



# PYTHON PROGRAMMING LECTURE FIVE

LECTURE FIVE

Python Programming  
Lecture Five

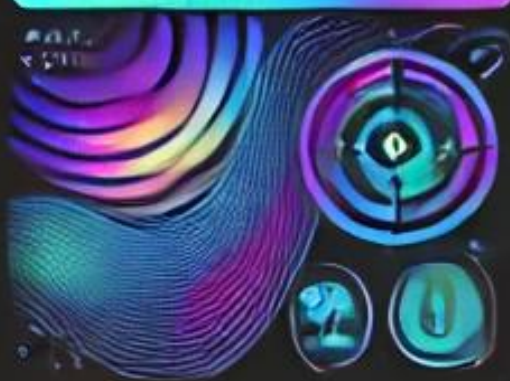
# LECTURE FIVE FUNCTIONS

LECTURE FIVE



FUNCTIONS  
Python Programming  
Lecture Five

## EXERCISES



## OPTIONALLY

Python Programming  
Lecture Five



Python Programming  
Lecture Five

Python Programming  
Lecture Five



## EXERCISES



## OPTIONALLY

- 1. (function)
- 2. (function (parameters) )
- 3. (function)
- 4. (function (parameters) )
- 5. (function)
- 6. (function (parameters) )
- 7. (function (parameters) )
- 8. (function (parameters) )
- 9. (function (parameters) )
- 10. (function (parameters) )

ZIED O. AHMED

## EXERCISES





# LECTURE FIVE: FUNCTIONS

## 5.1 Introduction to functions

Most programs perform tasks that are large enough to be broken down into several subtasks. For this reason, programmers usually break down their programs into small manageable pieces known as functions. A *function* is a group of statements that exist within a program for the purpose of performing a specific task.

A function is a group of statements that exist within a program for the purpose of performing a specific task.

Instead of writing a large program as one long sequence of statements, it can be written as several small functions, each one performing a specific part of the task. These small functions can then be executed in the desired order to perform the overall task.



This approach is sometimes called *divide and conquer* because a large task is divided into several smaller tasks that are easily performed. Figure below illustrates this idea by comparing two programs: one that uses a long complex sequence of statements to perform a task, and another that divides a task into smaller tasks, each of which is performed by a separate function.

This program is one long, complex sequence of statements.

statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement  
statement

In this program the task has been divided into smaller tasks, each of which is performed by a separate function.

```
def function1():  
    statement  
    statement  
    statement
```

function

```
def function2():  
    statement  
    statement  
    statement
```

function

```
def function3():  
    statement  
    statement  
    statement
```

function

```
def function4():  
    statement  
    statement  
    statement
```

function

## 5.2 Benefits of Modularizing a Program with Functions

A program benefits in the following ways when it is broken down into functions:

- Simpler Code
- Code Reuse
- Better Testing
- Faster Development
- Easier Facilitation of Teamwork

### 5.3 Defining and Calling a Void function

#### Function Names

Before we discuss the process of creating and using functions, we should mention a few things about function names. Just as you name the variables that you use in a program, you also name the functions. A function's name should be descriptive enough so anyone reading your code can reasonably guess what the function does.

The code for a function is known as a function definition. To execute the function, you write a statement that calls it.



#### Defining and Calling a function

To create a function, you write its *definition*. Here is the general format of a function definition in Python:

```
def function_name():  
    statement  
    statement  
    etc.
```

The first line is known as the *function header*. It marks the beginning of the function definition. The function header begins with the key word `def`, followed by the name of the function, followed by a set of parentheses, followed by a colon.

Let's look at an example of a function. Keep in mind that this is not a complete program.

```
def message():  
    print('I am Arthur,')  
    print('King of the Britons.')
```

This code defines a function named `message`. The `message` function contains a block with two statements. Executing the function will cause these statements to execute.

## Calling a function

A function definition specifies what a function does, but it does not cause the function to execute. To execute a function, you must *call* it. This is how we would call the `message` function:

```
message()
```

When a function is called, the interpreter jumps to that function and executes the statements in its block. Then, when the end of the block is reached, the interpreter jumps back to the part of the program that called the function, and the program resumes execution at that point. When this happens, we say that the function *returns*. To fully demonstrate how function calling works, we will look at following Program.

