MODULE 2: Functions

What is a Module?

A **module** is simply a Python file with a **.py** extension that a programmer can import inside another Python program.

The name of the Python file becomes the name of the module.

The module consists of definitions and implementation of -

1. **Classes**
2. **Variables**
3. **Functions**

That can be utilized within another program.

Benefits of modules

1. **Reusability:**Working with modules makes the code reusable.
2. **Simplicity:**The module targets a small proportion of the problem rather than aiming at the complete problem.
3. **Scoping:**A Module defines a distinct namespace that supports avoiding collisions between identifiers.

What is a Function?

A **function** is a block of organized and reusable code that we can perform a single, associated activity. The functions are classified into different types -

1. User-defined functions
2. Built-in functions
3. Lambda functions
4. Recursive functions

User-defined functions:
Functions that are defined by us in order to perform a particular activity are termed User-defined functions.

Benefits of User-defined functions

1. We can use the User-defined functions to decompose a large program into small segments, which makes the program easy to understand, maintain and debug.
2. Suppose repeated code occurs in a program. We can use the function to include those codes and execute when required by calling that function.
3. Programmers working on large project can divide the workload by making different functions.

Built-in functions:

Python has different functions that are readily available for use. These functions are known as Built-in functions.

List of Built-in functions

abs(), delattr(), hash(), memoryview(), set(), all(), dict(), help(), min(), setattr(), any(), dir(), hex(), next(), slice(), ascii(), divmod(), id(), object(), sorted(), bin(), enumerate(), input(), oct(), staticmethod(), bool(), eval(), **int**(), open(), str(), breakpoint(), exec(), isinstance(), ord(), sum(), bytearray(), filter(), issubclass(), pow(),**super**(), bytes(), **float**(), iter(), print(), tuple(), callable(), format(), len(), property(), type(), chr(), frozenset(), list(), range(), vars(), classmethod(), getattr(), locals(), repr(), zip(), compile(), globals(), map(), reversed(), **__import__**(), complex(), hasattr(), max(), round()

Difference between module and function in Python

Python is a programming language that is considered to be progressive and known for its optimization capabilities. Python skims down redundant characteristics of programming and makes the tools rich in utilization. In the following tutorial, we will discuss the difference between module and function in the Python programming language.

Differences between lambda function and non lambda function

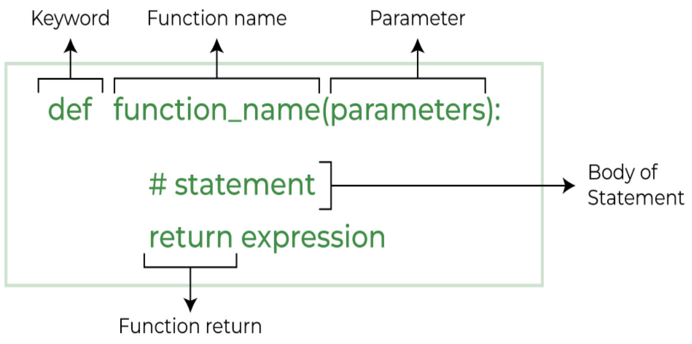
lambda function	**def function**
Keyword lambda is the name of the function	Keyword def is used followed by the function name
Supports single-line statements	Supports multiple lines inside a function

Creating a Function

In Python a function is defined using the def keyword:

Python Function Declaration

The syntax to define a function is:



Example

```
def my_function():
  print("Hello from a function")
```

Calling a Function

Basic syntax to calling a function is:

```
function_name()
```

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")

my_function()
```

Arguments: Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
  print(fname + " Refsnes")
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:
A parameter is the variable listed inside the parentheses in the function definition.
An argument is the value that is sent to the function when it is called.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example
This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil", "Refsnes")
```

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Arbitrary Arguments are often shortened to **args* in Python documentations.)

