Functions

You're already familiar with the print(), input(), and len() functions. Python provides several builtin functions like these, but you can also write your own functions. **A *function* is like a mini-program within a program that performs a specific task.**

Why functions?

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.
- **A major purpose of functions is to group code that gets executed multiple times. Without a function defined, you would have to copy and paste this code each time.**

When you call the print() or len() function, you pass in values, called ***arguments*** in this context, by typing them between the parentheses.

Common Built-in Functions:

- abs()
- help()
- min()
- max()
- hex() - hexadecimal representation of an integer
- bin() - binary
- oct() - octal
- id()
- input()
- int()
- float()

- str()
- print()
- bool()
- range()
- round()
- pow()
- sum()
- ord()
- len()
- type()

Common str methods:

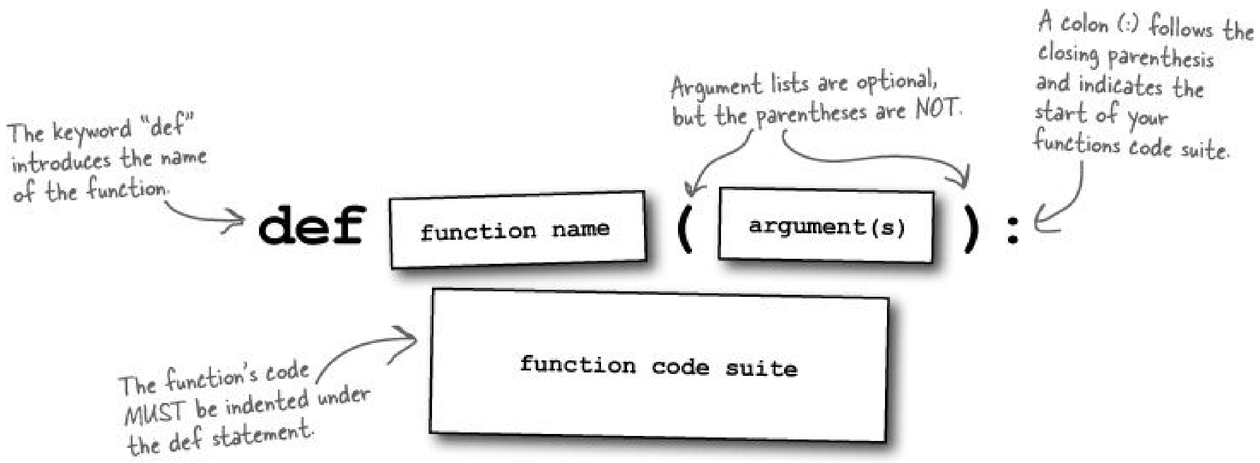
- capitalize()
- count()
- encode()
- endswith()
- expandtabs()
- find()
- format()
- isalpha()
- isdigit()
- isdecimal()
- islower()
- isupper()
- join()
- startswith()
- swapcase()
- title()

User-defined functions

Definition of a function

Syntax

```
def functionName(parameters) :
      statement(s)
```


The keyword "def" introduces the name of the function.
def
function name
(
argument(s)
)
:
Argument lists are optional, but the parentheses are NOT.
A colon (:) follows the closing parenthesis and indicates the start of your functions code suite.
function code suite
The function's code MUST be indented under the def statement.

- The keyword **def** introduces a function *definition*.
- It must be followed by the function name and the parenthesized list of formal parameters.
- There can be number of arguments in a function.
- The statements that form the body of the function start at the next line, and must be indented.
- Functions are used to perform a specific task multiple types.
- This code is executed when the function is called, not when the function is first defined.

- In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses.
- When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there.
- When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Example

```
def food(x): #defining of a function
    print(" i like" ,x)  #statement
```

Here, this function can be used by multiple users.

```
>>> food('pizza') #calling of function
 i like pizza     #output
```

```
>>>food("burger")
 i like burger
```

```
def coding(x):
    print(x, "studies at coding blocks")


>>> coding("mark") #calling
mark studies at coding blocks  #output

>>> coding("arushi")
arushi studies at coding blocks

>>> coding("jatin")
jatin studies at coding blocks
```

Function can also take integers as arguments.

```
def multiply(x,y):
    print(x*y)


>>> multiply(5,6)
30
>>> multiply(200,500)
100000
>>> multiply(398,890)
354220
```

Basically, in python it is not required to give each argument a datatype like other languages.

```
def student(name,rollno):
    print("name",name,"rollno",rollno)


>>> student("AA",71)
```

```
name AA rollno 71
>>> student("BB",72)
name BB rollno 72
>>> student("CC",73)
name CC rollno 73
```

```
 #Default parameters
def student(name="arushi",rollno=7): #default
    print("name",name,"rollno",rollno)

>>> student() #function with no argument
name arushi rollno 7

>>> student("jatin") function with one argument
name jatin rollno 7
```

Keyword Arguments

Functions can also be called using keyword arguments of the form `kwarg=value`.

In a function call, keyword arguments must follow positional arguments.

All the keyword arguments passed must match one of the arguments accepted by the function and their order is not important.

