MA1008 Introduction to Computational Thinking Cheat Sheet

Hankertrix

October 11, 2024

Contents

1 Definitions

1.1 Programming

Programming is the process of:

- Implementing a representation of the solution for the computer to execute
- Taking an algorithm and encoding it using certain programming language

1.2 Programming language

- A medium through which programmer may give instructions to a computer.
- It must support certain control constructs and data types needed to implement algorithms.

1.3 High-level programming languages

- Provide abstraction from the internal operating detail of the computer.
- Enable the programmers to focus on solving the problem.
- Make the process of developing the algorithm simpler

1.4 Microprocessor

The microprocessor in a computer is a digital device consisting of electronic components like transistors that operate in two states (either **on** or **off**).

1.4.1 The two states as voltage levels

- Low voltage (0 volt)
- High voltage (depends on the power supply used by the microprocessor)

1.4.2 The two states represented in binary

- 0 for low voltage
- 1 for high voltage

1.5 Datum

A datum is a number.

1.6 Byte

1 byte = 8 bits (2 hex digits)

1.7 Nibble

1 nibble = 4 bits (1 hex digit)

1.8 Word

1 word = 16 or 32 bits depending on CPU architecture (32 bits is used in this course)

1.9 DWord

1 DWord = 64 bits

1.10 Algorithms

- Algorithms are the method, procedure, or steps for solving a problem.
- They are the step-by-step instructions given to the computer on how to solve a given problem.

1.10.1 Expressing an algorithm

- Must be unambiguous, every step must be clear and precise
- Specify the order of steps precisely (sequence)
- Consider all possible decision points (branching and looping)
- Must terminate (no matter which representation is used)

1.11 Flow chart

A representation of an algorithm using diagrams for effective visualisation.

Name	**Use in flowchart**
Oval	Denotes the beginning or end of a program
Flow line (an arrow)	Denotes the direction of logic flow in a program
Parallelogram	Denotes either an input operation (INPUT) or an output operation (PRINT)
Rectangle	Denotes a process to be carried out, like addition
Diamond	Denotes a decision or branch to be made; the program should continue along one of two routes

1.12 Pseudocode

Directly uses informal English to describe an algorithm step by step with one step per line. It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading. Syntax doesn't matter in pseudocode.

1.12.1 Guidelines

- Write one statement per line only
- Capitalise the keywords
- Indent to show hierarchy
- End multi-line structures like loops and if statements
- Keep statements programming-language independent

1.13 Variable

A variable is like a labelled box that contains a value inside it. Each variable has its name, like `price_of_chicken_rice` and value, like `2.8`.

1.13.1 Why use variables?

- Reuse names instead of values, which is helpful in keeping track of useful information without needing to remember a bunch of numbers
- Easier to change or refactor the code later

1.14 Expressions

Expressions are anything that produces or returns a value. It usually consists of a combination of values (literals, variables, etc) and operations (operators, functions, etc).

1.14.1 Examples

- 3.14
- 100 * 15
- Result * 100

1.15 Assignment operator (=)

The assignment operator binds variables and values. The "=" sign is the **assignment operator**, not the equality in mathematics.

1.15.1 Syntax

Left-Hand Side (LHS) = Right-Hand Side (RHS)
This means:

- Evaluate the expression on the right-hand side
- Take the resulting value and assign it to the name (variable) on the left-hand side

1.16 Arithmetic operators (`+, -, *, /`)

- Used in common arithmetic
- Each arithmetic operator is a mathematical function that takes one or two operands and performs a calculation on them
- Most computer languages contain a set of such operators that can be used within equations to perform a number of types of sequential calculation

1.17 Identifiers

A name given to an entity in a programming language.

- Helps in differentiating one entity from another
- Name of the entity must be unique to be identified during the execution of the program

1.17.1 Attributes of identifiers in Python

- Uppercase and lower case letters A through Z
- The underscore "_"
- The digits 0 through 9
- However, the first character cannot be a digit
- Identifiers are case-sensitive

1.17.2 Naming conventions in Python

- Variables names should be in lowercase, with words separated by underscores as necessary to improve readability (`snake_case`).

1.18 Keywords

Keywords are special words that are reserved by a programming language. Programmers should not use keywords to name things.

1.19 Conditional statement

A conditional statement performs different actions depending on whether the condition evaluates to true or false.

1.20 Conditional expression (Boolean expression)

A conditional expression may be composed of a combination of the Boolean constants True or False, Boolean-typed variables, Boolean-valued operators, and Boolean-valued functions.

1.21 Nested `if` statements

A nested `if` statement is an `if` statement inside another `if` statement.

1.22 Relational operators (`==, !=, <, <=, >, >=`)

A relational operator compares two numbers (float or int) and returns a Boolean value of either True or False.

