Introduction to Programming

using Python

Table of contents

Preface

This book is intended for teachers to teach an introductory course on computer programming. The book should serve as reference textbook for both students and teachers, slides should help teachers teach the course.

For absolute beginners, it is recommended to learn computer programming, if possible, within an interactive classroom type setup with demos and labs. Although, students and learners can use it for learning to program, but computer programming is a practical skill which needs demos and live interactive sessions.

For learners, with existing knowledge of programming, this should provide an independent perspective to structure knowledge related to computer programming.

The book is available to download as a package for offline usage from this link, which contains

- Book: html and pdf formats
- Slides: pdfs in handout and presentation formats

All the content has been created using Quarto, a tool for automating publishing, documentation and more. The source code is at this GitHub link and there is an icon in the html format of the book in the tools section, towards the top area. This provides the option to edit the content as required.

The website shunya.acad is a placeholder for more general discussion on approach to create teaching content.

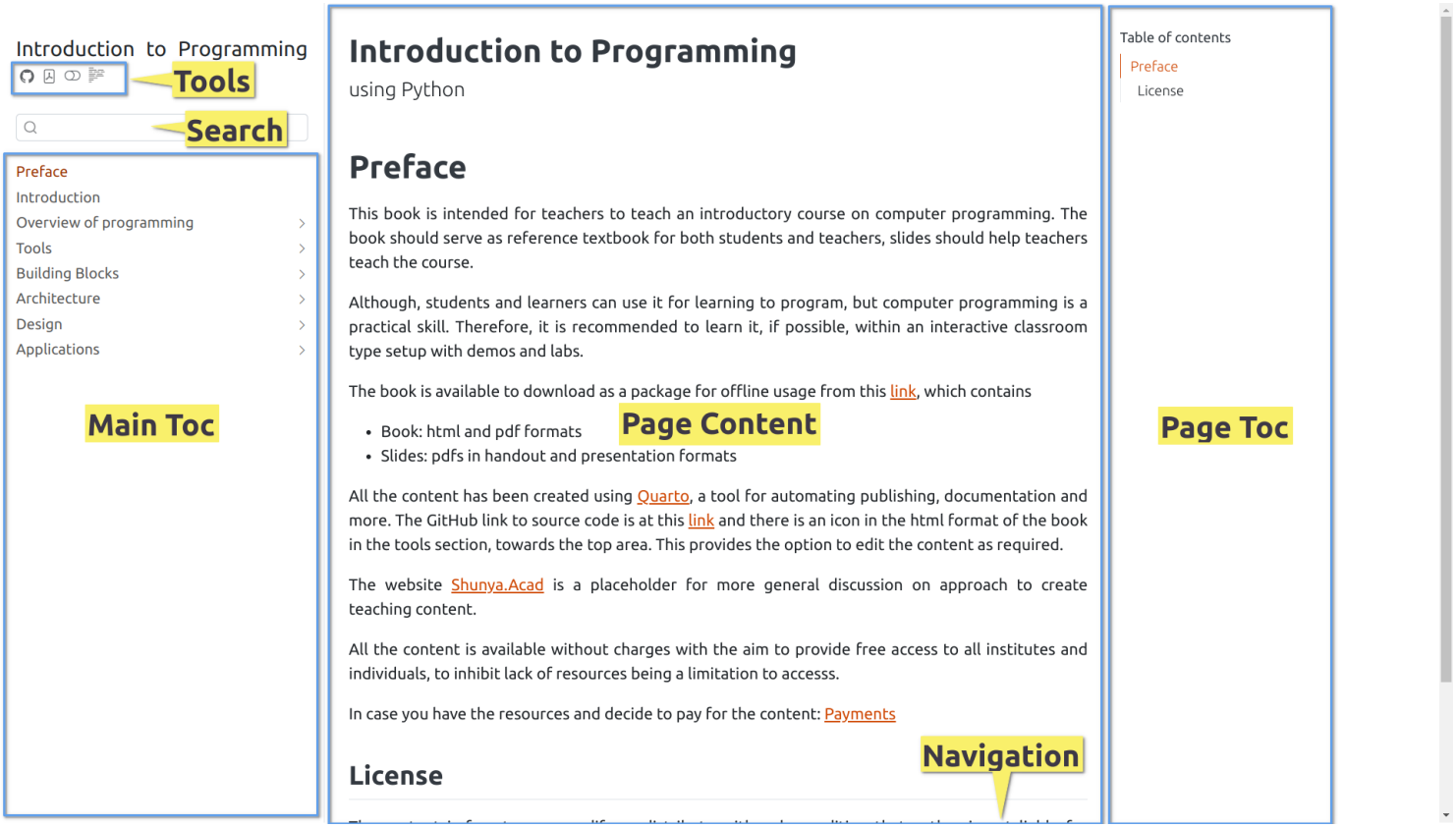
All the content is available without charges with the aim to provide free access to all institutes and individuals, to inhibit lack of resources being a limitation to access.

In case you have the resources and decide to pay for the content: Payments

License

Introduction

Book Anatomy (html)



Background

Use of technology is increasing and impacting more and more domains. Most of the workforce in cities use technology for work and personal work. This provides its own challenges but also opportunities, and requirements to use those opportunities. Introduction to programming is one of those requirements.

Learning computer programming serves few important purposes.

- Use technology more efficiently and resourcefully
- Provides a common language for communicating with those developing solutions
    - Be aware of the possibilities during communication
- Create your own solutions

Therefore it makes sense to make introduction to computers, operating systems, computer programming and its basic applications, part of the school education than college for everyone.

Objectives

Though this book can be read to learn to program or get an independent perspective to programming, the main goal of this book is to serve as a supplement to the course taught in class. This is because for beginners learning without demonstrations adds unnecessary complexity. On the other hand, in class setup demos, experimentation and QnA decrease complexity and aid understanding.

For **learners**, this should help as reference notes to quickly find information as needed while learning and years after initial learning.

For **instructors**, this book along with slides should be helpful to teach the course. The content is available with freedom to use, modify and distribute. It should help focus efforts on creating other material (assignments, quizes, exams, labs, projects) and student interaction and monitoring.

The source code for the book and slides is available to edit as required.

The offline version for the content is available too and can be downloaded as required.

Link to all the resources is provided in preface/home page of the book.

Content

The attempt is to abstract the core concepts and contexts related to computer programming, which acts as a framework to learn not just Python but any programming language.

The focus is on concretizing the framework to understand evolution of programming. The concepts involved are generally the same for most part.

There are certain topics which are avoided in beginner courses. These have been included with an attempt to reduce the complexity but still give enough exposure. This is with the intent that this will provide a better understanding of how developers provide solutions which eventually should improve the usage of those solutions and writing programs in general.

The part **Overview of programming** sets the outline for the book and programming in general. For those who are new to programming, this will not make sense in first reading but will clearly layout the outline for learning. Second reading is recommended after completing the remaining sections when many of the components will be understood better. This is also recommended for classroom environment, going through the overview part at the end will provide a summary and better understanding of the components compared to the coverage in initial lectures.

Similarly, **Tools** part has a lot of information which will not be useful in the beginning but becomes clearer as you progress forward in the course and use the tools.

Why Python?

Popular & Generic

Python is one of the most popular generic programming language at the moment and this increases the chances of using the language you learn programming with.

Python is a generic programming language with contributions in almost every domain, specially compared to other languages. So chances are higher that you will find an existing solution, irrespective of the field you work in.

Another advantage is that there is lot of help available online as there are more users/developers.

Ease of use

Python has well designed syntax which is clear and easy to write and read which makes it easy to focus on what you need the code to do.

Python provides features like garbage collection which does the memory management in background making it easier to code.

Terminology

- **Programming** is used in context of **computer programming**
- **Program** is used in context of **computer programs**
- **Pseudo code** has been used at several places. Pseudo code is structured natural language code without using the specified syntax, to explain what the code does. Spaces, line breaks etc. are used to make it easier to understand.
- **Flow diagrams** are used to explain schematics of code through visualization
- **Symbols**
  - `[]` typically used to represent optionality
    - e.g. item[s] means 's' is optional as there can be one or more item[s]
  - `<>` is typically used as place holder
    - e.g. `cp <src path> <dst path>` implies source and destination paths are to be entered as needed

Part I

Overview of programming

1 Background

1.1 Definition

Computer hardware is a machine which can be configured to do several tasks as needed.

A **program**, in context of computers, is a set of instructions for the computer. A program is also referred to as **software**.

Configuring the computer hardware to do some tasks to solve a problem is called **programming**.

1.2 Hardware

There are a lot of good short videos on web which you get by searching, e.g. "how do computers work". It is worth spending some hours on this general search given the amount of time spent with technology.

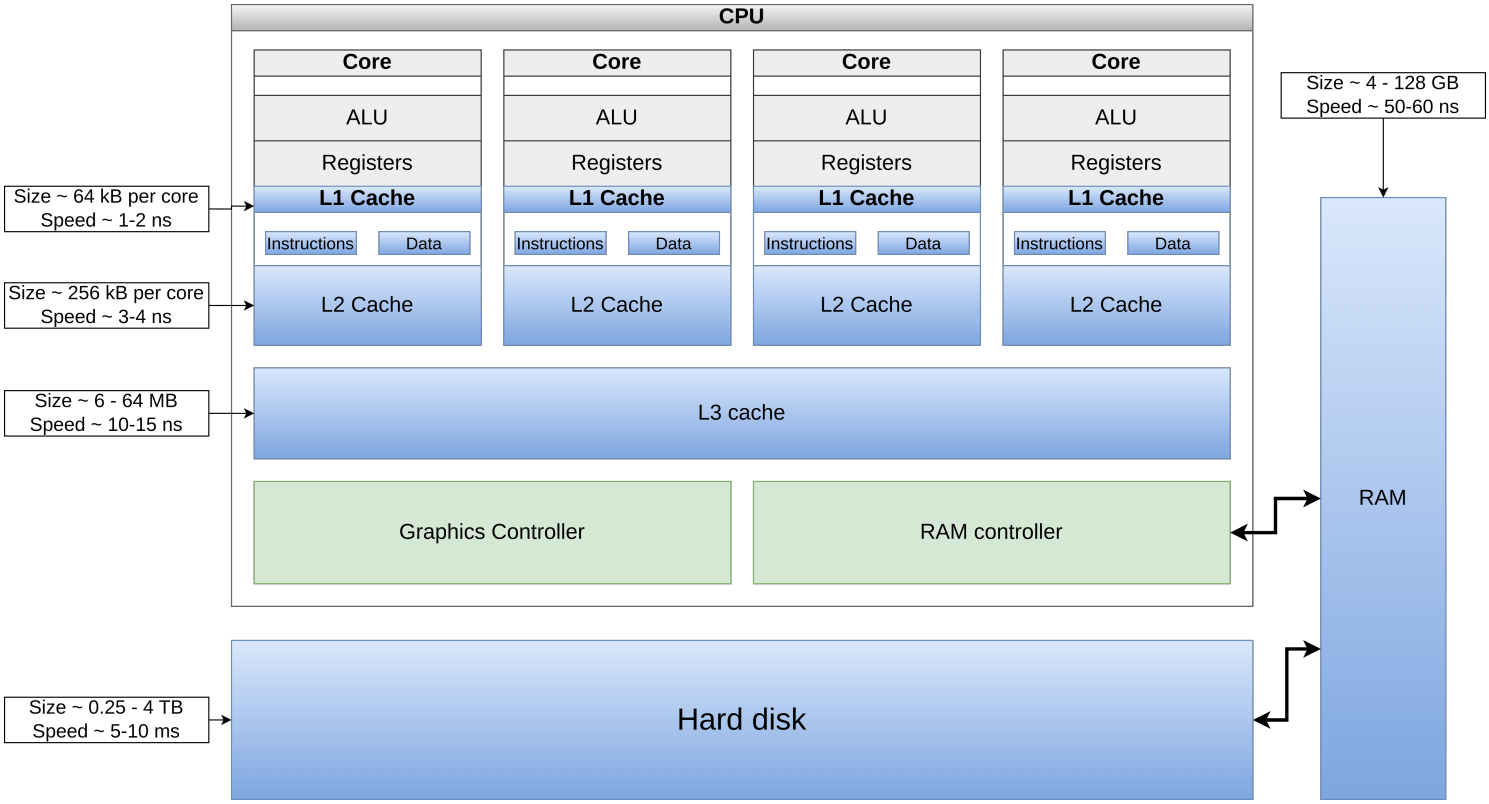
Understanding everything in detail is beyond the scope. Still it is helpful to understand the flow of information and operations at a high level.

Computers are based on binary arithmetic. Computer hardware comprises of a large number of electronic circuits which communicate with each other through electrical signals, which essentially means communicating in binary, 0 or 1. The hardware is designed to understand different sequences of binary signals to perform different tasks.

Data and instructions are stored in binary format. 1 bit is the atomic unit of storage which stores 0 or 1. Instructions are also data in terms of computer memory, but usually term "data" is used for static information other than instructions.

At most basic level, hardware operates using below basic elements

- CPU: central processing unit or processor
    - ALU (Arithmetic logic unit): operates on very small amount of binary data to perform arithmetic and logical operations
    - Controller unit: controls the input/output signals from/to different parts of the computer
    - Registers: smallest memory, closest to processing
    - Cache (L1, L2, L3): next level memory for preprocessing
- RAM: random access memory
    - larger than CPU cache but smaller than hard disk
    - slower than CPU cache but faster than hard disk
    - all data slots are equally accessible in same time
- Hard drive: long term storage
    - largest in size but slowest
- Input/output devices: keyboard, mouse, monitor, speakers



The CPU decides the architecture of information flow in a computer. Architecture has 2 main components related to design of conversion to machine code

- **Instruction set architecture (ISA)** that processor follows, e.g. x86-64, ARMv8-64
- **Processing Width**: maximum number of bits the processor can process at a time
  - currently most CPU's support 64 bit, earlier it was 32 bit
  - the 64 in x86-64, ARMv8-64 is 64 bit width

1.3 Programming hardware

The computer hardware understands machine code, which is in binary, but it is not feasible to write, read and modify the machine code.

For illustration, if `010011001010 ...` is a set of instruction in binary to tell the computer to beep, in programming language you type `beep`, the code will be mapped to instruction in binary and sent to processor.

Therefore programming language is needed. The key idea is to have a set of instructions in natural language which can be mapped to different set of instructions in machine code.

2 Programming languages

2.1 Overview

Machines understand machine code which is in binary and is hard to work with. While writing instructions for the machine, it is easier to understand and work with natural language like english. Therefore, a mapping is needed to translate natural language like english to machine code.

The general idea of **mapping natural language to machine code (binary)** lead to evolution of programming languages.

There are multiple ways to solve the mapping problem based on the intended use case. Therefore there are numerous programming languages.

Typically a programming language refers to 2 distinct parts.

- **Specification**: rules for writing code
  - **lexicon**: allowed keywords, symbols, characters
  - **syntax**: rules of arranging lexicons and creating new names (variables)
  - **semantics**: rules to assign meaning to different combinations of lexicons and syntax
- **Implementation**: A system application that implements the specification, like Compiler or Interpreter, to convert the natural language code into machine code and execute.
  - Referred to as **'language engine'** henceforth.

Some common type of language engines, applications to create this mapping or conversion are

- **Assembler**
- **Compiler**
- **Interpreter**

Note that the implementation for a language, the engine, is itself a program in machine code stored on hardware. A programming language defines the mapping rules, specifications, which can be implemented using any of the above approaches, but usually a particular combination becomes more popular.

For example Python has many implementations. The official and most popular is CPython which is interpreter written mostly in C. There are other implementations as well, which have their own specifications which are similar but not the same.

Another example is C++, the specifications are maintained and revised by a central organization, The Standard C++ Foundation. Then there are various organizations that implement the compiler using the latest specifications. The GNU compiler collection (GCC), Microsoft Visual C++ (MSVC), CLang (LLVM) etc.

2.2 Types

Languages can be classified into below main categories in order of decreasing complexity and based on method of conversion.

- *low level languages*
    - **assembly languages**
- *high level languages*
    - **compiled languages** (*compiler*): C, C++, Go
    - **interpreted languages** (*interpreter*): shell (bash, etc.)
        - **hybrid**: Python
            - **JIT**: Just in Time compilation, e.g. Java, JavaScript, C#

2.2.1 Assembly languages

Assembly languages have very limited set of specifications and are difficult to code in. They are machine specific and used to provide basic configuration to hardware, e.g. BIOS (basic input output setup) on computers. They are closest to the hardware and hence provide access to all components. They are very fast as they are directly translated to machine code using an assembler and specifications to be checked and converted are very limited.

Since there are very few mappings, every instruction has to be explicit, therefore the code written in assembly language is much longer compared to that in high level languages.

2.2.2 High level languages

High level languages is the next step in the mapping problem. They define their own set of specifications (lexicon, syntax and semantics) to provide a lot more features for writing programs, e.g. data types, control flow mechanisms, ways to define functions and objects, managing storing data on memory, managing input/output etc.

For example in C/C++ memory management and data types have to be dealt with by the programmer manually, but in Python a lot of these features are provided as default where programmer does not have to deal with them.

Since there are many pre-built instructions built in, the code written in high level languages tend to be shorter and easier to understand, compared to low level languages and machine language.

2.2.2.1 Compiled languages

Compiled languages solve the mapping problem using a **compiler**.

The compiler reads the whole program and converts it to machine code before execution. This process is also referred to as Ahead of time compilation (**AOT**).

The compiler is dependent on a few major factors

- Hardware architecture (this is dependent on CPU primarily)
    - Instruction set: x86, ARM64, …
    - Processing size: 64 bit or 32 bit
- Operating system (OS)

Therefore compiled languages like C/C++ have compilers available for most common architecture and OS combinations. The compiled machine code produced is different for different combinations.

- **Pros**
  - **Fast**
    - machine code is generated as standalone output
- **Cons**
  - **Portability**
    - have to be compiled for different OS + Hardware combinations
    - this is more of an inconvenience, as compilers for most popular combinations are available
  - **Difficulty**
    - compared to hybrid interpreter languages which are higher level with more features towards ease of use
    - have higher development time, takes more time to code

How to write the compiler itself?

This seems to be a chicken and egg problem but it is not. Imagine there are no high level languages available and the first compiler for C language has to be written. The first compiler can be written in assembly language on a given system. This gives the machine code for C compiler. But now you can write compiler for any high level language in C and get the machine code for that compiler by compiling it with C compiler. From there on any compiler can be written in any compiled language and turned to machine code for needed systems.

2.2.2.2 Interpreted languages

The interpreter produces machine code at run time and can run code interactively. Interpreter takes care of system architecture and OS dependencies so can be run on any machine as long as the interpreter can be installed on that machine.

Interpreters are of different types depending on implementation for a language. In simplest form they execute one instruction at a time, like most shell programs do, e.g. bash or zsh.

2.2.2.2.1 Hybrid

Languages like Java, C# and CPython (Official and most used Python version) use a hybrid approach. The mapping is done in 2 main steps

- **compilation step**: source code is checked and compiled into intermediate **bytecode**
- **interpreter step**: the bytecode is interpreted into machine code and run at runtime

The run time behavior is still same as interpreter, in the sense that machine code is produced at run time rather than at compile time.

- **Pros**
  - **Portability**
    - same program can be run on different machines by just installing the language interpreter
  - **Ease of use**
    - generally these languages are higher than compiled languages like C, C++
    - time to write code is reduced
- **Cons**
  - **Slower**
    - program is converted to machine code at run time
    - this depends on the use case, specially considering the increase in hardware capacity, for most of common tasks this can be ignored

2.2.2.2.1.1 JIT

Some implementations of some languages use **just in time compilation (JIT)** to optimize the interpreter.

The key idea is to compile certain pieces of code to optimized machine code at run time.

This is more of an optimization technique for increasing speed, but this is still slower compared to compiled languages.

Examples of language implementations that use JIT:

- **Java**: Java Virtual Machine (**JVM**)
- **C#**: Common language runtime (**CLR**)

2.3 Popular languages

Use Case	Python	R	JavaScript	C	C++	Java
Automation	x	x		x	x	
Data analysis	x	x		x	x	
Scientific Computing	x	x		x	x	
Web dev	x		x	x	x	x
System App dev	x		x	x	x	x
Mobile App dev	x		x	x	x	x
Cloud dev	x		x	x	x	x
Network dev	x			x	x	x

2.3.1 Desktop & Mobile applications

Major operating systems create their own language and frameworks to develop applications to run on the os. This is changing now to target cross platform development, i.e. application built using a framework + language combination can be built for multiple devices and OS, just like compilers.

Frameworks are an extra layer of functionalities and ways of binding them for a specific hardware architecture + os combination. For example, for creating graphical user interfaces building blocks are prebuilt and can be glued into the main application program without having to write them from scratch.

The OS specific languages and platforms are now shifting to being cross platform.

Language	Device	OS	Framework
C#, Visual Basic, F#	Desktop	Windows (Microsoft)	.NET, XAMARIN
Swift	Desktop, Mobile	MacOS, IOS (Apple)	SwiftUI
C++	Cross-platform	Cross-platform	QT
Dart	Cross-platform	Cross-platform	Flutter
Java	Cross-platform	Cross-platform	Multiple frameworks
Kotlin	Cross-platform	Cross-platform	Multiple frameworks

2.3.2 Websites (web dev)

Web technologies have become popular with the creation and development of browsers and web networks.

- Front end (GUI) is based on HTML, CSS and JavaScript
- Backend functionalities: like databases, analytics can be developed in language of choice

In practice there are many frameworks available for development which provide standard templates of pre built code for different functionalities.

Name	Description
HTML	Markup language to structure user interface structure and text
CSS	Responsible for the user interface styling
JavaScript	Responsible for the functionality behind web pages, e.g. click button actions, form submissions, ...
TypeScript	Stricter version of Javascript which is static typed

How do browsers run code?

Browsers are primarily running **JavaScript** using their own set of mapping engines.

Earlier the browser JavaScript engines were using interpreter approach but now have shifted to JIT approach like Java.

- Google Chrome: V8 engine
- Mozilla Firefox: SpiderMonkey
- Apple Safari: JavaScriptCore
- Microsoft Edge: Chakra

2.3.3 General purpose

2.3.3.1 C and C++

C and C++ are very popular compiled languages for writing high performance code. They provide access to most of the functionality hardware has to offer. Most of the critical applications like operating systems, compilers, virtual machines are written in C/C++.

These have the steepest learning curve as the programmer has to know the how the hardware works to use these languages and most of the interactions like managing data stored on RAM, directing CPU for parallel processing have to be coded by the programmer.

C is the older among the two with lesser features available. Everything has to be built from scratch or using some one else's code.

C++ is a newer version of C with OOP, few changes in syntax and a lot more features. Also C++ is under active development.

Both are compiled languages and compilers are available from various sources, most common being GNU GCC, which is a collection of compilers for different hardware architecture and OS combinations.

Although these are not used directly in all use cases but much of the functionality under the hood uses these for performance reasons. For example, much of the libraries for scientific computing and data analysis in languages like

Python and R are developed using C/C++ and a wrapper is implemented in Python or R. All major operating systems are written using C/C++ in part or in full.

2.3.3.2 Python

Python is a general purpose programming language. The official and most common version of Python that is used and covered in this course is implemented in C, and therefore also referred to as CPython.

Python is very popular language with applicability in most of the application areas. Also it has a lot of pre built code available for majority of the regular tasks.

2.4 Programming paradigms

÷ A programming paradigm defines the approach or style of solving problems through programming.

Programming paradigms are grouped into 2 main categories.

- Declarative: focusses on what to solve
    - Logic programming
    - Functional programming
    - Data driven programming
- Imperative: focusses on how to solve the problem
    - Procedural programming
    - Object oriented programming
    - Structured programming

A language, through design of specifications and implementation, can support one or more of paradigms.

Almost all popular programming languages support multiple paradigms and hence the term multi-paradigm is used. e.g Python, C, C++, Java, Javascript etc.

For example, in Python, same problem can be solved using functional programming or OOP. Python provides all the options.

2.5 General structure

This is an attempt to structure all knowledge related to programming, including practical aspects.

- **Language specifications**
    - **Building blocks**: *specifications for elements and blocks of code*
    - **Architecture**: *specifications for writing programs*
        - means and specifications to combine elements and blocks to build programs
        - management of execution of blocks (under the hood)
- **Design**: *knowledge of how to write good programs*
- **Tools**: *needed to write, test, debug and run programs*

Language specification

Building blocks

- Basic specifications
  - Lexicons
  - Variables
  - Comments
- Data types and operations
- Control flow blocks
- Functions
- OOP
- Special features

Architecture

- Environment = Namespace + Scope
- Scripts and packages
- External packages
- Language engine
- Debugging

Design

- Components
  - Design patterns
    - Map reduce
  - DSA
  - Regular expressions
- Frameworks
- Workflow

Tools

- Command line interface
- Version control system
- Language installation
  - Engine
  - External packages
  - Virtual environments
- Editor

Almost all programming languages can be fitted to this generic structure/framework, therefore getting started with any language will become easier if you keep this structure in mind. This book is structured to follow this structure as well.

The wiki link provides a comparison of different languages where you can use this structure to look for relevant basic syntax.

The implementation and implications will differ for some pieces from language to language which you learn by experience in coding in it as use case arise. For example, in Python you do not need to worry about hardware management as it is taken care by the interpreter, but in C/C++ you have to manage them yourself. The scope rules are different from language to language.

2.6 Underlying Concepts

The 3 basic concepts (means of abstraction, encapsulation and combination) have been foundational in development of computer science from hardware to software. Once you start seeing this pattern in every aspect of definition of a programming language, you will understand and implement programming better.

Underlying concepts

- Means of **abstraction**
  - building blueprints of data, operations, functionality, ...
- Means of **encapsulation**

    - hiding details of implementation
- Means of **combination**
    - using basic data structures to create compound data
    - combine basic functions to build bigger functionality

2.6.1 Means of Abstraction

Abstraction in general means to isolate ideas or concepts from actual physical reality. In other words it is a process of generalizing physical phenomenon without worrying about the details around it. Below are examples to understand application of abstraction in programming.

Programming language is itself built on this idea where abstract specification is isolated from actual implementation. For example C++ language specification is provided and maintained by a group of people. The actual implementation of the compiler can be done by anyone using those specifications. Therefore there are many C++ compilers available like GCC, LLVM etc.

Data types have evolved by isolating the specification and implementation. Interfaces, or abstract data types (ADT), define general desired properties a certain data type should possess. Data structure with algorithms provide a concrete implementation for an interface. For example, dictionary is an ADT while Python `dict` is a data structure with algorithms for different operations. The performance of a dictionary in CPython might differ from a dictionary in C++ or even JPython (Implementation of Python VM using Java).

Functional programming uses this idea in many different ways. Example 1, map, filter and reduce are general concepts related to a collection of items, the actual output depends on the function supplied while using the concept. Example 2, factory functions provide a blueprint of functionality, the arguments supplied decide the actual function created.

Object oriented programming (OOP) is built around this idea where class is an abstract blueprint of an object and instance is the actual object which depends on the data it holds.

2.6.2 Means of Encapsulation

Encapsulation in general means hiding the details of implementation.

Programming language: hides the details of mapping problem, which is the actual implementation of the engine, and lets the programmer create new solutions using natural language, rules and syntax without worrying about details of implementation of compiler, hardware management etc.

Data types: Once a data type is implemented you can create multiple instances without worrying about the details of implementation. For e.g. most languages provide way of defining numbers and strings by directly entering them without worrying about implementation details of how and where to store the data.

Functional programming: Once a function is created you can use them as and when required without worrying about the implementation details during usage.

Object oriented programming (OOP): Classes provide a way to define blue prints of objects with certain attributes. After that multiple instances of the data type can be created and operated upon without worrying about the implementation details during usage.

2.6.3 Means of Combination

Combining in general means joining. In programming it means the same, ways to combine simple parts to make a more complex part and make it simple using encapsulation.

Programming language is built around the same concept, smaller pieces are the elements of specification like lexicons and syntax, then semantics and architecture defines ways to combine them to make more complex programs.

Data types: Data structures like tuple, list, dictionaries etc. provide means of combining different data types to create arbitrarily larger and more complex data.

Functional programming: Using the rules of scopes and namespaces functions can be joined and nested in many interesting ways to create more complex functions.

Object oriented programming (OOP): Class itself allows combining data and operations to create arbitrarily large and complex data types as needed. Further more rules of inheritance provide ways of relating class definitions to create and structure even more complex data types.

3 Applications

3.1 Embedded Devices

Across industries more and more robotic devices are in use for automation of manual tasks. Most of this is done using assembly languages and C/C++ by embedding the final machine code in the robotic device itself.

3.2 Operating Systems

Major commercial operating systems like Microsoft Windows, Apple Mac, Linux based distributions are built with C/C++.

There a lot of Linux and GNU/Linux based operating systems, built by various communities of developers and commercial organizations, available for free.

An operating systems can be understood as 2 main parts, a kernel and applications. Kernel manages and provides means for applications to interact with the hardware. Applications provide the user interface, e.g. file manager, terminal, shell, media player etc.

Unix was a commercial operating system. Linus Torvalds developed a kernel based on Unix and provided it for free. Independently, Richard Stallman established GNU, which developed most of the applications needed for an operating system. There are several other foundations which create free software like KDE. Mixing these allowed creation of free operating systems commonly referred to as Linux or GNU/Linux operating systems. Linux in general refers to any operating system using Linux kernel as its base, applications can or cannot be from GNU's collection.

Mobile devices have their own operating systems like Google's Android and Apple's IOS. These are written in multiple languages but use C heavily.

OS provides a default set of operations to interact with the hardware. There are several smaller applications to interact with components of hardware

- input devices: keyboard, mouse, camera, microphone
- output devices: display, speakers
- network devices
- hard disk: file system

All the smaller applications are glued by the kernel and a graphical user interface is provided to interact with the hardware. This is done through desktop applications like file manager, text editor, photo viewer etc.

3.3 CLI = Terminal + Shell

Terminal is the first layer of interaction provided by the os. Terminal is a text based user interface (tui) application which hosts a shell program. Shell program is an interpreter running the commands written in shell language. Terminal and shell programs are mostly written in C/C++ too. Terminal and shell together provide the command line interface or CLI. CLI is covered in detail in Tools part of the book in Chapter 5.

3.4 Desktop applications

Examples of desktop applications are the file manager, media viewer etc. A desktop application is simply a program that can run on the os and provide means to perform certain tasks.

Some are provided by the os as default applications while others like Adobe's Acrobat, Photoshop etc. are provided by companies. In addition individual developers and group of developers can provide desktop applications through Windows app store or Apple store. Linux distributions have their own app stores.

Microsoft Office applications (word, excel, powerpoint, access database) are written in C/C++.

LibreOffice is the free version of Microsoft Office which is developed using multiple languages like C++, Java, XML, Bash, SQL, etc.

3.5 Databases

Databases are applications which specialize in storage, maintenance and operations on data. Most of them are built using C or C++.

Name	Type	Free	Language
MySQL	SQL	Yes	C, C++
PostgreSQL	SQL	Yes	Multiple
Redis	NoSQL, Key-value	Yes	C, ANSI C
SQLite	SQL, Embedded	Yes	C
Cassandra	NOSQL, Wide-column	Yes	Java
Oracle database	Multiple	No	C, C++, Assembly
SQL Server	SQL	Mixed	C, C++, C#
Microsoft Access	SQL	Mixed	C++
MongoDB	NoSQL, Document	Mixed	Multiple

3.6 Websites

Web technologies have become popular with the creation and development of browsers and web networks.

- Front end (GUI) is based on HTML, CSS and JavaScript
- Backend functionalities: like databases, analytics can be developed in language of choice

In practice there are many frameworks available for development which provide standard templates of pre built code for different functionalities.

Name	**Description**
HTML	Markup language to structure user interface structure and text
CSS	Responsible for the user interface styling
JavaScript	Responsible for the functionality behind web pages, e.g. click button actions, form submissions, ...
TypeScript	Stricter version of Javascript which is static typed

How do browsers run code?

Browsers are primarily running **JavaScript** using their own set of mapping engines.

Earlier the browser JavaScript engines were using interpreter approach but now have shifted to JIT approach like Java.

- Google Chrome: V8 engine
- Mozilla Firefox: SpiderMonkey
- Apple Safari: JavaScriptCore
- Microsoft Edge: Chakra

3.7 Mobile Applications

Mobile applications for Android and IOS are like desktop applications but designed to use limited resources of mobile. These are primarily oriented to use touch input.

Google play and Apple store host applications developed by companies, individuals and group of developers.

3.8 Data Analysis

With the continuous increase in volume of data collected, data analysis & visualization, artificial intelligence, machine learning are getting developed and used heavily for analysis and decision making.

Python and R are the languages used in this field as there is a lot of dependence on scientific computing, statistics and math.

Basic data analysis is helpful for everyone using technology across all domains.

Chapter 25 introduces data analysis in more detail.

3.9 Automation

3.9.1 Documentation

There are tools like Quarto available to automate documentation in multiple output formats. Quarto is based on Pandoc which is an application written in Haskell programming language for conversions between different markup languages. Quarto uses Pandoc's markdown syntax which makes writing documents as easy as writing text in a word processor or notepad. Content once written can be exported to different formats, html based website, pdf, word etc. This book has been written using Quarto.

3.9.2 System tasks

Shell and Python scripts are used to automate a large number of system tasks both for personal and official work.

Task can be as simple as creating a backup of your important files to more complex tasks to create templates of a project or task which you can create with automatic time and date stamps and other content pre-filled.

This book therefore introduces system task automation later as an application of programming.

4 Learning map

4.1 Introduction

Acquiring knowledge of any subject involves answering 3 fundamental questions.

- **Why**? *Most fundamental question that teaches the motivation*
- **What**? *Declarative knowledge that teaches the definition*
- **How**? *Imperative knowledge that teaches ways to apply the knowledge*

Similarly to learn computer programming you can use the below map.

- ***Why*** *do you learn computer programming?* → **Use cases**
- ***What*** *is computer programming?* → **Theory**: **Building blocks**, **Architecture**, **Design**
- ***How*** *do you do computer programming?* → **Practical experience**: **Tools**, **Application**

Learning theory equip you with knowledge about the subject, but to develop skill in the subject you need the tools and practical experience of applying the knowledge.

While covering theory, there is an attempt to provide exposure to some common scenarios of where and how to apply the concepts along with some common mistakes. Topic of design is related to this. Whereas by experience, you get better at the skill of deciding what concepts to apply under which situations.

For example, there are limited specifications of a language which you can learn quickly, but there are much larger possible ways to combine them. Operating systems, language compilers and interpreters have been written using the same specifications, without even using OOP, which is related to the skill of writing programs (applying theory in practice). For basic usage it is not required to have skills to write an operating system, but the example is to illustrate the difference between the "what" and "how" parts above.

The learning map, in context of depth of knowledge, will depend on the career path. There are 3 distinct possible career paths in context of programming.

- Generic: Basic
- STEM (minus CSE): Intermediate
- Computer science & Engineering (CSE): Advanced

  STEM => Science, Technology, Engineering & Math related fields

4.2 Basic

There are some common elements of programming which can be the starting point for all career paths.

- Introduction to programming with basic applications (this book)
- Basics of data analysis

4.2.1 Theory

As a beginner, you learn the basics of any one programming language. Python is chosen in this book as it is most popular generic programming language.

This book covers the basics and some intermediate topics in detail. Additionally, some intermediate and advanced topics are introduced, but not covered in detail, to improve the understanding of context while programming, but keeping the complexity low.

Color coding rules used

- covered in detail
- high level introduction
- not covered

Language specification

Building blocks

- Basic specifications
  - Lexicons
  - Variables
  - Comments
- Data types and operations
- Control flow blocks
- Functions
- OOP
- Special features

Architecture

- Environment = Namespace + Scope
- Scripts and packages
- External packages
- Language engine
- Debugging

Design

- Components
  - Design patterns
    - Map reduce
  - DSA
  - Regular expressions
- Frameworks
- Workflow

Tools

- Command line interface
- Version control system
- Language installation
  - Engine
  - External packages
  - Virtual environments
- Editor

4.2.2 Applications

- Automation
  - system operations
  - documentation
- Basics of data analysis (including visualization)
- Domain specific basic applications

4.3 Intermediate

Most of the science and engineering fields require heavy usage of data science which requires good practical programming skills along with math involved, specially probability and statistics.

In context of programming, the assumption is you would know the basics (content of this book) and move on to learn intermediate level theory. The topics will be same but study will be intermediate to advanced level. For example, details of OOP, design patterns like generators, context managers, etc. and how to implement them from scratch.

In addition it would be good to learn a compiled language like C++, with lower level details like memory management and parallel processing.

The key aim will be to be able to build basic domain specific packages and applications using frameworks if required.

- Theory
    - programming (intermediate level)
    - data science (intermediate level)
    - math: probability & statistics to model data
- Application
    - data analytics
    - building domain specific packages
    - building basic applications using frameworks

4.4 Advanced

In terms of theory there will be advanced coverage of topics depending on specialization.

- Math for computer science
- Language specification, implementation & design
- Data structures & Algorithms
- Design patterns
- Frameworks

In terms of application the path will depend on the choices.

- Embedded devices
- Operating systems
- Desktop applications
- Mobile applications
- Databases
- Websites
- Network applications
- Media
- ...

Part II

Tools

Overview

Background

To be able to use any of the tools a computer with an operating system is required. Computers come with 2 major commercial operating system options, Windows and Mac. Then there are a large number of free (in context of price) and open source Linux and GNU/Linux distributions. Mac is itself based on Linux so has similar workflow as other Linux distributions. Newer windows machines can run a Linux distribution using wsl or a virtual machine.

With the evolution of programming, there has been evolution of tools to write and execute programs. Learning these tools is not a one time task and takes time.

Introduction

Below is an overview of main tools used with their primary uses in context of programming.

- **Operating system**
  - **Command line interface** (**CLI**)
    - manage files
    - build and run files and projects
    - manage system installations and updates
  - **Version control system** (**VCS**): manage history of changes
  - **Python**:
    - **CPython installation**: interpreter, built-ins, standard library
    - **External packages**
  - **Editor**: compilation of tools needed in one place

Resources

Linux foundation offers a course, Introduction to Linux. This should provide a good background on how an operating system provides an interface to work on a computer. There are some parts of the course, example bash scripting, which should be referred to after getting some experience with programming.

MIT Course: The Missing Semester of Your CS Education covers the tools needed. You can refer to cli and Vim (editor) part ignoring what seems advanced or irrelevant.

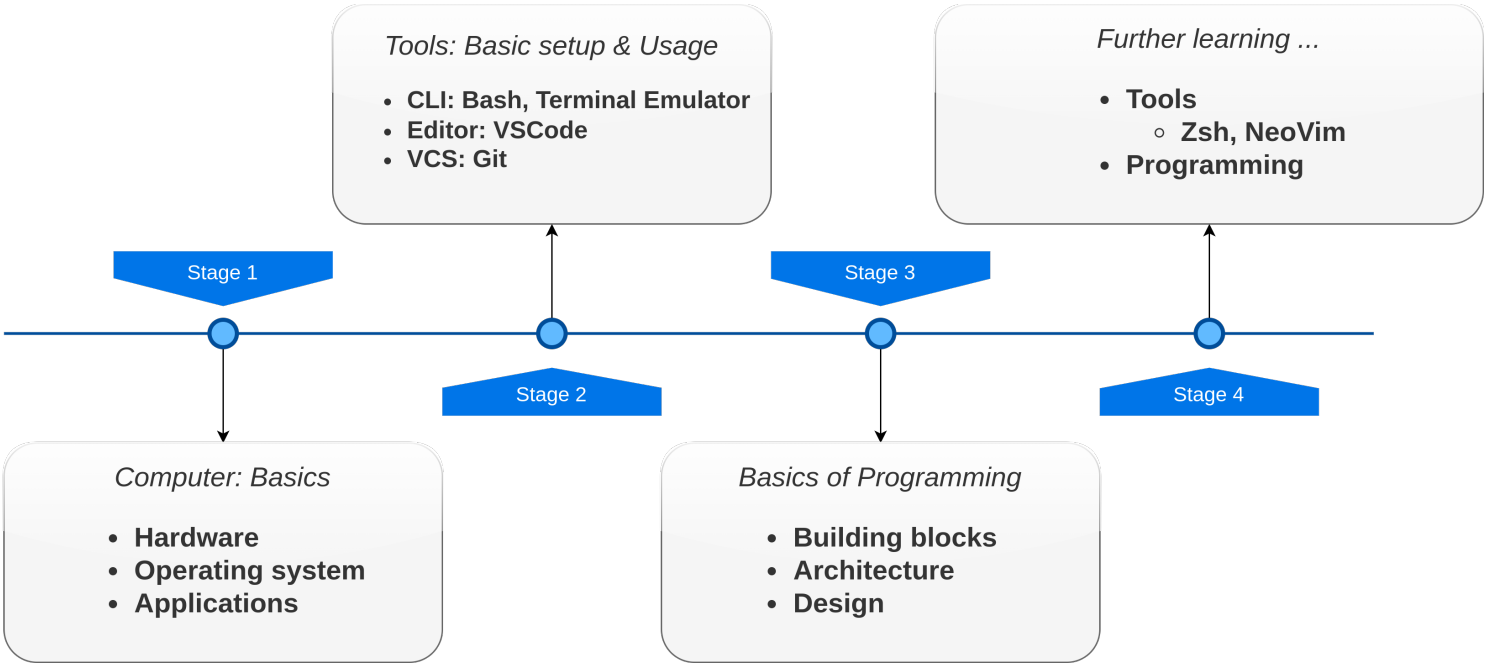
These can be considered as an optional pre-requisites.

Objectives

The chapters in this part provide background and introduction to these tools, to help get started and act as notes for reference in the beginning. There are a lot of choices for each of the tools, some recommendations are provided.

There is a lot of learning involved with the tools, specially cli and editor, as there is a lot of functionality and it keeps evolving. Therefore it makes sense to start gradually, focus on using the features when needed, research when there is spare time.

