

Table of Contents

Introduction	2
Videos	3
1 User Defined Functions	4
1.1 Arguments and Parameters	7
1.2 The Return Statement	9
1.3 Namespaces and the main Function	12
1.4 Advantages of Using User Defined Functions	15
1.4.1 Structure, Organization, and Separation of Tasks	16
1.4.2 Reducing Non-Adjacent Duplicate Statement Groups	17
1.4.3 Solution Generalization	19

Introduction

This document introduces the purpose of user defined functions, how to create them and when to use them. Links to videos are provided for more examples and explanations on user defined functions.

Videos

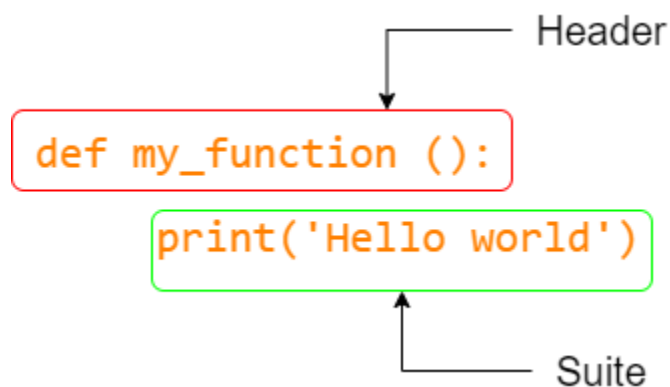
- [Introduction to user defined functions](#) (0:00 - 6:31)
 - Covers the benefits and purpose of using user defined functions, how to create user defined functions, and shows an example of it in use
- [Evaluation of programs with user defined functions](#) (0:00 - 3:12)
 - Covers how a program evaluates step by step with user defined functions, and how to use the step into/over function in the wingIDE debugger
- [Function argument and parameters](#) (0:00 - 8:48)
 - Covers what are arguments and parameters, and how they work. It also covers how to define user defined functions with parameters, and how to call the function with arguments
- [Namespaces](#) (0:00 - 6:03)
 - Covers what are namespaces, and how they affect the scope of identifiers
- [The *main* function of a program](#) (0:00 - 6:59)
 - Covers how to use a main function, local vs global namespaces, and the difference between local and global variables
- [The return statement](#) (0:00 - 6:19)
 - Covers what a function returns, what a return statement is, and how to use it to return values from user defined functions
- [Using functions for problem decomposition](#) (0:00 - 11:19)
 - Covers how a problem can be solved by decomposing it into smaller tasks, and how user defined functions can be used to solve the subproblems through the use of an example
- [Functions and immutable objects](#) (0:00 - 10:17)
 - Covers how immutable objects work with functions in regards to their namespaces, and why and how to return immutable objects through the use of an example

- [Functions and mutable objects part 1](#) (0:00 - 4:21)
 - Covers function side effects, how mutable objects like lists are modified, and why the return statement is not needed at times
- [Functions and mutable objects part 2](#) (0:00 - 4:06)
 - Covers why and when a return statement is needed to reflect the changes applied to mutable objects like lists

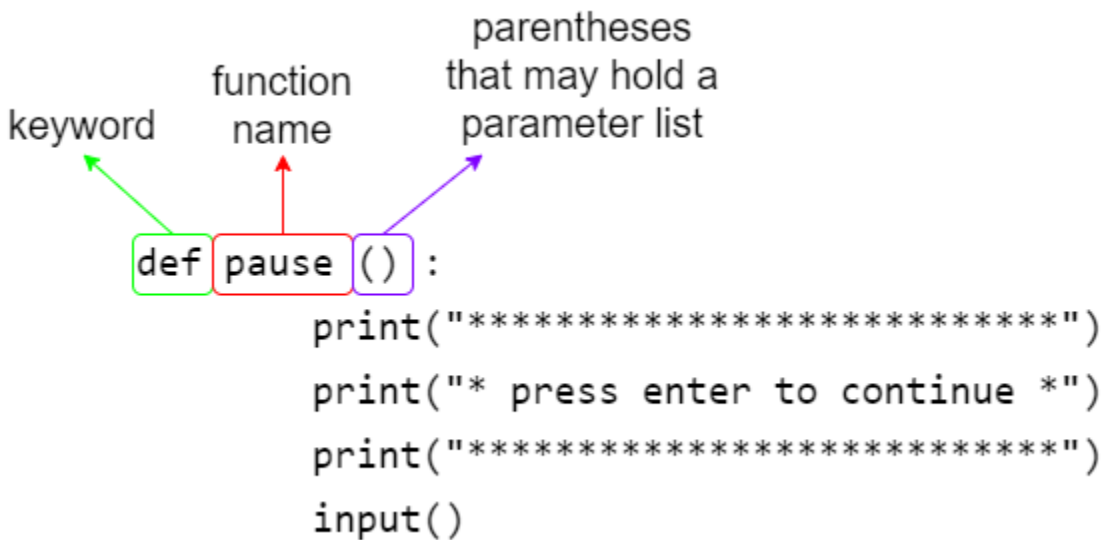
1 User Defined Functions

While there are many built-in or imported functions available in Python, it is important to know how to define our own functions. These functions that are defined by you are called user defined functions.

