Functions

Functions

Functions, Built-In Functions, Commonly Used Modules, Function Definition and Calling the Function, The return Statement and void Function,

```
def ppsession(, , ):
    '''
    '''

ppsession( , ,)
```

Functions

- You can define functions to provide the required functionality. Here are simple rules to define a function in Python.
  - Function blocks begin with the keyword *def* followed by the function name and parentheses ( ) .
  - Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
  - The first statement of a function can be an optional statement - the documentation string of the function or docstring.
  - The code block within every function starts with a colon (:) and is indented.
  - The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Functions

- Functions are used when you have a block of statements that needs to be executed multiple times within the program. Rather than writing the block of statements repeatedly to perform the action, you can use a function to perform that action.
- This block of statements are grouped together and is given a name which can be used to invoke it from other parts of the program.
- You write a function once but can execute it any number of times you like.
- Functions also reduce the size of the program by eliminating rudimentary code. Functions can be either Built-in Functions or User-defined functions.

Functions

- A Function is a self block of code which is used to organize the functional code.
- Function can be called as a section of a program that is written once and can be executed whenever required in the program, thus making code reusability.
- Function is a subprogram that works on data and produces some output.

❖ **Types of Functions:**

There are two types of Functions.

✔ a) **Built-in Functions:** Functions that are predefined and organized into a library. We have used many predefined functions in Python.

✔ b) **User- Defined:** Functions that are created by the programmer to meet the requirements.

Built-In Functions

The divmod() function takes two numbers as arguments and return a pair of numbers consisting of their quotient and remainder.

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

https://docs.python.org/3/library/functions.html

https://www.w3schools.com/python/python_ref_functions.asp

		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()	

Commonly Used Modules

- Modules in Python are reusable libraries of code having *.py* extension, which implements a group of methods and statements. Python comes with many built-in modules as part of the standard library.
- To use a module in your program, import the module using *import* statement. All the *import* statement [illegible] the program.
- The syntax for import statement is,

For example, you can import the *math* module as

- 1. >>>import math

Import()

The syntax for using a function defined in a module is,

module_name.function_name()

The module name and function name are separated by a dot.

Here we list some of the functions supported by *math* module.

```
>>> import math
>>> print(math.ceil(5.4))
6
>>> print(math.sqrt(4))
2.0
>>> print(math.pi)
3.141592653589793
```

dir()

- The built-in function *dir()* returns a sorted list of comma separated strings containing the names of functions, classes and variables as defined in the module.

1. >>> dir(math)

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp','expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

```
>>> print(math.sqrt(4))
```

help()

- Another built-in function you will find useful is the *help()* function which invokes the built-in help system.

```
>>> help(math.gcd)
    Help on built-in function gcd in module math:
    gcd(…)
    gcd(x, y) -> int
    greatest common divisor of x and y
```

Random module

Another useful module in the Python standard library is the *random* module which generates random numbers.

- The syntax for *random.randint()* function is *random.randint(start, stop)* which generates a integer number between start and stop argument

numbers (including both)

```
 >>> import random
>>> print(random.random())
0.2551941897892144
 >>> print(random.randint(5,10))
9
```

OTP generation

```
import random
print(random.randint(0,9),random.randint(0,9),random.randint(0,9),
random.randint(0,9))
```

pip

- Third-party modules or libraries can be installed and managed using Python’s package manager ***pip***.
- The syntax for pip is,

  pip install module_name

- Arrow is a popular Python library that offers a sensible, human-friendly approach to creating, manipulating, formatting and converting dates, times, and timestamps.
- To install the *arrow* module, open a command prompt window and type the below command from any location.

pip

Below code shows a simple usage of arrow module.

```
C:\> pip install arrow
>>> import arrow
>>> a = arrow.utcnow()
>>> a.now()
<Arrow [2017-12-23T20:45:14.490380+05:30]>
```

Function Definition and Calling the Function

- You can create your own functions and use them as and where it is needed. User-defined functions are reusable code blocks created by users to perform some specific task in the program.
- The syntax for function definition is,

docstring

- The *def* keyword introduces a function definition. The term parameter or formal parameter is often used to refer to the variables as found in the function definition.
- The first statement among the block of statements within the function definition can optionally be a documentation string or docstring. There are tools which use docstrings to produce online documents or printed documentation automatically. Triple quotes are used to represent docstrings. For example,

```
""" This is single line docstring """
OR
""" This is
multiline
docstring """
```

Calling a function

- Calling the function actually performs the specified actions with the indicated parameters.

