1

Custom-Defined Functions

Overview

In this handout, we will explore custom-defined functions. You will learn how to define functions using the def keyword, understand the importance of return statements, and utilize parameters, including positional, keyword, and default parameters. We will also cover advanced topics such as lambda functions. By the end of this handout, you will be able to create and use your functions.

# 1 Introduction

So far, you have used built-in Python functions such as len(), min(), print ( ) and id(). In this section, we will learn how to define and use our own functions. Defining functions makes reading and maintaining a program significantly easier. A function is a group of statements that perform a specific task. Once defined, a function can be called and used in different parts of a program. The concept of a function in programming is similar to a function in mathematics. It usually accepts inputs (arguments), performs specific tasks, and returns outputs.

# 2 Definition

The following is the template for defining a function:

|  |  |
| --- | --- |
| def function\_name (parameters) : stat ement (s)  return expression | # a function is defined using the def keyword  # tasks to be performed  # return output (s), if any |

A Python function is defined using the keyword def, followed by the function name (which must be a valid identifier), a set of parentheses with possible parameters inside, and a colon Parameters are separated by commas ,. The lines after the colon constitute the function block and must be indented. A function does not have to return an output; if it does, the output is specified using the return keyword.

|  |
| --- |
| Example: Defining the max\_value function |
| def max\_value (num\_ 1, num\_ 2) :  This function accepts two numbers and returns their macimum if num 1 < num 2: return num 2 |

I These notes and examples adapt the references listed at the end. They are compiled to fit the scope of this specific course.

|  |
| --- |
| else:  return num 1 |

This defines a function named max\_value. This function takes two arguments, num\_ 1 and num\_ 2. The function returns the maximum of the two input parameters.

So far, you've defined (declared) a function. Now, it's time to execute (call) it. You can call a function by typing its name and passing the necessary arguments, like this:

|  |
| --- |
| Calling the max\_value function |
| max\_value (2, 5) |

When you run this line, Python assigns 2 to num\_ 1 and 5 to num\_ 2, and returns 5. As you can see, when arguments are not explicitly named, Python assigns them in the order they are provided to num\_ 1 and num\_ 2. Alternatively, you can call the function like:

max\_value (num\_l = 2, num\_ 2 = 5) or max\_value (num\_ 2 = 5 , num\_ 1 = 2) , where you explicitly specify which argument corresponds to num\_l and num\_ 2.

|  |
| --- |
| Function Definition Template |
| To define a function, use the def keyword followed by the function name and parentheses. Inside the parentheses, specify any parameters. Use the return statement to send back a value from the function. def function name (parameters) :  Function body return value |

# 3 Function docstring

In the above example, the string inside triple quotes  in the first line of the function definition is a docstring. This is the recommended way to document the purpose of a function. The Style Guide for Python Code (https://www.python.org/dev/peps/pep-0257/) states that each function should contain a docstring that explains its purpose and details. When you call a function, you can access its docstring using one of the following methods:

## 3.1 Displaying docstring in Jupyter

In JupyterLab or Jupyter Notebook, use one of the following methods:

Method 1 : Type a ? following the function name (without parentheses) and run it. This will provide information about the function, including its docstring.

Method 2: Type ?? following the function name (without parentheses) and run it. This will also give you the function's source code (if available).

Method 3: Type the function's name (without parentheses) and press Shift-Tab. Once you read the doc string, press Esc to close the menu.

You can also apply the above methods to get information about Python's built-in functions such as len(), print ( ) , etc.

### 3.2 Displaying docstring in PyCharm

In PyCharm, by typing the function's name (without parentheses) and pressing Ctrl-Q, you can get the function's docstring. Alternatively, you can choose Quick Documentation from the View menu.

|  |
| --- |
| Docstring |
| A docstring is a special type of comment used to describe what a function does. It is written as the first statement in the function and is enclosed in triple quotes. Docstrings help improve code readability and can be accessed using the help() function. |

#### 3.3 Variable Scope

A variable's scope is where you can access it. Consider the following snippet returning the sum of two numbers, x and y:

|  |
| --- |
| Defining the calc \_ sum Function |
| def calc\_sum(x, y) :  Calculates sum of two numbers result = x + Y return result |

The variable result is created inside the function and can be accessed only within the function. Trying to access it outside the function block results in an error. Here is an example:

|  |
| --- |
| Attempting to Access a Local Variable Outside Its Scope |
| def calc \_ sum (x, y) :  Calculates sum of two numbers result = x + Y return result  print (result) |

This yields an error:

|  |  |
| --- | --- |
| NameError : | name 'result' is not defined |

In other words, the variable result is local to the function calc sum.

