Chapter 11
Introduction to Structured Analysis

11.1 Introduction

In the preceding chapters what we have seen is typical of the procedural approach to programming. In the next chapter we will begin to explore the definition of functions which allow a more modular style of programming.

In this chapter we will introduce an approach to the analysis and design of software systems called structured analysis/design. Within this area there are many specific and well-documented methods including Structured Systems Analysis and Design Method (SSADM) and the Yourden structured method. However, we will not focus on any one specific approach; instead we will outline the key ideas and the two basic elements of most structured analysis methods: functional decomposition and data flow analysis. We will then present flowcharts for designing algorithms.

11.2 Structured Analysis and Function Identification

The structured analysis methods typically employ a process-driven approach (with a set of prescribed steps or stages) which in one way or another consider what the inputs and outputs of the system are and how those inputs are transformed into the outputs. This transformation involves the application of one or more functions. The steps involved in structured analysis identify these functions and will typically iteratively break them down into smaller and smaller functions until an appropriate level of detail has been reached. This process is known as functional decomposition.

Although simple Python programs may only contain a sequence of statements and expressions, any program of a significant size will need to be structured such that it can be:

- understood easily by other developers,
- tested to ensure that it does what is intended,
- maintained as new and existing requirements evolve,

© Springer Nature Switzerland AG 2023

J. Hunt, *A Beginners Guide to Python 3 Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-35122-8_11

- debugged to resolve unexpected or undesirable behaviour.

Given these requirements it is common to want to organize your Python program in terms of functions and sub functions.

Functional decomposition supports the analysis and identification of these functions.

11.3 Functional Decomposition

Functional decomposition is one way in which a system can be broken down into its constituent parts. For example, for a computer payroll system to calculate how much an hourly paid employee should receive it might be necessary to:

1. load the employee's details from some form of permanent storage (such as a file or a database),
2. load how many hours the employee has worked for that week (possibly from another system that records the number of hours worked),
3. multiple the hours worked by the employee's hourly rate,
4. record how much the employee is to be paid in a payroll database or file,
5. print out the employee's pay slip,
6. transfer the appropriate funds from the company's bank account to the employees bank account,
7. record in the payroll database the everything has been completed.

Each of the above steps could represent a function performed by the system.

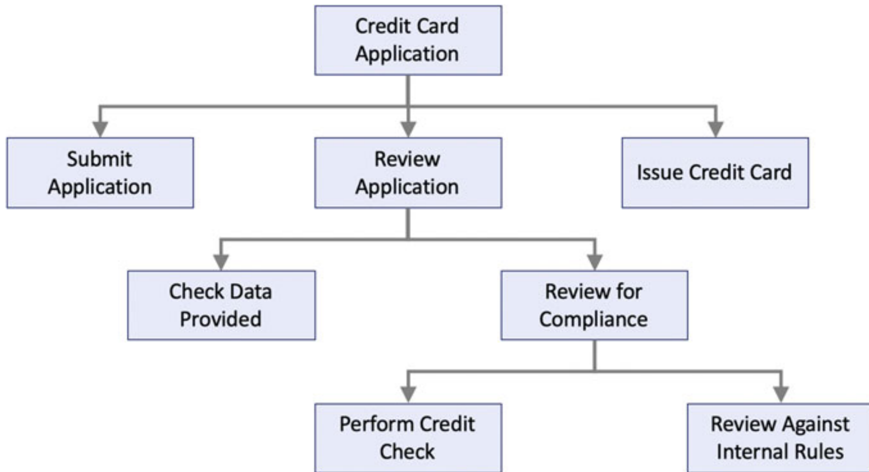
These top-level functions could themselves be broken down into lower-level functions. For example, printing out the employees payroll slip may involve printing their name and address in a particular format, printing the employee number, social security number, etc., as well as printing historical information such as how much they have been paid in the current financial year, how much tax they have paid, all in addition to printing the actual amount they are being paid.