User defined functions are compound statements, meaning that they span multiple lines and include other statements, such as simple statements or other compound statements. Remember that compound statements are made up of 2 parts: a header, which is the first line, and a suite, which is indented one level from the header.



Let's take a closer look at the header of a user defined function:

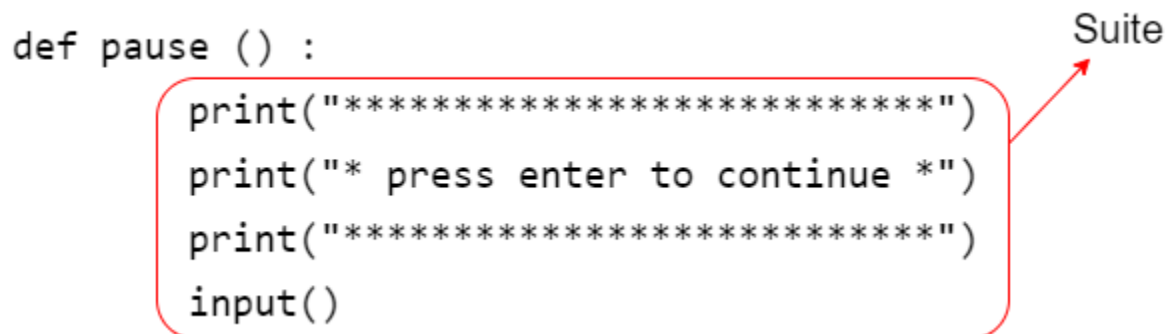


The diagram shows the function header `def pause () :` with three labels and arrows pointing to its components:
- **keyword**: points to `def` (green arrow)
- **function name**: points to `pause` (red arrow)
- **parentheses that may hold a parameter list**: points to `()` (purple arrow)
The full function definition is shown below the header:
`print("*****")
print("* press enter to continue *")
print("*****")
input()`

We start with the keyword `def`. This tells the program that we will be defining a function.

Next, is the function name. This is the identifier that will be used to call the function later on. In this case, our user defined function will be called `pause`.

Lastly, it is followed by two parentheses `()` and a colon `:`. These parentheses hold a parameter list, which in this case is empty. Parameters will be explained in more detail later on.



The diagram shows the function header `def pause () :` followed by a red rounded rectangle containing the function's suite:
`print("*****")
print("* press enter to continue *")
print("*****")
input()`
A red arrow points from the label **Suite** to the red rounded rectangle.

The suite of a user defined function is similar to the suite of a compound statement. It is indented one level from the header, and can be composed of simple or compound statements.

Calling a user defined function follows the same format as calling any built-in or imported functions in Python.

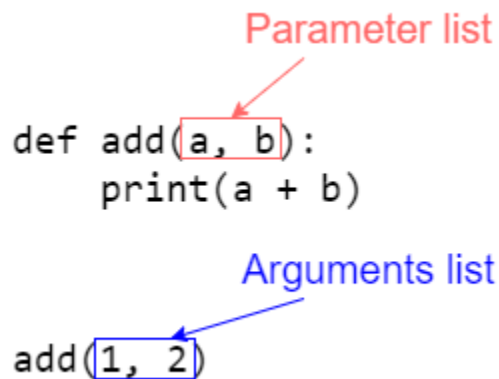
```
def pause():  
    print("*****")  
    print("* press enter to continue *")  
    print("*****")  
    input()  
  
name = input("What is your name?")  
print("Hello " + name + "!")  
pause()  
print("Have a nice day!")
```

Here we have a user defined function called `pause`. Using its function name from the header of its function definition, the user defined function can be called. Since the function definition of `pause` does not contain any parameters, no arguments are needed when the function is called.

1.1 Arguments and Parameters

Arguments and parameters are important parts of functions. Arguments are used to “send” the information to functions, so that functions can do their work properly.

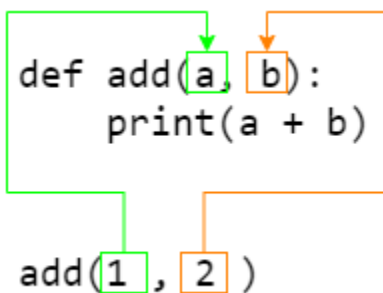
Parameters are found in function definitions, which are used to tell the program whether or not a function call needs arguments. They are used to generalize functions by passing values to them.



