Python – Functional

Learn to Code with Python

Functional: Higher Order Functions as Arguments

- A decorator enhance a function with additional features without changing its core functionality
- Higher Order Function – accepts a function as an argument and/or returns a function as a return value
- Example
  - def one():
    - return 1
  - print(type(one))   // No () so the function will not be executed. The type is <class ‘function’>
    - type is an example of a higher order function. It is accepting the one function
- Example
  - def add(a,b):
    - return a + b
  - def sub(a,b):
    - return a - b
  - def calculate(func, a, b):
    - return = func(a,b)
  - print(calculate(add,3,5))   → print 8

Functional: Nested Function

- Example
  - def convert_gallons_to_cups(gallons):
    - def gallons_to_quarts(gallons)
      - print(f”Converting {gallons} to quarts”)
      - return gallons * 4
    - def quarts_to_pints(quarts):
      - print(f”Converting {quarts} quarts to pints”)
      - return quarts * 2
    - def pints_to_cups(pints):
      - print(f”Converting {pints} to cups”)
      - return pints * 2
    - quarts = gallons_to_cups(gallons)
    - pints = quarts_to_pints(quarts)
    - cups = pints_to_cups(pints)
    - return cups
  - print(convert_gallons_to_cups(4)) which prints 64

Functional: Higher Order Functions II: Functions as Return Values

- Example
  - def calculator(operation):
    - def add(a, b):
      - return a + b
    - def subtract(a, b):
      - return a - b
    - if (operation == “add”):
      - return add
    - elsif (operation == “subtract”):
      - return sub
  - print(calculator(“add”)(10,4))   // Returns 14
  - print(calculator(“subtract”)(7,7)   // Returns 0
- Example
  - def add(a, b):
    - return a + b
  - def subtract(a, b):
    - return a - b

Functional: Higher Order Functions II: Functions as Return Values

- Example
  - def square(num):
    - return num ** 2
  - def cube(num):
    - return num ** 3
  - def times10(num)
    - return num * 10
  - operations = [square, cube, times10]
  - for func in operations:
    - print(func(5)); print 25, 125, 50 on three different lines
- What is wrong with the code : In the calculate() the value of func(a,b) is not returned
  - def multiply(a,b):
    - return a * b
  - def divide(a,b):
    - return a / b
  - def calculate(func, a, b):
    - func(a,b)

Decorators : Scope 1: Global vs Local Variables

- The locations in a program in which a variable/function can be used
- Global Scope : A variable assigned outside a function , but inside a file
- local Scope : A variable assigned inside a function
- Example
  - age = 28
  - def fancy_func():
    - print(age)   // Will print 28
- Example
  - age = 28
  - def fancy_func():
    - age = 100
    - print(age)   // The age from the function body will be printed out
  - fancy_func()
  - print(age)   // prints the global age
- Shadow Variable: A local variable that shares the same name as the global variable

Decorators: Scope 1: Global vs Local Variables

- Many times global variables will be used to declare constants
- Example
    - TAX_RATE = .06
    - def calculate_tax(price):
        - return(round(price * TAX_RATE, 2)
- Example
    - egg_count = 0
    - def buy_eggs():
        - egg_count += 12
    - buy_eggs
    - When executed get the UnboundLocalError: local variable 'egg_count' referenced before assignment
        - When modifying a variable it must be local scope
        - Python does not allow functions to modify variables that are not in the functions’s scope

Decorators: Scope 2 : The LEGB Rule

- (L)ocal/ (E)nclosing Functions/ (G)lobal / (B)uiltin
- Python will search through the scope in order to find the name
- If Python does not find the name then it raise a NameExcpetion
- Example
  - def outer():
    - x = 10
    - def inner():
      - x = 5
      - return x
    - return inner()
  - print(outer())   prints x = 5 since is in the local scope for inner
- Example
  - def outer():
    - x = 10
    - def inner():
      - return x
    - return inner()
  - print(outer())   prints x = 10 since x does not exist in the local scope of inner, but the global.

Decorators: Scope 2 : The LEGB Rule

- Example
  - x = 15
  - def outer():
    - def inner():
      - return x
    - return inner()
  - print(outer())   // prints x = 10 since in the Local Scope, Enclosing Scope, but the Global Scope
- Example
  - x = 15
    - def outer():
      - def inner():
        - return x
      - return inner()
      - print(outer())   // prints len since in the Local Scope, Enclosing Scope, Global Scope, Built In Scope

Decorators: The LEGB Rule

- Example – The global variable was defined after it has been invoked.
  - def a():
    - def b():
      - def c():
        - return val
      - return b
    - print(a())   // val is defined after it has been accessed
    - val = “Hello”

Decorators: The Global Keyword

- Allows use to introduce global scope into local scope
- Global Key will create a global variable
- Strongly discouraged to use global in a local function since the value of a variable now depends on the order of the function calls
- Example
  - x = 10
  - def change_stuff():
    - x = 15
  - print(x) // Print x = 10
  - change_stuff()
  - print(x) // Print x = 10
- Example
  - x = 10
    - def change_stuff():
      - **global** x
      - x = 15
    - print(x)   // Print x = 10
    - change_stuff()
    - print(x)   // Print x = 15

