

Python Chapter 5 Functions Activities

Contents

Python Chapter 5 Functions Activities	1
Online Tutorials.....	2
Functions	2
Tutorial 5.1: Miles to Kilometers.....	3
Tutorial 5.2: Rectangle Calculator	4
Scope and Lifetime	4
Local Variables	4
Modules	5
Tutorial 5.3: KM to Miles Calculator Module	6
Keyword Arguments	7
Default Arguments	8
Functions with Unknown Number of Arguments.....	8
Tutorial 5.4: Argument Soccer Mayhem	9
Docstrings	12
Input Validation with Boolean Functions	14
Try Except.....	14
Tutorial 5.5: Fahrenheit to Celsius	15
Tutorial 5.6: Fahrenheit to Celsius Try Except	16
lambda Function	17
Assignment 1: Conversion	18
Assignment Submission.....	18



No AI use.

Time required: 90 minutes

Online Tutorials

Go through the following tutorials.

- [LearnPython.org Functions](#)
- [Python Functions](#)

Functions

Functions in Python are blocks of organized, reusable code that perform a single, related action.

- **The `def` Keyword:** We start with the **`def`** keyword, which tells Python we're defining a function.
- **Function Name:** Choose a descriptive name that reflects what the function does. Let's call it **`greet`**.
- **Parentheses:** Functions can take inputs, called arguments, which we put inside parentheses. For our `greet` function, let's have a `name` argument.
- **Colon:** A colon follows the parentheses to mark the start of the function's body.
- **Indented Code:** The code that makes up the function's work needs to be indented. This shows Python what belongs to the function.

Parameters: Functions can take parameters, which are values you pass into the function. These parameters act as variables inside the function.

Arguments: Provide values for the parameters when calling the function.

Example: **`greet("Alice")`**

```
def greet(name):  
    print("Hello, " + name)  
  
greet("Alice")
```

Return Values: Functions can also return values using the **`return`** keyword. The value returned by the function can be stored in a variable when the function is called.

- We use the **`return`** keyword to send a value back from the function.
- We store the returned value (the result of the calculation) in a variable `answer`.

```
def square(x):  
    answer= x * x  
    return answer  
  
result = square(3)  
print(result)  
# Output: 7
```

Tutorial 5.1: Miles to Kilometers

Create a Python program called **miles_to_kilometers.py**

This program uses a function with a parameter and a return value. It uses a main() function to start the program.

```
1  """  
2      Name: miles_to_kilometers.py  
3      Author:  
4      Created:  
5      Purpose: Convert from miles to kilometers  
6  """  
7  
8  
9  Codiumate: Options | Test this function  
10 def miles_to_kilometers(miles):  
11     """  
12         Convert miles to kilometers.  
13         :param miles: Distance in miles  
14         :return: Equivalent distance in kilometers  
15     """  
16     kilometers = miles * 1.609344  
17     return kilometers  
18  
19 # Get user input for miles  
20 miles_input = float(input("Enter distance in miles: "))  
21  
22 # Call the function to convert miles to kilometers  
23 kilometers_result = miles_to_kilometers(miles_input)  
24  
25 # Display the result  
26 print(f"{miles_input} miles is: {kilometers_result:.2f} kilometers.")
```

Example run:

```
Enter distance in miles: 60
60.0 miles is: 96.56 kilometers.
```

Tutorial 5.2: Rectangle Calculator

- The **calculate_area()** function takes two arguments, length and width.
- Inside the function, calculate the area and return it using return.

```
1 def calculate_area(length, width):
2     """Calculates and returns the area of a rectangle."""
3     area = length * width
4     return area
5
6
7 length = float(input("Length of rectangle: "))
8 width = float(input("Width of rectangle: "))
9
10 # Call the function with two arguments
11 # Assign returned value to variable
12 rectangle_area = calculate_area(length, width)
13 print(f"The area of the rectangle is {rectangle_area}.")
```

Example run:

```
Length of rectangle: 3
Width of rectangle: 5
The area of the rectangle is 15.0.
```

Scope and Lifetime

Scope refers to visibility. When discussing the scope of a variable, it refers to the places in a program where a variable is visible and can be changed. Lifetime refers to how long a variable exists in memory. Lifetime is closely related to scope.

