Natural Language Command & Control for UAV's

Assignment 1 Solutions

Marut Garg (240633)

IIT Kanpur – Electrical Engineering

December 10, 2025

Question 1: Mutable Default Arguments

Given Function:

```
def add_item(item, box=[]):
```

a) In this function, the default value of `box` is an empty list. Whenever the function is called without passing the `box` argument, Python uses the same list again and again instead of creating a new one.

So, if we call the function multiple times like this:

```
add_item(1)
add_item(2)
add_item(3)
```

All the values will get added to the same list. The output will become:

```
[1, 2, 3]
```

Even though we never passed the list explicitly.

b) The list persists between function calls because default arguments in Python are created only once when the function is defined, not every time the function is called. Since lists are mutable, any change made to the list remains stored in memory and is reused again in the next function call.

So, the same list object is shared across all calls.

c) To fix this problem, we should use `None` as the default value and then create a new list inside the function.

Corrected Function:

```
def add_item(item, box=None):
    if box is None:
        box = []
    box.append(item)
    return box
```

Now, every time the function is called without passing the list, a new empty list is created, and the data does not persist across calls.

Question 2: `__str__` vs `__repr__`

a) The `__str__` method is meant for the **user**. It should return a simple and readable output. This is what we see when we use the `print()` function.

On the other hand, the `__repr__` method is meant for **developers**. It is mainly used for debugging and should return an accurate and detailed description of the object.

b) If `__str__` is not defined, then Python automatically uses `__repr__` while printing the object.

But if only `__str__` is defined and `__repr__` is missing, then Python shows the default memory address while debugging.

Question 3: Class Variables vs Instance Variables

a) The difference is

- Instance variable → One copy per object
- Class variable → One shared copy for all objects

b) If we change a class variable using:

```
ClassName.var = new_value
```

Then the value of the class variable changes for:

- All existing objects
- All future objects

This happens because all objects refer to the same shared memory location for the class variable.

c) If we change the same variable using:

```
instance.var = new_value
```

Then Python creates a **new instance variable** with the same name inside that specific object. This does **not change the class variable**.

So now:

- The modified object has its own separate value
- All other objects still use the original class variable

Question 4: Complex Dictionary Parsing (Log Analysis)

Function Code:

```
def parse_logs(log_string):
status = {}
logs = log_string.split(";")

for entry in logs:
entry = entry.strip()
user, action = entry.split(":")
user = user.strip()
action = action.strip()

if action == "Login":
status[user] = "Online"
else:
status[user] = "Offline"

return status
```

Function Call:

```
logs = "User1:  Login; User2:  Login; User1:  Logout; User3:  Login;
User2:  Logout"
result = parse_logs(logs)
print(result)
```

Output:

```
{'User1':  'Offline', 'User2':  'Offline', 'User3':  'Online'}
```

Question 5: The "Safe" Calculator (Error Handling)

Program Code:

```
while True:
try:
a = float(input("Enter first number:  "))
b = float(input("Enter second number:  "))
op = input("Enter operator (+, -, *, /):  ")

if op == "+":
result = a + b
elif op == "-":
result = a - b
elif op == "*":
result = a * b
elif op == "/":
result = a / b
else:
print("Invalid Operator")
continue

except ZeroDivisionError:
print("Cannot divide by zero")

except ValueError:
print("Invalid input.  Please enter numbers only")

else:
print("Result:", result)

finally:
print("Execution attempt complete")

choice = input("Do you want to continue?  (y/n):  ")
if choice.lower() != "y":
break
```

Question 6: Class Interaction & State Management (The Library System)

Program Code:

```
class Book:
def __init__(self, title, author):
self.title = title
self.author = author
self.is_checked_out = False

class Library:
def __init__(self):
self.books = []

def add_book(self, book_obj):
self.books.append(book_obj)

def checkout_book(self, title):
for book in self.books:
if book.title == title:
if not book.is_checked_out:
book.is_checked_out = True
print("Book checked out successfully")
else:
print("Error:  Book is already checked out")
return
print("Book not found")

def return_book(self, title):
for book in self.books:
if book.title == title:
book.is_checked_out = False
print("Book returned successfully")
return
print("Book not found")
```

Question 7: Encapsulation with Property Decorators

Program Code:

```
class Employee:
def __init__(self, first, last, salary):
self.first = first
self.last = last
self.salary = salary

@property
def email(self):
return f"{self.first}.{self.last}@company.com"

@property
def fullname(self):
return f"{self.first} {self.last}"

@salary.setter
def salary(self, value):
if value < 0:
raise ValueError("Salary cannot be negative")
self._salary = value

@salary.getter
def salary(self):
return self._salary

@fullname.deleter
def fullname(self):
self.first = None
self.last = None
```

Question 8: Operator Overloading (Magic Methods)

Program Code:

```
class TimeDuration:
def __init__(self, hours=0, minutes=0):
self.hours = hours
self.minutes = minutes
self._normalize()

def _normalize(self):
extra, self.minutes = divmod(self.minutes, 60)
self.hours += extra

def __add__(self, other):
return TimeDuration(self.hours + other.hours, self.minutes + other.minutes)

def __str__(self):
return f"{self.hours}H:{self.minutes}M"
```

Testing the Class:

```
t1 = TimeDuration(1, 50)
t2 = TimeDuration(2, 30)
t3 = t1 + t2
print(t3)
```

Expected Output:

```
4H:20M
```