argument order into account.

In these examples, we have seen that we need to give a value for every parameter (be it supplied as an argument or set as a default value which we can change). **But**, what if we do not know ahead of time the number values that will be supplied? We need a way for our functions to deal with a varied number of arguments that it may receive.

To introduce this, we will consider the **print()** function again.

Consider this example:



```
print("One")
print("One", "Two")
print("One", "Two", "Three")
```

→


```
One
One Two
One Two Three
```

?

1. If you look at the output for the second and third function calls, there are single spaces between each string **but** we did not specify any 'space' characters in the original strings. How did the `print()` function *know* to add these characters?
2. I have used the same function in all three calls, yet I was able to give one, then two and then three parameters and the function still worked

The **print()** function has another parameter called “**sep**” which is used to specify the separator character that we want between the parameters we supply. The default is “ ” (an empty space) which is the reason for our final, output strings containing spaces between the words given. As is the case with default values, we can change this value:

```
print("Learn", "Practice", "Ask for Help", sep=" and ")
```




```
Learn and Practice and Ask for Help
```

with the separator set to “ **and** ” we can specify how we want the parameters supplied to be separated from one another in the final output

Varying the number of Parameters

Now we want to understand why the **print()** function was able to take varying numbers of parameters, but if I try to do the same in the example below then I would get a **TypeError** in my program

```
def add(num1, num2):  
    return num1 + num2  
  
print(add(1, 2, 3))
```



```
TypeError: add() takes 2 positional arguments but 3 were given
```

Python has two special types of parameters declared with the use of a single asterisk (*) or double asterisk (**). The default names to use for these parameters are ***args** and ****kwargs**. The use of these names is not compulsory, but these are the usual names.


***args**

A parameter specified with a single asterisk (*) informs the interpreter that there are multiple values that would be given in the function call and they can all be found under **args**. It is this type of parameter that the **print()** function uses to allow it to accept multiple parameters and we will now implement it in our **add()** function. The name of this parameter in the **print()** function is ***values**

```
def add(*args):
    sum = 0
    for number in args:
        sum += number

    return sum

print(add(1, 2, 3))
print(add(10, 20, 30, 40, 50, 60))
```



6
210

Our `add()` function now understands that multiple arguments will be given when the function is called and now has the capability to accept them. Notice that we use a 'for ... in' loop to work through each argument and perform some action with the values (in this case we sum them)

****kwargs** - key word arguments

In Python, we also have the ability to work with **key-value** pairs. This can be seen with the **dictionary** data type.

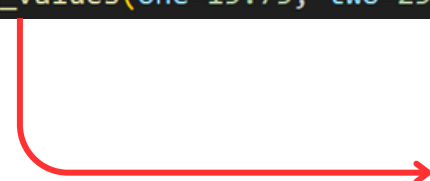
Consider wanting a function to accept key-value arguments, but also without knowing how many of these may be supplied when the function is called. Here we make use of ****kwargs**. Note that this is recognized **not** by the word *kwargs*, but by the double asterisks ******. In addition, prior knowledge of the **Dictionary** data type would be beneficial.

```
def sum_values(**kwargs):
    sum = 0
    print("Products being summed:")
    for product_id, value in kwargs.items():

        print(f'Product: {product_id}, Price: R{value}')
        sum += value

    print(f"\nTotal Value: R{sum}")

sum_values(one=15.75, two=25.59, three=36.00)
```



```
Products being summed:
Product: one, Price: R15.75
Product: two, Price: R25.59
Product: three, Price: R36.0
Total Value: R77.34
```

***args and **kwargs with other parameters**

12


We may want a function to also work with other parameters in addition to *args and **kwargs. This can be done, **but** the *args and **kwargs **must** come last in the function declaration. This is how the function can differentiate between parameters that belong to *args and **kwargs and those that do not.

Consider the previous `sum_values()` function, but we may want to add in a running total. We may have a value of other products already and would like to include this value when we call our function

```
def sum_values(running_total = 0, **kwargs, ):
    sum = running_total
    print("Products being summed:")
    for product_id, value in kwargs.items():
        print(f'Product: {product_id}, Price: R{value}')
        sum += value

    print(f"\nPrevious Total: R{running_total}")
    print(f"New Total Value: R{sum}")

sum_values(100, one=15.75, two=25.59, three=36.00)
```



```
Products being summed:
Product: one, Price: R15.75
Product: two, Price: R25.59
Product: three, Price: R36.0

Previous Total: R100
New Total Value: R177.34
```

In the function call we put the running total value first which is then passed to 'running_total' parameter. The remaining values are given to **kwargs

Purpose Note:

We know that to use a function, we have to **call** it and this includes passing values to the function if they are needed. We also said in the very beginning that functions are very good at breaking down a complex problem into smaller, more manageable steps and allow us to prevent a repetition of code.