The syntax for function call or calling function is,

- Arguments are the actual value that is passed into the calling function. There must be a one to one correspondence between the formal parameters in the function definition and the actual arguments of the calling function.

Function flow

- A function should be defined before it is called and the block of statements in the function definition are executed only after calling the function.
- Normally, statements in the Python program are executed one after the other, in the order in which they are written.
- Function definitions do not alter the flow of execution of the program. When you call a function, the control flows from the calling function to the function definition. Once the block of statements in the function definition is executed, then the control flows back to the calling function and proceeds with the next statement.
- Python interpreter keeps track of the flow of control between different statements in the program.

When the control returns to the calling function from the function definition then the formal parameters and other variables in the function definition no longer contain any values.

Program to Find the Area of Trapezium Using the Formula Area = (1/2) * (a + b) * h Where a and b Are the 2 Bases of Trapezium and h Is the Height

```
def area_trapezium(a, b, h):
    area = 0.5 * (a + b) * h
    print(f"Area of a Trapezium is {area}")


area_trapezium(10, 15, 20)
```

program

```python
colg_name = input('your college name ' )

def NameOfCollege(college_name):
    print(f"The college name is {colg_name}")

NameOfCollege(colg_name)
```

The *return* Statement and *void* Function

- Most of the times you may want the function to perform its specified task to calculate a value and return the value to the calling function so that it can be stored in a variable and used later. This can be achieved using the optional *return* statement in the function definition.

The syntax for *return* statement is.

return

- If an expression list is present, it is evaluated, else *None* is substituted. The *return* statement leaves the current function definition with the expression_list (or *None*) as a return value.
- The *return* statement terminates the execution of the function definition in which it appears and returns control to the calling function. It can also return an optional value to its calling function.
- In Python, it is possible to define functions without a *return* statement. Functions like this are called **void functions**, and they return *None.*

- Functions without a return statement do return a value, albeit a rather boring one. This value is called *None* (it is a built-in name) which stands for “nothing”. Writing the value *None* is normally suppressed by the interpreter if it would be the only value written.
- If you want to return a value using *return* statement from the function definition, then you have to assign the result of the function to a variable.

- In some cases, it makes sense to return multiple values if they are related to each other. If so, return the multiple values separated by a comma which by default is constructed as a tuple by Python.
- When calling function receives a tuple from the function definition, it is common to assign the result to multiple variables by specifying the same number of variables on the left-hand side of the assignment as there were returned from the function definition. This is called tuple unpacking.

Functions

- Example:The following function takes a string as input parameter and prints it on standard screen

```
def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return;
```

Functions

Calling a Function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function –

```
# Function definition is here
>>> def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return;

# Now you can call printme function
 printme("I'm first call to user defined function!")
 printme("Again second call to the same function")
```

Functions

- **Function Arguments**
  - You can call a function by using the following types of formal arguments –

❖ positional arguments
❖ Keyword arguments
❖ Default arguments
❖ Variable-length arguments

Functions

Required arguments

- Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

```
# Function definition is here
>>> def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return;

# Now you can call printme() function
printme()
```

- When the above code is executed, it produces the following result –

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1
argument (0 given)
```

Functions

Keyword arguments

- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

```
# Function definition is here
>>> def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return;

# Now you can call printme() function
printme(str = ‘suresh’)
```

- When the above code is executed, it produces the following result –

Functions

- **Keyword arguments**

Functions

- **Default arguments**
  - A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed –

Functions

- **Variable-length arguments**
  - You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.
  - Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

Functions

- **Variable-length arguments**
  - An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example –

Functions

The Anonymous Functions

- Those functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Functions

- The syntax of *lambda()* functions contains only a single statement, which is as follows –

lambda [arg1 [,arg2,.....argn]]:expression

Functions

- **The return Statement**
  - The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Functions

- **The return Statement**