## 4 Default Parameter Values in Function Definitions

You can assign default values to function parameters. This means that if the user does not provide a value for a parameter, the function will use the default value.

Here is an example showing how to define a function with default parameter values:

|  |
| --- |
| Function with Default Parameter Values |
| def calculate \_total (price, tax\_rate 0.08) :  Calculates the total cost including tac for a given price. return price tax\_rate) |

In this example, the calculate \_total function has two parameters: price and tax\_rate. The tax\_rate parameter has a default value of 0.08.

You can call it with both arguments, as shown in the code below:

|  |
| --- |
| Calling the Function with Both Arguments |
| print (calculate \_total (100, 0. 1)) |

The output will be:

110.0

You can also call the same function by providing only the price, as shown in the line below:

|  |
| --- |
| Calling the Function with Default Parameter |
| print (calculate \_total (100) ) |

The output will be:

108.0

## 5 Understanding Positional and Keyword Arguments

Before we proceed, it's important to briefly explain the difference between positional and keyword arguments.

### 5.1 Positional arguments

Positional arguments are arguments that are passed to a function in order without specifying the parameter names. The function assigns each value to the corresponding parameter based on its position (order). Here is an example:

|  |
| --- |
| Defining a Function with Positional Arguments |
| def send\_message (arg\_l, arg\_ 2) :  " " "This function sends a message return f "{arg\_ 1}, is {arg\_ 2} ! |

Let's call this function:

|  |
| --- |
| Calling send\_message with Positional Arguments |
| print (send\_message ( "BARM" , "The Best")) |

The output will be:

BARM is The Best!

But if you call it with reversed order of the arguments, such as:

|  |
| --- |
| Calling send\_message with Reversed Positional Arguments |
| print (send\_message("The Best " , "BARM")) |

The output will be:

The Best is BARM!

As you can see, the order we feed the arguments matters because arg\_ 1 and arg\_ 2 are positional arguments, meaning they are assigned values based on their position in the function call.

#### 5.2 Keyword arguments

Keyword arguments are arguments that are passed to a function by explicitly specifying the parameter names along with their values. This allows us to provide arguments in any order, enhancing both flexibility and readability of the code.

For example, we can call the same function using keyword arguments:

|  |
| --- |
| Calling send\_message with Keyword Arguments |
| print (send\_message (arg\_ 2= 1' The Best , arg\_ BARM")) |

The output will be:

BARM is The Best!

In this example, we used the parameter names arg\_ 1 and arg\_ 2 when calling the function. Because the arguments are passed by name, the order of the arguments no longer matters.

Using keyword arguments improves clarity, especially in functions with many parameters, making the code more understandable and less prone to errors related to argument order.

Now, let's explore how this connects to \*args and which allow even more flexibility when working with both positional and keyword arguments.

|  |
| --- |
| Positional and Keyword Arguments |
| Positional arguments are the most common way to pass values to a function. They are assigned to parameters based on their position in the function call.  Keyword arguments allow you to specify which parameter you are passing a value to by using the parameter name. This makes the code more readable and allows you to skip optional parameters. |

6 A Function with an Arbitrary Number of Arguments

#### 6.1 \*args as a Function Argument

We are often unaware of the number of arguments the user will feed into our custom-defined function. In this case, we can use \*args to signal Python that the user may enter an arbitrary number of arguments. The \*args collects all the arguments passed to the function into a tuple named args.

The following snippet defines a function that accepts an arbitrary number of parameters and calculates their average:

|  |
| --- |
| Defining calc\_avg with Arbitrary Arguments |
| def calc\_avg(\*args) :  Calculates the average of given numbers if len(args) - o.  return 'You need to input at least one number ' else :  average = sum (args) / len(args) return average |

Here, \*args allows the function to accept any number of arguments, the values are passed based on their position. Each value can be of any type (string, integer, list, etc.). These values are stored in a tuple named args. You can then use the args tuple inside the function block. Now, you can call this function with as many input arguments as you wish:

|  |
| --- |
| Calling calc\_avg with Different Numbers of Arguments |
| print (f Average of no number: {calc \_ avg ( ) print (f ' Average of one number:  print (f ' Average of three numbers: {calc\_avg(14, 2, print (f ' Average of five numbers: {calc\_avg(2.5, 3, 0.4, 30, |

This yields:

Average of no number: You need to input at least one number

Average of one number : 20.0

Average of three numbers : 8.0 Average of five numbers: 7.18

Note: The word args is just a convention, and you can use any name of your choice. In the above example, would have worked and would have stored all input arguments in a tuple named numbers.