This process of breaking down higher-level functions into lower-level functions helps with:

- testing the system (functions can be tested in isolation),
- understanding the system as the organization of the functions can give meaning to the code, as well as allowing each function to be understood in isolation from the rest of the system,
- maintaining the system as only those functions that need to be changed may be affected by new or modified requirements,
- debugging the system as issues can be isolated to specific functions which can be examined independently of the rest of the application.

It is also known as a top-down refinement approach. The term top-down refinement (also known as stepwise design) highlights the idea that we are breaking down a system into the sub-systems that make it up.

It is common to represent the functions identified in the form of a tree illustrating the relationships between the higher-level functions and lower-level functions. This is illustrated below:



This diagram illustrates how a credit card approval process can be broken down into sub functions.

11.3.1 Functional Decomposition Terminology

The key terms used within functional decomposition are:

- **Function**. This is a task that is performed by a device, system or process.
- **Decomposition**. This is the process by which higher-level functions are broken down into lower-level functions where each function represents part of the functionality of the higher-level function.
- **Higher-Level Function**. This is a function that has one or more sub functions. This higher-level function depends on the functionality of the lower-level functions for its behaviour.
- **Sub Function**. This is a function that provides some element of the behaviour of a higher-level function. A sub function can also be broken down into its own sub functions in a hierarchical manner. In the above diagram the *Review for Compliance* function is both a sub function and has its own sub functions.

- **Basic Function**. A basic function is a function that has no smaller sub functions. The *Perform Credit Check* and *Review Against internal Rules* functions are both basic functions.

11.3.2 Functional Decomposition Process

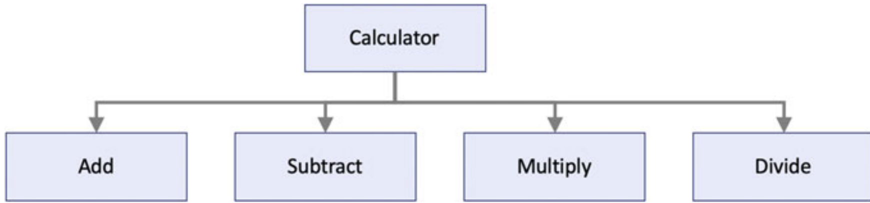
At a very high level, functional decomposition consists of a series of steps such as those outlined below:

1. Find/identify the inputs and outputs of the system.
2. Define how the inputs are converted to the outputs. This will help identify the top most, high-level function(s).
3. Look at the current function(s) and try to break them down into a list of sub functions. Identify what each sub function should do, and what its inputs and outputs are.
4. Repeat Step 2 for each function identified until the functions identified can't or should not be decomposed further.
5. Draw a diagram of the function hierarchy you have created. Viewing the functions and their relationships is a very useful thing to do as it allows developers to visualize the system functionally. There are many computer-aided software engineering (CASE) tools that help with this but any drawing tool (such as Visio) can be used.
6. Examine the diagram for any repeating functions. That is, functions that do the same thing but appear in different places in the design. These are probably more generic functions that can be reused. Also examine the diagram to see if you can identify any missing functions.
7. Refine/design the interfaces between one function and another. That is, what data/information is passed to and from a function to a sub function as well as between functions.

11.3.3 Calculator Functional Decomposition Example

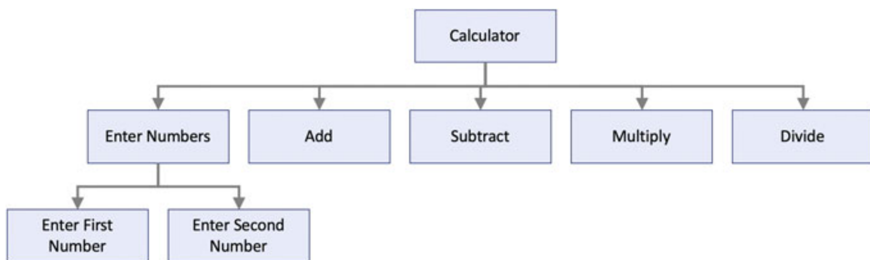
As an example of functional decomposition let us consider a simple calculator program.