Repeated code is not just a problem in terms of increasing the amount of written code. Repeated code = repeated logic and if we decide to make a change to the logical steps we are following, then we have to make the same change in **every place** that we had repeated logic. Consider a very large application made up of thousands of lines of code and multiple files. Trying to find every place the logic has been repeated (without just creating a callable function for it) and then reliably make the same change to every section would be a monumental task.

Functions prevent this from happening, but different functions may also use identical, internal logic steps and this goes back to the initial problem. Thankfully, there is a solution and this is when one function can make a call to another

Extra: input() function

Python provides an input() function to retrieve user input from the console. The function allows us to specify a message to the user asking for an input and then returns the input as a string (this is true for numerical inputs too)

```
language = input("Please enter a language that you can speak: ")
print(f"You can speak: {language}")
```

The following will display in the console

```
Please enter a language that you can speak: 
```

If we then type a language (English as an example) and then press 'Enter':

```
Please enter a language that you can speak: English
You can speak: English
```

New Resident Application

We will create a simple application to model a system (without a database component) to add new residents for a fictional housing complex.

In this first demonstration (one of many possible implementations), we will **not** make use of internal, user-defined function calls which are functions we create. This will demonstrate the effect that repeated lines of logic can have on even a small, simple program

Comments Note: Many of the comments in the code below are not recommended in enterprise code. Comments should explain **why** a section of code is performing a task that the code **may not** clearly explain on its own. The unrecommended comments are included only for beginners to fully understand this example.

```
def new_resident():
    # resident attributes
    name = ''
    house_number = None
    vehicle = False
    vehicle_count = 0

    # menu option variable
    menu_option = ''

    # Application Welcome Message
    print("Welcome to the New Resident Database")

    # run application until user enters 'quit'
    while menu_option != 'quit':

        # print Main Menu
        print("\nPlease enter an option number below or 'exit' to quit")
        print('1 - Add New Resident')
        print('2 - Add New Resident and Vehicle')
        print('quit - Exit Application')

        # ask the user for an option, .strip() removes
        # excess whitespace from the front and end of the input string
        menu_option = input("Your Option: ").strip()

        # check if user wants to exit application
        # .lower() removes case sensitivity from user input
        if menu_option.lower() != 'quit':

            # -----
            # only add new resident
            if menu_option == "1":
                print("\nAdd Resident Only")

                name = input("Please Enter Resident Name: ")
                # check an input was received:
                if name.strip() == "":
                    print("A name was not recieved")
                    # move back to start of loop
                    continue

                try:
                    house_number = int(input("Please enter the house number: "))
```

1

```

# a house number that is not a valid integer was received
except ValueError:
    print("House Number provided is not valid")
    # move back to the start of the loop
    continue

print(f"New Resident has been saved as:")
print(f"Name: {name}, House Number: {house_number}")

# -----
# add a new resident with a vehicle(s)
elif menu_option == "2":
    print("\nAdd Resident and Vehicle")

    name = input("Please Enter Resident Name: ")
    # check an input was received:
    if name.strip() == "":
        print("A name was not recieved")
        # move back to start of loop
        continue

    try:
        house_number = int(input("Please enter the house number: "))

        # a house number that is not a valid integer was received
        except ValueError:
            print("House Number provided is not valid")
            # move back to the start of the loop
            continue

    # option selection means resident has a vehicle
    vehicle = True

    # add number of vehicles
    try:
        vehicle_count = int(input("Please enter the number of vehicles: "))
    except ValueError:
        print("An Invalid Number of Vehicles has been entered")
        continue

    print(f"\nNew Resident has been saved as:")
    print(f"Name: {name}, House Number: {house_number}")
    print(f'with {vehicle_count} vehicles')

new_resident()

```

The sections shown in red boxes indicate repeated code which is bad programming practice even in a simple program. Consider wanting to add another attribute to the resident such as 'number_of_people' indicating the number of people living permanently in the residence.

