4 Functions

Learning Outcome

Programming Elements and Contructs
- Use functions and procedures to modularise problem into chunks of code
- Understand the concept of recursion
- Trace the steps and list the results of recursive and non-recursive programs

Function is an abstraction mechanisms that hides details and allows us to view many things as just one thing. We can use functions to organize our codes more effectively.

4.1 Advantage of Functions

- Functions serve as abstraction mechanisms by eliminating redundant, or repetitious, code.

```
def sum(lower, upper):
    """
    Arguments: A lower bound and an upper bound
    Returns: the sum of the numbers between the arguments
             and including them
    """
    result = 0
    while lower <= upper:
        result += lower
        lower += 1
    return result

>>> sum(1, 4)      # The summation of the numbers 1..4
10
>>> sum(50, 100)   # The summation of the numbers 50..100
3825
```

- Functions serve as abstraction mechanisms by hiding complicated details.

For example, consider the previous sum function. The idea of summing a range of numbers is simple; the code for computing a summation is not.

A function call expresses the idea of a process to the programmer without forcing him/her to wade through the complex code that realizes that idea

- Functions Support General Methods with Systematic Variations
  - An algorithm is a **general method** for solving a class of problems
  - The individual problems that make up a class of problems are known as **problem instances**
  - Algorithms should be general enough to provide a solution to many problem instances
  - A function should provide a general method with systematic variations

- Functions Support the Division of Labor
  - In a well-organized system, each part does its own job in collaborating to achieve a common goal
  - In a computer program, functions can enforce a division of labor
  - Each function should perform a single coherent task
  - Each of the tasks required by a system can be assigned to a function, including the tasks of managing or coordinating the use of other functions

4.2 Problem Solving with Top-Down Design

- **Top-down design** starts with a global view of the entire problem and breaks the problem into smaller, more manageable subproblems, process known as problem decomposition
- As each subproblem is isolated, its solution is assigned to a function
- As functions are developed to solve subproblems, solution to overall problem is gradually filled out, process known as stepwise refinement

4.3 Define Simple Functions

4.3.1 The Syntax of Simple Function Definitions

A function can be defined in a Python shell, but it is more convenient to define it in an IDLE window.

Definition of a function consists of header and body. Syntax of a function definition:

```
def <function name>(<parameter-1>, …, <parameter-n>):
    <body>
```

```
def square(x):
    """Returns the square of x. """
    return x * x
```

```
>>> square(2)
4
```

Docstring contains information about what the function does; to display, enter help(square)

4.3.2 Parameters and Arguments

A parameter is the name used in the function definition for an argument that is passed to the function when it is called. Arguments provide the function's caller with the means of transmitting information to the function.

The number and positions of arguments of a function call usually match the number and positions of the parameters in the definition.
Some functions expect no arguments. They are defined with no parameters.

Programmer can specify optional arguments with default values in any function definition:

```
def <function name>(<required args>,
                    <key-1> = <val-1>, … <key-n> = <val-n>)
```

Following the required arguments are one or more **default or keyword arguments.** When function is called with these arguments, default values are overridden by caller's values.

```
def repToInt(repString, base):
    """Converts the repString to an int in the base
    and returns this int."""
    decimal = 0
    exponent = len(repString) - 1
    for digit in repString:
        decimal = decimal + int(digit) * base ** exponent
        exponent -= 1
    return decimal
```

```
def repToInt(repString, base = 2):

>>> repToInt("10", 10)
10
>>> repToInt("10", 8)    # Override the default to here
8
>>> repToInt("10", 2)    # Same as the default, not necessary
2
>>> repToInt("10")       # Base 2 by default
2
```

The default arguments that follow can be supplied in two ways:

- **By position**
- **By keyword**

```
def example(required, option1 = 2, option2 = 3):
    print required, option1, option2

>>> example(1)                 # Use all the defaults
1 2 3
>>> example(1, 10)             # Override the first default
1 10 3
>>> example(1, 10, 20)         # Override all the defaults
1 10 20
>>> example(1, option2 = 20)   # Override the second default
1 2 20
>>> example(1, option2 = 20, option1 = 10)  # Note the order
1 10 20
```

4.3.3 The return Statement

Place a **return** statement at each exit point of a function when function should explicitly return a value.