The diagram illustrates the relationship between a function definition and a function call. The function definition is `def add(a, b):` followed by an indented `print(a + b)`. A red arrow points from the text "Parameter list" to the parameters `a, b` in the definition. The function call is `add(1, 2)`. A blue arrow points from the text "Arguments list" to the arguments `1, 2` in the call.

```
def add(a, b):  
    print(a + b)  
  
add(1, 2)
```

At the time of a function call, each parameter in the parameter list is bound to the object that the corresponding argument in the argument list is bound to. In the following example at the time of the function call `add(1, 2)`, the parameter `a` is bound to the literal object `1` and parameter `b` is bound to the literal object `2`.



The diagram shows the binding of variables during a function call. In the function definition `def add(a, b):`, the parameter `a` is enclosed in a green box and the parameter `b` is enclosed in an orange box. In the function call `add(1, 2)`, the argument `1` is enclosed in a green box and the argument `2` is enclosed in an orange box. A green line connects the green box around `a` to the green box around `1`. An orange line connects the orange box around `b` to the orange box around `2`.

```
def add(a, b):  
    print(a + b)  
  
add(1, 2)
```

In this case, when the suite of the user defined function definition is evaluated, the expression in the print statement `a + b` will evaluate to `3`, which will then be displayed to the user in the output.

Not all user defined functions will need arguments and parameters; however, sometimes we can further simplify our code by using arguments and parameters:

```
def pause(symbol):  
    border = symbol * 27  
    message = " press enter to continue "  
    print(border)  
    print(symbol + message + symbol)  
    print(border)  
    input()  
  
star = '*'  
name = input("What is your name?")  
print("Hello " + name + "!")  
pause(star)  
print("Have a nice day!")
```

Here we've included a parameter called `symbol` in the user defined function `pause`. The string value bound to the parameter `symbol` will be used to create the message's border. Having this as a parameter will now require us to pass in an argument when the user defined function is called, but it will also allow us the flexibility to change what symbol we want to use for the border by simply passing in a different symbol as the argument.

1.2 The Return Statement

Some functions do not return any value. For example the `print` function is used to display something in the output, so it would not make sense for it to return something. We can check if a function returns something by assigning the function call to an identifier, and printing the value of that identifier.

<pre>>>> a = print('hello') hello >>> print(a) None</pre>	Here, we can see that the <code>print</code> function does not return any value as the identifier <code>a</code> holds a <code>NoneType</code> object.
---	--

However, sometimes we might want to call a function to obtain a value from it. For example when we want to obtain the length of a given string, calling the built-in `len` function will return an `int` object. The returned value can then be stored in a variable for future use.

<pre>>>> a = len('hello') >>> print(a) 5</pre>	Here, we can see that the <code>len</code> function returns a value as the identifier <code>a</code> holds an <code>int</code> object.
--	--

Sometimes we may want to create user defined functions that return us certain values. The return statement allows us to do this. In the following example, a user defined function called `add` is created. The `add` function will take in two arguments, add them together, and return the result.

<pre>>>> def add(a, b): ... c = a + b ... >>> result = add(1, 2) >>> print(result) None</pre>	<p>We can see here that the identifier <code>result</code> is bound to a <code>NoneType</code> object. This means that the user defined function <code>add</code> does not return a value.</p>
--	--

To make it such that the user defined function will return a value, we can use the return statement.

<pre>>>> def add(a, b): ... c = a + b ... return c ... >>> result = add(1, 2) >>> print(result) 3</pre>	<p>Here, we can see that printing the identifier <code>result</code> displays an a 3, meaning that the user defined function <code>add</code> does return a value now.</p>
--	--

Note that once a return statement in a user defined function is evaluated, the rest of the user defined function's suite will not be evaluated. The program will return to the line the user defined function was called, and continue evaluating the rest of the code.

<pre>>>> def add(a, b): ... c = a + b ... return c ... print("result = " + str(c)) ... >>> result = add(5, 2) >>> print(result) 7</pre>	<p>The print statement in the user defined function won't be evaluated as once the return statement is evaluated, the program returns to where the function was called.</p>
--	---