Usage, e.g. configuration of advanced editors like NeoVim, requires knowledge of programming. Therefore just an introduction to the tools is provided to act as quick start guides, to use them while learning to program. Once you learn to program, you can learn more about the tools using reference to resources provided and also do an independent research.



CLI

Basic usage of CLI is independent from programming. The built-in commands can be used as is for normal tasks while working on a computer. These are included.

Scripting is typically used to automate repetitive tasks. The scripting part can be learnt gradually after introduction to programming as the requirements arise. The Bash manual and the Linux course should be enough to get to a decent stage.

VCS

Git is introduced here to provide a high level introduction for familiarity and basic usage.

After learning basics of Git, you can start experimenting with it using CLI. For example, initialize a rough repository for the files you create while learning this content and create some commits, view logs. Once you move to editor you can use the integrated gui support for Git.

VCS is typically used for large coding projects and in practical set up has a lot more to it. Actual details will depend on the set up, which you can learn if needed.

Python

Installation and management of Python and external packages is a requirement for programming with Python and will be used throughout.

Jupyter notebooks

Jupyter notebooks are introduced to do experiments by executing small pieces of code in isolation with annotation, like while learning the building blocks.

Although experimentation with small pieces of code can be done using a regular Python script, Jupyter notebooks provide an extra option with other functionality like basic documentation, interactively view data and visualization, required for data analysis.

Editor

Editor is used after learning about scripts and packages in Architecture part.

It is the main tool to do programming, so it will be used throughout.

5 Command line interface

5.1 Overview

5.1.1 Introduction

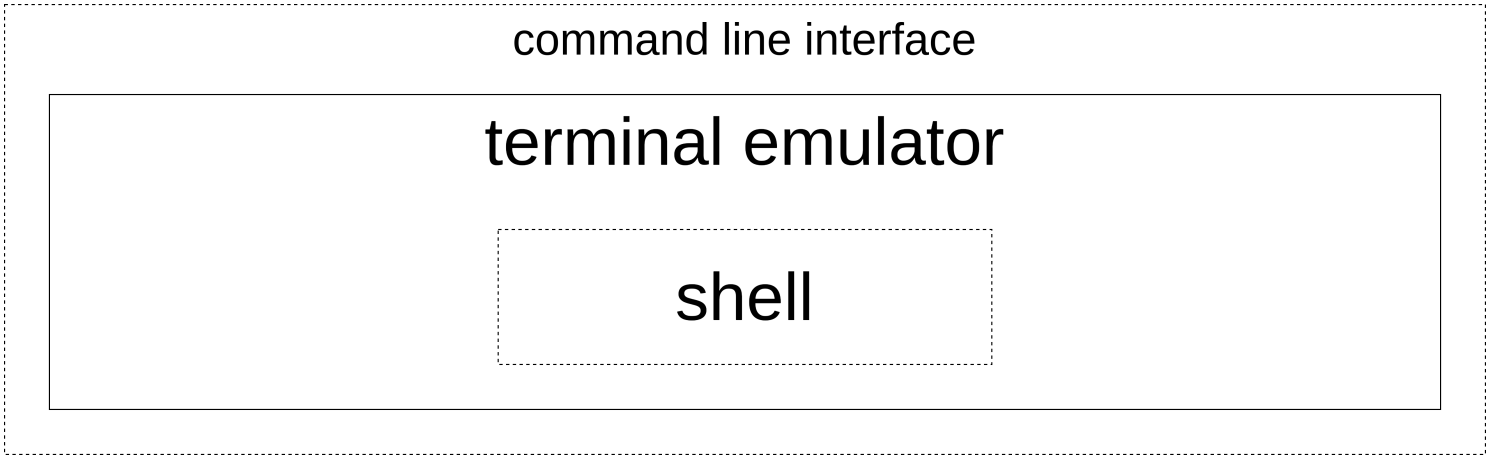
There is a lot of ambiguity in terminology. **Terminal** is an old term which refers to a physical interface to access the system. **Console** and **tty** is also often used in similar context, **tty** derives its name from hitorical tele typewriter machine.

Terminal emulator is a generic term for an application to provide text input/output environment.

Shell is a generic term for an interpreter that provides commands to interact with the system.

Terminal emulator and the **shell** program together provide **command line interface**, aka **cli**, to interact with the system (hardware, processes, applications, …) using text based interface.

Compared to graphical user interface (gui) based interaction with the system, cli is much faster and more powerful, once you learn it.



5.1.2 Terminal Emulator

Every operating system gives a default and often is called **terminal**.

- **Windows**
  - **Windows Console Host**: old
  - **Windows Terminal**: latest (as of 2023)
- **Linux**: default depends on the version
- **Mac**: has its own default terminal emulator referred to as terminal

5.1.3 Shell

There are several popular shells for Linux based systems.

- **Windows**: command prompt (cmd), powershell
- **Linux**: bash, zsh, fish, …
- **Mac**: earlier it was bash, now (as of 2023) it is zsh

5.1.3.1 Bash

`bash` is the default shell on most linux based operating systems and is available across all operating systems.

- **Linux**: default on most distributions
- **Mac**: earlier bash was default, latest is zsh, extension of bash
- **Windows**: easily accessible through wsl, cygwin, git/gitbash

5.1.3.2 What do shell programs do?

Shell provides commands and scripts to manage system.

- Hardware (monitor, keyboard, mouse, network, camera, …)
- Processes (logon, startup, users, security, …)
- Applications (browser, file explorer, …)
- File system (create/delete/rename/move files and folders)

Scripts contain commands and variables, combined using the language specifications, as any other programming language. Scripts behave as other commands and can be used from cli.

5.1.3.3 Use cases

Some examples of where cli is used.

5.1.3.3.1 Managing Workflow

- Files and directory operations
- Installation, update, maintenance of applications (software)
- Automation

5.1.3.3.2 System Administration

- Files and directory operations
- Installation, update, maintenance of applications (software)
- Automation
- Security: Group and user access
- Managing databases, servers, processes

5.1.3.4 Note

Most of the tasks that can be done from command line can be done with Python but understanding command line will help use Python better.

> ⚠ Caution
>
> Command line can be dangerous as one wrong command can destroy a large chunk of your work or even bring the system down.

> ❗ Important
>
> … and this happens more frequently with experienced users

5.2 Bash

5.2.1 Resources

- Homepage
- Bash manual

5.2.2 Setup

Installation depends on operating system.

- Linux distributions: `bash` is available and default on most distributions
- MacOS: `zsh` is the default but `bash` is available
- Windows: `git` comes with `gitbash` which should be enough

For windows, Git is covered in the next chapter. There are some other options for installing bash on windows, `msys2`, `cygwin`, `wsl`.

5.2.3 Configuration

There are certain aspects which are driven by the terminal emulator program. These are colors, fonts, terminal related keyboard shortcuts, windows and tabs management etc. These are configured directly through terminal program.

Bash configurations involve

- **Environment**: What are the default variables like `$PATH`, `$HOME` etc. available
- **Prompt**: What information does the prompt show
- **History**: How and where history is saved
- **Aliases**: Short codes for frequently used commands

Such settings are configured through `.bashrc`, `.bash_profile`, `.bash_aliases` files which are stored in `$HOME` path, which depends on operating system.

Actual configuration has more details to it and more configuration files. Additionally there are dependencies on the operating system. To learn more, it is advisable to do web search based on OS.

5.2.4 Commands

Bash is an interpreter that runs tasks using the concept of commands which are usually short codes (functions) to do certain task. They can optionally take flags, options and (positional) arguments.

5.2.4.1 General format

`<command name> [-flag[s]] [-option[s][=/ ]value[s]] [<argument[s]>]`

- **Command name**: are scripts containing code to perform tasks
  - e.g. `ls` prints files and folders contained in a folder
  - comes first
- **Flags (Switches)**: are short codes to alter the behavior of the program
  - multiple flags can be combined
  - it is safer to combine and provide all flags before any options and arguments
  - e.g. `ls -al` changes the way `ls` command lists the files and folders
    - note that `a` and `l` are 2 flags combined
    - `ls -a -l` works as well
- **Options**: are codes that take value
  - e.g. `head -n=15 <file path>` prints starting 15 lines of the given file
  - options can use `=` or space, e.g. `head -n 15 <file path>` works
- **Arguments**: inputs to the command interpreted based on position
  - e.g. `cp <src> <dst>`: first argument is provided to source to be copied from and second to destination where to copy

The flags and options additionally have long and short forms. e.g.

- `ls -l --all` $\iff$ `ls -al`

There are many variations to how flags, options and arguments can be positioned. The allowed placements vary by commands as well. The safe rule of thumb is

- *Beginning* (after command name): combine **flags** using short forms
- *Middle*: **options**
- *End*: **arguments**

5.2.4.2 Types

Commands can be grouped into below types

- **built-ins**: present by default
  - `man`, `less`, `ls`, `cp`, `mv`, `mkdir`, `rm`, `find`, `grep`, ...
- **command line utilities**
  - many applications give command line utility commands
  - e.g. create a Python virtual environment
    - `python3 -m venv <path to virtual environment>`
- **custom scripts**: create your own

Bash, like other shell programs, have its own *scripting language* which can be used to create scripts. Bash scripts end with `.sh` extension and can be run using
`bash <custom_script_name>.sh` command.

This is basic way to run scripts. There are advanced ways using file permissions which can be avoided in the beginning.

Below are some common commands based on category of tasks.

5.2.4.3 Summary

- help: `man`, `<cmd_name> --help`
- list and print: `ls`, `pwd`, `echo`, `| less`
- change directory: `cd`
- create file/directory: `touch`, `mkdir`
- remove file/directory: `rm`
- move/rename file/directory: `mv`

5.2.4.4 Getting help

- search web by use-case
- within terminal: `man <command name>`
    - `man man`
    - **to sroll and search** `man man | less`
- on windows/git-bash `man` will not work
    - `<command name> --help`
- check installation location
    - `which <app name>`

5.2.4.5 List and Print

- list files and folders: `ls`
- list **all** files and folders: `ls -a`
- list **all** files and folders in **long format**: `ls -al`
- print current directory: `pwd`
- print to console: `echo "content"`

5.2.4.6 File & Folder

5.2.4.6.1 Change directory

- change directory 1 level up: `cd ..`
    - change directory 2 level up: `cd ../..`
- `cd <relative dir path>`
- `cd <absolute dir path>`
- tab completion

5.2.4.6.2 Create

- create file: `touch <file-path>`
    - directory should exist
- create directory: `mkdir <dir-path>`
- create directory and intermediate directories if does not exist:
    - `mkdir -p <dir-path>`

5.2.4.6.3 Remove

- remove file: `rm <file path>`
- remove empty directory: `rm -r <dir path>`
- remove non-empty directory: `rm -rf <dir path>`

5.2.4.6.4 Copy

- copy file:

  `cp <src file path> <dst file path>`

- copy source dir with its content recursively:

  `cp -r <src dir path> <dst dir path>`

- copy only contents of source dir to destination recursively:

  `cp -r <src dir path>/. <dst dir path>`

5.2.4.6.5 Move (rename)

- move/rename: `mv <src> <dst>`
  - `src` = old file/directory name
  - `dst` = new file/directory name
  - if the `src` and `dst` path are
    - different => move
    - same => rename

5.2.5 Keyboard shortcuts

Main

- clear: `ctrl-l` (L)
- quit: `ctrl-d`
- copy: `ctrl-shift-c`
- paste: `ctrl-shift-v`

Delete

- delete till end (from cursor): `ctrl-k`
- delete till start (from cursor): `ctrl-u`

Movement

- move to start: `ctrl-a`
- move to end: `ctrl-e`

Processes

- cancel (kill) running jobs: `ctrl-c`
- put current process in background: `ctrl-z`

History

- view history: `history`
- run command from history: `!<#>`
- put command from history: `!<#>:p`
- search previous command: `ctrl-r`
    - *keep pressing* `ctrl-r` *to go through search list*

6 Version control system: Git

6.1 Introduction

Version control system, aka **VCS** or **source control**, is a software to maintain history of changes to a project, in context of folders, files and content of files.

In context of programming projects, a vcs is used for code and config files needed to reproduce a project. All the information is stored in a compressed directory.

Major use cases for a **VCS** are

- **Track**: history of changes with related information
- **Rollback**: in case needed
- **Collaborate**: changes to different parts can be made in parallel and merged

Some popular VCS are

- **Git**
- **Apache Subversion**
- **Mercurial**

This book covers **Git** as it is most popular. Getting familiar with vcs through cli should help with the following

- Basic usage of vcs without branching
- Introduce how basic things work without gui
- Understand and use gui tools better
- There are few complex tasks that can only be done from cli, when needed

There are gui tools in most editors so most of the interaction is through them.

To learn Git in-depth, refer to Pro Git. It is recommended to go through first 2 sections of the book to understand the basic fundamentals and operations.

A VCS is complemented by an online (cloud) service for hosting repositories. There are several different online providers with their own set of functionalities.

- **SourceForge**: supports multiple VCS
- **GitHub**: supports git only
- **GitLab**: supports git only
- **BitBucket**: supports multiple VCS

The key ideas behind working of Git are summarized in the first section of Pro Git, which is highly recommended.

6.2 Git CLI

Below is a summary of common Git cli commands based on use cases.

6.2.1 Config

- Check Git config

```
git config --list --show-origin
```

- Setup username and email for Git
    - mandatory after fresh installation
    - these are stamped for each commit to track who made changes

```
git config --global user.name "name"
```

```
git config --global user.email "email@example.com"
```

6.2.2 Initialize repository

```
git init
```

The above command is run from the folder to be version controlled.

When a repository is initialized

- `.git` folder is created which is the repository
- All files and sub-folders are tracked by default
    - Add `.gitignore` to ignore tracking certain files and folders

6.2.3 Check status and logs

```
git status --short
```

```
git log --oneline
```

6.2.4 Add changes to staging area

```
git add <file or dir path>
```

6.2.5 Commit

6.2.5.1 Using text editor

```
git commit
```

- Opens the default text editor to enter message and commit the staged changes
    - in this case first line will be the short message
    - remaining typed lines will form the detailed message

6.2.5.2 Quick commit

```
git commit -m "message text"
```

6.2.5.3 Stage all changes and commit

```
git commit -am "message text"
```

- Stage all changes
- Commit the staged changes with short message
- In the -a option no need to add files separately

6.2.6 Check differences

6.2.6.1 Since last commit

```
git diff
```

6.2.7 Undo changes from last commit

```
git restore <file name>
```

- Discard changes to file in working directory

7 Python

7.1 Installation

Python installation resources and instructions, based on operating system, are available at the official website.

- Download installer
- Instructions
    - Unix
    - Mac
    - Windows

Python installation includes

- CPython interpreter
- Pre built Python objects, scripts and packages with additional functionality
    - Built-ins
    - Standard library

7.1.1 Check Python installation from bash

- Windows

```
py --list-paths
```

- Linux/Mac: there are several methods one of which is given below

```
ls -ls /usr/bin/python*
```

7.2 Ways to run/interact with Python

- Repl
    - IPython repl: enhanced version of default Python repl
- Jupyter notebooks
- Run scripts from command line
- Run scripts from editor

Python repl and jupyter notebooks are covered in this part. The later two are included in the architecture part of the book.

7.3 Python repl

repl = read evaluate print loop

`repl` is a tool to interact with the interpreter.

To start a `repl` simply type `python3`/`py` in bash and press enter, actual command will depend on the operating system and python version.

`repl` is rarely used directly, instead editors use it under the hood to provide functionalities like running and debugging the code. Below are some reasons why this is the case.

- repl does not save work
- not designed write large programs as it does not have all the tools needed

It is useful in conjunction with editor, to test code short pieces of code or while debugging.

IPython repl is newer modern repl. To use ipython instead of default repl, ipython has to be installed. To use it from editor, additional configuration is needed.

7.4 `pip`

The Python Package Index (PyPI) is a repository of packages (collection of Python programs) written for use in Python programming language. `pip` is one of the listed packages.

The term package here has special meaning, more than just a regular or namespace package used in Python programming. Package here refers to collection of Python scripts and packages for distribution. In the beginning, details can be avoided, but when needed refer to Python docs: Installing packaging.

`pip` is the package installer for Python which manages the installation and maintenance of other external packages contributed by developers. It has its own independent website with all the details. `pip` website.

Typically pip is included in Python installation but you can check using bash with any of the below commands.

```
pip -V
```

```
pip -h
```

```
pip --help
```

7.4.1 Usage

To install or update refer to installation guide. Once `pip` is installed it is easy to install any external package from cli, `pip install <package name>`, e.g.

```
pip install jupyterlab
```

Although above works in most cases, it is advisable to use full command with Python version, as given below. This takes care of edge cases when there are multiple Python installations on a pc. Each Python installation can have its own pip or not. Using the below form of installation command ensures relevant pip is used and installation is done in the specified installation. The below command is explicit, use python3 installation's pip and install jupyterlab into python3 installation. If python3 does not have pip installed, it will give error asking to install pip first.

In general use python installation path. For system installation, `python3` for example links to the path directly. For virtual environments, explicit path is needed.

```
python3 -m pip install jupyterlab
```

When working with virtual environments, which are discussed later (Section 18.4), it is safer to use the below form of command. For Unix based systems (Linux/BSD/MAC)

```
<venv path>/bin/python3 -m pip install jupyterlab
```

7.4.2 Requirements file

One of the important feature of `pip` is to manage a project's dependencies, through a `requirements.txt` file. In this book, this is discussed in section on virtual environment in architecture. There is a section dedicated in the official documentation, link to `requirements` section.

8 Jupyter notebooks

8.1 Overview

8.1.1 Introduction

Jupyter is an independent project that provides

- **Jupyter Notebooks**: interactive code cells, markdown text, tables, plots, latex math formulas
- **Jupyter Lab**: web-based interactive development environment (editor)
- **Jupyter Hub**: multi-user version of notebooks

Jupyter notebook is new version of interactive python notebook that have extension `.ipynb`, which is still used.

8.1.2 Objectives

In this book intent is to provide notes and references to get started with jupyter notebooks using jupyter lab. Note that jupyter notebooks can be used independent from jupyter lab. There are other editors which support using jupyter notebooks, e.g. VSCode, RStudio etc.

8.1.3 Resources

Jupyter project's website has extensive documentations for further learning.

8.2 Installation

Jupyter notebook can be installed independently but it is recommended to install jupyter lab which contains notebook and related dependencies. Link to help page.

```
<python installation path> -m pip install jupyterlab
```

Refer to pip usage section (Section 7.4.1).

8.3 Features

- Run code interactively
- Mix rich text, code and results using markdown and code cells
- Export results in different formats like html, pdf, word etc.

8.4 Use cases

- **Interactive programming** for quick trials or learning with annotation
- **Reproducible Research**: quick and easy sharing of ideas/analysis with code and results

8.5 Keyboard Shortcuts

- add cell
  - below: `b`
  - above: `a`
- change cell type
  - markdown: `m`
  - code: `y`
- cut cell: `x`
- copy cell: `c`
- paste cell below: `p`
- enter/exit cell edit mode: `enter/esc`
- run cell: `Ctrl + Enter`
- run cell and move to cell below: `Shift + Enter`

8.6 Markdown

Markdown is a markup language which provides simple syntax for text to be converted to different formats. It is used extensively by many tools which try to automate documentation, e.g. quarto, pandoc, github, jupyter notebooks and many more. Each have their own subtle variations, but are very similar as the root is markdown.

Markdown syntax is easy and quick to learn.

8.6.1 Syntax cheat sheet

Description	Syntax

Heading	`# H1`
	`## H2`
	`### H3`

Bold	`**bold text**`
Italic	`*italicized text*`
Blockquote	`> blockquote`

Ordered List	`1. First item`
	`2. Second item`
	`3. Third item`

Description	Syntax

Unordered List	`- First item`
	`- Second item`
	`- Third item`

Task List	`- [x] task 1 (checked)`
	`- [ ] task 2`
	`- [ ] task 3`

Code	`` `code` ``
Horizontal Rule	`---`
Link	`markdown [title](www.example.com)`
Image	`![alt text](image.jpg)`
Strikethrough	`~~This text is striked.~~`
Subscript	`H~2~O`
Superscript	`X^2^`
Math: inline	`$e = mc^2$`
Math: display	`$$e = mc^2$$`

9 Editor

9.1 Overview

9.1.1 Introduction

Editor combines most of the tools and functionality needed for programming in one interface. Therefore, it is the tool which is used most while programming.

9.1.2 Background

As the programming languages evolved, so did the tools, specially editors. Historically there were two primary types of editors, text based lightweight editors (e.g. Emacs, Vim, Nano, Atom) and full fledged integrated development environments or IDE's (e.g. Visual Studio, IntelliJ Idea).

IDE's were loaded with lot of features for developers, but used a lot of CPU and RAM. IDE's conventionally were designed for use with specific languages and provided a lot of features for development in that language. IDE's were focussed towards ease of use to attract developers towards specific platforms and frameworks.

The text based editors were lightweight with less features but generic. Text based editors have evolved to be able to be configured to as IDE while being generic, they can be configured for use with almost any programming language. e.g. VSCode (based on Atom), Emacs, NeoVim (Based on Vim). Emacs was an exception, it was light weight by default, but provided extensibility to configure it as much more powerful than a conventional IDE, but that required good programming skills in Lisp (programming language).

Currently (as of 2023) the boundary is blurred because of technologies like language server protocol (lsp). For example, VSCode, a text editor based on Atom, is light weight but can work like an IDE using extensions (plugins) without much knowledge of programming. Emacs and NeoVim are also similar but not as easy to configure, but once you know programming then they can be configured to be much more powerful.

Plugins or extensions are pieces of code to add some functionality to a computer application.

9.1.3 Features

Below are some features that an editor can have. Some are independent of programming language and some depend on the programming language.

Even features that are specific to a programming language, need initial configuration specific to the programming language. Post that much of the interaction in editor, like keyboard shortcuts and commands, remain the same.

Therefore, it makes sense to choose an editor which has support for most languages.

Common features

- User interface
  - themes
  - windows
  - frames
- Editing
  - tabs/spaces
  - folding
  - multiple cursors
  - keybindings
- Outline
  - project files
  - vcs
  - extensions
- Terminal
- Projects & Workspaces
- Version control

Language specific features

- Syntax highlighting
- Auto
  - completion
  - formatting
  - linting
  - refactoring
  - snippets
  - help/documentation
- Object browser
- Debugger
- Compile/build/run
- Testing

9.1.4 Options

There are a lot of editors available. There are some editors specific to a language and some are generic. Below are examples of some popular editors.

It makes sense to use a generic editor as it reduces effort while using multiple languages. This is because you can leverage the learning involved for language independent common features. Some popular open source options are listed below with features.

9.1.4.1 Python Specific

- PyCharm
- Spyder
- RStudio

9.1.4.2 Generic

9.1.4.2.1 Vim

- Homepage
- Plain text editor
- Steep learning curve
- Modal key bindings are powerful
- Useful for quick changes directly from terminal
- Extensions are available for additional functionality
- NeoVim is based on Vim
  - advanced features
  - offers extensibility with Lua programming language
  - neorg, inspired by Emacs org mode, under development

- Faster than most other editors
- Works from cli

9.1.4.2.2 Emacs

- Homepage
- Very powerful and hackable using `emacs lisp` language
- Steep learning curve
- Spacemacs (a modified distribution of Emacs) is a good blend of vim keybindings and emacs
- Extensions are available for additional functionality

9.1.4.2.3 VSCode

- Works out the box
- Easy to configure
- Easy to use
- Supports Jupyter notebooks
- Highly configurable
- Light weight
- Supports most of the languages
- Supports most of functionality required
- Extensions available for additional functionality

9.2 Recommendation

VSCode is recommended to begin programming with, specially because of ease of use in getting started.

Keybindings in VSCode can be configured. It is recommended to learn and use Vim key bindings. Learning Vim keybindings will help using Vim when needed. VSpaceCode extension is useful as it includes Vim keybindings and additionally allows integration of all keybindings in more mnemonic style.

9.3 Resources

VSCode is well documented at VSCode docs. Below are recommended sections to get started.

- Setup
    - Keybindings
- Introductory video series
- Python setup
- Profiles

As you progress you can learn and use more features using the documentation.

Part III

Building Blocks

Overview

Background

Learning a programming language starts with learning the specifications of elemenatary pieces or building blocks.

- **Language specifications**
  - **Building blocks**: *specifications for elements and blocks of code*
  - **Architecture**: *specifications for writing programs*
    - means to combine elements and blocks to build programs
    - management of execution of blocks behind the scenes
- **Design**: *knowledge of how to write good programs*
- **Tools**: *needed to write, test, debug and run programs*

Introduction

Building blocks comprise of following elements which provide the basic elements to write code.

- *Basic specifications*
  - **Lexicons**: allowed symbols and characters
  - **Variables**: mechanism of storing and re-using information
  - **Comments**: annotating code
- **Data types and operations**: storing and operating on numbers, strings, collections
- *Control flow blocks*
  - **Conditional exectution blocks**: `if-elif-else`, `match-case`
  - **Loops (Iteration)**: `for`, `while`
- **Functions**: re-use code with configurable inputs
- **Object Oriented Programming** (**OOP**): re-use predefined type of data and associated operations
- *Special features*: Python specific extra features
  - **Conditional expressions**: short-circuit, context aware evaluation
  - **Comprehensions**: special short syntax to combine `transformation iteration filter`

Objectives

Building blocks are introduced with related specifications (lexicons, syntax, semantics). Examples are provided to demonstrate implications of the specifications to demonstrate experimentation. Specifications are limited and easy to understand, implications are a lot as they grow with application and combination of the specifications.

Building blocks can be combined in multiple directions to solve problems. Multiple direction means a conditional block can contain a function, and a function can contain a conditional block. Therefore there are multiple ways to combine different elements, but not all are same in terms of readability elegance and efficiency.

The objectives for this part are to

- Learn specifications (lexicons, syntax, semantics) for the building blocks
    - Understand semantic specifications and their implications
    - do experiments in isolation
- Introduce basic examples of combining different building blocks
- Tools: Jupyter notebook in editor (VSCode or Jupyter lab)

10 Basics

10.1 Objects

In Python everything is an **object** stored on RAM (random access memory), and is looked up using object reference that is a memory address on RAM.

Objects are fundamental to **object oriented programming**, which will be covered in the OOP section.

Object stores information in attributes which are of 2 fundamental forms.

- data attributes
- operations

To understand the idea think of an object as ***object = data + operation*** where,

- **data = noun** and **operation = verb**
- **data = state** and **operation = behavior**

For example, consider an integer "text" stored as an object in Python. It has data type attributes like value which is the "text" and numerous operations like capitalize which returns another string "Text". The actual implementation has more details, this is simplification of the general concept.

Data attributes are responsible for storing the information that defines the state of an object.

Every object has special data attribute which defines what type of object it is. In Python it is `__class__`.

Operations are other type of attribute, which provide functionality. They are functions with some additional features and are often referred to as methods.

10.1.1 dot operator

Dot operator `.`gives access to object's attributes, data and operations.

```
num_1 = 1.5
```

```
num_1.as_integer_ratio()
```

```
>>>  (3, 2)
```

10.2 Variables

10.2.1 Introduction to variables

Variables are named references that provide a handle to the object. They are stored separately on RAM in a separate table called namespace.

Namespaces are mapping between variable names and reference to objects which can be used to lookup memory address of the object.

Note that memory address is not constant, the language interpreter stores created objects at available memory address and deletes them when not needed. On many occasions the memory address will change, e.g. re running a program. That is where variables and namespaces come in.

Variables provide handle to objects for

- reuse and passing around objects
- performing operations

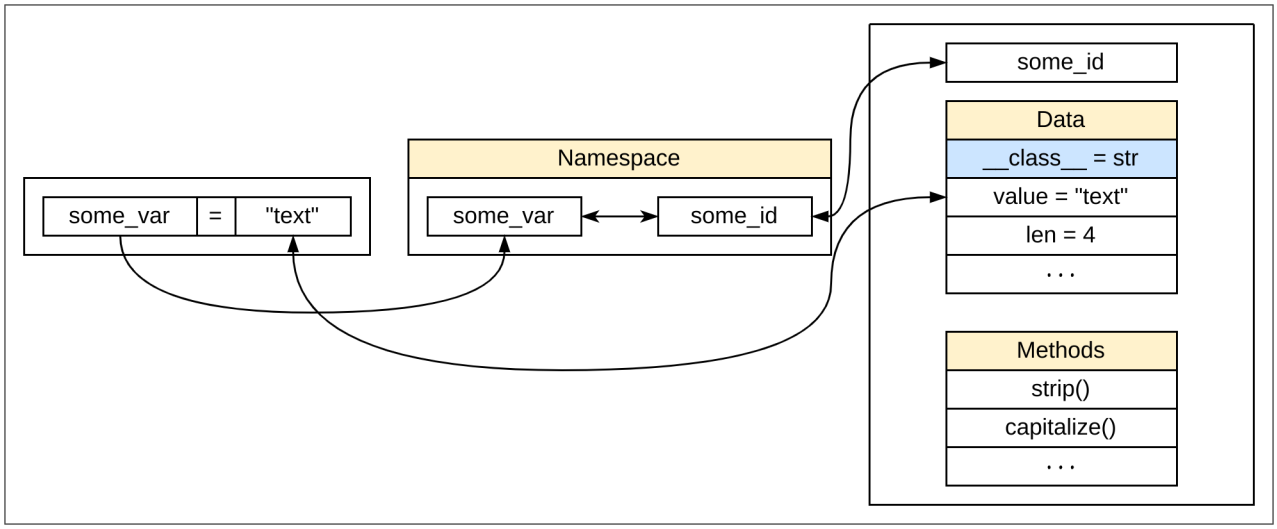
In Python, variable are defined using the `=` operator.

When a variable is defined for example

```
some_var = "some text"
```

- a string type object `"some text"` is created in RAM which has `some_id`
- `some_var` variable name is bound to `"some text"` object created
    - this is referred to as **name binding**
- variable name `some_var` and memory address of object `"some text"` are stored in a namespace

Below figure illustrates the relationship between a variable name and object.



Once the language interpreter sees what type of object it is, it knows what is the structure of data and operations stored in the object. Hence, the associated variable name has access to the attributes.

10.2.2 Deciding variable names

10.2.2.1 Must-follow

- **case sensitive**
- **start** with `_` or `letter` (a-z A-Z)
- **followed by** any number of `_`, letters or digits
- **cannot** be one of **reserved words**

Below code can be used to check keyword list in Python.

```
import keyword
print(keyword.kwlist)
```

Table 10.1: Python keywords (35)

`False`	`None`	`True`	`and`	`as`	`assert`	`async`
`await`	`break`	`class`	`continue`	`def`	`del`	`elif`
`else`	`except`	`finally`	`for`	`from`	`global`	`if`
`import`	`in`	`is`	`lambda`	`nonlocal`	`not`	`or`
`pass`	`raise`	`return`	`try`	`while`	`with`	`yield`

10.2.2.2 Should-follow

- `_my_var`: names starting with single underscore
  - used for `private` or `internal` objects
  - not exported by `from module import *`
- **avoid using names with dunder**
  - `__my_var`: names starting with double underscore
    - used to *mangle* class attributes in class chains
  - `__my_var__`: names starting and ending in double underscore
    - system defined names used in class internal attributes

10.2.2.3 PEP-8 Conventions

PEP refers to Python enhancement proposals which have detailed documentation about the rationale and description of changes to Python language specifications.

PEP-8 is related to Python code styles and is highly recommended read. Below are the conventions suggested for variable names. These will be used all through the course with few exceptions.

Following these conventions makes the code readability and understanding the code easy for both the writer and users.

Object Type	Convention	Example
Packages	• short • all-lowercase names • preferably no underscores	`utilities`
Modules	• short • all-lowercase names • can have underscores	`db_utils.py` or `dbutils.py`
Classes	upper CamelCase	`BankAccount`
Class instances	lower snake_case	`bank_account`
Functions	lower snake_case	`my_func`
Variables	lower snake_case	`my_var`
Constants	upper SNAKE_CASE	`MY_CONST`
Dummy variables	underscore	`_`

10.2.3 Python features

10.2.3.1 Dynamic type

In Python, variable type do not need to be declared and same variable can be bound to different type of objects.

This is leads to terms like Python is a dynamically typed language.

Opposite to **dynamic type** is **strict type**, e.g. `C`, where once a variable is declared to point to string objects it can only store string objects.

```
some_var = 10
some_var = "some text string"
```

```
print(f'{some_var=}')
```

```
>>>  some_var='some text string'
```

Variable type can be declared for assistance and code readability, but does not enforce the type, which means it can be bound to other object type.

It is not recommended to declare type and then use different type of object.

PEP-487 has more detailed discussion around this.

```
some_var:str = "some text string"
some_var:int = 10
```

```
print(f'{some_var=}')
```

```
>>>  some_var=10
```

10.2.3.2 Multiple assignment

To exchange values of a set of variables you do not need temporary assignments. Below examples illustrates this.

```
a = 10; b = "some text"; c = ["this", "is", "a", "list"]
```

```
print(f'{a=}, {b=}, {c=}')
```

```
>>>  a=10, b='some text', c=['this', 'is', 'a', 'list']
```

```
a, b, c = c, a, b
```

```
print(f'{a=}, {b=}, {c=}')
```

```
>>>  a=['this', 'is', 'a', 'list'], b=10, c='some text'
```

10.3 Commonly used syntax

10.3.1 Comments (#)

Comments are text within code which is not evaluated and is used for documentation and code readability.

There are 2 basic rules related to comments.

- python ignores and does not parse/evaluate the line content after `#`
- cannot use `#` just after assignment operator `=`

10.3.1.1 Examples

10.3.1.1.1 Example 1

Below is simple usage to document the code.

```
# Enter the value of inputs below
x = 10
y = 20

# Calculate sum
z = x + y
```

10.3.1.1.2 Example 2

In the example below, variable `num_in_comment` gives an error while trying to print on line 2 because line 1 is commented and not evaluated, therefore there is no variable `num_in_comment` in namespace.

```
# num_in_comment = 10
print(num_in_comment)
```

```
>>>  Error: NameError: name 'num_in_comment' is not defined
```

10.3.1.1.3 Example 3

Below example illustrates incorrect usage of `#` after `=`.

```
some_num = # value on next line
10
print(some_num)
```

```
>>>  invalid syntax (<string>, line 1)
```

10.3.2 Newline

- New lines can be introduced in multiple ways
  - explicit method
    - break lines using \
    - join lines using ;
  - implicit method: Expressions within (), [], {}
    - can be broken into multiple lines
    - can contain comments
    - can have trailing commas
- New line syntax can be used to enhance code readability

10.3.2.1 Explicit method example

```
some_var_1 = 10
some_var_2 = 20
some_var_3 = 30
some_var_4 = 40
if some_var_1 > 5 and some_var_2 > 10 and some_var_3 > 20 and some_var_4 > 30:
    print('yes')
```

When the code becomes too long to fit or is too short it is useful to use explicit method to join or break lines to improve code organization and readability.

```
some_var_1 = 10; some_var_2 = 20
some_var_3 = 30; some_var_4 = 40

if some_var_1 > 5 and some_var_2 > 10 \
    and some_var_3 > 20 and some_var_4 > 30:
    print('yes')
```

10.3.2.2 Implicit method example

Expressions in (), [] or {} can be split into multiple lines without needing explicit use of backslash (\). Optionally they can contain comments.

Note that trailing commas are allowed, illustrated in second example.

```
a = (
    "item 1",
    "item 2",
    "item 3"
)
```

```
a = [
    1, # first item
    2, # second item
    3, # third item
    ]
```

10.3.3 Blocks (indentation)

Python uses indentation to isolate distinct blocks, like control flow blocks (`if`, `while`), functions (`def`), classes (`class`). This improves code readability.

Indentation can be made using special `tab` characters or `spaces`. It is recommended to use 4 spaces and is generally a good choice. Below is an example, to illustrate how indentation improves code readability.

```
import math
def calc_circle_area(r=1, pi=math.pi):
    if r < 0:
        print("radius should be >= 0")
    else:
        return pi*(r**2)
```

Editors allow you to choose the method and amount of indentation to use. So in VSCode it is recommended to set it to 4 spaces. "Editor: Tab Size" is the relevant setting which is documented at link.

10.4 Functions

This section provides introduction to some basic functions which will be used to explain some underlying concepts in topics that follow.

Focus on

- understanding at a high level what the function does
- getting used to executing small pieces of code (in jupyter notebooks)
- using variable assignment

Ignore the details of `f''` string formats for now, they are there for formatting output. They will be covered later in data types section under string operations.

10.4.1 type

- `type(_obj)`: returns object type

```
var_1 = 10
var_2 = 10.0
var_3 = "string"
```

```
type(var_1), type(var_2), type(var_3), type(10.2)
```

```
>>>  (<class 'int'>, <class 'float'>, <class 'str'>, <class 'float'>)
```

10.4.2 id

- `id(_obj)`: returns object memory reference id
    - `hex(_integer)` converts to hexadecimal for better readability

```
_string_1 = "some text"
```

```
id(_string_1), hex(id(_string_1))
```

```
>>>  (134091648949168, '0x79f4a51a5bb0')
```

10.4.3 is

`a is b`: check if `a` and `b` refer to same object

This means checking if the memory address of the given variables or objects is same.

In the example below a list object `[10]` is created and both `a` and `b` are bound to the same object. Therefore `a is b` returns true.

```
a = b = [10]
```

```
print(f'{a is b = }\n{hex(id(a))=}\n{hex(id(b))=}')
```

```
>>>  a is b = True
>>>  hex(id(a))='0x79f4afdec200'
>>>  hex(id(b))='0x79f4afdec200'
```

In the example below a list object `[10]` is created and variable `a` is bound to that object. Then variable `b` is bound to the object that `a` is bound to.

```
a = [10]
b = a
```

```
print(f'{a is b = }\n{hex(id(a))=}\n{hex(id(b))=}')
```

```
>>>  a is b = True
>>>  hex(id(a))='0x79f4ad596fc0'
>>>  hex(id(b))='0x79f4ad596fc0'
```

Since `[10]` is a list type object, even though `c` and `d` assignments look the same, they point to different objects.

```
c = [10]
d = [10]
```

```
print(f'{c is d = }\n{hex(id(c))=}\n{hex(id(d))=}')
```

```
>>>  c is d = False
>>>  hex(id(c))='0x79f4afdec200'
>>>  hex(id(d))='0x79f4a51aa4c0'
```

Above examples illustrate subtle points about variable and object bindings which lead to a lot of implications while using mutable and immutable objects. This is covered in more detail in next section on data types and rest of the book.

10.5 Modules and `import`

Modules in Python refer to different things, based on context.

- a file with `.py` extension containing Python code
- an object once a `.py` file or package is imported using `import` statement

Packages are special type of modules, a folder containing Python files.

One of the important use-case of these specifications is to use external code. Python standard library has a lot of built in functionality for various use cases. The modules and packages contained in the standard library can be imported and used as required.

Import system in Python provides ways of managing objects in code and files. This is introduced here for basic usage in examples. All this is covered in detail in Architecture part.

In the example below,

- `math` module is imported using `import` statement
- objects: definitions are accessed using dot operator

```
import math

print(math.pi)
```

```
>>>  3.141592653589793
```

```
print(math.ceil(10.2))
```

```
>>>  11
```

10.6 Executing Python files

There are 2 main ways to execute a Python (`.py`) file using command line.

- `<path to python executable> <path to file>`
  - can take relative path
  - for a regular package
    - only `__main__.py` is executed if package name is used
    - `__main__.py` or `__init__.py` can be given explicitly
- `<path to python executable> -m <file name without extension>`
  - cannot take relative paths

- command has to be executed from the directory containing the file
- if the file is in a directory below the shell directory then dot notation can be used
  - e.g. `python3 -m sub_dir.file_1`
- in case of a regular package, `__init.py` and `__main__.py` both are executed

Since using the first approach, a Python file can be executed from any directory it is the preferred approach.

10.6.1 Executing one script from another

The recommended approach for executing one script from another is to keep the files in same folder and then use `import`.

For example, `file_1.py` has to execute `file_2.py`. Place `file_2.py` in same folder as `file_1` then use `import file_2` in `file_1.py`. Whenever `file_1.py` is executed it will execute `file_2.py`.

11 Data types

11.1 Introduction

Data type refers to object types that store information and provide some operations on that information.

Compare a function and a number, both are objects but a function stores code and numbers stores numeric data primarily. That is why, function, class etc. are not referred to as data types like `int` or `float` which are numeric data types.

Data types are the most critical part of any language. They are used to store, access and operate upon information within code.

Numbers and text are the most fundamental data types.

Some languages like C, distinguish between characters and strings, where strings are treated as sequence of characters. Python has just strings for text, which are a sequence of characters, and can be a sequence of single character.

Then there are collections which provide ways to combine objects to create more complex data. Data types like `list`, `dictionary` etc. are provided in higher level languages. Every high level language has its own implementations and syntax with differences, but underlying design principles are the same.