Syntax:

```
return <expression>
```

If a function contains no **return** statement, Python transfers control to the caller after the last statement in the function's body is executed. The special value **None** is automatically returned.

4.3.4 Functions with Multiple Return Variables

A function can return multiple values. For example, the code below returns both sum and average.

```
#calculate the sum and average of numbers
#between the arguments and including them
def calc(lower,upper):
      total = 0
      count = 0
      while lower <= upper:
            total += lower
            count += 1
            lower += 1
      average = total / count
      return (total,average)
```

```
>>> calc(1,10)
(55, 5.5)
>>> calc(10,50)
(1230, 30.0)
```

4.3.5 Boolean Functions

A Boolean function usually tests its argument for the presence or absence of some property. It returns **True** if property is present; **False** otherwise

For example, the function odd below tests if a number is odd.

```
>>> odd(5)
True
>>> odd(6)
False

def odd(x):
    """Returns True if x is odd or False otherwise."""
    if x % 2 == 1:
        return True
    else:
        return False
```

4.3.6 Defining a main Function

main serves as the entry point for a script. It usually expects no arguments and returns no value. Definition of **main** and other functions can appear in no particular order in the script, as long as **main** is called at the end of the script. Script can be run from IDLE, imported into the shell, or run from a terminal command prompt

```
"""
File: computesquare.py
Illustrates the definition of a main function.
"""

def main():
    """The main function for this script."""
    number = float(input("Enter a number: "))
    result = square(number)
    print("The square of", number, "is", result)

def square(x):
    """Returns the square of x."""
    return x * x

# The entry point for program execution
main()
```

4.4 Recursive Functions

In a **recursive** subprogram, the body of the subprogram contains a call to itself.

Recursion is used when the original task can be reduced to a simpler version of itself. The idea is that after a number of successive reductions, the reduced problem will eventually be simple enough to be solved directly, and its solution will be used to piece together a solution to the original problem.

As an illustration, recall that for n, a positive integer, **n!** is equal to the product of the first n integers. For example,

$$4! = 4 * 3 * 2 * 1 = 24$$

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

1! is defined as 1.

Note that for any positive n, **n! = n * (n-1)!**

Thus, finding **n!** can be reduced to the similar but simpler task of finding **(n-1)!**.

4.4.1 Defining a Recursive Function with Terminal Case

Remember that whenever a subprogram call has been completed, control is returned to the point at which the subprogram was called. The same rule applies to recursive calls.

Terminal Case

In order that successive recursive calls not continue indefinitely, the body of a recursive subprogram should include at least one terminal case – a case that contains no further calls to the recursive subprogram (As you will see, terminal cases are similar to loop exit conditions.)

In the following example, the terminal case is n = 1.

Example (Recursive Factorial Function)

```
def FACT(n):
     if n==1:
          return 1
     else:
          return n * FACT(n-1)

def main():
     n = int(input("ener a positive
integer: "))
     print (n, " factorial is ", FACT(n))

main()
```

Here is the output to compute **4!**

enter a positive integer 4
4 factorial is 24

When this program is run to compute **4!**, there will be four calls to the recursive function **FACT**.

