NUS AI SUMMER EXPERIENCE 2019

FUNCTIONS

PAN BINBIN

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF ISEM

OUTLINE

01 DEFINITION AND SYNTAX

02 LOCAL AND GLOBAL VARIABLES

03 FUNCTIONS IN VARYING CONTEXT

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF ISEM

01 DEFINITION AND SYNTAX

DEFINITION OF FUNCTION

- Block of code that can be repeatedly called
- Called by accessing the function name
- Read in arguments, operates on arguments, returns a result
- Function are created to
  - reduce code duplication
  - modularize code

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF ISEM

SYNTAX OF FUNCTION

def function_name(arg1, arg2, . . .):
statements
return return-expression (optional)

Notes on the syntax

1. The def expression must end with colon :
2. The statements AFTER the def expression (the function block) must be indented
3. By default, function return None. Must have the return statement if you want function to return a result

SIMPLE FUNCTION

- How the function is used.
- Pass two numbers 1 and 2 as arguments
- The order of placement implies that you are passing: x =1, y=2
- You can also use the function this way
  oper_function(x=1,y=2)

```
def oper_function(x,y):
    summ = x+y
    product = x*y
    return summ, product

oper_function(1,2)
```

```
(3, 2)
```

- Function takes two arguments
- Applies two operators on the arguments
- Puts the results in two local variables, summ and product
- These two variables are local to the function – cannot be accessed outside the function
- A return statement gets the function to return the values of summ and product
- The return statement from the function gives this result in a tuple

SIMPLE FUNCTION: ASSIGNING VALUES

```
def oper_function(x,y):
    summ = x+y
    product = x*y
    return summ, product

add, multiplication = oper_function(1,2)
print('sum = ', add, 'product = ', multiplication)
```

```
sum =  3 product =  2
```

- Same function as previous slide
- Assign the output of return statement to two variables, add and multiplication
- Normal use of function
- Print out add and multiplication

SIMPLE FUNCTION: NO RETURN STATEMENT

- This is same code as before with the following exception
- The return statement is commented out, i.e., no longer executed

```
def oper_function(x,y):
    summ = x+y
    product = x*y
#    return summ, product    #comment out the return function

add, multiplication = oper_function(1,2)          #assign the value of summ to add, product to multiplication
print('sum = ', add, 'product = ', multiplication)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-f3292379aa20> in <module>()
      4 #    return summ, product    #comment out the return function
      5
----> 6 add, multiplication = oper_function(1,2)          #assign the value of summ to add, product to multiplication
      7 print('sum = ', add, 'product = ', multiplication)

TypeError: 'NoneType' object is not iterable
```

- Error
- No return statement so nothing is output from the function

- Try to assign output of the function to two variables, add and multiplication
- Print add and multiplication

DEFAULT VALUES FOR ARGUMENTS

- Possible to have default values in arguments
- Common in python library functions
- Default value used when no arguments passed during function call

```
def oper_function_default(x=10,y=20):
    summ = x+y
    minus = x-y
    return summ, minus

oper_function_default()
```

```
(30, -10)
```

- Default values are given for arguments
- Possible for only some arguments to have default values when default values are used

- Call function without passing arguments

- Default values are used in the operations within the function
  summ = 10 + 20
  minus = 10 -20

ARBITRARY NUMBER OF ARGUMENTS

- Possible case where number of arguments not known before hand
- Use the *args to denote the number of arguments to be passed
- *args allows any number of arguments (including 0)
- Example: a function to sum the numbers passed in.

```
def summVariable(*args):
    print("The arguments passed in are " + str(args))
    summ = 0
    for i in args:
        print("The argument referenced by i is " + str(i))
        summ += i
    return summ
```