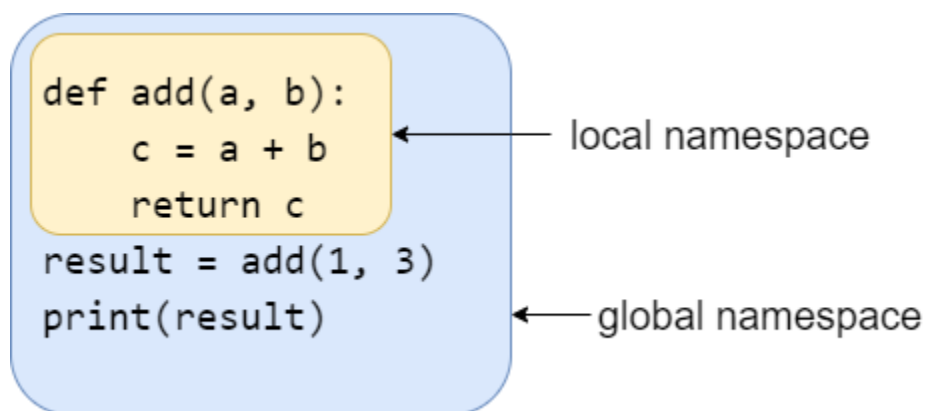
This can be helpful in some situations where you want to create a user defined function that returns a different result when specific conditions are met.

<pre>>>> def same_len(str1, str2): ... if len(str1) == len(str2): ... print("same") ... return True ... print("different") ... return False ... >>> a_str = "Hello" >>> b_str = "Panda" >>> result = same_len(a_str, b_str) >>> print(result) True</pre>	<p>The user defined function <code>same_len</code> checks if the two given strings are of the same length. It returns True if they are the same length, and False otherwise.</p> <p>If the two strings have the same length, the return statement inside the suite of the <code>if</code> statement will be evaluated, and the remaining two lines after the <code>if</code> statement will not be evaluated.</p>
---	---

1.3 Namespaces and the main Function

Namespaces are an important concept in coding. A namespace is a collection of identifiers and the object that the identifier refers to. They affect the scope of identifiers, or in other words, where certain identifiers are available and can be accessed within a program.

In general, there are two types of namespaces: global and local. A local namespace is relative to its global namespace. For example, the namespace of a user defined function or the main function is a local namespace relative to the namespace of the program.



Think of a function namespace like a discussion forum, and identifiers as the usernames of students that use that forum. Let's say that there is a CMPUT 174 discussion forum and a chemistry class discussion forum. There may be a student with the name 'John' in the CMPUT 174 forum, and another student with the same name in the chemistry class forum. Even though the same name exists in both forums, they do not refer to the same person. If someone is calling for John in the CMPUT 174 forum, they would not be referring to John from the chemistry class forum.

Similarly in Python, the global namespace and the namespaces of different functions may contain identifiers with the same name, but those identifiers may be bound to different values.

```

>>> def pause():
...     symbol = '~'
...     border = symbol * 27
...     message = " press enter to continue "
...     print(border)
...     print(symbol + message + symbol)
...     print(border)
...     input()
...
>>> symbol = '*'
>>> print(symbol)
*
>>> pause()
~~~~~
~ press enter to continue ~
~~~~~
>>> print(symbol)
*

```

In this simple program, we can see that there are two identifiers with the same name `symbol`: one in suite of the user defined function definition and one outside the function definition. While they both have the same name, they are bound to different values. When this program is run, you'll notice that the border is printed using the `~` symbol rather than `*`.

Some identifiers and the value they are bound to are only accessible in certain namespaces, and they are also known as 'local variables'. They are considered 'local' as they are only accessible within a certain function or section of the code. In contrast, 'global variables' are identifiers that are accessible throughout the entire code.

```
>>> symbol = '-'
>>> def pause():
...     border = symbol * 27
...     message = " press enter to continue "
...     print(border)
...     print(symbol + message + symbol)
...     print(border)
...     input()
...
>>> pause()
-----
- press enter to continue -
-----

''
>>> print(border)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'border' is not defined
```

The identifier `symbol` that is bound to the string `'-'` is a global variable. Even though we did not define an identifier called `symbol` in the suite of the user defined function definition, the program was able to access the value from the global variable to print the message.

However, the identifier `border` that is defined within the suite of the user defined function definition is a local variable, and can only be accessed while in the scope of the user defined function definition. Attempting to access it outside of its namespace will cause an error since it only exists inside that namespace.

While there are instances where global variables are great to have, it is usually best to use local variables. The reason being that if both a global and local variable share the same name, then how will the program decide which variable to use?

To avoid using global variables, we'll be using a special user defined function called the *main* function. The main function is special in that if a main function has been defined, then that function will be the point where the program begins. Otherwise, a program will begin at the top of the code.

```
>>> def main():  
...     print('Hello world!')  
...     print('This is the main function')  
...  
>>> main()  
Hello world!  
This is the main function
```

Defining and calling the main function is the same as defining and calling a user defined function.

