Solutions for Exercise Sheet 1

Lecture: Advanced Programming and Algorithms – Part I

Author: Anja Rey

Problem 1: Find the Error

a) Compute remainder of integer division

Error: Loop subtracts b until a <= 0, returning negative or zero instead of remainder.

Fix: Stop when a < b.

Corrected Python Code:

```
def get_remainder(a, b):
    while a >= b:
        a -= b
    return a
```

Optimized: Use modulo operator:

```
def get_remainder(a, b):
    return a % b
```

b) Check divisibility by 3

Error: Line 3 always sets result to False, overriding True.

Fix: Use else clause.

Corrected Python Code:

```
def is_divisible_by_three(n):
    return n % 3 == 0
```

c) Compute power m^n

Original code works for positive n but can be improved.

Optimized Python Code:

```
def power(m, n):
    return m ** n
```

d) Sum of squares

Error: range(n) goes from 0 to n-1, but sum should be from 1 to n.

Corrected Python Code:

```
def sum_squares(n):
    total = 0
    for index in range(1, n + 1):
        total += index ** 2
    return total
```

Optimized using formula:

```
def sum_squares(n):
    return n * (n + 1) * (2 * n + 1) // 6
```

Problem 2: Efficiency and Refactoring

a) Running Times:

- get_remainder: O(a/b)
- is_divisible_by_three: O(1)
- power: O(n)
- sum_squares: O(n)

b) Improvements:

- get_remainder: Use modulo operator (O(1))
- is_divisible_by_three: Already optimal
- power: Use operator (O(log n) internally)
- sum_squares: Use formula (O(1))

Problem 4: Example Algorithm

a) Inspecting the provided pseudocode

Given pseudocode:

```
is_divisible_by_three(n):
  if $n \equiv 0 \mod 3$ then
    result ← True
  result ← False
  return result
```

Issues with the pseudocode:

- Missing else branch / incorrect control flow: The False assignment overwrites True.
- Unnecessary variable: Can return boolean directly.
- Ambiguous input domain: Does not specify integer requirement.

Improved pseudocode:

```
is_divisible_by_three(n):
  if n mod 3 == 0 then
    return True
  else
    return False
```

Even cleaner:

```
is_divisible_by_three(n):
  require $n \in \mathbb{Z}$
  return (n mod 3 == 0)
```

b) What does it do? What happens in the background? What problem does it solve?

The algorithm checks if an integer n is divisible by 3. It solves the decision problem: Given n, decide if $\exists k \in \mathbb{Z}$ such that $n = 3k$.

Background: It uses $n \bmod 3$ to compute the remainder. If remainder is 0, n is a multiple of 3.

Complexity: $O(1)$ in word-RAM model; $O(\log n)$ in bit complexity. Memory: $O(1)$.

Alternative approaches: Digit sum method (based on $10 \equiv 1 \mod 3$) or DFA with 3 states for streaming digits.

c) How can the algorithm and the code be improved? Useful quality criteria

Improvements:

- Use direct return of boolean expression.
- Validate input type and domain.
- Handle edge cases: negatives, zero, non-integers.
- Consider streaming approach for very large numbers.

Quality criteria:

1. Correctness: Meets specification.
2. Clarity: Simple and readable.
3. Robustness: Handles invalid inputs gracefully.
4. Efficiency: O(1) for fixed-width integers.
5. Maintainability: Clear structure and documentation.
6. Portability: Avoid language-specific quirks.
7. Testability: Easy to test with unit and property-based tests.
8. Security: Avoid overflow and unsafe parsing.
9. Documentation: State preconditions and examples.

Jupyter Notebook Example: Python Introduction

```
print('Hello EMON')
```

```
Hello EMON
```

Just get started! Code can be typed and interpreted dynamically.

```
12 + 13
```

```
25
```

```
True + False
```

```
1
```

You can find help in the python documention, as well as here via: help, ?, find methods and attributes, auto-completion with tab, further info with shift+tab

```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current
sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

```
print ('ASHIK EMON HHU')
```

```
ASHIK EMON HHU
```

Structure

```
import numpy as np
```

```
a = 7
```

Linebreaks start a new command!

```
b = 1 + 2
+ 3
print(b)
```

```
3
```

How can we fix this?

```
a = 7
b = 3
print(f'a + b = {a + b}')
```

```
a + b = 10
```

Conditional Statements

```
if True:
    print('Condition holds.')
else:
    print('Condition does not hold.')
```

Indentation (4 spaces) has syntactic meaning!

```
if False:
    a = 1
print('Condition holds.')
```

What went wrong here?

```
int('5')          # converts a string to integer → 5
int.bit_length    # refers to the bit_length method of int objects
```

```
<method 'bit_length' of 'int' objects>
```

```
from __future__ import braces
```

Loops

```
n = 1
while n < 64:
    n = 2 * n
print(f'{n} ', end='')
```

```
64
```

What happens if we change different aspects of this code?

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

```python
for i in range(0, 5):
    print(i)
```

```
0
1
2
3
4
```

How does range count? (Note: Details on lists and iterators later.)