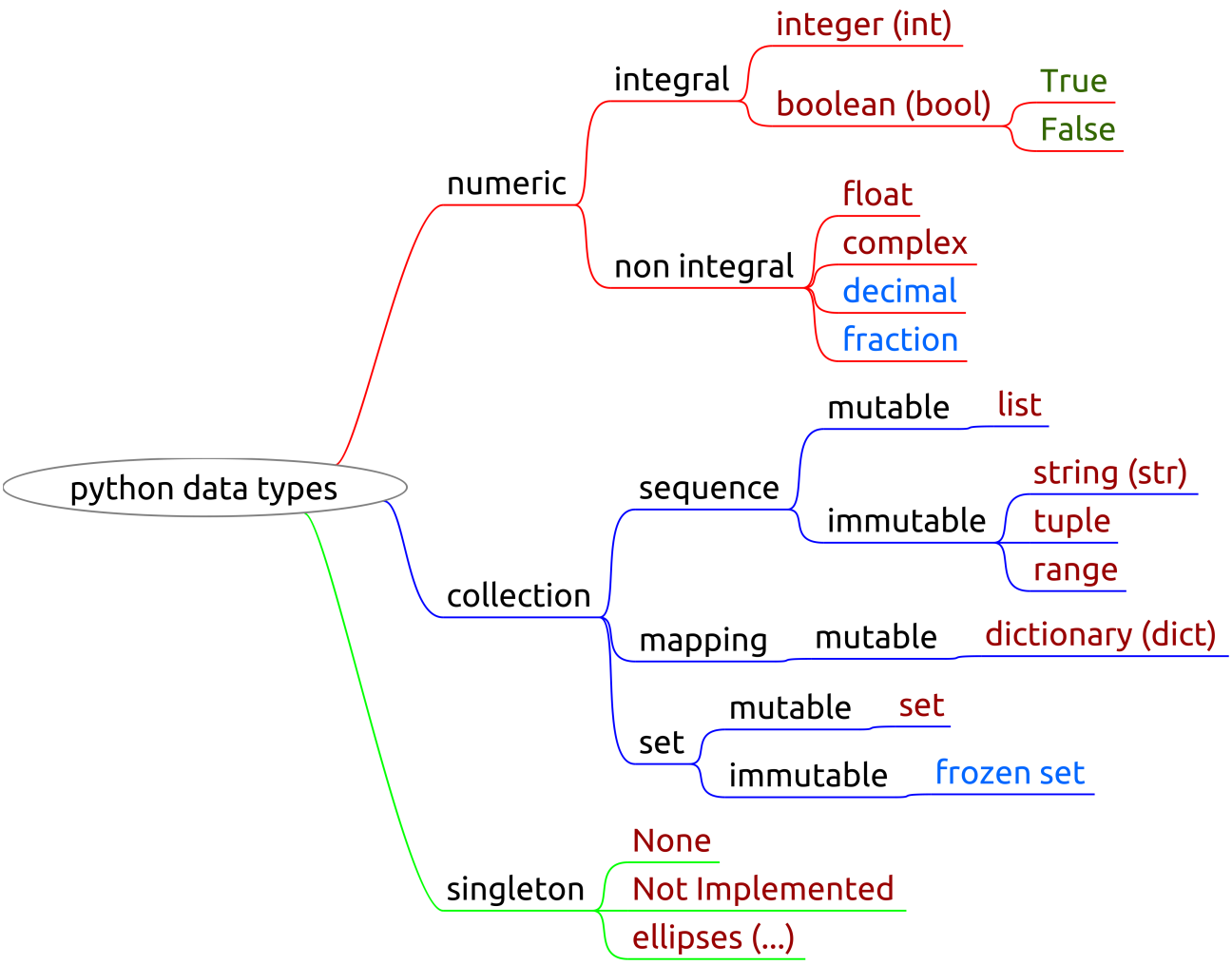
Data Structures and Algorithms part of the book gives a high level background of how the data designs have evolved.

11.1.1 Overview

Below tree provides an overview of data types implemented in Python with categories.

- The nodes with red text are the data types which are `builtin` in Python and are covered in the book
- The nodes with blue text are data types available by loading from standard library

  Standard library is discussed in architecture part of the book: Section 18.2



Collection is generally used for collection of objects, and it can be compounded, collection of collection of objects.

Sequence type is a collection of objects with preserved order. In base Python, strings, tuples and lists are sequence type.

Sequences can be **mutable** or **immutable**.

Iterable is any collection of objects from which objects can be retrieved one at a time and hence can be looped through. `str`, `tuple`, `list`, `dict`, `set` are all iterables.

Sequence is a subset of **collections**. All sequences are iterables.

Methods available in any sequence type can be categorized as below.

- common methods for sequence types
    - sequence level operations (e.g. length of a sequence)
    - element level operations (e.g. indexing and slicing)
    - methods available for iterables (can be used in loops)
- if mutable then methods for mutable sequence types
- methods specific to a sequence type

Iterable methods are related to use in looping and are discussed in respective sections.

Based on this categorization, methods are discussed for respective sequence types, string, list and tuple.

All this information is end product of developments in the field of data structures. The ***DSA*** part of the book has more information on this, post reading which context of why Python data types behave the way they do will be more clear.

11.1.2 Objectives

Focus on understanding the underlying concepts and where to look for information in a structured way.

- **Creation and syntax**
- **Operations**: understand how operations are **structured** across data types
- **Indexing** and **slicing** for sequence types
- **Implications**: understand the usage and implications using example use cases given
    - **Mutable** vs **Immutable** data types
- **Choosing data types**: understand **when to use which data type**

11.2 None

`None` signifies absence of value. A variable might be defined but not bound to any object. `None` is a placeholder to signify this state.

It is specially useful in conditionals, to avoid error when checking if a value has been assigned to a variable. This is covered in chapter on conditionals (Section 11.10.1.2.3, Section 15.1.5).

11.2.1 Example

```
some_var = None
print(f'{some_var=}, {type(some_var)=}')
```

```
>>>  some_var=None, type(some_var)=<class 'NoneType'>
```

11.3 Numeric types

11.3.1 Summary

Boolean	Integers	Rationals	Real	Complex
`bool`	`int`	`fractions.Fraction`	`float` `decimal.Decimal`	`complex`

- numeric data types are **immutable**
- for basic calculations `int` and `float` are sufficient
    - others are listed for completeness
- **boolean** and **comparison operations** are discussed separately

Immutable implies that once an object is created, its value cannot be modified.

For variable assignment this implies that if a variable is storing some number and is assigned another number, a new object is created in background. This does not have any significant impact in case of numeric data types.

Mutability is discussed in more detail at the end of this chapter.

11.3.2 Specifications

Numbers, integers or floats, can be typed as done in regular math. There are some special syntax available for code readability.

Underscores can be used for better code readability. During execution they are treated normally.

```
one_million_int = 1_000_000
one_million_float = 1_000_000.00
print(f'{one_million_int = }')
```

```
>>>  one_million_int = 1000000
```

```
print(f'{one_million_float = }')
```

```
>>>  one_million_float = 1000000.0
```

Scientific notation can be used with floats. `e` has to be preceded by a number.

```
x = [1e-2, 3.314e+5]
print(x)
```

```
>>>  [0.01, 331400.0]
```

11.3.3 Operations

Regular math operations can done using the symbols provided as listed below. Other functions commonly used are

- `round(x[, n])` is a builtin function provided
- Standard library has more options for numeric operations
  - `math` module
    - e.g. `math.floor(x)`, `math.ceil(x)`, `math.trunc(x)` etc.
  - `random` module for pseudo random number generations

addition	substraction	multiplication	division	exponents	floored division	modulo
`+`	`-`	`*`	`/`	`**`	`//`	`%`

- using `division` always returns `float`
- operations with `int` and `float` return `float`

11.3.3.1 Increment/Decrement

Incrementing and decrementing a value is provided through operators `+=` and `-=`.

- `x += n` is same as `x = x + n`
- `x -= n` is same as `x = x - n`

Python uses a special syntax for these common operations and can be extended to below operations.

- `x *= n` is same as `x = x * n`
- `x /= n` is same as `x = x / n`
- `x **= n` is same as `x = x ** n`

> Caution for `float`
>
> There are some issues and limitations with floating point arithmetic using `float`.
> It is recommended to go through them at python documentation on limitations of using float type.

11.3.4 Examples

11.3.4.1 Example 1

Below is a basic example of assigning int and float. Note that if decimal is present then, even if number is integer, it is stored as float.

```
num1 = 10; num2 = 10.0
```

```
print(f'{num1 = }, {type(num1) = }')
```

```
>>>  num1 = 10, type(num1) = <class 'int'>
```

```
print(f'{num2 = }, {type(num2) = }')
```

```
>>>  num2 = 10.0, type(num2) = <class 'float'>
```

11.3.4.2 Example 2

Operations with `int` and `float` return `float`.

```
num1 = .25; num2 = 100
num3 = num2 * num1
```

```
print(f'{num1 = }, {type(num1) = }')
```

```
>>>  num1 = 0.25, type(num1) = <class 'float'>
```

```
print(f'{num2 = }, {type(num2) = }')
```

```
>>>  num2 = 100, type(num2) = <class 'int'>
```

```
print(f'{num3 = }, {type(num3) = }')
```

```
>>>  num3 = 25.0, type(num3) = <class 'float'>
```

11.3.4.3 Example 3

Objects of type `int`, within certain range (-5 to 256), are not duplicated for performance reasons.

```
some_int_1 = 10; some_int_2 = 10
```

```
some_int_1 is some_int_2
```

```
>>>  True
```

> The basic idea is to **intern** for memory optimizations. Sometimes useful for strings, **string interning**. This causes surprizes such as this example.

11.3.4.4 Example 4

Numeric data types are immutable. In the example below, when `some_int` is assigned a new value, a new object is created in memory and bound to `some_int`.

```
some_int = 10
print(hex(id(some_int)), f'{some_int=}')
```

```
>>>  0x7c78c5500210 some_int=10
```

```
some_int += 1
print(hex(id(some_int)), f'{some_int=}')
```

```
>>>  0x7c78c5500230 some_int=11
```

11.4 String

11.4.1 Overview

In Python, a string (`str` type object) is an **immutable sequence** of unicode code points. More generally speaking it is an immutable sequence of characters, numbers and symbols.

- Strings are **sequence type**
  - like mathematics, order has meaning in sequences
    - `string` is not same as `trgins`
  - this helps enable support for **indexing** and **slicing**
- Strings are **immutable**
  - a **new object** is created in memory on modification
    - referred as **copy-on-modify**
  - adding/deleting/changing elements is not provided by default
- Python Tutorial: Gentle introduction to text
- Python library reference: Detailed documentation on `str`

11.4.2 Specifications

11.4.2.1 Overview

- Strings can be created using
  - **single quotes**: `'some string'`
  - **double quotes**: `"some string"`
  - **multi-line** strings
    - **triple single quotes**: `'''some string'''`
    - **triple double quotes**: `"""some string"""`
  - **raw** strings just need a preceding `r` character for any method
    - `r"string with \"`, `r'string with \'`
- Special string types
  - **multiline strings**
  - **raw strings**
  - **formatted string literals**
- `print` changes the way results are displayed

11.4.2.2 Basic strings

```
string_1 = 'using single quotes'
```

```
string_2 = "using double quotes"
```

```
string_3 = "including \"double quotes\" using double quotes"
```

```
string_4 = 'including "double quotes" using single quotes'
```

11.4.2.3 Multiline strings

Multiline strings can be created using triple quotes (single/double). A physical new line within a string is not included in the string. The spaces and tabs on a line are included, see `string_2` below.

```
string_1 = """This is a multiline string
with no tabs using triple double quotes"""
string_2 = '''This is a multiline string
              with tabs using triple single quotes'''
```

```
print(string_1)
```

```
>>>  This is a multiline string
>>>  with no tabs using triple double quotes
```

```
print(string_2)
```

```
>>>  This is a multiline string
>>>                with tabs using triple single quotes
```

11.4.2.4 Using backslash (\)

Backslash can be used to insert some special character sequences in a string, which are used by the print and similar functions which can parse such special character sequences.

Examples:

- newline: `\n`
- tabs: `\t`
- escape quote symbol: `\'` or `\"`

Note in below examples, when variables are output without print function, special character sequences like newline and tab are not parsed and shown as is.

```
string_1 = "Line 1\nLine 2"
string_2 = "text 1\ttext 2"
```

```
string_1; print(string_1)
```

```
>>>  'Line 1\nLine 2'
>>>  Line 1
>>>  Line 2
```

```
string_2; print(string_2)
```

```
>>>  'text 1\ttext 2'
>>>  text 1 text 2
```

11.4.2.5 Raw strings

Raw strings do not escape backslash (\). To create a raw string prepend string with `r` or `R` character.

One typical use case is to store windows path which have backslashes. Note in below example since `\u` has special meaning it gives error while creating the path which contains such sequence of characters.

```
string_1 = "C:\user\name"
```

```
>>>  (unicode error) 'unicodeescape' codec can't decode bytes in position 2-3: truncated \uXXXX escape
```

```
string_2 = r"C:\User\name"
```

```
print(string_2)
```

```
>>>  C:\User\name
```

11.4.2.6 Formatted strings

Formatted strings are used for **mixing** **hard coded text** and **variable values with formatting**.

- old syntax
  - `"text with {0[:fs]} and {1}".format(var1, var2)`
- new `f-string` syntax (Python version >= 3.6)
  - `f'text with {var1[:fs]} and {var2[:fs]}'`

  where `fs` to be read as format specifier

This is specially useful in controlling the format of output message from the code. Message could be an error, warning or a regular informative message.

There are a lot of options to play around which can be found at link.

11.4.2.6.1 Examples

11.4.2.6.1.1 Ex 1

Using old format style.

```
user_name = "First Last"
user_age = 20
my_string = "Name: {0}\nAge: {1}".format(user_name, user_age)
```

```
print(my_string)
```

```
>>>  Name: First Last
>>>  Age: 20
```

11.4.2.6.1.2 Ex 2

Using old format style with format specifier.

```
user_name = "First Last"
user_age = 20
user_balance = 1000001
my_string = "Name: {0:^30}\nAge: {1:^30}\nBalance: {2:,.2f}".format(\
  user_name, user_age, user_balance)
```

```
print(my_string)
```

```
>>>  Name:           First Last
>>>  Age:              20
>>>  Balance: 1,000,001.00
```

11.4.2.6.1.3 Ex 3

Using new f-string with format specifier.

```
user_name = "First Last"
user_age = 20
user_balance = 1000001
my_string = f"Name: {user_name:>15}\nAge: {user_age:>16}\
    \nBalance: {user_balance:>12.2f}"
```

```
print(my_string)
```

```
>>>  Name:      First Last
>>>  Age:              20
>>>  Balance:   1000001.00
```

11.4.3 Operations

Below are the 2 major categories of operations a string supports.

- **common operations on sequence types**
- **operations specific to strings**

Common sequence operations like indexing and slicing are provided with examples below, but are same for all sequence types like tuple and list.

11.4.3.1 Sequence

- operations *on* **sequence** itself
    - **length** (`len(s)`)
    - **concatenate** (`s1 + s2`), *for same type sequences*
    - **repeat** (`s*n` or `n*s`) where `n` is the number of repeats
    - **comparisons**, *for same type sequences*
- operations *on* **items** in sequence

- **retrieve by position**: index/slice (`s[i[, j[, k=1]]]`)
- `min/max`
- **check element's**
  - **existence**: `e in s`, `e not in s`
  - **index**: `s.index(e)`
  - **count**: `s.count(e)`

11.4.3.1.1 Index & Slice

Indexing refers to retrieving elements by position. Slicing refers to extracting subset of elements of a sequence.

- indexing starts at 0 and ends at n - 1
- negative indices are allowed
- usage
  - `s[i]`: return item at index `i`
  - `s[i:j]`: return items from index `i` to `j-1`
    - returns `j - i` items
  - `s[i:j:k]`: return items from index `i` to `j-1` with step `k`
    - `k=1` by default

0	1	2	3	4
item: 1	item: 2	item: 3	item: 4	item: 5
-5	-4	-3	-2	-1

11.4.3.1.2 Examples

```
string_1 = "abcdefgh"
```

Enumerate function is used to get pairs of index and elements of a sequence.

```
print([*enumerate(string_1)])
```

[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f'), (6, 'g'), (7, 'h')]

- select `c` to `f`

```
string_1[2:6]
```

```
>>>  'cdef'
```

- select second last - `g`

```
string_1[-2]
```

```
>>>  'g'
```

[(0, ‘a’), (1, ‘b’), (2, ‘c’), (3, ‘d’), (4, ‘e’), (5, ‘f’), (6, ‘g’), (7, ‘h’)]

- select last 3 elements

```
string_1[-3:]
```

```
>>>  'fgh'
```

[(0, ‘a’), (1, ‘b’), (2, ‘c’), (3, ‘d’), (4, ‘e’), (5, ‘f’), (6, ‘g’), (7, ‘h’)]

- select `d` onwards

```
string_1[3:]
```

```
>>>  'defgh'
```

- select up till `d`

```
string_1[:4]
```

```
>>>  'abcd'
```

```
string_1[:3] + string_1[3:]
```

```
>>>  'abcdefgh'
```

11.4.3.2 String specific

- There are a lot of default operations (link)
  - case, find/replace, checks, strip, split, ...
  - usually easy to use, look at reference as needed
- regular expressions
  - searching and matching patterns in strings
  - depends on module `re` in standard library
  - advanced topic, avoid at this stage

11.4.3.3 Arithmetic Operators

The `+` and `*` operations can be used with strings, other operators will give error.

Note that operations work with compatible type of objects.

- `+`: concatenate strings
  - works with `str` type objects, i.e. strings
- `*`: repeat a string
  - works with a `str` and an `int`

11.4.3.3.1 Example 1

```
some_str_1 = "some string 1"; some_str_2 = "some string 2"
concat_1_2 = some_str_1 + " " + some_str_2
print(concat_1_2)
```

```
>>>  some string 1 some string 2
```

11.4.3.3.2 Example 2

```
some_str = "xyz"
print(some_str*5)
```

```
>>>  xyzxyzxyzxyzxyz
```

11.4.3.3.3 Example 3

```
some_str_1 = "some string 1"; some_str_2 = "some string 2"
concat_1_2 = some_str_1 * some_str_2
```

```
>>>  Error: TypeError: can't multiply sequence by non-int of type 'str'
```

11.5 Tuple

11.5.1 Overview

Tuple is an **immutable collection** of **ordered**, **heterogeneous** objects with below features

- **sequence type** (collection of ordered objects)
  - this helps enable support for **indexing** and **slicing**
  - the position of data has meaning
  - useful in passing and operating on set of objects within code
- **heterogeneous**: *can contain any type of object*
  - more efficient if items are homogeneous
- **immutable**
  - modify **in-place** operations are not supported
    - a new object is created in memory on modification if elements are immutable
  - adding/deleting/changing elements is not provided by default
- Python Tutorial: Gentle introduction to tuples
- Python library reference: Detailed documentation on sequences

11.5.2 Specifications

11.5.2.1 Creation syntax

- using **commas**
  - comma decides the tuple
  - parenthesis are just for code readability
- using tuple **constructor**: `tuple()`
- using **unpacking** (Python special syntax)
- using **comprehension** (covered in Python special features)

11.5.2.2 Creation by use case

- using **elements**
  - *0 item*: `()` or `tuple()`
  - *1 item*: `i,` or `(i,)`
    - `(i)` will give error, comma is needed
  - *more than 1 item*: `i1, i2, i3` or `(i1, i2, i3)`
- using **elements from another iterable**[s]:
  - using tuple **constructor**: `tuple(iterable[s])`
  - using **unpacking**
    - `t = *l,`, `t = (*l,)`, `t = (*s, *l)`
    - `t = (*l)` will give error, comma is needed

11.5.3 Operations

Tuple has access to **common** operations on **sequence types** with no additional methods.

- operations *on* **sequence** itself
  - **length** (`len(s)`)
  - **concatenate** (`s1 + s2`), *for same type sequences*
  - **repeat** (`s*n` or `n*s`) where `n` is the number of repeats
  - **comparisons**, *for same type sequences*
- operations *on* **items** in sequence
  - **retrieve by position**: index/slice (`s[i[, j[, k=1]]]`)
  - `min/max`
  - **check element's**
    - **existence**: `e in s`, `e not in s`
    - **index**: `s.index(e)`
    - **count**: `s.count(e)`

11.6 List

11.6.1 Overview

List is a **mutable collection** of **ordered**, **heterogeneous** objects with following features.

- **sequence type**
  - this helps enable support for **indexing** and **slicing**
  - the position of data has meaning
  - useful in passing and operating on set of objects within code
- **heterogeneous**: can contain any type of object
  - more efficient if items are homogeneous
- **mutable**
  - adding/deleting/changing elements is provided by default
  - modify **in-place** operations are supported
- Python Tutorial: Gentle introduction to list: part 1, part 2
- Python library reference: Detailed documentation on sequences

11.6.2 Specifications

- **empty** list: `[]` or `list()`
- using **elements**: `[i1, i2, ...]`, `[i1]`
- using **elements from iterable**[**s**]:
  - using list **constructor**: `list(iterable)`
  - using **unpacking**
    - `[*t]`, `[*t,]`, `[*s, *t, *l]`
  - using **comprehension** (covered in Python special features)

11.6.3 Operations

There are 2 sets of operations a list supports.

- **common** operations on **sequence types**
- operations on **mutable** sequence types

11.6.3.1 Sequence operations

- operations *on* **sequence** itself
  - **length** (`len(s)`)
  - **concatenate** (`s1 + s2`), *for same type sequences*
  - **repeat** (`s*n` or `n*s`) where `n` is the number of repeats
  - **comparisons**, *for same type sequences*
- operations *on* **items** in sequence
  - **retrieve by position**: index/slice (`s[i[, j[, k=1]]]`)
  - `min/max`
  - **check element's**

- **existence**: `e in s`, `e not in s`
- **index**: `s.index(e)`
- **count**: `s.count(e)`

11.6.3.2 Mutable sequence operations

- most of the operations are in-place
  - implies no new object creation on modification
- **operations on sequence itself**
  - copy (shallow), extend, repeat, reverse
- **operations on items**
  - delete, replace, append, clear all, remove, insert, pop
- link

11.7 Range

Range is a special **iterable** to **generate a sequence of integers** with following characteristics.

- **immutable sequence type**
- cannot see all elements at a time
  - have to be unpacked into a list or tuple
- syntax for creation
  - `range(stop)`
  - `range(start, stop[, step])`
    - **start** is **included**
    - **stop** is **excluded**
- primarily used for loops which is discussed at Section 12.3.1

11.7.1 Examples

```
print(range(10))
```

```
>>>  range(0, 10)
```

```
print(list(range(10)))
```

```
>>>  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print([*range(11)])
```

```
>>>  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
print([*range(1, 11)])
```

```
>>>  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
print([*range(-1, -11)])
```

```
>>>  []
```

```
print([*range(-1, -11, -1)])
```

```
>>>  [-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
```

11.8 Dictionary

11.8.1 Overview

A dictionary is a **mutable mapping type** collection of **heterogeneous objects mapped** to **keys** that are **hashable** and **unique** objects.

- in newer versions (>3.9) the order is guaranteed
- collection of `{key: value}` pairs where
  - **key** can be any **hashable object**
    - **strings, numeric data types can be used**
    - immutable type which contain only immutable objects
    - tuples with immutable objects can be used
    - **lists, dictionary cannot be used**
  - **value** can be any **Python object**

A dictionary is useful when a collection of objects is needed with the option to do quick searches based on keys rather than index, unlike sequences.

- Python Tutorial: Gentle introduction to dictionary
- Python library reference: Detailed dictionary documentation

Concept of hashable objects is introduced in DSA (Section 21.4.4).

11.8.2 Specifications

- using key value pairs separated by commas
  - `d = {"key1": value1, "key2": value2, ...}`
- using type constructor
  - `d = dict([("key1", value1), ("key2", value2), ...])`
  - `d = dict(key1=value1, key2=value2)`
- create empty dictionary

    - `d = {}`
    - `d = dict()`

- using comprehensions (covered in Python special features)
- if a key is passed multiple times, final value exists

11.8.3 Operations

- operations on **dictionary itself**
- operations on **keys** and **values**

11.8.3.1 Operations on dictionary

- length: `len(d)`
- clear: `d.clear()`
- shallow copy: `d.copy()`
- update from another dictionary: `d.update([other])`

11.8.3.2 Operations on keys and values

- **check** keys: `key in d` / `key not in d`
- **view all** keys/values: `d.keys()` / `d.values()` / `d.items()`
- **get all** keys/values as list of tuples: `list(enumerate(d))`
- **get all** keys as list: `list(d)`
- **get all** keys as list reversed: `list(reversed(d))`
- **get value**, error if key not present: `d[key]`
- **set value**, inserts key if key not present: `d[key] = value`
- **del key/value**, deletes last entry and returns deleted key, value: `d.popitem()`
- if key not present return default if defined else error
    - **get value**: `d.get(key[, default])`
    - **del key/value** and return deleted value: `d.pop(key[, default])`

11.9 Set

Set is a collection of unique objects with operations related to math sets available, e.g. union, intersection.

In other words, set is a special dictionary with keys only.

In Python specifically, set is an unordered collection of hashable objects. In newer versions (>3.9) the order is guaranteed.

- Python tutorial for sets
- Python documentation on set type

Sets are commonly used for

- membership testing: search by value
- removing duplicates from a collection

11.9.1 Specifications

A set can be created using curly braces with the exception of empty set.

- create a set with valid keys
    - curly braces
    - `set` constructor

```
some_set = {key1, key2, ...}
some_set = set(iterable)
```

- empty set can be created using `set()` constructor
    - using `{}` creates an empty dictionary

11.10 Boolean data type

Boolean data type, `True` and `False`, is the fundamental unit for implementing boolean conditional expressions.

Boolean comparison operator are used to create elementary conditions. Combination operators allow for building larger conditions by combining multiple conditions.

Conditional control flow blocks, `if` and `match`, use conditions.

The idea is based on boolean math. It is essential in controlling the flow of the program based on state of one or more objects in the program.

- `bool` in Python, based on usage, can refer to
    - data type
    - function
- `bool` type is used to represent `boolean values`
- `bool` type inherits from `int` type
- `bool` data type can take value from 2 built-in constants, `True` and `False`
    - underlying `int` values are `int(True) = 1` and `int(False) = 0` respectively
- *can be stored in variables like other objects*
    - *useful in conditional blocks*

Below are some examples for familiarity.

- basics

```
bool(True), int(True), type(True)
```

```
>>>  (True, 1, <class 'bool'>)
```

```
bool(False), int(False), type(False)
```

```
>>>  (False, 0, <class 'bool'>)
```

- storing in variables
    - it is useful to name boolean variables like `is_<some check>`

```
is_int = True
is_int, bool(is_int), int(is_int), type(is_int)
```

```
>>>  (True, True, 1, <class 'bool'>)
```

11.10.1 Boolean comparison operators

Boolean comparison operators are used for object comparisons and return `True` or `False`, if used with compatible object types.

They are mostly used to create boolean **conditions** which are used in conditional blocks, `if` and `match..case`.

When used with sequence types (string, tuple, list)

- equality operators (`==`, `!=`) return `True` if all elements are equal (not equal) in content
- inequality operators only test minimum and maximum as appropriate
- membership testing check for existence of elements and is most useful

11.10.1.1 Options and syntax

Operation	Python Operator	Comments
basic value comparisons	`==`, `!=`, `<`, `>`, `<=`, `>=`	• compares values • different types ok, but must be compatible
membership testing	`in`, `not in`	• used with sequence types
object id comparison	`is`, `is not`	• compares memory address • works on all type

11.10.1.2 Examples

11.10.1.2.1 Numeric

```
num_1 = 10; num_2 = 15; num_3 = 10.0
```

```
num_1 == num_2, num_1 < num_2, num_1 <= num_3
```

```
>>>  (False, True, True)
```

- can be stored in variables

```
cnd = num_1 > num_3
```

```
print(f"{cnd = }, {type(cnd) = }")
```

```
>>>  cnd = False, type(cnd) = <class 'bool'>
```

11.10.1.2.2 Sequence type

Membership testing is more useful for sequence types. Below examples illustrate the usage.

11.10.1.2.2.1 Strings

- test character in a string

```
some_string = "abcd"
some_chr_1 = "a"
some_chr_2 = "e"
```

```
some_chr_1 in some_string
```

```
>>>  True
```

```
some_chr_2 in some_string
```

```
>>>  False
```

- test a small string in a longer string

```
some_long_str = "A reasonably long string"
some_short_str = "long"
```

```
some_short_str in some_long_str
```

```
>>>  True
```

11.10.1.2.2.2 Tuples and lists

```
some_list = [1, 2, 3, 4, (1, 2, 3)]
some_tuple = 1, 2, 3
num_1 = 3; num_2 = 5
```

```
num_1 in some_list
```

```
>>>  True
```

```
some_tuple in some_list
```

```
>>>  True
```

```
num_2 in some_tuple
```

```
>>>  False
```

- store a boolean operation in a variable

```
cnd = some_tuple in some_list
```

```
>>>  cnd = True, type(cnd) = <class 'bool'>
```

11.10.1.2.3 None type

Since there is a single instance of `None` type object created in a Python session, object id comparison is useful. `None` is used to signify if a variable is defined but not assigned a value yet.

Below example illustrates the object id comparison for `None` type in isolation.

```
some_var = None
some_var is None
```

```
>>>  True
```

11.10.2 Boolean combination operators

Boolean combination operators use boolean math to provide means of combining multiple comparison operations and conditions to form larger conditions.

Operation	Python Operator	Comments
not	`not x`	• inverts the bool value • if `x` is false, then `True`, else `False`
and	`x and y`	• test if both conditions are `True`
or	`x or y`	• test if any condition is `True`
()	`(cnd1 and cnd2) or (cnd3)`	• used to group conditions • conditions inside `()` are evaluated first

- basic examples of combining conditions
  - `a == 10`
  - `a >= 5 and a <= 10`
  - `(a > 0 and a < 10) or (a >= 10 and a < 25)`
- **order of precedence** used for evaluation
  1. `()`
  2. comparison operators have same priority (`==`, `!=`, `<`, `>`, `<=`, `>=`)
  3. `not` > `and` > `or`
- **chained comparisons**
  - are automatically converted to paired `and` comparisons
  - example: `a < b < c` is same as `a < b and b < c`
  - this is specific to Python
- for conditions with too many nested combinations it is recommended to use `()`
  - best for code readability
  - avoid errors due to precedence order
- *can be stored in variables like other objects*
  - and used later in control flow

`and` and `or` operators are based on logic gates in boolean math. Truth tables, given below, summarize results for logic gates. 0 and 1 are used instead of `False` and `True` for better readability.

x	y	x and y	x or y
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

There are some additional features which Python provides related to boolean data type and are discussed in the Python special features chapter (Section 15.1). They are left from this section to keep the complexity low at this stage.

11.11 Generic concepts

11.11.1 Iterable unpacking

Iterable unpacking is a special feature in newer versions of Python. Some features were introduced in Python version 2, more features added using PEP-3132: Extended Iterable Unpacking in version 3.

- * unpacks remaining items
- returns a list
- advantages
  - **better code readability**
  - **easier** than using indexing
  - **faster**
- gives **error** if there is a **mismatch** in number of items and variables

11.11.1.1 Examples

- Unpack and assign elements of an iterable to variables
  - get first and remaining items of an iterable

```
some_list = [1, 2, 3, 4]; some_tuple = (1, 2, 3, 4)
```

```
first_item = some_list[0]
end_items = some_list[1:]
```

```
print(f'{first_item = }, {end_items = }')
```

```
>>>  first_item = 1, end_items = [2, 3, 4]
```

```
first_item, *end_items = some_list
```

```
print(f'{first_item = }, {end_items = }')
```

```
>>>  first_item = 1, end_items = [2, 3, 4]
```

- Unpack and assign elements of an iterable to variables
    - get last and remaining items of an iterable

```
some_list = [1, 2, 3, 4]; some_tuple = (1, 2, 3, 4)
```

```
begin_items = some_tuple[0:-1]
last_item = some_tuple[-1]
```

```
print(f'{begin_items = }, {last_item = }')
```

```
>>>  begin_items = (1, 2, 3), last_item = 4
```

```
*begin_items, last_item = some_tuple
```

```
print(f'{begin_items = }, {last_item = }')
```

```
>>>  begin_items = [1, 2, 3], last_item = 4
```

- Unpack and assign elements of an iterable to variables
    - get first two, last and remaining middle items of an iterable

```
some_list = [1, 2, 3, 4, 5, 6, 7]; some_tuple = (1, 2, 3, 4, 5, 6, 7)
```

```
first_item, second_item, *remaining_items, last_item = some_tuple
```

```
print(f'{first_item = }, {second_item = }')
```

```
>>>  first_item = 1, second_item = 2
```

```
print(f'{remaining_items = }, {last_item = }')
```

```
>>>  remaining_items = [3, 4, 5, 6], last_item = 7
```

- Combine iterables into another

```
some_list_1 = [1, 2, 3]; some_tuple_1 = (4, 5);
some_tuple_2 = (*some_list_1, *some_tuple_1)
```

```
>>>  some_tuple_2=(1, 2, 3, 4, 5)
```

- ** for mapping types - dictionary

```
some_dict_1 = {"key1": "value1", "key2": "value2.1"}
some_dict_2 = {"key2": "value2.2", "key3": "value3"}
some_dict_3 = {**some_dict_1, **some_dict_2}
```

```
>>>  some_dict_3={'key1': 'value1', 'key2': 'value2.2', 'key3': 'value3'}
```

11.11.2 Implications of mutability

- **Modify in-place**: any modification to the object does not lead to creation of a new object
- **Copy on modify**: create a new copy of object if modified, opposite of modify in-place
- **Mutable $\implies$ modify in-place**
  - Lists and dictionaries can be modified without creation of new object on RAM
- **Immutable $\implies$ copy on modify**
  - Strings and tuples create new objects on RAM if modified

Modify in-place means any modification to the object does not lead to creation of a new object. For e.g. strings and tuples create new objects on RAM if modified, whereas lists and dictionaries can be modified without creation of new object on RAM.

- Implications
  - **flexibility**
  - **efficiency** (in terms of speed and memory)
- Immutable objects (strings, tuples) are
  - **efficient** for **constant data**
  - **less flexible**
- Mutable objects (lists, dictionaries) are
  - **less efficient** for **constant data**
  - **flexible**, support in-place modification
  - ***can be more efficient if data keeps changing over time***

11.11.2.1 Changing elements

Since strings and tuples are immutable, elements cannot be assigned new values though indexing. This is unlike mutable types where this is allowed, e.g. lists.

```
some_string = "abcdee"
some_string[-1] = "f"
```

```
>>>  Error: TypeError: 'str' object does not support item assignment
```

```
some_tuple = (0, 1, 1)
some_tuple[2] = 2
```

```
>>>  Error: TypeError: 'tuple' object does not support item assignment
```

```
some_list = [0, 1, 5]
some_list[2] = 2
print(some_list)
```

```
>>>  [0, 1, 2]
```

Strings have internal methods that can change elements, but then they follow copy-on-modify.

```
some_string_1 = "abc"
some_string_2 = some_string_1.replace("a", "b")
```

```
print(f'{some_string_1=}, {some_string_2=}\n\
    {some_string_1 is some_string_2 = }')
```

```
>>>  some_string_1='abc', some_string_2='bbc'
>>>      some_string_1 is some_string_2 = False
```

```
some_string = "abc"; some_string_orig_id = hex(id(some_string))
```

```
>>>  some_string='abc', some_string_orig_id = '0x7c78c543cd70'

>>>  hex(id(some_string.replace("a", "b"))) = '0x7c78b9f8b2f0'

>>>  some_string='abc', some_string_orig_id = '0x7c78c543cd70'
```

```
some_string = some_string.replace("a", "b")
```

```
>>>  some_string='bbc'
>>>  hex(id(some_string)) = '0x7c78b9fc32f0'
>>>  some_string_orig_id = '0x7c78c543cd70'
```

11.11.2.2 Propagation of changes

Mutable types like lists or dictionaries, when passed around through variable assignment, changes are propagated.

- when you make changes to original list they propagate
    - this is required in many cases
    - but can also lead to a bug
    - *be aware of the concept*
- to *avoid default behavior* when needed use
    - **constructor** `list(iterable)`
    - **unpacking** `[*iterable]`
    - **loops**

```
some_list_1 = [1, 2, "a", "b"]
some_list_2 = some_list_1
```

```
print(f'{some_list_1=}, {some_list_2=}\n{some_list_2 is some_list_1 = }')
```

```
>>>  some_list_1=[1, 2, 'a', 'b'], some_list_2=[1, 2, 'a', 'b']
>>>  some_list_2 is some_list_1 = True
```

```
some_list_1[2] = "abc"
```

```
print(f'{some_list_1=}, {some_list_2=}\n{some_list_2 is some_list_1 = }')
```

```
>>>  some_list_1=[1, 2, 'abc', 'b'], some_list_2=[1, 2, 'abc', 'b']
>>>  some_list_2 is some_list_1 = True
```

```
some_list_2[-1] = "xyz"
```

```
print(f'{some_list_1=}, {some_list_2=}\n{some_list_2 is some_list_1 = }')
```

```
>>>  some_list_1=[1, 2, 'abc', 'xyz'], some_list_2=[1, 2, 'abc', 'xyz']
>>>  some_list_2 is some_list_1 = True
```

- using constructor or unpacking does not pass the object itself

```
some_list_1 = [1, 2, "a", "b"]
some_list_2 = list(some_list_1)
```

```
>>>  some_list_1=[1, 2, 'a', 'b'], some_list_2=[1, 2, 'a', 'b']
>>>  some_list_2 is some_list_1 = False
```

```
some_list_1[2] = "abc"
```

```
>>>  some_list_1=[1, 2, 'abc', 'b'], some_list_2=[1, 2, 'a', 'b']
>>>  some_list_2 is some_list_1 = False
```

- using constructor or unpacking does not pass the object itself

```
some_list_1 = [1, 2, "a", "b"]
some_list_2 = [*some_list_1]
```

```
print(f'{some_list_1=}, {some_list_2=}\n{some_list_2 is some_list_1 = }')
```

```
>>>  some_list_1=[1, 2, 'a', 'b'], some_list_2=[1, 2, 'a', 'b']
>>>  some_list_2 is some_list_1 = False
```

```
some_list_1[2] = "abc"
```

```
print(f'{some_list_1=}, {some_list_2=}\n{some_list_2 is some_list_1 = }')
```

```
>>>  some_list_1=[1, 2, 'abc', 'b'], some_list_2=[1, 2, 'a', 'b']
>>>  some_list_2 is some_list_1 = False
```

11.11.2.3 Mutable in immutable

- `tuple` is immutable in terms of its element objects
- the contained object remains mutable if it is mutable

```
some_list = [1, 2, 3, 4, 5]
some_tuple = (some_list, "some other object")
```

```
print(f'{some_list=}\n{some_tuple=}\n{some_list is some_tuple[0] = }')
```

```
>>>  some_list=[1, 2, 3, 4, 5]
>>>  some_tuple=([1, 2, 3, 4, 5], 'some other object')
>>>  some_list is some_tuple[0] = True
```

```
some_list.pop()
```

```
>>>  5
```

```
print(f'{some_list=}\n{some_tuple=}\n{some_list is some_tuple[0] = }')
```

```
>>>  some_list=[1, 2, 3, 4]
>>>  some_tuple=([1, 2, 3, 4], 'some other object')
>>>  some_list is some_tuple[0] = True
```

- if only contents are needed and propagation is to be avoided use **unpacking** or **constructor**

```
some_list = [1, 2, 3, 4, 5]
some_tuple_1 = *some_list,; some_tuple_2 = tuple(some_list)
```

```
print(f'{some_list=}\n{some_tuple_1=}\n{some_tuple_2=}')
```

```
>>>  some_list=[1, 2, 3, 4, 5]
>>>  some_tuple_1=(1, 2, 3, 4, 5)
>>>  some_tuple_2=(1, 2, 3, 4, 5)
```

```
print(f'{some_list is some_tuple_1 = }')
```

```
>>>  some_list is some_tuple_1 = False
```

```
print(f'{some_list is some_tuple_2 = }')
```

```
>>>  some_list is some_tuple_2 = False
```

```
some_list.pop()
```

```
>>>  5
```

```
print(f'{some_list=}\n{some_tuple_1=}\n{some_tuple_2=}')
```

```
>>>  some_list=[1, 2, 3, 4]
>>>  some_tuple_1=(1, 2, 3, 4, 5)
>>>  some_tuple_2=(1, 2, 3, 4, 5)
```

```
print(f'{some_list is some_tuple_1 = }')
```

```
>>>  some_list is some_tuple_1 = False
```

```
print(f'{some_list is some_tuple_2 = }')
```

```
>>>  some_list is some_tuple_2 = False
```

11.11.2.4 Shallow vs deep copy

Shallow copy creates a new object for the collection being copied but does not create new objects, if items in the collection are themselves collection. This can have side effects.

Regular copy method available in all collections (string, tuple, list, dictionary) makes a shallow copy.

This works fine if elements of the collection are immutable objects like numbers, strings or tuples, but if there are mutable types like list or dict then propagation will occur.

This might not be desirable at times, so a **deep copy** is needed which creates new objects for all elements of the original container going through nested structure of the collection recursively.

The standard library has `copy` module which has `deepcopy` function to achieve this.

Below example illustrates the point. It is recommended to do experiments to understand the concept.

`some_list_1` is a list containing a list, tuple and a dictionary.

`some_list_2` is a regular copy, so is different object from `some_list_1`, but elements point to the same underlying `some_list`, `some_tuple` and `some_dict`

`some_list_3` is a regular copy from `copy` module, so behaves similar to `some_list_2`.

`some_list_4` is a deep copy from `copy` module, so elements are different objects as well.

```
import copy
some_list = [1,2,3]; some_tuple = (4, 5); some_dict = {"six": 6, "seven": 7}
some_list_1 = [some_list, some_tuple, some_dict]
some_list_2 = some_list_1.copy()
some_list_3 = copy.copy(some_list_1)
some_list_4 = copy.deepcopy(some_list_1)
```

```
>>>  some_list_2 is some_list_1 = False
```

```
>>>  some_list_3 is some_list_1 = False
```

```
>>>  some_list_4 is some_list_1 = False
```

```
>>>  some_list_2[0] is some_list_1[0] = True

>>>  some_list_3[0] is some_list_1[0] = True

>>>  some_list_4[0] is some_list_1[0] = False
```

12 Control Flow

Control the flow of a computer program

12.1 Introduction

12.1.1 Overview

When do you need to control the flow of a program?