1.4 Functions and Immutable Objects

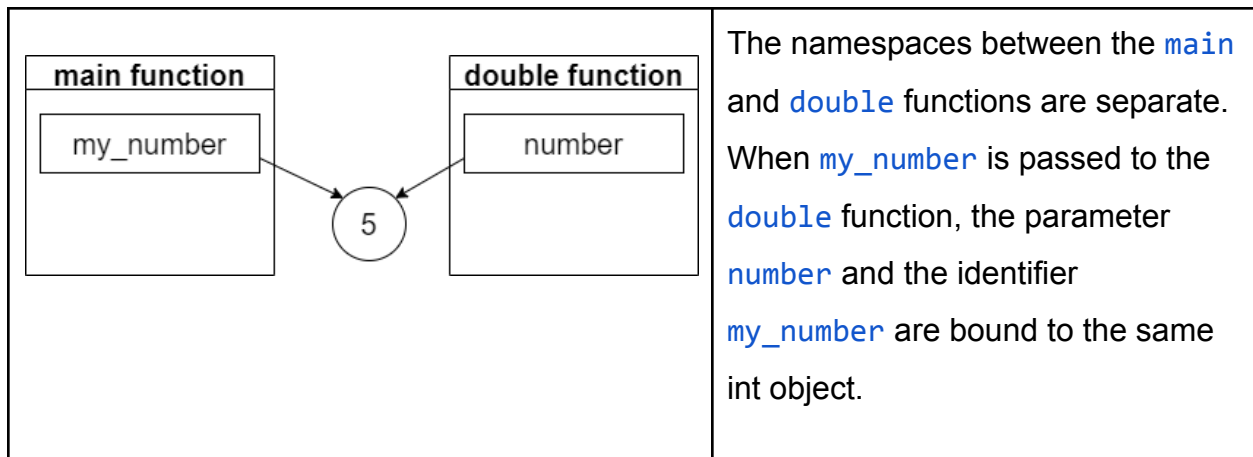
Remember that some object types are immutable, meaning that a new object must be created to reflect any changes. How does this affect how functions work with immutable objects? To obtain the result of what was changed in the function, a return statement is needed.

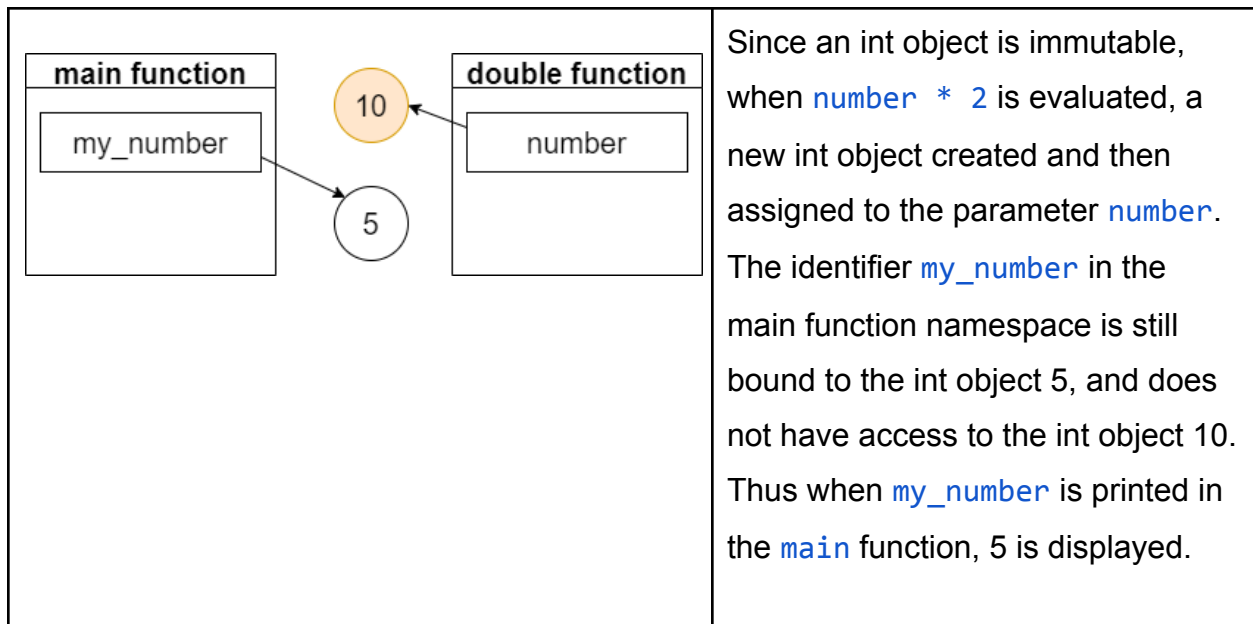
In the following example, we'll show what happens when the int object bound to `number` is changed, and why a return statement is needed for immutable objects.

```
def double(number):  
    number = number * 2  
def main():  
    my_number = 5  
    double(my_number)  
    print(my_number)  
main()
```

Here is a simple program. `double` is a user defined function that multiplies a given number. When `my_number` is printed, it will be displayed as 5, and not 10.