\*args

|  |
| --- |
| The \*args syntax in a function definition allows you to pass a variable number of positional arguments to the function. It collects extra positional arguments into a tuple, enabling the function to handle more inputs than the number of parameters defined. |

#### 6.2 as a Function Argument

In the previous example, \*args collects all the positional arguments into a tuple. However, sometimes we may want to accept arguments that are provided with a name. In this case, we can use  to allow the function to accept any number of keyword arguments. The  collects these keyword arguments into a dictionary named kwargs.

Here is an example:

|  |
| --- |
| Defining a Function with |
| def order\_ summary (\*\*kwargs) :  "Displays a summary of a food order with detai Is . print ( "Order Summary :  for key, value in kwargs . items ( ) :  print (f " {key . capitalize ( ) } {value} |

Now, you can call this function with any number of keyword arguments, allowing for flexible input based on the details of the order:

|  |  |  |
| --- | --- | --- |
| Calling order \_ summary with Keyword Arguments |  |  |
| order\_ summary (item= "Burger " , quantity extras="Cheese" , | side="Fries" , | drink=" Cola") |

This will output:

Order Summary :

Item: Burger

Quantity : 2

Extras : Cheese Side: Fries

Drink: Cola

In this example,  allows us to pass in any number of keyword arguments in the form of key-

value pairs. These arguments can vary depending on what kind of details we want to include. The keys represent different attributes of the order (such as item, quantity, extras), and the values represent their corresponding details (like "Burger" 2, "Cheese").

When you use \*\* before a parameter name in a function definition, it collects all keyword arguments into a dictionary. This allows your function to accept any number of keyword arguments, which can be very useful when you do not know beforehand which arguments will be passed.

Similar to \*args, the name kwargs is a convention; you can use any name you like, such as Let's create a function that summarizes personal information:

|  |
| --- |
| Defining summarize \_ info with \*\*kwargs |
| def summarize\_info(\*\*data) :  Summarizes personal information summary — "Summary of Information:  for key, value in data. items ( ) : summary key . capitalize ( ) return summary |

Now, you can call this function:

|  |  |  |
| --- | --- | --- |
| Using summarize \_ info |  |  |
| info summarize \_ inf o (name = Johny'l , age=25, print (info) | profession=" Sr . | Business Analyst Il ) |

This will output:

Summary of Information:

— Name : Johny

— Age: 25

— Profession: Sr. Business Analyst

While it is difficult to provide a precise template for using  in function definitions, the following structure is common in many practical applications:

|  |
| --- |
| def function name :  \_  Process key-value pairs.  This processing is done using  — kwargs. keys ()  — kwargs. values ()  — or kwargs. items ()  Simi tar to what you see below  for key, value in kwargs . items ( ) :  # Process each key—value pair. |

# Below Line simpl,y prints them print (f " {key} {value} " )

A common structure for passing key-value pairs to this function is in the form of:

function\_name (keyl = value 1, key2 = value2, key3 = value3,

In this template, keyl, key2, and key3 must be valid Python identifiers (strings without quotes). You should not use quotations like or around keyl, key2, or key3 in the function call. Python will automatically treat these as strings and convert them into the appropriate keys for the kwargs dictionary. valuel, value2, value3 could be any type (string, integer, list, etc.).

Understanding the difference between positional and keyword arguments helps us to use \*args and effectively in our functions. While \*args allows us to pass a variable number of positional arguments, lets us pass a variable number of keyword arguments, giving our functions greater flexibility.

You can also define functions that accept arbitrary positional and keyword arguments. I encourage you to explore this on your own.

The syntax in a function definition allows you to pass a variable number of keyword arguments to the function. It collects extra keyword arguments into a dictionary.

7 Return Multiple Values from a Function

When writing a function that returns multiple values, we can use different data structures like tuple, list, and dictionary. Here's an example function that accepts four numbers and returns the minimum, maximum, and average using each of these structures.