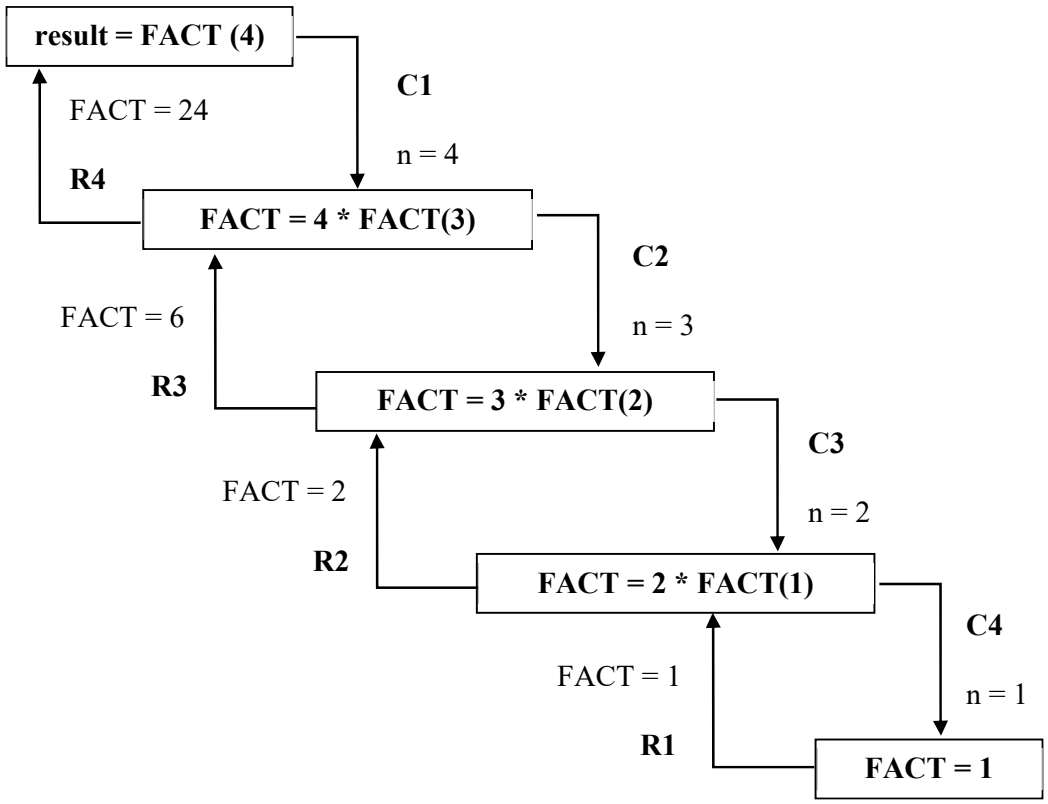
The first call is to **FACT(4)** from the main body. This call cannot be completed immediately since **FACT(4)** calls **FACT(3)**, which calls **FACT(2)**, which calls **FACT(1)**.

The call to **FACT(1)** is the first call that can be completed. Then the computer winds back up and assigns **FACT** the value **1**.

That value is sent to the place where **FACT(1)** was called. Then the call of **FACT(2)** is completed and the value **2** is sent to the place where **FACT(2)** was called. Then the call of **FACT(3)** is completed, and the value **6** is sent to the place where **FACT(3)** was called.

Finally, the original call of **FACT(4)** is completed.

Here is the trace diagram. C refers to call, and R refers to Return.



The following example takes a further look at "winding back up" to finish unfinished calls.

```
def JOB (n):
     if n == 1:          // terminal case
          print("n = 1  GO BACK")
     else:               // recursive step
          print(n, " hi")
          JOB (n-1)
          print(n, " bye")
```

The output for **JOB (4)** will be

4 hi
3 hi
2 hi
n = 1 GO BACK
2 bye
3 bye
4 bye

First, the call **JOB (4)** is started.

The computer outputs **4 hi**, and then **JOB (3)** is called. **Note that JOB (4) still has not been completed.**