Let's take a look at what's happening using diagrams:





For the identifier `my_number` to be bound to the int object 10, a return statement needs to be added. An assignment statement also needs to be added, so that it can 'catch' what is being returned.

<pre>def double(number): number = number * 2 return number def main(): my_number = 5 my_number = double(my_number) print(my_number) main()</pre>	<p>The return statement is added in the user defined function <code>double</code>. An assignment statement is also added, so the identifier <code>my_number</code> can 'catch' the returned object from the function call. This time when <code>my_number</code> is printed, 10 will be displayed.</p>
--	--

1.5 Functions and Mutable Objects

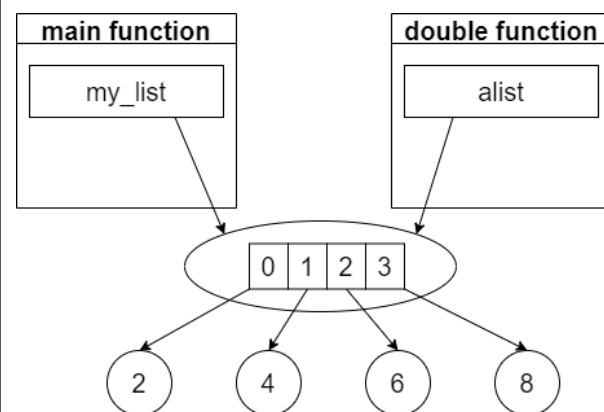
Mutable objects allow us to make changes on the current object without needing to create a new object. Depending on how the object is changed, a return statement may or may not be needed. We'll be using a list object to show this.

First, we'll take a look at an example that does not need a return statement:

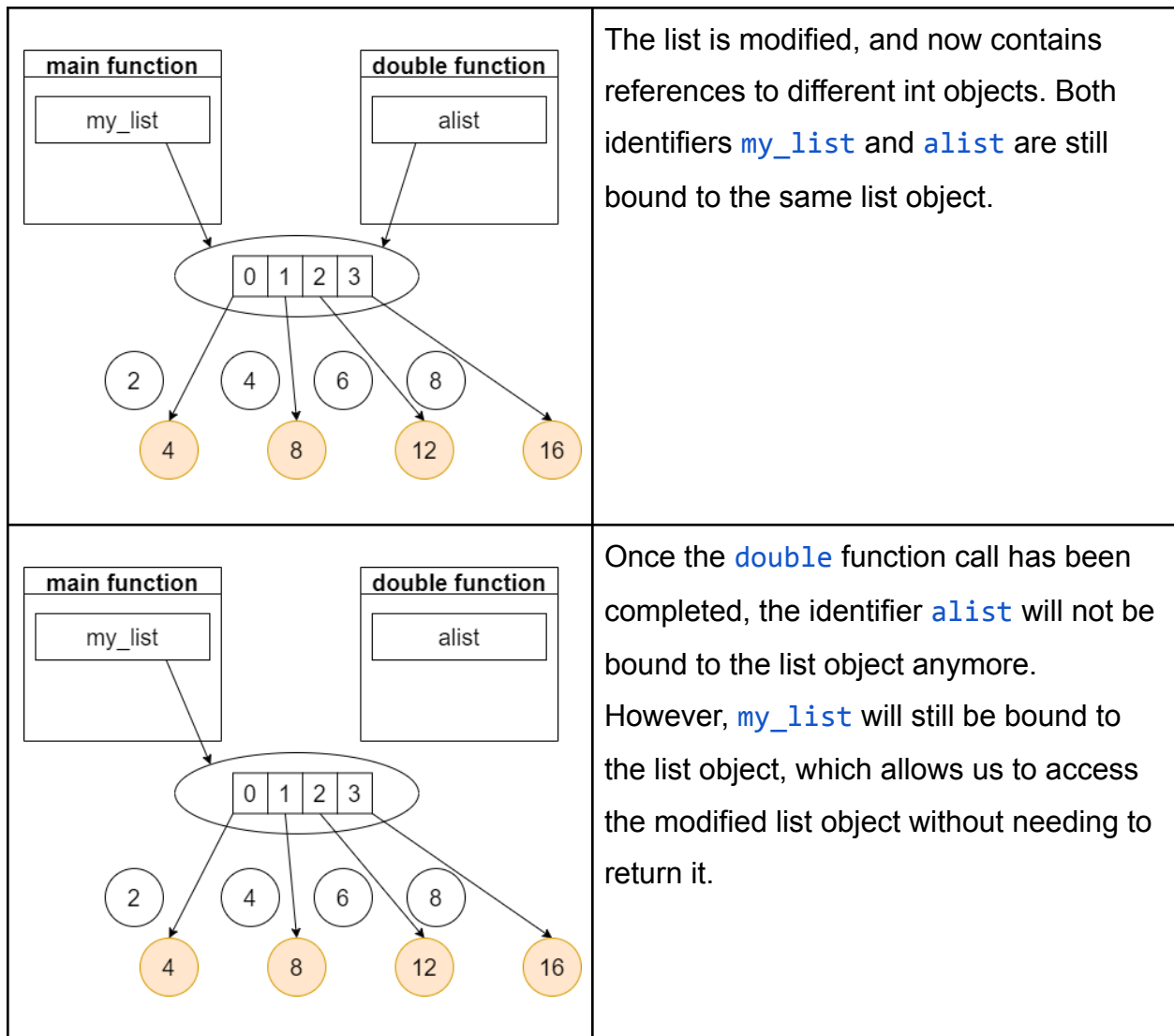
```
def double(alist):  
    for i in range(len(alist)):  
        alist[i] = alist[i] * 2  
def main():  
    my_list = [2,4,6,8]  
    double(my_list)  
    print(my_list)  
main()
```