Decorators: The Nonlocal Keyword

- Can only be used inside the body of a nested function
- Applies to the same affect a global , but does it to variables enclosing function scope
- Example Use : Have a function that gets passed in the name variables then have enclosed function that is responsible for providing the value if the value of name is null
  - In the sub function the global name would be changed by include nonlocal name and then changing the name
- example
  - def outer():
    - bubble_tea_flavor = “Black”
    - def inner():
      - bubble_tea_flavor = “Taro”
    - inner()
    -
    - return bubble_tea_flavor
  - print(outer())   // Get Black
- example
  - def outer():
    - bubble_tea_flavor = “Black”
    - def inner():
      - **nonlocal** bubble_tea_flavor   // Stick with the bubble_tea_variable in the enclosing function scope
      - bubble_tea_flavor = “Taro”
    - inner()
      - return bubble_tea_flavor
    - print(outer())   // Get Tario

Decorators: Intro to Decorators

- A decorator enhance a function with additional features without changing its core functionality
  - Example time a function → Start the timer, run the function, end the timer
    - Reusable with any other function
  - Whenever you need to execute business logic before or after a business. For example (before a function check if the user is logged in.)
    - Remember each function can use the @<function name> to specify decorate function (see in the “Intro to Decorators (may be on another page))
- Example
  - You have wrapped the fn() to include bonus functionality
  - def be_nice(fn):
    - def inner_function():
      - print(“Nice to meet you”)
      - fn()
      - print(“It was my pleasure to meet you”)
    - return inner_function
  - def complex_business_logic():
    - print(“something complex!”)
  - print(be_nice(complex_business_logic)
- Example – Invoke a function inline
  - be_nice(complex_business_logic)()

Decorators: Intro to Decorators

- Alternative Syntax use @Decorator
  - Advantage can be reused multiple times.
- can reuse the decorator as many times as you want
- Example
  - @be_nice   // executes be_nice(complex_business_logic)
  - def complex_business_logic():   // The wrapper function must be a wrapper function that accepts one function and returns another function
    - print(“Something complex”)
    -
  - @be_nice
  - def another_fancy_function()
  -
  - complex_business_logic()
  - another_fancy_function()

Decorators: Intro to Decorators

- A decorator enhance a function with additional features without changing its core functionality
  - Example time a function → Start the timer, run the function, end the timer
    - Reusable with any other function
  - Whenever you need to execute business logic before or after a business. For example (before a function check if the user is logged in.)
    - Remember each function can use the @<function name> to specify decorate function (see in the “Intro to Decorators (may be on another page))
- Example
  - You have wrapped the fn() to include bonus functionality
  - def be_nice(fn):
    - def inner_function():
      - print(“Nice to meet you”)
      - fn()
      - print(“It was my pleasure to meet you”)
    - return inner_function
  - def complex_business_logic():
    - print(“something complex!”)
  - print(be_nice(complex_business_logic)
- Example – Invoke a function inline
  - be_nice(complex_business_logic)()

Decorators: Intro to Decorators

- Alternative Syntax use @Decorator
  - Advantage can be reused multiple times.
- can reuse the decorator as many times as you want
- Example
  - @be_nice // executes be_nice(complex_business_logic)
  - def complex_business_logic(): // The wrapper function must be a wrapper function that accepts one function and returns another function
    - print(“Something complex”)
    -
  - @be_nice
  - def another_fancy_function()
  -
  - complex_business_logic()
  - another_fancy_function()

Decorators: The Global Keyword

- Allows use to introduce global scope into local scope
- Global Key will create a global variable
- Strongly discouraged to use global in a local function since the value of a variable now depends on the order of the function calls
- Example
  - x = 10
  - def change_stuff():
    - x = 15
  - print(x) // Print x = 10
  - change_stuff()
  - print(x) // Print x = 10
- Example
  - x = 10
    - def change_stuff():
      - **global** x
      - x = 15
    - print(x) // Print x = 10
    - change_stuff()
    - print(x) // Print x = 15

Decorators: The Nonlocal Keyword

- Can only be used inside the body of a nested function
- Applies to the same affect a global , but does it to variables enclosing function scope
- Example Use : Have a function that gets passed in the name variables then have enclosed function that is responsible for providing the value if the value of name is null
    - In the sub function the global name would be changed by include nonlocal name and then changing the name
- example
    - def outer():
        - bubble_tea_flavor = “Black”
        - def inner():
            - bubble_tea_flavor = “Taro”
        - inner()
        -
        - return bubble_tea_flavor
    - print(outer())   // Get Black
- example
    - def outer():
        - bubble_tea_flavor = “Black”
        - def inner():
            - **nonlocal** bubble_tea_flavor   // Stick with the bubble_tea_variable in the enclosing function scope
            - bubble_tea_flavor = “Taro”
        - inner()
            - return bubble_tea_flavor
        - print(outer())   // Get Tario