Return Values

To let a function return a value, use the return statement:

Example

```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

The pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example

```
def my_function():
    pass  # Function body will be added later
if x > 5:
    pass  # Action will be taken later if x is greater than 5
else:
    print("x is not greater than 5")
```

Examples:

Function with Parameters and Return Value

```
def add(a, b):
    return a + b
result = add(5, 3)
print("Sum:", result)  # Output: Sum: 8
```

Function Without Parameters

```
def greet():
    print("Hello, Welcome to Python!")
greet()  # Output: Hello, Welcome to Python!
```

Function with Default Parameters

```
def greet(name="Guest"):
    print("Hello,", name)
greet("Alice")  # Output: Hello, Alice
greet()         # Output: Hello, Guest
```

Function with Keyword Arguments

```
def person_details(name, age):
    print("Name:", name)
    print("Age:", age)
person_details(age=25, name="John")
```

Function Returning Multiple Values

```
def arithmetic_operations(a, b):

    return a + b, a - b, a * b, a / b

add, sub, mul, div = arithmetic_operations(10, 5)

print("Addition:", add)       # Output: Addition: 15

print("Subtraction:", sub)    # Output: Subtraction: 5

print("Multiplication:", mul) # Output: Multiplication: 50

print("Division:", div)       # Output: Division: 2.0
```

Lambda (Anonymous) Function

```
square = lambda x: x * x

print(square(5))  # Output: 25
```

Standard Mathematical Functions

The standard math module provides much of the functionality of a scientific calculator. Table 6.1 lists only a few of the available functions

`math` Module	
`sqrt`	Computes the square root of a number: `sqrt`$(x) = \sqrt{x}$
`exp`	Computes e raised a power: `exp`$(x) = e^x$
`log`	Computes the natural logarithm of a number: `log`$(x) = \log_e x = \ln x$
`log10`	Computes the common logarithm of a number: `log`$(x) = \log_{10} x$
`cos`	Computes the cosine of a value specified in radians: `cos`$(x) = \cos x$; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent, hyperbolic cosine, hyper-bolic sine, and hyperbolic tangent
`pow`	Raises one number to a power of another: `pow`$(x, y) = x^y$
`degrees`	Converts a value in radians to degrees: `degrees`$(x) = \frac{\pi}{180} x$
`radians`	Converts a value in degrees to radians: `radians`$(x) = \frac{180}{\pi} x$

Importing the math Module:

Most mathematical functions are available in the `math` module. It must be imported before use.

import math

i) Absolute Value (abs())

Returns the absolute (positive) value of a number.

print(abs(-10)) # Output: 10

ii) Power (pow())

Raises a number to a specified power.

print(pow(2, 3)) # Output: 8 (2^3)

iii) Square Root (math.sqrt())

Returns the square root of a number.

print(math.sqrt(25)) # Output: 5.0

iv) Natural Logarithm (math.log())

Computes the natural logarithm (base e).

print (math.log(10)) # Output: 2.302585092994046 (ln(10))

v) Logarithm Base 10 (math.log10())

Computes logarithm to the base 10.

print(math.log10(1000))   # Output: 3.0

vi) Cosine (math.cos())

Computes the cosine of an angle in radians.

print(math.cos(math.pi)) # Output: -1.0 (cos(180°))

vii) Convert Degrees to Radians (math.radians())

Converts an angle from degrees to radians.

print(math.degrees(math.pi)) # Output: 180.0

viii) Convert Radians to Degrees (math.degrees())

print(math.degrees(math.pi)) # Output: 180.0

Random Functions

The random module has a set of methods:

Python provides the random module, which includes functions to generate **random numbers, shuffle data, and make random selections**. This module is widely used in games, simulations, and cryptographic applications.

Importing the `random` Module

Before using random functions, you need to import the `random` module

import random

Generating Random Numbers

i) random.random() - Generate a Random Float Between 0 and 1

- Returns a floating-point number in the range **0.0 ≤ x < 1.0**.

print(random.random()) # Example Output: 0.7532

ii) random.randint(a, b) - Generate a Random Integer in a Range

- Returns a random integer in the range **[a, b]** (both inclusive).

  print(random.randint(1, 100)) # Example Output: 42

v) random.randrange(start, stop, step) - Generate a Random Integer with Steps

- Returns a random integer from **start to stop - 1** with an optional step.

  print(random.randrange(1, 10, 2)) # Example Output: 3 (1, 3, 5, 7, 9)

Selecting Random Elements

i) random.choice(sequence) - Select a Random Element

- Returns a random item from a sequence (list, tuple, or string).

  fruits = ["Apple", "Banana", "Cherry", "Mango"]

  print (random.choice(fruits)) # Example Output: Cherry

Shuffling Data

i) random.shuffle(sequence) - Shuffle a List

- Randomly rearranges the elements of a list **in place**.