- *args allow any number of arguments to be passed

- the for-loop runs until args is empty – all the numbers read in are used

- each number read in the iteration is added to the sum

- the output of the function is the sum of the numbers that are read in

EXAMPLE OF CALLING summVariable

```
summVariable(1,2,3,4)
```

```
The arguments passed in are (1, 2, 3, 4)
The argument referenced by i is 1
The argument referenced by i is 2
The argument referenced by i is 3
The argument referenced by i is 4
10
```

- print statement before the "for –loop"
- print statement during the 1st iteration of the "for –loop"
- print statement during the 4th (last) iteration of "for –loop"
- output from the return statement of the summVariable function

02 LOCAL AND GLOBAL VARIABLES

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF ISEM

LOCAL VARIABLE AND GLOBAL VARIABLE

- Variables created inside a function (within the def statement) are local to the function
- These local variables is available for use inside the function only
- Variables created outside ALL functions are global to that particular module (program)
- There are no global variables that covers all possible modules that you write

ILLUSTRATION OF LOCAL VARIABLE AND GLOBAL VARIABLE

```
x_global = 'Global Variable x'

def print_global_local(anything):

    y_local = anything
    print ('outside function = ', x_global,'.', 'inside function = ', y_local)

print_global_local('passing this to function')
```

```
outside function =  Global Variable x . inside function =  passing this to function
```

- define global variable outside of any function
- define a function that takes in one argument with name anything
- define a local variable within the function and assign it the value that you pass in
- function prints out both the value of the global and local variable
- call the function and pass it an argument
- the function was able to use both the global variable (x_global) that was defined outside the function and the local variable (y_local) defined within the function

GLOBAL VARIABLE USED ANYWHERE IN MODULE
LOCAL VARIABLE CANNOT BE USED OUTSIDE FUNCTION

```
print(x_global)
```

```
Global Variable x
```

- print global variable outside function
- no problem

```
print(y_local)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-c1465b24b208> in <module>()
----> 1 print(y_local)

NameError: name 'y_local' is not defined
```

- print local variable outside function
- error – variable deemed as not defined although it was defined inside a function

LOCAL VARIABLE AND GLOBAL VARIABLE GIVEN SAME NAME

- Global variable unchanged anywhere outside function
- Assumes value assigned to it within the function inside the function

```
x_global = 'Global Variable x'
def print_local(EPP):
    x_global = EPP
    print ('Value of x_global inside function is: ', x_global)

print_local('Mary had a little lamb')
print('Value of x_global outside the function remains unchanged.  It is: ', x_global)
```

- a global variable x_global defined
- a local variable x_global defined
- value of x_global printed out within function
- pass this string to the function
- x_global inside function will take this value

```
Value of x_global inside function is:  Mary had a little lamb
Value of x_global outside the function remains unchanged.  It is:  Global Variable x
```

- output from the print_local function.
- Value of x_global

- output from the print statement outside the function.
- Value of x_global

03 FUNCTIONS IN VARYING CONTEXT

FUNCTION AS ARGUMENTS

- The argument of a function can be another function
- Takes the result of another function for its procedure
- Easier to read compared to embedding a number of conditional procedures in one function

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF ISEM

FUNCTION AS ARGUMENTS EXAMPLE

- Suppose you want to write a function to compute the cost of one of two possible materials: tiles or concrete in renovation projects
- We can write a generic function to compute the cost:
  - Cost = Quantity x Unit Cost
- However, Quantity itself has to be computed by
  - Quantity = (Area of Floor to be tiled)/(Area of tile) # for tiles
  - Quantity = (Area of Floor to be concreted)*(Thickness of concrete) # for concrete
- You can see that we can write two functions: one for computing the cost of tiles and the other for computing cost of concrete
- The user have to use either one of the two functions.
- We can simplify the task for the user by writing one extra function that is called no matter whether the cost of tile or cost of concrete is required
- This extra function will have a function as its argument