```python
for i in range(1, 3):
    print(i)
```

```
1
2
```

```python
for i in range(3, 1, -1):
    print(i)
```

```
3
2
```

```python
beginning_incl = 2
end_excl = 12
steps = 2
for i in range(beginning_incl, end_excl, steps):
    print(i)
```

```
2
4
6
8
10
```

Variables, Assignments, & Pointers

Variable assignments don't copy values, but point to an object. Integers are immutable: An integer variable points to an object with a fixed value.

```
first_value = 7
second_value = 7
third_value = first_value

print(id(first_value))
print(id(second_value))
print(id(third_value))

4377084336
4377084336
4377084336
```

Compare value with '=='

```
print(first_value == second_value)
print(first_value == third_value)

True
True
```

Compare pointers with 'is'

```
print(first_value is second_value)
print(first_value is third_value)

True
True

first_value += 1
print(id(first_value))
print(first_value is third_value)

4377084368
False
```

This is different for, e.g., lists. They can be changed dynamically.

```
first_collection = [1, 2, 3]
second_collection = [1, 2, 3]
third_collection = first_collection

print(id(first_collection))
print(id(second_collection))
print(id(third_collection))

4454271040
4454166336
4454271040
```

```
print(first_collection == second_collection)
print(first_collection == third_collection)
print(first_collection is second_collection)
print(first_collection is third_collection)
```

```
True
True
False
True
```

```
first_collection.append(4)
print(first_collection)
print(second_collection)
print(third_collection)
```

```
[1, 2, 3, 4]
[1, 2, 3]
[1, 2, 3, 4]
```

```
first_collection = [0, 0, 0]
second_collection.append(4)
print(first_collection)
print(second_collection)
print(third_collection)
```

```
[0, 0, 0]
[1, 2, 3, 4]
[1, 2, 3, 4]
```

```
print(second_collection == third_collection)
print(second_collection is third_collection)
```

```
True
False
```

Note: More details on lists and other mutable objects later.

Basic Types & Operators

Basic types include, e.g., integers, strings, and booleans.

```
a = 7
b = 'Hello'
c = True
```

```
type(a)
```

```
int
```

```
type(b)
```

```
str
```

```
type(c)
```

```
bool
```

Note: bool is a subtype of int

Arithmetic Operators

Let's take a look at some basic operators for numbers, type upcast, and precision

```
a = 5
b = 710235846812354878794123636478794512464123458796423121647 94
c = 5 - 7
d = 5 / 7
e = 5 // 7
print(f'{c}, {d}, {e}')
```

```
-2, 0.7142857142857143, 0
```

```
type(a)
```

```
int
```

```
type(b)
```

```
int
```

```
type(c)
```

```
int
```

```
type(d)
```

```
float
```

integers have unlimited precision (unlike overflow in, e.g. 64 bit)

float precision depends on machine on which programme is running

there is also a type complex (real and im both float)

```
type(e)
```

```
int
```

```
type(4 / 2)

float

a = 1 / 3

print(a * 3)

1.0

1.2 * 3

3.5999999999999996

float.hex(1.2)

'0x1.3333333333333p+0'

-0

0

type(0)

int

print(-a)

-0.3333333333333333

1 ** 0

1

0 ** 0

1

bin(3)

'0b11'

type(bin(3))

str

int(0b1001)

9

0b0011 * 3

9

0b11 + 0b110
```

```
9
```

Try out other operators like %, **

```
a = 10
b = 3

print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Floor Division:", a // b)
print("Modulus:", a % b)
print("Power:", a ** b)
```

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Floor Division: 3
Modulus: 1
Power: 1000
```

Bitwise Operators

Find out what these operators do: &, |, ^, <<, >>.

```
a = 5   # binary: 0101
b = 3   # binary: 0011

print("a =", a, "→", bin(a))
print("b =", b, "→", bin(b))

# 1 AND (&)
# 0101 & 0011 = 0001 (1)
print("a & b =", a & b)   # 1

# 2 OR (|)
# 0101 | 0011 = 0111 (7)
print("a | b =", a | b)   # 7

# 3 XOR (^)
# 0101 ^ 0011 = 0110 (6)
print("a ^ b =", a ^ b)   # 6

# 4 Left Shift (<<)
# Shift bits of a to the left → multiply by 2 each shift
# 0101 << 1 = 1010 (10)
print("a << 1 =", a << 1)  # 10
print("a << 2 =", a << 2)  # 20

# 5 Right Shift (>>)
# Shift bits of a to the right → divide by 2 each shift
# 0101 >> 1 = 0010 (2)
print("a >> 1 =", a >> 1)  # 2
print("a >> 2 =", a >> 2)  # 1
```

```
a = 5 → 0b101
b = 3 → 0b11
a & b = 1
a | b = 7
a ^ b = 6
a << 1 = 10
a << 2 = 20
a >> 1 = 2
a >> 2 = 1
```

