Lab Manual

September 6, 2024

1 Software Construction and Development Lab

Muhammad Saood Sarwar

Instructor (CS), National University Of Computer and Emerging Sciences, Peshawar.

1.0.1 Python Functions

A function is a block of code which only runs when it is called.

You pass parameters to a function and function return data as a result.

Function in Python always return something

Fuction in python can returns multiple value (In C++ and Java, you cannot directly return multiple values as a built-in feature. However, to return multiple values a collection of objects of class or struct can be used)

To return value use keyword 'return'

2 Best Practices for Functions in Software Construction

It's important to emphasize several best practices and construction rules to ensure clean, maintainable, and efficient code. Here are some key points to cover:

2.0.1 1. Single Responsibility Principle (SRP)

Each function should have a single, well-defined responsibility. This makes the function easier to understand, test, and maintain.

2.0.2 2. Function Naming

Names should be descriptive and clearly reflect what the function does. Avoid vague names like `doSomething()`; instead, use `calculateSum()` or `fetchUserData()`.

2.0.3 3. Parameter Count

Limit the number of parameters passed to a function. Ideally, a function should have between 0 to 3 parameters. If more are needed, consider grouping related parameters into a structure (like a tuple, dictionary, or class).

2.0.4 4. Avoid Side Effects

Functions should avoid changing variables or states outside their scope. This makes them easier to debug and reuse.

2.0.5 5. Pure Functions

Where possible, use pure functions—functions that always return the same result for the same inputs and do not depend on or alter external state.

2.0.6 6. Avoid Long Functions

Keep functions short and focused. Long functions are harder to read, understand, and maintain. If a function exceeds around 20-30 lines, consider splitting it into smaller helper functions.

2.0.7 7. DRY Principle (Don't Repeat Yourself)

Avoid duplicating code across functions. If a task is repeated, create a reusable function for it.

2.0.8 8. Error Handling

Ensure functions handle potential errors gracefully, either through exceptions or proper validation of inputs.

2.0.9 9. Testability

Functions should be easy to test in isolation. Pure functions without dependencies on external states are particularly good candidates for unit testing.

[318]:
```python
def my_function():
    print("This is my firt function in Python")
    return 0.0

ret_value = my_function()
print(ret_value)
```

```
This is my firt function in Python
0.0
```

[324]:
```python
def my_function():
    print("This is my firt function in Python")
    return 66
    #return None     def __str__ (self):
        return self.name + self._age + self.__gender
```

[325]:
```python
ret_value = my_function()
```

```
This is my firt function in Python
```

[326]:
```python
print(ret_value)
```

```
66
```

[ ]:
```
# Parametrized method and Number of Arguments

# Single argument
def my_function1(name):
    print("Hello,", name)

# Two arguments
def my_function2(message, name):
    print(message , name)
```

[ ]:
```
my_function1("World")
my_function2("Hello,", "World")
```

[328]:
```
# Arbitrary Arguments, *args
# Add a * for arbitrary arguments before the parameter name in the function␣
 ↪definition .
# This way the function will receive arguments as tuples

def my_function(*args):
    print(type(args))

    for i in range(len(args)):
        print(args[i])

my_function("Ali", 40, "Street#123", "BCS")
```

```
<class 'tuple'>
Ali
40
Street#123
BCS
```

[344]:
```
def sum(*args):
    sum = 0
    for arg in args:
        sum += arg
    return sum
```

[345]:
```
sum_val = sum(1,2,3,4.7,8,8)
print(sum_val)
```

```
26.7
```

[ ]:
```
```

[351]:
```
# Keyword Arguments, You can also pass argument with its name
# Here order of the arguments passed is not necessary.
```

```
def my_function(name, age, address):
    print(name, age, address)


my_function(name = "Ali", address = "Street#134", age = 20)
```

```
Ali 20 Street#134
```

[ ]:

```
```

[364]:

```
# Arbitrary Keyword Arguments, **kwargs

def my_function(**args):
    #print(args['name'])
    #print(args['age'])
    #print(args['address'])

    for index, arg in args.items():
        print(index, arg)

my_function(name="Ali", age=40, address="Street#123")
```

```
name Ali
age 40
address Street#123
```