- *change outcome based on state of object[s] (condition[s])*
  - **conditional blocks**: `if...[elif]....[else]`, `match...case`
- *repeat certain task[s] or iterate through a collection of data*
  - **loops**: `for...[else]`, `while...[else]` blocks
- *avoid stopping of program on errors and handle the flow differently*
  - **error handlers**: `try...[except]...[else]...[finally]` blocks
- *remove dependency on certain tasks*
  - **asyncio** (asynchronous input output)
- *increase efficiency by optimizing redirection of tasks to multiple cpu cores/threads*
  - **parallel computing**, …

Note: square brackets are generally used to represent something optional

As a beginner to programming conditional blocks and loops are sufficient to get started.

Error handlers are not needed for most of the basic stuff, but there are a few cases where it is useful to know the basics. For example it is useful to know the basics of using `try` block for a few situations. Most of debugging is based on the concepts related to error handlers. Editors provide support for debugging using these concepts. Developers rely heavily on using these concepts to handle expected errors differently. Therefore it is useful to understand the basics of error handling as it is certain to make errors and handle them.

Asynchronous input/output (asyncio) and parallel computing are advanced topics used to increase efficiency of larger programs. They are not covered.

12.1.2 Objectives

For all of the control flow techniques, **conditionals**, **loops** and **error handlers**

- get acquainted to specifications
- understand the rules and few implications conceptually
- examples of common use cases
- understand conceptually when to use what
- experiment with individual pieces

12.2 Conditional blocks

Conditional blocks use conditions to control and change the flow of a program. Conditions are created using boolean operations.

- `if...[elif]...[elif]...[else]` blocks
    - covers everything needed from basic conditional blocks
- `match...case...[case_]` blocks
    - special case where a variable has to be tested for different values
    - in most cases can be implemented using `if` block
    - better for code readability
    - has some additional special features related to pattern matching

12.2.1 `if` blocks

`if` blocks are used to execute some code only when a condition is evaluated to `True`.

12.2.1.1 Specifications

Below are possible forms of `if` block.

```
if condition:
    # if block content
```

```
if condition:
    # if block content
else:
    # else block content
```

```
if condition_1:
    # if block content
elif condition_2:
    # elif block 1 content
elif condition_3:
    # elif block 2 content
else:
    # else block content
```

```
# execute some short expression conditionally
if condition: <short expression>
```

- start an `if` block
    - `if` statement with an expression ending in colon (`:`)
- end an `if` block
    - short expression on the same line
    - **indented** block of one or more lines of code
- there can be 0 or more `elif` (else if) blocks
- there can be 0 or 1 `else` block
- syntax of `elif` and `else` is same as `if`

12.2.1.1.1 Ternary operator

Python additionally provides a ternary form for short, one line conditionals. Below is the syntax, where `X` and `Y` are short expressions to be evaluated and returned.

Ternary operator is useful for code readability. It removes the requirement for a multiline `if` block for quick checks required in code.

```
X if condition else Y
```

```
print("condition True") if 1 > 0 else print("condition False")
```

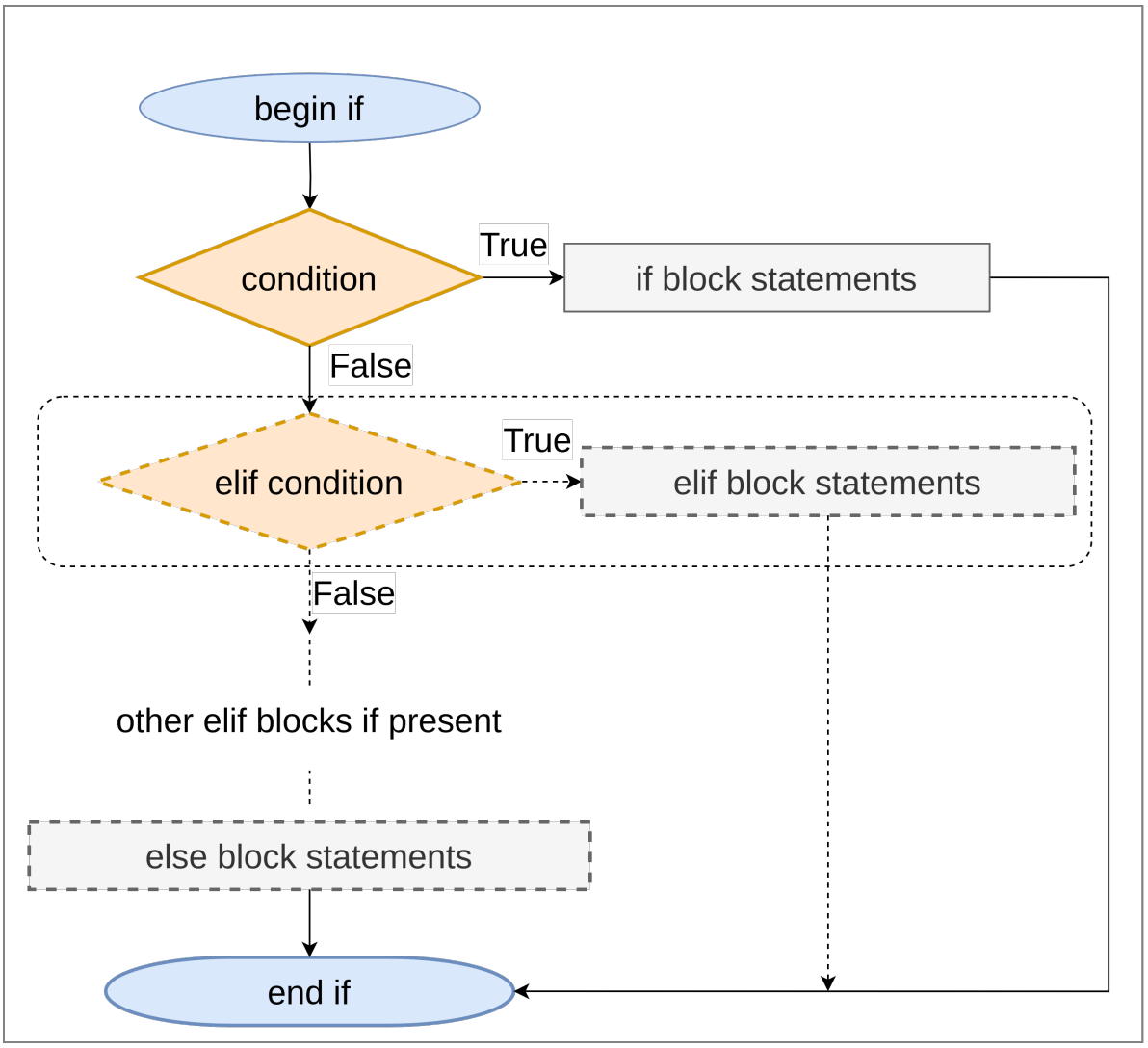
```
>>>  condition True
```

```
print("condition True") if 1 == 0 else print("condition False")
```

```
>>>  condition False
```

12.2.1.2 Control flow

Flow diagram below illustrates the control flow for a if block.



Basic idea

- check `if` and `elif` conditions sequentially
    - if any of the conditions evaluate to `True`
        - execute the corresponding block content and exit
    - if all of the conditions evaluate to `False`
        - execute `else` block if present and exit

12.2.1.3 Examples

12.2.1.3.1 Basic

```
some_num = 12
if some_num >= 0:
    print(f'{some_num} is positive')
```

```
>>>  12 is positive
```

12.2.1.3.2 With `else` block

```
some_num = 12
if some_num >= 0:
    print(f'{some_num} is positive')
else:
    print(f'{some_num} is negative')
```

```
>>>  12 is positive
```

12.2.1.3.3 With `elif` block

```
some_num = 12
if some_num == 0:
    print(f'{some_num} is zero')
elif some_num > 0:
    print(f'{some_num} is positive')
elif some_num < 0:
    print(f'{some_num} is negative')
```

```
>>>  12 is positive
```

12.2.1.3.4 With `elif` and `else`

```
some_num = 12
if some_num == 0:
    print(f'{some_num} is zero')
elif some_num > 0:
    print(f'{some_num} is positive')
else:
```

```
    print(f'{some_num} is negative')
```

```
>>>  12 is positive
```

12.2.2 `match...case` block

Match blocks are used if some object is to be tested against multiple cases. It can be achieved using `if` blocks but `match` blocks are better for code readability and ease of use for the given use case.

`_` is optional and for case when none of the options match.

```
match some_obj:
    case option_1:
        # do something and exit
    case option_2:
        # do something and exit
    [case _:
        # do something and exit
    ]
```

```
if some_obj == option_1:
    # do something and exit
elif some_obj == option_2:
    # do something and exit
[else
    # do something and exit
]
```

More information can be found at Python documentation for match statements.

12.3 Loops

- Loops are needed for iteration
  - **repeating** certain pieces of code
  - iterating over collections to **access**, **operate** or **modify** elements
- When **number of repetitions** is
  - **known**: use `for` block
  - **not known**: use `while` block
- It is recommended not to modify the collection that is being iterated, instead use any of the below solutions
  - create a new collection
  - create a copy

  Iterating is a common term which refers to going through elements of a collection. **Iterables** and **iterators** in Python are based on this idea.

12.3.1 `for` block

12.3.1.1 Specifications

```
for item in iterable:
    # do something
    # item is available
    print(item)
for i in range(n):
```

```
    # do something
    # i is available
    print(item)
```

```
for item in iterable: print(item)
```

Fundamental form

- `for` keyword declares the start of a `for` block
- `item in iterable` is the generic form
- `:`, colon, to declare end of `for` declaration
- code to be repeated, which has access to an item for a given iteration
    - if a short expression has to be repeated it can be used on the same line

Using this fundamental form other variation are created. e.g.

- `range(n)` function is used to loop through fixed number of times
    - example to repeat 10 times
        - `for i in range(10):` i takes values 0 through 9
        - `for i in range(1, 11):` i takes values 1 through 10
    - `range()` function is discussed at Section 11.7
- loops can be nested using indentation
    - useful for working with nested data structures

12.3.1.1.1 `continue`

`continue` clause causes the loop to jump to next iteration without executing remaining lines in loop.

In the below structure, when the condition is true, item will not be printed.

```
for item in iterable:
    # do something
    # item is available
    if condition:
        continue
    print(item)
```

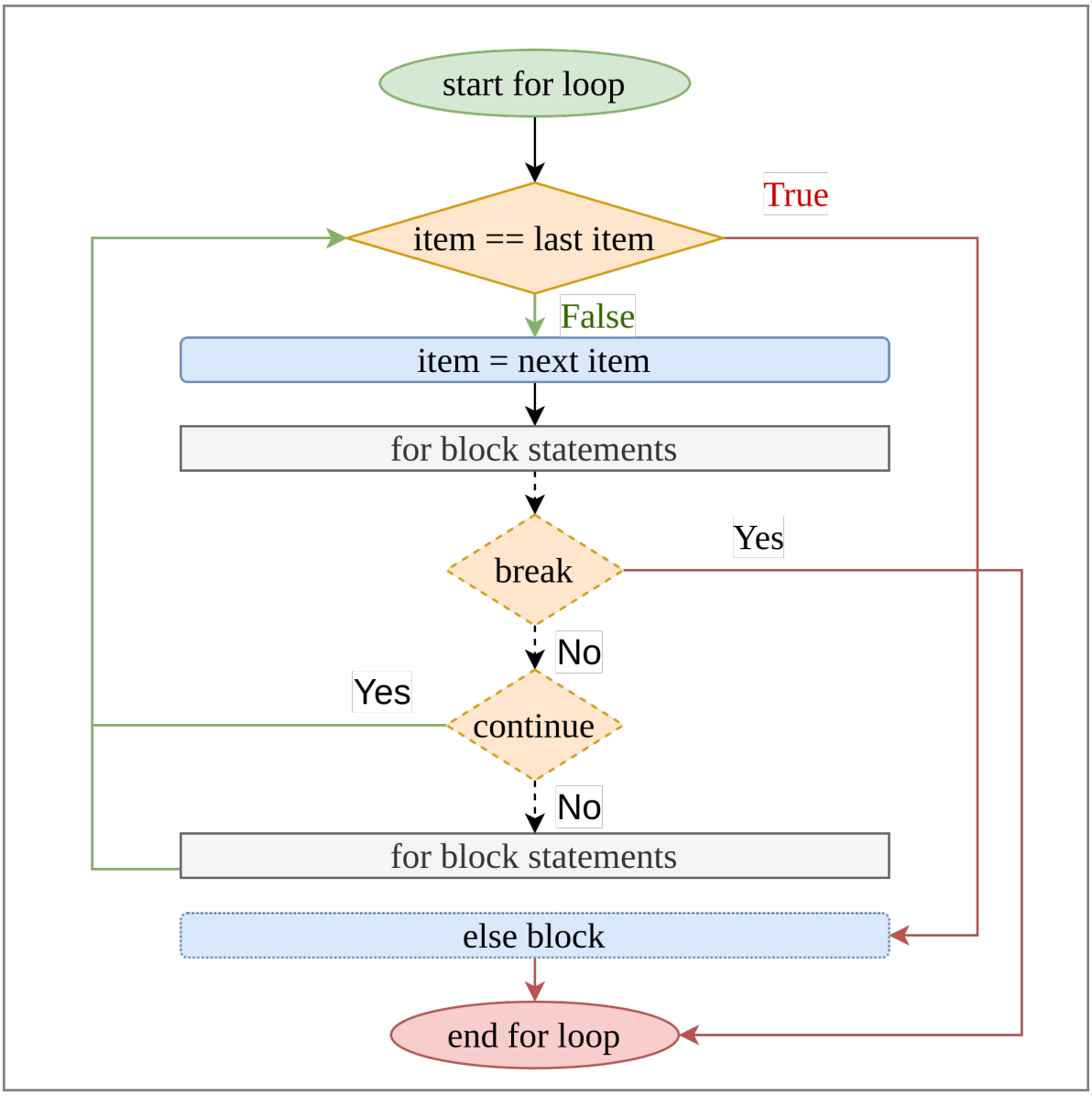
Continue is useful if you need to skip execution of some code for certain elements of the iterable.

12.3.1.1.2 `break` and `else`

- `break` clause causes exit of the **innermost** loop
    - is **optional**
    - **innermost** is critical when there are nested loops
- `else` clause
    - is **optional**
    - is **executed** only if loop ends normally, i.e. no `break` is hit

```
for item in iterable:
    # do something
    # item is available
    if condition:
        break
    print(item)
else:
    print("No break found")
```

12.3.1.2 Control flow



12.3.1.3 Examples

12.3.1.3.1 Basic

- Loop through 1 to `n_max = 20`
  - store even numbers in a list `evens`
  - store odd numbers in a list `odds`

```
n_max = 20
evens = []; odds = []
for i in range(1, n_max + 1):
    evens.append(i) if i % 2 == 0 else odds.append(i)
```

```
print(evens)
```

```
>>>  [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
print(odds)
```

```
>>>  [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

12.3.1.3.2 `continue`

In the below example only odd numbers are printed as the loop hits `continue` for even numbers.

```
for i in range(5):
    if i % 2 == 0: continue
    print(i)
```

```
>>>  1
>>>  3
```

12.3.1.3.3 `break` and `else`

- For a given list of numbers
  - validate if the numbers are within some limits, e.g. [0, 100]
  - if all values are within limits
    - print “validation successful”
  - if any of the values are outside limits
    - print “validation failed” with value
    - exit the loop

```
correct_list = [65, 24, 53, 91, 59, 81, 93, 7, 78, 10]
for num in correct_list:
    if num < 0 or num > 100:
        print(f"validation failed, list contains {num}")
        break
else:
    print("validation successful")
```

```
>>>  validation successful
```

```
incorrect_list = [97, 144, 115, 127, 33, 99, 85, 109, 21, 110]
for num in incorrect_list:
    if 0 <= num <= 100:
```

```
        continue
    else:
        print(f"validation failed, list contains {num}")
        break
else:
    print("validation successful")
```

```
>>>  validation failed, list contains 144
```

Notice that `else` block is useful in this situation as it is run only when for loop iteration is complete without hitting `break` statement.

There are different ways to structure the same conditions and required outcome.

12.3.1.3.4 Nested loops

Print table of numbers from 1 through 12 for multiplication with 1 through 10.

```
for i in range(1, 13):
    for j in range(1, 11):
        end = "\n" if j == 10 else "\t"
        print(i*j, end=end)
```

```
>>>  1  2   3   4   5   6   7   8   9   10
>>>  2  4   6   8   10  12  14  16  18  20
>>>  3  6   9   12  15  18  21  24  27  30
>>>  4  8   12  16  20  24  28  32  36  40
>>>  5  10  15  20  25  30  35  40  45  50
>>>  6  12  18  24  30  36  42  48  54  60
>>>  7  14  21  28  35  42  49  56  63  70
>>>  8  16  24  32  40  48  56  64  72  80
>>>  9  18  27  36  45  54  63  72  81  90
>>>  10 20  30  40  50  60  70  80  90  100
>>>  11 22  33  44  55  66  77  88  99  110
>>>  12 24  36  48  60  72  84  96  108 120
```

12.3.2 `while`

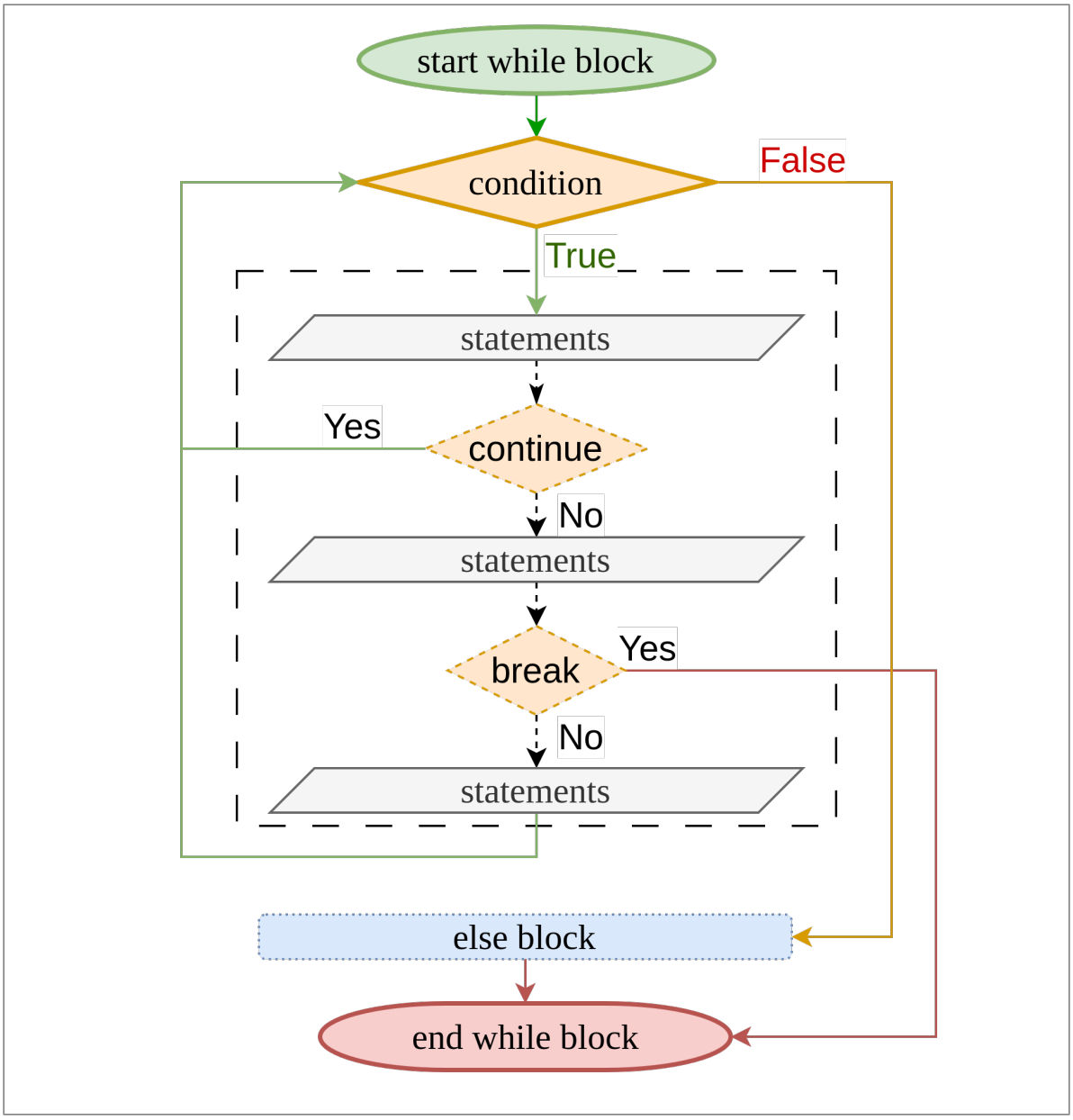
12.3.2.1 Specifications

```
while condition:
    # code block
    [continue]
    # code block
    [break]
[else]:
    # code block
```

- loop repeats while **condition** is `True` or `break` statement is hit
- `continue`, `break`, `else` clauses are optional

Below diagram illustrates the control flow for a while loop.

- main `while` block contents are executed repeatedly until condition is `True`
  - if `continue` statement is hit
    - loop restarts and condition is checked again
- if `break` statement is hit anywhere loop exits
  - without going through `else` block even if present
- when condition is `False`
  - `else` block is executed if present
  - loop is exited



12.3.2.2 Use cases

A while loop is needed when number of repetitions is not known in advance

- When data structure is dependent on external sources
  - e.g. user input, web data base, ...

- Recursive algorithms
    - Computer science: binary search, merge sort, etc.
    - Math:
        - numerical methods of approximations
        - algorithm to find greatest common divisor

12.3.2.3 Examples

12.3.2.3.1 Infinite loop

- to be avoided
- very easy to create by not modifying the condition in while block
- ***remember `ctrl + c` to bail out***

```
condition = True
while condition:
    # forget to set condition = False
    print("inside infinite loop")
```

12.3.2.3.2 Basic

```
a = 6
while a != 0:
    a -= 1
    if a == 2:
        print(f'break block: {a = }')
        break
    if a == 3:
        print(f'continue block: {a = }')
        continue
    print(f'main block: {a = }')
```

```
>>>  main block: a = 5
>>>  main block: a = 4
>>>  continue block: a = 3
>>>  break block: a = 2
```

12.3.2.3.3 GCD algorithm

Below is gcd algorithm to find the greatest common divisor of 2 integers using while loop.

- given 2 numbers a, b
    1. find remainder of a, b (*modulo operator `%` gives the remainder*)
    2. if remainder is zero then b is the gcd
    3. replace a with b and b with remainder (*in Python this is 1 step using multiple assignment*)
    4. goto to step 1

Since the number of repetitions needed are based on input, while loop is suitable for this.

```
a, b = 9, 6
rem = a % b
```

```
while rem != 0:
    print(f'{a = }, {b = }, {rem = }')
    a, b = b, rem
    rem = a % b

print(f'{a = }, {b = }, {rem = }')
print(b)
```

```
>>>  a = 9, b = 6, rem = 3
>>>  a = 6, b = 3, rem = 0
>>>  3
```

12.3.3 Looping techniques

12.3.3.1 Iterators and Iterables

Iterator is an object that can be iterated upon its elements only **once**. It is **exhaustible**.

Iterable is an object that can be iterated upon its elements repeatedly.

Both cannot be viewed directly and have to be converted into a list or tuple to view contents.

> Caution
>
> Be careful while using iterators since they are **consumable** or **exhaustible**.
> Do not store them in variables for re-use.

```
some_tuple = "a", "b", "c", "d"
some_iterator = enumerate(some_tuple)
```

```
print(list(some_iterator))
```

```
>>>  [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

```
print(list(some_iterator))
```

```
>>>  []
```

Table 12.1: Python functions and methods that return iterators and iterables

Iterators	Iterables
`zip`	`range`
`enumerate`	`dictionary.keys`
`open`	`dictionary.values`
	`dictionary.items`

12.3.3.2 Accessing index of iterables (sequences)

It is very common situation where both index and value is needed while iterating over a sequence (sequence is also an iterable).

Python provides a special function, `enumerate(iterable)` to access index and value of items in an iterable.

```
some_tuple = "a", "b", "c", "d"
list(enumerate(some_tuple))
```

```
>>>  [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

```
for idx, item in enumerate(some_tuple):
    print(f'{idx=}, {item=}')
```

```
>>>  idx=0, item='a'
>>>  idx=1, item='b'
>>>  idx=2, item='c'
>>>  idx=3, item='d'
```

It is possible to have access to index without `enumerate` function, which is lengthier, inconvenient and less efficient, hence not required while using Python.

```
for idx in range(len(some_tuple)):
    print(f'{idx=}, item={some_tuple[idx]}')
```

```
>>>  idx=0, item=a
>>>  idx=1, item=b
>>>  idx=2, item=c
>>>  idx=3, item=d
```

12.3.3.3 Multiple iterables

Python provides a `zip(iterable1, iterable2, ...)` function which returns an iterator with tuples of elements of iterables provided until the shortest iterable is exhausted.

This is helpful when multiple iterables are needed to be iterated through in some connected way.

```
some_list = [1, 2, 3]
some_tuple = (4, 5, 6)
list(zip(some_list, some_tuple))
```

```
>>>  [(1, 4), (2, 5), (3, 6)]
```

```
result = []
for e1, e2 in zip(some_list, some_tuple):
    result.append(e1 + e2)
print(result)
```

```
>>>  [5, 7, 9]
```

This can be achieved without `zip` but again since it is lengthier and inconvenient, it is not recommended while using Python.

```
result = []
for idx in range(len(some_list)):
    result.append(some_tuple[idx] + some_list[idx])
print(result)
```

```
>>>  [5, 7, 9]
```

12.3.3.4 Dictionary

Python provides multiple functions to help iterating over dictionaries.

- keys: `d.keys()`
- values: `d.values()`
- keys, values: `d.items()`

All of these return a view object which is an iterator. Most commonly used is `d.items()` as it provides access to both key and value.

```
some_dict = {"key 1": "value 1", "key 2": "value 2", "key 3": "value 3"}
for k, v in some_dict.items():
    print(f'key = {k}, value = {v}')
```

```
>>>  key = key 1, value = value 1
>>>  key = key 2, value = value 2
>>>  key = key 3, value = value 3
```

12.3.3.5 Mutable collections

It is important to remember that while iterating through mutable collections (`list`, `dict`, `set`) it is recommended not to modify the collection as it gets complicated to handle unexpected situations. So best to use the solutions provided when in doubt rather than introducing a bug.

If the collection is immutable then modifying is not allowed and generally not used.

Another thing to note is **modifying the collection** in this context refers to changing the structure of collection, e.g. deleting an item. It does not refer to modifying the value of an item within the collection.

Solutions to this are, if a mutable collection is to be modified then

- create a new collection
- create a copy while iterating

12.3.3.5.1 Examples with errors

Below are some examples to illustrate when things go wrong when modifying a mutable collection while iterating through it.

12.3.3.5.1.1 List

In the list in example all numbers <= 3 were to be removed, but there is bug due to modifying the list while iterating over it.

Note that 2 is not removed. Figuring out in this case is simple. In the 2nd iteration 1 was already removed from the list and second item was 3 not 2.

This is a simple example hence finding the bug is easy, but as programs get larger it becomes difficult and inefficient to find such bugs.

```
nums = list(range(1, 6))
print(nums)
```

```
>>>  [1, 2, 3, 4, 5]
```

```
for num in nums:
    if num <= 3:
        nums.remove(num)

print(nums)
```

```
>>>  [2, 4, 5]
```

12.3.3.5.1.2 Dictionary

In case of dictionary it gives a run time error.

```
some_dict = {"k1": "v1", "k2": "v2", "k3": "v3"}
for k, v in some_dict.items():
    if k == "k2":
        del some_dict[k]
```

```
>>>  Error: RuntimeError: dictionary changed size during iteration
```

12.3.3.5.2 Solutions

12.3.3.5.2.1 Create a new collection

List

```
nums = list(range(1, 6))
new_nums = []
for num in nums:
    if not num <= 3:
        new_nums.append(num)
```

```
print(nums)
```

```
>>>  [1, 2, 3, 4, 5]
```

```
print(new_nums)
```

```
>>>  [4, 5]
```

Dictionary

```
some_dict = {"k1": "v1", "k2": "v2", "k3": "v3"}
new_dict = {}
for k, v in some_dict.items():
    if not k == "k2":
        new_dict[k] = v
```

```
print(some_dict)
```

```
>>>  {'k1': 'v1', 'k2': 'v2', 'k3': 'v3'}
```

```
print(new_dict)
```

```
>>>  {'k1': 'v1', 'k3': 'v3'}
```

12.3.3.5.2.2 Create a copy

List

```
nums = list(range(1, 6))
print(nums)
```

```
>>>  [1, 2, 3, 4, 5]
```

```
for num in nums.copy():
    if num <= 3:
        nums.remove(num)
```

```
print(nums)
```

```
>>>  [4, 5]
```

Dictionary

```
some_dict = {"k1": "v1", "k2": "v2", "k3": "v3"}
print(some_dict)
```

```
>>>  {'k1': 'v1', 'k2': 'v2', 'k3': 'v3'}
```

```
for k, v in some_dict.copy().items():
    if k == "k2":
        del some_dict[k]
```

```
print(some_dict)
```

```
>>>  {'k1': 'v1', 'k3': 'v3'}
```

12.4 Error handlers

12.4.1 Introduction

12.4.1.1 Errors

- given the number of rules and syntax there are a lot of opportunities for errors
- 2 broad categories
    - **Syntax error**: caught when the code is being parsed
        - parsed means interpreter is reading the code to figure out what is to be done
        - commonly referred to as **compile time error**
    - **Exception**: error detected during execution
        - commonly referred to as **run time error**
- Syntax error

```
a =, 2
```

```
>>>  invalid syntax (<string>, line 1)
```

- Exception

```
# try:
#     1/0
# except Exception as e:
#     print(e)
eg_num = 1/0
```

```
>>>  Error: ZeroDivisionError: division by zero
```

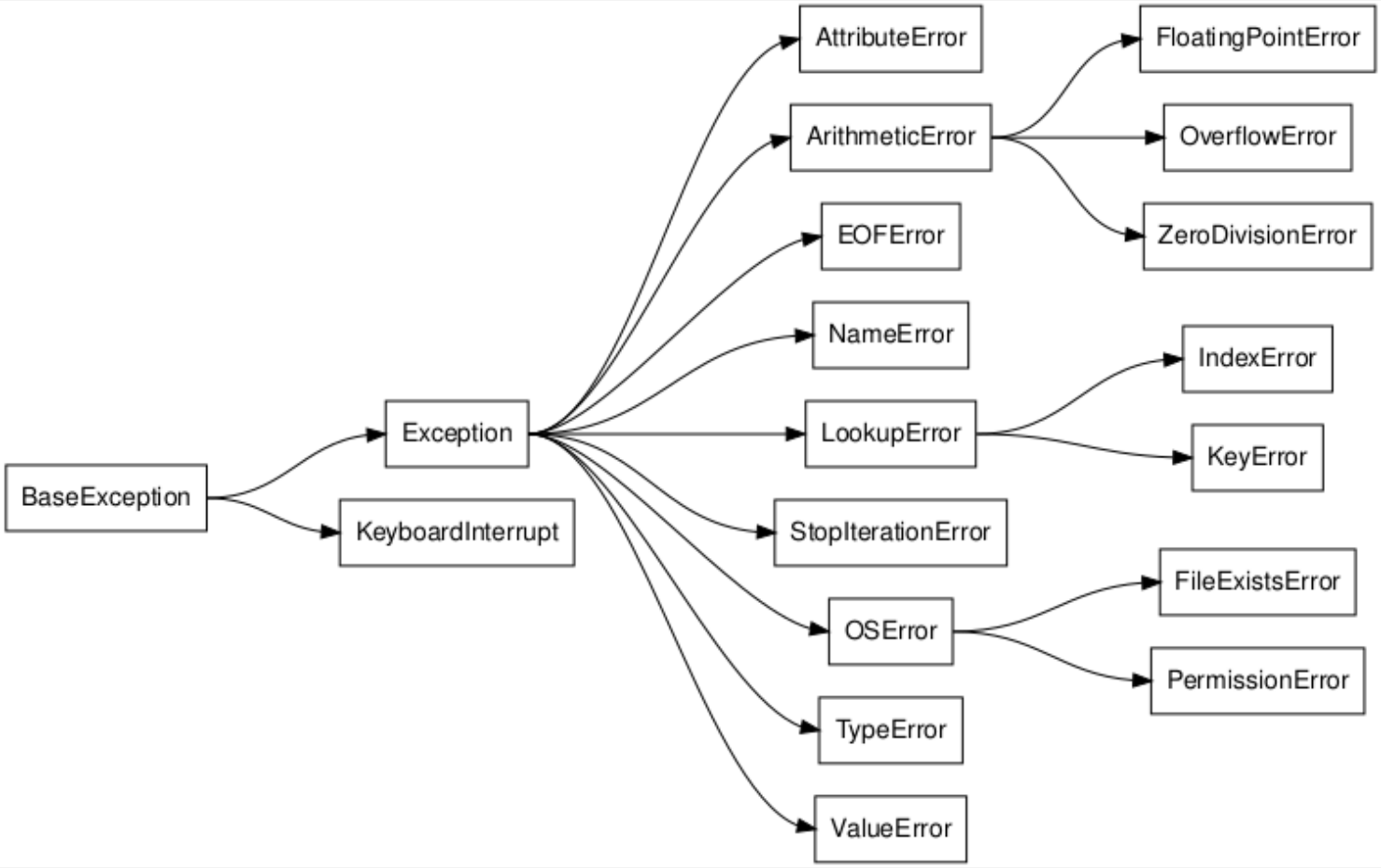
When a run time error occurs, program stops and exception messages are printed along with the traceback. Traceback is a track of where the error occurred in the program to which part of the program lead to running of the line which caused the error.

12.4.1.2 Exceptions

Python has some built-in known exceptions or error types. Exceptions are objects built using object oriented programming.

- Examples
  - `Exception`: base catch almost all errors
  - `ZeroDivisionError`: division by zero
  - `TypeError`: operation not supported by `type`[s] used in operation
- On run time error, an `Exception` type object is created which has information regarding
  - `Exception` type
  - **traceback**

Below figure shows Python's built-in `Exception` hierarchy. More details can be found at Python documentation for built in exceptions.



12.4.1.2.1 `raise`

`raise` statements causes the program to stop with default exception. Optionally a known error can be passed with a custom message, e.g. `raise ValueError("Invalid input")`. `raise` statement can be used anywhere in the program but is suited for use with `try` and `while` blocks, when an expected error is intercepted and has to handled differently.

```
raise ValueError("some custom message")
```

12.4.1.3 Examples

Below are some examples of common errors which are recommended to be tried and experimented in jupyter notebook cells. See the error messages and map them to the Exception hierarchy. Look at the traceback to see how it is structured.

```
1/0
```

```
>>>  Error: ZeroDivisionError: division by zero
```

```
undefined_var_name
```

```
>>>  Error: NameError: name 'undefined_var_name' is not defined
```

```
some_list = [1, 2, 3]
print(some_list[10])
```

```
>>>  Error: IndexError: list index out of range
```

```
float("text")
```

```
>>>  Error: ValueError: could not convert string to float: 'text'
```

```
"abc"*"xyz"
```

```
>>>  Error: TypeError: can't multiply sequence by non-int of type 'str'
```

12.4.2 `try` blocks

`try...except` blocks are used to intercept and handle expected run time errors differently. It provides many options like

- ignore the error and continue running the program
- do some clean up before letting the error stop the program
- run some code irrespective of error or not

Note that exception handling has a large number of specifications which can be combined in a lot of ways which can lead to complexity in the beginning. This is being covered with the objective of simple usage. All the specifications are for completeness and information.

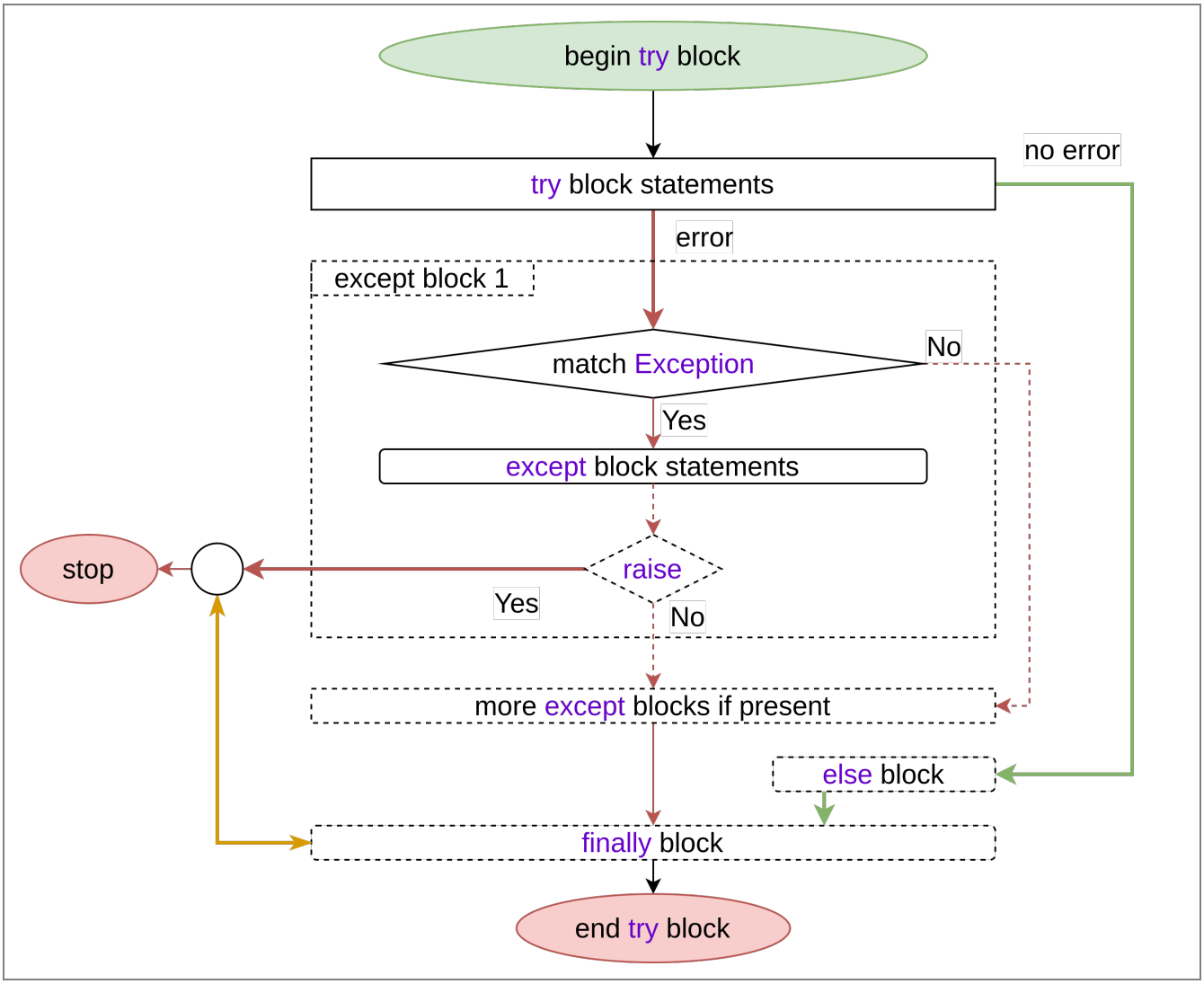
12.4.2.1 Specifications

`try...[except]...[else]...[finally]`

```
try:
    # try code block
```

```
[except Exception as e:
    # except code block
    [raise]]
[else:
    # else code block]
[finally:
    #  finally code block]
```

- **required**
  - there must be at-least one `except` or `finally` block
  - `else` clause (if present) should be after the `except` block
- **optional**
  - there can be multiple `except` clauses
  - `raise` statement and `else` block are optional
- `try` block statements are always evaluated first
- `except` blocks are run when they trap specified **error**
  - if `raise` statement is hit
    - `finally` block is run
    - program is stopped with the `<Exception type>` raised
- `else` block is run in case of **no error**
- `finally` block is run ***always***, error or no error
  - generally used for clean up



12.4.2.1.1 `except` blocks

`except` statement in a `try` block is used to declare an `except` block. They can be used in 2 forms.

- `except <Exception type>`
- `except <Exception type> as <variable name>`

`except` block is run if there is a run time error while executing `try` block and the error matches the `<Exception type>` specified.

When second form is used, `<variable name>` is bound to the `<Exception type>` object.

There are 2 concepts involved.

- which `Exception` is supposed to be handled, e.g. `NameError`, `TypeError`, etc.
- error object

The first concept is of using the exception type. Base `Exception` catches all known errors and is the most generic. In the beginning it should be enough. As the requirements increase there is a need to specify more specific errors. For multiple `except` blocks, it is recommended to have specific exceptions before generic exceptions

The second is the error object. In the beginning it is not of much use as handling the exception should be enough, but as you progress it is this object which helps in dealing different errors differently as it holds a lot of information, like the traceback.

12.4.2.2 Use cases

Try blocks are generally used to **intercept expected errors** and control the outcome if error occurs, use case depends on the context. e.g.

- avoid stopping program on an expected error
    - when program depends on external sources, e.g. db operations, web requests, …
- do something on an expected error and then raise the error
- handle different error types differently

12.4.2.3 Examples

12.4.2.3.1 Ask user input until it is correct

This is an example for nesting different control flow techniques.

The same problem serves as a use case for using recursive functions which is discussed in Section 13.9.1.3.

