Computational Thinking with Programming

Lecture - 10

Function : Continues….

@cse_bennett @csebennett

Today…

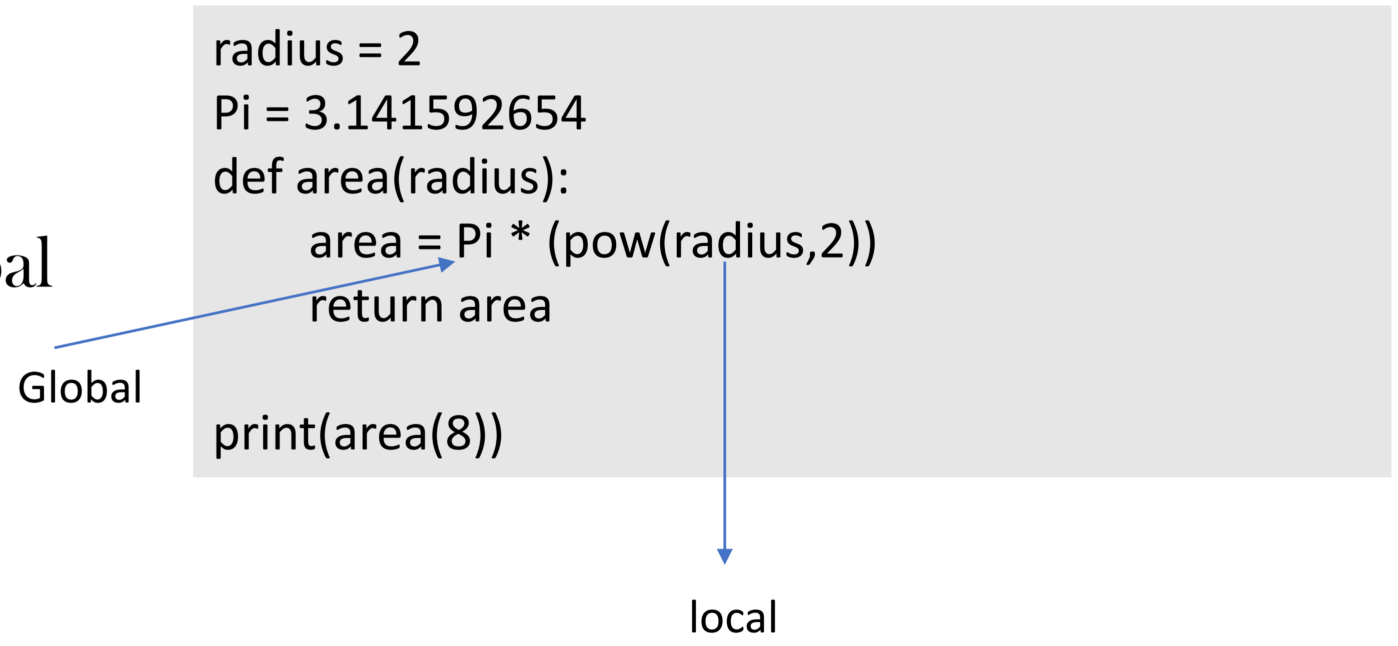
- Last Session:
  - Functions (Definition)

- Today's Session:
  - Functions- Part III:
    - More on Function Argument and Return
    - Recursion
- Hands on Session with Jupyter Notebook:
  - We will practice on the user defined function in Jupyter Notebook.