[ ]:

```
```

[375]:

```
# Default Parameter Value
# If a function with default arguments is called without arguments, it uses the␣
 ↪default values
# Default argument should not be followed by non-default arguments

def my_function(age, name='No Name', address="No Address"):
    print(name)
    print(age)
    print(address)

my_function(40)
```

```
No Name
40
No Address
```

[ ]:

```
# Passing a List as an Argument
def my_function(values):
    print(values)

my_function(['Ali',30,'Adress']) # Here we are passing a list collection
```

[ ]:
```
# Function definitions cannot be empty, but if you want to have a function␣
↪definition without content,
# Use the pass statement
```

[ ]:
```
def my_function():
    pass

my_function()
```

[ ]:
```

```

2.1 Recursion

Recursion is when a statement in a function calls itself repeatedly

Recursion is a common programming concept.

[67]:
```
#Calculate Factorial using loops
def calculateFact(n):
    fact = 1
    for i in range(1, n+1):
        fact = fact * i;    # fact = fact + fact * i ==> 5! = 4! + 4!*4
    return fact;
print(calculateFact(5))
```

```
120
```

[68]:
```
#Calculate Factorial using recursion
def calcFact(n):
    if n == 1:
        return 1
    else:
        return n*calcFact(n-1)

calcFact(5)
```

[68]:
```
120
```

[69]:
```
#Fibonacci Series
# 0, 1, 1, 2, 3, 5, 8, 13, 21
def fib(n):
    if(n <= 1):
        return 0

    elif(n == 2):
        return 1

    else:
        return fib(n-2) + fib(n-1)
        def __str__ (self):
        return self.name + self._age + self.__gender
n = 1
print("\nFibonacci number at position ", n, " is", fib(n))
```

```
Fibonacci number at position  1  is 0
```

[70]:
```
# Combination
def comb(n, r):
    if r <= 1:
        return n
    elif r == n:
        return 1

    else:
        return comb(n-1, r-1) + comb(n-1, r)

n = 5
r = 2
print("\nCombiation of ", n, " C", r, " = ", comb(n,r))
```

```
Combiation of  5  C 2  =  10
```

[71]:
```
# Permutation
def perm(n, r):
    if r <= 1:
        return n
    elif r == n:
        return 1

    else:
        return perm(n-1, r-1) + perm(n-1, r)

n = 5
r = 2
print("\nPermutation of ", n, " C", r, " = ", perm(n,r)*calcFact(r))
```

```
Permutation of  5  C 2  =  20
```

[ ]:
```

```

[72]:
```
#Sum integer nubmer to specific range using recursion
def sum(n):
    if(n <= 1):
        return 1
    else:
        return n + sum(n-1)

print("\nSum: ", sum(5))
```

```
Sum:  15
```

2.2 Python Lambda

A lambda function is a small anonymous function

A lambda function can take any number of arguments, but can only have one expression

[198]:
```
x = lambda a, b : a + b

val = x(5,15)
print(val)
```

```
20
```

[204]:
```
#Creating dynamically double, thrice etc of a number method

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

lamda_func = myfunc(7) # Here the method is created dynamically, which will␣
↪double/ thrice of every number
# lamda_func = lambda a : 3 * a               # Actually, this happens

print(lamda_func(9))
```

```
63
```

[206]:
```
thrice = myfunc(3)
```

[207]:
```
thrice(12)
```

[207]: 36

[2]:
```
#Equivalent Code for above lambda
def myfunc(n):
    def myfunc2(a):
        return n * a
    return myfunc2

multiplier_func = myfunc(3)
print(multiplier_func(11))

# Note: In C++ and Java, you cannot define a function within another function
```

```
33
```

[215]:
```
#Creating dynamically square, cube etc method

def myfunc2(n):
    return lambda a : n ** a  # Power function

lamda_func2 = myfunc2(5) # Here the method is created dynamically, which will␣
 ↪calculate power of every number
# lamda_func = lambda a : 3 ** a               # Actually, this happens


print(lamda_func2(4))
```

```
625
```

2.3 Python Classes and Objects

Python is an object oriented programming language.