In general, the scope of a variable is the code block in which it is created and all the code blocks that are nested within that code block at a deeper indent level.

Local Variables

What happens in the function, stays in the function.

Let's say we have two functions like the ones below that each use a variable `i`:

```
def function1():
    for i in range(10):
        i = i + 1
    return i

def function2():
    i = 100
    function1()
    return i
```

A problem that could arise here is that when we call **function1**, we might mess up the value of **i** in **function2**. In a large program it would be a nightmare trying to make sure that we don't repeat variable names in different functions.

Fortunately, we don't have to worry about this. When a variable is defined inside a function, it is local to that function, which means it does not exist outside that function. Each function can define its own variables and not have to worry about if those variable names are used in other functions.

Modules

Python modules are like Lego bricks for your code. They store functions, variables, and other definitions that you can import and use in your programs.

Create a module named **circle_module.py** that contains a function to calculate the area of a circle.

```
from math import pi
def calculate_circle_area(radius):
    """Calculate and return the area of a circle."""
    area = pi * radius * radius
    return area
```

In the same folder, create a program named **circle_program.py** to import and use the function.

```
import circle_module

# Get the area of a circle with radius 5
circle_area = circle_module.calculate_circle_area(5)

# Output: Display the area of the circle
print(f"The area of the circle is {circle_area}.")
```

Tutorial 5.3: KM to Miles Calculator Module

This tutorial uses a module to convert KM to miles.

```
1  """
2      Name: km_to_miles_module.py
3      Author:
4      Created:
5      Purpose: A module to convert Kilometers to Miles function
6              with 1 argument and one return value
7  """
8  # Global constant for conversion
9  KM_TO_MILES = 0.621371
10
11
12  Codiumate: Options | Test this function
13  def calculate_miles(kilometers: float) -> str:
14      """The calculate_miles function accepts kilometers as a float argument
15          | converts to miles, returns a formatted string
16      """
17      # Convert passed in kilometers variable to miles
18      miles = kilometers * KM_TO_MILES
19
20      # Return the conversion result as a formatted string
21      return f"{kilometers:.2f} kilometers is: {miles:.2f} miles."
```

The main program.

```

1  """
2      Name: km_to_miles.py
3      Author:
4      Created:
5      Purpose: Convert Kilometers to Miles using a module
6  """
7  import km_to_miles_module
8
9
10     Codiumate: Options | Test this function
11     def main():
12         """Main program function starts here."""
13
14         # Get distance in kilometers
15         kilometers = float(input("Enter the distance in kilometers: "))
16
17         # Call calculate_miles with a float argument
18         result = km_to_miles_module.calculate_miles(kilometers)
19
20         print(result)
21
22     main()

```

Example run:

```

Enter the distance in kilometers: 5
5.00 kilometers is: 3.11 miles.

```

Keyword Arguments

Python offers another convention for passing arguments, where the meaning of the argument is dictated by its name, not by its position - it's called keyword argument passing.

```

def introduction(first_name, last_name):
    print(f"Hello, my name is {first_name} {last_name}")

introduction(first_name = "James", last_name = "Bond")
introduction(last_name = "Skywalker", first_name = "Luke")

```

The concept is clear - the values passed to the parameters are preceded by the target parameters' names, followed by the = sign. The position doesn't matter here - each argument's value knows its destination on the basis of the name used.

Default Arguments

It happens at times that a particular parameter's values are in use more often than others. Such arguments may have their default (predefined) values taken into consideration when their corresponding arguments have been omitted.

They say that the most popular English last name is Smith. Let's try to take this into account. The default parameter's value is set using clear and pictorial syntax:

```
def introduction(first_name, last_name="Smith"):  
    print(f"Hello, my name is {first_name} {last_name}")  
  
introduction(first_name = "James")  
introduction(last_name = "Skywalker", first_name = "Luke")
```

```
Hello, my name is James Smith  
Hello, my name is Luke Skywalker
```

Functions with Unknown Number of Arguments

Python allows functions with an unknown number of arguments. The argument definition uses the * to indicate that any number of arguments can be sent to the function. The function does have to be written to accommodate an unknown number of arguments.

```
def calculate_total(*args):  
    total = 0  
    for number in args:  
        total += number  
    print(total)  
# function calls  
calculate_total(5, 4, 3, 2, 1)  
calculate_total(35, 4, 3)
```