We want this program to be able to perform a set of mathematical operations on two numbers, such as add, subtract, multiple and divide. We might therefore draw a functional decomposition diagram (or FDD) such as:



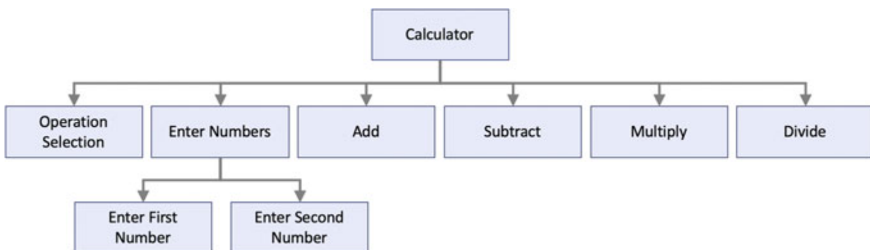
This illustrates that the calculator function can be decomposed into add, subtract, multiply and divide functions.

We might then identify the need to enter the two numbers to be operated on. This would result in one or more new functions being added:



We might also identify the need to determine which numerical operation should be performed based on user input. This function might sit above the numerical functions or alongside them. This is in fact an example of a design decision that the designer/developer must make based on their understanding of the problem and how the software will be developed/tested/used, etc.

In the following version of the functional decomposition diagram the *operation selection* function is placed at the same level as the numerical operations as it provides information back to the top-level function.



11.4 Functional Flow

Although the decomposition hierarchy presented in the functional decomposition diagram illustrates the functions and their hierarchical relationships, it does not capture how the data flows between the functions or the order in which the functions are invoked.

There are several approaches to describing the interactions between the functions identified by functional decomposition including the use of pseudo-code, data flow diagrams and sequence diagrams:

- **Pseudo-code**. This is a form of structured English that is not tied to any particular programming language but which can be used to express simple ideas including conditional choices (similar to if statements) and iteration (as typified by looping constructs). However, as it is a pseudo-language, developers are not tied to a specific syntax and can include functions without the need to define those functions in detail.
- **Data Flow Diagrams**. These diagrams are used to chart the inputs, processes and outputs of the functions in a structured graphical form. A data flow diagram typically has no control flow; there are no decision rules and no loops. For each data flow, there must be at least one input and one end point. Each process (function) can be refined by another lower-level data flow diagram, which subdivides this process into sub-processes.
- **Sequence Diagrams**. These are used to represent interactions between different entities (or objects) in sequence. The functions invoked are represented as being called from one entity to another. Sequence diagrams are more typically used with object-oriented systems.

11.5 Data Flow Diagrams

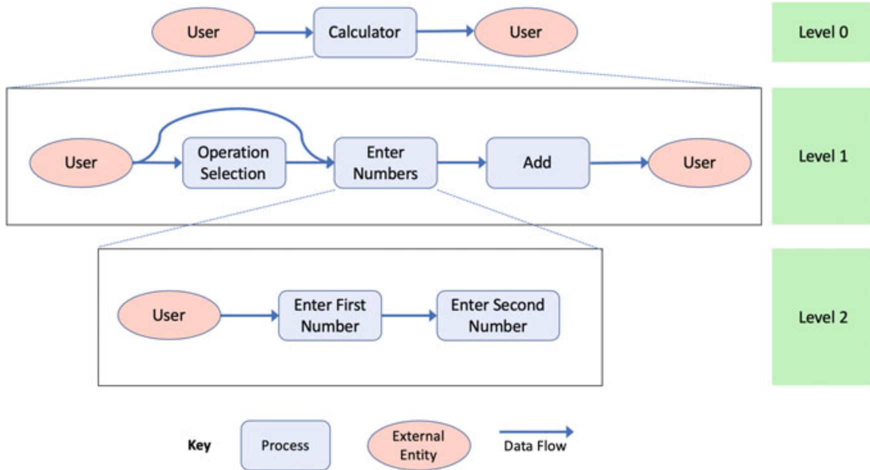
A data flow diagram consists of a set of inputs and outputs, processes (functions), flows, data stores (also known as warehouses) and terminators.

