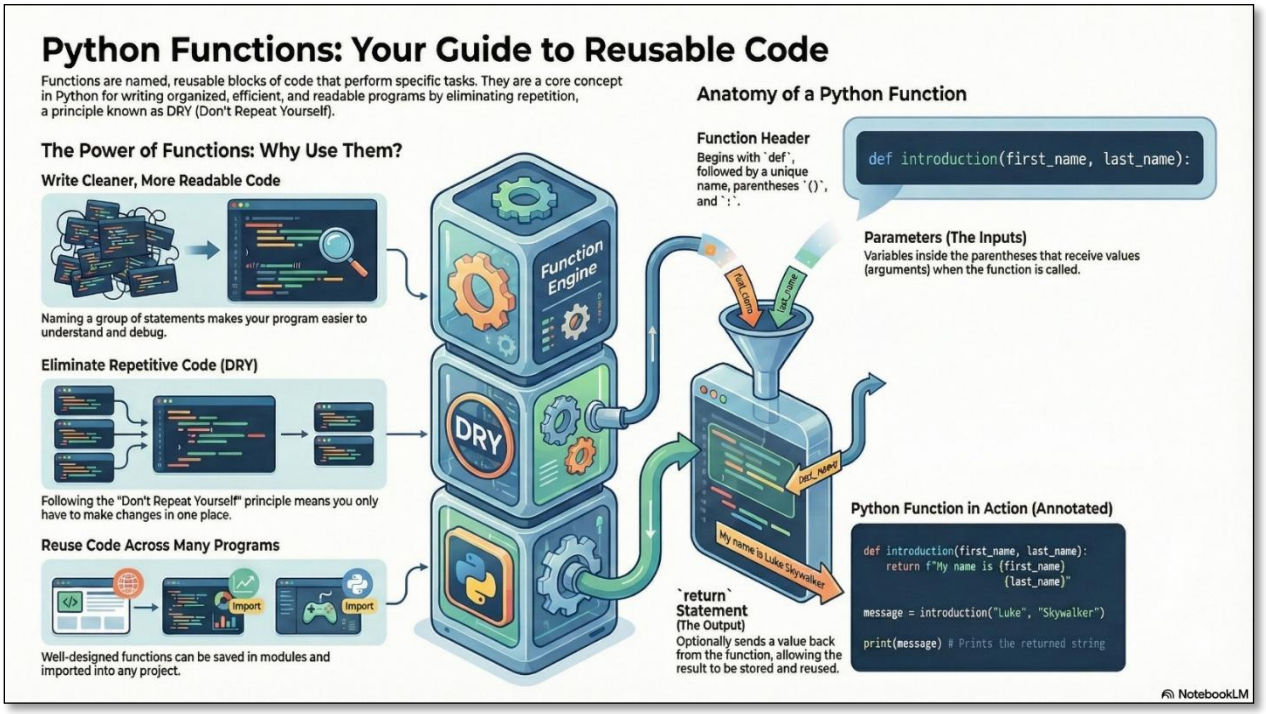
Chapter 5: Python Fun with Functions!

Contents



Red light: No AI

Time required: 180 minutes

DRY

Don't **R**epeat **Y**ourself (**DRY**) is a principle of software engineering aimed at reducing repetition of software patterns. It you are repeating any code, there is probably a better solution.

Visualize and Debug Programs

The website www.pythontutor.com helps you create visualizations for the code in all the listings and step through those programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. This will help you to better understand the behavior of the programs we are working on.

This is a great way to debug your code. You can see the variables change as you step through the program.

www.pythontutor.com

Learning Outcomes

Students will be able to:

- Define the components of a function header
- Define and produce a function body
- Understand and use scope and lifetime
- Understand argument passing and use
- Understand and properly call methods, void and value returning
- Understand and use the proper return syntax
- Understand how to use returned values in your calling code
- Understand how to create and use modules

Comments and Docstrings

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Docstrings are a nice-looking way to comment your code.

Single Line Docstrings

```
''' Three single quotes is fun. '''
"""
    We can also use three double quotes.
"""
```

A docstring can use ''' or """ to begin and end a docstring.