A tuple is useful when you want to return a fixed set of values. In this case, it returns the minimum, maximum, and average of the numbers.

|  |
| --- |
| Returning Multiple Values Using a tuple |
| def stats\_tuple(a, b, c, d) :  Returns min, mac, and average as a tuple minimum = min (a, b, c, d) maximum = max (a, b, c, d) average return minimum, maximum, average |

Calling this function:

|  |  |  |
| --- | --- | --- |
| Calling the function |  |  |
| min\_val, max\_val, avg print (f "Min: {min val}, | stats \_tuple (5,  Max: {max\_val} , | 8, 2, 10)  Average: {avg |

This will output:

Min: 2, Max: 10, Average : 6.25

A list is ideal when you need a sequence of values that can be modified. Here, we return the same statistics in a list.

|  |
| --- |
| Returning Multiple Values Using a list |
| def stats \_ list (a, b, c, d) :  Returns min, mac, and average as a list minimum = min (a, maximum = max (a, average return [minimum , maximum, average] |

Calling this function:

|  |  |  |  |
| --- | --- | --- | --- |
| Calling the function |  |  |  |
| result list stats \_ list (5, print (result\_list) | 8, | 2, | 10) |

This will return:

##### [2, 10, 6.25]

A dictionary is perfect when you want to label the values you're returning, making it clearer what each value represents.

|  |  |
| --- | --- |
| Returning Multiple Values Using a dictionary |  |
| def stats \_ dict (a, b, c, d) :  Returns min, mac, and average as a dictionary minimum — min (a, b, c, d) maximum — max (a, b, c, d) average return {"min" . minimum , "max" . maximum, " average " | average} |

Calling this function:

|  |  |  |  |
| --- | --- | --- | --- |
| Calling the function |  |  |  |
| result dict stats \_ dict (5, print (result\_dict) | 8, | 2, | 10) |

This will return:

'min' 2, 'max' : 10, 'average' : 6.25

|  |
| --- |
| Returning Multiple Values |
| You can return multiple values from a function using a tuple, list, or dictionary. This allows you to pass back more than one value in a single return statement. |

1. return or NO return?

8.1 Is There a Difference?

So far, we have used the return keyword to specify the output of a function. You might wonder if the same result can be achieved with the print ( ) function. Let me be clear: Don't do it! Let's see a few simple examples:

|  |
| --- |
| Defining add\_numbers Function |
| def add\_numbers (x, y) :  This function returns the sum of two numbers return x + Y |

Now, if you try:

|  |  |
| --- | --- |
| Calling add\_numbers |  |
| c add numbers (  \_  print (f 'sum is: {c} l ) print (f 'type of output : | {type (c) |

The output would be:

sum is: 7 type of output : <class 'int

There is no surprise here. Now, let's try something that, on the surface, looks identical:

|  |
| --- |
| Defining add numbers\_revised Function  \_ |
| def (x, y) :  This function prints the sum of two numbers print (x + y) |

By calling the same set of commands:

|  |
| --- |
| Calling add\_numbers\_revised |
| c print (f 'sum is: {c} print (f 'type of output: {type (c) } ' |

The output would be:

7 sum is: None type of output : <class 'NoneType'>

That's not what you expected, is it? You probably expected to see:

7

7

<class 'int

Why did this happen? Keep reading...

#### 8.2 Importance of return Keyword in Function Definition

A function is a block of code that accepts some input and returns a result. Inputs are optional, but from Python's internal perspective, a return value is not. If you write a function without a return statement, like the above function, Python will still return something. If your function does not have a return statement, Python will automatically return None 2.

In the case of the  function, Python returns None because there is no return statement. This can cause problems when you're testing or debugging your code. Imagine you expect the function to return a number, like 7, but instead it returns None. You might not immediately realize what went wrong, making it harder to figure out the issue. When a function silently returns None instead of a useful value, it can make it difficult to trace bugs or understand why the function isn't working as expected.

For these reasons, we will have this firm rule in this class and moving forward:

##### Every function must return a useful value

Even if you are writing a function that doesn't seem to need a return value, always make it a habit to return True, False, or a meaningful message. The worst thing you can do is omit the return statement from a function.

|  |
| --- |
| Importance of the return Statement |
| The return statement is essential in a function as it allows the function to send back a value to the user, enabling further processing and enhancing code reusability. Without it, a function will return None by default, limiting its usefulness. |

2

We have introduced the concept of None in the previous handout.

1. Argument Validation (Sanity Check)

Consider the following simple function:

|  |
| --- |
| Defining add\_numbers Function |
| def add\_numbers (x, y) :  This function returns the sum of two numbers return x + Y |

Now, if you try:

|  |
| --- |
| Calling add\_numbers with Invalid Arguments |
| add\_numbers (3 ' testing' ) |

Since it is not possible to add an int and a str, you will understandably get an error:

|  |  |  |
| --- | --- | --- |
| TypeError : | unsupported operand type (s) for + : | int' and 'str' |

The input argument was invalid; hence, we got an error.

When writing functions, it's important to ensure that the input arguments are valid to prevent errors and unexpected behavior. One common issue is attempting operations between incompatible data types, which can lead to runtime errors. To avoid this, we can implement argument validation to check the input before performing any operations. There are two standard methods for argument validation:

* 1. Using the isinstance built-in function.
  2. Using the try. . . except. . . statement.