LGB Rule

- Name references search the three scopes in order:
  - Local
  - Global
  - Built-in

- Local variable override Global
- Local and global variables can override built-in



```
radius = 2
Pi = 3.141592654
def area(radius):
    area = Pi * (pow(radius,2))
    return area

print(area(8))
```

Indirect Function Calls

- Functions can be assigned to names and passed to the other functions

```
def hello():
        print(“Hello Bennett”)
X = hello
X()
```

```
def call(function, *args):
        function(args)

call(multiply,2,2)
```

```
obj_dict = { ‘function’ : multiply, ‘args’ : (2,2)}

apply(obj_dict[‘function’],obj_dict[‘args’])
```

More on the Print Function

- There are different forms of the print function

```
>>> print() # print blank line

>>> print(3+4)
7
>>> print(3, 4, 3+4)
3 4 7
>>> print("The answer is ", 3 + 4)
The answer is  7
>>> print("The answer is", 3 + 4)
The answer is 7
```

```
>>> def answer():
...     print("The answer is:", end = " ")
...     print(3 + 4)
...
>>> answer()
The answer is: 7
>>>
```

Notice how we used the end parameter to allow multiple prints to build up a single line of output!

Function with Multiple Return Values

- Multiple variable can also be returned as a tuple. However, this tends not to be very readable when returning many value and can easily introduce errors when the order of return values is interpreted incorrectly.
- Example:

```
# Function definition is here
def Arithmatic(a, b):
    sum = a+b
    sub = a-b
    mul = a*b
    div  = a/b

    return sum, sub, mul, div

# Now you can call this function
r, s, t, u = Arithmatic(15, 2)
print(r, s, t, u)

print(Arithmatic(15, 2))
```

Output:

```
17 13 30 7.5
(17, 13, 30, 7.5)
```

Pass By Value

- The function only receives the *values* of the parameters

1000

addInterest(…) gets ONLY the *value* of amount (i.e., 1000)

```
>>> def addInterest(balance, rate):
...     newBalance = balance * rate
...     balance = newBalance
...
>>> def test():
...     amount = 1000
...     rate = 1.05
...     addInterest(amount, rate)
...     print(amount)
...
>>> test()
1000
>>>
```

Python is said to *pass parameters by value*!

Pass By Value vs. Returning a Value

- Consider the following code

```
def increment_func(x):
        x = x + 1

x = 1
increment_func(x)
print(x)
```

Run

```
1
```

```
def increment_func(x):
        x = x + 1
        return x

x = 1
x = increment_func(x)
print(x)
```

Run

```
2
```

Passing Sequence in The Function

- You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.
- If you send a List as an argument, it will still be a List when it reaches the function.

```
def Fun(list1):
    print ("In Fun: ", list1)

#Calling change function
mylist = [10,20,30];
Fun( mylist );
print ("Outside Fun: ", mylist)
```

```
Output:
In Fun: [10, 20, 30]
Outside Fun: [10, 20, 30]
```

```
def Fun(S1):
    print (" Inside Fun: ", S1)

# Calling change function
S = "Hello"
Fun( S);
print ("Outside Fun: ", S)
```

```
Output:
In Fun: Hello
Outside Fun: Hello
```

```
def Fun(t1):
   t1[0] = t1[0] +1
   print (" Inside: ", t1)

# Calling change function
tup = (10,20,30);
Fun( tup);
print ("Outside: ", tup)
```

```
Output:
Error: Tuple object does not
support item assignment
```

Pass by Reference

- All parameters in the Python language are passed by reference.
- It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.
- **Example: What will be the output ?**

```
def change(mylist1):
    mylist1.append(40);
    print ("In Function: ", mylist1)
    return

#Calling change function
mylist = [10,20,30];
change( mylist );
print ("Outside Function: ", mylist)
```

```
Output:
In Function: [10, 20, 30, 40]
Outside Function: [10, 20, 30, 40]
```

```
def change(mylist):
    mylist = [1,2,3,4];
    print (" In Function: ", mylist)
    return

# Calling change function
mylist = [10,20,30];
change( mylist );
print ("Outside Function: ", mylist)
```

```
Output:
In Function: [1, 2, 3, 4]
Outside Function: [10, 20, 30]
```

Arbitrary Argument (*args)

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

- This way the function will receive a *tuple* of arguments, and can access the items accordingly:
- **Example:** If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Note: Arbitrary Arguments are often shortened to **args* in Python documentations.