Multiple Line Docstrings

```
"""
    Takes in a number n, returns the square of n
    Second line
"""
```

Why Functions?

Why is it worth the trouble to divide a program into functions?

- **Clearer Code:** Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- **Simpler Code:** Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only make it in one place.
- **Easier Debugging:** Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- **Code Reuse:** Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Elements of a Function

A **function** is a named block of reusable code that performs a single, related task. Functions provide modularity and promote code reuse.

Python includes many built-in functions like print() and len(). You can also **create your own**—these are called **user-defined functions**.

To use a function, you need to know:

- **The name** of the function
- **The parameters** it requires (if any)
- **The return value** it provides (if any)

Syntax for a Python function definition.

```
def function_name(parameters):
    """Optional docstring explaining what the function does"""
    # Function body (statements to execute)
    return [expression]  # Optional return value
```

Example: Simple Function.

```
def simple_function():
    """Prints a message to show it was called"""
    print("I am a simple function.")

# Call the function
simple_function()
```

Function Name

Each function in Python has a **name**.

- A function name follows the same rules as variable names:
  - ✅ Letters, digits, and underscores
  - ❌ Cannot start with a digit
  - ❌ Cannot use Python reserved words
- **Naming conventions:**
  Use **lowercase letters** with **underscores** to separate words (snake_case).
  Example: **calculate_total()**, **print_report()**
- A function name should clearly describe what the function does.

Function Call Format

When you use (or **call**) a function:

- Write the function name
- Follow it with parentheses: **function_name()**

Even if the function has **no parameters**, **you must include the parentheses** when calling it.

Void and Value-Returning Functions

Void function: does a specific job and then terminates.

Value returning function: returns a value back to the point where it was called.

Creating Functions

Functions are defined using the **def** statement.

This starts the **function header**, which always ends with a **colon (:)**.

The lines that follow are the **function body**, which must be **indented**.
Indentation tells Python which statements belong to the function.

Example: Simple Function

```
def print_hello():
    print("Hello!")

print_hello()     # Function call
print("1234567")  # Regular print statement
print_hello()     # Function call again
```

```
Hello!
1234567
Hello!
```

- Lines 1–2 define the function.
- Lines 4–6 are **function calls**.
- The function prints "Hello!" whenever it is called.

Why Use Functions?

Functions help reduce **repetition** in code.

If you need the same output or logic in many places, you define it **once** in a function.

Example: Drawing a Box of Stars

Instead of repeating the same print statements:

```
def draw_box():
    print("*" * 15)
    print("*"," "*11, "*")
    print("*"," "*11, "*")
    print("*" * 15)
draw_box()
```

```
***************
*             *
*             *
***************
```

One benefit of a function like this is that if you decide to change the size of the box, you modify the code in the function. If you had copied and pasted the box-drawing code everywhere you needed it, you would have to change all of them.

Tutorial 5.1: Function Junction

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

Pay attention to the flow of the code.

```python
"""
Name: message_function.py
Author:
Created:
Purpose: Demonstrate a Simple Function
"""


def main():
    """Main program function starts here"""
    print("First message function call.\n")

    message()
    print("Second message function call.\n")

    message()
    print("Exit main function, program ends.")


def message():
    """Define the message function"""
    print("The best time to plant a tree was 20 years ago.")
    print("The second best time is now.")
    print("Chinese Proverb\n")


# Call the main function to start the program
main()
```

Example run:

```
First message function call.

The best time to plant a tree was 20 years ago.
The second best time is now.
Chinese Proverb

Second message function call.

The best time to plant a tree was 20 years ago.
The second best time is now.
Chinese Proverb

Exit main function, program ends.
```

Assignment 5.1: Create and Use Functions

Objective

Write a Python program named **function_1.py** that uses **functions** to make the code cleaner and reusable.

Customize the messages to make it your own program.

