Function supporting variable number of arguments:

Have you observed that we can call the print function with any number of arguments? How is this possible?

```
# name: 1_fn_variable_parameter

# Variable number of parameters: is a tuple

def fn(*arg) :
      print(arg, type(arg))

fn(11, 22)
fn(66)
fn(33, 44, 55)

# function to find the min
def mymin(*arg): # arg has at least one element
      # assume that the first element is the smallest
      x = arg[0]
      # walk through the remaining elements
      for e in arg[1:]:
            #if any element in the tupple is smaller than x, update x
            if e < x :
                  x = e
      return x

print(mymin(20, 10, 30, 5, 40))
print(mymin(400, 200, 100, 300))
```

Consider this function. Observe the * before the name of the parameter. **This parameter can receive any number of arguments**. **It is called a variable parameter**. **It stores all these arguments as a tuple**. So the tuple could even have 0 arguments.

```
def fn(*arg) :
      print(arg, type(arg))
```

There are a few rules.

1. There can be only one such parameter in a function.
   def foo(*p, *q) : is an error
2. This follows all the positional parameters.

```
def fn(x,y,*arg) :
     print(x,y,arg)
fn(1,2)
fn(1,2,3,4)

def fn(*arg,x,y) :
     print(arg,x,y)
fn(1,2,3,4) # TypeError: fn() missing 2 required keyword-only arguments: 'x' and 'y'
fn(1,2,x=3,y=4)
```

3. This parameter cannot be used as a keyword parameter.

We cannot specify the parameter name and give a tuple of arguments. The second example shows how to use variable parameter to find the smallest in number of arguments assuming that there is at least one argument.

Example:

The full syntax of print() is:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- **objects** - object to the printed. * indicates that there may be more than one object
- **sep** - objects are separated by sep. **Default value**: ' '
- **end** - end is printed at last
- **file** - must be an object with write(string) method. If omitted it, sys.stdout will be used which prints objects on the screen.
- **flush** - If True, the stream is forcibly flushed. **Default value**: False

Function supporting key value pairs as arguments:

Another way of parameter passing is to specify key value pairs which will be stored as a single parameter of the type dict. This is much simpler than having a number of keyword parameter.

Let us have a look at the function defined below. Observe that the parameter name is preceded by **two asterisk symbols.**

Observe the call. **The key is specified as an identifier and then symbol = follows and then any expression as argument.** A parameter named **kw** here **is created of type dict** where **all these identifiers become keys of type str**. The corresponding value specified in the call becomes the value for that key.

 # name: 2_fn_key_value_pairs.py

```
# key value pairs
def foo(**kw):
      print(kw, type(kw))

foo(king = 'dasharatha', wives = ['kousalya', 'sumitra', 'kaikeyi'],
    sons = ['rama', 'lakshmana'])
print();print()
```

The example below illustrates how we can use key value pairs to mimic default parameters.
A text to be depicted on a screen has number of attributes – font name, size, style, background colour, foreground colour and so on. The user normally may not want to specify all of these and may be happy with the defaults and in certain cases, he may want to override. We do not want to specify a long list of keyword parameters with defaults. Here is the solution.

The function definition has a default `dict` called *`default_attr`*. When the user sends key value pairs to the parameter kw, then the *`default_attr`* is updated with the keyvalue pairs specifed by the user and the rest still hold on to the default values.

```
def attr(**kw):
      default_attr = {
                  'font' : 'times new roman',
                  'size' : 10,
                  'style' : 'regular',
                  'bgcolor' : 'blue',
                  'fgcolor' : 'green'
                  }
      default_attr.update(kw)
      print(default_attr)
attr(font = 'calibri', size = 12, style = 'bold', bgcolor = 'white', fgcolor ='black')
attr()
attr(size = 20, bgcolor = 'blue')
```

Function with no name:

There are cases where the function body is just an expression and we may want to use it once and so we do not care to give a name for it. In such cases, we use what are called **lambda functions – functions with no name.**

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

Syntax

`lambda` *`arguments`* `:` *`expression`*

The expression is executed and the result is returned.

This function squares a given number. This expression is callable.

```
>>>lambda x : x * x
   <function <lambda> at 0x000000000942D9D8>
```

The following expression evaluates to 100.

```
>>>(lambda x : x * x)(10)
    100
```

```
# name: 3_fn_lambda.py
#      function with no name
#      has a list of parameters
#      :
#      a single expression

lambda x : x * x
print("res : ", (lambda x : x * x)(10))
```

```
f1 = lambda x, y : x * y
f2 = lambda x, y : x * y

print(f1, type(f1))
print(f2, type(f2))
print(f1(10, 20))
print(f2(10, 20))


# Python code to illustrate cube of a number showing difference between def()and lambda().
def cube(y):
    return y*y*y;

g = lambda x: x*x*x
print(g(7))

print(cube(5))
```

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function. Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles/triples the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Lambda definition does not include a “return” statement, it always contains an expression which is returned. We can also put a lambda definition anywhere a function is expected, and we don’t have to assign it to a variable at all. This is the simplicity of lambda functions.

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

Defining Function Again:

We know that a variable can be re-assigned giving a new value for it.

Example:

```
a = 10  # a is an int with the value 10
a = 20  # a is still an int with the value 20
```

What happens if we redefine a function?

The earlier definition gets replaced and from that point onwards, the new function manifests.

```
#name : 4_fn_redefne.py
def foo():
      print("one")

foo() # one
# replaces the old function with the new one
def foo():
      print("two")
foo() # two
```

Callback:

The mechanism of passing a function name as argument and calling it indirectly is called callback.

A **callback** is a function that's called from within another, having initially been "registered" for use at an outer level earlier on.