```
numbers = [1, 2, 3, 4, 5]

random.shuffle(numbers)

print(numbers)  # Example Output: [3, 5, 1, 4, 2]
```

Time Functions

Python provides the **time** module to handle various time-related functions such as **retrieving the current time, formatting dates, introducing delays, and measuring execution time**.

Importing the `time` Module

Before using time functions, import the time module:

```
import time
```

Getting the Current Time

i) time.time() - Get Current Time in Seconds Since Epoch

- Returns the current time in **seconds** since **January 1, 1970 (Epoch time).**
- The returned value is a floating-point number.

```
print(time.time())  # Example Output: 1710938384.763847
```

i) time.ctime([secs]) - Convert Time to Readable Format

- Converts **epoch time** (seconds) into a human-readable string.
- If no argument is given, it returns the **current date and time**.

```
print(time.ctime())  # Example Output: 'Fri Mar 21 14:33:04 2025'

print(time.ctime(1710938384))  # Convert a given timestamp
```

System specific functions

Python provides **system-specific functions** through the sys module, which allows interaction with the interpreter, retrieving system information, and handling runtime exceptions.

The sys module provides various functions related to the Python interpreter and system environment.

1.1 Importing the `sys` Module

```
import sys
```

1.2 Getting Python Version

- Retrieves the version of Python currently running.

```
print(sys.version)  # Example Output: 3.10.11 (default, Mar 21 2025)

print(sys.version_info)  # Output: sys.version_info(major=3, minor=10, micro=11,
releaselevel='final', serial=0)
```

1.3 Exiting a Python Program

- `sys.exit([status])` #stops the execution of the program.

```
sys.exit("Exiting the program!")  # Stops execution
```

sys.exit("Exiting the program!") # Stops execution

1.4 Getting System Path (`sys.path`)

- Returns a list of directories where Python searches for modules.

```
print(sys.path)
```

1.5 Getting the Platform (OS Name)

- Identifies the operating system.

```
print(sys.platform)  # Example Output: 'win32' (Windows), 'linux', 'darwin' (Mac)
```

eval() and exec() functions

In Python, eval() and exec() are built-in functions used to execute code dynamically. However, they serve different purposes and have distinct behaviors.

eval()

- Used for evaluating mathematical expressions or Python expressions stored as strings.
- Cannot execute statements like loops, function definitions, or class definitions

Example1:
```
expression = "5 + 10 * 2"
result = eval(expression)  # Evaluates the expression and returns the
resultprint(result)  # Output: 25
```

Example2:
```
x = 10
print(eval("x * 2"))  # Output: 20
```

exec()

 Used when dynamically executing multiple lines of Python code.

 can execute assignments, loops, function/class definitions.

Example1:
```
code = """
def greet():
        print("Hello, World!")
greet()
"""
exec(code)  # Output: Hello, World!
```

Example2:
```
x = 5
exec("x = x + 10")
print(x)  # Output: 15
```

Differences between eval() and exec()

Feature	**eval()**	**exec()**
Purpose	Evaluates a single expression	Executes arbitrary code
Return value	Returns the result of the expression	Returns None
Input	Single expression	Code block (statements, functions, etc.)
Use cases	Evaluating simple expressions	Executing dynamic code, scripts

Importing a module using import statement

- All Python programs can call a basic set of functions called built-in functions, includingthe print(), input(), and len() functions.
- Python also comes with a set of modules called the standard library.
- Each module is a Python program that contains a related group of functions that can be embedded inyour programs.
- For example, the math module has mathematics-related functions, the random module has randomnumber–related functions, and so on.
- Before we can use the functions in a module, we must import the module with an import statement. Incode, an import statement consists of the following:
  1. The import keyword
  2. The name of the module
  3. Optionally, more module names, as long as they are separated by commas
- Once we import a module, we can use all the functions of that module.
- Example with output:

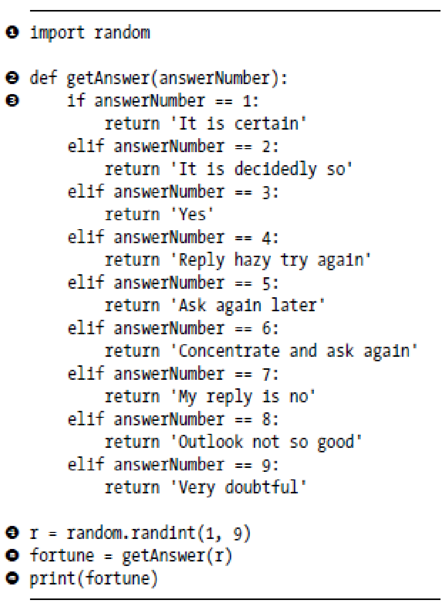
```
import random
for i in range(5):
    print(random.randint(1, 10))
```