```
while True:
    try:
        some_int = int(input('Enter an integer...'))
        print(some_int)
        break
    except ValueError:
        continue
```

13 Functions

13.1 Introduction

13.1.1 Terminology

Historically, as the programming languages were evolving, the terminology reflected the state at the time. The terminology has spilled over loosing its context. Below is what was the terminology with intended context.

- **Sub-routine**: any named block of code that can be called (run) using its name
  - **Procedure**: any sub-routine that
    - has side effects
    - does not take any input
    - does not return anything
  - (**Pure**) **Function**: any sub-routine that
    - does not have side effects, does not leave its trace after execution
    - takes inputs and operates only on the inputs
    - returns something

In current popular languages there are just functions which can take any of the forms, function is the only term in language specification.

13.1.2 Background

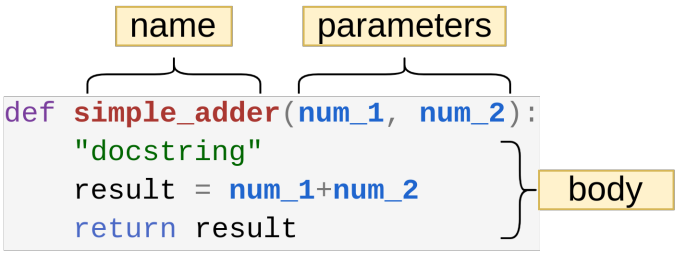
Functions are a major pillar in any programming language, that help to **repeat related tasks with code reuse**.

At more abstract level, functions provide

- **means of combination**
  - build smaller pieces and then join to make a bigger piece
- **means of encapsulation**
  - hide details of implementation during usage
- **means of abstraction**:
  - create blueprints of functionality

A function can be thought of as a black-box which may takes some input and produces some output or performs a task in background. Black boxes can be combined to create new black boxes.

- A simple function has following parts
  - *name*
  - *parameters* (optional)
  - *body* (code)
    - *docstring* (optional, beginning)
    - *return* statement (optional)



13.1.3 Components of a function

13.1.4 Usage

A function can be

- **called** `function_name(parameter_1 = argument_1, ...)`
- **stored** `variable_name = function_name`
- **passed** `function_name_2(variable_name = function_name)`

Functions can be called from any block, e.g. control flow block, body of another function. This allows **means of combination**, to build smaller functionalities and then join them to create a bigger function or program.

13.1.5 Functions are callable

- In simplest form a function call
  - (*optionally*) **receives** arguments (value) for some predefined parameters
  - (*optionally*) **runs** some code to execute certain operation[s]
  - (*optionally*) **returns** some object[s]
  - e.g. `some_func()`, `len(iterable)`, `add(1, 2)`, …
- A function call can have **side effects**
  - it does something in background, e.g. write to a database
  - may or may not return any object

13.1.6 Functions are objects

In Python, like many other high level programming languages, functions are objects, therefore they can be

- *assigned to a variable*
- *stored* in a data structure (such as `list`, `tuple`, `dictionary`, …)
- *passed to a function as an argument*
- *returned from a function*

13.1.7 Different forms of functions

- **Regular functions**
- **Anonymous functions**: **lambda expressions**
- **Partials**: new function from an existing function with partial set of arguments provided
- **Higher order functions**: take function[s] as arguments and optionally return function[s]

13.1.8 Lifetime of functions

This section might not be fully understood before going through namespaces and scopes (Chapter 16) in architecture part of the book, where this is covered in more detail.

- At *compile time* (when the function definition is read)
  - function object is created without evaluation, except
    - objects are created for parameters with default values
    - default values are objects stored in function object
  - code is stored in the function object
  - variables are assigned scopes
- At *run time* (when the function is called)
  - a new local scope is created in the calling environment
  - variable names are looked up in assigned scopes
  - function code is evaluated

13.2 Basic Specifications

13.2.1 First line

- statement starting with keyword `def` declares a function definition
- `def` *must be* followed by
  - **function name**
  - a pair of **parenthesis** ending in `:`
- parenthesis on the first line can *optionally* contain **parameters**
  - parameters can *optionally* be **annotated** with `type`
    - types are *not evaluated*
    - e.g. `def func(a:int, b:str) -> str:`
      - `-> str` indicates the function returns a `str` type object
  - detailed discussion on parameters separately in parameters section
- colon (`:`) must be followed by **function body**
  - if function body is too short it can be included on the same line

```
def function_name(a, b):
    # code block
    return result_object
```

Below are examples of some short functions that can be declared on a single line. Last 2 are also example of using functions as placeholders, functions which are declared but have to be implemented later.

```
def some_short_func(x, y): return x + y
def some_short_func(): pass
def some_short_func(): ""
```

13.2.2 Function body

- *must be* **indented**
- first line can *optionally* be a **doc string**
- *optionally* contain **return**
- *optionally* contain **pass**

```
def function_name(a, b):
    """function description
    parameter 1: description
    returns: description
    """
    # code block
    return result_object
```

13.2.3 Doc strings

Doc strings are used to document functions. They can be multiline strings.

Doc strings are critical for large projects or packages with a lot of code. Editor help pop ups use the doc strings.

sphinx is the most popular python framework for automating documentation.

Even for small projects documentation can be useful for later use. Comments can be used for documentation, but automated doc strings providers help structure the documentation and make the process efficient.

For example, VSCode extension, autoDocstring - Python Docstring Generator can be used to assist in creating docstrings.

Using such tools help use best practices evolved by experience of developers.

13.2.4 `pass` statement

- `pass` statement does not alter control flow
- used as place holder for functions to be implemented
- can be replaced by a doc string

```
def some_func(): pass

def some_func(): ""
```

13.2.5 `return` statement

- syntax: `return [expression_list]`
- `expression_list` can be
    - a single object
    - comma separated objects which are turned to a tuple
- control flow:
    - when a `return` statement is hit anywhere in the code block
        - expression_list, if present, is evaluated and returned
        - `None` returned if there is no expression
        - function call is exited

- **exception**: when `return` is hit from `try..except..else..finally` block
  - `finally` block is run before the function call is exited

13.3 Parameters and arguments

13.3.1 Definitions

- **Parameters**: are input variables in context of defining the function
- **Arguments**: are variables or objects passed to parameters in context of calling the function
- Example
  - `a` and `b` are *parameters* of `some_func`
  - `10` and `x` are *arguments* of `some_func`

```
def some_func(a, b):
  # code block
  pass

x = "a string"

some_func(10, x) # function call
```

13.3.2 Object passing

In Python, the arguments are passed as object references to the parameters. Therefore if the object passed is mutable, changes will be propagated.

```
an_int = 10; a_string = "abc"; a_tuple = (an_int, a_string)
a_list = [*a_tuple]

def a_func(x1, x2, x3, x4):
    x4.append("xyz")
    print('-'*20)
    print('from within function call')
    print(x1 is an_int, x2 is a_string, x3 is a_tuple, x4 is a_list)
    print('-'*20)
    return x1, x2, x3, x4

ret_tuple = a_func(x1=an_int, x2=a_string, x3=a_tuple, x4=a_list)

print(ret_tuple[0] is an_int, ret_tuple[1] is a_string, \
        ret_tuple[2] is a_tuple, ret_tuple[3] is a_list)

print(f'{a_list=}')
```

```
>>>  --------------------
>>>  from within function call
>>>  True True True True
>>>  --------------------
>>>  True True True True
```

```
>>>  a_list=[10, 'abc', 'xyz']
```

13.3.3 Argument types

Based on how parameters are defined arguments can be passed in different ways

- **positional arguments**: are passed to parameters using position during a function call
- **keyword arguments**: are passed to parameters using keyword (parameter name) during a function call, also called **named parameters**
- **optional/default arguments**: any parameter with default value specified makes the argument optional
    - with positional arguments order has to be kept in mind

13.3.3.1 Examples

```
def some_func(a, b):
    pass
```

```
some_func(10, 20) # passed as positional
some_func(a = 10, b = 20) # passed as keyword
```

```
def some_func(a, b=20): # b is optional
    print(f'{a = }, {b = }')
```

```
some_func(10)
```

```
>>>  a = 10, b = 20
```

```
some_func(10, 30)
```

```
>>>  a = 10, b = 30
```

```
some_func(a = 10)
```

```
>>>  a = 10, b = 20
```

```
some_func(b = 30, a = 10)
```

```
>>>  a = 10, b = 30
```

13.3.4 Specifications

- General structures for defining parameters (`/`, `*`, `**`)
  - `function_name(<pos or kw>)`
  - `function_name(<pos>, /, <po or kw>, *, <kw>)`
  - `function_name(<pos>, /, <po or kw>, *args, <kw>, **kwargs)`
- **regular arguments**: by default all arguments can be passed as positional or keyword subject to
  - **positional** arguments **must** come **before keyword arguments**
  - once a keyword argument is given all remaining arguments are keyword
- after one default argument, remaining must be default
  - except for keyword only arguments
- **separation**
  - `/` is used to separate positional only arguments
  - `*` is used to separate keyword arguments
- **collection** (**variadic arguments**)
  - `*args` collects all available positional arguments as a **tuple**
    - has to be defined after other positional arguments if present
    - `args` is just convention, can be name of choice, but recommended
  - `**kwargs` can collect all available keyword arguments as a **dictionary**
    - has to be defined at the end
    - `kwargs` is just convention, can be name of choice, but recommended
- `*` and `*args` mark the beginning of keyword arguments, hence cannot precede `/`

13.3.5 Use cases

- as a user of packages
  - regular arguments are needed mostly
  - while using external packages familiarity with the specifications will help
- **positional only** parameters are used when
  - parameter names have no meaning
  - reliance on keyword for passing arguments has to be avoided
  - e.g. `print` function: multiple objects can be passed before any kw arg
- **keyword only parameters** are used when
  - names have special meaning
  - reliance on positional arguments has to be avoided
  - e.g. `print` function: `end`, `sep` etc. have to be passed after all positional variadic args as kw

13.3.6 Examples

Regular arguments: by default all arguments can be passed as positional or keyword subject to

- positional arguments **must** come before keyword arguments
  once a keyword argument is given all remaining arguments **must** be keyword

```
def some_func(a, b, c):
    pass
```

Allowed

- `some_func(10, 20, 30)`
- `some_func(10, 20, c = 30)`
- `some_func(10, b = 20, c = 30)`

Not allowed

- `some_func(a = 10, 20, 30)`
- `some_func(10, b = 20, 30)`
- `some_func(a = 10, b = 20, 30)`

13.3.6.1 Separation

- `/` is used to separate positional only arguments
- `*` is used to separate keyword arguments

```
def some_func(pos_1, pos_2, /, pos_or_kw_1, *, kw_1, kw_2):
    print(f"{pos_1=}, {pos_2=}, {pos_or_kw_1=}, {kw_1=}, {kw_2=}")
```

```
some_func(1, 2, 3, kw_1=4, kw_2=5)
```

```
>>>  pos_1=1, pos_2=2, pos_or_kw_1=3, kw_1=4, kw_2=5
```

```
some_func(1, 2, pos_or_kw_1=3, kw_1=4, kw_2=5)
```

```
>>>  pos_1=1, pos_2=2, pos_or_kw_1=3, kw_1=4, kw_2=5
```

```
some_func(1, pos_2=2, pos_or_kw_1=3, kw_1=4, kw_2=5)
```

```
>>>  Error: TypeError: some_func() got some positional-only arguments passed as keyword arguments: 'po
```

13.3.6.2 Variadic arguments

- **collection** (**variadic arguments**)
  - `*args` collects all available positional arguments as a tuple
    - has to be defined after other positional arguments if present
    - `args` is just convention, can be name of choice, but recommended
  - `**kwargs` can collect all available keyword arguments as a dictionary
    - has to be defined at the end
    - `kwargs` is just convention, can be name of choice, but recommended

```
def some_func(*args, **kwargs):
    print(f'{args}')
    print(f'{kwargs}')
```

```
some_func(1, 2, 3)
```

```
>>>  (1, 2, 3)
>>>  {}
```

```
some_func(1, 2, 3, a = 4, b = 5)
```

```
>>>  (1, 2, 3)
>>>  {'a': 4, 'b': 5}
```

- `args` is just convention, can be name of choice, but recommended
- `kwargs` is just convention, can be name of choice, but recommended

```
def some_func(*args_tuple, **kw_dict):
    print(f'{args_tuple}')
    print(f'{kw_dict}')
```

```
some_func(1, 2, 3)
```

```
>>>  (1, 2, 3)
>>>  {}
```

```
some_func(1, 2, 3, a = 4, b = 5)
```

```
>>>  (1, 2, 3)
>>>  {'a': 4, 'b': 5}
```

13.3.6.3 More Examples

```
def func(a, b, *args)
```

- 2 regular arguments
- arbitrary remaining positional arguments collected in args tuple
- if any regular argument passed as keyword, args cannot be used

```
def func(a, b, *args, kw1, kw2=100)
```

- 2 regular arguments
- arbitrary remaining positional arguments collected in args tuple
- 2 keyword arguments with one optional
- if any regular argument passed as keyword args cannot be used

```
def func(a, b=10, /, reg1, reg2=100)
```

- this will not work
- b is positional only + default and there is non default reg1 after it

```
def func(a, b=10, *, kw1, kw2=100, **kwargs)
```

- this will work
- order of default parameters is not important for keyword only arguments

```
def func(*args, **kwargs)
```

- collects arbitrary number of positional arguments
- collects arbitrary number of keyword arguments

13.4 Examples

13.4.1 Check primes

Prime is a positive integer greater than 1 which is divisible only by 1 and itself. E.g. 2, 3, 5, ...

Given a positive integer check if it is a prime. Print a message confirming the result.

```
def check_prime(num: int) -> None:
    """
    check if a number is prime and print the result
    num: positive integer
    returns: None
    """
    if not isinstance(num, int):
        print(f'{num} is not an integer')
        return
    if num <= 1:
        print(f'{num} is not a positive integer greater than 1')
        return
    divisors = []
    for i in range(1, num + 1):
        if num % i == 0: divisors.append(i)
    if divisors == [1, num]:
        print(f'{num} is a prime')
    else:
        print(f'{num} is not a prime')
```

```
check_prime(10)
```

```
>>>  10 is not a prime
```

```
check_prime(23)
```

```
>>>  23 is a prime
```

```
check_prime(1)
```

```
>>>  1 is not a positive integer greater than 1
```

```
check_prime(-2)
```

```
>>>  -2 is not a positive integer greater than 1
```

13.4.2 GCD

Implement the gcd algorithm using a function that takes 2 numbers as input and returns the greatest common divisor.

Below is gcd algorithm to find the greatest common divisor of 2 integers.

- given 2 numbers a, b
    1. find remainder of a, b (*modulo operator % gives the remainder*)
    2. if remainder is zero then b is the gcd
    3. replace a with b and b with remainder (*in python this is 1 step using multiple assignment*)
    4. goto to step 1

Below is what was implemented using while loop in control flow chapter. Note that every time gcd has to be calculated inputs have to be changed and the whole code has to be run.

```
a, b = 9, 6
rem = a % b
while rem != 0:
    print(f'{a = }, {b = }, {rem = }')
    a, b = b, rem
    rem = a % b

print(f'{a = }, {b = }, {rem = }')
print(b)
```

```
>>>  a = 9, b = 6, rem = 3
>>>  a = 6, b = 3, rem = 0
>>>  3
```

```
def calc_gcd(num_1: int, num_2: int) -> int:
    """
    Calculate and return the greatest common divisor of 2 integers
    """
    rem = num_1 % num_2
    while rem != 0:
        num_1, num_2 = num_2, rem
        rem = num_1 % num_2
    return num_2
```

Having defined the function, it can be called multiple times with different values without worrying about the implementation. This serves as an example of **means of encapsulation**.

```
calc_gcd(3, 9)
```

```
>>>  3
```

```
calc_gcd(12, 45)
```

```
>>>  3
```

13.5 Caveat for default values

When using mutable data types (like list, dictionary or set) as default values for function parameters, the behavior has to be looked out for. Default values are typically used for immutable data types like numbers, strings or bool.

It is important to note that default values are created once in memory, when the `def` statement is executed for creating the function object, i.e. at compile time. Not during a function call.

Check the result of below function calls. Trying out python tutor will help understand this more clearly.

```
def some_func(num, some_list=[]):
    some_list.append(num)
    return some_list

print(some_func(1))
print(some_func(2))
```

```
>>>  [1]
>>>  [1, 2]
```

Note that during second call same list was used. There are situations when this default behavior has to be avoided. Below is an approach to get around this.

```
def some_func(num, some_list=None):
    if some_list == None: some_list = []
    some_list.append(num)
    return some_list

print(some_func(1))
print(some_func(2))
```

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

13.6 `lambda` expressions

Lambda expressions are **anonymous** and **short** functions, typically used to create very short functions to be passed around or for cleaner syntax.

Generic name is anonymous functions. Most languages provide a mechanism to create and pass anonymous functions. In Python, they are called lambda expressions.

Syntax

- `lambda` keyword is used to create lambda expressions
- *limited to a single expression*
- `return` keyword is not required, expression is returned
- example: `lambda x, y: x * y`

```
f = lambda x, y: x * y
print(f)
```

```
>>>  <function <lambda> at 0x778d9696aa70>
```

```
print(f(2, 3))
```

```
>>>  6
```

13.7 Partials

Partials are functions created from other functions by passing a subset of required arguments.

`functools` module in standard library provides a higher order function `partial` to create partials.

13.7.1 Use cases

- Partials are often used in functional programming where functions are passed around as arguments. e.g. functionals like map, filter and reduce take functions as argument where partials are needed. They are discussed in Section 13.8.1.
- Another use is when an existing function has to be used multiple times with certain set of arguments specified.

As an example in an application if print function is needed to be used multiple times with argument `sep="\n"` then a partial can be created using original `print` function.

```
import functools as ft
custom_print = ft.partial(print, sep="\n")
some_list = [1, 2, 3]
custom_print(*some_list)
```

```
>>>  1
>>>  2
>>>  3
```

13.8 Higher order functions

Higher order functions evolved as part of functional programming paradigm where functions are treated as objects.

- Higher order functions are functions that
  - take function[s] as input
  - optionally return function[s]
- Higher order functions along with rules of scoping are used to create different types of functions
- Design patterns created using higher order functions
  - **Map-Reduce**: apply some function to elements of a collection
  - **Factory** functions: create new functions based on some input argument

    - **Decorator** functions: add some standard functionality to a function

- Python standard library has some modules/packages to help with these

    - `functools`: Higher-order functions and operations on callable objects
    - `operators`: Standard operators as functions

13.8.1 Map Reduce

Map-Reduce is a design pattern to work with collections. These are newer features in high level languages like Python. They provide a better and cleaner alternative to iterative solutions using loops for working with collections.

Filter is a special case of map. Collectively these are also referred to as functionals.

Functionals are a good example of design patterns to improve desired properties of program. All the 3 forms improve code readability as the syntax is concise and it is easier to spot what is being done by isolating it from iteration.

For example, map is a generic concept which improves

- Readability
    - iteration is isolated from what operation is being done, hence better readability
- Modularity
    - map is responsible for iteration
    - function passed is responsible for operation to be applied to each element
- Extensibility
    - function passed can be anything, so different operations can be applied by defining new functions without impacting the core functionality of iterating and applying the function.
- Testability
    - it is easier to test and debug as the structure is modular
- Efficiency
    - map is faster than loop but slower than comprehensions

13.8.1.1 Map

Map is a generic concept of applying (mapping) a function to all elements of a collection or multiple collections in parallel.

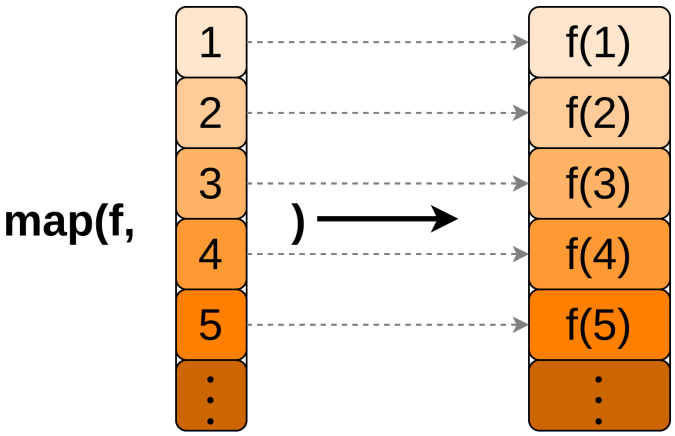
Map is a better alternative to `for` loops as code readability is improved as what is being done is isolated from iteration.

Syntax

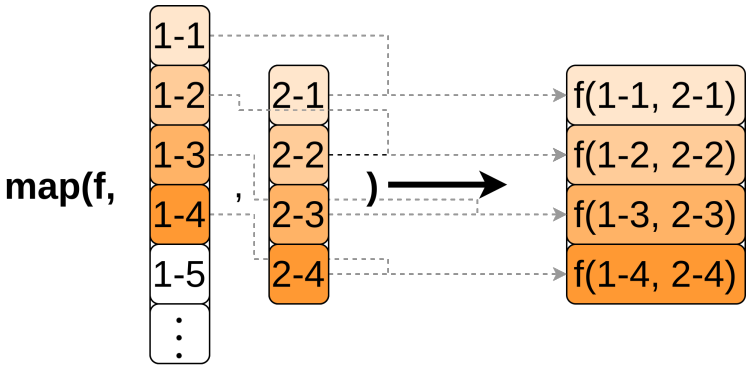
- `map(function, iterable[, iterables])` is the Python implementation
    - returns an *iterator* (consumable, can be used once)
- With single iterable: function applied should take 1 argument which will be elements of the iterable
- With multiple iterables

- function applied should take as many arguments as iterables
- `map` stops at iterable of shortest length, if lengths are different

---



(a) single iterable



(b) multiple iterables

13.8.1.1.1 Example: Single iterable

To find the square of all numbers in a list (container) below are different solutions. Note that underlying operation is simple, apply some operation to elements of a collection. This is a very common situation encountered while programming.

```
some_list = list(range(5))
```

13.8.1.1.1.1 Loop

```
sqrd_list = []
for x in some_list:
    sqrd_list.append(x**2)
```

```
>>>  sqrd_list = [0, 1, 4, 9, 16]
```

13.8.1.1.1.2 Map

```
def sqr(x):
    return x**2

sqrd_itr = map(sqr, some_list)
```

```
>>>  list(sqrd_itr) = [0, 1, 4, 9, 16]
```

13.8.1.1.1.3 Map & lambda

```
sqrd_itr = map(lambda x: x**2, some_list)
```

```
>>>  list(sqrd_itr) = [0, 1, 4, 9, 16]
```

13.8.1.1.2 Example: Multiple iterables

- add numbers in 2 tuples

Note that lengths of iterables is different in example so result is accordingly of length of shortest iterable.

If function to map is complex, define a regular function and pass the name to `map`.

```
tuple_1 = 1, 2, 3
tuple_2 = 4, 5
```

```
tuple(map(lambda x, y: x + y, tuple_1, tuple_2))
```

```
>>>  (5, 7)
```

```
tuple(map(lambda x, y: x + y, tuple_2, tuple_1))
```

```
>>>  (5, 7)
```

13.8.1.2 Filter

Filter is a generic concept of filtering values from a collection using certain conditions. Note that it is a special case of map.

Syntax

- `filter(function, iterable)` is provided in Python
  - returns an iterator (consumable, can be used only once)
  - function should return true or false when acting on an element
  - if function is `None` then all truthy elements are returned

13.8.1.2.1 Example: `None`

```
some_list = [1, 0, None, '', 'abc', tuple()]
```

```
list(filter(None, some_list))
```

```
>>>  [1, 'abc']
```

Below is the same task done using iterative solution.

```
filtered_list = []
for item in some_list:
    if item: filtered_list.append(item)
filtered_list
```

```
>>>  [1, 'abc']
```

13.8.1.2.2 Example

- filter positive integers from a list

```
some_list = [-2, -1, 0, 1, 2]
```

```
[*filter(lambda x: x > 0, some_list)]
```

```
>>>  [1, 2]
```

Below is the same task implemented using iterative solution.

```
filtered_list = []
for item in some_list:
    if item > 0: filtered_list.append(item)
```

```
filtered_list
```

```
>>>  [1, 2]
```

13.8.1.3 Reduce

Reduce is a generic concept of aggregating elements of a collection into single result.

The actual underlying operation is to apply a function (operation) to 2 items at a time recursively.

Let a collection of n elements be $collection = [e_0, e_1, e_2, e_3, e_4, \cdots, e_{n-1}]$. What reduce does is

- result of step 1: $r_1 = f(e_0, e_1)$
- result of step 2: $r_2 = f(r_1, e_2)$
- result of step 3: $r_3 = f(r_2, e_3)$
- ...
- result of step n - 1: $r_{n-1} = f(r_{n-2}, e_{n-1})$

Optionally an initial value can be given which is used as the base case, step 1 uses this value and the first element. This is also used in case the collection has 0 or 1 element.

Note that there will be an error if the collection is empty and no initializer is specified.

For example, sum of some numbers is applying reduce and cumulative sum of some numbers is the intermediate result of reduce.

Reduce is less often used in comparison to map and filter most often used to create aggregate tables and data.

Python standard library has tools to apply reduce and accumulate.

- `functools.reduce(function, iterable[, initializer])`
- `itertools.accumulate(iterable[, func, *, initial=None])`

13.8.1.3.1 Example

To find the sum of numbers in a list using iterative solution and reduce.

```
some_list = list(range(10))
print(some_list)
```

```
>>>  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
sum_itr = 0
for num in some_list:
    sum_itr += num
```

```
sum_itr
```

```
>>>  45
```

```
import functools as ft
import operator as op

sum_red = ft.reduce(op.add, some_list, 0)
```

```
sum_red
```

```
>>>  45
```

Instead of defining `add_func = lambda x, y: x + y`, `operator` module was used.

13.9 Recursive functions

- Recursion is a generic concept of repeating a smaller well define task to get to a solution using base case[s]
  - e.g. gcd algorithm
- Recursion can be implemented using iterative solution or recursive functions
  - recursive functions are not most efficient but better in terms of
    - code readability
    - maintainability
- Recursive functions call themselves from within themselves to terminate when the base case[s] is reached
  - base case[s] need to be defined carefully to **avoid infinite recursive calls**
- Do not use recursive functions unless it is unavoidable
  - merge sort is an example of recursive algorithm that is efficient

To understand how recursive functions create nested scopes, Python tutor is a good tool. It helps visualize how nested local scopes are created and destroyed at run time during recursive function calls.

Recursive functions are used in algorithms, specially when normal approaches like iteration become infeasible. Therefore in the beginning there is not much point to spend a lot of time on this, but it is good to understand the concept as it might come handy in some situations. An example would be when there is an expected error that you want to handle and retry.

13.9.1 Examples

13.9.1.1 Factorial

$$
\begin{aligned}
n! &= n * (n-1) * (n-2) * \cdots * 1 \\
&= n * (n-1)! \\
0! &= 1
\end{aligned}
$$

```
def fact_iter(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
def fact_rec(n):
    if n == 0:
        return 1
    else:
        return n*fact_rec(n-1)
```

13.9.1.1.1 Iterative Solution

13.9.1.1.2 Recursive solution

Note that recursive solution can be cleaned further using the ternary operator and lambda expression.

```
fact_rec = lambda n: 1 if n == 0 else n*fact_rec(n-1)
```

13.9.1.2 GCD Algorithm

```
a, b = 9, 6
rem = a % b
while rem != 0:
    print(f'{a=}, {b=}, {rem=}')
    a, b = b, rem
    rem = a % b

print(f'{a=}, {b=}, {rem=}')
print(b)
```

- Given 2 numbers a, b
    1. find remainder of a, b (*modulo operator* **%** *gives the remainder*)
    2. if remainder is zero then b is the gcd
    3. replace a with b and b with remainder (*in Python this is 1 step*)
    4. goto to step 1
- Earlier this was solved using `while`
- Implement this using
    - regular function
    - recursive function

```
>>>  a=9, b=6, rem=3
>>>  a=6, b=3, rem=0
>>>  3
```

13.9.1.2.1 Regular function

```
def find_gcd(a: int, b: int) -> int:
    """
    find the greatest common divisor of 2 integers a and b
    """
    rem = a % b
    while rem != 0:
        a, b = b, rem
        rem = a % b
    return b
```

```
find_gcd(35, 28)
```

```
>>>  7
```

13.9.1.2.2 Recursive function

```
def find_gcd_rec(a: int, b: int) -> int:
    """
    find greatest common divisor of 2 integers a and b using recursion
    """
    rem = a % b
    if rem == 0:
        result = b
    else:
        result = find_gcd_rec(b, rem)
    return result
```

```
find_gcd_rec(35, 28)
```

```
>>>  7
```

Note that above code can be further cleaned as below by simply reorganizing the placement of `return` statement as only one option is to returned and a function ends if it hits the `return` statement.

```
def find_gcd_rec(a: int, b: int) -> int:
    """
    find greatest common divisor of 2 integers a and b using recursion
    """
    rem = a % b
    if rem == 0:
        return b
    else:
        return find_gcd_rec(b, rem)
```

This can be further cleaned by using ternary operator, `X if condition else Y`.

```
def find_gcd_rec(a: int, b: int) -> int:
    """
    find greatest common divisor of 2 integers a and b using recursion
```

```
    """
    rem = a % b
    return b if rem == 0 else find_gcd_rec(b, rem)
```

13.9.1.3 Handling exceptions

In the control flow section, the example, Section 12.4.2.3.1, was to ask user input until it is correct. The solution was implemented using `while` and `try` blocks.

The same thing can be achieved using a recursive function with max number of tries allowed.

```
def get_an_integer(max_count=5, count=1):
    """
    Ask user to input an integer until integer is provided
    or maximum number of trials expire
    """
    try:
        some_int = int(input('Enter an integer...'))
        return some_int
    except ValueError:
        print(f'This was try number {count}, and was not an integer.')
        if count < max_count:
            count += 1
            get_an_integer(count=count)
        else:
            print(f"maximum tries ({max_count}) reached")
```

14 OOP

14.1 Introduction

Object oriented programming, commonly referred to as **OOP**, is fundamental to how objects are defined and used in a programming language.

There are 2 components of OOP

- **object definition**: aka **class** definition, **type** definition
- **object instance[s]**: creation of the object[s] of a certain class (type)

For example `int` is a class (type) and there can be multiple instances like `1`, `2` etc. Note that `int` is a class and its definition (code) to create `int` type objects is stored once on RAM. All instances, on creation are stored on RAM separately.

In Python, `type` is the root class of all classes, that is the reason `type` and class are used interchangeably. `type` can also be used as a function to check the class of an object.

```
type(10), type(int), type(type)
```

```
>>>  (<class 'int'>, <class 'type'>, <class 'type'>)
```

Class is the blueprint of a certain type of object, what attributes all instance objects of this type will share

- **state**: data attributes define the state of an instance
- **operations**: callable attributes (**methods**)

Functions defined in a class are called methods. Methods are regular functions with some differences to provide features related to classes and instances.

Class instance object is an instance object of certain class (`type`) created and stored on RAM

- multiple instances can be created
- usually referred to as objects

14.1.1 Use cases

At a more abstract level, OOP provide

- **means of combination**
  - combine data and operations
  - build smaller pieces and then join to make a bigger piece
- **means of encapsulation**
  - hide details of implementation during usage
- **means of abstraction**:

- create blueprints of functionality
- although functions can be used, OOP is better for this

OOP is generally used by developers for building tools and packages. Almost everything that can be implemented using OOP, can be implemented using functional programming. Functional programming keeps the complexity low. As a user it is recommended to know basics of OOP to use solutions provided by developers efficiently and actually use OOP only if necessary after getting some experience.

14.1.2 Namespaces

Class and instances have their own namespaces. Instance object namespace is searched first then class namespace is searched by the interpreter for variables.

14.1.3 Attributes of an object

14.1.3.1 Data attributes

- class level
  - bare
  - properties: static or calculated
- instance level
  - bare
  - properties: static or calculated

14.1.3.2 Methods

- instance methods
- class methods
- static methods

14.1.3.3 Data attributes

Class data attributes are shared across all instance objects. They are suited for data attributes that do not define the state of an object. Main advantage is that they can be configured centrally. If the value is changed all instance objects have access to the new value.

Instance data attributes define the state of an instance object. If the value is changed in one of the instances, it does not impact the value of the attribute in other instance objects.

Bare data attributes are attributes that can be accessed and modified directly without any pre or post functionality.

Properties hide bare data attributes behind a property name, which if referenced from any instance object, act upon the bare attribute. Properties are also used to add some pre and post code while accessing or modifying the bare data attribute.

Static properties do not calculate the values of the underlying bare data attribute.

Dynamic properties do calculate the values of the underlying bare data attribute based on some other values.

14.1.3.4 Methods

Methods are callable attributes. Methods are simply functions defined in a class and they behave with minor difference compared to regular functions, to provide some basic functionality related to classes and objects.

By **default** methods defined inside a class are **instance methods**. They can be declared to be class or static using decorator syntax.

Instance methods are callable attributes intended to be called from instance objects. They are functions with access to the instance object itself. The first parameter is enforced to be the instance object. They cannot be called from the class directly.

Class methods are callable attributes that are accessible from both class and all instance objects, with access to class attributes. The first parameter is enforced to be the class object.

Static methods are callable attributes that are accessible from both class and all instance objects. They are regular functions without any enforced parameter.

14.2 Basics

14.2.1 Create a class and access attributes

- it is a convention, in Python, to use CamelCase for class names

```
class BasicClass:
    data_attr = "bare class attribute"
```

- attributes are accessed using dot notation (.)

```
BasicClass.data_attr
```

```
>>>  'bare class attribute'
```

14.2.2 Create instance and access attributes

- create instances by calling the class

```
obj_1 = BasicClass()
obj_2 = BasicClass()
```

- access attributes using dot notation (.)

```
obj_1.data_attr
```

```
>>>  'bare class attribute'
```

```
obj_2.data_attr
```

```
>>>  'bare class attribute'
```

- check type of any instance object

```
type(obj_1), type(obj_2)
```

```
>>>  (<class '__main__.BasicClass'>, <class '__main__.BasicClass'>)
```

- class attributes are shared by instance objects and are useful for data attributes to be changed centrally

```
BasicClass.data_attr = "new value"
obj_1.data_attr, obj_2.data_attr
```

```
>>>  ('new value', 'new value')
```

14.3 Instance methods

14.3.1 Incorrect example

- define the class

```
class CustomClass:
  def custom_method():
    print("running custom_method")
```

- call the method from class directly
  - **this will work**

```
CustomClass.custom_method()
```

```
>>>  running custom_method
```

- create an instance object and call the method from instance object
  - **this will not work**

```
obj_1 = CustomClass()
obj_1.custom_method()
```

```
>>>  Error: TypeError: CustomClass.custom_method() takes 0 positional arguments but 1 was given
```

14.3.2 `self` argument

- Error reason
  - methods by default are bound to instance object
  - first parameter is passed by Python as the instance object itself
- It is a convention to call this first parameter `self`
  - It can be named anything
  - it is recommended to use `self` for consistency
- Instance methods are not meant to be called directly from class

```
class CustomClass:
  def custom_method(self):
    print("running custom_method")
    print(self)
```

14.3.2.1 Calling from class

- Call instance method from class directly: **this will not work**
- This is because there is no instance object to pass as first argument
- How class and static methods are defined will be covered later

```
CustomClass.custom_method()
```

```
>>>  Error: TypeError: CustomClass.custom_method() missing 1 required positional argument: 'self'
```

14.3.2.2 Calling from instance

```
class CustomClass:
  def custom_method(self):
    print("running custom_method")
    print(self)
```

- Create an instance and call the method from instance directly: this will work

```
obj_1 = CustomClass()
obj_1.custom_method()
```

```
>>>  running custom_method
>>>  <__main__.CustomClass object at 0x749c41356650>
```

14.4 Instance data attributes

14.4.1 `__init__`

- `__init__` is an instance method called at the time of instance object creation
  - this is by Python design
  - it is optional
- Used to create instance attributes at the time of instance object creation

```
class CustomClass:
    def __init__(self, bare_data_attr_1_val, bare_data_attr_2_val):
        self.bare_data_attr_1 = bare_data_attr_1_val
        self.bare_data_attr_2 = bare_data_attr_2_val

    def custom_method(self):
        print('running custom_method')
        print(f'with access to {self.bare_data_attr_1 = }')
        print(f'and {self.bare_data_attr_2 = }')
```

14.4.1.1 Create instances

```
obj_1 = CustomClass(2, 4)
obj_1.bare_data_attr_1, obj_1.bare_data_attr_2
```

```
>>>  (2, 4)
```

```
obj_1.custom_method()
```

```
>>>  running custom_method
>>>  with access to self.bare_data_attr_1 = 2
>>>  and self.bare_data_attr_2 = 4
```

```
getattr(obj_1, "bare_data_attr_1")
```

```
>>>  2
```

```
setattr(obj_1, "bare_data_attr_1", "abc")
getattr(obj_1, "bare_data_attr_1")
```

```
>>>  'abc'
```

14.5 Instance properties

- In other languages there is a concept of making certain attributes `private`
  - in Python properties are used for this
  - attribute is still accessible but direct access is discouraged
- To define a property manually use
  - `property(fget, fset, fdel, doc)`
  - `fget`, `fset` and `fdel` are instance methods to get, set and delete property value
- `del` reserved word is used to delete attributes
- It is convention to name the underlying attribute name to be `_<property_name>`
  - this is to indicate that attribute is for internal use
- Using methods to access, modify and delete attributes helps with
  - adding checks and other functionality as needed
  - hiding attribute name behind property name
    - change in attribute name does not break other code using the class

14.5.1 Example

Below example illustrates all aspects of defining a property in a class. Note the difference in names to illustrate different components of the property.

It is recommended to experiment in jupyter notebook by creating instance objects and accessing/modifying/deleting property and bare instance attribute.

```
class CustomClass:
    """This is CustomClass with a bare data attribute and a property"""
    def __init__(self, property_1_val, bare_data_attr_1_val):
        self.property_1_name = property_1_val
        self.bare_data_atrr_1_name = bare_data_attr_1_val

    def get_property_1_name(self):
        print("getter called..")
        return self._property_1_name

    def set_property_1_name(self, property_1_val):
        print("setter called..")
        # required checks or calculations
        self._property_1_name = property_1_val

    def del_property_1_name(self):
        print("delete method called..")
        # required checks or calculations
        del self._property_1_name

    property_1_name = property(fget=get_property_1_name,
        fset=set_property_1_name, fdel=del_property_1_name,
        doc="""The property's description.""")
```

14.5.2 Defining property using decorator syntax

Below is the same example with Python decorator syntax. It is useful for cleaner syntax which is much easier to read.

```
class CustomClass:
    """This is CustomClass with a bare data attribute and a property"""
    def __init__(self, property_1_val, bare_data_attr_1_val):
        self.property_1_name = property_1_val
        self.bare_data_atrr_1_name = bare_data_attr_1_val

    @property
    def property_1_name(self):
        """Property description"""
        print("getter called..")
        return self._property_1_name

    @property_1_name.setter
    def property_1_name(self, property_1_val):
        print("setter called..")
        # required checks or calculations
```

```
        self._property_1_name = property_1_val

    @property_1_name.deleter
    def property_1_name(self):
        print("delete method called..")
        # required checks or calculations
        del self._property_1_name
```

14.6 Class methods

- Class methods can be defined using `@classmethod` decorator
- Like instance methods, first parameter is mandatory and gives access to class
    - it is bound to the class
    - it is convention to name this first parameter as `cls`
    - all instances created have access as well

14.6.0.1 Example

```
class CustomClass:
    x = 10
    @classmethod
    def some_class_method(cls):
        print(f"This is a class method bound to class - {cls}")
        print(f"has access to class attributes {cls.x = }")

CustomClass.some_class_method()
```

```
>>>  This is a class method bound to class - <class '__main__.CustomClass'>
>>>  has access to class attributes cls.x = 10
```

```
obj = CustomClass()
obj.some_class_method()
```

```
>>>  This is a class method bound to class - <class '__main__.CustomClass'>
>>>  has access to class attributes cls.x = 10
```

14.7 Static methods

- Static methods are regular functions defined in a class
- Defined using `@staticmethod` decorator
- Can be accessed from class and all instances
- There is no mandatory first parameter

14.7.0.1 Example

```
class CustomClass:
  @staticmethod
  def some_static_method(a=10):
    print("This is a static function")
    print(f"a regular function with parameters, e.g. {a = }")

CustomClass.some_static_method()
```

```
>>>  This is a static function
>>>  a regular function with parameters, e.g. a = 10
```

```
obj = CustomClass()
obj.some_static_method(a = 100)
```

```
>>>  This is a static function
>>>  a regular function with parameters, e.g. a = 100
```

14.8 Full example

Below is a full example to illustrate all the pieces together. There is a `Circle` class to create circle objects with the following components

- `pi` is a class data attribute
- `_radius` is instance data attribute marked private
    - this is only available in instances
    - not recommended for direct use
- `radius` is a static property to access `_radius` from instances
- `area` is a dynamic property with no set method
    - provides access to attribute `_area`
    - automatically updated when radius changes
- `circumference` is an instance method

```
class Circle:
    def __init__(self, radius_val):
        self.radius = radius_val

    pi = 3.141592653589793

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, radius_val):
        print("setting radius and calculating area")
        self._radius = radius_val
```

```
        self._area = self.pi * (self.radius**2)

    @property
    def area(self):
        return self._area

    def calc_circumference(self):
        print('calculating circumference')
        return 2*self.pi*self.radius
```

```
c1 = Circle(1)
```

```
>>>  setting radius and calculating area
```

```
c1.radius, c1.area, c1.calc_circumference()
```

```
>>>  calculating circumference
>>>  (1, 3.141592653589793, 6.283185307179586)
```

```
c1.area, c1.calc_circumference()
```

```
>>>  calculating circumference
>>>  (3.141592653589793, 6.283185307179586)
```

```
c1.radius = 2
```

```
>>>  setting radius and calculating area
```

14.9 Summary

- `class` defines classes which are callable and used to create instance objects
- `__init__` method is used to control creation of attributes at instance object creation
- Access attributes
  - `<obj_name>.<attr_name>[()]`
  - `getattr(<obj_name>, <attr_name>)`
- Modify attributes
  - `<obj_name>.<attr_name> = <something>`
  - `setattr(<obj_name>, <attr_name>, <something>)`
- Delete attributes
  - `del <obj_name>.<attr_name>`
  - `delattr(<obj_name>, <attr_name>)`
- Data attributes

    - class level
        - bare: defined directly
        - properties: defined using meta classes (not covered)
    - instance level
        - bare: defined using `__init__`
        - properties: defined using `property()` or `@property`, static or calculated

- Methods (operations)
    - instance: have access to the instance object (`self`)
        - available for use with instances not with class itself
    - class methods
        - created using `@classmethod` decorator
        - have access to the class object (`cls`)
    - static methods
        - regular functions
        - created using `@staticmethod` decorator
        - available across instance objects and class

14.10 Advanced topics

Below are some topics needed for advanced usage of programming. Understanding these topics leads to understanding of how much of the functionality provided in Python is implemented.

- Inheritance
    - single inheritance
    - multiple inheritance
- Python special methods
    - called *dunder* (double underscore) methods
    - provide some default functionality with minimal code
    - e.g. `__init__`, `__str__`, `__repr__`, `__add__`, …
    - this is the reason for convention to avoid `__` in variable names to avoid clashes
- Meta programming and meta classes

14.10.1 Inheritance

Inheritance means classes can inherit attributes from other classes, which allows classes to be structured and joined in efficient ways.

Single inheritance refers to case where a class can inherit attributes from a single class. A hierarchy is built when a class inherits from another class. The class that inherits is called the **sub class**. The class from which the sub class inherits is called the **base class**.

Multiple inheritance refers to case when a class can inherit attributes from multiple classes. Attributes are inherited following a recursive algorithm. More details are documented at Python documentation.

There are some in-built functions provided to check the class hierarchy.

- `isinstance(<obj>, <type>)`
- `issubclass(<sub class>, <base class>)`

For example `bool` is derived from `int` therefore `isinstance(True, int)` and `issubclass(bool, int)` both return `True`.