To do this, we would have to make identical changes to both of these sections. Making a change is perfectly normal in programming as programs need to be agile and ready to adapt to change, **however** we must design our program to make the change in one place

The red sections of code can be corrected if we create a separate function and then call it within the 'new_resident()' function

```
def create_resident():
    name = input("Please Enter Resident Name: ")
    # check an input was received:
    if name.strip() == "":
        print("A name was not recieved")
        return "Invalid"

    try:
        house_number = int(input("Please enter the house number: "))
    # a house number that is not a valid integer was received
    except ValueError:
        print("House Number provided is not valid")
        return "Invalid"

    return f"Name: {name}, House Number: {house_number}"

def new_resident():
    # resident attributes
    name = ''
    house_number = None
    vehicle = False
    vehicle_count = 0

    # menu option variable
    menu_option = ''

    # Application Welcome Message
    print("Welcome to the New Resident Database")

    # run aplication until user enters 'quit'
    while menu_option != 'quit':

        # print Main Menu
        print("\nPlease enter an option number below or 'exit' to quit")
        print('1 - Add New Resident')
        print('2 - Add New Resident and Vehicle')
        print('quit - Exit Application')

        # ask the user for an option, .strip() removes
        # excess whitespace from the front and end of the input string
        menu_option = input("Your Option: ").strip()

        # check if user wants to exit application
        # .lower() removes case sensitivity from user input
        if menu_option.lower() != 'quit':

            # -----
            # only add new resident
            if menu_option == "1":
                print("\nAdd Resident Only")

                # make a function call to create the new resident
                resident = create_resident()
                if resident != "Invalid":
                    print(f"New Resident has been saved as:")
                    print(resident)

            # -----
            # add a new resident with a vehicle(s)
            elif menu_option == "2":
                print("\nAdd Resident and Vehicle")

                # make a function call to create the new resident
                resident = create_resident()
```

```

    if resident != "Invalid":
        # option selection means resident has a vehicle
        vehicle = True

        # add number of vehicles
        try:
            vehicle_count = int(input("Please enter the number of vehicles: "))
        except ValueError:
            print("An Invalid Number of Vehicles has been entered")
            continue

        print(f"\nNew Resident has been saved as:")
        print(resident)
        print(f'with {vehicle_count} vehicles')

new_resident()

```

With the improved program, if we wanted to add more attributes to a new resident, make modifications to verification of the input or make any other changes then we can do so in **one place** which is within the `create_resident()` method.

Higher Order Functions

A function can do more than call other functions within itself - we can also **pass a function as an argument to another function** (like we do with values). This type of function is known as a **Higher Order Function**

We will create a new application that uses a Higher Order Function to display different messages as seen below with analysis to follow

```

def greeting():
    return "Hello"

def full_greeting():
    return "Hello and how are you?"

def farewell():
    return "Goodbye"

def print_message(message):
    print(message)

# print a simple greeting:
print_message(greeting())
# print a full greeting:
print_message(full_greeting())
# print a farewell:
print_message(farewell())

```

Higher Order Function

message is a parameter that we want to use to accept a function being passed in

Hello
Hello and how are you?
Goodbye

1. We created three initial functions that each return a different message as a string.
2. The function **print_message()** is a higher order function. It has specified what appears to be a normal parameter, but we intend on using it to hold the return value from a function call
3. Notice that when we call **print_message()** we are passing in functions (underlined in green). **Notice** that we can use the same function (print_message()) and pass in different functions. This is the fantastic thing about Higher Order Functions!

We will continue to explore Higher Order Functions with a second example

In Mathematics we have patterns that we can work with and here we will focus on two of them:

- Arithmetic Pattern - constant difference
- Geometric Pattern - constant product (or ratio)

A simple example of an **Arithmetic Pattern** would be: 2, 4, 6, 8, 10

Here we are continuously adding the number 2 to a value to produce the next value

A simple example of a **Geometric Pattern** would be 2, 4, 8, 16, 32

Here we are continuously multiplying a value by 2 to get the next value

Create a Pattern Generating Program

Goals:

1. Have two separate functions that can generate an Arithmetic and Geometric Pattern respectively. These functions should take in the number of terms, the starting term value and the constant difference/ratio
2. Have a Higher Order Controlling Function that can request a pattern to be generated of a specific type. The pattern should have each value stored in a list
3. Print the values for the pattern

```
def arithmetic_pattern(start_value, difference, term_count):
    # list to store the generated arithmetic pattern
    generated_pattern = []

    for i in range(0, term_count):
        # create the new term value
        new_value = start_value + difference * i
        # add new value to the pattern list
        generated_pattern.append(str(new_value))

    return generated_pattern
```

Geometric Pattern Function

The Geometric Pattern uses the following formula: $a \times r^{n-1}$ where:

a = first term (start_value)

r = constant ratio

n = current term value (i)

```
def geometric_pattern(start_value, ratio, term_count):
    # list to store the generated geometric pattern
    generated_pattern = []

    for i in range(1, term_count + 1):
        # create the new term value
        new_value = start_value * ratio**(i - 1)    # ** means to the power of
        # add new value to the pattern list
        generated_pattern.append(str(new_value))

    return generated_pattern
```

Now we want to create a higher order function that relies on its call to determine which pattern generating function (the two above) it must use. This is achieved by passing the desired function in as an argument

To add some functionality we can pretend this higher order function acts as an assistant to a Math teacher in which it can generate a few standard questions for students to practice

Solving some of these questions requires a bit of Algebra which is not explored here

In some of the questions we want to randomly choose values from the patterns and the 'random' module allows us to generate random numbers to do this. This import statement must be at the top of the entire program

```
import random

def pattern_question_generator(pattern_creator, start_value, difference_ratio, term_count):

    # call function to create a new pattern
    pattern = pattern_creator(start_value, difference_ratio, term_count)

    # print the pattern. .join() is a string method to join items of a list into a string
    print("\nConsider the following pattern: " + " ; ".join(pattern) + " ; ... \n")

    # Generate Questions
    print("1. State the type of pattern shown and prove the type")

    # use the randint() function from the random module to generate a random term number
    # within the range of the pattern
    print("2. Determine the term number for the value: " + pattern[random.randint(1, term_count - 1)])

    # use the randint() function from the random module to generate a random term number
    # greater than the number of terms in the pattern
    print(f"3. Determine the value of term number: {random.randint(term_count, term_count + 20)}")
```

As the `pattern_question_generator()` is a higher order function, we can pass in whichever pattern creation function we want to, as long as the program has been designed well that either of the two we created can be used

Now we can call the function

```
# Call the Higher Order Function with an Arithmetic Pattern
pattern_question_generator(arithmetic_pattern, 2, 2, 15)

# Call the Higher Order Function with a Geometric Pattern
pattern_question_generator(geometric_pattern, 5, 2, 4)
```

With output:

```
Consider the following pattern: 2 ; 4 ; 6 ; 8 ; 10 ; 12 ; 14 ; 16 ; 18 ; 20 ; 22 ; 24 ; 26 ; 28 ; 30 ; ...

1. State the type of pattern shown and prove the type
2. Determine the term number for the value: 18
3. Determine the value of term number: 32

Consider the following pattern: 5 ; 10 ; 20 ; 40 ; ...

1. State the type of pattern shown and prove the type
2. Determine the term number for the value: 20
3. Determine the value of term number: 5
```

Why is this useful?

There are more patterns than the two shown here. If we wanted to create another we could do so separately and the `pattern_question_generator()` function could still work with it. Some patterns have different attributes than the Arithmetic and Geometric pattern shown, but if we use default values or design the program to work with more pattern types, then we can have one `pattern_question_generator()` that can call and work with a variety of functions

The type of program that we have created is part of a larger idea in Programming and Higher Order Functions (as well as classes) help us implement this crucial principle:

Dependency

Dependency is a measure of how strongly one component depends on another. Strong dependency means that one component is **tightly-coupled** to another and this is not a good thing. We want to design applications that are **loosely-coupled**. Dependency is strongly considered with classes as well, but is not discussed in these notes.

In the previous section, we have loosely-coupled the `pattern_question_generator()` function to functions that can create different types of patterns. **Higher Order Functions** has allowed us to pass in a different pattern generator component, making the `pattern_question_generator()` less dependent on one particular type of pattern generator.

This idea of reduced dependency exists all around us.

Consider a simple light-bulb socket. If this socket required a very specific brand and type of bulb then it would be problematic. Manufactures of lightbulbs have instead made their bulbs conform to a general connection type (screw-in or clip-in for example). For this reason, one can buy multiple brands of light bulbs.