Step-by-Step Instructions

1. Define a Function print_welcome()

- It should print:

```
======================
Welcome to My Program!
======================
```

2. Define a Function print_menu()

- It should print:

```
1. Start
2. Help
3. Exit
```

3. Main Program

- Call **print_welcome()**
- Call **print_menu()**
- Print: **Please enter your choice:**

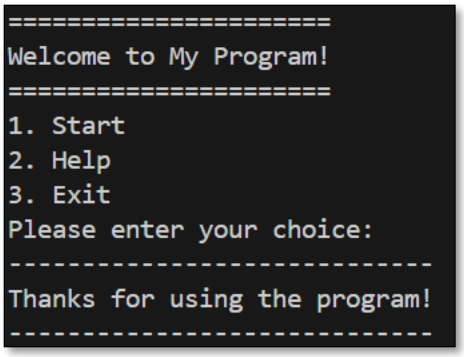
4. Bonus Challenge

- Create a function called **print_goodbye()** that prints:

```
----------------------------
Thanks for using the program!
----------------------------
```

- Call it at the end of your script.

Example run:



Scope and Lifetime of Variables in Python

Scope

Scope refers to where a variable can be **seen** and **used** in a program.

- A variable's scope is limited to the block of code where it was defined.
- Blocks are usually functions, loops, and conditional statements.

Lifetime

Lifetime is how long a variable **exists in memory** during program execution.

- A variable’s lifetime begins when the function is called.
- It ends when the function finishes running.

Local Variables

Variables created **inside a function** are called **local variables**.

"What happens in the function, stays in the function."

Local Variables

What happens in the function, stays in the function.

Each time the function is called:

- New local variables are created.
- They do **not affect** variables in other functions, even with the same name.

```
def function1():
    for i in range(10):
        print(i)  # i exists only in this loop inside function1

def function2():
    i = 100      # This i is separate from the one in function1
    function1()
    print(i)     # Prints 100, not affected by function1

function2()
function1()
```

When **function2()** is called:

1. It sets **i = 100**
2. Calls **function1()**, which uses a **different i**
3. After **function1()** finishes, the original **i = 100** is still intact

No Conflict

Even though both functions use the name **i**, they **do not interfere** with each other. This is because each i is local to its own function.

Summary

Term	**Meaning**
Scope	Where a variable can be accessed
Lifetime	How long a variable stays in memory
Local Variable	A variable defined inside a function

Passing Arguments to Functions

When you **pass a value** to a function, that value is called an **argument**.

Inside the function, it is stored in a **parameter**, which is just a local variable.

Key Concept

- **Argument** → the value you send **into** the function
- **Parameter** → the variable in the function that **receives** the value

argument → parameter

Example: Passing a Single Argument

```
# n is the parameter
def print_hello(n):
    print("Hello" * n)
    print()

# 3 is the argument
print_hello(3)

# 5 is the argument
print_hello(5)

# times is a variable holding an argument
times = 2
print_hello(times)
```

```
Hello Hello Hello
Hello Hello Hello Hello Hello
Hello Hello
```

How It Works

- **print_hello(3)** → passes 3 into the function
  So n = 3, and it prints "HelloHelloHello"
- **print_hello(5)** → n = 5, so it prints "Hello" five times
- **times = 2**, so print_hello(times) → passes 2 into n

Assignment 5.2: Functions with One Parameter

Objective

Practice defining and calling functions that accept **one parameter**.

Instructions

Write a Python program that includes **three separate functions**, each using **one parameter**.

HINT: Create and test each function one at a time.

1. Function: print_stars(n)

- **Parameter:** n – the number of stars to print on one line
- **Behavior:** Print * repeated n times
- **Example Call:** print_stars(10)
- **Expected Output:**

```
**********
```

2. Function: print_square(size)

- **Parameter:** size – the width and height of a square
- **Behavior:** Print a square made of # characters
- **Example Call:** print_square(4)
- **Expected Output:**

```
####
####
####
####
```

3. Function: repeat_hello(times)

- **Parameter:** times – how many times to print "Hello!"
- **Behavior:** Print "Hello!" times number of times
- **Example Call:** repeat_hello(3)
- **Expected Output:**

```
Hello!
Hello!
Hello!
```