### Program 5-1 (function\_demo.py)

```
1 # This program demonstrates a function.
2 # First, we define a function named message.
3 def message():
4     print('I am Arthur,')
5     print('King of the Britons.')
6
7 # Call the message function.
8 message()
```

### Program Output

```
I am Arthur,
King of the Britons.
```

Let's step through this program and examine what happens when it runs.



First, the interpreter ignores the comments that appear in lines 1 and 2. Then, it reads the `def` statement in line 3. This causes a function named `message` to be created in memory, containing the block of statements in lines 4 and 5. (Remember, a function definition creates a function, but it does not cause the function to execute.)

Next, the interpreter encounters the comment in line 7, which is ignored. Then it executes the statement in line 8, which is a function call. This causes the `message` function to execute, which prints the two lines of output.

---

```
# This program demonstrates a function.  
# First, we define a function named message.  
def message():  
    print('I an Arthur,')  
    print('King of the Britons.')
```

```
# Call the message function.  
message()
```

These statements cause the message function to be created.

This statement calls the message function, causing it to execute.

Above Program has only one function, but it is possible to define many functions in a program.

In fact, it is common for a program to have a `main` function that is called when the program starts. The `main` function then calls other functions in the program as they are needed. It is often said that the `main` function contains a program's *mainline logic*, which is the overall logic of the program. Program below shows an example of a program with two functions: `main` and `message`

**Program 5-2** (two\_functions.py)

```

1  # This program has two functions. First we
2  # define the main function.
3  def main():
4      print('I have a message for you.')
5      message()
6      print('Goodbye!')
7
8  # Next we define the message function.
9  def message():
10     print('I am Arthur,')
11     print('King of the Britons.')
12
13 # Call the main function.
14 main()

```

**Program Output**

```

I have a message for you.
I am Arthur,
King of the Britons.
Goodbye!

```

The definition of the `main` function appears in lines 3 through 6, and the definition of the `message` function appears in lines 9 through 11. The statement in line 14 calls the `main` function, as shown in Figure below.

**Calling the main function**

The interpreter jumps to the `main` function and begins executing the statements in its block.

```

# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()

```

The first statement in the main function calls the print function in line 4. It displays the string 'I have a message for you'. Then, the statement in line 5 calls the message function. This causes the interpreter to jump to the message function, as shown in Figure below.

#### 4 Calling the message function

The interpreter jumps to the message function and begins executing the statements in its block.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

#### 5 The message function returns

When the message function ends, the interpreter jumps back to the part of the program that called it and resumes execution from that point.

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```



## The main function returns

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

When the main function ends, the interpreter jumps back to the part of the program that called it. There are no more statements, so the program ends.

When a program calls a function, programmers commonly say that the control of the program transfers to that function. This simply means that the function takes control of the program's execution



## 5.4 local Variables

Anytime you assign a value to a variable inside a function, you create a *local variable*. A local variable belongs to the function in which it is created, and only statements inside that function can access the variable.

An error will occur if a statement in one function tries to access a local variable that belongs to another function. For example, look at Program below.

**Program 5-4** (bad\_local.py)

```

1 # Definition of the main function.
2 def main():
3     get_name()
4     print('Hello', name)      # This causes an error!
5
6 # Definition of the get_name function.
7 def get_name():
8     name = input('Enter your name: ')
9
10 # Call the main function.
11 main()

```

A local variable is created inside a function and cannot be accessed by statements that are outside the function. Different functions can have local variables with the same names because the functions cannot see each other's local variables.

**Scope and local Variables**

A variable's *scope* is the part of a program in which the variable may be accessed. A variable is visible only to statements in the variable's scope.

A local variable's scope is the function in which the variable is created. As you saw demonstrated in above Program, no statement outside the function may access the variable.

In addition, a local variable cannot be accessed by code that appears inside the function at a point before the variable has been created.

```

def bad_function():
    print('The value is', val)    # This will cause an error!
    val = 99

```

For example, look at Program below. In addition to the `main` function, this program has two other functions: `texas` and `california`. These two functions each have a local variable named `birds`.