Relational Operator	Meaning	Example
==	Equal to	a == 1
!=	Not equal to	b != 2
<	Less than	c < 3
<=	Less than or equal to	d <= 4
>	Greater than	f > 5.0
>=	Greater than or equal to	g >= 6.0

1.23 Program execution and control flow

- Control flow controls which instruction should be executed next.
- By default, program instructions are executed one after another.
- However, some structures can alter the flow, like selection.
- Selection (branching) occurs when an "algorithm" makes a choice to do one of two or more things.
- The flow control in a program is, in essence, logic.
- When writing or reading a program, ensure that you could understand the flow, i.e., what should be executed next for every step.

1.24 Logical operators (Boolean operators)

Logical operators connect Boolean values and expressions and **return** a Boolean value as a result.

Operator	Example	Meaning
not	**not** number < 0	Change True to False, or vice versa
and	(num1 > num2) **and** (num2 > num3)	Return True only if **both** are True
or	(num1 > num2) **or** (num2 > num3)	Return True if **either** one is True

1.25 Looping

A computer program can dynamically choose how many times it repeats certain instructions during the program runtime.

1.25.1 General structure of a loop

1. **Initialise** the loop control variable.
2. **Test**: continue the loop or not?
3. **Loop body**: main computation being repeated.
4. **Update**: Modify the value of the loop control variable so that next time we test, we may exit the loop.

Sometimes, a loop may not have all of them, such as an infinite loop, where the test condition is always true.

1.25.2 Types of loops

1. Counter-controlled loop The number of repetitions can be **known** before the loop body starts. This loop just repeats the loop on each element in a preset sequence.
2. Sentinel-controlled loop The number of repetitions is **not known** before the loop body starts. Hence, a sentinel value that differs from normal data, like -1, is used to stop the loop.

1.26 Iteration

A one-time execution of a loop body is referred to as an iteration of the loop.

1.27 Nested loop

A nested loop is a loop inside another loop. An outer loop may enclose an inner loop.

1.28 Abstraction

- **Simplifies** things
- Identifies what is **important** without worrying too much about the details
- Allows us to **manage complexity**.

1.28.1 Why is abstraction important?

- A key element of computing is the complexity of the systems we build.
- Abstraction provides a means to distil what is essential, giving a manageable approach to create computational solutions.
- Abstractions are sometimes represented as **layers** or **hierarchies**, allowing us to view things at different degrees of detail.

1.29 String

A string is a sequence of characters. The sequence of characters is important and is maintained.

1.30 ASCII

- Uses 8 bits to store a character.
- $2^8 = 256$ different characters.

1.31 Unicode

- An extension of ASCII
- Able to include more characters
- Uses 16 bits to store a character
- $2^{16} = 65{,}536$ characters
- The Unicode space is divided into 17 planes.
- Each plane contains 65,536 code points (16-bit).
- Total of 1,114,112 characters, 96,000 used.

1.32 Parameters

Parameters are the variables names used in the function definition to hold the function inputs.

1.33 Arguments

Arguments are the actual values passed to the function when calling the function.

1.34 Function

- A function is a **piece of code** that performs some operation.
- The details are hidden (encapsulated) and only it's interface is exposed.
- It is a way to arrange a program to make it easier to understand.
- A function has arguments as inputs and may return one output.
- A function can have multiple `return` statements.
- The first executed `return` statement **ends the function**.
- Functions can also be called from other functions, and it works the same as users calling functions.
  - There is no limit to the "depth" of multiple function calls.
  - Deep function calls could make following the flow of a program difficult.

1.34.1 In mathematics

A function performs some operation and returns **one** value or thing.

1.34.2 In Python

Python functions "**encapsulates**" the performance of its particular operation, so they can be used by others.

- A function represents a single operation to be performed.
- A function takes zero or more arguments as input.
- A function returns one value or object as output.

1.34.3 Importance of functions

- Abstraction
- Divide-and-conquer problem-solving
- Reuse
- Sharing
- Security
- Simplification and readability

1.34.4 Principles of writing a function

- A function should only **do one thing**. If it does too many things, it should be broken down into multiple functions (refactored).
- A function should be **readable**. If you write it, it should be readable. Give comments when necessary.
- A function should be **reusable**. If it does one thing well, then when a similar situation (in another program) occurs, use it there as well.
- A function should be **complete**. A function should check for all the cases where it might be invoked. Check for potential errors.
- A function should **not be too long**. This is kind of synonymous with "**does one thing**". Use it as a measurement of doing too much.

1.35 Procedures

- Procedures are functions **without return statements**.
- In other words, they don't have an output.
- In Python, procedures will return `None`.
- Procedures are often used to perform some operation, like printing output, store a file, etc.
- A return statement is not always required in functions.

1.36 Method

- A method is a variation on a function.
- It represents a program and has input arguments and output.
- Unlike a function, it is applied in the context of a **particular object**.
- This is indicated by the **dot notation** invocation.

1.36.1 Method chaining

Methods can be chained together. For example:

```
string = "Python is cool!"
print(string.upper())
print(string.upper().find("C"))
```

Output:

```
PYTHON IS COOL!
10
```

1.37 Composite type

- Composite type is a data type which is constructed (composed) using primitive and other composite types.
- A composite type is basically a new data type that is made from existing ones.
- Some examples in Python include tuples, lists, dictionaries (hash maps in most other programming languages) and strings.

