Seminar 4

ES12003

2024–25

1 Introduction

In the lecture notes it's claimed that by utilising functions we can break down complex problems into smaller, manageable components. Code written in this way is easier to read, review and maintain. As a demonstration, let's revisit the problem of prime numbers (see Seminar 3), this time making separate subroutines first and then linking and combining them to form a complete system.

The following program defines the function `is_prime` and the procedure `print_prime_numbers`. The former determines whether a number is prime and the latter prints all prime numbers up to `number`, here 60. Notice that we have a function call within another function.

```
def is_prime(number):
    divisor = 2
    while divisor <= number/2:
        if number % divisor == 0: # if true, number is not a prime
            return False
        divisor += 1
    return True # number is a prime

def print_prime_numbers(stop):
    for number in range(2, stop+1):
        if is_prime(number):
            print(number, end = ", ")

print_prime_numbers(60)
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
```

The program divides a large problem into two subproblems. Modularising makes code easy to maintain and debug, and enables the code to be reused.

2 Home exercises

Exercise 2.1

Reuse the `is_prime` function to write a function that returns the first $n = 20$ prime numbers.

Solution 2.1

```
def get_prime_numbers(n):
    count = 0 # count the number of prime numbers
    number = 2 # a number to be tested whether it is a prime
    while count < n:
        if is_prime(number):
            print(number, end = " ") # print the prime number
            count += 1 # increase the count
        number += 1 # check if the next number is a prime
get_prime_numbers(20)
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
```

Exercise 2.2

Write a function that takes in a list of integers and returns the count of odd and even numbers in the list.

Solution 2.2

The function below returns a tuple (comma-separated values are treated as tuples, even without parentheses).

```
def count_odd_even(numbers):
    even_counter = odd_counter = 0
    for number in numbers:
        if number % 2 == 0:
            even_counter += 1
        else:
            odd_counter += 1
    return odd_counter, even_counter
```

To access elements within a tuple, we can use indexing. Recall that indexing in Python starts at 0 (meaning the first element in a tuple has an index of 0 and the second element has an index of 1).

```
L1 = list(range(1, 20, 3))
result = count_odd_even(L1)
print(f'There are {result[0]} odd and {result[1]} even numbers in
 ↪  {L1}')
```

```
There are 4 odd and 3 even numbers in [1, 4, 7, 10, 13, 16, 19]
```

Exercise 2.3

Take a moment to study the following code and predict the result before running it.

Case 1

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

Case 2

```
def g(y):
    print(x)
    print(x+1)
x = 5
g(x)
print(x)
```

Case 3

```
def z(y):
    x += 1
x = 5
z(x)
print(x)
```

Solution 2.3

Case 1: On line 6 we call the `f` procedure with actual parameter `5`. So, the formal parameter `y` is assigned to `5`. In the body of the function we define the

local variable `x`, which initially takes the value of `1` and then increases to `2` (`y` is not used anywhere in the body of the function). So the printout will be:

```
2
5
```

The first (second) line prints the local (global) value of `x`.

Case 2: On line 5, you call procedure `g` with argument `5`. In the body of `g`, we do not define any local variable `x`, so `x` is assigned its global value (`5`). The printout is:

```
5
6
5
```

The first two lines result from executing the print commands inside the body of the procedure. The last line results from executing line 6.

Case 3: The printout is:

```
UnboundLocalError: cannot access local variable 'x' where it is
not associated with a value
```

The type of `x` is `int` (an immutable). Inside the procedure, you can access a global variable of type `int` (like we did in Case 2), but we can't modify it.

Exercise 2.4

Write a Python procedure that

a) using list comprehension (see Topic 2 slides, page 19) creates a single list with all 36 different dice combinations from $(1,1)$ to $(6,6)$, and

b) prints all combinations with the number on the top of the second die being n, where $1 \leq n \leq 6$ is provided as an argument of the function.

Solution 2.4

```
1 def second_dice(n):
2     combinations = [(d1, d2) for d1 in range(1,7) for d2 in
      ↪  range(1,7)]
3     for combination in combinations:
4         if combination[1] == n:
```

```
            print(combination)
second_dice(5)
```

```
(1, 5)
(2, 5)
(3, 5)
(4, 5)
(5, 5)
(6, 5)
```

3 Class exercises

Exercise 3.1

Assuming `L = list(range(1,11))`, the following list comprehensions create new lists by applying a function to every element of another iterable (that—in the case of `L2` and `L3`—satisfies a certain condition).

```
L1 = [x**2 for x in L]
L2 = [x**2 for x in L if x%2 == 0]
L3 = [[x,x**2] for x in L if x%2 == 1]
```

Can you create functions that have exactly the same effect?

Exercise 3.2 *(Exercise 6 in the 2023–24 ES12003 S1 Exam)*

(a) Copy & paste the following three lists into a Jupyter Notebook cell.

```
List1 = [3, 4, 2, 5, 1, 12, 3, 15, 1, 5]
List2 = [13, 16, 19, 4, 14, 19, 5, 2, 1, 12]
List3 = [5, 17, 15, 13, 17, 0, 12, 15, 3, 6]
```

(b) Write a function that takes *three* numerical arguments, and returns the *smallest* of these arguments if their sum is an even number, and the *largest* of these arguments if their sum is an odd number (use in-built functions `min` and `max`).

(c) Write a loop that does the following:

- takes the first elements of the lists from (a) (that is, 3, 13 and 5), inserts these values as arguments in the function from (b), and stores the resulting output in a new list called `out` (as its first element).

- takes the second elements of the lists from (a) (that is 4, 16 and 17), inserts these values as arguments in the function from (b), and again stores the resulting output in `out` (as its second element).
- repeats the procedure with subsequent list elements, until the last elements of the lists from (a) are processed.

Print the final state of out.