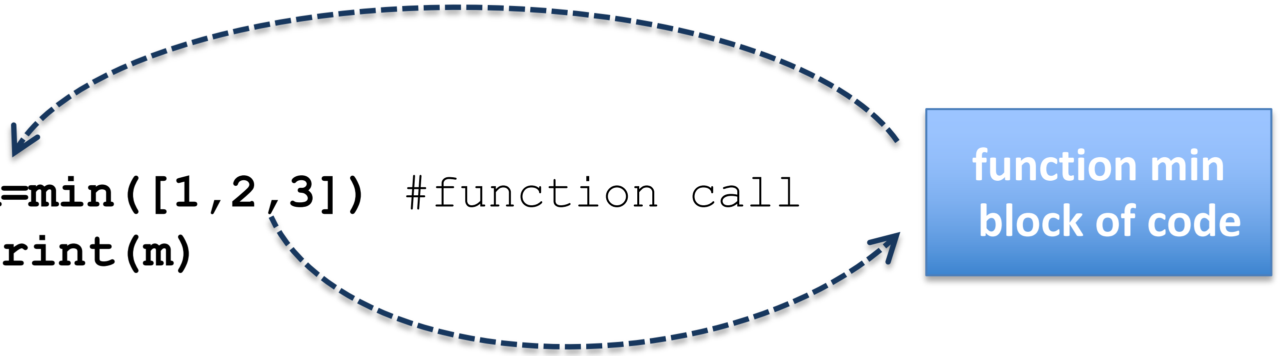
Functions

In programming, a **function** is a self-contained block of code that encapsulates a specific task or related group of tasks.
A function’s block of code is executed only when a function is called.

To call a function, use the function name followed by parentheses.

Call built-in functions - you need to know
- What arguments it takes
- What values it returns



```
m=min([1,2,3]) #function call
print(m)
```

```
l=len('Hello')
print(l)
```

Creating your own functions is important

You write some code that implements some task and find that you need that task in many different locations within your script.
Don't Repeat Yourself (DRY) Principle of software development – reducing repetition and avoiding redundancy

Fahrenheit to Celsius formula

$$°C = \frac{°F - 32}{1.8}$$

Math style

$$fc(x) = \frac{x - 32}{1.8}$$

```
xf=100
xc=(xf-32)/1.8
print(xc)
…
…
xf=75
xc=(xf-32)/1.8
print(xc)
```

Code reusability

Define Fahrenheit to Celsius function

```
def fc(x):
     C=(x-32)/1.8
     return C






#Call the function
xc=fc(100)
print(xc)

xc1=fc(75)
print(xc1)
```

Defining and calling functions - return statement

A function’s block of code is executed only when a function is called

```
#function definition
def function_name(parameters):
    "function_docstring"
     code block
     return expression

#function call
var=function_name(arguments)
print(var)
```

Make a script called ***myfunc.py:***

```
#Define this function
def fc(x):
     "convert temp from F to C"
     C=(x-32)/1.8
     return C

#Call the function
y=fc(100)
print(y)
y1=fc(75)
print(y1)
```

Return statement

A parameter is a name of a variable listed inside the parentheses in the function definition. An argument is a value that is assigned to a parameter when a function is called.

```
#Function definition
def fc(x):          # x is a parameter
    "convert temp from F to C"
    C=(x-32)/1.8
    return C




#Function call. Use the function name followed by parentheses
 y=fc(100)      # 100 is the argument

when you call the function
# argument 100 is assigned to the parameter x
  Mechanism of argument passing
# block of code function is executed
# return: the value of C is passed back to the function call
  and can be stored in variable y

 print(y)
```

Defining and calling functions – no return statement

A function’s block of code is executed only when a function is called

```
#function definition
def function_name(parameters):
    "function_docstring"
     code block

#function call
function_name(arguments)
```

Make a script called ***myfunc1.py***

```
#Define this function
def fc(x):
    "convert temp from F to C"
    C=(x-32)/1.8
    print(C)

#Call the function
fc(100)
fc(75)
```

No return statement

```
#Function definition
def fc(x):          # x is a parameter
       " convert F in C"
       C = (x-32)/1.8
      print(C)



#Function call. Use the function name followed by parentheses
fc(100)             # 100 is the argument
```

```
when you call the function
# argument 100 is assigned to the parameter x
  Mechanism of argument passing
# block of code function is executed
  None is returned
```

Common error: storing None in a variable when calling a function without a return statement

```
x=fc(100)    #in x None is stored
print(x)
None
```

Parameters of a function

A parameter is a variable listed inside the parentheses in the function definition.
An argument is a value that is assigned to a parameter when a function is called.

The number of arguments in the function call should match exactly with the number of parameters in the function definition (positional order).

This function takes **two parameters** and returns one value.

```
def division(par1,par2):    #two parameters
    "divide two numbers"
    D=par1/par2
    return D

# we run the function on two values
y=division(10,5) # we pass two arguments or values
z=division(5,10)
```

Do variables y and z store the same results? Print results to screen. Run the script

Different data types can be arguments of a function

```
def multiply(a,b):
    " multiplication or repetition”
     return a*b


x=multiply(10,20) #the arguments are two numbers
print(x)

y=multiply("hello",4) # the arguments are a string and a number
print(y)
```

Now let’s define the same function but without a return statement.

```
def multiply(a,b):
    " multiplication or repetition”
     print(a*b)

L=[1,2]
n=4
multiply(L,n) # the arguments are a list and a number
```

Try to pass two lists. Would this work?