9.1 Using isinstance for Argument Validation

isinstance is a built-in function used to determine whether an object is of a particular type. This function returns

True or False.

Here is an example:

|  |
| --- |
| Using isinstance |
| x = 10 isinstance (x, int) |

returns

True and trying:

|  |
| --- |
| Checking Type with isinstance |
| str 1 — 'Carey' isinstance (str 1, float) |

returns

False

Here is another example:

|  |
| --- |
| Using isinstance with Multiple Types |
| z isinstance(z, (float, int, str, list, dict)) |

Checks if (1, 2, 3) is one of the types mentioned in (float, int, str, list, dict) and returns:

False

Because z is a tuple and tuple is not listed inside isinstance function.

isinstance can be used for argument validation. Here is one example:

|  |
| --- |
| Argument Validation Using isinstance |
| def add\_numbers (x, y) :  This function returns the sum of two numbers if both are valid. if isinstance(x, (int, float)) and isinstance(y, (int, float)) : return x + Y else :  return 'Error: please enter valid numbers! ' |

If both x and y are numerical types (int or float), the add\_numbers() function returns the sum of its arguments. Otherwise, the function will return 'Error: please enter valid numbers!

9.1.1 isinstance VS. type VS. is

When checking the type of an object in Python, you might consider using isinstance, type ( ) , or the is operator. While all of these can check types, they work differently. Let's look at the key differences.

* isinstance(): This checks if an object belongs to a specific class. It returns True if the object is of the given class and False otherwise. It's useful when you need to verify that an object is of the expected type before performing certain operations.
* type() : This function returns the exact type of an object. It checks whether the object is of the specified type by comparing the type directly. This is helpful when you need to know the precise class of an object, but it's more strict and less flexible than isinstance ( ) for type checking.
* is: This checks whether two objects are the exact same object in memory. It doesn't just compare values but checks if both variables point to the same memory location. Use this when you need to know if two variables refer to the same object, not just equal values. Let's see a few examples:

|  |  |
| --- | --- |
| Using isinstance() with Multiple Types |  |
| z isinstance (z, (float, int, str, list , | dict)) |

False

In this example, isinstance checks if z is any of the types listed (float , int , str, list, dict). The result is False because z is a tuple, which is not in the list of types.

|  |
| --- |
| Using type ( ) to Check Exact Type |
| print (type (x) — list) # True, because c is ecactLy a List |

True

Here, type (x) returns the exact type of x, which is list. This will return True if x is a list.

|  |
| --- |
| Using is |
| a 3] b 3]  if a is b:  print ( 'a and b point to the same object in memory ' ) else:  print ( 'a and b are different objects in memory ' # This will be printed |

a and b are different obj ects in memory

Although a and b contain the same values, they are different objects in memory, so a is b returns False 3. The is operator checks if two variables point to the exact same object, not just if they have the same value.

Let's see a practical example of how to validate the types of arguments passed to a function.

Here is a simple example

3 Alternatively, you can check this using the id() function.

|  |
| --- |
| Argument Validation Using isinstance |
| def add \_numbers (x, y) :  " " "This function returns the sum of two numbers if both are valid. " " " if isinstance(x, (int, float)) and isinstance(y, (int, float)) :  return x + Y else :  return 'Error: please enter valid numbers! ' |

Here, isinstance() is used to check if both x and y are either integers or floats. If they are, the function returns their sum; otherwise, it returns an error message. Calling this function:

|  |
| --- |
| Calling the function with Invalid Arguments |
| add numbers (3  \_ |

returns

|  |  |
| --- | --- |
| Error: | please enter valid numbers ! |

This example shows how isinstance helps ensure that only valid types are passed to the function. Alternatively, let's use type() for the same function:

|  |
| --- |
| Argument Validation Using type ( ) |
| def add \_numbers (x, y) :  This function returns the sum of two numbers using type() .  if type (x) in [int, float] and type (y) in [int, float] :  return x + Y else :  return 'Error: please enter valid numbers! |

Calling this function:

|  |  |
| --- | --- |
| Calling the function with Invalid Arguments | |
| add numbers (3 ' 5 ' )  \_ | |
| Error: | please enter valid numbers ! |

Expectation in this course: We recommend using the isinstance function for type checking as it is more flexible and supports checking multiple types easily.