- **Process**. This is the process (or function or transformation) that converts inputs into outputs. The name of the process should be descriptive indicating what it does.
- **Data Flow**. The flow indicates the transfer of data/information from one element to another (that is a flow has a direction). The flow should have a name that suggests what information/data is being exchanged. Flows link processes, data stores and terminators.
- **Data Store/Warehouse**. A data store (which may be something such as a file, folder, database or other repository of data) is used to store data for later use. The name of the data store is a plural noun (e.g., employees). The flow from the data store usually represents the reading of the data stored in the data store, and the flow

to the waredata storehouse usually expresses data entry or updating (sometimes also deleting data).

- **Terminator**. The terminator represents an external entity (to the system) that communicates with the system. Examples of entities might be human users or other systems, etc.

An example of a data flow diagram is given below using the functions identified for the calculator:



In this diagram the hierarchy of DFDs is indicated by the levels which expand on how the function in the previous level is implemented by the functions at the next level. This DFD also only presents the data flow for the situation where the user selects the `add` function as the numerical operation to apply.

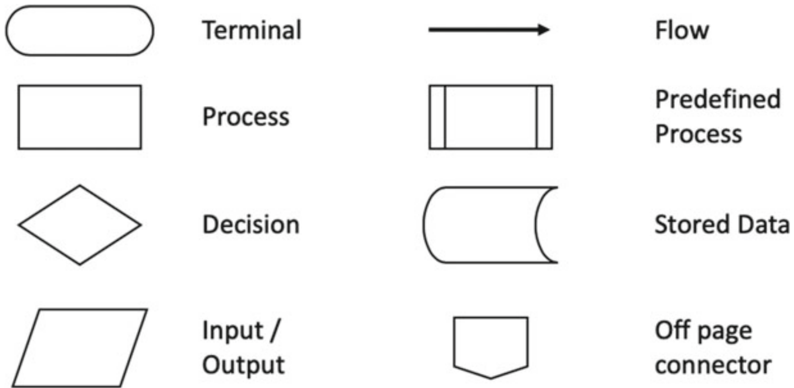
11.6 Flowcharts

A flowchart is a graphical representation of an algorithm, workflow or process for a given problem.

Flowcharts are used in the analysis, design and documentation of software systems. As with other forms of notation (such as DFDs) flowcharts help designers and developers to visualize the steps involved in a solution and thus aid in understanding the processes and algorithms involved.

The steps in the algorithm are represented as various types of boxes. The ordering of the steps is indicated by arrows between the boxes. The flow of control is represented by decision boxes.

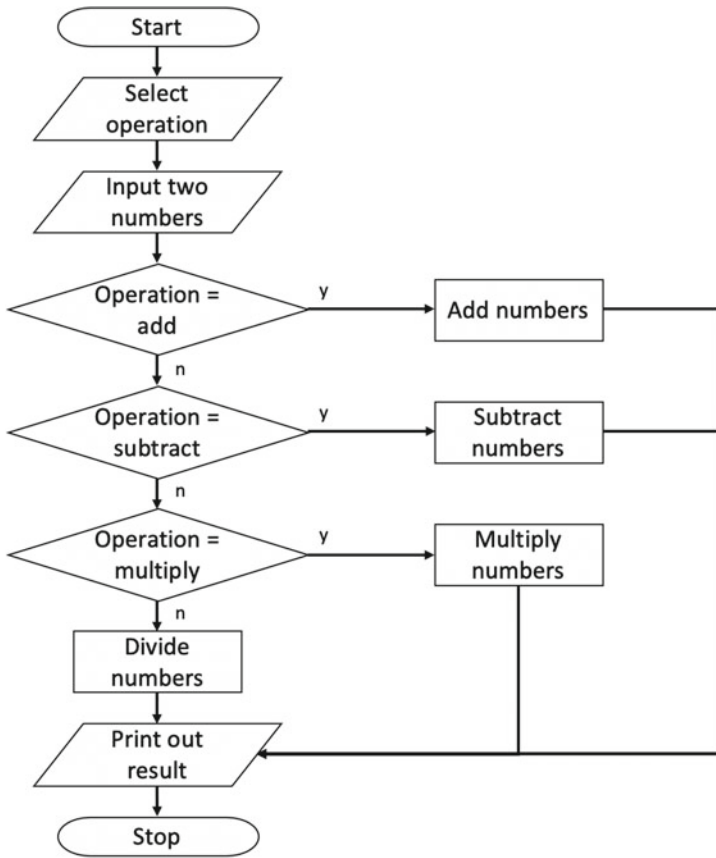
There are a number of common notations used with flowcharts, and most of those notations use the following symbols:



The meaning of these symbols is given below:

- **Terminal**. This symbol is used to indicate the start or end of a program or subprocess. They usually contain the words 'Start', 'End' or 'Stop' or a phrase indicating the start or end of some process such as 'Starting Print Run'.
- **Process**. This symbol represents one or more operations (or programming statements/expressions) that in some way apply behaviour or change the state of the system. For example they may add two numbers together, run some form of calculation or change a Boolean flag, etc.
- **Decision**. This represents a decision point in the algorithm; that is it represents a decision point which will alter the flow of the program (typically between two different paths). The decision point is often represented as a question with a 'yes'/'no' response, and this is indicated on the flowchart by the use of 'yes' (or 'y') and 'no' (or 'n') labels on the flowchart. In Python this decision point may be implemented using an if statement.
- **Input/Output**. This box indicates the input or output of data from the algorithm. This might represent obtaining input data from the user or printing out results to the user.
- **Flow**. These arrows are used to represent the algorithms order of execution of the boxes.
- **Predefined Process**. This represents a process that has been defined elsewhere.
- **Stored Data**. Indicates that data is stored in some form of persistent storage system.
- **Off page connector**. A labeled connector for use when the target is on another page (another flowchart).

Using the above symbols we can create a flowchart for our simple integer calculator program:



The above flowchart shows the basic operation of the calculator; the user selects which operation to perform, enters the two numbers and then depending upon the operation the selected operation is performed. The result is then printed out.

11.7 Data Dictionary

Another element commonly associated with structured analysis/design is the data dictionary. The data dictionary is a structured repository of data elements in the system. It stores the descriptions of all data flow diagram data elements. That is it records details and definitions of data flows, data stores, data stored in data stores and the processes. The format used for a data dictionary varies from method to method and project to project. It can be as simple as an Excel spreadsheet to an enterprise-wide software system such as Semanta (https://www.semantacorp.com/data-dictionary).

11.8 Online Resources

There are many online resources available that discuss functional decomposition, both from a theoretical and a practical Python-oriented point of view including:

- https://en.wikipedia.org/wiki/Structured_analysis Wikipedia Structured Analysis Page.
- https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design Wikipedia page on Top Down and Bottom Up design.
- https://en.wikipedia.org/wiki/Edward_Yourdon#Yourdon_Structured_Method Wikipedia page on the Yourden method.
- https://en.wikipedia.org/wiki/Structured_systems_analysis_and_design_method Wikipedia page on SSADM.
- https://en.wikipedia.org/wiki/Functional_decomposition The Wikipedia page on Functional Decomposition.
- https://docs.python.org/3/howto/functional.html The Python standard documentation on functional decomposition.
- https://en.wikipedia.org/wiki/Data-flow_diagram Wikipedia page on Data Flow Diagrams (DFDs)
- https://en.wikipedia.org/wiki/Sequence_diagram Wikipedia page on Sequence Diagrams.
- https://en.wikipedia.org/wiki/Data_dictionary Wikipedia page on Data Dictionaries.
- https://en.wikipedia.org/wiki/Flowchart Wikipedia Flowchart page.