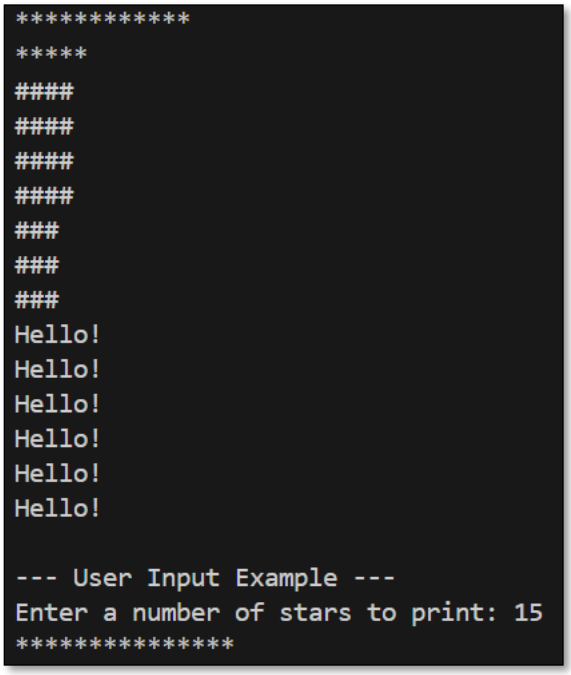
4. Main Program Requirements

- Call each function at least **two times** with different values.
- At least one call should pass a **variable** instead of a number.

5. Extra Challenge

Ask the user to input a number and pass that to one of the functions.

Example run:



Functions with Multiple Parameters

Functions can accept **more than one argument**. Each argument corresponds to a **parameter** in the function definition.

Key Concept: Positional Arguments

- Arguments are matched to parameters **by their position** in the function call.
- The first argument goes to the first parameter, the second to the second, and so on.

Example: Introduction Function

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

introduction("Luke", "Skywalker")
introduction("Jesse", "Quick")
introduction("Clark", "Kent")
```

Example: Function to Calculate nth Root

```
def nth_root(number, root):
    return number ** (1 / root)

# Calculate the square root of 16
print(nth_root(16, 2))  # Output: 4.0

# Calculate the cube root of 27
print(nth_root(27, 3))  # Output: 3.0
```

Summary

- Functions can have multiple parameters.
- Arguments are passed in order (positional arguments).
- Inside the function, parameters behave like local variables.
- You can return values from functions using return.

Assignment 5.3: Functions with Multiple Parameters

Objective

Practice defining and calling functions that take **two or more parameters**.

Tasks

1. Function: calculate_area(length, width)

- Parameters:
    - length (number)
    - width (number)
- Returns the area of a rectangle (length * width)
- Print the result inside the function

2. Function: full_name(first_name, last_name)

- Parameters:
    - first_name (string)
    - last_name (string)
- Prints: "Full name: first_name last_name"

3. Function: convert_to_fahrenheit(celsius, decimal_places)

- Parameters:
    - celsius (float) — temperature in Celsius
    - decimal_places (int) — number of decimal places to round the result
- Converts Celsius to Fahrenheit (F = C * 9/5 + 32)
- Prints the converted temperature rounded to decimal_places

4. Main Program

- Call each function **at least twice** with different arguments.
- Use variables as arguments for at least one call per function.

Example run:

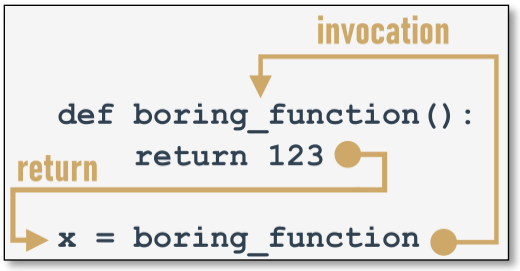
```
Area: 50
Area: 12
Full name: Alice Johnson
Full name: Bob Smith
Temperature: 98.6°F
Temperature: 23.0°F
```

Value-Returning Functions in Python

A **value-returning function** performs a task and then **returns a result** using the return statement.

Key Concepts

- The return statement:
    - Ends the function immediately
    - Sends a value **back to the caller**
- You can store this returned value in a **variable** for further use



Example: Convert Celsius to Fahrenheit

```
def convert(celsius):
    fahrenheit = ((celsius * 9.0) / 5.0) + 32
    return fahrenheit