**Program 5-5** (birds.py)

```

1  # This program demonstrates two functions that
2  # have local variables with the same name.
3
4  def main():
5      # Call the texas function.
6      texas()
7      # Call the california function.
8      california()
9
10 # Definition of the texas function. It creates
11 # a local variable named birds.
12 def texas():
13     birds = 5000
14     print('texas has', birds, 'birds.')
15
16 # Definition of the california function. It also
17 # creates a local variable named birds.
18 def california():
19     birds = 8000
20     print('california has', birds, 'birds.')
21
22 # Call the main function.
23 main()

```

**Program Output**

```

texas has 5000 birds.
california has 8000 birds.

```

## 5.5 Passing Arguments to Functions

Sometimes it is useful not only to call a function, but also to send one or more pieces of data into the function. Pieces of data that are sent into a function are known as *arguments*. The function can use its arguments in calculations or other operations.

If you want a function to receive arguments when it is called, you must equip the function with one or more parameter variables. A *parameter variable*, often simply called a *parameter*, is a special variable that is assigned the value of an

argument when a function is called. Here is an example of a function that has a parameter variable:

```
def show_double(number):  
    result = number * 2  
    print(result)
```

This function's name is `show_double`. Its purpose is to accept a number as an argument and display the value of that number doubled. Look at the function header and notice the word `number` that appear inside the parentheses. This is the name of a parameter variable. This variable will be assigned the value of an argument when the function is called. Program below demonstrates the function in a complete program.

**Program 5-6** (pass\_arg.py)

```
1  # This program demonstrates an argument being  
2  # passed to a function.  
3  
4  def main():  
5      value = 5  
6      show_double(value)  
7  
8  # The show_double function accepts an argument  
9  # and displays double its value.  
10 def show_double(number):  
11     result = number * 2  
12     print(result)  
13  
14 # Call the main function.  
15 main()
```

**Program Output**

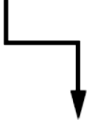
10



### The value variable is passed as an argument

```
def main():
    value = 5
    show_double(value)

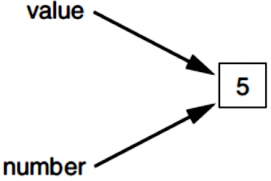
def show_double(number):
    result = number * 2
    print(result)
```



### The value variable and the number parameter reference the same value

```
def main():
    value = 5
    show_double(value)

def show_double(number):
    result = number * 2
    print(result)
```



## Passing Multiple Arguments

Often, it's useful to write functions that can accept multiple arguments. Program below shows a function named `show_sum`, that accepts two arguments. The function adds the two arguments and displays their sum.

#### Program 5-8 (multiple\_args.py)

```
1 # This program demonstrates a function that accepts
2 # two arguments.
3
4 def main():
5     print('The sum of 12 and 45 is')
```

**Program 5-8** (continued)

```

6      show_sum(12, 45)
7
8      # The show_sum function accepts two arguments
9      # and displays their sum.
10     def show_sum(num1, num2):
11         result = num1 + num2
12         print(result)
13
14     # Call the main function.
15     main()

```

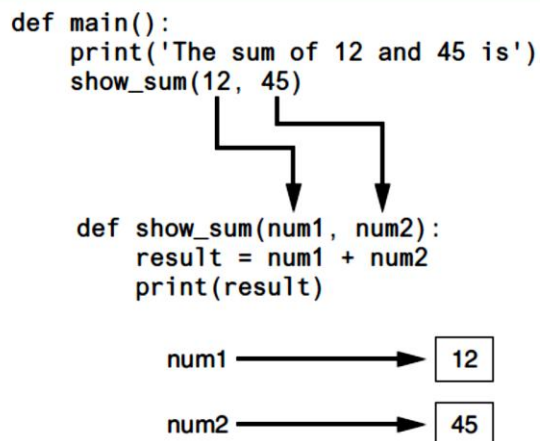
**Program Output**

```

The sum of 12 and 45 is
57

```

Notice two parameter variable names, `num1` and `num2`, appear inside the parentheses in the `show_sum` function header. This is often referred to as a parameter list. Also notice a comma separates the variable names.

**Two arguments passed to two parameters**

This program passes two strings as arguments to a function.