Decorators: Arguments with Decorator Funcitons (*args, **kwargs)

- Example – Showing the problem
  - def be_nice(fn):
    - def inner_function():
      - print(“Nice to meet you”)
      - fn()
      - print(“It was my pleasure to meet you”)
    - return inner_function
  - @be_nice // See previous slides
  - def complex_business(stackholder):
    - print(f”Something complex for {stackholder}”)
  - complex_business(“Boris”) // Get a type error inner() takes 0 positional arguments, but 1 was given
    - Problem inner_function is not expecting an argument
    - Don’t add a parameter to function to keep it generic. Another call to be_nice might have two arguments.
    - solution accepts *args, *kwargs

Decorators: Arguments with Decorator Funcitons (*args, **kwargs)

- Example – Showing the solution adding *args, **kwarg
  - def be_nice(fn):
    - def inner_function(*args, **kwargs):
      - print(“Nice to meet you”)
      - fn() // With inserting *args, **kwargs as parameters into innner function there is a new error
      - print(“It was my pleasure to meet you”)
    - return inner_function
  - @be_nice // See previous slides
  - def complex_business(stackholder):
  - print(f”Something complex for {stackholder}”)
  - complex_business_logic(“Borris”) // The value Borris would be stored in *args
  - complex_business_logic(stakeholder = “Borris”) // The value Borris would be stored in **kwargs as { “stackholder”:”Borris” }

Decorators: Arguments with Decorator Funcions (*args, **kwargs)

- Example – Showing the solution adding *args, **kwarg
  - def be_nice(fn):
    - def inner_function(*args, **kwargs):
      - print(“Nice to meet you”)
      - fn(*args, **kwargs)
      - print(“It was my pleasure to meet you”)
    - return inner_function
  - @be_nice // See previous slides
  - def complex_business(stakeholder):
    - print(f”Something complex for {stackholder}”)
  - complex_business_logic(“Borris”) // The value Boris would be stored in *args
  - complex_business_logic(stakeholder = “Borris”) // The value Boris would be stored in **kwargs as { “stackholder”:”Boris” }

Decorators: Returned Values from Decorated Funcitons

- Example Return statement
  - def be_nice(fn):
    - def inner_function(*args, **kwargs):
      - print(“Nice to meet you”)
      - result = fn(*args, **kwargs) // If we returned here then the it would not execute the next print
      - print(“It was my pleasure to meet you”)
      - return result
    - return inner_function
  - @be_nice // See previous slides
  - def complex_business(a,b):
    - return a + b
  - complex_business_logic(a=3,b=5) // Returns None. Fix : Capture the variable in result and return from inner function

Decorators: The functools wrap Decorator

- Doc String – String at the top of function serve as documentation for the function
  - through the help function
  - def complex_business(a,b):
    - “Add two numbers together”
    - return a + b
  - help(complex_business_sum)   // prints out “Add two numbers together”
- Example Return statement where the help documentation is now what you expect
  - def be_nice(fn):
    - def inner_function(*args, **kwargs):
    - print(“Nice to meet you”)
    - result = fn(*args, **kwargs)   // If we returned here then the it would not execute the next print
    - print(“It was my pleasure to meet you”)
    - return result
  - return inner_function
- @be_nice   // See previous slides
- def complex_business(a,b):
  - “Add two numbers together”
  - return a + b
- complex_business_logic(a=3,b=5)
- help(complex_business_sum)   // prints out inner(*args, **kwargs) instread of “Add two numbers together”

Decorators: The functools wrap Decorator

- Example getting the correct help string
- import functools
- def be_nice(fn):
  - @functools.wraps(fn)
  - def inner_function(*args, **kwargs):
    - print(“Nice to meet you”)
    - result = fn(*args, **kwargs)   // If we returned here then the it would not execute the next print
    - print(“It was my pleasure to meet you”)
    - return result
  - return inner_function
- @be_nice   // See previous slides
- def complex_business(a,b):
  - “Add two numbers together”
  - return a + b
- complex_business_logic(a=3,b=5)
- help(complex_business_sum)   // Due to the addition of functools.wraps “Add two numbers together”

Functional : Scope 3 : Closures

- A programming pattern in which a scope retains access to an enclosing scope’s names even if the enclosing scope no longer exist
- Example
  - def outer():
    - candy = “snickers”
    - def inner():
      - return candy
    - return inner()   // Returns Snickers
  - print(outer())
  - Explanation
    - The inner function has access to the candy variable that is defined in the enclosing scope.
    - The inner scope retains access to the outer level scope and its names.
    - When we invoke the outer function we get back the invocation of the inner()
  - Example
    - the func = outer()   // After the outer function Python will be thrown out, however there is still a reference to candy in this un invoked function.
    - candy still exist even thought the function has been garbage collected.
- Closure is the scope where the data or variables will be executed