

Solutions

## 2.1 Parameters and Arguments

### Reference example

```
def scale(value, factor):  
    return value * factor  
  
print(scale(4, 3))
```

#### 2.1.1 - Trace

Determine the exact output and explain the state changes.

```
def change(values):  
    values.append(9)  
  
data = [1]  
change(data)  
print(data)
```

### Output

[1, 9]

The local parameter refers to the caller list, so mutation is visible.

#### 2.1.2 - Write code

Write `add_tax(price, rate)` and show a price of 100 at rate 0.08.

Return the price multiplied by one plus the rate.

```
def add_tax(price, rate):  
    return price * (1 + rate)  
  
print(add_tax(100, 0.08))
```

### Expected output

108.0

Solutions

## 2.2 Return Values

### Reference example

```
def double(value):  
    return 2 * value  
  
result = double(6)  
print(result + 1)
```

### 2.2.1 - Trace

Determine the exact output and explain the state changes.

```
def show(value):  
    print(value)  
  
result = show(4)  
print(result)
```

### Output

4  
None

The function prints 4 and implicitly returns None because it has no explicit return value.

### 2.2.2 - Write code

Write absolute(value) without using abs and test -3 and 2.

Return from the negative and nonnegative cases.

```
def absolute(value):  
    if value < 0:  
        return -value  
    return value  
  
print(absolute(-3), absolute(2))
```

### Expected output

3 2

Solutions

## 2.3 Scope

### Reference example

```
count = 10
def local_count():
    count = 3
    return count

print(local_count())
print(count)
```

#### 2.3.1 - Trace

Determine the exact output and explain the state changes.

```
x = 5
def f(x):
    x += 1
    return x
print(f(x), x)
```

#### Output

6 5

The parameter `x` is local, and incrementing that local integer binding does not change the caller binding.

#### 2.3.2 - Explain

Why is returning a value often clearer than changing a global variable?

It makes the dependency and result explicit at the call site, reducing hidden shared state and simplifying isolated tests.

Solutions

## 2.4 Default and Keyword Arguments

### Reference example

```
def greet(name, prefix="Hi"):
    return f"{prefix}, {name}"

print(greet("Ada"))
print(greet(prefix="Hello", name="Lin"))
```

### 2.4.1 - Explain

Why is `def add_item(item, values=[])` risky across calls?

The default list is created once and reused. Mutating it can leak state into later calls. Use a `None` default and allocate a new list when needed.

### 2.4.2 - Write code

Write `collect(item, values=None)` that uses a new list when no list is supplied. Show two independent calls.

Allocate the default container inside the function.

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

```
print(collect(1))
print(collect(2))
```

### Expected output

```
[1]
[2]
```