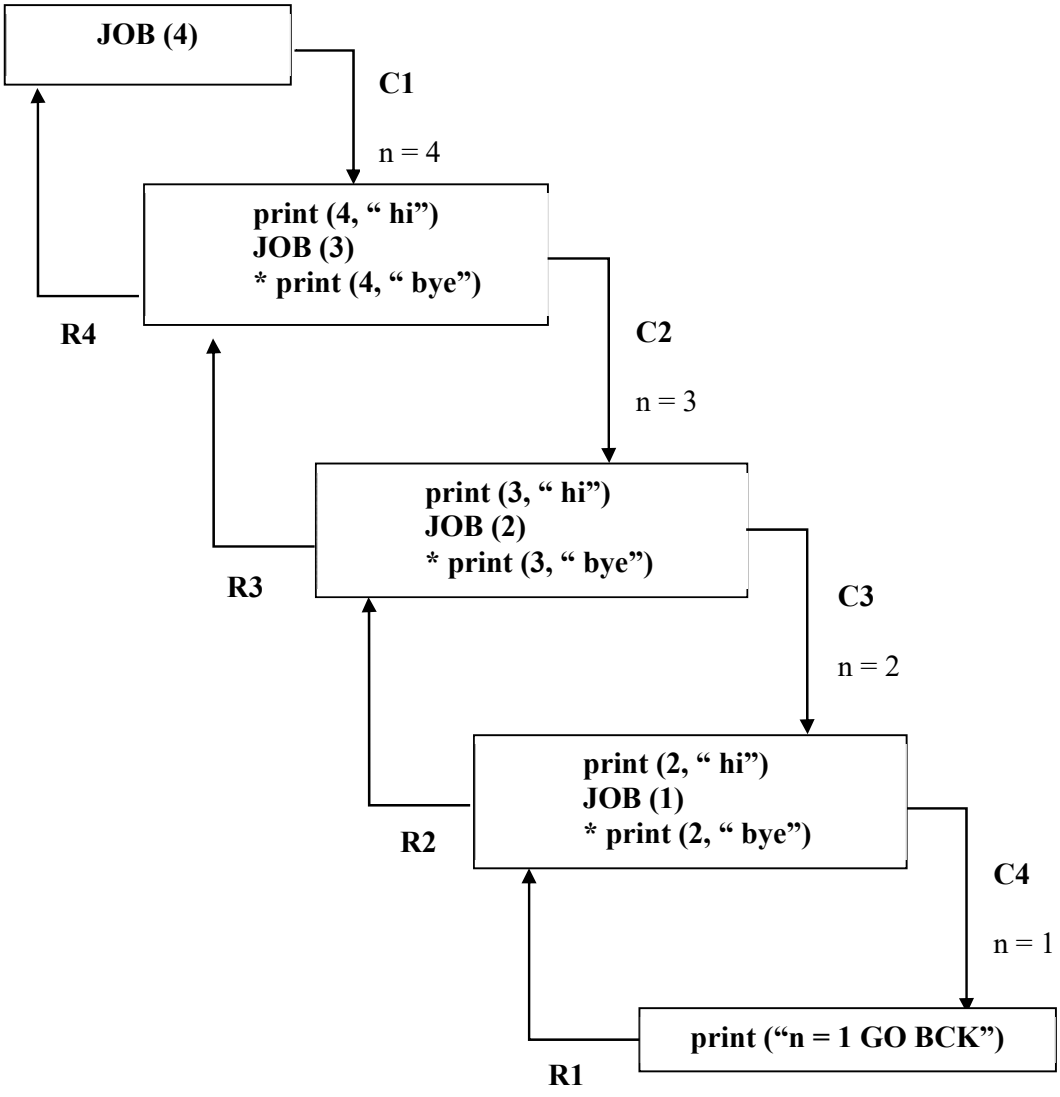
In starting **JOB (3)**, the computer outputs **3 hi** and then calls **JOB (2)**, in which it first outputs **2 hi** and then calls **JOB (1)**.

JOB (1) is a terminal case since its execution does not produce further calls to **JOB** – it outputs **n = 1 GO BACK**.

The computer must still ***wind back up*** and finish ***all the unfinished calls***.
(The unfinished part of each call is marked with *.)

It returns to the point in **JOB (2)** where **JOB (1)** was called and outputs **2 bye**. Then it finishes **JOB (3)**, output **3 bye**, and **JOB (4)**, output **4 bye**.

Here is the trace diagram.



4.4.2 Defining a Recursive Function with Base Case

Another way of recursive method is to use selection statement to examine **base case** to determine whether to stop or to continue with another **recursive step.**

Let's convert the following function displayRange to a recursive function:

```
def displayRange(lower, upper):
    """Outputs the numbers from lower to upper."""
    while lower <= upper:
        print(lower)
        lower = lower + 1
```

We can replace the assignment statement **lower = lower + 1** with a recursive call.

```
def displayRange(lower, upper):
    """Outputs the numbers from lower to upper."""
    if lower <= upper:
        print(lower)
        displayRange(lower + 1, upper)
```

4.4.3 Using Recursive Definitions to Construct Recursive Functions

A recursive definition consists of equations that state what a value is for one or more base cases and one or more recursive cases.

Example: Fibonacci sequence

```
Fib(n) = 1, when n = 1 or n = 2
Fib(n) = Fib(n - 1) + Fib(n - 2), for all n > 2
```

1 1 2 3 5 8 13 . . .