```
4
1
8
4
1
```

Return Values and Return Statements

- The value that a function call evaluates to is called the return value of the function.
- Ex: len('Hello') → Return values is 5
- When creating a function using the def statement, we can specify what the return value should be witha return statement.
- A return statement consists of the following:
  1. The return keywords.
  2. The value or expression that the function should return.
- When an expression is used with a return statement, the return value is what this expression evaluates to.

- ➢ For example, the following program defines a function that returns a different string depending on whatnumber it is passed as an argument.



```
❶ import random

❷ def getAnswer(answerNumber):
❸     if answerNumber == 1:
          return 'It is certain'
      elif answerNumber == 2:
          return 'It is decidedly so'
      elif answerNumber == 3:
          return 'Yes'
      elif answerNumber == 4:
          return 'Reply hazy try again'
      elif answerNumber == 5:
          return 'Ask again later'
      elif answerNumber == 6:
          return 'Concentrate and ask again'
      elif answerNumber == 7:
          return 'My reply is no'
      elif answerNumber == 8:
          return 'Outlook not so good'
      elif answerNumber == 9:
          return 'Very doubtful'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

Global Variables

- A **global variable** is a variable declared outside of a function, making it accessible throughout the program.
- You can access a global variable inside a function, but to modify it, you need to use the global keyword.

Example:

```
  x = 10  # Global variable

def modify_global():
       global x  # Using the global keyword to modify x
       x = 20
       print("Inside function:", x)

modify_global()
print("Outside function:", x)
```

Local Variables

- Defined inside a function and only accessible within that function.
- They do not affect variables outside the function.
- Created when the function is called and destroyed when the function exits.

```
def my_function():
    x = 10  # Local variable
    print("Inside function:", x)

my_function()
# print(x)  # This will cause an error because x is not defined outside the function.
```

Default Parameters

• Python allows function parameters to have default values, which are used if no argument is provided.

 Default parameters must come **after** non-default parameters.

Example :

```
def greet(name="Guest"):
    print("Hello,", name)

greet("Alice")  # Output: Hello, Alice
greet()         # Output: Hello, Guest
```

Importing a module using import statement

All Python programs can call a basic set of functions called built-in functions, including the print(), input(), and len() functions.

➢ Python also comes with a set of modules called the standard library.

➢ Each module is a Python program that contains a related group of functions that can be embedded in your programs.

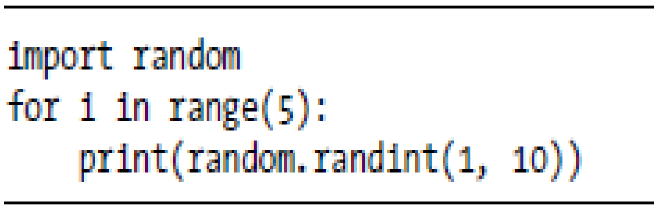
➢ For example, the math module has mathematics-related functions, the random module has random number–related functions, and so on.