# Print the returned value
print(convert(20))        # Output: 68.0

# Use the returned value in an expression
print(convert(20) + 5)    # Output: 73.0
```

Why Use return Instead of print() in Functions?

- If you use print() inside the function:
    - The result is displayed on screen, but cannot be reused
- If you use return, you can:
    - Store the result in a variable
    - Use it in calculations
    - Pass it to other functions

Example Comparison

❌ Only prints inside the function:

```
def convert(celsius):
    fahrenheit = ((celsius * 9.0) / 5.0) + 32
    print(fahrenheit)

result = convert(20)   # result is None — can't use it
```

✅ Returns the value:

```
def convert(celsius):
```

```
    return ((celsius * 9.0) / 5.0) + 32

result = convert(20)    # result is 68.0
print(result + 5)       # Output: 73.0
```

Assignment 5.4: Value-Returning Functions

Objective

Create a program to practice writing and using functions that **return values**, not just print them.

1. Function: square(num)

- Parameter: num (number)
- Returns: the square of the number (num ** 2)
- In the main program, call the function and:
    - Print the result
    - Add 10 to the result and print that too

2. Function: calculate_discount(price)

- Parameter: price (float)
- Returns: the price after a 20% discount
- In the main program:
    - Call the function with a price
    - Store the result in a variable
    - Print both the original and discounted prices

3. Function: average(a, b, c)

- Parameters: a, b, c (numbers)
- Returns: the average of the three numbers
- In the main program:
    - Call the function with 3 values

- Use the result in a message:
  "The average is: ___"

Example run:

```
Square of 4 is 16
Adding 10: 26

Original price: $100.0
Discounted price: $80.0

The average is: 7.5

98.6°F is 37.0°C
```

Tutorial 5.2: Creating Modules in Python

A **module** is simply a .py file that contains reusable code—usually **functions**, **variables**, or **classes**. You can import it into other Python programs to keep your code organized and modular.

1. Create a Module File

Filename: hello.py

```python
# hello.py
def world():
    print("Hello, World!")
```

This file defines a function, but doesn't call it. Running hello.py will produce no output.

2: Create a Program to Use the Module

Filename: hello_program.py *(Make sure it's in the same directory as hello.py)*

```python
# hello_program.py

# Import the hello module
import hello

# Call the function using dot notation
hello.world()
```

Example run:

```
Hello, World!
```

How It Works

- **import hello** loads the **hello.py** file
- **hello.world()** calls the **world()** function inside that module using **dot notation**

Why Use Modules?

Modules may seem unnecessary for small programs, but they offer **big advantages** as your projects grow:

Benefit	Description
Reusability	Write code once, use it in multiple projects
Organization	Keep related code grouped in separate files
Readability	Make complex programs easier to understand
Maintainability	Update or fix code in one place only

Assignment 5.5: Create and Use Your Own Python Module

Objective

Learn how to organize reusable code in a module and import it into another program.

1. Create a Module File

- Name the file: my_tools.py
- Inside this file, define **two functions**:
    - A function that takes a **name** as a parameter and prints a greeting (e.g., "Hello, Jordan!")
    - A function that takes a **number** as a parameter and returns its **square**

Do **not** call the functions in this file. Just define them.

2. Create a Program File

- Name the second file: my_tools_main.py
- This file should:

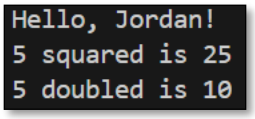
- Import your **my_tools** module
- Call the greeting function and pass your name as the argument
- Call the square function, store the result in a variable, and print it

3. Add a third function

Add a third function to your module that takes a number and returns **double** its value.

Use it in your main program.

Example run:



Tutorial 5.3: Print Title Function in utils.py

Create a module **utils.py** that includes a function to **print a formatted title**.
The function should work both:

- When called from another program, and
- When the file is run directly (to test the function).

Create a Python file named **utils.py**

```
"""
    Name: utils.py
    Author:
    Created:
    Purpose: Module which contains general utility functior
"""


Codiumate: Options | Test this function
def main():
    """main method is used to test the functions"""
    print(title("Print Title Test!"))