**Program 5-9** (string\_args.py)

```

1  # This program demonstrates passing two string
2  # arguments to a function.
3
4  def main():
5      first_name = input('Enter your first name: ')
6      last_name = input('Enter your last name: ')
7      print('Your name reversed is')
8      reverse_name(first_name, last_name)
9
10 def reverse_name(first, last):
11     print(last, first)
12
13 # Call the main function.
14 main()

```

**Program Output** (with input shown in bold)

```

Enter your first name: Matt 
Enter your last name: Hoyle 
Your name reversed is
Hoyle Matt

```

**Making Changes to Parameters**

When an argument is passed to a function in Python, the function parameter variable will reference the argument's value. However, any changes that are made to the parameter variable will not affect the argument. To demonstrate this, look at Program 5-10.

**Program 5-10** (change\_me.py)

```

1  # This program demonstrates what happens when you
2  # change the value of a parameter.
3

```

**Program 5-10** (continued)

```

4 def main():
5     value = 99
6     print('The value is', value)
7     change_me(value)
8     print('Back in main the value is', value)
9
10 def change_me(arg):
11     print('I am changing the value.')
12     arg = 0
13     print('Now the value is', arg)
14
15 # Call the main function.
16 main()

```

**Program Output**

```

The value is 99
I am changing the value.
Now the value is 0
Back in main the value is 99

```

**Keyword Arguments**

Most programming languages match function arguments and parameters this way. In addition to this conventional form of argument passing, the Python language allows you to write an argument in the following format, to specify which parameter variable the argument should be passed to:

*parameter\_name=value*

In this format, *parameter\_name* is the name of a parameter variable, and value is the value being passed to that parameter. An argument that is written in accordance with this syntax is known as a *keyword argument*.



Program below demonstrates keyword arguments. This program uses a function named `show_interest` that displays the amount of simple interest earned by a bank account for a number of periods. The function accepts the arguments `principal` (for the account principal), `rate` (for the interest rate per period), and `periods` (for the number of periods). When the function is called in line 7, the arguments are passed as keyword arguments.

#### **Program 5-11** (keyword\_args.py)

```

1  # This program demonstrates keyword arguments.
2
3  def main():
4      # Show the amount of simple interest, using 0.01 as
5      # interest rate per period, 10 as the number of periods,
6      # and $10,000 as the principal.
7      show_interest(rate=0.01, periods=10, principal=10000.0)
8
9  # The show_interest function displays the amount of
10 # simple interest for a given principal, interest rate
11 # per period, and number of periods.
12
13 def show_interest(principal, rate, periods):
14     interest = principal * rate * periods
15     print('The simple interest will be $',
16           format(interest, ',.2f'),
17           sep='')
18
19 # Call the main function.
20 main()

```

#### **Program Output**

The simple interest will be \$1000.00.

Notice in line 7 the order of the keyword arguments does not match the order of the parameters in the function header in line 13. Because a keyword argument specifies which parameter the argument should be passed into, its position in the function call does not matter.

Program below shows another example. This program uses keyword arguments to call the `reverse_name` function.

**Program 5-12** (`keyword_string_args.py`)

```
1 # This program demonstrates passing two strings as
2 # keyword arguments to a function.
3
4 def main():
5     first_name = input('Enter your first name: ')
6     last_name = input('Enter your last name: ')
7     print('Your name reversed is')
8     reverse_name(last=last_name, first=first_name)
9
10 def reverse_name(first, last):
11     print(last, first)
12
13 # Call the main function.
14 main()
```

**Program Output** (with input shown in bold)

```
Enter your first name: Matt 
Enter your last name: Hoyle 
Your name reversed is
Hoyle Matt
```

**Mixing Keyword Arguments with Positional Arguments**

It is possible to mix positional arguments and keyword arguments in a function call, but the positional arguments must appear first, followed by the keyword arguments. Otherwise, an error will occur.

```
show_interest(10000.0, rate=0.01, periods=10)
```

## 5.6 Global Variables and Global Constants

You've learned that when a variable is created by an assignment statement inside a function, the variable is local to that function. Consequently, it can be accessed only by statements inside the function that created it. When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is global. A global variable can be accessed by any statement in the program file, including the statements in any function. For example, look at Program below.

A global variable is accessible to all the functions in a program file.



### Program 5-13 (global1.py)

```
1 # Create a global variable.
2 my_value = 10
3
4 # The show_value function prints
5 # the value of the global variable.
6 def show_value():
7     print(my_value)
8
9 # Call the show_value function.
10 show_value()
```

### Program Output

10

The assignment statement in line 2 creates a variable named `my_value`. Because this statement is outside any function, it is global. When the `show_value` function executes, the statement in line 7 prints the value referenced by `my_value`.

