```python
# -*- coding: utf-8 -*-
"""
Created on Tue Oct  6 22:06:24 2015
@author: hina
Reference: https://docs.python.org/3/tutorial/index.html
"""

#######################################

print ()

# The keyword 'def' introduces a function definition.
# It must be followed by the function name and the parenthesized list of formal
# parameters.
# The statements that form the body of the function start at the next line,
# and must be indented.
def fib (n):
    # The first statement of the function body can optionally be a fucntion's
    # documentation string or docstring that summarizes what the function does
    """This function returns the Fibonacci Series."""
    result = []
    a, b = 0, 1
    for i in range(n):
        result.append(b)
        a, b = b, a+b
    return result

# print Document String
print (fib.__doc__)

# call function with arguments
print(fib(10))

print ()

# Parameters versus Arguments:
#    - Parameters are the variables in the function
#      (function fib(n) has parameter n)
#    - Arguments are the values given to the variables at the point of call
#      (function call fib(10) has argument 10)
#    - So outside the function, it is more common to talk about arguments.
#      Inside the function, you can really talk about either.

#######################################

# Let's understand how function calls work in Python in terms of
# Objects, Bindings, and Scope

# Python is neither "call-by-reference" nor "call-by-value".
# In Python a variable is not an alias for a location in memory.
# Rather, it is simply a binding to a Python object.

# If I call foo(bar), I'm merely creating a binding within the scope of foo
# to the object that the argument bar is bound to when the function is called.

# If bar refers to an immutable object, the most that foo can do is create
# a name bar in its local namespace and bind it to some other object.
def foo (bar):

    # (2) The Local Name bar is Bound to String Object 'old value'
    print (bar)

    # (3) The Local Name bar is now Bound to String Object 'new value'
    #     (and no longer bound to String Object 'old value')
    bar = 'new value'
    print (bar)

# (1) The Name answer is Bound to the String Object 'old value'
answer = 'old value'
foo (answer)

# (4) The Name answer is still Bound to String Object 'old value'
print (answer)

print ()

# If bar refers to a mutable object and foo changes its value,
# then these changes will be visible outside of the scope of the function.
def foo (bar):

    # (2) The Local Name bar is also now Bound to the List Object created
    print (bar)

    # (3) The 0th Element of the List Object created now contains the Int Object 42
    bar.append(42)
    print (bar)

# (1) The Name answer_list is Bound to the List Object []
answer_list = []

# (4) The Name answer_list is still Bound to the List Object
#     and reflects the changes made to it in the function call
foo (answer_list)
print (answer_list)

print ()

#######################################

# The execution of a function introduces a new symbol table used for the local
# variables of the function.
# Variable references first look in the local symbol table, then in the local
# symbol tables of enclosing
# functions, then in the global symbol table, and finally in the table of
# built-in names.
# Thus, global variables cannot be directly assigned a value within a function
# (unless named in a global statement), although they may be referenced.

def demoFunc (arg1, arg2):

    # this will change local values of a1 and a2
    a1, a2 = -100, -200
    # this will change global value of a3
    global a3
    a3 = -300

    # this will pick up local copy of a1 and a2, and global copy of a3
    print ("demoFunc: ", a1, a2, a3)

# decalare global varaibles a1, a2 and a3
a1, a2, a3 = 10, 20, 30
print ("main: ", a1, a2, a3)

# function call will not change global copy of a1 and a2,
# but will change global value of a3
demoFunc (a1, a2)
print ("main: ", a1, a2, a3)

# this will change global copy of a1, a2 and a3
a1, a2, a3 = a1+5, a2+5, 35
print ("main: ", a1, a2, a3)

# function call will not change global copy of a1 and a2,
# but will change global value of a3
demoFunc (a1, a2)
print ("main: ", a1, a2, a3)

print ()

#######################################

# It is  possible to define functions with a variable number of arguments.
# This can be done in three ways:
#     (1) Default argument values
#     (2) Keyword arguments
#     (3) Arbitrary argument lists

# (1) Default Argument Values

# You can specify a default value for one or more arguments
def isNumberInRange (num, i=25, n=100):
    if num in range(i, n):
        print ("found")
    else:
        print ("not found")
isNumberInRange(20)
isNumberInRange(20, 0)
isNumberInRange(20, 0, 10)
print ()

# Note: the default value is evaluated only once.
# This makes a difference when the default is a mutable object.
# So for instance this will print [1] on the first call,
# [1, 2] on the second, and [1, 2, 3] on the third.
def lstAppend(a, L = []):
    L.append(a)
    return L
```

```
print (lstAppend(1))
print (lstAppend(2))
print (lstAppend(3))
print()

# You can override this behavior as follows.
# This will print [1] on the first ccall, [2] on the second, and [3] on the third.
def lstAppend(a, L = None):
    if L is None:
        L = []
    L.append(a)
    return L
print (lstAppend(1))
print (lstAppend(2))
print (lstAppend(3))
print ()

# (2) Keyword Arguments

# Arguments can be of two kinds:
#     Keyword Argument:
#         An argument preceded by an identifier (e.g. name=) in a function call
#     Postional Argument:
#         An argument that is not a keyword argument

# 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 func (arg1, arg2=0, arg3=-1, arg4=0):
    print (arg1, arg2, arg3, arg4)

func (1000)                 # 1 positional argument
func (arg1=10)              # 1 keyword argument
func (arg1=10, arg2=100)    # 2 keyword arguments
func (arg2=100, arg1=10)    # 2 keyword arguments - order is not important
func (10, 100, 1000)        # 3 positional arguments
func (10, arg2=100)         # 1 positional and 1 keyword argument

# func (arg4=10, 100)       # this will yield non-keyword arg after keyword arg error
# func (arg5=10)            # this will yield unexpected keyword argument error
# func (0, arg1=0)          # this will yield multiple values for argument error

print()

# Keyword Argument can also passed as values in a dictionary preceded by **.
# Postional Argument can also be passed as elements of an iterable preceded by *.

def foo(*positional, **keywords):
    print ("Positional:", positional)
    print ("Keywords:", keywords)

# The *positional (tuple)  argument will store all of the positional arguments passed
# to foo(), with no limit to how many you can provide.
foo('one', 'two', 'three')
```

```
# The **keywords (dictionary) argument will store any keyword arguments:
foo(a='one', b='two', c='three')

# And of course, you can have both at the same time:
foo('one','two',c='three',d='four')
print()

# (3) Arbitrary Argument Lists

# You can also call a function with an arbitrary number of arguments.
# These arguments will be wrapped up in a tuple.
# Zero or more normal arguments may occur before the variable number of arguments.
# Any formal parameters which occur after the *args parameter can only be used
# as keyword (and not positional) arguments.

def concat (*args, sep='/'):
    print (sep.join(args))

concat ('john', 'jane')
concat ('john', 'jane', 'sandra')
concat ('john', 'jane', 'sandra', sep=',')

print()

#######################################

# test

def func (lst):

    global a2
    a1 = 10

    lst.append(a1)
    lst.append(a2)

myList = []
print (myList)

a1, a2 = 5, 15
func(myList)
print (myList)

a1, a2 = 25, 35
func(myList)
print (myList)

print ()


def argListsCheck (argNorm1, argNorm2=10, *argPos, **argKey):
    print ("argNorm1 = ", argNorm1)
    print ("argNorm2 = ", argNorm2)
    print ("*argPos = ", argPos)
    print ("**argKey = ", argKey)
```

```
argListsCheck (0, 1, 2, 3, a=10, b=20, c=30)

print ()
```