FUNCTION AS ARGUMENTS EXAMPLE

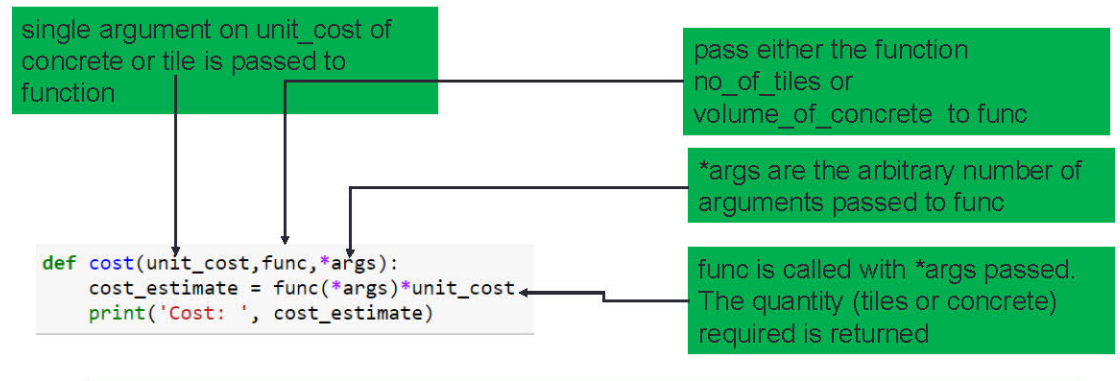
```
def no_of_tiles(area_of_floor,area_of_tile):
    no_of_tiles_needed = round(((area_of_floor/area_of_tile)*1.1)+0.5)
    print ('Number of tiles needed: ', no_of_tiles_needed)
    return no_of_tiles_needed

def volume_of_concrete(area_of_floor, thickness_of_concrete):
    vol_of_concrete_needed = round(((area_of_floor* thickness_of_concrete)*1.2)+0.5)
    print ('Volume of concrete needed: ', vol_of_concrete_needed)
    return vol_of_concrete_needed
```

Two functions above for computing quantity needed

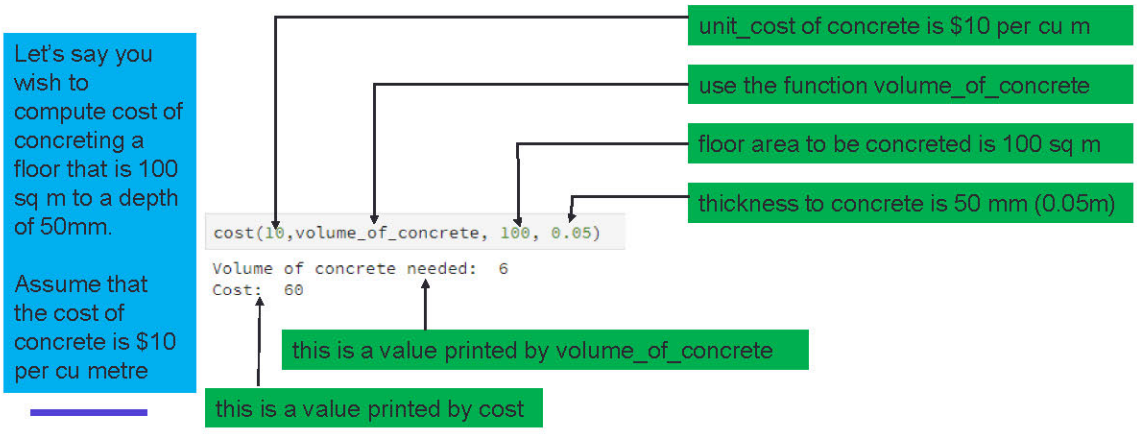
Each has two arguments

Each return the quantity needed

FUNCTION AS ARGUMENTS EXAMPLE



FUNCTION AS ARGUMENTS EXAMPLE USING THE COST FUNCTION



NESTED FUNCTIONS

- Functions can be nested within (defined inside) another function
- This nested function is called an inner function
- The function enclosing the nested function is called the outer function
- The inner function can access variables from the outer function
- The structure of nesting makes the outer function a function factory or function generator

NESTED FUNCTIONS EXAMPLE

Suppose you want to make a function for a multiplication table of x:

1 x (x)
2 x (x)
and so on.