Arbitrary Argument (*args)

- How can write a function that provides sum of its arguments. The number arguments can be varying like 1, 2, 3, 4, .. while calling of the function.
- **Example:**

```
def Sum(*Arg):
   sum = 0
   for i in range(len(Arg)):
      sum = sum + Arg[i]
    return sum

# Calling Sum Function
Sum_2 = Sum(4, 5)
Sum_3 = Sum(4, 5, 6)
Sum_4 = Sum(4, 5, 6, 7)
print(Sum_2, Sum_3, Sum_4)
```

Output:

```
9 15 22
```

Keyword Argument (kwargs)

- You can also send arguments with the **key = value syntax.**
- This way the order of the arguments does not matter.

Example:

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Note: *The phrase Keyword Arguments are often shortened to kwargs in Python documentations.*

Arbitrary Keyword Argument (**kwargs)

- If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.
- This way the function will receive a *dictionary* of arguments and can access the items accordingly.

Example:

```
def my_fun(**name):
  print("Last name is:", name["lname"])
  print("Full Name is:", name)

my_fun(fname ="Vijay",lname ="Rathor")
```

Output:

```
Last name: Rathor
Full Name: {'lname’: 'Rathor','fname’: 'Vijay'}
```

Note: *The phrase Keyword Arguments are often shortened to **kwargs in Python documentations.*

Using pass Statement In Function

- Function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example:
```
def myfunction():
  pass
```

Anonymous Function

- Function definitions

These 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.
- **Syntax:**

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

Anonymous Function

Example:

- Following is the example to show how *lembda* form of function works:

```
sum = lambda arg1, arg2: arg1 + arg2;
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

- This would produce following result:

```
Value of total : 30
Value of total : 40
```

Python Lambda Function

- A lambda function is a small anonymous function, which 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.
- **Example:** Add 10 to argument a and return the result.

```
x = lambda a : a + 10
print(x(5))   #15
```

- Lambda functions can take any number of arguments.
- **Example:** Multiply argument a with argument b and return the result.

```
x = lambda a, b : a * b
print(x(5, 6))    #30
```

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))    #13
```

Use of Lambda Function

- The power of lambda is better shown when you use them as an anonymous function inside another function.
- E.g. 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 this function definition to make a function that always *doubles* the number you send in.

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

mydoubler = myfunc(2)
print(mydoubler(11))
```

Use of Lambda Function

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

- Use the same function definition to make a function that always triples the number you send in. Or use the same function definition to make both functions, in the same program

```
Example: Triple

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

mytripler = myfunc(3)
print(mytripler(11))  #33
```

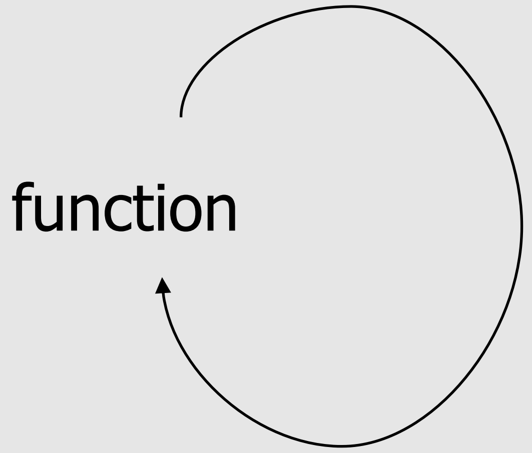
```
Example: Both Functions in same program
def myfunc(n):
  return lambda a : a * n