Consider further that older light bulbs used much more electricity than modern LEDs do. The modern lightbulbs use the same connectors as the older, less-efficient types and we therefore did not have to remove and change light-bulb sockets to simply become more efficient with our energy usage.

We want programs to behave in a similar way, the more a program can work with different components (assuming they conform to a standard the program is designed to use), the more adaptable the program is to change and the greater its range of usage.

Example

To demonstrate the power that loosely-coupled components have over tightly-coupled components, we will use an example.

Validation of input data is needed in programs. We want to check various things such as the datatype, the allowed range of a value, the number of values, if a value was received in the first place and many other aspects.

Consider a program that will sum the values it receives in a list and return the sum. This program will validate the data received and then perform the sum action. We want this program to sum the the values in a list that contains the quantity for items sold in a store which should be 0 or positive integer values.

1. To ensure the data received is correct, we will create a validation function that checks for non-decimal values, empty values and values containing invalid characters such as letters

```
def validate_item_nums(data_list):  
    # check values have been received  
    if len(data_list) == 0:  
        return False  
  
    for item in data_list:  
        # check if item is an integer  
        if isinstance(item, int):  
            # item count can only be 0 or greater than 0  
            if item < 0:  
                return False  
        else:  
            return False  
  
    # checks passed, return true  
    return True
```

2. Now we create a summation function that uses the validation function above and then sums the values in the list (if the validation returned True)

```
def perform_sum(data_list):  
    # perform validation on the data list  
    if validate_item_nums(data_list):  
        sum = 0  
        # move through list and add value to sum  
        for i in data_list:  
            sum += i  
        return sum  
    else:  
        return "Invalid value(s) in data list"
```


3. Now we perform a few simple tests to verify that the program does work

```
print(perform_sum([1, 10, 15, 20]))      # valid
print(perform_sum(["2", 10, 15, 20]))    # contains a string
print(perform_sum([]))                   # contains no data
print(perform_sum([1.1, 2.2, 3.3, 4]))   # contains decimals
```

With output:

```
46
Invalid value(s) in data list
Invalid value(s) in data list
Invalid value(s) in data list
```

Program Successful!

The program has successfully accomplished its task, however if we look more closely we will see that we have **tightly-coupled** the `perform_sum()` function to one, very specific `validation()` function.

Consider the same store wanting to sum the prices of items within the store. The `perform_sum()` function can do this, but product prices can contain decimals and our validation function does not allow decimals.

The most logical thing (seemingly) to do is to create a new validation function and then modify the `perform_sum()` function to call this new validation function instead. Now we pretend that someone in the store wants us to once again sum a list of items which needs the old validation function again...

Now we could say that we then just create a new summation function to handle decimals instead using its specific decimal validation function. The thing is, our current summation function can handle summing decimals. Creating a new function increases the code base and is not the best way to deal with this issue. It may not seem to be an issue in a small program, but with much larger programs we want to deal with these kinds of issues in a better way.

Constantly modifying the source code is not the answer! This means that making one function overly dependent on another, specific implementation of some solution type is not a good thing. Instead, we want our functions to be capable of working with multiple ‘types’ of another functional solution to some task (extending to classes too).

1. We make no changes to the `validate_item_nums()` function. We first create a new validation function to validate a list of decimal values.

```
def validate_decimal_list(data_list):
    # check values have been received
    if len(data_list) == 0:
        return False

    for item in data_list:
        # check if item is an integer or float
        if isinstance(item, int) or isinstance(item, float):
            # item value can only be 0 or greater than 0
            if item < 0:
                return False
        else:
            return False

    # checks passed, return true
    return True
```

2. With two validation functions, we can now modify the summation function to accept a function as an argument that states the validation function it should use

```
def perform_sum(data_list, validator):
    # perform validation on the data list
    if validator(data_list):
        sum = 0
        # move through list and add value to sum
        for i in data_list:
            sum += i
        return sum
    else:
        return "Invalid value(s) in data list"
```

3. Perform Tests

```
# check with integer values (item numbers)
print(perform_sum([5, 10, 15, 20], validate_item_nums))
# check with decimal values (item prices)
print(perform_sum([3.75, 157.80, 200], validate_decimal_list))
```

With output:

```
50
361.55
```

In the function calls notice that we can now specify the validation function we want to use. Our `perform_sum()` function is no longer dependent on one specific validation function and can accept others specific to the task it needs to accomplish.