Codiumate: Options | Test this function
def title(statement):
    """Takes in a string argument
       returns a string with ascii decorations
    """
    # Get the length of the statement string variable
    text_length = len(statement)

    # Create the title string
    # Initialize the result string variable
    result = ""
    result = result + "+--" + "-" * text_length + "--+\n"
    result = result + "|  " + statement + "  |\n"
    result = result + "+--" + "-" * text_length + "--+\n"

    return result


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run testing the module:

```
+--------------------+
|  Print Title Test!  |
+--------------------+
```

How to use the **utils.py** module.

```python
"""
    Name: utils_main.py
    Author:
    Created:
    Purpose: How to use utils.py
"""
# Import utils.py
import utils


Codiumate: Options | Test this function
def main():
    print(utils.title("Print Title Test!"))
    print(utils.title("EOL"))


main()
```

Example run:

```
+--------------------+
|  Print Title Test!  |
+--------------------+
+-------+
|  EOL  |
+-------+
```

Boolean Data Type

A **Boolean** is a data type that can only have **two values**:

- True
- False

In Python, Booleans are often used to:

- **Control program flow** (e.g., with if, while)
- **Check conditions** (e.g., comparisons like x > 0 return a Boolean)
- **Validate input** or **make decisions**

Example:

```
is_valid = True
if is_valid:
    print("Input accepted.")
```

Booleans help programs **decide what to do next** based on simple true/false logic.

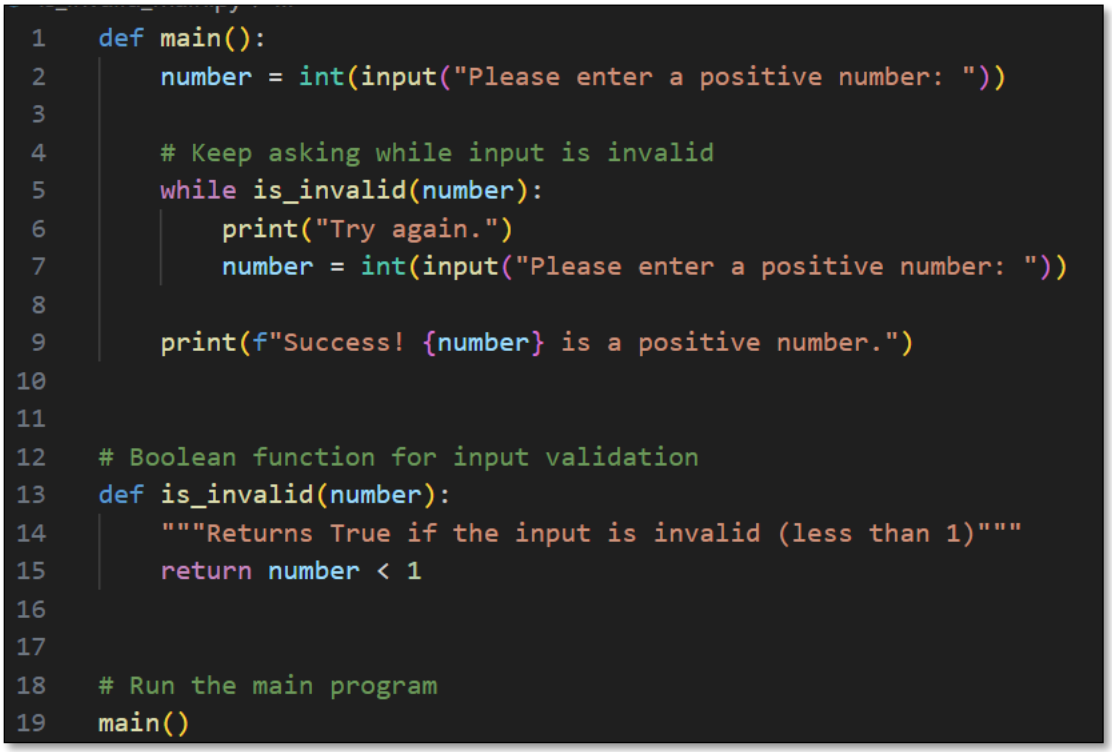
Tutorial 5.4: Input Validation with Boolean

Instead of handling input errors with exceptions, we can **check the input using a Boolean function**.

This approach uses a loop that keeps asking the user until the input is valid.