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

print(mydoubler(11))   #22
print(mytripler(11))   #33
```

***Note**: Use lambda functions when an anonymous function is required for a short period of time.*

Recursion In Programming

“A programming technique whereby a function calls itself either directly or indirectly.”

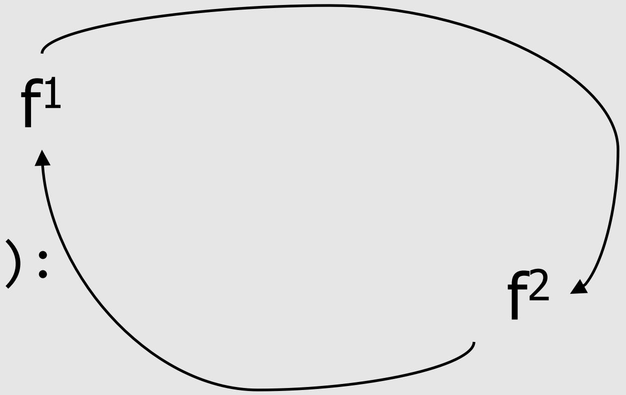
Direct Call



```
def fun ():

        …

    fun ()

        …
```

Indirect Call

```
def fun_f1 ():

        …

    fun_f2 ()

        …

def fun_f2 ():

        …

    fun_f1 ()

        …
```



Indirect Call

Name of the example program: `recursive.1py`

```
def fun1():
   fun2()

def fun2():
   fun1()

fun1()
```

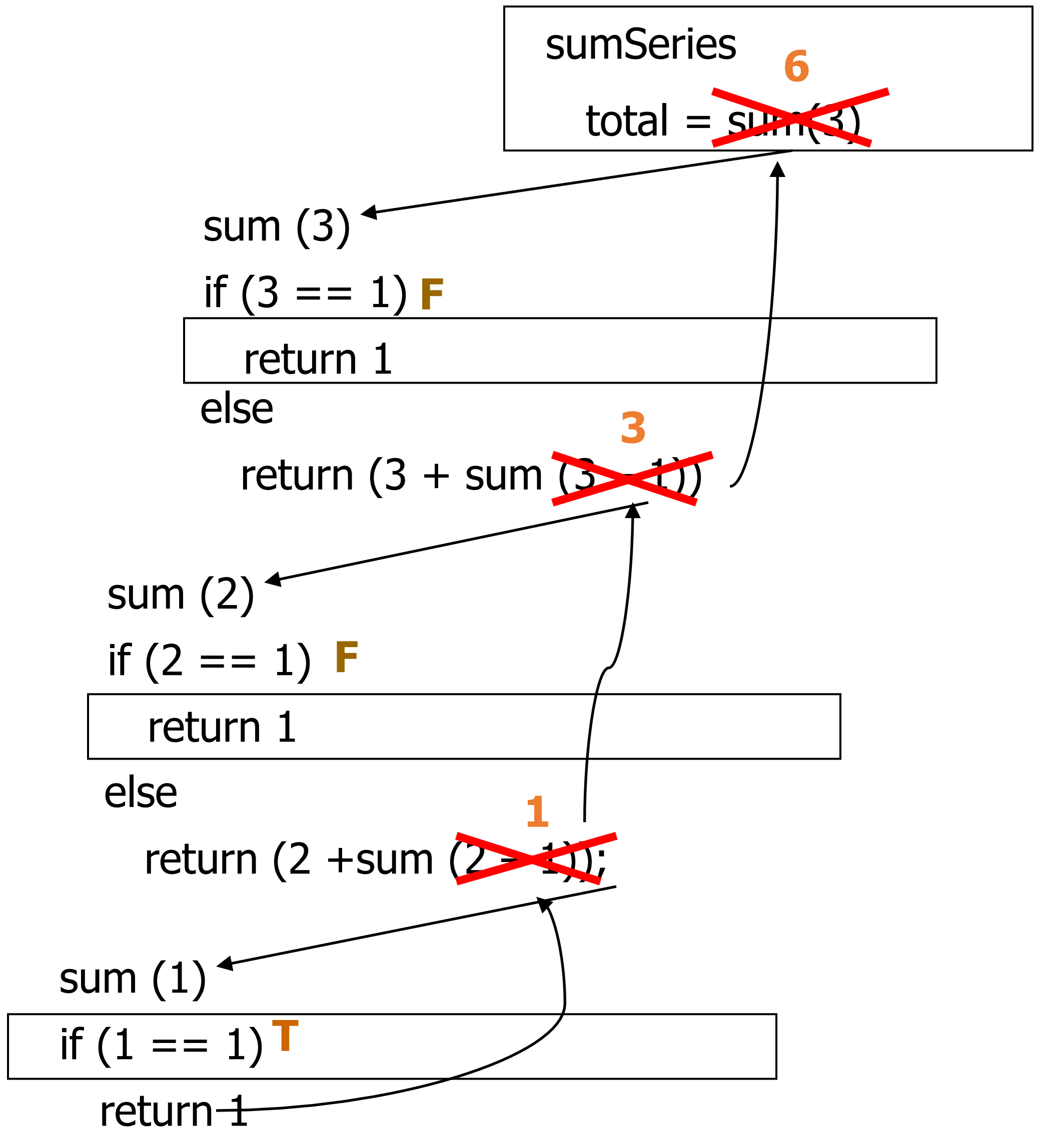
$$\begin{tikzcd}
f^1 \arrow[dr, bend left] & & & & \\
& f^2 \arrow[dr, bend left] & & & \\
& & f^3 \arrow[dr, bend left] & & \\
& & & \cdots \arrow[dr, bend left] & \\
& & & & f^n \arrow[uuuullll, bend left]
\end{tikzcd}$$

Requirements For *Sensible* Recursion

1) Base case
2) Progress is made (towards the base case)

Example Program: `sumSeries.py`

