Infrastructure as Code- Walkthrough

Functions

Contents

Introduction 2
Deliverables 2
Prerequisites 2
Functions 3
  Anatomy 3
  Passing and returning values 4
  Type Hints 6
  Tuple Unpacking 7
  Using map and lambda functions 8
Scope 9
Exercise 10
Finally 11

A **walkthrough** is intended to bring you through a technical exercise. A walkthrough shows you how I completed a task in a particular context, on specific systems, at a point in time. The document is a good basic guide, but you should always confirm that the information is current and if a more recent best practice exists, you should make yourself aware of it.

Introduction

This walkthrough is provided to demonstrate how to get started coding with Functions in Python. In any programming language, we will have blocks of code which we may need to call repeatedly. It would be very wasteful to insert the same code multiple times. when we write code in almost any programming language, we break up reusable blocks of code into functions.

There is a deeper philosophy behind the concept of functional programming, feel free to do some background reading.

Deliverables

You will be required to keep a list of commands with notes as to why you are using them as well as results of any testing you have done. You may be asked to present these notes as evidence of work completed or as part of a lab book. Check with your lecturer!

Prerequisites

1. I will carry out my exercises using a Windows 11 desktop. You can carry out this work on almost any modern platform.
2. You should have already installed Python as per my notes.
3. You should have already installed VSC as per my notes.
4. You have created a directory to keep example files in and you are ready to code.
    a. This could be on your OneDrive, in these examples, I am using OneDrive\Python\Exercises_05

Functions

A function in Python uses the **def** keyword to tell Python you are writing a function and indentation is used to define the code block that will be run. In the parenthesis (), parameters can be passed to the function.

The name of the function uses *snake_case* and the parenthesis indicates that this is a function.

As with loops, the colon denotes the start of a block.

We include a docstring to explain the function.

Anatomy

The anatomy of my first test function is

```
def name_of_function():
    """
    Simple test function
    """
    print("Yoo hoo!")
```

If I run this, nothing happens. I have created a function, but I have not called the function. I add the following line underneath, with no indentation and rerun.

```
name_of_function():
```

This final line calls the function. Try doing this without the brackets. What happens and why?

Passing and returning values

We normally pass variables to a function, and we may refer to them as arguments. Try the example below, where I pass a string to the function.

```
def name_of_function(first_name):
    """
    Simple test function
    """
    print(f"Yoo hoo, hello {first_name}!")

name_of_function("JOR")
```

We normally use the keyword **return** to send the output of a function back to the main program as a variable. I pass the value 5 to the function and then make the return value equal to **a**, finally I print **a**.

```
def calculate_circumference(radius):
    """
    Calculate the circumference of a circle based on its radius
    """
    return 2 * 3.142 * radius

a = calculate_circumference(5)
print(a)
```

If I leave out the value when I call the function, I will get an error:

```
TypeError: calculate_circumference() missing 1 required positional argument: 'radius'
```

One way to avoid this would be to use a default value.

```
def calculate_circumference(radius = 1):
    """
    Calculate the circumference of a circle based on its radius
    """
    return 2 * 3.142 * radius

a = calculate_circumference()
print(a)
```

I can use the input statement to take a value from the operator. Unfortunately, the input keyword returns a string, you cannot add a string to an integer and a float, I also need to do a conversion from string to float.

```
def calculate_circumference(radius):
    """
    Calculate the circumference of a circle based on its radius

    """
    return 2 * 3.142 * radius

# Get a radius value as a string
r = input("Provide a radius value: ")
# Convert it to a float
r_float = float(r)
# Call the function and create the variable for the return value
a = calculate_circumference(r_float)
print(a)
```

There may be cases where you want to pass an unknown number of arguments to a function. We could use the asterisk symbol * for this.

```
def auto_adder(*my_numbers):
    final_number = 0
    for number in my_numbers:
        final_number = final_number + number
    return final_number

print(auto_adder(12,34,23,66,8, 99))
```

In real coding, we can use the **sum()** function, we do not need to use a loop. This was a demonstration only!

Type Hints

Python is dynamically typed, so I could easily pass a string to the function and generate a bug! From Python v3.5 onwards, we have type hints.