#### 9.2 Using try. . . except. Statement

The try. . . except. statement is widely used for error handling. Discussing try. . . except . . . statements can be very elaborate; interested readers can consult the Python documentation at https : //docs .python.org/3/ tut orial/errors . html. Here, we will provide a simple example showing a generic use of try . . . except. . . For most applications, this structure is a good starting point.

When defining a function using try. . . except. . . , you need to embed the function logic under the try block and use the except block for handling possible errors.

Here is the previous example with the try . . . except . . . statement:

|  |
| --- |
| Argument Validation Using try. . . except. . . |
| def add\_numbers (x, y) :  " " "This function returns the sum of two numbers try :  return x + Y except Exception as e:  return e |

Trying:

|  |
| --- |
| Calling add\_numbers with Valid Arguments |
| add numbers (  \_ |

Yields:

5

Whereas:

|  |
| --- |
| Calling add\_numbers with Invalid Arguments |
| add numbers (2  \_ |

Yields:

TypeError("unsupported operand type (s) for + : int' and 'str'

try. . . except. blocks can also utilize raise, else, and finally clauses. Details of error handling are outside the scope of this course.

|  |
| --- |
| Sanity Checks: isinstance and try except |
| Sanity checks help ensure that inputs to functions are valid. Two common methods are:  • isinstance: Checks the type of a variable. This helps confirm that the input is of the expected type.  except: Catches exceptions that occur during execution, allowing the program to handle errors more strategically. |

### 10 A Few Hints About Writing Good Functions

Defining good functions dramatically improves the readability and maintainability of your code. On the other hand, having a few poorly written functions may cause headaches for you and anyone else who reads or uses your code.

Here are some rules of thumb for writing good functions in Python:

* A Good Function Name: Writing a good function starts with a sensible and clear name. Clear, long names are always better than short, unclear ones! Try to use full English words instead of abbreviations and acronyms.

— Avoid using names like get \_ something or do \_ something. Avoid referring to the input type in the function's name. Terms such as get \_ something\_from\_list and do \_ something\_with\_dictionary can be reduced to something. All other details, such as input data type, can be elaborated in the doc string.

* Clear docstring: Include a docstring in every function you write, no matter how short or simple. Use complete sentences and proper grammar and punctuation to describe the purpose of the function, explain the input arguments, and discuss the return value.
* Proper Length: The rule of thumb is to keep function length under 20—30 lines. If your function definition exceeds this limit, you can most likely break it into several smaller functions. Multiple smaller functions are much better than one very long function. Longer functions are much harder to debug and understand.
* return Statement: Any well-written function must have a return statement, and it must return something useful; no exceptions.
* Function Specialization: A well-written function does one thing and does it well. Do not try to accomplish too much with just one function! There is no shame in defining multiple functions that perform different tasks.

Remember:

A small, single-purpose function is a good function

# 11 lambda Functions

Functions improve code organization and code reusability. Earlier, we saw that the general syntax of a function definition is:

|  |
| --- |
| def function\_name (parameter \_list) :  <function block> |

return expression

Sometimes, a function has such a simple definition that instead of formally defining it, we can define it on the go without a name. A lambda function (expression), or simply a lambda, is a small anonymous (no name) function that can take any number of arguments but can only have one expression. In other words, lambda functions are small anonymous one-line functions. lambda functions in Python have their roots in Lambda Calculus (https: //en.wikipedia.org/wiki/Lambda\_calculus)

A lambda has a very concise syntax beginning with the lambda keyword followed by a comma-separated parameter list, a colon : , and an expression.

lambda parameter\_l , parameter\_2 parameter\_n : expression

Here is an example:

|  |
| --- |
| Defining a lambda Function |
| lambda x, y: x + Y |

This defines a simple lambda function that accepts two input parameters, x and y, and returns their sum x + y. You can apply the defined lambda to arguments by surrounding the lambda definition and its arguments with parentheses, as in:

|  |
| --- |
| Using a lambda Function |
| (lambda x, y: x 4) |

Yields:

7

Please note that:

|  |  |
| --- | --- |
| lambda parameter\_ list: | expression |

is equivalent to:

def function\_name (parameter \_list) : return expression

The following section will show one common application of lambda functions.

lambda Function

A lambda function is a small anonymous function defined using the lambda keyword. It can take any number of arguments but can only have a single expression. lambda functions are often used functions like map(), filter(), and sorted().