Here is a simple program. `double` is a user defined function that multiplies a given number. This user defined function will double the value of every integer in the list. When `my_list` is printed, it will display the list with its contents all doubled `[4,8,12,16]`.

The user defined function `double` is able to modify the given mutable object, which is a list in this case. The user defined function `double` has a 'side effect'. This means that the function is able to modify an object inside the function, which can then be used outside of the function. Let's take a look at what's happening using diagrams:



During the function call, both identifiers `my_list` and `alist` are bound to the same list object.

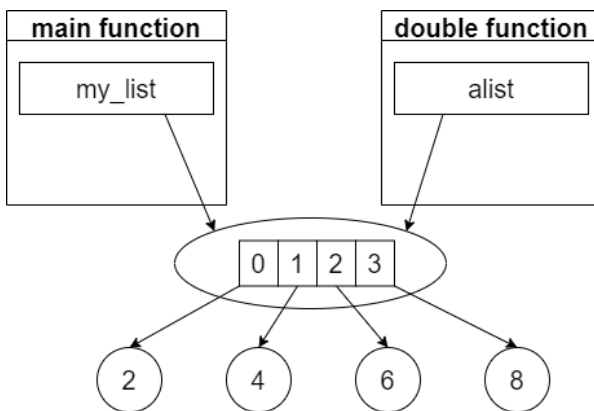


Now we'll take a look at an example that will require a return statement:

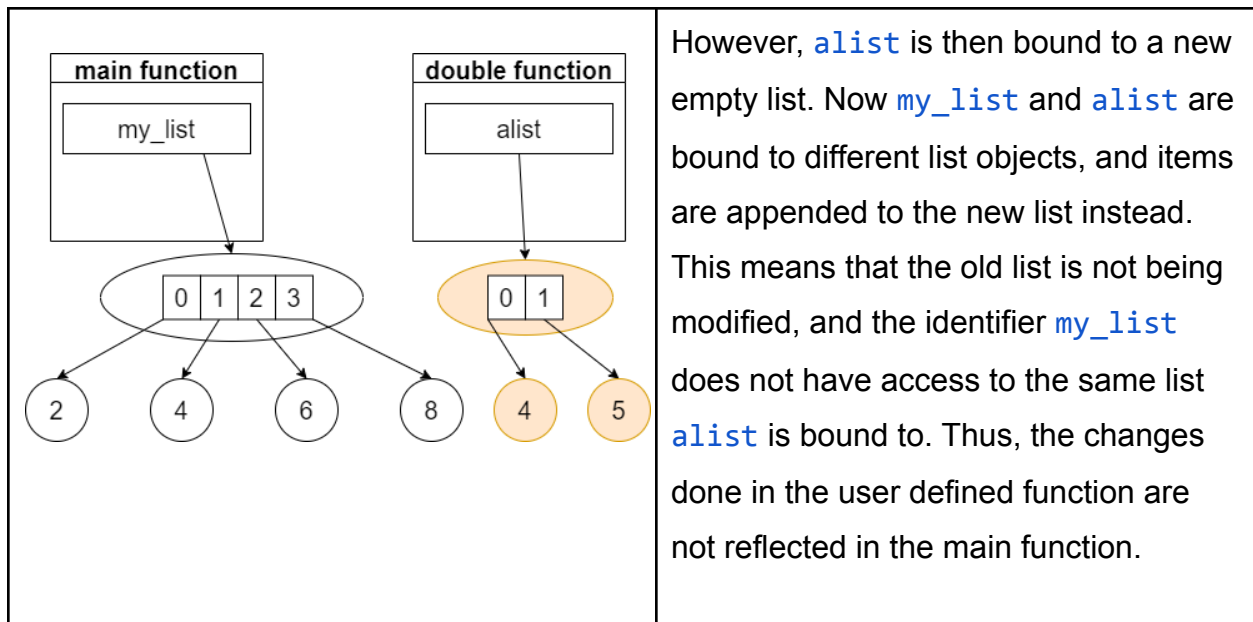
```
def change(alist):  
    alist = []  
    alist.append(4)  
    alist.append(5)  
def main():  
    my_list = [2,4,6,8]  
    change(my_list)  
    print(my_list)  
main()
```

Here is a simple program. `change` is a user defined function that creates a new list and adds two int objects to the list. When `my_list` is printed, it will display the list with its contents a `[2,4,6,8]`.