Almost everything in Python is an object.

A Class is a template or blueprint for creating objects.

3 Classes Should Be Small!

The principle of keeping classes small is fundamental to effective software design. A small class adheres to the Single Responsibility Principle (SRP) and exhibits high cohesion, making it easier to understand, maintain, and extend. Here's how to apply these concepts:

3.0.1 Single Responsibility Principle (SRP)

A class should have only one reason to change, meaning it should manage only one aspect of functionality. By adhering to SRP, you ensure that each class has a clear and focused responsibility, which simplifies understanding and modifications. For example, a `UserAuthenticator` class that solely handles user authentication is more manageable than a `UserManager` class that tries to handle authentication, user details, and session management.

3.0.2 Cohesion

Cohesion refers to how closely related and focused the responsibilities of a class are. A highly cohesive class has methods and attributes that are strongly related to its core purpose. High cohesion is a sign that a class is well-designed and adheres to SRP. For instance, a `ReportGenerator` class with methods for generating and formatting reports exhibits high cohesion, whereas a class with methods for report generation, user management, and data storage would have low cohesion.

3.0.3 Concise Naming

The name of a class should clearly reflect its responsibility. If a class name is too vague or includes ambiguous terms like "Processor," "Manager," or "Super," it might indicate that the class is trying to handle too many responsibilities. Instead, choose specific names that convey the exact purpose of the class. For example, a class named `InvoiceProcessor` is more descriptive and focused than one named `SuperDashboard`.

3.0.4 Descriptive Summary

Aim to describe a class in a brief, precise statement—ideally within 25 words. Avoid using conjunctions like "if," "and," "or," or "but" to keep the description focused. For example, a well-defined class description might be: "The `OrderHandler` manages the processing of customer orders and updates inventory levels." This clarity indicates a class with a focused responsibility.

3.0.5 Avoid Aggregation

Classes with names that include terms like "Super" or "All-In-One" often indicate that they have accumulated multiple responsibilities. Break these classes down into smaller, more focused classes. For example, instead of a `SuperDashboard` that handles various unrelated functions, create distinct classes for managing components, tracking version numbers, and handling user interactions.

3.0.6 Responsibility Segregation

Evaluate whether each class handles only one type of responsibility. If a class is responsible for multiple tasks, consider refactoring it into smaller, more specialized classes. This segregation enhances clarity and reduces the complexity of managing and modifying the code.

[226]:
```python
# __init__() Function
class MyClass:
    def __init__(self, name, age=0, address='No Address'):
        #instance variables
        self.name = name
        self.age = age
        self.address = address

    #self.name = name ## Not allowed

my_obj = MyClass('Khan', 43, 'Hayatabad')
my_obj.address

# You cannot define instance variables outside of methods or the constructor
# directly in a Python class
```

[226]:
```
'Hayatabad'
```

[ ]:
```
```

[231]:
```python
# Access private attribute with class method

class MyClass:
    def __init__(self, name, age=0, address='No Address'):
        self.name = name             #Public
        self._age = age              #Protected
        self.__address = address     #Private

    def get_name(self):
        return self.name

    def get_age(self):
        return self._age - 10

    def get_address(self):
        return self.__address

my_obj = MyClass('Khan', 43, 'Hayatabad')
print(my_obj.get_address())
print(my_obj._age)
```

```
Hayatabad
43
```

[ ]:
```
# Note: Protected not working as it works in C++ or Java
# The single underscore for protected doesn't prevent instance variables from␣
 ↪accessing or modifying
# the instance from outside class. Although, if your are using the property to␣
 ↪modify the attribute, it is
# still accessible in Python. Hence, the responsible programmer would avoid/
 ↪refrain from accessing and modifying
# the instance variables prefixed with _ from outside its class.
```

[ ]:
```

```

[233]:
```
# Class Level attributes or static attributes
class MyClass:
    student_count = 0  # This is a class variable. In C++ or java static is used

    def __init__(self, name, age=0, address='No Address'):
        #instance variables
        self.name = name
        self.age = age
        self.address = address
        MyClass.student_count += 1

    def __del__(self):
        MyClass.student_count -= 1
```