You want to be able to automatically create the x multiplication function for whatever value of x.

You can use a nested function to do so.

The outer function, make_multiplier, creates another function, multiply, that is a multiplication table of x

```
def make_multiplier(x):
    def multiply(y):
        return x*y
    return multiply
```

outer function: take argument x which will be used by inner function

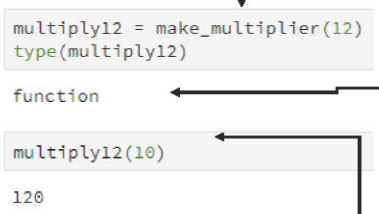
inner function take argument y

inner function, multiply returns a value, a multiplication of x and what is recently passed to it, y.

outer function returns multiply which is a function. At the same time, makes x available for use in multiply function

NESTED FUNCTIONS EXAMPLE

Here, make_multiplier is used to create a function, called multiply12, that multiplies any number by 12



the number 12 is passed as argument to the make_multiplier function and assign result to a variable called multiply12.

multiply12 is another function.

multiply12 acts like a calculator for 12x table. Pass another value to it and the value is multiplied by 12.

muyltiply12 retains the value of 12 that was passed to its outer function.

RECURSIVE FUNCTIONS

- Recursive function calls itself one or more times
- It makes code clean and elegant
- Breaks down complex tasks into simpler sub-tasks
- Sequence generation easy with recursive function

RECURSIVE FUNCTION EXAMPLE

Simple function to compute factorial of a number n.

n! = n x (n-1) x (n-2) x . . . 1

```
def factorial(n):
    if n == 1:
        return 1
    return n*factorial(n-1)
```

this is stopping condition for recursion.

all recursive function must have stopping condition

the function calls itself pass a different argument each time

CALLING A RECURSIVE FUNCTION

This is what happens when factorial(4) is executed

iteration 1, factorial(4), n = 4, return 4*factorial(3)
iteration 2, factorial(3), n = 3, return 3*factorial(2)
iteration 3, factorial(2), n = 2, return 2*factorial(1)
iteration 4, factorial(1), n = 1, return 1 (the if statement executed)

So putting all the iterations together, the recursive equation becomes:

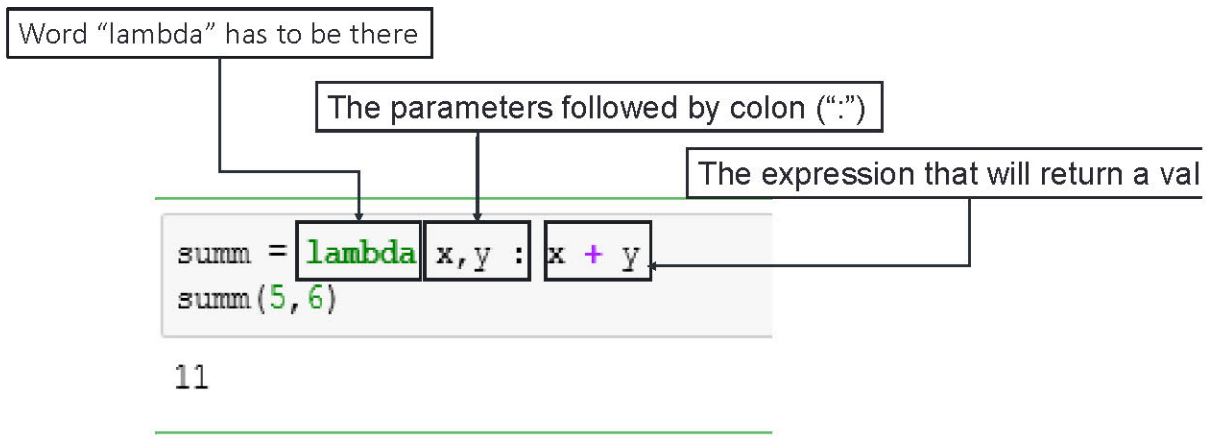
4 x factorial(3)
= 4 x 3 x factorial(2)
= 4 x 3 x 2 x factorial(1)
= 4 x 3 x 2 x 1

```
factorial(4)

24
```