Recursive Functions

Functions can be called within a program, functions can call other functions and we have seen that functions can even accept another function as an argument (Higher Order Functions).

Another useful feature of functions is that they can even call themselves. These types of functions are known as **Recursive Functions**.

Consider an example in which we want to print “Hello” to the console 5 times. We can use a simple for loop to do this:

```
def print_message(message, count):  
    for i in range(0, count):  
        print(message)  
  
print_message("Hello", 5)
```



```
Hello  
Hello  
Hello  
Hello  
Hello
```

Or we could use a recursive function:

```
def print_recursive_message(message, count):  
    # base case  
    if count == 0:  
        return  
    else:  
        print(message)  
        # RECURSIVE CALL  
        print_recursive_message(message, count - 1)  
  
print_recursive_message("Hello", 5)
```



```
Hello  
Hello  
Hello  
Hello  
Hello
```

The green section is the **Base Case** and the red section shows the function calling itself

If you have worked with a 'while loop' before, then you may have accidentally created an infinite loop. This is a loop whose termination condition is never met and the loop runs infinitely.

With recursive functions, they call themselves to perform some task, however they will keep doing so unless we provide some means to stop the calls. This is the **Base Case**. The base case differs for each function but is essentially some condition that is met which is usually the point at which the task has been broken down to its end value or simplest component and we want the calls to stop.


Recursive Functions do not work in the same way that a loop does. At the very beginning of the note set, we briefly considered how a program's line of execution works (page 3). We considered how a program calls a function and moves into a new line of execution with the previous line waiting for the results of this new line before it may proceed.

How do Recursive Functions work?

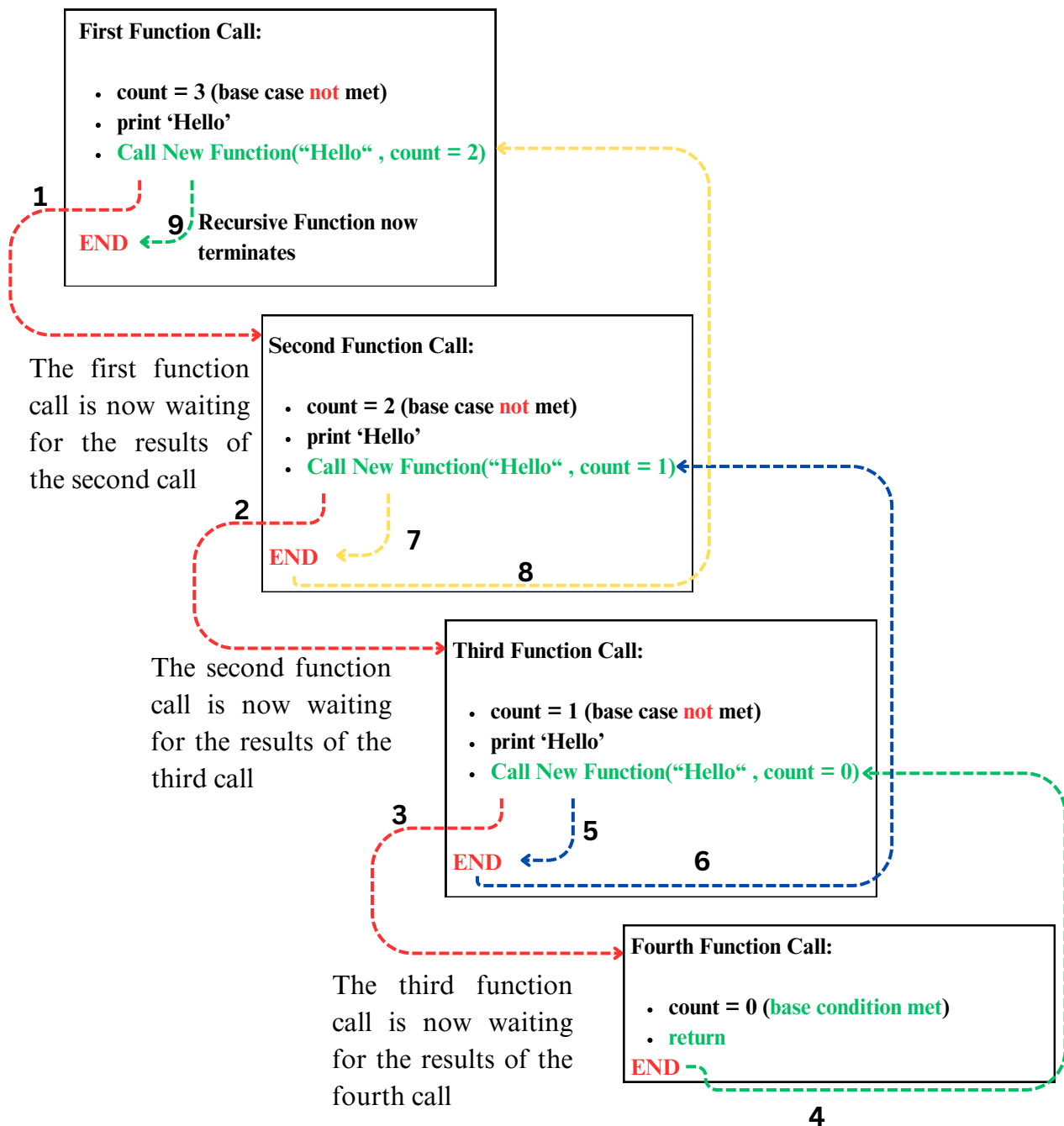
A recursive function will continuously call itself until a base condition has been met, but when the condition is met the function call (as a whole) is not yet completed. We will consider the previous recursive function and use a visualization to help:

In this example, we will print 'Hello' three times to simplify the visualization

```
def print_recursive_message(message, count):  
    # base case  
    if count == 0:  
        return  
    else:  
        print(message)  
        # RECURSIVE CALL  
        print_recursive_message(message, count - 1)  
  
print_recursive_message("Hello", 3)
```



```
Hello  
Hello  
Hello
```



In the diagram above:

- The first function call is made, but the count is not zero. This function prints 'Hello' and then calls itself again. This time with the count decreased by 1. The important thing to note is that the function has not terminated. Its execution is 'paused', waiting for a return of the function it has just called
- The second function call does the same as the first, but with a count now equal to 2. The base condition (count == 0) is not met so it prints 'Hello' and then calls itself again
- The third function, with a count now equal to 1 also prints 'Hello' and then calls itself again. Remember that it decreases the count value by 1

- **Finally**, in the fourth function call the value of count is now equal to 0 and the base condition has been met

Remember that there are now **four** 'open' function calls. None of the previous functions have terminated and we now have to move backwards; up through every prior function call until we get back to the first one.

The fourth function call terminates then execution goes back to third call. This function terminates and execution goes back to the second call. This function terminates and execution goes back to the first function. The first function call now terminates and the program can now terminate as well.

Factorial Recursive Function

Recursive functions and the repeated self-calls that they can perform can be very useful. An extremely common example is to create a recursive function that can determine the factorial of a number

Consider the following expression:

$$5 \times 4 \times 3 \times 2 \times 1$$

This type of expression can be very helpful and one instance in which it is used is in Probability, in which it can be created with the Fundamental Counting Principle. For our purposes here, we just need to focus on a shorter way to write this expression

With the use of a factorial (using !), we can say:

Factorial \rightarrow $5! = 5 \times 4 \times 3 \times 2 \times 1$
 $5! = 120$

For comparison:

$$3! = 3 \times 2 \times 1$$


$$3! = 6$$

A factorial demonstrates that we **repeatedly** decrease the next number by 1 and keep doing this until we reach 1. Once completed we can then multiply the numbers.

This is a perfect task to accomplish with a Recursive Function. In the code below, notice how the 'pause' in execution allows us to keep dropping the value of the current number by 1 in each recursive call

Once the base condition that the current number is equal to 1 has been met, we then return back up through the function calls. When we do so we also multiply the numbers, returning each product as we do so to its function call.

```
def factorial_solution(number):  
    if number == 1:  
        return number  
    else:  
        return number * factorial_solution(number - 1)  
  
print(factorial_solution(3))  
print(factorial_solution(5))
```



6
120

Recursive Functions and Memory

A function call is added to the **stack memory**. This means that each time we call a function, we add a new execution block that must be kept in memory until it terminates.

When we move through the layers of function calls in recursive functions, we add each and every call on to the stack. The stack has a limit and we cannot keep doing this indefinitely. Python has protection against this and will allow up to 1000 recursive function calls as its default limit.