How It Works

- **main()** gets input from the user.
- It calls the **is_invalid()** function to check the value.
- If the value is invalid, the program prompts again.
- This is called a **validation loop**.



```
def main():
    number = int(input("Please enter a positive number: "))

    # Keep asking while input is invalid
    while is_invalid(number):
        print("Try again.")
        number = int(input("Please enter a positive number: "))

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


# Boolean function for input validation
def is_invalid(number):
    """Returns True if the input is invalid (less than 1)"""
    return number < 1


# Run the main program
main()
```

Why This Works Well

- Keeps validation logic **separate** from input logic
- Makes code **easier to test and reuse**
- Returns True if data is invalid → makes loop logic simple

Example run:

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

Assignment 5.6: Boolean Functions for Validation

Objective

Create a program to use functions that return Boolean values to check conditions.

1. Create a function is_adult(age)

- Input: age (integer)
- Return: True if age is **18 or older**, else False

2. Create a function is_valid_password(password)

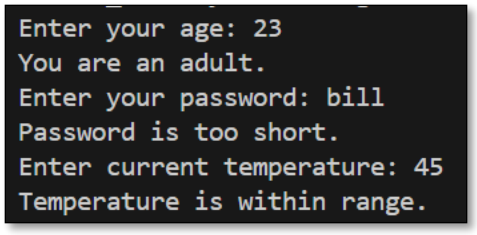
- Input: password (string)
- Return: True if the password length is **at least 8 characters**, else False

3. Create a function is_valid_temperature(temp)

- Input: temp (float)
- Return: True if temp is between **-50.0 and 50.0** inclusive, else False

Main Program

- Ask the user for their age, password, and current temperature
- Use the functions to validate each input
- Print messages based on the results, for example:
    - "You are an adult." or "You are not an adult."
    - "Password is valid." or "Password is too short."
    - "Temperature is within range." or "Temperature out of range."



try and except Statements

Handling Errors Gracefully

Sometimes, code may **cause errors** when run—such as dividing by zero or entering a letter when the program expects a number.

Rather than letting your program crash, you can use a **try/except block** to **catch the error and keep the program running**.

Basic Syntax

```
try:
    # Code that might cause an error
    risky_code()
except:
    # Code to run if there's an error
    handle_the_error()
```

Example 1: Catching Input Errors

```
try:
    num = int(input("Enter a number to divide 100 by: "))
    result = 100 / num
    print(f"Result is {result}")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Example 2: Loop Until Valid Input

```
while True:
    try:
        age = int(input("Enter your age: "))
        break
    except:
        print("Oops! Please enter a valid number.")
print(f"You are {age} years old.")
```

Summary:

- Use try to wrap code that might fail.
- Use except to handle specific or general errors.
- Combine with loops to ensure valid input.

Assignment 5.7: Try Except

Objective

Practice using try and except to handle different types of runtime errors gracefully.

Instructions

- Write a Python program named **try_except.py** that performs the following tasks.
- Use try and except blocks to handle possible errors for each task.

Step 1: Input and Integer Conversion
Ask the user to enter a number. Convert the input to an integer.

- If the input is not a valid integer, print:
  "Error: That is not a valid integer."

Step 2: Division Calculation
Ask the user to enter another number to divide 100 by.

- Handle the case where the user enters zero. Print:
  "Error: Cannot divide by zero."
- Also handle invalid input (not a number).

Example run:

```
Enter a number: 1.0
Error: That is not a valid integer.
Enter a number to divide 100 by: 0
Error: Cannot divide by zero.
```

Tutorial 5.5: Calculations with the Python math Library

These functions are built-in into Python.

`abs()`	`One numerical parameter`	`Returns the positive value of a number`

min()	two or more numerical parameters	Returns the smallest of a range of numbers
max()	two or more numerical parameters	Returns the largest of a range of numbers
round()	one numerical parameter	Rounds mathematically, to a whole number. It has an optional second parameter. The second parameter must be an integer, and if it is provided, the function will round the first parameter to the number of decimals specified by the second parameter.
ceil()	One numerical parameter	Rounds the number up to the nearest integer
floor()	One numerical parameter	Rounds a number down to the nearest integer
pow()	has two numerical parameters	Returns the first to the power of the second
sqrt()	one numerical parameter	Returns the square root of a number
pi		Returns the value of pi

Example program showing each type of calculation. Some require the math library, some are built in to the Python standard library.

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