Example run:

```
15  
42
```

You can combine both specific arguments, and multiple arguments.


```
def the_rest(first, second, third, *args):
    print(f"First: {first}")
    print(f"Second: {second}")
    print(f"Third: {third}")
    print(f"And all the rest... {list(args)}")

the_rest(1, 2, 3, 4, 5)
```

Example run:

```
First: 1
Second: 2
Third: 3
And all the rest... [4, 5]
```

```
def sum(*args):
    result = 0
    for i in args:
        result += i
    return(i)

print(sum(1, 3, 5, 6))
return_sum = sum(100, 80, 33)
print(return_sum)
```

Tutorial 5.4: Argument Soccer Mayhem

Function with default arguments: We define the **player_stats** function with default values for goals, assists, and minutes played. If these values are not provided, they default to 0. Default arguments are useful when you want to make a parameter optional and provide a reasonable default value if the caller doesn't specify one.

Function with keyword arguments: In **player_goals** we specify values for specific statistics using the ****kwargs** parameter. This allows flexibility in specifying the values.

When calling the function, we use keyword arguments to specify values for these parameters. The order in which we provide the keyword arguments doesn't need to match the order of the parameters in the function definition, making it more flexible and readable.

Function with unknown number of arguments: We create a function that accepts an unknown number of arguments using the ***args** parameter. This function will display additional statistics for the player. You can call the function with different numbers of arguments, and it will still work correctly.

```

1  """
2  * Name: soccer_stats.py
3  * Written by:
4  * Written on:
5  * Purpose: Demo of 3 different types of arguments
6  """
7
8
9  # ----- DEFAULT ARGUMENTS ----- #
10 # Function with default arguments
11 def player_stats(
12     player_name: str = "John Doe", goals: int = 0, assists=0, minutes_played=0
13 ):
14     # There are default values for some of the arguments
15     # If an argument is not supplied, the default values are used
16     msg = (f"Player: {player_name}\n")
17     msg += (f"Goals: {goals}\n")
18     msg += (f"Assists: {assists}\n")
19     msg += (f"Minutes Played: {minutes_played}\n")
20     return msg
21
22
23 # ----- UNKNOWN NUMBER OF ARGUMENTS ----- #
24 # Function with an unknown number of arguments
25 def additional_stats(player_name, *args):
26     print(f"Player: {player_name}")
27     # Loop through however many arguments are supplied
28     statistics = ""
29     for stat in args:
30         # Concatenate the statistics into a single string
31         statistics += f"{stat}\n"
32     return statistics

```

```

35 # ----- KEYWORD ARGUMENTS ----- #
36 # Function with keyword arguments
37 def player_goals(player_name, **kwargs):
38     print(f"Player: {player_name}")
39     # Display only the arguments that match the keywords
40     msg = ""
41     if "regular_goals" in kwargs:
42         msg += f"Regular Goals: {kwargs['regular_goals']}\n"
43     if "penalty_goals" in kwargs:
44         msg += f"Penalty Goals: {kwargs['penalty_goals']}\n"
45     return msg
46
47
48 print("Default Arguments")
49 # Call the function with default arguments
50 statistics = player_stats(player_name="Lionel Messi", assists=10)
51 print(statistics)
52
53 print("Unknown Number of Arguments")
54 # Call the function with various statistics
55 stats = additional_stats(
56     "Neymar",
57     "Yellow Cards: 3",
58     "Red Cards: 1", "Assists: 10"
59 )
60 print(stats)
61
62 print("Keyword Arguments")
63 # Call the function with keyword arguments
64 goals = player_goals("Cristiano Ronaldo", regular_goals=20, penalty_goals=5)
65 print(goals)

```

Example run:

```
Default arguments
Player: Lionel Messi
Goals: 0
Assists: 0
Minutes Played: 0

Unknown number of arguments
Player: Neymar
Yellow Cards: 3
Red Cards: 1
Assists: 10

Keyword arguments
Player: Cristiano Ronaldo
Regular Goals: 20
Penalty Goals: 5
```