```
isinstance(True, int), issubclass(bool, int)
```

```
>>>  (True, True)
```

15 Python special features

15.1 Conditional expressions

Python has 3 specials concepts related to conditional expressions.

- **truthy/falsy**: object's associated truth value
- **execution context**: decides evaluation result
- **short circuit**

All 3 concepts can be combined and used in interesting ways which allow some newer cleaner code styles for some common situations that occur while coding.

15.1.1 Truthy and Falsy

- Every object in Python has an **associated truth value**
  - referred to as **object's truth value**
  - if the **object's truth value** is `True` $\implies$ **truthy**
  - if the **object's truth value** is `False` $\implies$ **falsy**
  - hence the object is **truthy** or **falsy**
- Below cases are **falsy** and everything else **truthy**:
  - `None`
  - `False`
  - `0` in any numeric type (e.g. 0, 0.0, 0+0j, ...)
  - `len(c) = 0`: **empty collections**
  - custom classes that implement `__bool__` or `__len__` method that return `False` or `0`
- `bool(any_object)` function returns **object's truth value**

15.1.1.1 Examples

15.1.1.1.1 Numeric type

```
num_1_1 = 1; num_1_2 = 1.0
```

```
>>>  num_1_1 =   1, bool(num_1_1) = True,   type(num_1_1) = <class 'int'>
>>>  num_1_2 = 1.0, bool(num_1_2) = True,   type(num_1_2) = <class 'float'>
```

```
num_2_1 = 0; num_2_2 = 0.0
```

```
>>>  num_2_1 =   0, bool(num_2_1) = False,  type(num_2_1) = <class 'int'>
>>>  num_2_2 = 0.0, bool(num_2_2) = False,  type(num_2_2) = <class 'float'>
```

15.1.1.1.2 Collections

```
empty_string = ""; empty_tuple = (); empty_list = []; empty_dict = {}
```

```
>>>  bool(empty_string) = False, bool(empty_tuple) = False
>>>  bool(empty_list) = False, bool(empty_dict) = False
```

```
non_empty_string = "abc"; non_empty_tuple = (1, 2)

non_empty_list = ["a", 1]; non_empty_dict = {"key 1": empty_list}
```

```
>>>  bool(non_empty_string) = True, bool(non_empty_tuple) = True
>>>  bool(non_empty_list) = True, bool(non_empty_dict) = True
```

15.1.2 Short circuit

Short circuit is general optimization strategy used by many languages. Main idea is to avoid evaluating unnecessary condition in an boolean combination.

For example in `and` combination if first condition is false then there is no point checking the second condition. Therefore, second condition of the combination is not evaluated.

Similarly for `or` combination if the first condition is `true` then the expression is `true` in both possible cases, therefore second condition is not evaluated.

15.1.3 Execution context

Python treats conditional expressions differently based on where they are used.

Boolean combinations used in conditional expression, can contain objects directly rather than comparisons. Object's truth value (`bool(<object>)`) is used rather than object itself.

15.1.3.1 `if/elif` conditions

Conditional expression used in `if/elif` statement's condition return booleans values (`True/False`).

Condition can contain object's as well, `bool(object)` is used rather than object's value.

In below code, since `x` is an empty list it is falsy, `bool(x) = False`, therefore `else` block is executed.

```
x = []
if x:
    print("x is truthy")
else:
    print("x is falsy")
```

```
>>>  x is falsy
```

In below code, since `x` is a string, it is truthy (`bool(x) = True`), therefore `if` block is executed.

```
x = "abcd"
if x:
    print("x is truthy")
```

```
else:
    print("x is falsy")
```

```
>>>  x is truthy
```

In below example, since `bool(x) = False`, `else` block is executed without evaluating `bool(y)`.

```
x = []; y = None
if x and y:
    print("x and y returned True")
else:
    print("x was false, therefore y was not evaluated")
```

```
>>>  x was false, therefore y was not evaluated
```

15.1.3.2 Outside `if/elif` condition

If a boolean combination is used outside `if` statement and contains objects rather than comparison, then the underlying object's value is returned, it may or may not be boolean data type.

Below regular comparisons are used outside `if` block and they are treated as usual, returning boolean values (`True/False`).

```
x = []; y = 2
y < 5; y == 2
```

```
>>>  True
>>>  True
```

Since boolean combination is used outside `if/elif` statement, `bool(x)` is checked and found to be falsy, using short circuit for `and`, x is returned and y is not evaluated.

```
x = []; y = 2
x and y
```

```
>>>  []
```

Since boolean combination is used outside `if/elif` statement, `bool(x)` is checked and found to be falsy, using short circuit for `or`, y is evaluated and returned.

```
x = []; y = 2
x or y
```

```
>>>  2
```

15.1.4 Summary

Tables below summarize the scenarios, where `0` and `1` signify boolean `True` and `False`.

`x`	`y`	`x and y`	`x and y` (outside `if`/`elif`)
1	1	1	`y`
1	0	0	`y`
0	1	0	`x`
0	0	0	`x`

`x`	`y`	`x or y`	`x or y` (outside `if`/`elif`)
1	1	1	`x`
1	0	1	`x`
0	1	1	`y`
0	0	0	`y`

For `x and y`, where `x` and `y` can be variables, objects or conditions:

- evaluate `bool(x)`
  - if `bool(x)` is `True`
    - if `x and y` is part of a condition in `if`/`elif` block
      - evaluate and return`bool(y)`
    - if `x and y` is not part of a condition in `if`/`elif` block
      - return `y`
  - if `bool(x)` is `False`
    - if `x and y` is part of a condition in `if`/`elif` block
      - return `False`
    - if `x and y` is not part of a condition in `if`/`elif` block
      - return `x`

Similarly, for `x or y`, where `x` and `y` can be variables, objects or conditions:

- evaluate `bool(x)`
  - if `bool(x)` is `True`
    - if `x and y` is part of a condition in `if`/`elif` block
      - return `True`
    - if `x and y` is not part of a condition in `if`/`elif` block
      - return `x`
  - if `bool(x)` is `False`
    - if `x and y` is part of a condition in `if`/`elif` block
      - evaluate and return`bool(y)`
    - if `x and y` is not part of a condition in `if`/`elif` block
      - return `y`

15.1.5 Use cases

15.1.5.1 Iterables

It is common situation where state of an iterable is not known in advance.

For example a list generated by some operation which can result in `None` or empty list or a list with values. There are some operations to be done only if the list is truthy.

Using the 3 concepts the code can be simplified as below. Both the condition are to ensure that the object is truthy.

Note that `len(some_itr) > 0` is used as second condition in `and` combination. Due to short circuit it is not evaluated if first condition is `False`, i.e. `some_itr` is `None`. If it is used as first condition, it is evaluated always, therefore will give error when the iterable is `None`.

```
some_itr =  # some iterable object
if some_itr is not None and len(some_itr) > 0:
    # code block
else:
    # code block
```

```
some_itr =  # some iterable object
if some_itr:
    # code block
else:
    # code block
```

15.1.5.2 Assign default

Another common situation is to assign a default value in case the expected value is falsy.

Below form of assignment will work. If `expected_var` is **truthy**, because of short circuit `or`, it will be returned and assigned to `new_var`. If it is **falsy**, default object will be evaluated and returned.

```
new_var = expected_var or default_object
```

15.2 Comprehensions

Comprehensions are a newer feature in Python which combine the fundamentals of map and filter in a more concise syntax.

They are relevant for iterables, i.e. tuples, lists, sets and dictionaries. Mostly they are used with lists and dictionaries.

Generic idea is

- transformation iteration filter or
- expression loop condition

i.e. map expression to each item in iterable which satisfy the filter.

- Filter/condition is optional
- Comprehensions can be nested

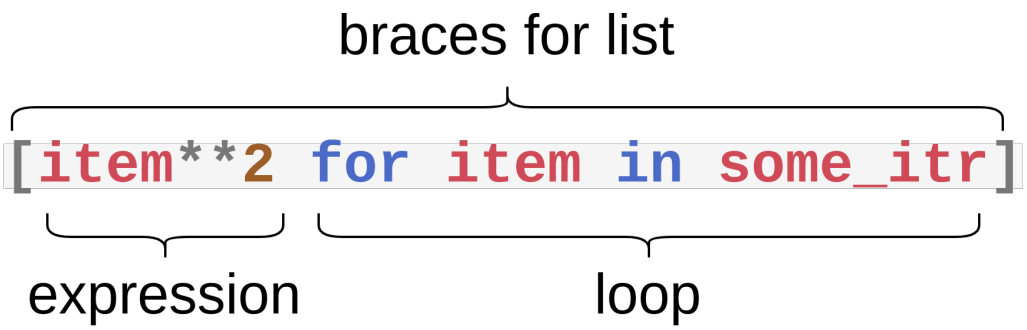
15.2.1 List comprehensions

For list comprehension generic syntax is

- [expression using item for item in list if condition on item]
- example without filter: *list of squares from a list of integers*

```
some_itr = list(range(10))
squared_itr = [item**2 for item in some_itr]
print(some_itr, squared_itr, sep="\n")
```

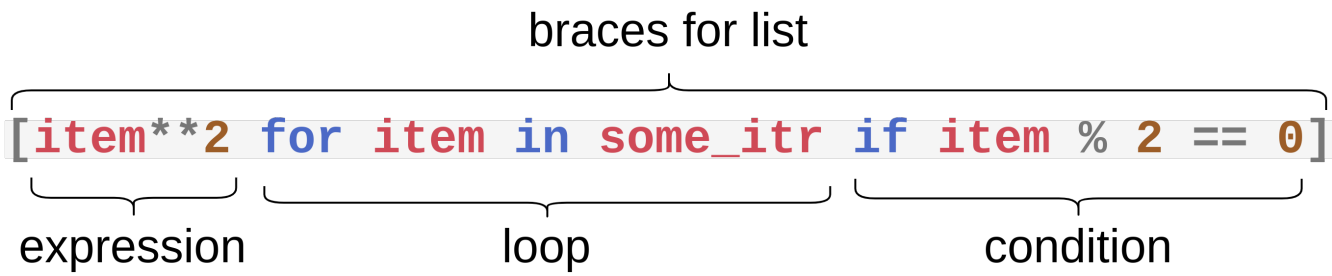
```
>>>  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```



- example with filter: *list of squares from a list of integers if integer is even*

```
some_itr = tuple(range(10))
evens_squared_itr = [item**2 for item in some_itr if item % 2 == 0]
print(some_itr, evens_squared_itr, sep="\n")
```

```
>>>  (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>>  [0, 4, 16, 36, 64]
```



- example with multiple iterables: list of sum of squares of 2 iterables of numbers

```
t_1 = (*range(1,4),); t_2 = (*range(4,7),)
sum_of_sqrs = [x**2 + y**2 for x, y in zip(t_1, t_2)]
t_1, t_2, sum_of_sqrs
```

```
>>>  ((1, 2, 3), (4, 5, 6), [17, 29, 45])
```

- example of nested comprehensions: make all possible combinations of letters of 2 strings if they are not equal

```
string_1 = 'abc'; string_2 = 'axy'
combinations = [l1 + l2 for l1 in string_1 for l2 in string_2 if l1 != l2]
print(combinations)
```

```
>>>  ['ax', 'ay', 'ba', 'bx', 'by', 'ca', 'cx', 'cy']
```

For better code readability, this can be written as

```
string_1 = 'abc'; string_2 = 'axy'
combinations = [l1 + l2
                for l1 in string_1
                    for l2 in string_2
                        if l1 != l2]
print(combinations)
```

```
>>>  ['ax', 'ay', 'ba', 'bx', 'by', 'ca', 'cx', 'cy']
```

This is same as

```
string_1 = 'abc'; string_2 = 'axy'
combinations = []
for l1 in string_1:
    for l2 in string_2:
        if l1 != l2:
            combinations.append(l1 + l2)
print(combinations)
```

```
>>>  ['ax', 'ay', 'ba', 'bx', 'by', 'ca', 'cx', 'cy']
```

15.2.2 Tuple, set, dict

To return a tuple instead of a list from a comprehension use tuple constructor instead of square brackets.

For sets and dictionary just use curly braces instead of square brackets.

- example with multiple iterables: tuple of sum of squares of 2 iterables of numbers

```
t_1 = (*range(1,4),); t_2 = (*range(4,7),)
sum_of_sqrs = tuple(x**2 + y**2 for x, y in zip(t_1, t_2))
t_1, t_2, sum_of_sqrs
```

```
>>>  ((1, 2, 3), (4, 5, 6), (17, 29, 45))
```

- example with set

```
some_set = set((1, 2, 3))
print({e**2 for e in some_set if e > 1})
```

```
>>>  {9, 4}
```

- example with dictionary

```
import string
dict_1 = dict(zip(string.ascii_letters[0:10], list(range(10))))
print(dict_1)
```

```
>>>  {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 5, 'g': 6, 'h': 7, 'i': 8, 'j': 9}
```

```
dict_2 = {k: v**2 for k, v in dict_1.items() if k in {'a', 'e', 'i'}}
print(type(dict_2))
print(dict_2)
```

```
>>>  <class 'dict'>
>>>  {'a': 0, 'e': 16, 'i': 64}
```

Part IV

Architecture

Overview

Background

After learning about building blocks, the missing piece is the knowledge of higher level specifications of the language implementations which allow combining everything into a complete program. Architecture part covers this. Knowledge of architecture part also helps in understanding and debugging programs.

- **Language specifications**
  - **Building blocks**: *specifications for elements and blocks of code*
  - **Architecture**: *specifications for writing programs*
    - means and specifications to combine elements and blocks to build programs
    - management of execution of blocks (under the hood)
- **Design**: *knowledge of how to write good programs*
- **Tools**: *needed to write, test, debug and run programs*

Introduction

- **Environment = Namespace + Scope**: *how to isolate and identify (variable) names and objects*
- **Scripts and packages**: *how to organize and use building blocks in scripts and folders*
- **External packages**: *overview of external packages available for use*
- **Language engine**: *overview of workings of Python interpreter*
- **Debugging**: *investigating and troubleshooting errors while writing programs*

Objectives

- Understand the specifications related to architecture and do experiments in isolation
- Tools: use editor and cli to manage installation, scripts and projects

16 Namespaces and scopes

16.1 Introduction

A **namespace** is a mapping from names (variables) to object references. When python interpreter reads the code it needs to lookup objects referenced by variable names used. As discussed earlier, objects are stored on computer RAM and can be accessed using object's memory address. Namespaces help find and access the associated objects using variable names. Technically, in Python, they are dictionaries that store this mapping.

A **scope** is a textual region of the code which has its own separate namespace. This helps isolate and manage different variable names and objects used in different blocks of code. Many languages do this by keeping track of namespace for all blocks, e.g. control flow blocks (`if`, `for`, …). Python restricts this to just function and class/instance object blocks. Control flow blocks like `if`, `for` etc. do not have their local scopes.

There is a lot of ambiguity in terminology related to this. For example environments and frames are used interchangeably to refer to the concept of namespace and scope discussed here. To avoid this issue, try to understand the underlying concepts and the issue with the terminology will not impact.

Python tutor is an excellent resource to do small experiments to understand the rules explained in this section. A good starting point would be to try the examples provided in the tool.

16.2 Specifications

16.2.1 Basic

Below are the scopes in the order they are searched

- **Innermost scope for a function**
  - namespace contains **local names** (*only if present*)
- **Scope of enclosing functions**
  - namespace contains **non local names** (*only if present*)
- **Global scope** (scope of the module)
  - namespace contains **global names**
  - for functions: scope of the Python file where the function is defined
- **Built-in scope**
  - namespace contains **built-in names**

16.2.2 Creation and destruction times

- **Built-in** scope's namespace
  - created when the interpreter starts up
  - never deleted, exists till interpreter is terminated
- **Global** (module) scope's namespace
  - created at compile time, when the module is read for the first time
  - never deleted, exists till interpreter is terminated
- **Local** scope's namespace for a function
  - compile time (when the definition is read)
    - local scope's namespace is not created at this stage
    - variables are tagged scopes
    - objects for default parameters are created and stored in function object
    - the function object is created in the enclosing scope's namespace
  - run time (during function call)
    - *every time function is called a new local scope's namespace is created*
    - deleted when the call terminates, i.e. on `return` or **error**

16.2.3 Function scope

- Scope of variable names used in a function definition
  - Explicit declaration of scope
    - `global`: variable is looked up in global name space skipping enclosing functions namespaces
    - `nonlocal`: variable is looked up only in the enclosing function scope
  - Without explicit declaration of scope
    - If a **variable is assigned** within a function
      - compile time: **tagged to local scope**
    - If a **variable is referenced without being assigned** within the function
      - compile time
        - the next level up enclosing scope is searched until builtin scope
        - **tagged to the first scope it is found in**
      - run time: **error is raised if not found**
- On end of function execution the function scope's namespace is deleted
- When cross referencing functions from other modules
  - the global scope for a function is the scope of the module in which it is defined
  - this will become clear after understanding modules and packages discussed in next chapter

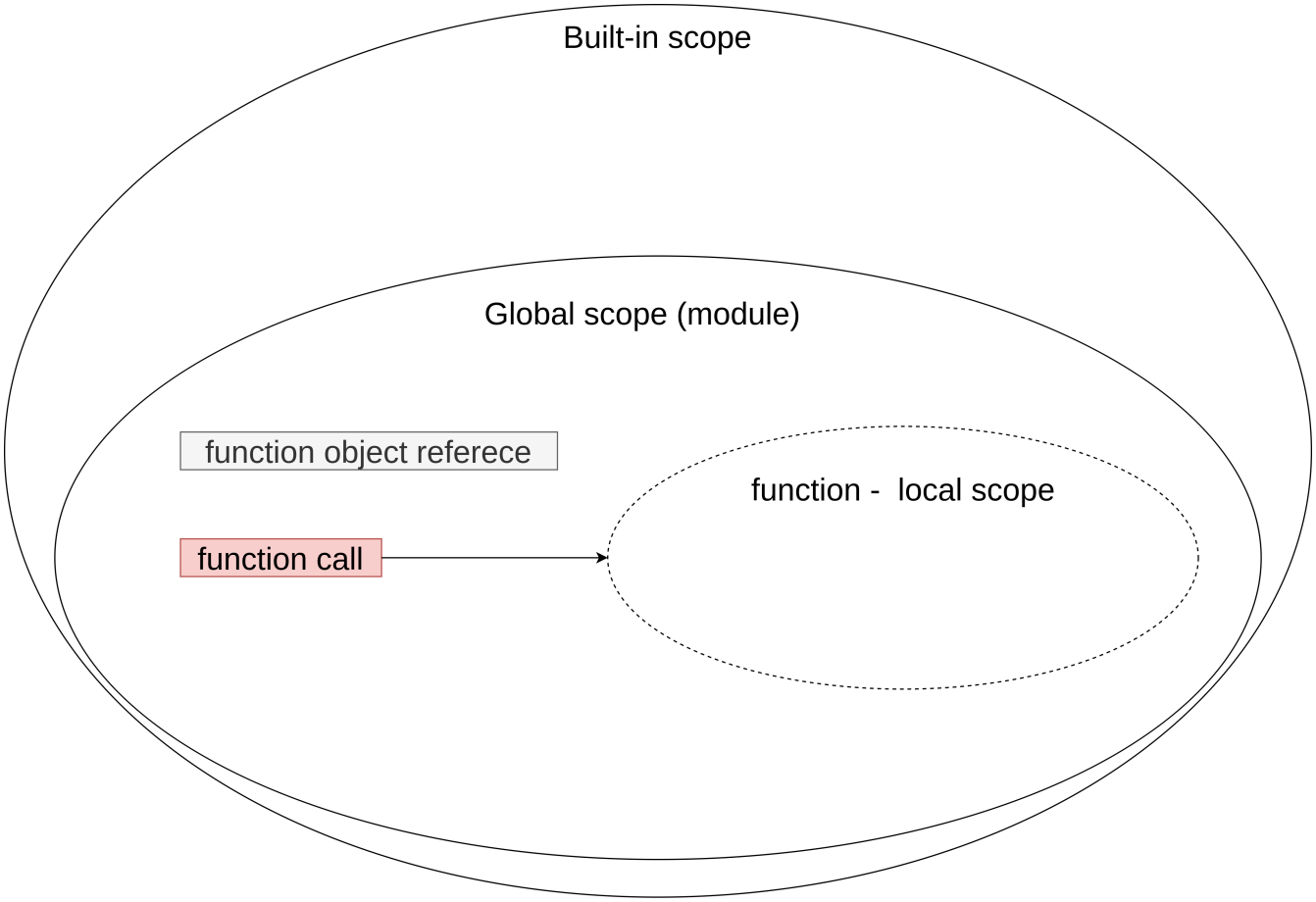


Figure 16.1: Scopes

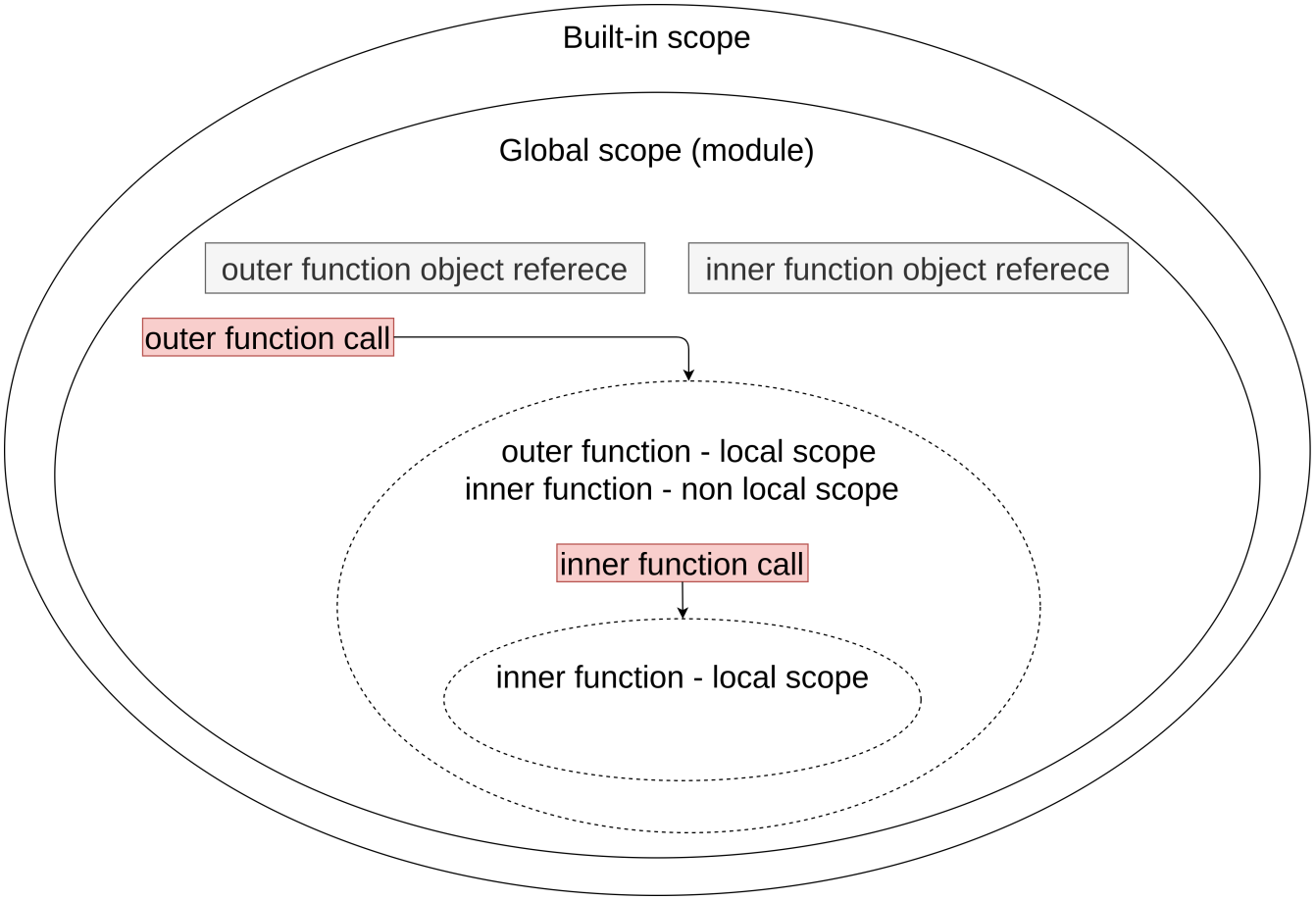


Figure 16.2: Nested scopes

16.3 Examples

16.3.1 `global`

```
var = "global"

def func():
    print("inside func call var = " + var)

func()

print("in global scope var = " + var)
```

```
>>>  inside func call var = global
>>>  in global scope var = global
```

Since `var` is not assigned, only referenced in the `func` definition, it is searched in local namespace then in global namespace. Since global namespace has `var` defined its value is used.

16.3.2 `local`

```
var = "global"

def func():
    var = "local"
    print("inside func call var = " + var)

func()

print("in global scope var = " + var)
```

```
>>>  inside func call var = local
>>>  in global scope var = global
```

Since `var` is assigned in `func` definition, it is tagged to local scope. Therefore there are 2 different `var` names in 2 different namespaces, global namespace and func's namespace created during its call.

16.3.3 `global` with nested scopes

```
var = "global"

def outer_func():

    def inner_func():
        print("inside inner_func call var = " + var)

    inner_func()

outer_func()

print("in global scope var = " + var)
```

```
>>>  inside inner_func call var = global
>>>  in global scope var = global
```

16.3.4 `nonlocal`

```
var = "global"

def outer_func():
    var = "nonlocal"

    def inner_func():
        nonlocal var
        print("inside inner_func call var = " + var)

    inner_func()
    print("inside outer_func call var = " + var)
```

```
outer_func()

print("in global scope var = " + var)
```

```
>>>  inside inner_func call var = nonlocal
>>>  inside outer_func call var = nonlocal
>>>  in global scope var = global
```

16.3.5 Declaring `global` in nested scopes

```
var = "global"

def outer_func():
    var = "nonlocal"

    def inner_func():
        global var
        print("inside inner_func call var = " + var)

    inner_func()
    print("inside outer_func call var = " + var)

outer_func()

print("in global scope var = " + var)
```

```
>>>  inside inner_func call var = global
>>>  inside outer_func call var = nonlocal
>>>  in global scope var = global
```

16.4 Use cases

Namespaces and scopes are always getting used by the interpreter while running any piece of code. In the beginning, being aware of the rules is sufficient for regular use in avoiding and resolving errors and understanding code. It is advisable to keep the use of namespaces simple.

For example while using functions, do not try to use namespace and scope rules to create complex solutions if the problem can be solved using without them.

There are many program design techniques implemented using the rules of namespaces and scopes. For example factory and decorator functions are created using these rules.

÷

- Basic use cases
    - avoiding errors
    - resolving errors
    - understand code

÷

- Advanced use cases
    - implementing design patterns
        - factory functions
        - decorators
        - generators

17 Modules & Packages

Python repl does not offer saving and repeating code. All the work is gone once the repl is closed. Jupyter notebooks are good for interactive tasks but not useful for large programs.

Scripts provide additional functionality.

- Save and run large programs from command line
- Save and re use definitions, like variables, functions and classes, related to a task
    - e.g. standard library, external modules and packages

When requirements of a project increase, so does the size of the program. Size grows quickly from hundreds to thousands of line. Putting all the code in a single script is difficult to manage. That is why breaking up the code into files and cross referencing objects across files is required to manage the coding project efficiently.

Modules and packages help organize large coding projects. The knowledge of how modules and packages work in Python also help in using scripts more efficiently.

Python documentation on modules and packages

- Python tutorial: Modules & Packages
- Python reference: `__main__`
- Python reference: Import system
- Namespace packages
    - Python reference
    - PEP 420

17.1 Module

In Python a module can refer to several different things.

- Module commonly is referred to a file type module
  folder type modules are commonly referred to as packages explicitly
- `module` is also a `type` or `class`
- Module also refers to Python object once the module file or a package is loaded using `import`
  which is instance object of `type module`

Commonly module refers to a file with `.py` extension.

There are 3 cases when module type object is created using import in Python

- *file*: a file or script containing Python code
- *package*: a folder containing sub-folders and scripts
    - *regular* package: declared using `__init__.py` file
    - *namespace* package

For example, if there is a file `abc.py` with some code and `import abc` is run on Python repl. `abc` is a module object. `abc.py` is also a file type module.

A regular package, commonly referred to as package, is also a module type object once loaded into memory by Python interpreter, the only difference is it is a folder containing `__init__.py`, even if `__init__.py` is empty.

File type and regular packages are covered here. Namespace packages are a newer feature and generally not required, hence are not covered. Information about namespace packages can be found at Python documentation and PEP 420.

17.2 Regular package

Any folder with a `__init__.py` file, even if empty, is treated as a **regular package**.

Packages provide a way to organize and exchange code as they can contain packages (sub-packages) and file type modules.

Packages are generally used to provide some functionality using functions and class definitions, which can be used anywhere in the code.

Optionally, a regular package can also contain a `__main__.py` file, this is run when the package is run as the top level environment. This is discussed in detail later in Section 17.4.1.

One of the key features of regular packages is that they allow cross referencing between modules in sub packages using relative imports.

17.3 Naming convention and rules

Convention: lower snake case

Rule: names containing special characters other than _ give error, essentially same rules as for variable names

17.3.1 Pep-8 Conventions

Object Type	Convention	Example
Packages	• short • all-lowercase names • preferably no underscores	`utilities`
Modules	• short • all-lowercase names • can have underscores	`db_utils.py` or `dbutils.py`

17.4 Usage

There are 2 main uses of modules and packages.

- Execution as top level environment
  - Python `'__main__'` system manages this
- Organizing and cross referencing objects
  - Python `import` system manages this

17.4.1 `__main__`

In Python, the system of `__main__` manages the run time behavior of modules and packages, when they are run as top level environment. It helps in differentiating how a module or package is intended to be used.

To execute a Python module or package as top level environment from command line, there are 2 main ways

- `<path to python executable> <path to file>`
  - can take relative path
  - for a regular package
    - only `__main__.py` is executed if package name is used
    - `__main__.py` or `__init__.py` can be given explicitly
- `<path to python executable> -m <name without extension>`
  - cannot take relative paths
  - command has to be executed from the directory containing the file or package
  - if the file is in a directory below the shell directory then dot notation can be used
    - e.g. `python3 -m sub_dir.file_1`
  - in case of a regular package, `__init.py` and `__main__.py` both are executed

Since using the first approach, a Python file can be executed from any directory it is the preferred approach.

Note that, while using `python3 -m <module or package name>`, if module name is `abc.py` the above command needs only `abc` without the `.py` extension.

The recommended approach for executing one script from another is to keep the files in same folder and then use `import`.

For example, `file_1.py` has to execute `file_2.py`. Place `file_2.py` in same folder as `file_1` then use `import file_2` in `file_1.py`. Whenever `file_1.py` is executed it will execute `file_2.py`. Note that if there is a main block present in `file_2`, it will not be executed.

17.4.1.1 Modules

If a module is run as top level environment, the `__name__` attribute of the module object is set to `'__main__'` by the Python interpreter in background, this is used to isolate code which is run only when module is run directly

- the code is isolated using `if __name__ = '__main__'` block
- the code is not run with `import`

Another important thing to note is that function and class definitions are bound to the module they are declared in. So it does not matter where you call the function from, the global scope for a function is the module's global scope in which it is declared.

Following coding exercise will demonstrate these fundamentals.

- create a Python file with following content and save with some name, e.g. `sample_mod.py`

```
print("running sample module")
print("__name__ attribute of the current module = " + __name__)

if __name__ == "__main__":
    print("running main block of sample module")
```

- from command line goto the folder containing `sample_mod.py` and execute the command
  `python3 -m sample_mod`
  - notice that print statement of the `if` block is executed because `__name__` attribute of the module is set to `__main__`
- now start Python repl by entering the command `python3`
  in repl enter `import sample_mod`
  - notice that main block is not run because name of the loaded module is set to `sample_mod`
  - import system is discussed in later section

17.4.1.2 Regular packages

Any folder containing `__init__.py` is treated as a regular package by Python. Packages can additionally contain `__main__.py`.

For packages, instead of a `if __name__ = '__main__'` block, `__main__.py` is used. If `if __name__ = '__main__'` block is used in `__init__.py`, it will give error when the package is called directly as the top level environment.

Executing *with* `-m` switch

If `__main__.py` is present in the regular package it is run along with the `__init__.py` when the package is run as the top level environment using the `-m` switch. If `__main__` is not present, error occurs. Again this helps isolate code which needs to run only when the package is run directly as top level environment. When calling the same package using `import`, `__main__.py` is not run.

Executing *without* `-m` switch

If a regular package is run without `-m` switch, only `__main__.py` is run else error is given.

Regular packages in general are supposed to be used with import for reusing object definitions rather than executing code. `__main__.py` is for special cases, e.g. testing.

This is simple to test.

- In a folder in system create below folder structure

```
mkdir reg_pkg_1 reg_pkg_2
cd reg_pkg_1; touch __init__.py __main__.py
cd ../reg_pkg_2; touch __init__.py; cd ..
```

- write contents to the files

```
# reg_pkg_1.__init__.py

print("running regular package 1")
```

```python
# reg_pkg_1.__main__.py

print("running main from regular package 1")
```

```python
# reg_pkg_2.__init__.py

print("running regular package 2")

if __name__ == "__main__":
    print("running main block of regular package 2")
```

- experiment
  - run each package as top level environment with and without `-m` flag
  - import each package from repl
- note that running `reg_pkg_2` as top level environment will give error because it is a regular package and `if __name__ = '__main__'` does not work

17.4.2 `import` system

Import system in Python helps manage using object definitions (variables, functions, classes) from other scripts.

Once imported, a module or package is a module type object with attributes. Python adds some special attributes to module type objects.

- `<module name>.__name__`: module's name or `__main__` if run directly
- `<module name>.__file__`: absolute path to the module location
- `<module name>.__package__`: is populated if it is a regular package or a part of one, else is empty string
- `<module name>.__path__`: exists if module is a regular package or is part of one, else gives error

All this is supported using Python's import system. Detailed documentation can be found at Python language reference: Import system.

17.4.2.1 Modules

Full module (file or package) can be imported using below statements.

- `import <module name>`
- `import <module name> as <module alias>`

17.4.2.2 Objects & sub-modules

While importing, only subset of a module or package can be imported. It can be specific objects or modules or sub-packages within a parent package.

- `from <module_name> import <object_name>`
  - `from <module name> import *`
  - `from <module name> import <object name>`
  - `from <module name> import <sub module name>`
  - `from <module name> import <object name> as <object alias>`
  - `from <module name> import <sub module name> as <module alias>`
- `from <module name> import *` is not recommended when using other packages

- it pollutes the namespace
- creates name clashes
- `<module name>.<obj name>` is better for code readability
  It is provides information about source of the object used

17.4.2.3 Runtime behavior

Whenever a file or package type module is imported the following things happen

- Python interpreter checks if the module/package is already loaded into current namespace
  - if the module/package is loaded
    - nothing is done
  - if the module/package is not loaded
    - Python interpreter runs the code in module or `__init__.py` for a package
      - main code is not run
    - all the exported objects are loaded
- When only a sub package is imported Python still loads the parent package
  - therefore there is no efficiency gain if only a sub package is imported

17.5 Applications

A Python project might be intended to do one or both type of below tasks.

- Perform some task[s] by running a script as top level environment
- Create a package to define objects (functions, classes, data-objects, …) for re-use and sharing

Below are the use cases with recommendations.

- **Adhoc and small projects**: jupyter notebooks are sufficient most of the times.
  - if code grows, definitions can be kept in modules and imported into notebook
- **Adhoc and large projects**: modules may be needed.
  - If the size of code grows, it is useful to organize the code using regular packages.
- **Recurring tasks**: it makes sense to use modules which can be run from command line.
  - If the size of code grows, it is useful to organize the code using regular packages.

17.5.1 Small projects

For small projects where the code can be well structured using 4-5 file type modules kept in a root directory.

As an example, root of the project can contain

- `main.py` to run the task from top level
  - it refers to function definitions and input objects from other modules as described below
  - it is a convention to name this file as `app.py` or `main.py`
- `category_1.py`, `category_2.py`, … contain all functions for a category of tasks
- `inputs.py` contains all inputs required for configuration

Folder structure will be flat in this case. Objects can be cross referenced without need of relative imports.

- project root
    - `main.py`
    - `inputs.py`
    - `category_1.py`
    - `category_2.py`

17.5.2 Large projects

For large projects it is advisable to use regular packages. This allows use of relative imports and using sub-packages, i.e. each sub directory is a regular package in itself with file type modules.

Packages allow categorizing and storing modules in folders and sub folders, then **cross referencing objects** as needed using relative imports.

18 Available Modules & Packages

Modules and packages allow code re-use and distribution. There are several modules and packages available which provide definitions (functions, constants, classes) for a variety of use cases.

There are 3 main resources for using external modules and packages for use in Python code.

- Built-in
- Standard library
- PyPI: Python package index

18.1 Built-in

These are built-in objects that are always available. They are loaded by default and do not need the use of `import`.

- Built-in **functions**
- Built-in **constants**
- Built-in **types**
- Built-in **Exceptions**

18.2 Standard library

Standard library consists of several modules (files and packages) available with standard installation.

These have to be loaded using `import`and are not loaded by default like the built-in objects.

Some of these, like `math` and `sys`, are written in `C` for speed.

Full list of available modules with documentation can be found in Python docs: library reference.

18.2.1 Frequently used modules

Topic	Module	Description
Python Runtime Services	sys	System-specific parameters and functions
Generic Operating System Services	os	Miscellaneous operating system interfaces
	io	Core tools for working with streams
	time	Time access and conversions
	argparse	Parser for command-line options, arguments and sub-commands
File and Directory Access	pathlib	Object-oriented filesystem paths
	os.path	Common pathname manipulations
	glob	Unix style pathname pattern expansion
	shutil	High-level file operations
Data Persistence	sqlite3	DB-API 2.0 interface for SQLite databases
File Formats	csv	CSV File Reading and Writing
Functional Programming Modules	itertools	Functions creating iterators for efficient looping
	functools	Higher-order functions and operations on callable objects
	operator	Standard operators as functions
Data Types	datetime	Basic date and time types
	zoneinfo	IANA time zone support
Text Processing Services	re	Regular expression operations
Numeric and Mathematical Modules	math	Basic math
	statistics	Statistics

18.3 Python package index (PyPI)

`PyPI` handles open source contributions to the language. These are external packages which have to be installed before they can be loaded using`import`.

As Python is one of the most popular languages, it is easy to find a package for almost every use case by searching the web or `PyPI`.

`pip` is the installer for external packages on `PyPI`.

18.3.1 Some important packages

Topic	Module	Description
Installation	`pip`: home, doc	Python install package
Jupyter	`jupyterlab`: home, doc	Jupyterlab, interactive notebooks
Scientific Computing	`numpy`: home, doc	Fundamental package for scientific computing with Python
	`scipy`: home, doc	Mathematics, science, and engineering. It includes modules for statistics, optimization, integration, linear algebra, Fourier transforms, signal and image processing, ODE solvers, and more.
Data Analysis	`pandas`: home, doc	Data structures and operations helpful for data analysis. Dataframes, series, …
Data Visualization	`matplotlib`: home, doc	Generic visualization
	`seaborn`: home, doc	Based on matplotlib, for statistical visualizations

18.4 Virtual Environments

Virtual environments are used to creates separate installation location for external packages, which helps with following

- keep installation of external packages organized for different tasks
- keep track and manage version requirements for external packages needed for specific projects

At a high level, virtual environment isolates the use of external packages from built-in and standard library.

18.4.1 Why use a virtual env?

The external packages keep releasing new versions for bug fixes, enhancements and new features.

When working on projects which depend on external packages it is critical to keep track of version of external packages as latest version might no longer support a feature that the project code needed.

When installing packages in base installation there might be a conflict between dependencies of different project's. Some projects might depend on a version of a package while other projects might depend on a different version of the same package.

Some projects might need a specific version of Python itself.