```
def sum(no):
   if (no == 1):
      return 1
   else:
      return (no + sum(no-1) )


def start():
   last = input ("Enter the last
                number: ")
   last = (int)last
   total = sum(last)
   print ("The sum of the series
           from 1 to", last, "is",
           total)

start()
```



When To Use Recursion

- When a problem can be divided into steps.
- The result of one step can be used in a previous step.
- There is a scenario when you can stop sub-dividing the problem into steps (step = recursive call) and return to a previous step.
  - Algorithm goes back to previous step with a partial solution to the problem (back tracking)
- All of the results together solve the problem.

When To Consider Alternatives To Recursion

- When a loop will solve the problem just as well
- Types of recursion (for both types a return statement is excepted)
    - **Tail recursion**
        - The last statement in the function is another recursive call to that function This form of recursion can easily be replaced with a loop.
    - **Non-tail recursion**
        - The last statement in the recursive function is not a recursive call.
        - This form of recursion is very difficult to replace with a loop.

Example: Tail Recursion

- Tail recursion: A recursive call is the last statement in the recursive function.
- Name of the example program: `tail.py`

```
def tail(no):
   if (no <= 3):
      print (no)
      tail(no+1)
   return()

tail(1)
```

Example: Non-Tail Recursion

- Non-Tail recursion: A statement which is not a recursive call to the function comprises the last statement in the recursive function.
- Name of the example program: `nonTail.p`

```
def nonTail(no):s
   if (no < 3):
      nonTail(no+1)
   print(no)
   return()

nonTail(1)
```

Drawbacks Of Recursion

Function calls can be costly

- Uses up memory
- Uses up time

Benefits of Using Recursion

- Simpler solution that's more elegant (for some problems)
- Easier to visualize solutions (for some people and certain classes of problems – typically require either: non-tail recursion to be implemented or some form of "backtracking")

Common Pitfalls When Using Recursion

- These three pitfalls can result in a runtime error
  - No base case
  - No progress towards the base case
  - Using up too many resources (e.g., variable declarations) for each function call

No Base Case

```
def sum(no):
    return(no + sum (no - 1))
```

No Base Case

When does it stop???