In this program, we define two functions *foo1* and *foo2*. We define a function *bar* which takes one argument. We invoke the function *bar* and we pass the function name foo1 as the argument in the first call. The parameter **fn** gets foo1 and so becomes callable. When the bar executes to call to fn, it calls the function foo1. In the second call, bar ends up calling foo2.

Observe that the function bar is flexible. It does not know which function will be invoked when fn is called. It depends on what the user specifies as the argument. This mechanism of passing a function name as argument and calling it indirectly is called callback. We use this concept very heavily in programming.

```
#name: 5_fn_callback.py
def foo1():
      print("this is foo1")

def foo2():
      print("this is foo2")

def bar(fn): # the parameter fn gets function reference and so becomes callable
      fn()

bar(foo1)
bar(foo2)
```

Some Examples of callback:

There are many built-in functions which take the callback as an argument.
Let us examine the function sorted. It returns a sorted list. It takes as the first argument an iterable.

By default, the elements of the iterable are compared as they are and rearranged if the elements are not in the right order.
We can provide a function by which each of the elements is changed before comparing using the less than operator. For this purpose, we should provide a callable which takes one argument. This argument will be provided as the sorted function walks through the iterable. Instead of comparing the elements of the iterable directly, the function is called on each of these elements and the results are compared and based on this comparison, the elements in the list are arranged.

Sorted creates a **temporary list** by applying `str.upper` on each element.
`['ABCD', 'XYZ', 'ABEF', 'BX', 'ZSS'].`

Based on the comparison of these elements, the elements of a are ordered.

```
a = ['abcd', 'XyZ', 'Abef', 'Bx', 'zss']
print(sorted(a, key = str.upper))   # Syntax:  sorted(iterable, key=None, reverse=False)
                                    #   key -> A function to execute to decide the order. Default is None
```

This is the output.
['abcd', 'Abef', 'Bx', 'XyZ', 'zss']

Let us look at the second example.
The callback key is a lambda function – which combines 0^{th} and 2^{nd} character by concatenation. This value decides the ordering.

```
a = [ 'abcd', 'axae', 'xcyz', 'xwxz' ]
# combine 0th and 2nd char
print(sorted(a, key = lambda s : s[0] + s[2]))
```

This is the result.
`['axae', 'abcd', 'xwxz', 'xcyz']`

```
#name : 6_fn_callback_examples.py
# sort
a = ['abcd', 'XyZ', 'Abef', 'Bx', 'zss']

# sort in a case-sensitive way
print(sorted(a))

# sort in a case-insensitive way
print(sorted(a, key = str.upper))

a = [ 'abcd', 'axae', 'xcyz', 'xwxz' ]
# combine 0th and 2nd char
print(sorted(a, key = lambda s : s[0] + s[2]))

print(sorted(a, key = len))
```

The callback is an amazing concept. The sorted function is flexible. It does not hardcode the way of comparison. The user can decide how to compare based on the function being passed as a callback.

Example 1:

```
def add(numbers, callback):
    results = []
    for i in numbers:
        results.append(callback(i))
    return results

def add2(number):
    return number + 2

def mul2(number):
    return number * 2

print add([1,2,3,4], add2) # [3, 4, 5, 6]
print add([1,2,3,4], mul2) # [2, 4, 6, 8]
```

Example 2:

```
authors = ['Octavia Butler', 'Isaac Asimov', 'Neal Stephenson', 'Margaret Atwood',
'Usula K Le Guin', 'Ray Bradbury']

# Returns list ordered by length of author name
print(sorted(authors, key=len))

# Returns list ordered alphabetically by last name.
print(sorted(authors, key=lambda name: name.split()[-1]))
```

Global variables:

We talk about two terms in programming – **life** and **scope**.

Life of a variable is about existence of the variable – the variable has a location and therefore some value. The variable loses its life when the reference count becomes 0.

The variable has scope if it can be seen – is visible – in the current suite.

We talk about local and global symbols.
All names created outside of functions are global.
All names created within a function by default are local to those functions.

We will discuss about the finer points of local and global variables.

```
# name: 7_fn_global.py
```

```
# scope:
#     Where we can access; visibility
# life:
#     Existence of the variable
#     depends reference counting
a = 100
e = 600
f = 700
h = 1111
def foo() :
      print(a) # 100 # global variable
      b=150
      print(b) # 150
      c = 30   # local variable
      if c == 30 :
            d = 40      # local variable
      else:
            d = 50      # local variable

      print(d)    # ok; d is local to the function; no concept of local to a suite of
                  # a control structure

      e = 60      # local variable

      # cannot directly modify a global variable
      # to access the global variable, for its value, nothing is required
      # to modify a global variable, we should declare that it is global
      global f
      f = 70
      # can we create a global variable?
      global g
      g = 1000

      #print(h) # UnboundLocalError: local variable 'h' referenced before assignment
      #h = 2222
b = 200
foo()
#print(c) # not defined
print(e)  # 600
print(f)  # 70
print(g)  # 1000
```

Observe the variable name a. It is global. It is accessible by value within any function defined within this global scope. So, the value of a can be accessed in the function foo.

The variable b is created within the function foo – it is local to it. This is not related to the global variable b.
The variable c is local to foo. The variable d belongs to the function foo and not the suite of if or else or any control structure – is also local to foo.
The variable e is global. But the function foo defined a local variable e as well. The global variable e is no more directly accessible within foo.

A function can use the value of a global variable without modifying it. To change a global variable, the function should declare that it wants to play with the global variable.

Observe:

```
global f
f = 70
```

Now the function can use as well as change the global variable.
Unusually, we can even create a global variable within the function which would exist in the global environment when the function returns.

```
global g
g = 1000
```