I could tell the function that I am passing a float and expect it to return a float.

```
def calculate_circumference(radius: float) -> float:
```

Tuple Unpacking

Let's combine some of the things we've learned. I have a price list from the supermarket of the fruit I buy. I want to identify the most expensive single item.

```
def most_expensive(price_list):
    """
    Iterate through a list and find the most expensive item
    """
    # Set up the variables
    max_price = 0
    max_price_item = ""
    # Iterate, unpacking the tuple
    for description, price in price_list:
        # If this is the maximum price, record that in our variables
        if price > max_price:
            max_price = price
            max_price_item = description
        # If it is not the maximum price, do nothing
        else:
            pass
    # Return the maximum prices item and its price
    return max_price_item, max_price

# Provide the data
price_list = [("Pineapple", 1.0), ("Apples", .5), ("Pears", .7), ("Peaches", .8)]
# Call the function and unpack its return values
product, price  = most_expensive(price_list)
print(product, price)
```

Give this a go!

Change the penultimate line to

```
product, price, availability = most_expensive(price_list)
```

What error do you get and why?

Using map and lambda functions

The **map()** function provides us with a way to iterate through an iterable object without using a loop. I only pass the name of the function to map, and it has an anatomy like

map(function, iterable)

```
def double_number(n: int)->int:
    """
    Simple function to double a number
    """
    return n+n

# Create a list of numbers for testing
my_numbers = [1,2,3,4,5]
# Map my_numbers to the double_number function, return type is map
result = map(double_number, my_numbers)
# Print a list of the map
print(list(result))
# Or iterate through it
for item in map(double_number, my_numbers):
    print(item)
```

A lambda function is like a single-use function which is not needed perpetually, and it has an anatomy like

lambda arguments : expression

For example

```
circumference = lambda radius : 2 * 3.142 * radius
area = lambda radius : 3.142 * radius * radius
radius = 5
print(circumference(radius), area(radius))
```

Give this a try!

I use lambdas for very simple functions only. With a complex function, they are less readable than the full function.

Scope

When we create variables in any programming language, they are normally the concatenation of the variable and its namespace. We say that a variable has scope.

Run the following code with a nested function. What results do you get and why?

```
my_string = "global"

def my_function():
    my_string = "enclosing"
    def nested_function():
        my_string = "local"
        print(my_string)
    nested_function()
    return my_string

# Print the variable my_string
print(my_string)
# Print the output of the function, my_function
print(my_function())
```

Python has a hierarchy with which it looks for variables as **LEGB**.

1. **Local** names assigned in a function and not declared as global
2. **Enclosing** function locals, names in an enclosing function where we next functions
3. **Global** names assigned at the top level of a Python module, or declared as global
4. **Built in** names in Python, do not override these!

Using the global keyword, I can change the global variable name from within a function.

```
my_string = "global"

def my_function():
    global my_string
    print(f"At the moment, my_string is: {my_string}")
    my_string = "mangled!"

my_function()
print(f"Now the global value of my_string is: {my_string}")
```

Avoid using the global keyword unless you have a very specific reason to. Its much less confusing to change a module level variable using the return value of a function.

Exercise

Briefly explain and comment this code.

What values in the final statement will result in the function returning **True**? Why?

```
def divisible(numerator: int, denominator: int)->boolean:
    return numerator % denominator == 0

print(divisible(30,4))
```

Briefly explain and comment this code. Why do you get the value **None**?

What values in the final statement will result in the function returning **True**? Why?

Can you modify this function to return **False** instead of **None** if the value is not found?

```
def find_num(number_list: list, number: int)->boolean:
    for iterate_number in number_list:
        if iterate_number == number:
            return True
        else:
            pass

result = find_num([1,2,3,4,5,6,7,8], 9)
print(result)
```

Write a function to search for an even number in a list of numbers. Return **True** if you find an even number. Return **False** if you do not.

Write a lambda function to calculate the volume of a cylinder.

Finally

I have mentioned functional programming, but I have only provided an informal definition.

As a definition, functional programming does computation by combining functions that take arguments and return value(s) as a result, without modifying the input arguments or the program state.