```
def sum (no):
  return (no + sum (no - 1))
```

No Progress Towards The Base Case

```
def sum (no):
    if (no == 1):
        return 1
    else:
        return (no + sum (no))
```

No Progress Towards The Base Case

```
def sum (no):
    if (no == 1):
        return 1
    else:
        return (no + sum (no))
```

The recursive case doesn’t make any progress towards the base (stopping) case

Using Up Too Many Resources

- Name of the example program: `recursiveBloat.py`

```
def fun(no):
    print(no)
    aList = []
    for i in range (0, 10000000, 1):
       aList.append("*")
    no = no + 1
    fun(no)

fun(1)
```

Using Up Too Many Resources

- Name of the example program: `recursiveBloat.py`

```
def fun(no):
    print(no)
    aList = []
    for i in range (0, 10000000, 1):
       aList.append("*")
    no = no + 1
    fun(no)

fun(1)
```

Using Up Too Many Resources

- Name of the example program: `recursiveBloat.py`

```
def fun(no):
    print(no)
    aList = []
    for i in range (0, 10000000, 1):
       aList.append("*")
    no = no + 1
    fun(no)

fun(1)
```

What will be the output?

```
def Do(x):
    if(x > 0):
            Do(x-1)
            print(x)
            Do(x-1)
    return

# Calling the Do
n =3
Do(n)
```

```
def X(n):
    if(n < 3):
            return 1
    else:
            return X(n-1) + X(n-3) + 1

n =5
val=X(X(n))
print(val)
```

Output: 1 2 1 3 1 2 1

Output: 17

Exercise

- Write a recursive function to find the nth terms of Fibonacci series.
- Write a recursive function to get the factorial of a given integer number.

```
def fib(int n):
     if(n ==0):
               return 0
   elif(n ==1):
               return 1
     else:
             return fib(n-1) + fib (n-2)

n =5
term= fib(n)
print(term)
```

```
def fact(n):
     if(n ==0 or n ==1):
               return 1
     else:
             return n*fact (n-1)

n =5
res= fact(n)
print(res)
```

Filter

- The filter() function returns an iterator where the items are filtered through a function to test if the item is accepted or not.

Syntax:

```
filter(function, sequence)
```

Parameter	Description
function	A Function to be run for each item in the iterable
iterable	The iterable to be filtered

- **Example:** Filter the array and return a new array with only the values equal to or above 18.

```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)

for x in adults:
  print(x)
```

Map

- The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Syntax: `map(function,  sequence)`

```
x = list(map(ord, ['A','B', 'C', 'D’]))
print(x)
```

Parameter	**Description**
function	Required. The function to execute for each item
iterable	Required. A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable.

- **Example:** Make new fruits by sending two iterable objects into the function:

```
def myfunc(a, b):
  return a + b

x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon', 'pineapple'))
```

Reduce

- The **reduce(fun, seq)** function is used to **apply a particular function passed in its argument to all of the list elements** mentioned in the sequence passed along. This function is defined in "**functools**" module.

Syntax: `map(function,  sequence)`

```
import functools

# initializing list
lis = [ 1 , 3, 5, 6, 2, ]

# using reduce to compute sum of list
print ("The sum of the list elements is : ",end="")
print (functools.reduce(lambda a,b : a+b,lis))

# using reduce to compute maximum element from list
print ("The maximum element of the list is : ",end="")
print (functools.reduce(lambda a,b : a if a > b else b,lis))
```

Output:

```
The sum of the list elements is : 17
The maximum element of the list is : 6
```

Reduce

- The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.
- If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.

Syntax: `zip(iterator1, iterator2, iterator3 ...)`

```
a = ("John", "Charles", "Mike")
b = ("Jenny", "Christy", "Monica", "Vicky")

x = zip(a, b)
print(list(x))
```

Output: `[('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica')]`

Thank You

?