No argument may receive a value more than once.

```
def names(a,b,c):
    print(a)
    print(b)
    print(c)

>>> names(a="Mark",b="Python",c="Coding Blocks") explicit assignments
Mark
Python
Coding Blocks
```

The return statement

A program without the return statement

```
def add(a,b):
    print(a+b)

>>> a= add(6,7)
13
>>> print(a)
None
```

None here shows that there is no value that as been assigned to the function.

But if we use the return statement, the function will evaluate and return a value to the function.

```
def add(a,b):
    return(a+b)
>>> a=add(7,8)
>>> print(a)
15
```

Global and Local variables

The global statement is a declaration which holds for the entire current code block.

```
x="pizza"     #global variable
def food():
    print("i like",x)

>>> food()
i like pizza
```

In the previous example, we cannot alter x in the function as it is declared globally.

To do alteration of a global variable within a function we declared the global variable in the function using the keyword **global.**

Example

```
x="pizza"
def food():
    global x #declaration of the global variable in the function
    x="burgers" #alter
    print("i like",x)

>>> food()
i like burgers
```

Local variable

The variables which are defined inside a function are treated as local variables.

All the alterations which are done to these variables, has no effect on the variables which are defined outside the function. (even if they have the same names)

```
def places():
    print("london") #local variable within a function
p="love for paris"

>>> print(p)
love for paris
>>> places()
london
```

Example:

```
def CB():
    global f
    print(f)
    f = "Code in Python"
    print(f)


f = "I am looking for a course in coding blocks!"


>>> CB()
I am looking for a course in coding blocks!
Code in Python      # variable f got updated
```

```
>>> print(f)
Code in Python
```

Case of enclosures; functions can also be nested.

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

A **Nested Function** is a function defined inside another function. It's very important to note that the nested functions can access the variables of the enclosing scope.

```
def outer():
         x="local"
         def inner():
             print(x)
         inner()
         print(x)

>>> outer()
local
local
```

NonLocal

The nonlocal statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals.

This is important because the default behavior for binding is to search the local namespace first.

The statement allows encapsulated code to rebind variables outside of the local scope besides the global (module) scope.

```
def outer():
         x=10
         def inner():
             nonlocal x
             x+=5
             print(x)
         inner()
         print(x)

>>> outer()
15
15
```

Keyword Arguments

In Python, we can explicitly assign values.

This concept is called as keyword arguments.

They are called using keyword arguments of the form `kwarg=value`.

In a function call, keyword arguments must follow positional arguments.

All the keyword arguments passed must match one of the arguments accepted by the function and their order is not important.

No argument may receive a value more than once.

```
def names(a,b,c):
    print(a)
    print(b)
    print(c)

>>> names(c="mark",a="Coding Blocks",b="Python") #a,b,c are the keywords
Coding Blocks
Python
mark
```

Unpacking of Arguments

We use ‘ * ’ to unpack the list so that all elements of it can be passed as different parameters.

Elements are packed always in a tuple.

```
def netflix(*myFav):
    print(myFav)

>>> netflix("Sherlock","The Big Bang Theory", "How I met your
Mother")

('Sherlock', 'The Big Bang Theory', 'How I met your Mother')
```

Unpacking with default arguments.

```
def routine(p,q,r,*s,d="coding", e="reading"):
    print(p)
    print(q)
    print(r)
    print(s)
    print(d)
    print(e)

>>> routine("wake up","breakfast","TV","Sherlock","The Big Bang
Theory", "How I met your Mother", d= "reading",e="coding")


wake up
breakfast
TV
('Sherlock', 'The Big Bang Theory', 'How I met your Mother')
reading
coding
```

Default arguments has to specified because if they will not then they will considered as the part of the tuple.

```
def routine(p,q,r,*s,d="coding", e="reading"):
    print(p)
    print(q)
    print(r)
    print(s)
    print(d)
    print(e)

>>> routine("wake up","breakfast","TV","Sherlock","The Big Bang
Theory", "How I met your Mother","reading","coding")


wake up
breakfast
TV
('Sherlock', 'The Big Bang Theory', 'How I met your Mother',
'reading', 'coding')
coding
reading
```

Packing can also done as dictionaries.

We use ' ** ' to pack.

```
def routine(p,q,r,*s, **kwargs):
    print(p)
    print(q)
    print(r)
    print(s)
    print(kwargs)

wakeup
breakfast
TV
('Sherlock', 'flash') #tuple
{'work': 'coding'} #dictionary
```

Lambda Function ; Syntactic Sugars

They are small anonymous functions created with the lambda keyword consisting of a single expression which is evaluated when the function is called.

There is no return statement in lambda function.

They are syntactically restricted to a single expression which will be evaluated and returned.

Like nested function definitions, lambda functions can reference variables from the containing scope.

```
Syntex

lambda arguments : expressions
```

Arguments will be comma separated.

Expression will be evaluated and returned.

Lambda functions are called just like any other function.

Example

```
add = lambda a,b,c : a+b+c #definition

>>> add(5,6,7) #function call
18
```