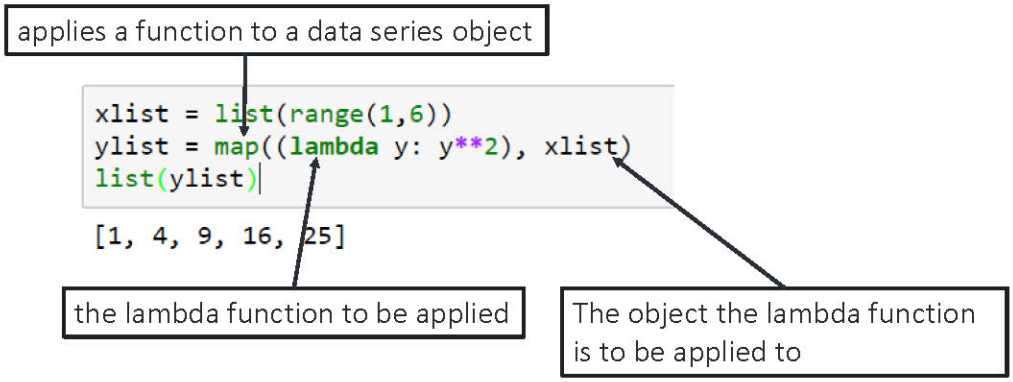
LAMBDA FUNCTIONS

- Un-named, anonymous functions
- Defined and called at point of use
- Always contains an expression that returns a value

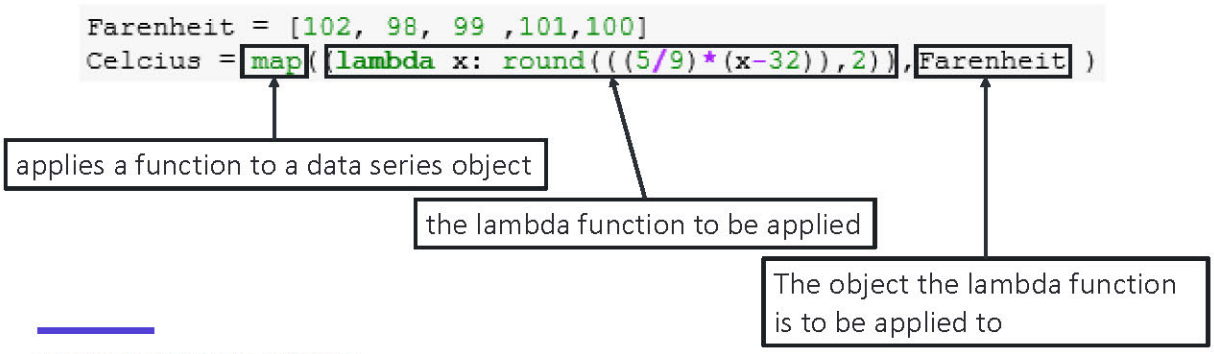
SYNTAX OF LAMBDA FUNCTIONS



USUAL APPLICATION OF LAMBDA FUNCTIONS



USUAL APPLICATION OF LAMBDA FUNCTIONS



PAN BINBIN

BINBIN.PAN@NUS.EDU.SG

NUS National University of Singapore

NUS AI SUMMER EXPERIENCE 2019

FUNCTIONS

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF ISEM