## 11.1 lambda Function with the sorted Function

lambda functions are used very frequently with the built-in sorted() function. sorted() sorts members of a given iterable in ascending or descending order and returns a new sorted list. The syntax of the sorted() function is as follows:

sorted (iterable, key=None, reverse = False)

sorted() has two important optional parameters:

* Optional reverse argument with a default value of False. For sorting in descending order, set reverse=True.
* Optional key argument with a default value of None. You can pass a function to the key argument for the sort comparison. This function is called on each iterable element before making comparisons. It is common practice to use a lambda function with the key argument. Here are a few examples:

|  |  |  |
| --- | --- | --- |
| Sorting a list Alphabetically |  |  |
| names list 'Milo' , 'Daisy' , sorted (names \_ list) | ' Ray' , | ' Howard ' |

Yields an alphabetically sorted list:

C ' Daisy Howard 'Milo' ' Ray ' ]

Now, let's try to sort the names \_ list according to their length:

|  |  |  |
| --- | --- | --- |
| Sorting by Length Using len |  |  |
| names list 'Milo ' , Daisy sorted (names \_ list, key=len) | Ray | Howard ] |

Yields:

C ' Ray 'Milo' 'Daisy' Howard ]

By passing the built-in function len as a key, only the lengths of the list items are compared and sorted. Let's do the same thing using a lambda function:

|  |
| --- |
| Sorting by Length Using lambda |
| names list - c 'Milo' , 'Daisy' , ' Ray' , ' Howard ' ] sorted (names \_ list, key=lambda item: len (item) ) |

This passes every item of names \_ list to the lambda function; this function returns len (item) for each item. Finally, the sorted function compares only the lengths of items and returns:

C Ray 'Milo' 'Daisy' Howard ]

Imagine, for some reason, you would like to sort members of names \_ list based on the third character of each member. This is easily doable by:

|  |
| --- |
| Sorting by Third Character Using lambda |
| names list 'Milo ' , Daisy Ray Howard ] sorted (names \_ list, key=lambda item: item [2] ) |

This passes every item of names \_ list to the lambda function; this function returns each item's third character (at index 2). Finally, the sorted function compares only the third characters and returns:

C ' Daisy' , 'Milo' , 'Howard' ' Ray

Let's conclude this section with a more elaborate practical example. Imagine you have a list of employees:

|  |
| --- |
| list of employee dictionaries |
| employees - c  {'name' : 'Tina' , age' : 23},  • Alexis age' . 35},  {'name' : 'Cynthia' , 'age' • 32} |

You are tasked with sorting and printing the employee dictionaries by their age. This can be easily accomplished using the lambda function, as shown in the snippet below:

|  |  |
| --- | --- |
| Sorting by Third Character Using lambda |  |
| employees  { 'name' • ' Tina ' 'age' . 23},  {'name' : ' Alexis ' age' 35},  { name ' Cynthia ' age' 32}  sorted\_ employees — sorted(employees, key=lambda x: print (sorted\_ employees) | age 'I ) |

The output will be:

C 'name' : 'Tina' , age' • 23, ' name Cynthia' , age • 32, name Alexis " age' • 35]

12 Distinction Between function, module, package, and library

So far, we have used Python's standard library and its vast capabilities.

Python's standard library has well over 200 modules such as math, random, and decimal. Each module has numerous submodules and functions, such as the sqrt function in the math module or the randint function in the random module.

Python has a hierarchical structure for organizing these capabilities, similar to the file structure in your computer, where a folder may contain a set of subfolders, and each of those subfolders may include files and more subfolders. The following describes this hierarchical structure:

* A collection of Python variables, classes, and functions is called a module. [[1]](#footnote-1)
* A collection of related modules is usually called a package.
* A library is a collection of one or more packages.

## 12.1 A Few Notes About Module Import Practices

This section explains a few important notes about module importation using the random module of the standard library as an example. This approach can be used with any other library/module.

Python has a standard library. You can import a module called random from this library using:

|  |
| --- |
| Importing the random Module |
| import random |

A function called random() in the random module generates a random number between 0 and 1 . You can access this function using:

|  |
| --- |
| Using random. random() |
| import random random random ( ) |

To shorten the notation, you can use:

|  |
| --- |
| Importing with an Alias |
| from random import random as rd rd ( ) |

By executing rd() without the need to precede it with the module name random and a dot you can generate a random number. It is possible to import more than one identifier from a module. Here is an example:

|  |
| --- |
| Importing Multiple Identifiers with Aliases |
| from random import random as rd, randint as r int |

Now you can simply use the shortened names rd and rint.

## 12.2 Wildcard Import Using \*

It is possible to import all identifiers in a module with a wildcard import in the form of:

from modulename import \*

This makes all the module's identifiers available in your code.