[237]:
```
my_obj1 = MyClass('Bob', 43, 'USA')
my_obj2 = MyClass('Alice', 433, 'Canada')
print("Total Students", MyClass.student_count)
```

```python
print("Total Students", my_obj1.student_count)
print("Total Students", my_obj2.student_count)

del(my_obj1)
del(my_obj2)
print("Total Students", MyClass.student_count)
```

```
Total Students 2
Total Students 2
Total Students 2
Total Students 0
```

[238]:
```python
class MyClass:
    name = 'A quick brown fox jumps over the lazy dog' # Class level attribute/␣
 ↪Static

my_obj1 = MyClass()
my_obj2 = MyClass()
my_obj1.name = "A quick brown fox jumps over the lazy dog" # id will be changed

print(my_obj1.name)
print(my_obj2.name)
print(MyClass.name)

print(id(my_obj1.name))
print(id(my_obj2.name))
print(id(MyClass.name))
```

```
A quick brown fox jumps over the lazy dog
A quick brown fox jumps over the lazy dog
A quick brown fox jumps over the lazy dog
140165126855152
140165126853040
140165126853040
```

[5]:
```python
# In Java, the built-in toString() method is overridden to provide a custom␣
 ↪string representation of an object.
# In Python we use __str__ or __repr__ methods for the to provide a custom␣
 ↪string representation of an object.

class Employee:
    def __init__(self, fname, lname):
        self.firstname = fname
        self._lastname = lname

    def __str__(self):
        return self.firstname + " " + self._lastname

    def __repr__(self):
        return 'First Name: {}, Last Name: {}'.format(self.firstname,self.
↪_lastname)

obj = Employee("Muhammad", 'Khan')
print(obj)              # By Default __str__ method is called (Implicitly called)
print(obj.__str__())    # Explicity calling __str__ method
print(obj.__repr__())   # Explicity calling __repr__ method,
                        #__repr__ method is implicilty called, when __str__␣
↪method is not there
```

```
Muhammad Khan
Muhammad Khan
First Name: Muhammad, Last Name: Khan
```

[ ]:

```
```

[250]:

```
# Inheritance
class Employee:
    def __init__(self, fname, lname):
        self.firstname = fname
        self._lastname = lname

    def __str__(self):
        return self.firstname + " " + self._lastname

    def __repr__(self):
        return 'First Name: {}, Last Name: {}'.format(self.firstname,self.
↪_lastname)

class Teacher(Employee):
    def __init__(self, fname, lname, specialization):
        #Employee.__init__(self, fname, lname)
        super().__init__(fname, lname)
        self.specialization = specialization

    def __str__(self):
        return super().__str__() + " with specialization  " + self.specialization

    def __repr__(self):
        return super().__repr__() + ', Specialization: {}'.format(self.
↪specialization)


teacher_obj = Teacher("Muhammad", 'Khan', "Information Security")
print(teacher_obj)
print(teacher_obj.__str__())
print(teacher_obj.__repr__())
```

```
Muhammad Khan with specialization  Information Security
Muhammad Khan with specialization  Information Security
First Name: Muhammad, Last Name: Khan, Specialization: Information Security
```

3.1 Python Built-in Functions

[278]:
```
#The type() function returns the type of the specified object.

list_of_fruits = (1, 2, 3, 4)
print(type(list_of_fruits))
```

```
<class 'tuple'>
```

[279]:
```
#The input() function allows taking the input from the user.
a = input('Enter Value:')
print(a)
```

```
Enter Value:a
a
```

[286]:
```
tuple = ("d", "e", "b", "a", "c")
print(sorted(tuple))
```

```
['a', 'b', 'c', 'd', 'e']
```

[287]:
```
tuple = ("d", "e", "b", "10", "9")
print(sorted(tuple))
#Note: If a list contains string values as well as numeric values, then sorted␣
 ↪method does not work properly.
```

```
['10', '9', 'b', 'd', 'e']
```

[ ]:
```
# The ord() function returns the number representing the Unicode code of a␣
 ↪specified character.
```

[289]:
```
a = ord("a")
print(a)

a = ord("A")
print(a)
```

```
97
65
```

[ ]:
```

```