```python
# Import math library for mathematical constants and functions
import math

# Math functions demonstration
# Initialize test variables
x = -2      # Negative number for testing absolute value
y = 3       # Positive integer
z = 1.27    # Decimal number for rounding functions

# Display the mathematical constant pi
print(math.pi)

# Absolute value function - returns positive value of x
print(abs(x))

# Find minimum value among the three numbers
print(min(x, y, z))

# Find maximum value among the three numbers
print(max(x, y, z))

# Round z to 1 decimal place
print(round(z, 1))

# Ceiling function - rounds up to nearest integer
print(math.ceil(z))

# Floor function - rounds down to nearest integer
print(math.floor(z))

# Power function - raises x to the power of y (x^y)
print(pow(x, y))

# Square root function - returns square root of y
print(math.sqrt(y))
```

Example program run:

```
3.141592653589793
2
-2
3
1.3
2
1
-8
1.7320508075688772
```

Tutorial 5.6: Random Numbers in Python

Python's random module creates **pseudorandom numbers** — numbers that seem random but are generated by a deterministic process.

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

random.random()

Returns a random float **≥ 0.0 and < 1.0**

```python
# Import the entire random module
import random

# Generate 10 random floating-point numbers between 0.0 and 1.0
for _ in range(10):
    print(random.random())
```

random.randint(low, high)

Returns a random **integer** between low and high **inclusive**

```python
# Import only the randint function from random module
from random import randint

# Generate 10 random integers between 1 and 3 (inclusive)
for _ in range(10):
    print(randint(1, 3))  # Random integer: 1, 2, or 3
```

random.choice(sequence)

Returns a random element from a list or other sequence

```python
# Import random module again for choice function
import random

# Create a list to choose random elements from
my_list = [1, 2, 3]

# Randomly select and print 10 elements from the list
for _ in range(10):
    print(random.choice(my_list))
```

Why use pseudorandom?

- True randomness is hard for computers
- Pseudorandom numbers are predictable if you know the seed
- Good enough for games, simulations, and many applications

time.sleep() in Python

The **time.sleep()** function pauses your program for a specified number of seconds.

```
import time

print("Wait for 2 seconds...")
time.sleep(2)  # Pause for 2 seconds
print("Done waiting!")
```

Assignment 5.7: Random Numbers and Delays

The following tutorial demonstrates random integers in a for loop.

Objective

Practice generating random numbers and using delays between outputs.

1. Import the required modules:

- from random import randint
- from time

2. Write a program that:

- Prints a message like: "Starting random number generator..."

- Uses a loop to generate and print 5 random integers between 1 and 100 (inclusive).
- After printing each number, the program waits for a random delay between 1 and 3 seconds before generating the next number.
- After the loop ends, print "Done!"

Example run:

```
Starting random number generator...
8
55
14
87
68
Done!
```

Glossary

algorithm A general process for solving a category of problems.

argument A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

body The sequence of statements inside a function definition.

composition Using an expression as part of a larger expression, or a statement as part of a larger statement.

deterministic Pertaining to a program that does the same thing each time it runs, given the same inputs.

dot notation The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

flow of execution The order in which statements are executed during a program run.

fruitful function A function that returns a value.

function A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function call A statement that executes a function. It consists of the function name followed by an argument list.

function definition A statement that creates a new function, specifying its name, parameters, and the statements it executes.

function object A value created by a function definition. The name of the function is a variable that refers to a function object.

header The first line of a function definition.

import statement A statement that reads a module file and creates a module object.

module object A value created by an import statement that provides access to the data and code defined in a module.

parameter A name used inside a function to refer to the value passed as an argument.

pseudorandom Pertaining to a sequence of numbers that appear to be random but are generated by a deterministic program.

return value The result of a function. If a function call is used as an expression, the return value is the value of the expression.

void function A function that does not return a value.

Assignment Submission

1. Attach all tutorials and assignments.
2. Attach screenshots showing the successful operation of each program.
3. Submit in Blackboard.