Warning: Avoid wildcard imports! This can lead to errors. Here is an example:

|  |
| --- |
| An Example of Naming Collision |
| Pi delicious  from math import \* print (pi) |

This defines an object 'delicious' and binds the variable pi to it. Then it uses a wildcard import and finally prints pi. This will yield:

3 . 141592653589793

Now let's try:

|  |
| --- |
| Another Example of Naming Collision |
| from math import \* ' delicious '  print (pi) |

This does a wildcard import, then defines an object 'delicious' and binds the variable pi to it. By printing pi, you will get:

delicious

As you can imagine, there is a high possibility of naming collisions. So please avoid wildcard imports as much as possible.

|  |
| --- |
|  |
| In the next handout, we will explore Pandas, a powerful data manipulation and analysis library for Python. We will cover essential functionalities, including data structures like Series and DataFrames, how to import and export data, and perform operations such as filtering, grouping, and aggregating. |

# 13 Exercises

1. Write a function calculate \_ sample \_variance (data) that accepts a list of numbers and calculates the sample variance.
2. Define a function (numbers) that takes a list of integers and returns a new list containing only the even numbers from the original list.
3. Create a function count \_vowels (string) that takes a string as input and returns the count of vowels (a, e, i, o, u) in the string, ignoring case.
4. Create a function  that takes a string as input and returns the string reversed.
5. Write a function calculate \_ factorial (n) that takes a non-negative integer n and returns its factorial.
6. Define a function merge \_ dictionaries (dictl, dict2) that takes two dictionaries and merges them into a single dictionary. For

|  |  |  |
| --- | --- | --- |
| • dictl = { | 1 | 2 } and |
| • dict2 = { | 3, | 4 } |

merge \_ dictionaries (dict 1, dict2) Should return { 'a' : 1 , 'b' : 3, 'c' : 4

1. Define a function merge \_dictionaries (dictl, dict2) that takes two dictionaries and merges them into a single dictionary. If there are overlapping keys, sum their values. For example:

• dictl = { 1, 'b' • 2 }



merge \_ dictionaries (dict 1, dict2) should return { 'a' : 1 , 'b' • 5 , 'c' : 4

1. Write a function roll\_dice (num\_rolls) that accepts the number of rolls as an argument and generates the results of rolling a six-sided die the specified number of times. Use the random module.
2. Define a function generate \_random\_password (length) that generates a random password containing uppercase letters, lowercase letters, digits, and special characters. The length of the password should be specified as an argument. Use the random module. Hint: Look into the random. choice() function to select random characters.
3. Create a function calculate\_mean (numbers) that takes a list of numbers as input and returns the mean (average) value.

1 1 . Create a function count \_ occurrences (data, target) that takes a list and a target value, and returns the number of times the target appears in the list.

12. Write a function percentage) that takes a list of grades and a percentage as input.

Of

The function should remove the specified percentage of the highest and lowest grades and then return the mean of the remaining grades. Ensure to handle cases where the percentage might result in removing more grades than are available.

## 14 Exercise Solutions

Solutions to these problems can be found on the following GitHub page: https : / /github . com/NaserNikandish/Python\_For\_Data\_Ana1ysis You can also access the same link using the QR code below:



## 15 References

|  |
| --- |
| References and Resources  The following references and resources were used in the preparation of these materials:  1 . Official Python website at https : //www.python.org/.   1. Introduction to Computation and Programming Using Python, John Guttag, The MIT Press, 2nd edition, 2016. 2. Python for Data Science Handbook: Essential Tools for Working with Data, Jake VanderPlas, O'Reilly Media, 1st edition, 2016. 3. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and [Python, Wes McKinney, O'Reilly Media, 2nd edition, 2017. 4. Introduction to Python for Computer Science and Data Science, Paul J. Deitel, Harvey Deitel, Pearson, 1st edition, 2019. 5. Data Visualization in Python with Pandas and Matplotlib, David Landup, Independently published, 2021 . 6. Python for Programmers with Introductory A/ Case Studies, Paul Deitel, Harvey Deitel, Pearson, 1st edition, 2019. 7. Effective Pandas: Patterns for Data Manipulation (Treading on Python), Matt Harrison, Independently published, 2021 . 8. Introduction to Programming in Python; An Interdisciplinary Approach, Robert Sedgewick, Kevin Wayne, Robert Dondero, Pearson, 1st edition, 2015.   1 0. Python tutorials at https : //betterprogramming . pub/.  1 1 . Python learning platform at https : //www.learnpython.org/.  1 2. Python resources at https : //realpython . com/.  1 3. Python courses and tutorials at https : / /www. datacamp . com/. |

of

1. Any Python file with the extension . py is considered a module. [↑](#footnote-ref-1)