Practice: function with two parameters and return

Write a script called ***myfunc1.py*** and in it do the following:

a) Make a function called *mysum* that takes two parameters and returns the sum of those two parameters. Write a doc string within the function describing what this function does

b) Call this function on two lists~~,~~ and store the result in variable **L**. Print the variable

c) Would you be able to use that same function on two strings and two numbers? Answer in a comment line

Function takes no parameters

A function's block of code is executed only when a function is called

This function takes **no parameters,** uses the input function and returns a value

```
def f1():
     num=float(input("enter a number: "))
     print("You entered the number:", num))
     return num*num

y=f1()
print(y)
```

This function takes **no parameters,** uses the input function and does not return

```
def f2():
    num=float(input("enter a number: "))
    print("You entered the number:", num))
    print(num*num)

f2()
```

Practice: Taking input from the input function

Make a script called ***input1.py*** and in it:

a) Make a function *input_and_sum()* that does the following:

- takes no parameters
- uses the input function to ask the user to enter two numbers separated by a comma.
- returns the sum (arithmetic) of those two numbers.

b) Run the function on two numbers, and print the result to screen

Practice: Taking input from the input function, and no return

Make a script called ***input2.py*** and in it:

a) Make a function *input_and_sum()* that does the following:

- takes no parameters
- uses the input function to ask the user to enter two numbers separated by a comma.
- **print** the sum (arithmetic) of those two numbers.

b) Run the function on two numbers. The result should be displayed to screen.
None should not appear in the output.

Functions can return multiple values

Multiple values can be returned as a tuple

```
def f3(arg):
    return arg, arg*2, arg*3

x=f3(10)
print(x)
(10,20,30)

print(type(x)) #tuple type
print(x[0]) #access first element of tuple x - first returned value
```

Common error: you cannot use the return statement more than once in this way:

```
    def f3(arg):
        return arg
        ~~return arg*2~~
        ~~return arg*3~~
```

Practice: function returns multiple values

Make a script ***myfunc2.py*** and in it:

a. Make a function called *dict_keys_values* that takes any dictionary as an input parameter and returns both the list of keys and the list of values of that dictionary.

b. Run your function on this dictionary and store the result in variable d_out

```
d1 = {'a': 1, 'b': 2}
```

c. Print variable d_out

d. Access values in d_out and print the list of keys

e. Access values in d_out and print the list of values

f. Would you be able to run this function on another data type?
Answer with a comment line

Local and global variables

Local variables can be used only inside the function in which they are declared.

```
def f4(n):
      var=5  #local variable - defined inside a function
      return var*n

print(var)
x=f4(5)
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'myvar1' is not defined
```

Global variables can be used throughout he program, including inside a function:

```
var=5   #global variable - defined outside a function

def f5(n):
     return var*n

print(var)
x=f5(5)
```

Functions can call other functions

We've seen functions that can print text or do simple arithmetic, but functions can be much more powerful than that.
For example, a function can call another function:

```
def square(x):
     s=x*x
     return s

def sum_of_squares(x,y,z):
    sx=square(x)
    sy=square(y)
    sz=square(z)
    return sx+sy+sz

a=1
b=2
c=3
result=sum_of_squares(a,b,c)
print(result)
```

Example: If-statements in functions

If-statements can be in the function's block. For example:

Make a script called if-func.py and define this function:

```
def c1(arg):
    if arg < 5:
        return arg
    else:
        return arg**2

x=c1(3)
print(x)


def c2(arg):
    if arg < 5:
        print(arg)
    else:
        print(arg**2)

c2(3)
```

Example: loops in functions

Loops can be in the function’s block . For example:

- In a script called my_upper.py make a function called *string_upper*. The function should take one string as a parameter, convert it to uppercase, and **print** each character on one line. Use a for loop. Run the function an arbitrary string

```
def string_upper(mystr):
    for c in mystr:
        print(c.upper())


string_upper("music")
```

- In a script called **conv_numstring.py,** make a function called *num_str* that takes one integer as parameter, and returns a string of the numbers from 0 to n-1. Example, if the integer value is 6, the returned string will be `'012345'`

```
def num_str(n):
    s1 = ''
    for i in range(n):
        s1 = s1 + str(i)
    return s1


s=num_str(6)
print(s)
```

Example: functions in comprehensions

Functions that have a return statement can be called within comprehensions.

For example:

```
def FCconversion(F):
    """
       Takes a temperature in Fahrenheit F,
       converts it to Celsius C, and return C
       """
    C=(F-32)/1.8
    return(C)
```

We create a list of temperature values in Celsius from temperature values 1-100 in Fahrenheit:

```
L=[ FCconversion(i) for i in range(1,101)]
```

Summary of what we learned about return

Return statement causes your function to exit and hand back a value to its caller

```
def f1(arg):
       return arg*arg

x=f1(10)
print(x)
```

Multiple values are returned as a tuple

```
def f2(arg):
    return arg, arg*2, arg*3

x=f2(10)
print(x)
(10,20,30)
print(x[1]) # print only the second value in the tuple
```

Function with no return - you can not save the result in a variable; in this case the calculated value only gets printed to screen

```
def f3(arg):
    value=arg*arg
    print(value)

f3(10)
```

Summary of what we learned about arguments of a function

This function takes **one parameter**, and returns one value

```
def f4(arg):
      a2=arg*arg
      return a2

x = f4(10) # call the function
print(x)
```

This function takes **two parameters**, and returns one value

```
def f5(arg1,arg2):
      M=arg1*arg2
      return M

y = f5(10,20) # call the function
print(y)
```

This function takes **no parameters**, and returns one value

```
def f6():
      return 'Have a good day!'

z = f6() # call the function
print(z)
```