Docstrings

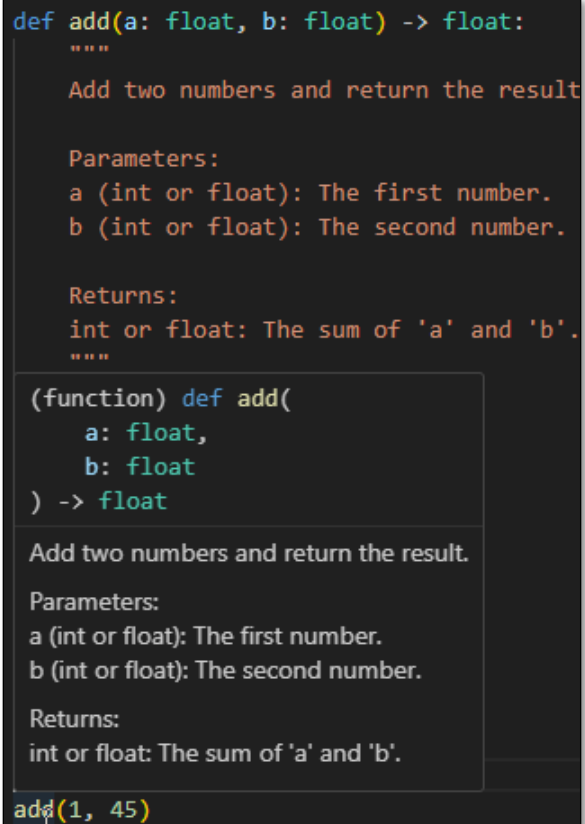
A docstring in Python is a string literal that serves as documentation for a module, function, class, or method. It is placed immediately after the definition and is enclosed in triple quotes (either single or double). Docstrings are used to provide information about what the code does, how it should be used, and any relevant details.

1. **Documentation Purpose:** Docstrings are primarily used for documenting code to make it more understandable for developers, including yourself and others who might use or maintain the code.
2. **Location:** Docstrings are placed immediately after the definition. This makes them easily accessible when someone wants to learn about the code's purpose and usage.
3. **Triple Quotes:** Python allows docstrings to be enclosed in either triple single quotes (```) or triple double quotes (````). Triple quotes are used to allow multi-line docstrings.
4. **Content:** Docstrings typically include information such as a brief description of the code's purpose, explanations of parameters and return values, usage examples, and any special considerations or notes.
5. **Access:** Docstrings can be accessed directly through many IDE's such as Visual Studio Code. Hover your mouse over the function call, the function definition and docstring appear. Many Python libraries use docstrings to in a similar fashion to help the programmer quickly understand how to use the function.

Docstring example:

```
def add(a, b):  
    """  
    Add two numbers and return the result.  
  
    Parameters:  
    a (int or float): The first number.  
    b (int or float): The second number.  
  
    Returns:  
    int or float: The sum of 'a' and 'b'.  
    """  
    return a + b  
add(1, 45)
```

In VSCode.



```
def add(a: float, b: float) -> float:  
    """  
    Add two numbers and return the result.  
  
    Parameters:  
    a (int or float): The first number.  
    b (int or float): The second number.  
  
    Returns:  
    int or float: The sum of 'a' and 'b'.  
    """  
add(1, 45)
```

The screenshot shows a VS Code editor with a Python function definition. A tooltip is visible, showing the function signature: `(function) def add(a: float, b: float) -> float`. The docstring content is also visible in the tooltip and in the background code.

Input Validation with Boolean Functions

The following code provides a Boolean return value. Instead of catching program exceptions, this function evaluates whether the input fits the range of the data. This function can be combined with the last input function.

```
def main():
    number = int(input("Please enter a positive number: "))
    # while the is_invalid function returns true
    # the loop keeps asking for valid input
    while is_invalid(number):
        print("Try again.")
        number = int(input("Please enter a positive number: "))

    print(f"Success! {number} is a positive number. ")

def is_invalid( number ):
    """While this function returns true, the data is invalid"""
    if number < 1:
        status = True
    else:
        status = False

    return status

main()
```

Example run:

```
Please enter a positive number: -1
Try again.
Please enter a positive number: 2
Success! 2 is a postive number.
```

Try Except

Things go wrong, errors happen, let's learn to handle those exceptional events with try except.