Boolean Comparison

In the context of conditional statements, we have already seen comparison operators like '==' and '<'. These map two objects to a boolean value.

```
'a' != a
```

```
True
```

```
1 == True
```

```
True
```

```
3 * 0.2 == 0.6
```

```
False
```

```
abs(3 * 0.2 - 0.6) <= 0.0000000000000002
```

```
True
```

```
round(3 * 0.2, 15) == 0.6
```

```
True
```

Logical Operators

There are also logical operators that map one or two boolean values to a boolean value.

```
(True and True or False and (True or False)) ^ (False and True)
```

```
True
```

```
(1 != 2) ^ (not(0))
```

```
False
```

String Operators

There are many operators for strings such as search, split, concatenate, convert, ... We'll take a closer look at them whenever relevant.

```
# Define a sample string
text = "Hello Python World"

#  Concatenation (+)
greeting = "Hello"
language = "Python"
result = greeting + " " + language
print("Concatenation:", result)

#  Repetition (*)
print("Repetition:", greeting * 3)  # repeats the string 3 times

#  Membership (in, not in)
print("'Python' in text →", "Python" in text)
print("'Java' not in text →", "Java" not in text)

#  Searching (find, index)
print("Find 'Python' →", text.find("Python"))  # returns index
# .find() returns -1 if not found, while .index() raises an error
try:
    print("Index of 'World' →", text.index("World"))
except ValueError:
    print("Not found!")

#  Splitting and Joining
```

```
words = text.split()           # splits by spaces → list of words
print("Split:", words)
joined = "-".join(words)       # joins list with a separator
print("Join:", joined)

# Changing Case
print("Uppercase:", text.upper())
print("Lowercase:", text.lower())
print("Title Case:", text.title())

#  Replacement
print("Replace 'World' with 'Universe':", text.replace("World",
"Universe"))

#  Conversion between strings and numbers
num_str = "42"
num_int = int(num_str)
print("Convert string to int:", num_int, "→ type:", type(num_int))
print("Convert int back to string:", str(num_int), "→ type:",
type(str(num_int)))

#  String formatting (f-strings)
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")

#  Length and slicing
print("Length:", len(text))
print("First 5 characters:", text[:5])
print("Last 5 characters:", text[-5:])
```

```
Concatenation: Hello Python
Repetition: HelloHelloHello
'Python' in text → True
'Java' not in text → True
Find 'Python' → 6
Index of 'World' → 13
Split: ['Hello', 'Python', 'World']
Join: Hello-Python-World
Uppercase: HELLO PYTHON WORLD
Lowercase: hello python world
Title Case: Hello Python World
Replace 'World' with 'Universe': Hello Python Universe
Convert string to int: 42 → type: <class 'int'>
Convert int back to string: 42 → type: <class 'str'>
My name is Alice and I am 25 years old.
Length: 18
First 5 characters: Hello
Last 5 characters: World
```

Functions

Next to built-in functions (e.g., print, range), we can create our own functions.

```
def name(parameter):
    pass

def factorial(n: int):
    x = 1
    for i in range(2, n+1):
        x = x * i
        print(x)
    return x

factorial('hello')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[73], line 1
----> 1 factorial('hello')

Cell In[72], line 3, in factorial(n)
      1 def factorial(n: int):
      2     x = 1
----> 3     for i in range(2, n+1):
      4         x = x * i
      5         print(x)

TypeError: can only concatenate str (not "int") to str
```

```
# 'hello', which is a string, not an integer

factorial(5)
```

```
2
6
24
120

120
```

```
factorial(True)
```

```
1

factorial(5)

2
6
24
120

120

factorial(-4)


1

def is_negative(n: int) -> bool:
    if n < 0:
        return True
    else:
        return False
```

How can you write this function with one line?

```
def is_negative(n: int) -> bool:
    return n < 0

def factorial_recursive(n):
    if is_negative(n):
        return 'negative value'
    if n == 1 or n == 0:
        return 1
    return n * factorial_recursive(n-1)

factorial_recursive(-4)

'negative value'

factorial_recursive(5)

120

import math
factorial_recursive(1000) == math.factorial(1000)

True
```

Note: Details on recursive functions and their algorithmic properties in Part II.

Multiple return values and default values

```
def division_remainder(number: int, divisor: int=3) -> (int, int,
bool):
    return (number // divisor, number % divisor, number % divisor ==
0)
```

```
division_remainder(6)
```

```
(2, 0, True)
```

```
division_remainder(6, 4)
```

```
(1, 2, False)
```

```
number = 103
divisor = 4
div, remainder, is_divisible = division_remainder(number, divisor)
if is_divisible == 1:
    print(f'{number} can be divided by {divisor}')
else:
    print(f'{number} equals {div} times {divisor} plus {remainder}')
```

```
103 equals 25 times 4 plus 3
```

Lambda Functions

Define short, temporarily needed functions via: = lambda : expression

```
def near(a, b, threshold):
    if a > b:
        distance = lambda a, b: a - b
    else:
        distance = lambda a, b: b - a
    if distance(a,b) <= threshold:
        return True
    return False, f'{distance(a,b) - threshold} beyond threshold'
```

```
near(5, 100, 50)
```

```
(False, '45 beyond threshold')
```

Note: Often used as input filters or quick mappings. Details later.