There might be some packages which are needed for one time tasks. This keeps on adding complexity of managing packages which are not needed any more.

All these reasons lead to development of virtual environments and its use is recommended.

18.4.2 Usage

- **How to create a virtual environment?**
  - `venv` module is part of standard library and can be used with bash commands
  - unix/mac: `python3 -m venv <path to new venv>`
  - win: `py [-v] venv <path to new venv>`

This creates a folder with the name in the given path. Usually it is helpful to name virtual environments starting with `.`, e.g. `.py-venv`. This helps identify that this is a Python virtual environment folder. This dot keeps the folder hidden which is helpful in keeping the folder view clean as the `venv` folder is seldom used directly.

It is recommended to use Python version in bash commands, while creating and restoring `venv`.

- **how to activate `venv` using bash?**
  - Linux/Mac:
    - `source <venv path>/bin/activate`
  - Windows:
    - `source <venv path>/Scripts/activate`
  - Editors, like VSCode, allow to select Python interpreter for running a project
    - `venv` can be set to be activated by default

Once the `venv` is activated, the associated Python version and external packages installed in the virtual environment are used. Installing packages using `pip` installs them into the activated virtual environment.

- Resources
  - Python docs: venv

18.4.3 Project dependencies

- `venv` along with `pip` can be used to manage project dependencies
- `venv` keeps installed packages isolated in a virtual environment
- `pip` is used to create a list of required external modules installed in `venv`
  - windows: `py [-v] -m pip freeze > py-requirements.txt`
  - unix/mac: `python3 -m pip freeze > py-requirements.txt`
  - version dependencies can be managed as well
- `pip` is used to re-create `venv` by installing required external modules contained in `<py-requirements.txt>` file in `venv`
  - windows: `py [-v] -m pip install -r py-requirements.txt`
  - unix/mac: `python3 -m pip install -r py-requirements.txt`
- resources: pip user guide

While using `pip` or `venv` it is advisable to use full command with specific Python version. This ensures that the associated `pip` and `venv` is used.

`pip` records `<package/module name>==<version number installed>`. `pip` documentation has more details how to manage packages for a project using `pip`.

18.4.4 Structuring `venv`'s

There are a couple of commonly used strategies for organizing and structuring virtual environments.

- **Centralized approach**: keep all virtual environments in a central location
  - related projects can share virtual environments
  - typically location is `$HOME/.venvs/<name of venv>`
- **De-centralized approach**: each project has its own virtual environment
  - simple but effective as there is no need to manage conflicts
  - similar projects can be grouped in a root directory with a common `venv` if needed

  Given the amount of memory used by virtual environments and simplicity of management, de-centralized approach makes more sense unless there is a specific reason.

A sample small project's folder structure with `venv` and `git` could look like below.

- project root
  - **.git/**
  - **.py-venv/**
  - **docs/**
  - `main.py`
  - `inputs.py`
  - `category_1_funcs.py`
  - `category_2_funcs.py`
  - `py-requirements.txt`
  - `.gitignore`

19 Language engine

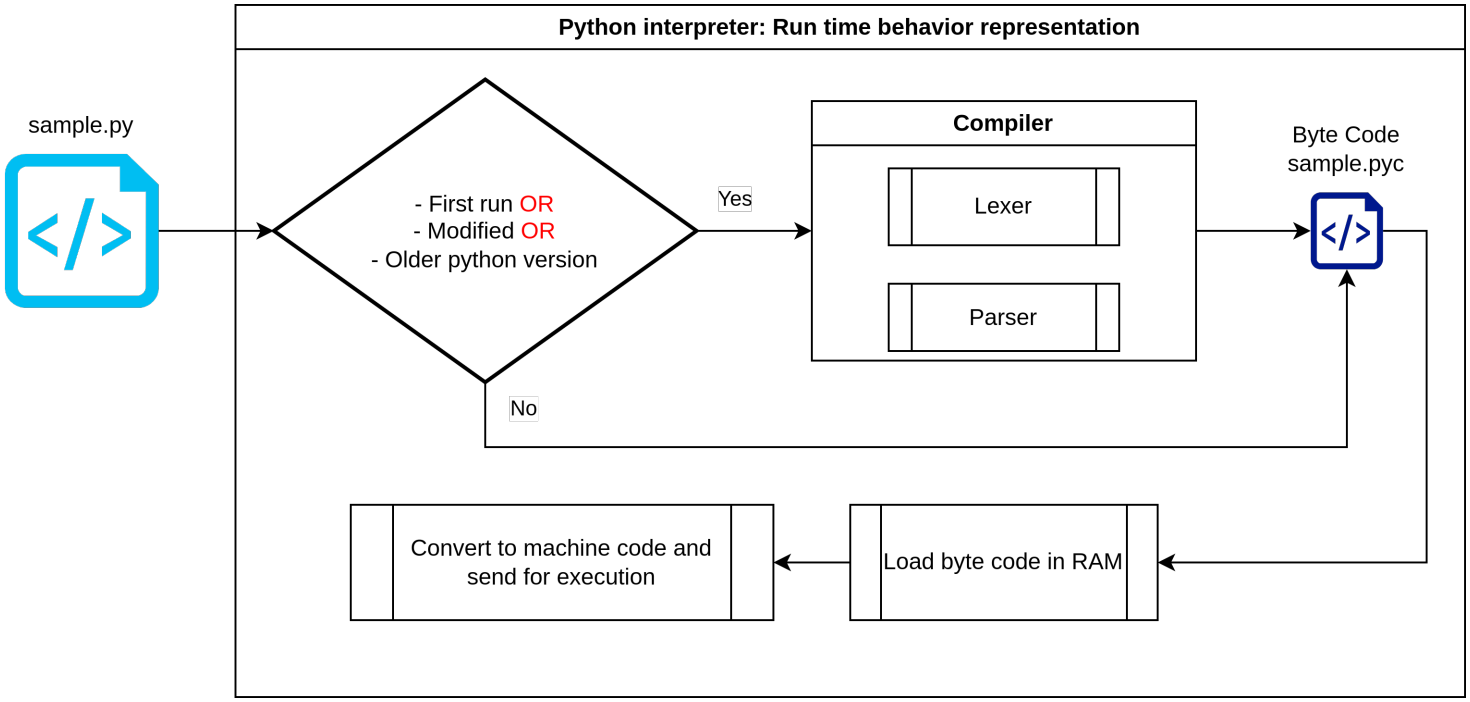
19.1 Overview

As discussed in Chapter 2, overview of programming languages, there are 2 distinct aspects related to a programming language, specification and implementation. This book refers to the official implementation of Python which is an **interpreter written in C**, and therefore also referred to as **CPython**.

Therefore below terms usually refer to the default Python implementation.

- Python interpreter
- CPython
- CPython interpreter

When a Python script is run, Python interpreter does a lot of things in the background. Below diagram is a representation of major steps to help understand the mechanics of run time behavior.



- The interpreter checks the conditions for .py file
  - is .py file a new file
  - is .py file modified since last compilation
  - was .pyc file written using an older Python version
- If any of the conditions is true then .py file is compiled to .pyc file (byte code)
- Byte code is converted to machine code and sent for processing

This is an simple representation to help understand run time behavior of interpreter, but under the hood there is much more going on.

Since Python is a very high level language, there are more number of things kept under the hood, the programmer does not need to deal with them directly. In relatively lower languages like C/C++ some of these have to be dealt

with by the programmer. Even in Python, some of the features can be turned off and dealt with directly for some advanced use cases.

Below is an introductory overview of things that happen under the hood which interpreter handles.

- **managing execution**: sending and receiving of operations and data, to and from the processor
- **garbage collection**: clearing un used objects created during execution
- **threads and processes**: python interpreter uses single thread and single core at a time by default

19.2 Hardware Management

To understand this with full context requires a lot of effort and is beyond the scope until you pursue computer scientist or developer paths, but understanding all this at a high level will help.

The first step is to understand the high level map of layers making the computer do what it does.

1. **Hardware**: CPU controls all the different components. It understands only machine code (binary).
2. **Operating system (OS)**: is the first major layer on top of hardware that controls all programs or applications interacting with the hardware.
3. **Programs**: also referred to as *computer applications* or just *applications*. This is the final layer offering specific interactions with the hardware. These have to go through the OS to finally send and receive instructions to and from hardware. Some examples include
    - **file explorer**: provides a graphical interface to interact with the file system stored on hard disk. It displays the information on monitor and provides interaction through mouse and keyboard events.
    - **browser**: provides a graphical user interface (gui) to interact with the data on the web. In the background it sends and receives data through network adapter and displays it on monitor.
    - **network**: there are network processes always running in the background to keep the machine connected to web and internal networks

The OS runs some default processes in background to provide the default functionality. In addition to this any application run, is run through the OS, it can create 1 or more independent process[es]. OS allocates space on RAM for each process. An application or program is stored on hard disk as executable or machine code. Once the executable is run, the OS allocates a dedicated space on RAM to the process, which holds the instructions and data to execute. The data and instructions are sent to CPU through OS. The OS manages data and instructions from all running processes.

For example when a Python program is run,

- OS
    - starts process[es] requested by Python interpreter
    - allocates space on RAM for these process[es] using some algorithm
    - manages sending/receiving data and instructions to/from CPU received from all running processes
- Python interpreter
    - sends/receives data and instructions to/from CPU through OS
    - loads all the resources needed to run the program from hard disk to RAM
    - manages the usage of allocated space on RAM
- CPU
    - does all the processing
    - manages all the data and instructions received from OS
    - sends back the results to OS

There are 2 critical resources that a program has to manage:

- Memory (RAM)
- Processor (CPU)

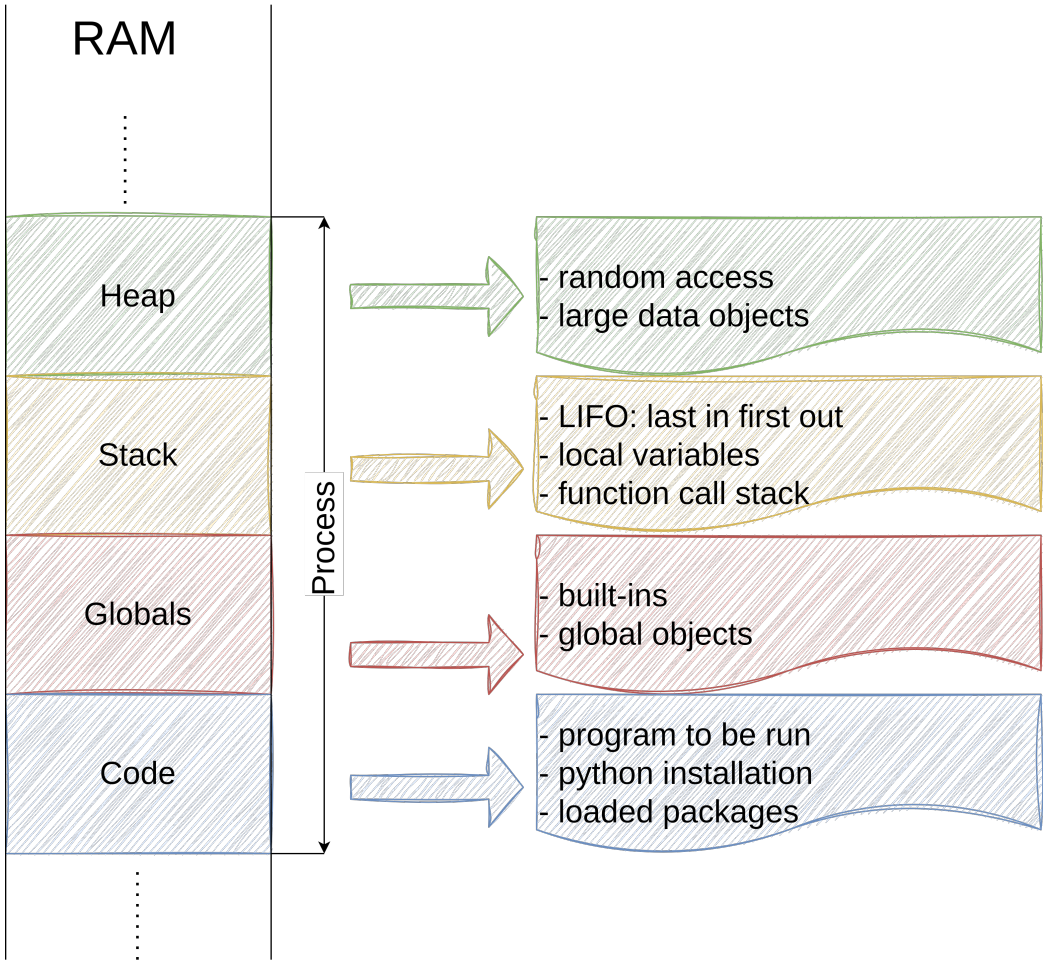
Python being a very high level language, manages both these automatically in background, unless programmer specifies not to, in which case there is manual configuration possible.

Languages, mostly compiled, like C/C++ provide minimal management and programmer has almost complete control over these aspects, which increases complexity. These are required when developing applications and require a lot of effort to learn them correctly. Typically as a beginner and for light usage of programming, this is not required, general purpose language like Python is sufficient.

19.2.1 Memory (RAM)

The space on RAM allocated by the system is used by the program as below representative diagram suggests. The space is broken into components for certain types of data.

This is a rough representation and almost all languages follow similar structure, even compiled programs in languages like C/C++.



Stack and heap are parts which can be accessed from the program.

Table 19.1: Stack Vs Heap

Stack	Heap
• used to store smaller data like references	• used for storing larger objects
• relatively smaller in size	• size grows as the need of the program
• size does not grow during execution	• **garbage collection** is used to clear space automatically

19.2.1.1 Stack

Stack is the component of memory used for program execution. It is modeled as stack data structure, like a book stack, where the last book added is removed first, also called last in first out principle or LIFO.

The code instructions and variable names are sent to the stack as the program is read and deleted when the instruction completes.

Terms like `stack trace`, `stack overflow` relate to this portion of programs memory allocation.

Stack overflow is self explanatory. It is easy to test, when you run an infinite recursive function with minimal data the stack gets full at around 3000 function calls and you get error as the stack gets full.

```
n = 0
def inf_rec():
    global n; n += 1; print(n)
    while True: inf_rec()
inf_rec()
```

Stack trace is the history of instructions sent to stack which helps in debugging and is used to print messages on errors.

19.2.1.2 Heap

Heap is the part of allocated memory which is used for storing data objects. Heap functions like RAM, random memory addresses can be accessed.

For example, in Python if code has statement, `x = [1, 2, 3, 4, 5]`, x is stored on stack and the list object is stored on heap.

High level languages like Python have garbage collection which clears heap memory by using internal algorithms.

19.2.2 Processor management

Managing the CPU through program is considered as one of the most complex topics in programming as it involves understanding multiple topics

- CPU: how CPU processes data and instructions sent to it
- OS: how does OS schedules and manages tasks from all running processes
- Program: what features the programming language provides to handle execution of different tasks in a program

All these parts have a lot of details to be understood and then applied together during design of a large program.

For regular use of programming there is not much requirement to increase performance by designing the program. This is needed when building large and advanced applications.

Python interpreter, by default, starts a single process with 1 main thread while running a program. Python has Global interpreter lock (GIL) which limits execution of one thread at a time.

20 Error handling (debugging)

In Section 12.4 on Error handling, source of different types of errors was discussed.

Errors in context of computer programming are also called **bugs** because of historical significance. Earlier, when programs were written on actual physical tapes, a large test program stopped and later it was found that there was an actual bug (a moth) that stuck to the program tape. Therefore preventing or removing errors is called **debugging**.

There are several contexts where error handling is used.

- Control flow: handling expected errors through code
- Debugging: investigating errors while writing code
    - **trace back**
    - **debugger**
- Testing: automate testing for reuse
    - **Unit tests**: Automating testing of small pieces of code
    - **Integration tests**: Automating testing of combination of pieces working together
    - **Resources**
        - Python standard library: `unittest`
        - PyPI package: `pytest`

This section discusses debugging, which is relevant for all stages of programming.

The second part, program design for making robust code through testing, is relevant for writing large programs when it is hard to test the whole program without designing tests. This is not included in the book because for most light uses of programming as a beginner this is not relevant.

20.1 Error types

While writing a program below are possible types of errors.

- **Lexical and syntax errors**: these are easily caught by the editor tools like linter and style checkers
- **Semantic errors**: when some rules of implementation details are violated
    - e.g. passing a string to function when number was required
    - some common type of errors are caught by the editor tools, but not all
- **Incorrect logic**: No error but the solution is incorrect
    - e.g. while using nested loops the order of loop matters

20.2 Tools

Most editors provide support for integrating tools for editing as discussed in tools section (Chapter 9). Related to debugging while writing code there are 2 main relevant parts

- Editing tools: syntax highlighting, auto completion, refactoring, etc.
- Lint tools

These are preventive tools that help in avoiding errors at run time. These help avoid lexical and syntax errors. Semantic errors are caught by lint tools to some extent.

Python interpreter, like most language implementations, provides 2 main tools.

- **Exceptions & trace back**
- **Debugger**: useful for Semantic errors and incorrect logic errors

20.2.1 Exceptions & trace back

Exceptions were discussed in Section 12.4. They contain 2 key information

- Exception type
- Trace back: also called stack trace, stack traceback, stack backtrace etc.

The stack trace is like history of execution instructions sent which caused the error.

When the error is raised, message printed contains both these to help identify

- Where the error occurred
- Type of error if it is a known type

It is useful to note the order of stack trace information. **Starting from the bottom, it tells where the error occurred and then traces back where the call to this error originated from.**

Below is a simple example to illustrate the point. Exception information is displayed at top and bottom and rest is traceback.

```
some_list = [1, 2]

def f1(): some_list.pop()
def f2():
    for i in range(3):
        f1()

f2()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[1], line 8
      5     for i in range(3):
      6         f1()
----> 8 f2()

Cell In[1], line 6, in f2()
      4 def f2():
      5     for i in range(3):
----> 6         f1()
```

```
Cell In[1], line 3, in f1()
----> 3 def f1(): some_list.pop()

IndexError: pop from empty list
```

20.2.2 Debugger

Python interpreter, like most languages, provides a set of tools for debugging, collectively referred to as debugger. These special set of commands which can interrupt the interpreter at given point saving the state of stack and heap and provide interactive execution options.

VSCode, like most editors, integrates this functionality in the editor. The respective section in VSCode: Python tutorial: Debugging provides a good walkthrough of the functionality.

Note that almost everything related to debugging can be done manually by writing breakpoints in code and running the script through command line, but editors provide a useful user interface around it.

Debugger is useful in case of

- Runtime errors where error message cannot provide enough information
  - e.g. Exception is too generic to spot the error
- Program is running correctly but with unintended results
  - it is not uncommon to get into this situation

Part V

Design

Overview

Design in context of programming will depend on the context. Since programming itself is relatively a new field of knowledge and evolving, compared to other subjects like math, structure and content of design concepts is not consistent and formalized. Therefore in current learning resources, for an introduction to programming, often this will be missing or if present, the structure will not be consistent. So consider this section to be an independent attempt to structure the concepts of the topic.

The motivation for design in context of computer programming comes from the rapid growth of scale and complexity of programs. The main purpose of studying design in context of programming is to improve the properties of a program.

It is one thing to write a program that solves the problem, but knowledge and judgement of designing a good program is a skill acquired through practice.

With experience of developers trying to solve more and more problems and building very large programs there has been abstraction of properties of a good program.

What are the recommended properties of a good program?

- **Readable**: easy to read and understand
- **Testable**: safe from errors, easy to test even after making changes
- **Modular**: responsibilities are separated out into small independent blocks
- **Extensible**: easy to extend functionality with little change in existing code
- **Efficient**: in terms of speed and memory

There are few independent sub contexts when it comes to design in context of programming.

1. Design **components of programs**
    - **Design patterns**
    - **Data structures & Algorithms (DSA)**
    - **Regular expressions**
    - **Testing**
    - **Specifications**
2. Design **structure of programs** (**Frameworks**)
3. Design **process of creating programs** (**Workflow**)
    - **Software requirement analysis**
    - **Documentation**
    - **Refactoring**

In context of this book, some of these topics have been included with the following objectives.

- It gives more context when using existing solutions from developers.
- It allows to benefit from best practices by using them early.
- It provides opportunity to apply and practice some of components in small code.
    - This should help those who happen to pursue programming to advanced levels.

Design patterns

Design patterns are abstraction of ways to organize building blocks using the language specifications and architecture to improve properties of a program. Typically a given pattern improves a subset of properties.

Design patterns can be specific to a domain within software development or generic.

There are 2 distinct dimensions

- abstraction of tasks
- abstraction of patterns to implement those tasks

As the size and number of programs grow, there is progress in both these dimensions. Some patterns emerge as good solution for certain tasks. A certain type of task might be solved using multiple patterns.

Map reduce is a design pattern which is covered with functions (Section 13.8) using functional programming paradigm. Filter is just a special case of map.

Python provides many more such design patterns which have not been included to keep the volume and complexity low for an introduction to programming. These topics are suited for an intermediate course, where these can be covered in more detail, like learning to implement these.

- **Map-reduce**
- **Iterators**
- **Generators**
- **Decorators**
- **Context managers**
- **Advanced OOP**
- **Meta-programming**

21 DSA

21.1 Introduction

Data structures and algorithms, referred to as **DSA**, is an advanced topic in computer science.

At macro level, data and correspondingly requirement for data analysis is growing exponentially, different use cases get created with this in every domain. It is thus a requirement to store and operate on data efficiently. Storage has become cheap hence efficiency/speed is more important now.

Efficiency in context of speed is one of the most needed property of a computer program. Managing storage and operations is a major factor on which efficiency of a program depends.

There are multiple ways to increase performance.

- Hardware: faster CPU, which is hitting a boundary
- Program Design: concurrency, multi-threading, multi-processing, async-io
- DSA: has been instrumental in increased performance of applications

---

At micro level, a program instructs CPU to perform operations which take time.

- load data into RAM
- fetch data from RAM
- operate on data
- write data onto RAM

All these operations happen at very large scale even in simple programs, e.g. basic data analysis. Therefore designing how the data is structured (data structure) and operated upon (algorithms) allows reducing these operations to improve performance.

Computer scientists are involved in rigorous study of the topic, including math involved, to come up with new data structures and algorithms. There are 2 main challenges

- find optimal solutions in terms of speed and design to the ever expanding field of data
- prove and communicate the solutions

As a user, the problem is to match the right data structure if it exists with the use case.

As an advanced user or developer there might be a need to implement needed data structures.

Some languages like Python provide implementation of basic data structures as default while others like C leave it to the users to implement.

In Python, sequence types like list and tuple, mapping types like set and dictionary are all end result of this field of study, and are implemented and provided by default.

The choice of data structure decides available algorithms for different operations implemented for a data structure.

Choosing the data structure depends on use case

- how data is to be structured?
- which type of operations are to be performed?
- how frequently?

This section of the course aims to present and introduce some of the core results which are used currently, along with context and intuition behind the developments in the field. This should help

- everyone
  - during usage of these concepts in writing programs
- someone whose target is to get into software development
  - lay foundation for advanced study of the topic

The study of data structures and algorithms can be summarized as below.

- **problem specification**: **interface**
  - specification for **structure** of data
  - specification for **operations** needed to be performed on data
- **solution/implementation**
  - **data structure**: implementation of how to store the data with given structure
  - **algorithms**: implementation of how to perform operations for a given data structure
- **measuring efficiency**
  - **asymptotic notation**
  - **computation model**: **WordRAM** model

21.1.1 Interface vs data structure

Interface	Data structure & Algorithms
Interface is a specification of structure of data and operations that should be supported	Data structure is a implementation to store data with specified structure Algorithms for a given data structure are the implementation of operations
Interface is a **specification**	Data structure with Algorithms is an **implementation**
Interface is the **problem definition**	Data structure with algorithms is a **solution**
There could be multiple data structures for the same interface, performance will be different	A data structure might solve multiple interfaces, performance will be different

21.2 Measuring efficiency

There are 2 major costs involved while using a data structure

- **time** taken to perform operations
- **memory** (**space**) used on **RAM**

Measuring time and memory usage depends on 2 factors

- computation model (hardware)
- how the data structure uses this model

Typically time is more critical as memory is considered cheap in the context of present day computer hardware.

21.2.1 Computation model

Computational model refers to abstracting hardware performance in general terms to study the behavior of different operations to be performed. Typically *word RAM* model is used in theoretical studies. In practice it may differ.

Implication of assumptions in *word RAM* model are that all elementary operations take constant amount of time

- read, write, delete in RAM for a location
- mathematical operations
- logical operations

RAM => Random access memory

21.2.2 Measuring time

21.2.2.1 Context

Time complexity of an algorithm can be measured in different ways based on different contexts.

- how the time complexity is measured?
  - measure time to run the algorithm
  - **measure operations performed and how they grow with size of data structure**
- what time is captured?

- best case time
- average time
- **worst case**

Measuring time directly has following disadvantages

- it depends on the quality of machine
  - tests can be run on various machines but that is inefficient
  - testing a slow algorithm on very fast machine leads to false conclusions
- it depends on size of data
  - tests can be run on different data sizes but it is difficult to test for very large sizes

It is more useful to measure time in terms of operations performed and then see how they grow with size of data. Using this approach isolates dependency on machine.

Using best case time for deciding time complexity might give false results as an algorithm might be fast on small data size but perform slow on large data size.

Using average time will give more information, but for that the probability distribution is needed to decide which data sizes will be used more frequently. It is very difficult to get an accurate distribution hence this option is not feasible.

Worst case time is used as it ensures bound on performance and reduces noise in comparing performance of different algorithms.

21.2.2.2 Solution

Time complexity is measured for **worst case** performance, using **asymptotic notation** for **number of operations performed depending on data size**, usually represented with n.

21.2.2.3 Asymptotic notation

Asymptotic notation is used to get an idea of asymptotic growth ignoring scaling factors and constants. More formally, asymptotic notation represents a set of functions.

Asymptotic growth simply means how the growth of a function behaves when the underlying variable grows too large. For example, if the the performance of an algorithm for a data structure takes the form $T(n) = 3n^2$, what is the growth in value of $T(n)$ as $n \to \infty$, where $T(n)$ is the time the algorithm takes and n is the size of data structure.

Below are the set of asymptotic notations.

Name	Notation	Description
Asymptotically Equal	$f \sim g$	
Big O	$f \in O(g)$	upper bound
Omega	$f \in \Omega(g)$	lower bound
Theta	$f \in \Theta(g)$	both upper and lower bound
Small O	$f \in o(g)$	strictly larger
Little Omega	$f \in \omega(g)$	strictly smaller

Theta and Big O are mostly used in measuring complexity.

The notation is some times used as $f(n) = O(g(n))$ but it is more accurate to use $f(n) \in O(g(n))$ as $O(g(n))$ denotes a collection of functions.

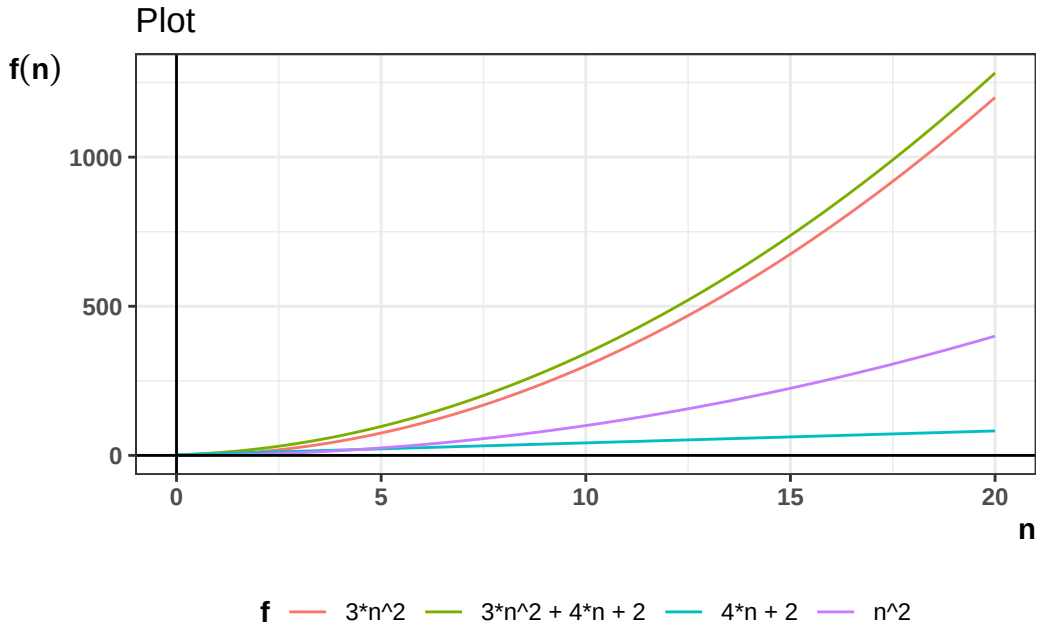
Definition 21.1 (O Notation)**.** A non-negative function $f(n)$ is in $O(g(n))$ if and only if there exist constants $c, n_0 \in \mathbb{R}$ such that $f(n) \leq c \cdot g(n) \quad \forall\ n > n_0$.

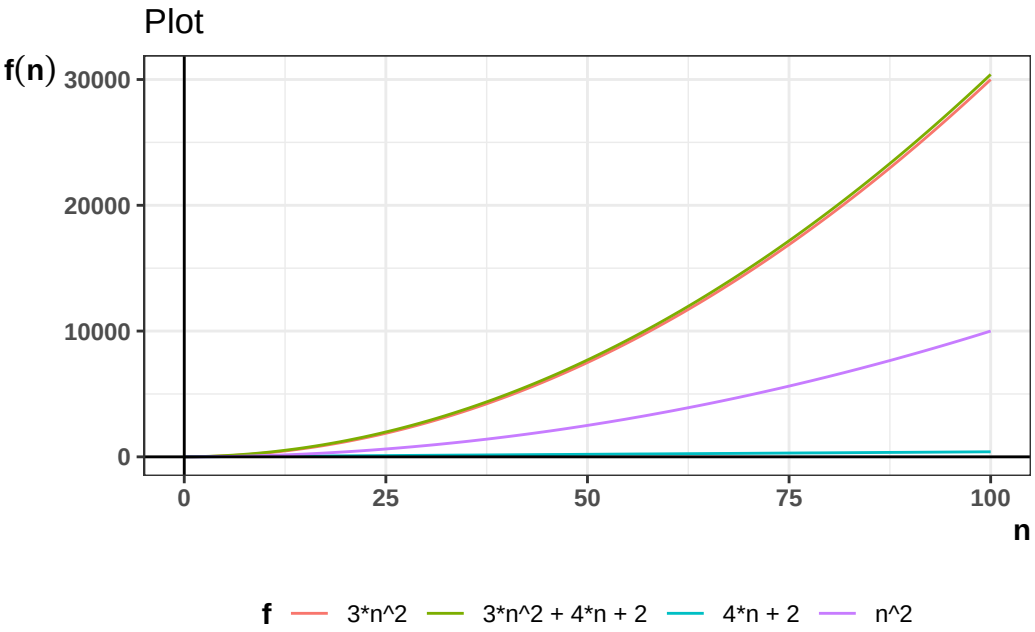
Definition 21.2 (Omega Notation)**.** A non-negative function $f(n)$ is in $\Omega(g(n))$ if and only if there exist constants $c, n_0 \in \mathbb{R}$ such that $f(n) \geq c \cdot g(n) \quad \forall\ n > n_0$.

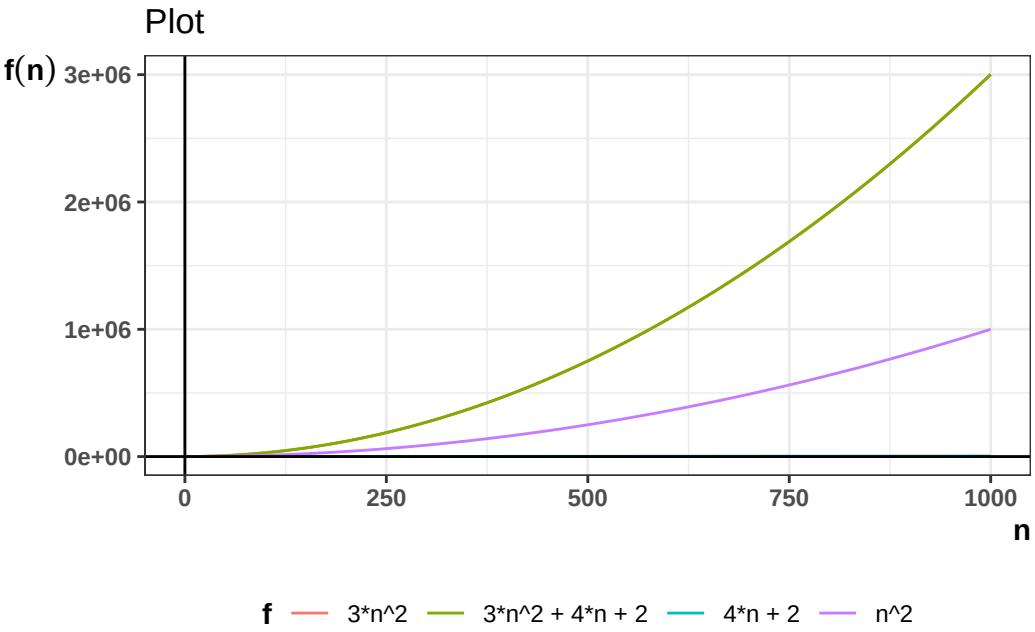
Definition 21.3 (Theta Notation)**.** A non-negative function $f(n)$ is in $\Theta(g(n))$ if and only if $f(n) \in (O(g(n)) \cap \Omega(g(n)))$.

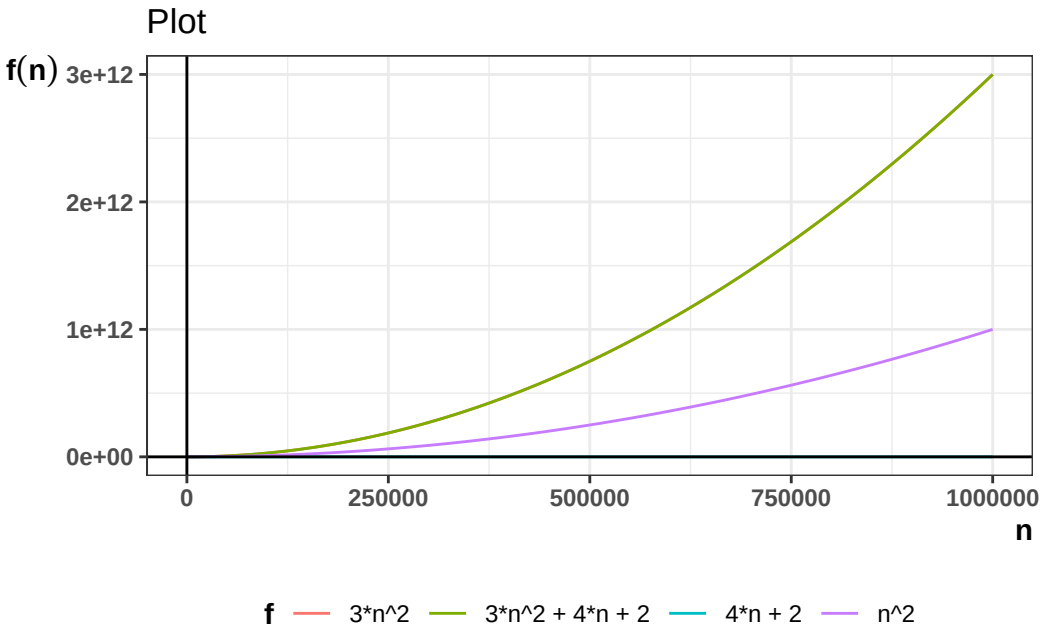
For example, if it is concluded that number of operations in an algorithm is given by the function $T(n) = 3n^2 + 4n + 2$ then $T(n) \in \Theta(n^2)$. Theta notation allows for stating complexity in simpler terms ignoring scaling and constant factors.

Below plots show how the n^2 term dominates as n grows larger.



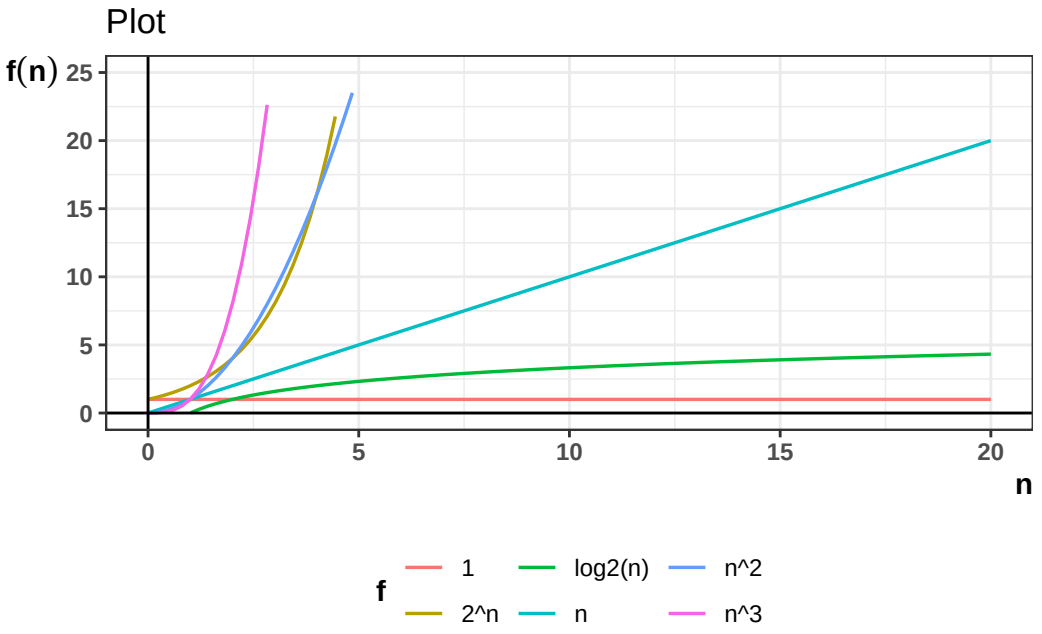

Plot
f(n)
30000
20000
10000
0
0
25
50
75
100
n
f
3*n^2
3*n^2 + 4*n + 2
4*n + 2
n^2


Plot
f(n)
3e+06
2e+06
1e+06
0e+00
0
250
500
750
1000
n
f
3*n^2
3*n^2 + 4*n + 2
4*n + 2
n^2



In terms of efficiency of algorithms it is often desired to keep the complexity low. Below are the common functions which complexity takes in increasing order. Anything below linear is considered good.

Constant	Logarithmic	Linear	Quadratic	Polynomial	Exponential
$\Theta(1)$	$\Theta(log(n))$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^c)$	$2^{\Theta(n^c)}$



21.3 Interfaces

Interface provides the abstract or theoretical requirements of storing and operating on certain type of data. For example, tuple, list or dictionary is used depending on the use case. Abstraction and generalization of use cases is interface. Tuple, list and dictionary are the implementation or data structures.

Interfaces are also referred to as

- Application programming interface (**API**): by developers
- Abstract data type (**ADT**): by computer scientists

There are 2 main types of interfaces.

- **sequence**
  - **extrinsic order**: order of items provided externally is preserved
  - special sequence types
    - **stack**: insert last and delete last (LIFO: last in first out)
    - **queue**: insert last and delete first (FIFO: first in first out)
    - **dequeue** (aka deque, double-ended queue)
- **mapping** (also referred to as **set**, **associative array**)
  - **intrinsic order**: items identified by unique keys, keys can optionally be stored in order
  - types of mapping types
    - **set**: items are keys themselves
    - **dictionary**: items are associated with keys

21.3.1 Operations

Operations on major interfaces can be categorized into 3 basic types

- **container operations**: operations on container itself
  - e.g. build, length
- **static operations**: operations on elements that do not alter the container
  - e.g. search (query) operations on elements by index (sequence) or value (mapping)
- **dynamic operations**: operations on elements that alter the container itself
  - e.g. inserting, deleting elements

Below tables summarize the common operation specifications for sequence and mapping interfaces.

21.3.1.1 Sequence interface

Category	Method	Description
Container	build(I)	build a sequence from items in iterable
	len(x)	return number of items
Static	iter_seq(x)	return stored items one at a time in sequence order
	get_at(i)	return the item at index i
	set_at(i, x)	replace the item at index i with x
Dynamic	insert_at(i, x)	add x as i^th item
	delete_at(i)	remove and return i^th item
	insert_first(x)	add x as the first item
	delete_first()	remove and return 1^st item
	insert_last(x)	add x as the last item
	delete_last()	remove and return last item

21.3.1.2 Map interface

Category	Method	Description
Container	build(I)	build a sequence from items in iterable
	len(x)	return number of items
Static	find(k)	return stored item with key k
Dynamic	insert(x)	add x to set with key x.key, replace if key exists
	delete(k)	remove and return the item with key k
Order	iter_order()	return the stored items one by one in key order
	find_min()	return the item with smallest key
	find_max()	return the item with largest key
	find_next(k)	return the item with key larger than k
	find_prev(k)	return the item with key smaller than k

21.4 Data structures

Data structures are the actual implementations of interfaces. There is no single data structure that solves all operations required in an interface efficiently. Different data structures solve a subset of operations efficiently.

Below are some basic categories of data structures with examples.