An additional step is required if you want a statement in a function to assign a value to a global variable. In the function, you must declare the global variable, as shown in Program below.

**Program 5-14** (global2.py)

```
1 # Create a global variable.
2 number = 0
3
4 def main():
5     global number
6     number = int(input('Enter a number: '))
7     show_number()
8
9 def show_number():
10    print('The number you entered is', number)
11
12 # Call the main function.
13 main()
```

**Program Output**

```
Enter a number: 55 
The number you entered is 55
```

The assignment statement in line 2 creates a global variable named `number`. Notice inside the `main` function, line 5 uses the `global` key word to declare the `number` variable. This statement tells the interpreter that the `main` function intends to assign a value to the global `number` variable. That's just what happens in line 6. The value entered by the user is assigned to `number`.

Most programmers agree that you should restrict the use of global variables, or not use them at all.



## 5.7 Introduction to Value-Returning Functions

A *value-returning function* is a special type of function. It is like a void function in the following ways.

When a value-returning function finishes, however, it returns a value back to the part of the program that called it. The value that is returned from a function can be used like any other value: it can be assigned to a variable, displayed on the screen, used in a mathematical expression (if it is a number), and so on.

## 5.8 Writing Your Own Value-Returning Functions

You write a value-returning function in the same way that you write a void function, with one exception: a value-returning function must have a **return** statement. Here is the general format of a value-returning function definition in Python:

```
def function_name():  
    statement  
    statement  
    etc.  
    return expression
```

One of the statements in the function must be a **return** statement, which takes the following form:

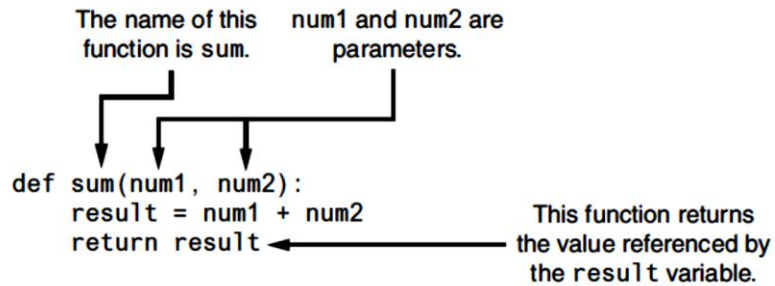
**return** *expression*

The value of the *expression* that follows the key word **return** will be sent back to the part of the program that called the function. This can be any value, variable, or expression that has a value (such as a math expression).

Here is a simple example of a value-returning function:

```
def sum(num1, num2):
    result = num 1 + num 2
    return result
```

### Parts of the function

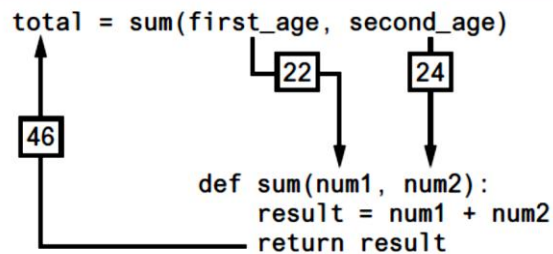


```
1  # This program uses the return value of a function.
2
3  def main():
4      # Get the user's age.
5      first_age = int(input('Enter your age: '))
6
7      # Get the user's best friend's age.
8      second_age = int(input("Enter your best friend's age: "))
9
10     # Get the sum of both ages.
11     total = sum(first_age, second_age)
12
13     # Display the total age.
14     print('Together you are', total, 'years old.')
15
16     # The sum function accepts two numeric arguments and
17     # returns the sum of those arguments.
18     def sum(num1, num2):
19         result = num1 + num2
20         return result
21
22     # Call the main function.
23     main()
```

### Program Output (with input shown in bold)

```
Enter your age: 22 
Enter your best friend's age: 24 
Together you are 46 years old.
```

Arguments are passed to the sum function and a value is returned



you can eliminate the result variable and rewrite the function as:

```
def sum(num1, num2):
    return num 1 + num 2
```