An exception is a signal that a condition has occurred that can't be easily handled using the normal flow-of-control of a Python program. Exceptions are often defined as being "errors" but this is not always the case. All errors in Python are dealt with using exceptions, but not all exceptions are errors.

Go through this short tutorial for a better understanding of exceptions.

- [Python Try Except](#)

Basic syntax of exception handling

```
try:
    # Code that might raise an exception
except ExceptionType as e:
    # Code to handle the exception
```

Tutorial 5.5: Fahrenheit to Celsius

Create and test a program named **fahrenheit_to_celsius.py** This program does not have any exception handling.

Note: You can get the ° degree from any number of websites or copy it from this sentence.

```
1  """
2      Name: fahrenheit_to_celsius.py
3      Author:
4      Created:
5      Purpose: Convert Fahrenheit input to Celsius
6  """
7
8  # Get Fahrenheit temperature from user
9  fahrenheit = float(input("Enter a Fahrenheit temperature: "))
10
11 # Calculate the Celsius equivalent
12 celsius = ((fahrenheit - 32.0) * 5.0) / 9.0
13
14 # Display the Celsius temperature
15 print(f"{fahrenheit:.2f}° Fahrenheit is: {celsius:.2f}° Celcius.")
```

Example program runs:

```
Enter a Fahrenheit temperature: -40
-40.00° Fahrenheit is: -40.00° Celcius.
```

If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
Exception has occurred: ValueError ×
could not convert string to float: 'barney'

File "Z:\_WNCC\Computer Science\Assignments Python\05 Functions
    fahrenheit = float(input("Enter a Fahrenheit temperature: "))
    ~~~~~
ValueError: could not convert string to float: 'barney'
```

There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called "try / except". The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.

You can think of the **try** and **except** feature in Python as an "insurance policy" on a sequence of statements.

Tutorial 5.6: Fahrenheit to Celsius Try Except

We rewrite our temperature converter as follows:

```
1  """
2      Name: fahrenheit_to_celsius_try.py
3      Author:
4      Created:
5      Purpose: Use try except to handle improper input
6  """
7
8  # Try to execute the program
9  try:
10     # Get Fahrenheit temperature from user
11     fahrenheit = float(input("Enter a Fahrenheit temperature: "))
12
13     # Calculate the Celsius equivalent
14     celsius = ((fahrenheit - 32.0) * 5.0) / 9.0
15
16     # Display the Celsius temperature
17     print(f"{fahrenheit:.2f}° Fahrenheit is: {celsius:.2f}° Celcius.")
18
19 # Catch and handle a program exception
20 except:
21     # Display an error message
22     print("Please enter a number next time.")
```


Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds. If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

```
Enter a Fahrenheit temperature: barney
Please enter a number next time.
```

Handling an exception with a try statement is called catching an exception. In this example, the except clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

lambda Function

A **lambda** function is an inline, single, or anonymous function which is defined without a name. It is used for simple, short functions, instead of writing out an entire function. Lambda functions are often used in graphical user interface programs.

- A lambda function can take any number of arguments.
- It can only have one expression.
- The result is automatically returned.
- It is defined using the **lambda** keyword.

```
# Filename: lambda_simple.py
# lambda arguments: expression
# argument: a
# expression: a * a
x = lambda a: a * a
# Call and print the lambda function
print(x(5))
```

Example run:

25

x is passed into the lambda function. "Hello" is return if x > 5, else "bye" is returned.

```
x = lambda x: "hello" if x > 5 else "bye"
print(x(10))
print(x(4))
```

Example run:



Assignment 1: Conversion

Time to create your own conversion program.

Create a Python program named **conversion.py**

- Convert a measurement from one unit to another unit.
- Look up the formula on the internet.
- Create a function in an external module to do the conversion.
- Include a docstring for the function.
- Return the conversion to the main program.

Assignment Submission

1. Use pseudocode or TODO.
2. Comment your code to show evidence of understanding.
3. Attach the program files.
4. Attach screenshots showing the successful operation of the program.
5. Submit in Blackboard.