- *array based*: **static array**, **dynamic array**
- *pointer based*: **linked list** (w[/o] tail), **doubly linked list** (w[/o] tail)
- **hash table** (*array and pointer mixed*)
- *tree based*: **binary search tree**, **AVL tree**, ...
- *graph based*

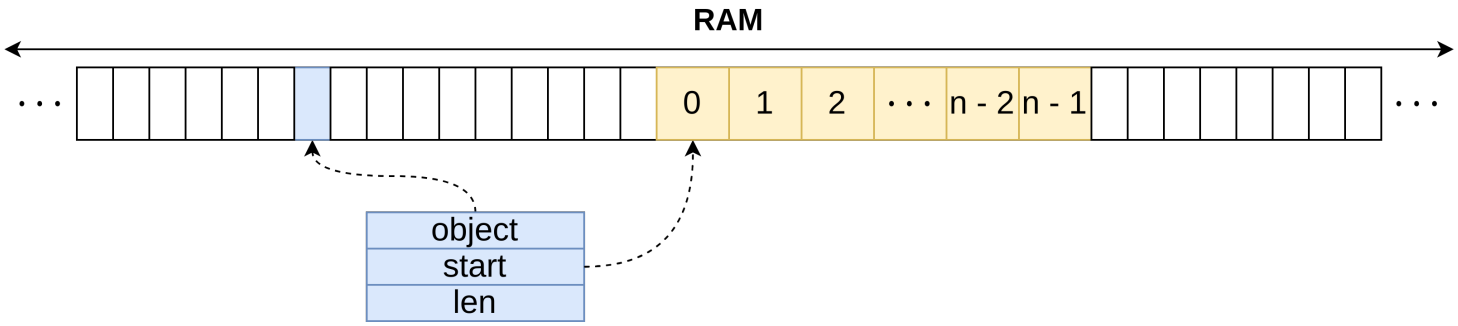
21.4.1 Static array

Random access memory (RAM) of computer hardware is like a giant continuous slots of memory addresses.

A static array finds and stores data in contiguous slots of equal size, depending on the size of data elements. The order is extrinsic, i.e. the items are stored in the order provided.

The static array object then needs to contain only the start memory address and length of the array.

- static operations: constant time ($\Theta(1)$)
  - memory address(i) = memory address(0) + i * item storage size
- dynamic operations: linear time ($\Theta(n)$)



Python tuples are approximately static array, one main difference is tuples are immutable, i.e. modification operations are not supported, a new object is created on modification.

Below are tables for performance of static array for sequence and map operations. Static arrays are efficient for sequences with static operations only.

For dynamic operations, in the worst case new contiguous slot of free memory has to be looked and container has to be build, therefore it is $O(n)$ operations.

	Sequence interface operations - $O(\cdot)$				
	Build	Static	Dynamic		
Data Structure	build(X)	get_at(i) set_at(i)	insert_first(x) delete_first()	insert_last(x) delete_last()	insert_at(i, x) delete_at(i)
array	n	1	n	n	n

Note:

$\cdot_{(a)}$ implies amortized, $\cdot_{(e)}$ implies expected, h is height of the tree

	Map interface operations - $O(\cdot)$				
	Build	Static	Dynamic	Order	
Data Structure	build(X)	find(k)	insert(k) delete(k)	find_min() find_max()	find_prev(k) find_next(k)
array	n	n	n	n	n

Note:

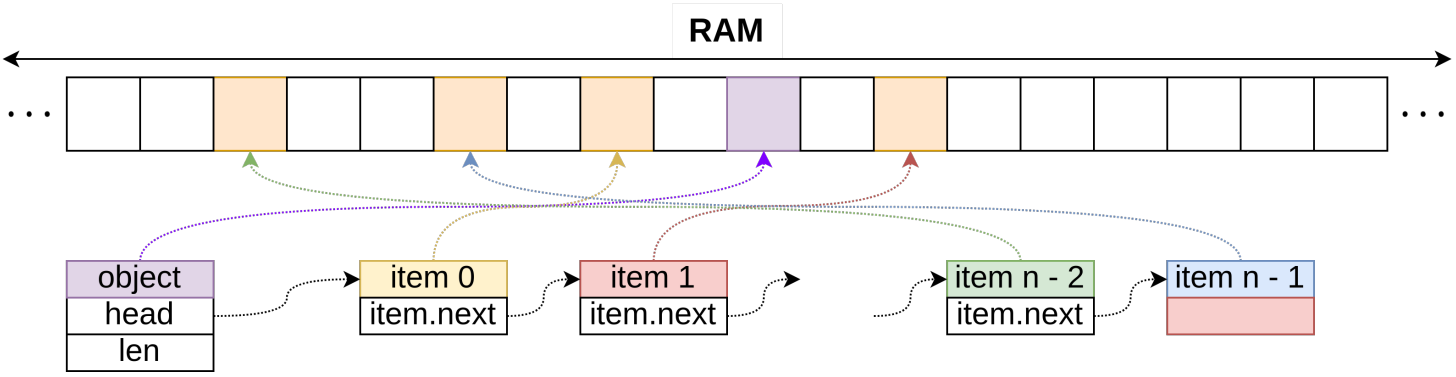
$\cdot_{(a)}$ implies amortized, $\cdot_{(e)}$ implies expected, h is height of the tree

21.4.2 Linked list

21.4.2.1 Singly linked list without tail

- random slots alloted
- object stores the
  - head memory address
  - len
- each node stores
  - item
  - link to next nodes memory address

- static operations: linear time ($\Theta(n)$)
- dynamic operations
  - insert/delete first: constant time ($\Theta(1)$)
  - other: linear time ($\Theta(n)$)



	Sequence interface operations - $O(\cdot)$				
	Build	Static	Dynamic		
Data Structure	build(X)	get_at(i) set_at(i)	insert_first(x) delete_first()	insert_last(x) delete_last()	insert_at(i, x) delete_at(i)
linked list	n	n	1	n	n

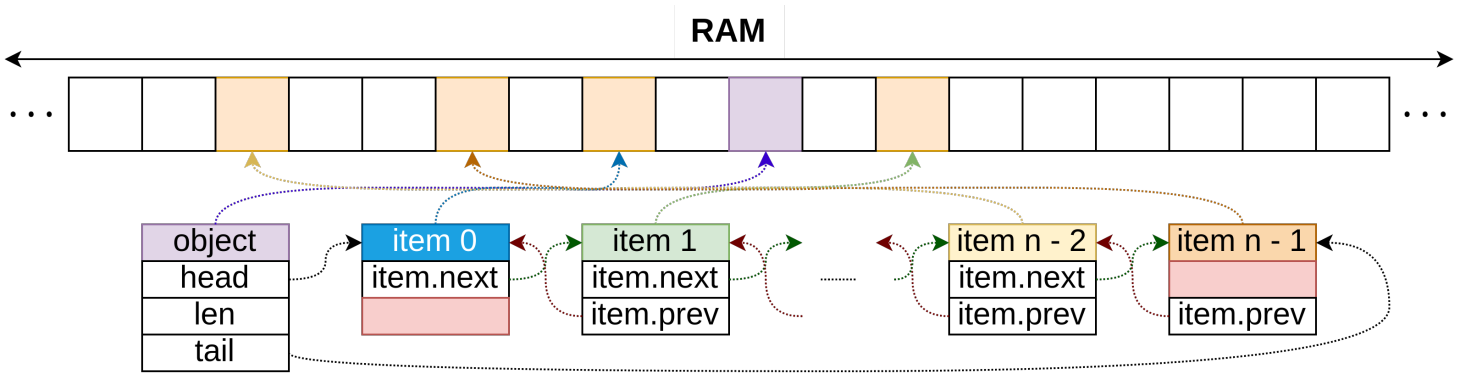
Note:

$\cdot_{(a)}$ implies amortized, $\cdot_{(e)}$ implies expected, h is height of the tree

- random slots alloted
- object stores the
  - head memory address
  - tail memory address
  - len
- each node stores
  - item
  - memory address of previous node
  - memory address of next node

- static operations: linear time ($\Theta(n)$)
- dynamic operations
  - insert/delete first: constant time ($\Theta(1)$)
  - insert/delete last: constant time ($\Theta(1)$)
  - other: linear time ($\Theta(n)$)
- good for implementing **stack** and **queue** with unknown length
  - insert/delete first/last
    aka push/pop/queue/dequeue

21.4.2.2 Doubly linked list with tail



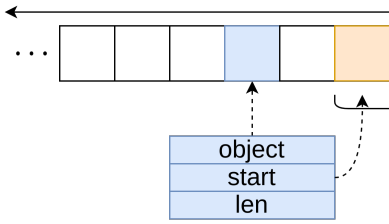
	Sequence interface operations - $O(\cdot)$				
	Build	Static	Dynamic		
Data Structure	build(X)	get_at(i) set_at(i)	insert_first(x) delete_first()	insert_last(x) delete_last()	insert_at(i, x) delete_at(i)
linked list	n	n	1	n	n
doubly linked list w/ tail	n	n	1	1	n

Note:

$\cdot_{(a)}$ implies amortized, $\cdot_{(e)}$ implies expected, h is height of the tree

21.4.3 Dynamic array

- Provides faster **insert/delete first/last** operations
- Extra space is alloted at beginning/end
- Idea is to **amortize the cost** of operations by providing extra space
  - when list is full, it is extended by a target **fill ratio**
  - when list is empty, the size is reduced to target **delete ratio**
- Python **list** is a dynamic array



- inserting or deleting at last position takes $O(1)$ time (amortized over n operations)

	Sequence interface operations - $O(\cdot)$				
	Build	Static	Dynamic		
Data Structure	build(X)	get_at(i) set_at(i)	insert_first(x) delete_first()	insert_last(x) delete_last()	insert_at(i, x) delete_at(i)
dynamic array	n	1	$1_{(a)}$	$1_{(a)}$	n

Note:

$\cdot_{(a)}$ implies amortized, $\cdot_{(e)}$ implies expected, h is height of the tree

21.4.4 Hash tables

21.4.4.1 Overview

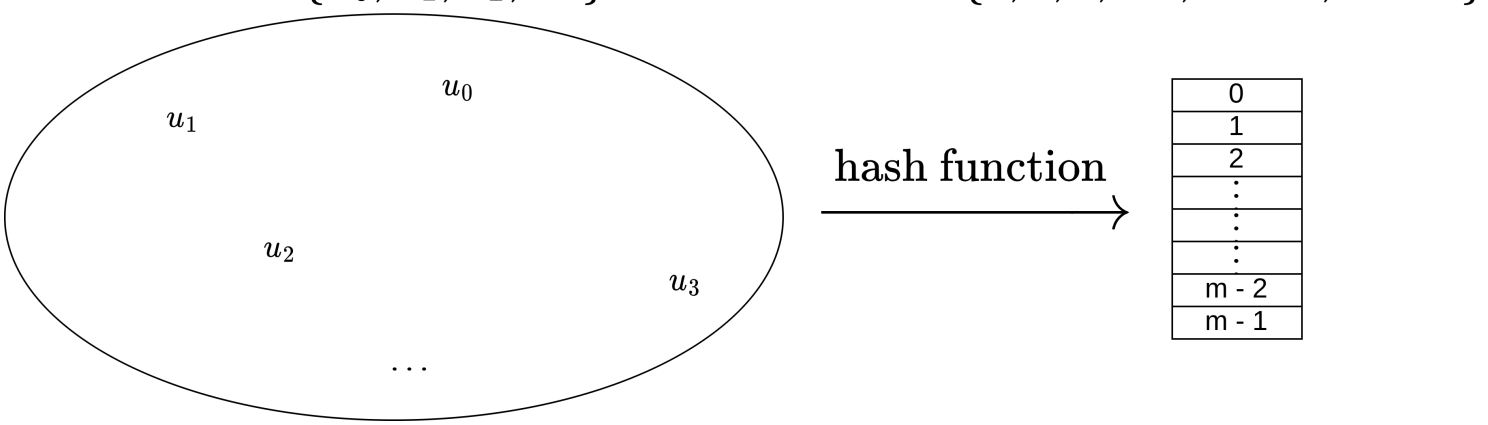
Hash tables are used to implement set interface: **sets** and **dictionaries**.

- Requirement: find items based on keys efficiently
  - arrays find item by looking up memory address based on index
  - in set interface the keys can be strings or other objects
  - search based on item value in array is not fast
- Key idea is to associate/map the keys with non negative integers (index)
  - integers map to memory address
  - given the key, find the associated integer with the key
  - given the integer, find the associated key
- Keys can be numeric, strings or other objects
  - generally, they must be unique

21.4.4.2 Hashing numbers

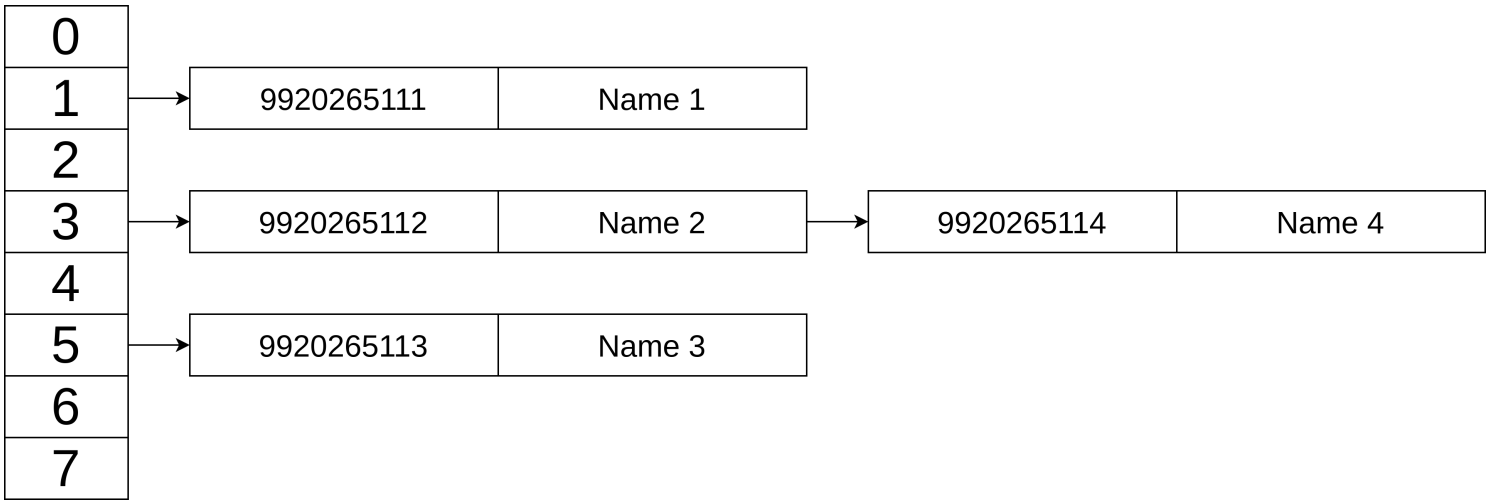
Taking example of a phone book where the requirements are

- store names and phone numbers
- search by phone number: *hashing numbers*
- search by name: *hashing strings*



- assuming phone numbers are 15 digit numbers
- u is the set of all possible 15 digit numbers
    - size of universe is $|u| = 10^{15} = 1,000,000,000,000,000$
    - not possible to store array of this size
    - this makes use of *direct access array* ($|m| = |u|$) impractical
- let n be the set of numbers to be stored $|n| = ?$, e.g. $|n| \leq 1000$
- m is the array of reduced size that you can store
    - choose $|m|$ such that $|n| \leq |m|$
    - size of the array is $|m|$, assume $|m| = 1000$
- *hash function* maps the key from $u \rightarrow m$
- *hash table* is the map of key from $u \rightarrow m$
- to search any key takes $\Theta(1)$ time
    - convert the key to array index using hash function
    - access the returned index

21.4.4.3 Collisions and chaining



- assuming $n = 4$ contacts have to be stored in array of size $|m| = 7$
- since $|u| >>>> |m|$ ($|u| = 10^{15}$) there will be collisions
    - *collisions*: hash function will give same integer for multiple inputs $h(k_1) = h(k_2)$, where $k_1 \neq k_2$
    - this is by *pigeon hole principle*

- there are 2 methods to resolve the issue of collisions
    - *chaining*: store collisions in a *linked list*
    - *open addressing*: store collisions in next free slot

21.4.4.4 Hashing strings

- convert characters into integers using ascii or unicode
- there are many methods to combine the codes to form an integer
- then rest of the problem is similar to hashing numbers
- there are many good solutions available
- to maintain the phone book by both name and number use 2 hash tables
  one hash table with hash of numbers, other with hash of strings

21.4.4.5 Key requirements for a good hash function

- **deterministic**: returns same integer for a given key always
- **fast** to compute: $\Theta(1)$
- keep the array size ($|m|$) low to **minimize space**
- keep number of **collisions low**

21.4.4.6 Use cases

- Hash tables have revolutionized searching and can be found everywhere in technology
- Sets and dictionaries are generally implemented using hash table
    - Python `dict` and `set`
- Language interpreter does fast lookup for keywords
- Contact list in phones
- Web and other text searches use hashing
- Applications use hash table for configuration files to search settings

21.4.5 Summary

21.4.5.1 Sequence interface

	Sequence interface operations - $O(\cdot)$				
	Build	Static	Dynamic		
Data Structure	build(X)	get_at(i) set_at(i)	insert_first(x) delete_first()	insert_last(x) delete_last()	insert_at(i, x) delete_at(i)
array	n	1	n	n	n
linked list	n	n	1	n	n
doubly linked list w/ tail	n	n	1	1	n
dynamic array	n	1	$1_{(a)}$	$1_{(a)}$	n
binary tree	n	h	h	h	h
avl tree	n	log n	log n	log n	log n

Note:

$\cdot_{(a)}$ implies amortized, $\cdot_{(e)}$ implies expected, h is height of the tree

21.4.5.2 Map interface

	Map interface operations - $O(\cdot)$				
	Build	Static	Dynamic	Order	
Data Structure	build(X)	find(k)	insert(k) delete(k)	find_min() find_max()	find_prev(k) find_next(k)
array	n	n	n	n	n
sorted array	n log n	log n (binary search)	n	1	log n
direct access array	u	1	1	u	u
hash tables	$n_{(e)}$	$1_{(e)}$	$1_{(e)(a)}$	n	n
binary tree	n log n	h	h	h	h
avl tree	n log n	log n	log n	log n	log n

Note:

$\cdot_{(a)}$ implies amortized, $\cdot_{(e)}$ implies expected, h is height of the tree

21.5 Algorithms

Algorithm is a procedure to solve a problem.

Study of algorithms involves study of finding correct and efficient procedures to solve problems.

Some classical algorithms are listed below along with their efficiency.

Application	Name	Performance	Comments
Search	Linear search	$O(n)$	• Iterative
	Binary search	$O(Log_2\ n)$	• Recursive • Works on sorted arrays
Sort	Insertion Sort	$O(n^2)$	• In-place • Unstable
	Selection Sort	$O(n^2)$	• In-place
	Merge Sort	$O(nLog_2\ n)$	• Creates new collection • Recursive

An algorithm is a solution to implementing operations like search and sort for a data structure.

An operation for a implementation of a data structure can use any of the compatible algorithms, but not all will be efficient. For example, for sorting an array merge sort is much quicker than most algorithms.

The algorithms provided here should also serve as mini projects to research and implement the algorithm. This should be a good practice to apply combining basic building blocks in previous sections.

22 Regular Expressions

Regular expressions, commonly referred to as "regex", are an abstract specification of creating patterns to do advanced search and replace operations on strings. They are like specification for a mini language for string (text) operations.

There are multiple implementations of this abstract specification with many common features but subtle differences for different use cases.

Python itself has multiple different flavors/implementations of regular expressions.

- Generic pattern matching in Python
  - `re` module in standard library: simple tutorial, exhaustive documentation
  - `regex` package in PyPI, extension of `re` module: homepage
- Unix style pathname pattern matching, used in bash commands
  - `glob` module
  - `pathlib` module in standard library provides `Path.glob(ptrn)`

The key idea is to provide much more powerful pattern searching than regular string methods provide. For example, to search all pdf files in a folder you can use `*.pdf` pattern to do a search on all file names. Or, `^project` pattern to find all files starting with the word project.

Regular expressions provide much more advanced features but are slower compared to regular string methods. The python tutorial has a section dedicated to describe the choice between using string methods and regular expressions.

Regular expressions can get very messy very fast. So, in the beginning, use only basics of regex, when it is clear that string method is not available or will be too complex. The most common situation is to use regex for path related operations.

23 Workflow

23.1 Overview

Designing workflow in context of programming can be split into 2 dimensions.

- **Program**: micro level, workflow while writing code
- **Project**: macro level, workflow of overall project

23.2 Program

While writing code there are 2 major dimensions.

- Editing tools: like prediction, linting etc.
    - these only help with basic checks and completion
    - these are managed at project level through editor settings
- Elements of code related to properties of a good program

This part of workflow design is related to the 2nd dimension and is dealt with refactoring.

23.2.1 Refactoring

Refactoring means reviewing and changing code to attain properties of a good program, without changing the actual output.

Recommended properties of a good program are: **readable**, **testable**, **modular**, **extensible**, **efficient**

Some practical aspects of writing programs are

- There are multiple ways to solve the same problem
- It is a cyclical process of writing code and refactoring
- More often than not, refactoring opportunities become apparent only while reviewing code

23.2.2 Recommendations

- Follow naming conventions
- Use doc strings
- Use and follow type annotations
    - make exceptions only if necessary
- Use comments where necessary
- Any task repeated more than a couple of times can be considered to be put into a function
- Functions should have minimum possible responsibilities, ideal is single responsibility
- Use appropriate data types

23.2.3 Sample workflow

Below is a workflow for writing code, to cover aspects that editing tools cannot cover.

- Step 1: focus on getting the the code to produce the correct result using recommended practices
- Step 2: review the code for opportunities for refactoring
- Step 3: refactor code
- Step 4: goto step 1

Note that the process given is an infinite recursion, the base case is when there is no further refactoring needed and it depends on size of the project, skills and experience. For small projects 2 to 3 recursions should be enough. For larger projects the requirements expand quickly.

23.3 Projects

Learning and practising project workflow management from the very start and for smallest of projects is recommended as it has many advantages like

- Workflow becomes operationally more
  - organized
  - efficient
- Reduces errors
- Allows more time on design and thought

Section 23.3.2 contains discussions on some key considerations while managing programming projects in general from a user of programming perspective. Python related solutions are discussed at respective places.

Section 23.3.3 illustrates a Python specific sample project structure which can work as a starter template.

Python documentation has a section dedicated to this for structuring Python projects specifically. Although Python documentation aims at developers who need to publish their packages on `PyPI`, still it gives good background to Python project management in general.

23.3.1 Tools: Settings

While programming, most of the interaction is with editor. Through editor all the underlying tools, like terminal, Python, git etc., are accessible.

Managing the tools and related settings, including extensions, is an essential part of project workflow.

VSCode related settings can easily be managed through use of workspaces and profiles.

Tasks in VSCode provide automation related to projects.

Sync can be used to keep the settings in the cloud, which makes it easy to switch between computers.

- VSCode: workspaces
- VSCode: profiles
- VSCode: tasks
- VSCode: sync

23.3.2 Components

23.3.2.1 Dependencies

There are 2 key dependencies of a Python project.

- Python version used: document using a `pyproject.toml` file
- External packages and their version used in the project
    - virtual environments provide solutions

23.3.2.1.1 Python version

Python, once installed, is machine and os independent. Python is available for most of the used computer system and operating system combinations.

If a project runs on a Python version on a pc then it will run on a different pc with a different architecture and operating system if the same Python version can be installed on it.

As one starts to use programming, recording the Python version should be sufficient, to be able to reproduce the project later. To be extra safe machine and operating system can be documented too if needed.

`pyproject.toml` file is used currently by Python developers for storing metadata about a Python project for creating and distributing packages, which can be used for storing Python version details and some other basic metadata about the project in a structured way. Note that the name is required to be `pyproject.toml` in case automation tools are used later as they check for a file with this specific name.

More details can be found at Python Docs: Packaging: `pyproject.toml`

23.3.2.1.2 External packages

As a user in programming most of the projects will use external modules and packages to find solution to a problem.

Dependencies and solutions related to this have been discussed in Section 18.4 related to virtual environments.

23.3.2.2 Documentation

Documentation is a critical part of any code project as it helps the author and users throughout the lifecycle of the project. The most common situation is when looking at the code written by self after some time becomes hard to understand. Documentation helps in this situation too.

There are 3 key areas of documentation.

- **Doc strings**: document functions, classes
- **Comments**: in the code itself to explain key concepts and logic applied
- `readme.md`: documentation for the project at high level

VSCode extension, autoDocstring - Python Docstring Generator can be used to assist in creating docstrings. Using such tools help use best practices evolved by experience of developers.

There are tools like `Sphinx` to automate parts of documentation of Python projects. These are generally needed for large coding projects.

One of the most important aspect is to structure the code so that it documents itself. For example naming objects, files and folders well so that they are self explanatory. Organizing function definitions and calls such that they self explain the flow of logic implemented. Giving thought to these aspects complements documentation.

23.3.2.3 Version control

Uses of version control systems has been discussed in Chapter 6. While using `git` below are some things that should be part of the workflow.

- Regularly maintain `.gitignore` file
  - `.py-venv` folder is large and need not be tracked as it can be restored using requirements file
  - anticipate and add directory and file patterns at the start of the project, it is inefficient to untrack files/folders later
- Do regular structured commits with helpful messages

23.3.3 Sample structure

- project root
  - **.git/**
  - **.py-venv/**
  - **docs/**
  - **src/**
    - `inputs.py`
    - `category_1_funcs.py`
    - `category_2_funcs.py`
  - `main.py` or `app.py`
  - `pyproject.toml`
  - `py-requirements.txt`
  - `readme.md`
  - `.gitignore`

Since the `main.py` (or `app.py`) can directly call functions from directories beneath it, e.g. `import src.category_1_funcs as <category_1>` and use functions as `<category_1>.<function_name>()`, there is no need for using packages with `__init__.py` for very small projects.

The only drawback is you cannot cross reference objects from files across folders, unless they are in flat hierarchy below.

Functions can further be put into subdirectories.

Part VI

Applications

Overview

Background

This section covers the *How?* part of the puzzle.

- ***Why** do you learn computer programming?* → **Use cases**
- ***What** is computer programming?* → **Theory**: **Building blocks**, **Architecture**, **Design**
- ***How** do you do computer programming?* → **Practical experience**: **Tools**, **Application**

Objectives

The key aim is to provide exposure to

- Application of theoretical concepts of specifications to solve practical problems
- Practice of writing programs
- Illustrate some common use cases

Use cases

In context of use cases, some of the basic areas of day to day operations are covered through projects to demonstrate the following benefits

- Automate repetitive manual tasks to
  - increases efficiency
  - reduces errors
- Create new solutions using programming, which standard tools do not offer

Process

As a user of programming for basic applications, some of the key steps are

- Finding relevant existing solutions from standard library or PyPI
- Understanding
  - how they are structured
  - what are the different ways they can be used
- Integrating them with the building blocks and architecture of Python to solve simple tasks
- For getting help on specific topic,
  - refer to the documentation
  - search the web
    - currently `stackoverflow` is a popular place for tech related QnA

Working with individual packages will serve as mini projects and which will be used in a major project.

Solutions are not provided for all projects. There are very less projects and most of them are small. Rest of the material provides solved examples. Consider these to be open book exam where main objective is to apply the concepts without seeing solutions to seal the concepts.

Some recommendations and reminders:

- Remember to refactor code
- Do not try to memorize everything, remember where to look for information. This is important because in programming there is a lot and it keeps changing at very fast pace compared to other areas.

24 Automation

24.1 Overview

24.1.1 Introduction

Programming languages, like Python, provide access to lower level system properties like file system and processes, through functions and classes. This leads to many opportunities to use these in multiple scenarios based on use case.

This is specially useful for situations where a certain task have to be repeated regularly. Doing it manually using applications with gui and mouse is slow and error-prone. Automation through programming not only makes the process faster and robust, but also provides opportunity to create new and custom solutions based on situation, which gui application might not be able to provide.

24.1.2 Use cases

Below are some examples to illustrate the usage.

- **Project templates**: Consider a situation where certain folder structure needs to be created regularly with timestamps and some prefilled content and structure in certain files. There is significant part of the process that can be automated.
- **System operations** like creating backups.
- **Pdf operations** like merging and splitting documents with custom index operations.
- and many more …

24.2 System Operations

24.2.1 Date time

Date time is a data structure required for many tasks related to programming. Python standard library provides datetime module for this and related operations.

It is very common to use a needed data structure implemented through standard library and external packages in `PyPI`.

- Documentation
    - Formatting

24.2.1.1 Exercise

Using Jupyter notebook

- read the system time
- create strings in following format
    - date stamp: "yyyy-mm-dd"
    - date-time stamp: "yyyy-mm-dd-h-m-s-ms"
    - time stamp: "h-m-s-ms"
- print time stamps

This is useful when creating any types of logs. Date or time stamps need to be inserted in names for files and folders or even text content into files.

24.2.2 Path manipulations

Python standard library provides pathlib and os.path modules for path manipulations. For most of the operations `pathlib` should be enough. If a solution does not exist in `pathlib`, then `os.path` can be used.

24.2.2.1 Mini project

Using Jupyter notebook

- Store source and destination paths in a variable
- Create a function that
    - takes source path as input
    - checks if source path is
        - empty: continue
        - non empty: prompt the user with message and confirm whether to continue
    - creates the below folder structure in the source path
        - .logs
        - sub_folder_1
            - `file_1.py`
        - sub_folder_2
            - `file_2.py`
        - `.gitignore`
        - `pyproject.toml`
- Create a function that
    - takes source and destination paths as input
    - checks if the destination path is
        - empty: continue
        - non empty: prompt the user with message and confirm whether to continue
    - copies the contents of source in destination without overwriting
        - checks if any file exists before copying
            - copies only if file in source is newer than in destination
                - hint: os.path has getmtime (last modified), getctime (created), getatime (last access)
- call the functions to test everything works

Notes:

- Before creating functions you can try small pieces to check how they work. This is typical of write-refactor cycles.
- This is an example of how to
    - create your own templates for creating boiler plate for repetitive projects and tasks
    - do basic file/folder operations like getting attributes and performing operations

24.2.3 File read/write

Python provides open as part of built-in functions to perform read/write operations on files.

For actual usage refer to the tutorial on Python i/o tutorial. Note the use of `with` statement which is an implementation of design pattern known as context managers.

There are multiple ways to write to file, as illustrated in the tutorial:

- write plain text to file
- write and read python objects through `json` or `pickle` package

For performing read/write operations on **csv**, **excel** and **database** formats:

- Resources
    - Python standard library: `csv`
    - `pandas`
- For most basic stuff csv module is sufficient

24.2.3.1 Mini-project

In the previous mini project, Section 24.2.2.1, add below functionalities using the work done in mini project, Section 24.2.1.1.

- create a current date time stamp
- create a file in the .logs folder with name as the current date time stamp
    - open the file and write details
        - number of files copied
        - number of files ignored
- copy the file to destination path .log folder
- **note**: creating and copying log file is dependent on order of placement in code, number of files will have to be adjusted for the new log file created. Plus care has to be taken to ensure copying the new log file to destination path.

24.2.4 Creating CLI's

Python scripts can be invoked from shell with Python CLI, e.g. `python -m <name>`. To create system utilities, it easier to pass input arguments directly from CLI rather than opening and running script to edit the inputs and then run the script.

For this there are multiple options available in Python.

- Basic: `sys.argv`
- Advanced: `argparse` package in standard library

- tutorial
- details

`argparse` is recommended as it is simple to use but can be extended to advanced use cases.

Once the script can take arguments, an alias can be added in `.bashrc` or `.bash_aliases` to refer to the python script. e.g. `alias cmd1="python3 -m ~/utils/cmd1.py"`. Then `cmd1` can be directly used from terminal like any other bash command, and inputs can be passed as well through cli.

Note that these projects cannot use Jupyter notebooks.

24.2.4.1 CLI Project - 1

Project templates

Create a python package which can be called from CLI using arguments to create a project template for a certain regular task at a specified path.

- use regular package for the project
- name can be for example `<create_data_analysis_template>`
- when called it will create a folder with certain predefined structure and files
    - the template structure can be stored within package folder and copied to any destination required
- input arguments from CLI
    - destination: required, `-d` or `--destination`
    - source: optional, `-s` or `--source`
        - if not provided use the default template folder path within package
    - add time stamp: `-t` or `--timestamp`, optional bool and defaults to true
        - when true: add date time stamp to the destination path name

Template folder can contain anything, actual files, folder structure and details will depend on the use case, but the design will be similar.

24.2.4.2 CLI Project - 2

Mirroring Software

Create a mirroring software implemented as a python cli package with the below features. The idea is to keep a mirror of a folder in a different location.

> Note: This is a bigger project so can be done at relaxed pace, in 2 to 4 weeks.

- cli arguments
    - source: optional with default value, `-s` or `--source`
    - destination: optional with default value, `-d` or `--destination`
    - verbosity: optional with default, `-v` or `--verbosity` with values from a list
    - dry run flag: optional bool defaulting to `false`, `-r` or `--dry-run`
- make 2 nested dictionaries for source and destination with below features
    - file path
    - file path if it exists in corresponding location else `None`
    - modified time for both source and destination if exists
    - flags
        - both dictionaries: file present in corresponding location

        - in source dictionary: modified time of source is greater than file in destination if present
- operations
    - source dictionary: file copied from source to destination if
        - file is not present in destination, create parent path as required
        - modified time in source is greater than in destination
    - destination dictionary: delete file from destination if it is not in source
        - delete the containing directory if the file is the last file

24.3 Documentation

Automating documentation can be most rewarding in all setups. The recommended tool for this is Quarto which is recommended because of reasons mentioned below. It is difficult to find all these features together in any other option at this time (2023).

- Easy to use, seems intimidating but for simple usage it is as easy as writing in Word or Powerpoint
- A lot of options in terms of document export (html, pdf, latex, word, ppt, etc.)
- Good default aesthetics in output with minimal configuration
- Good isolation of aesthetics, structure and content
- Support for table rendering is very advanced, using R packages like `huxtable`, `kableExtra`, `GT`
- Can be used for advanced usage
    - Can include automated calculations, tables, figures
    - Can include dynamic content in websites

This and many other websites and books are published using Quarto.

Some other alternatives are listed below for completeness.

- AsciiDoc
- Sphinx

25 Data Analysis

25.1 Background

Data analysis skill is a necessity in current time independent of the domain. Even if you do not do it yourself, once you learn, it will help understand what are the possible options. It is a skill that can be applied in any professional or personal setup.

For all domains knowing basic data analysis using programming is essential.

For engineering and science domains the requirements are higher but the first step is the same.

Therefore, learning to do basic data analysis using programming is a common subset.

The data science section tries to give a simple overview of everything related to data, which should help identify the common subset and how it fits in the broader context of data science.

The objective is to provide an overview and then point to resources for learning more. Data analysis in itself requires a separate course in itself which fits in overall learning path related to programming and its applications.

25.1.1 Data Science

25.1.1.1 Terminology

Since the field of data is relatively new at this time and evolving very fast, the terminology is not consistent. The reasons for this rapid change are

- advancement in computing capacities
- advancement in storage capacities
- advancement in scope of data collection

Below are some key terms with an attempt to define them.

Data science is the field of study which is dedicated to studying data, scientifically. This includes all aspects of the data cycle which includes several subjects related to each stage in cycle, which may require applying knowledge of several other supporting subjects like math, computer science (software and hardware), domain specific knowledge, process and operations, etc.

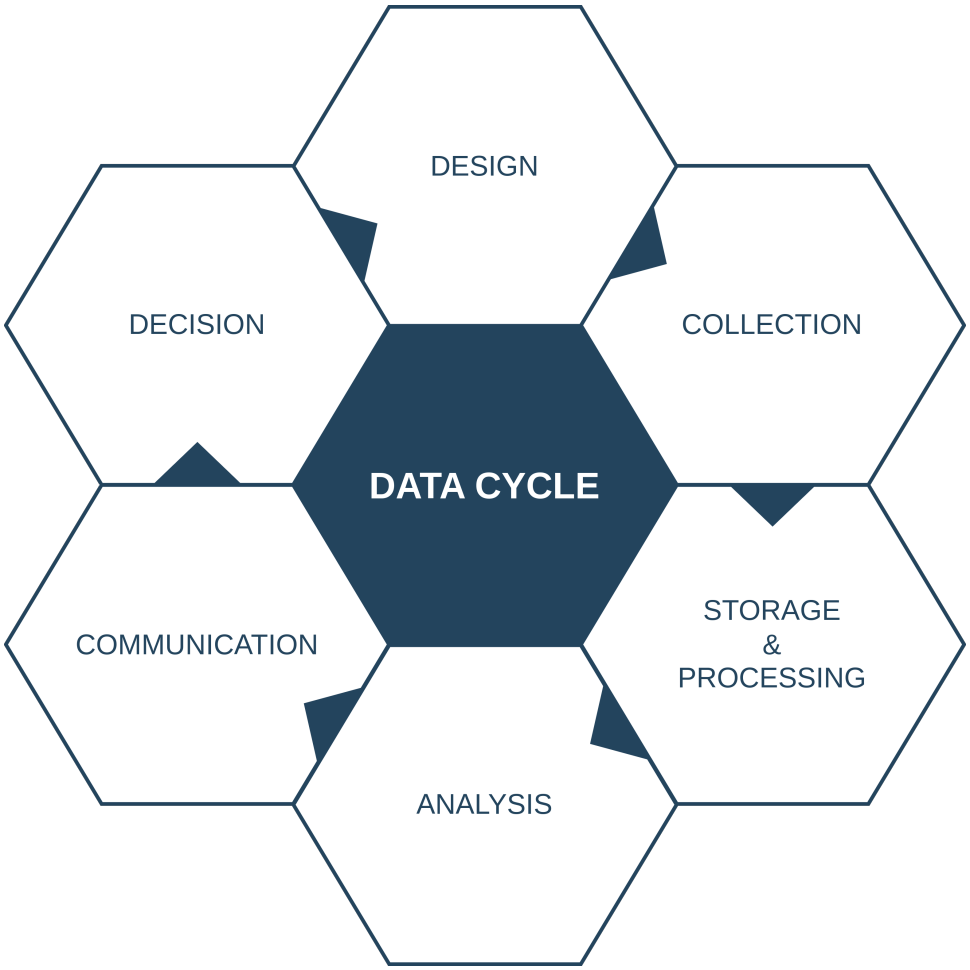
Data Analytics is the skill to apply advanced math techniques, like statistics, machine learning and artificial intelligence, to model data to do predictive and prescriptive analysis using data.

Data analysis is the skill to do basic static analysis using data which answers questions related to past and present. This involves very basic math.

25.1.1.2 Data cycle

Data science is a new and very vast field with no consistent formalization of structure. The data cycle presented in this section will try to summarize and give an overview of different components of data science.

Any project related to data, irrespective of scale or domain, has some core elements.



At small scale, one person might work on all elements. e.g. college projects requiring some analysis in which all stages can be performed using spreadsheets and programming.

At large scale, there might be multiple teams working on a single element. e.g. in a bank

- **Design**: All teams involved contribute to discussions
- **Collection**: All customers and several departments have access to corresponding applications which feed data to bank's database servers
- **Storage & Processing**:
    - There are multiple teams to perform validations and adjustments for loaded data coming in from different sources
    - There are separate teams for managing database servers
- **Analysis**: There are multiple teams across multiple functions analyzing data
- **Reporting**: There are multiple reporting teams
- **Decision**: Business teams

25.1.1.2.1 Design

Design stage involves using knowledge of all the components, the target is to capture every aspect preemptively. For a large recurring project it might mean involvement of a large number of teams specialized in specific areas of the the data cycle.

Every data related project starts with designing decision stage, framing good questions that drive the project, but then it is an iterative process.

25.1.1.2.2 Collection

- **Spreadsheets**
- **Web interface**
- **Desktop application**

25.1.1.2.3 Storage & Processing

- **Storage**: excel, text files, database
- **Processing**
  - Extract Transform Load (ETL) processes
  - pre processing
    - resolving duplicates
    - bad data values
    - missing data values

25.1.1.2.4 Analysis

- **Levels of analysis**
  - **Static**: answers simple questions using historical data
    - what has happened in the past?
    - what is the current state?
    - what is driving the current state?
  - **Predictive**: answers more complex questions using math (probability and statistics)
    - what would happen in future given a scenario?
    - static models
  - **Prescriptive**: answers more complex questions using math (probability and statistics)
    - predict various scenarios
    - answer, what should be done to ensure desired results?
    - static models, more advanced than predictive models
  - **Machine learning**, **Artificial intelligence**: come up with models based on data
    - models train themselves based on data
- **Operational Process**
  - Extract
  - Clean
  - Analyze
  - Model
- **Visualization** is part and key component of data analysis

25.1.1.2.5 Communication

- Components
  - tables
  - visualization
  - theory: how you arrived at the results
  - analysis: key findings
  - recommendations
- Tools: documentation using programming

25.1.1.2.6 Decision

- Check if results are satisfactory, if not re-design in next cycle
- If there are new questions or findings, re-design in next cycle

25.1.2 Learning paths

25.1.2.1 Data analysis

Data analysis is generic and applicable to all domains. It requires to learn the basic usage of

- Data storage technologies: spreadsheets, comma separated files, databases
- Data analysis using programming
    - import data in to a data structure on heap (RAM)
    - operate on data
    - visualization
    - documentation of analysis and results

Steps involved in data analysis using programming

- Import data from excel, csv or txt files, database
- Clean data
- Analyze data by performing operations and visualization
- Document analysis and results

25.1.2.2 Data analytics

Data analytics is relevant for science and engineering fields. Data analysis is a pre-requisite for this. Additionally it requires learning more about below topics for coming up with models for prediction, prescription or machine learning and artificial intelligence.

- Math: statistics, machine learning, neural networks, etc.
- Advanced knowledge of technologies used, including programming

25.2 Data analysis in Python

Data analysis in python is mostly supported through `pandas`, which is one of the most popular packages on `PyPI` for data science.

Below are some resources to get started with data analysis using python.

- pandas documentation
- Python4DS
- Python for Data Analysis, 3E

Note that the resources are meant for the STEM audience so not everything will be in scope for generic audience. Still some initial chapters that focus on basic data analysis should be useful.