➢ before we can use the functions in a module, we must import the module with an import statement. In code, an import statement consists of the following: The import keyword

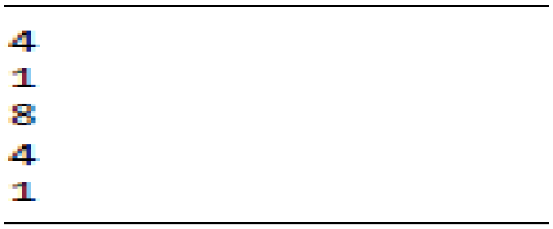
The name of the module

Optionally, more module names, as long as they are separated by commas

➢ Once we import a module, we can use all the functions of that module.
➢ Example with output



```
import random
for i in range(5):
    print(random.randint(1, 10))
```



```
4
1
8
4
1
```

Anonymous Function using Lambda Expression

- **Lambda functions** are anonymous (unnamed) functions in Python, defined using the `lambda` keyword.
- They can have any number of arguments but only one expression.
- Typically used for short, simple functions that are used once or as arguments to higher-order functions.

Syntax:

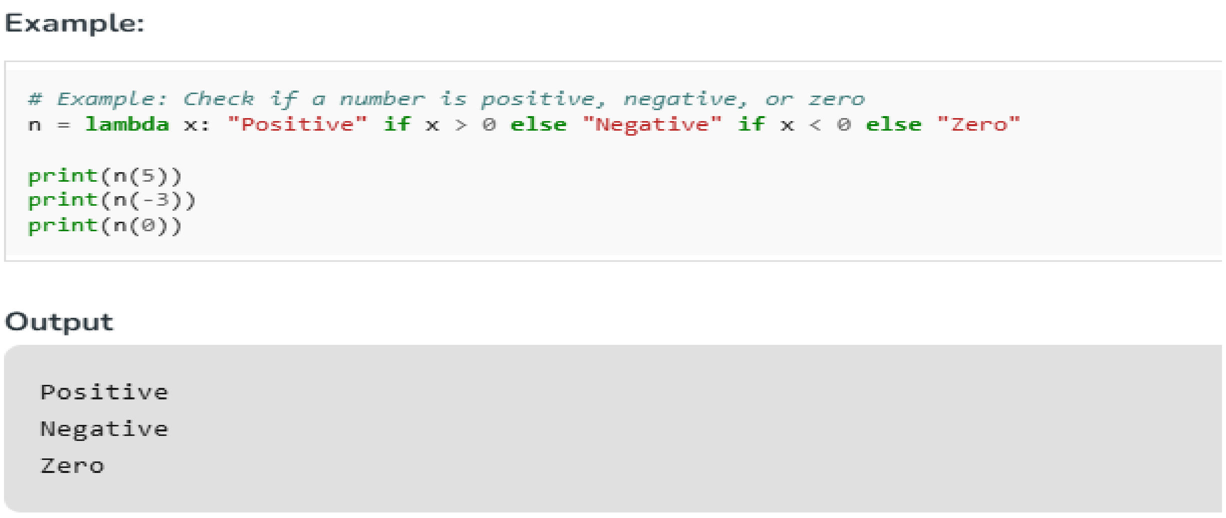
lambda arguments: expression

Example1:

```
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

Example2:

```
s1 = 'SNPSU'
s2 = lambda func: func.upper()
print(s2(s1))
Output:  SNPSU
```



Example:

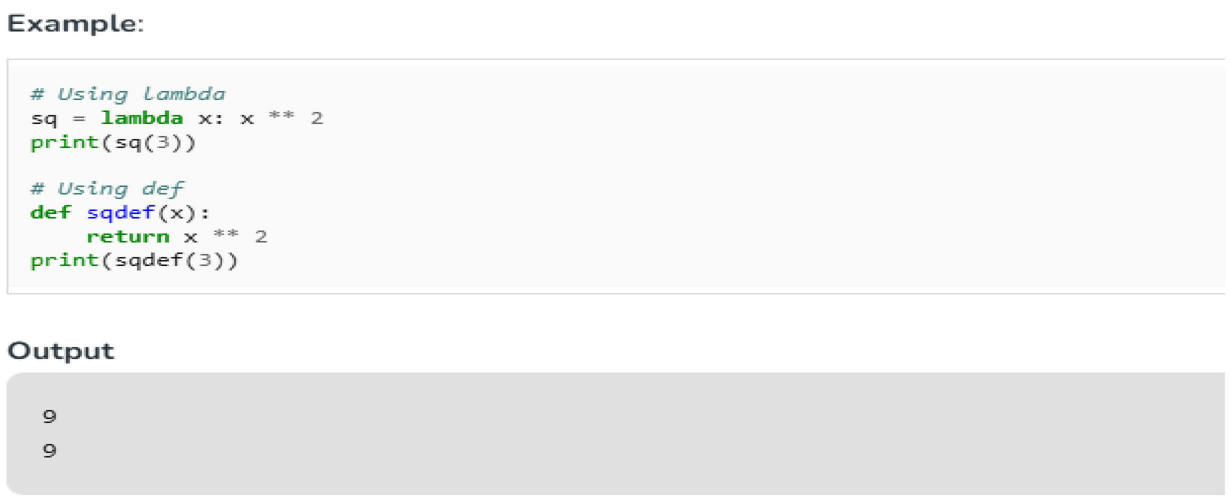
```
# Example: Check if a number is positive, negative, or zero
n = lambda x: "Positive" if x > 0 else "Negative" if x < 0 else "Zero"

print(n(5))
print(n(-3))
print(n(0))
```

Output

```
Positive
Negative
Zero
```

Example for def keyword



Example:

```python
# Using lambda
sq = lambda x: x ** 2
print(sq(3))

# Using def
def sqdef(x):
    return x ** 2
print(sqdef(3))
```

Output

```
9
9
```

Generators

- **Generators** are special functions that return an **iterator** using the `yield` keyword instead of `return`.
- They are used for **lazy evaluation**, meaning they generate values **on the fly** without storing them in memory.
- Once a generator function is called, it does not execute immediately but returns a generator object.

Example:

```python
def count_up_to(n):
    count = 1
    while count <= n:
        yield count  # Generates value instead of returning
        count += 1
counter = count_up_to(5)
print(next(counter))  # Output: 1
print(next(counter))  # Output: 2
for num in counter:
    print(num)  # Output: 3, 4, 5
```

Local Function Definitions (Nested Functions)

- A function can be defined **inside another function**, known as a **local function** or **nested function**.
- These are useful when a function is only needed within another function.

Example :

```
def outer_function(name):

    def inner_function():

        return f"Hello, {name}!"  # Accesses variable from outer function

    return inner_function()

print(outer_function("Alice"))  # Output: Hello, Alice!
```

Recursive function:

 Recursion is a process where a function **calls itself** to solve a problem.

 A recursive function **breaks down a complex problem** into smaller, simpler problems.

 It continues calling itself until it reaches a **condition that stops the recursion.**

- Recursion is widely used for tasks that can be divided into identical subtasks.

Example:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))

Output: 120
```

1. Programs to implement math functions such as sqrt, power.

```
import math
# sqrt computes the square root
square_root = math.sqrt(4)
print("Square Root of 4 is",square_root)
# pow() comptes the power
power = pow(2, 3)
print("2 to the power 3 is",power)
```

2. Write a Python function to find the maximum of three numbers

```
def max_of_two(x, y):
    if x > y:
        return x
    return y
def max_of_three(x, y, z):
    return max_of_two(x, max_of_two(y, z))
print(max_of_three(3, 6, -5))
```

3. Write a Python function to sum all the numbers in a given list.

```
def sum(numbers):
        total = 0
    for x in numbers:
        total += x
    return total
print(sum((8, 2, 3, 0, 7)))
```