## How to Use Value-Returning Functions

Because value-returning functions return a value, they can be useful in specific situations. For example, you can use a value-returning function to prompt the user for input, and then it can return the value entered by the user. Suppose you've been asked to design a program that calculates the sale price of an item in a retail business. To do that, the program would need to get the item's regular price from the user. Here is a function you could define for that purpose:

```
def get_regular_price():
    price = float(input("Enter the item's regular price: "))
    return price
```

Then, elsewhere in the program, you could call that function, as shown here:

```
# Get the item's regular price.
reg_price = get_regular_price()
```

Program below shows the complete sale price calculating program using the functions just described

```

1  # This program calculates a retail item's
2  # sale price.
3
4  # DISCOUNT_PERCENTAGE is used as a global
5  # constant for the discount percentage.
6  DISCOUNT_PERCENTAGE = 0.20
7
8  # The main function.
9  def main():
10     # Get the item's regular price.
11     reg_price = get_regular_price()
12
13     # Calculate the sale price.
14     sale_price = reg_price - discount(reg_price)
15
16     # Display the sale price.
17     print('The sale price is $', format(sale_price, ',.2f'), sep='')
18
19 # The get_regular_price function prompts the
20 # user to enter an item's regular price and it
21 # returns that value.
22 def get_regular_price():
23     price = float(input("Enter the item's regular price: "))
24     return price
25
26 # The discount function accepts an item's price
27 # as an argument and returns the amount of the
28 # discount, specified by DISCOUNT_PERCENTAGE.
29 def discount(price):
30     return price * DISCOUNT_PERCENTAGE
31
32 # Call the main function.
33 main()

```

**Program Output** (with input shown in bold)

Enter the item's regular price: **100.00**

The sale price is \$80.00



## Returning Multiple Values

The examples of value-returning functions that we have looked at so far return a single value. In Python, however, you are not limited to returning only one value. You can specify multiple expressions separated by commas after the return statement, as shown in this general format:

*return expression1, expression2, etc.*

```
def get_name():  
    # Get the user's first and last names.  
    first = input('Enter your first name: ')  
    last = input('Enter your last name: ')  
    # Return both names.  
    return first, last
```

## 8.9 The math Module

The math module in the Python standard library contains several functions that are useful for performing mathematical operations. These functions typically accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result.

For example, one of the functions is named `sqrt`. The `sqrt` function accepts an argument and returns the square root of the argument. Here is an example of how it is used:

```
result = math.sqrt(16)
```

The Python standard library's math module contains numerous functions that can be used in mathematical calculations.



math Module Function	Description
<code>acos(x)</code>	Returns the arc cosine of $x$ , in radians.
<code>asin(x)</code>	Returns the arc sine of $x$ , in radians.
<code>atan(x)</code>	Returns the arc tangent of $x$ , in radians.
<code>ceil(x)</code>	Returns the smallest integer that is greater than or equal to $x$ .
<code>cos(x)</code>	Returns the cosine of $x$ in radians.
<code>degrees(x)</code>	Assuming $x$ is an angle in radians, the function returns the angle converted to degrees.
<code>exp(x)</code>	Returns $e^x$
<code>floor(x)</code>	Returns the largest integer that is less than or equal to $x$ .
<code>hypot(x, y)</code>	Returns the length of a hypotenuse that extends from (0, 0) to ( $x$ , $y$ ).
<code>log(x)</code>	Returns the natural logarithm of $x$ .
<code>log10(x)</code>	Returns the base-10 logarithm of $x$ .
<code>radians(x)</code>	Assuming $x$ is an angle in degrees, the function returns the angle converted to radians.
<code>sin(x)</code>	Returns the sine of $x$ in radians.
<code>sqrt(x)</code>	Returns the square root of $x$ .
<code>tan(x)</code>	Returns the tangent of $x$ in radians.

Program below demonstrates the `sqrt` function. Notice the `import math` statement in line 2. You need to write this in any program that uses the math module.

```

1  # This program demonstrates the sqrt function.
2  import math
3
4  def main():
5      # Get a number.
6      number = float(input('Enter a number: '))
7
8      # Get the square root of the number.
9      square_root = math.sqrt(number)
10
11     # Display the square root.
12     print('The square root of', number, 'is', square_root)
13
14     # Call the main function.
15     main()

```

**Program Output** (with input shown in bold)

Enter a number: **25**

The square root of 25.0 is 5.0