This user defined function does not modify the current list, but creates a new list. This causes the changes made to not be reflected in the current list. Let's take a look at what's happening using diagrams:



At the start of the function call, both identifiers `my_list` and `alist` are bound to the same list object.



1.5 Advantages of Using User Defined Functions

There are many advantages of using user defined functions. Here are three main advantages that we will explore in more depth:

1. Improving the structure and organization of your code by separating specific tasks, which will improve readability and maintenance of the code.
2. Reusing code, and reducing non adjacent duplicate statement groups
3. Creating a generalized solution

1.5.1 Structure, Organization, and Separation of Tasks

Sometimes code can be hard and tedious to read, especially if there are many lines of code. User defined functions can be used to structure and organize code, which can help make the code easier to read, understand, and maintain.

A program can often be divided into smaller sections, where each section is a single task. A game, for example, can have many features, which can include displaying messages, allowing the user to move or interact with items. Creating user defined functions for each of these tasks can help break down the larger problem into smaller subproblems to solve, but also organize the code and improve readability.

In the following example, we'll create a simple program to demonstrate how we can separate tasks into their own user defined functions. The program will ask the user to input a word, and a counter will be used to keep track of which words have been entered so far. The program will ask the user to input more words five times.

```
instructions = "This is a simple
program that will prompt the user to
enter a word five times."
print("*****")
print(instructions)
print("*****")
counter = 0
words_list = []
while counter < 5:
    word = input("Enter a word: ")
    words_list.append(word)
    counter = counter + 1
print("Words you have entered:")
for a_word in words_list:
    print(a_word)
```

```
def instructions():
    instructions = "This is a simple
program that will prompt the user to
enter a word five times."
    print("*****")
    print(instructions)
    print("*****")
def obtain_words(a_list):
    counter = 0
    while counter < 5:
        word = input("Enter a word: ")
        a_list.append(word)
        counter = counter + 1
def display_result(a_list):
    print("Words you have entered:")
    for a_word in a_list:
        print(a_word)

instructions()
words_list = []
obtain_words(words_list)
display_result(words_list)
```

1.5.2 Reducing Non-Adjacent Duplicate Statement Groups

There may be times where several lines of code are used together multiple times, but in different areas in your code. This type of repetition is called non-adjacent duplicate statement groups. In these cases, where a for or while loop is not useful in reducing this repetition, user defined functions can be used. Creating a function of these lines of code that will be repeated throughout the code can help keep our code simple.

Code A	Code B
<pre>print('Hello!') print("*****") print("press enter to continue") print("*****") input() print('How are you?') print("*****") print("press enter to continue") print("*****") input() print('Have a nice day!') print("*****") print("press enter to continue") print("*****") input()</pre>	<pre>def pause(): print("*****") print("press enter to continue") print("*****") input() print('Hello!') pause() print('How are you?') pause() print('Have a nice day!') pause()</pre>

In code A, we can see that there is a group of four lines of code that are constantly being repeated. These are called non-adjacent duplicate statement groups. These statement groups are repetitive, but their repetition cannot be reduced by using `for` or `while` loops. In code B, we've created a user defined function which has the non-adjacent duplicate statement groups as its suite. This way we can easily reuse

code by calling the function whenever those lines of code are needed. Using a user defined function also gives a descriptive name to those lines of code, making it easier to read, and understand the code in general.

1.5.3 Solution Generalization

There are times when certain pieces of code can be seen as special cases of more general tasks. In such cases, a function can be implemented with parameters which would make the function a more generalized solution.

Code A	Code B
<pre>def pause(): print("*****") print("press enter to continue") print("*****") input()</pre>	<pre>def display_message(symbol, message): border = symbol * 27 print(border) print(message) print(border) input()</pre>

Code A shows the `pause` user defined function we have seen above. This user defined function is called whenever we would like to pause the and wait for the user before continuing the program.

In code B, a couple changes have been made. Two parameters have been added: `symbol`, and `message`. The `symbol` parameter allows the user to choose what type of symbol they would like to have printed along with the message, and the `message` parameter allows the user to have control over what message will be displayed. This user defined function can now be adjusted and used in multiple situations, thus generalizing the solution to include for other situations.