Recursive functions are frequently used to design algorithms that have a recursive definition

```
def fib(n):
    """Returns the nth Fibonacci number."""
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

4.4.4 Infinite Recursion

Infinite recursion arises when programmer fails to specify base case or to reduce size of problem in a way that terminates the recursive process

In fact, the Python virtual machine eventually runs out of memory resources to manage the process

4.4.5 The Costs and Benefits of Recursion

Recursion can simplify the design of algorithms and the codes. However, it requires more amount of memory than loops.

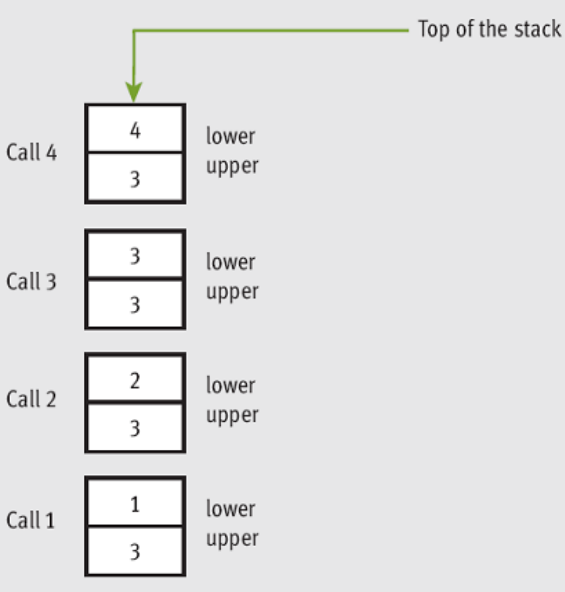
Amount of memory needed for a loop does not grow with the size of the problem's data set.

Python Virtual Machine (PVM) reserves an area of memory for the call stack.
For each call of a function, the PVM must allocate on the call stack a stack frame, which contains:

- Values of the arguments
- Return address for the particular function call
- Space for the function call's return value

When a call returns, return address is used to locate the next instruction, and stack frame is deallocated.

For example, when we call the function displayRange (1, 3), below is the memory allocated.



Tutorial 4

1. Write down the trace diagram when we call the function below sum (1, 4, 0).

```
def sum(lower, upper, margin):
    """Returns the sum of the numbers from lower to upper,
    and outputs a trace of the arguments and return values
    on each call."""
    blanks = " " * margin
    print(blanks, lower, upper)
    if lower > upper:
        print(blanks, 0)
        return 0
    else:
        result = lower + sum(lower + 1, upper, margin + 4)
        print(blanks, result)
        return result
```

2. Explain what happens when the following recursive function is called with the value **4** as an argument:

```
def example(n):
    if  n > 0:
        print n
        example (n-1)
```

3. Explain what happens when the following recursive function is called with the value **4** as an argument:

```
def example(n):
    if  n > 0:
        print n
        example(n)
    else:
        example(n-1)
```

4. Explain what happens when the following recursive function is called with the values **"hello"** and **0** as arguments:

```
def example(aString, index):
    if  index < len(aString):
        example(aString, index+1)
        print  aString[index]
```

5. Explain what happens when the following recursive function is called with the values **"hello"** and **0** as arguments:

```
def example(aString, index):
    if  index == len(aString):
        return ""
    else:
        return  aString[index] + example(aString, index + 1)
```

Assignment 4

1. Write a recursive function that accepts an integer argument, n. The function should display n lines of asterisks on the screen, with the first line showing 1 asterisk, the second line showing 2 asterisks, up to the nth line which shows n asterisks.

2. Design a function that accepts an integer argument and returns the sum of all the integers from 1 up to the number passed as an argument. For example, if 50 is passed as an argument, the function will return the sum of 1, 2 , 3, . . . ,50. Use recursion to calculate the sum.

3. (2013 HCI J1 Block Test)

   Write a menu-driven automatic teller program in which the user's bank balance is initialized to $1,000. The user should be allowed to perform as many transactions as he or she wishes from the menu.

   1. Deposit
   2. Withdrawal
   3. See balance
   4. Quit

   For the withdrawal option, the user should be prompted to select from the choices $50, $100, $200, and $500. (Do not allow user to overdraw.) After each transaction, the program should print the current balance. On exiting, the program should print a courteous message. In this program, design a function for each option.

4. Write a program that asks the user to enter 3 test scores. The program should display a letter grade for each score and the average test score. Write the following functions in the program:
   - `calc_average` -- This function should accept three test scores as arguments and return the average of the scores.
   - `determine_grade` -- This function should accept a test score as an argument and return a letter grade for the score, based on the following grading scale:

Score	Letter Grade
90 – 100	A
80 – 89	B
70 -- 79	C
60 – 69	D
Below 60	F