1.38 Data structures

- They have particular ways of storing data to make some operations easier or more efficient.
  - They are tuned for certain tasks, and they are often associated with algorithms.
- Different data structures have different characteristics.
  - One suited to solving a **certain problem** may not be suited for another problem.
- A few examples include arrays, linked lists, hash maps and trees.

1.38.1 Built-in data structures

Data structures that are so common that they are provided by most programming languages by default.

1.38.2 User-defined data structures

Data structures (classes in object-oriented programming) that are designed for a particular task.

1.39 Mutability

The ability to change.

1.40 Mutable

- After creation of the object, the object **can** be changed.
- **Lists** are mutable as you **can** change them after creating them.

1.41 Immutable

- After creation of the object, the object **cannot** be changed.
- **Strings** are immutable as you **cannot** change them after creating them.

1.42 Decomposition

Decomposition is the process of **breaking down** a complex problem into smaller manageable parts (sub-problems).

- Each sub-problem can then be examined or solved **individually**, as they are simpler to work with.
- It is a natural way to solve problems.
- It is also known as Divide-and-Conquer.

1.42.1 Importance of decomposition

1. Solve complex problems
    - If a complex problem is not decomposed, it is much harder to solve at once. Sub-problems are usually easy to tackle.
2. Enable collaboration and teamwork.
    - Each sub-problem can be solved by different parties.
3. Analysis
    - Decomposition forces you to analyse your problem from different aspects.

1.43 Divide-and-Conquer

1. Decompose a problem into several sub-problems.
2. Solve each sub-problem.
3. Compose the solution to the sub-problems.

Recursion naturally supports divide-and-conquer.

1.44 Recursive function

A recursive function is a function that invokes itself.

1.44.1 General form

- A recursive function is like a mathematical proof by induction, where you solve the problem for the base case, then solve the problem for the general until it reaches the base case.
- Generally, you will have a base case inside an if block where the function will return a value to stop the recursion.
- Then you will have the general case where the function will call itself on a new value.
- This general case will continue until the function reaches the base case and finally returns a value to stop the recursion.

1.44.2 Writing a recursive function

1. Determine the interface (signature) of the function
   - How many **parameters**? What are they?
   - What is the **return object**?
   - What is the **functionality** of the function?
2. Assume you have finished the implementation of the function
3. Develop the function body
   - Base case (Conquer) Solve the primitive case, and then return the result
   - Recursive step (Divide)
     - Decompose the problem into sub-problems (with the same structure)
     - Call the function to solve each sub-problem
     - Compose the final result from the sub-problems, and then return it.

1.44.3 Performance

A recursive function may be inefficient as it usually has redundant computation.

1.45 Binary tree

- A binary tree is a type of data structure that is made of nodes.
- It looks like an upside-down or inverted tree.
- The first node in the tree is called the root node, and there is only **one** of them
- The nodes that are connected to nodes below them are called **parent nodes**.
- The nodes that are only connected to nodes above them are called **leaf nodes**. These nodes have nothing after them and hence are like the leaves of an actual tree.
- Each node can only be connected to 2 nodes below them, hence the name binary tree.

1.46 Complete Binary Tree (CBT)

A complete binary tree is a binary tree where every parent node has **exactly two** child nodes.

1.47 Exceptions (Python-specific concept)

- Exceptions in Python can be thought of as **errors**.
- It usually means that the Python program has reached an **"exceptional"** situation that it **doesn't know** how to handle.

1.47.1 Why do we need exception handling?

- Most modern languages provide ways to deal with **"exceptional situations"**.
- Dealing with problems
- To try to capture certain situations or failures and deal with them gracefully.

1.47.2 What counts as an exception?

1. Errors
    - Indexing past the end of a list
    - Trying to open a non-existent file
    - Fetching a non-existent key from a dictionary, etc.
2. Events (not really errors)
    - Search algorithm doesn't find a value
    - Mail message arrives, queue event occurs

1.47.3 General idea

1. Keep **watching** a particular section of code.
2. If we get an exception, look for a catcher that can **handle** that kind of exception.
3. If **found** handle it.
4. Otherwise, let Python handle it (which usually halts the program).

1.48 Pattern

A pattern is a discernible regularity.

- The elements of a pattern repeat **predictably**.

In computational thinking, a pattern is the spotted **similarities** and **common differences** between problems.

1.49 Pattern recognition

Pattern recognition involves finding the similarities or patterns among small, decomposed problems, which can help in solving complex problems more efficiently.

1.49.1 Importance

- Patterns make problems simpler and easier to solve.
- Problems are easier to solve when they share patterns, as we can use the same problem-solving solution wherever the pattern exists.
- The more patterns we can find, the easier and quicker out problem-solving will be.

1.49.2 How to recognise patterns?

1. Identifying common elements or features in problems.
2. Identifying and interpreting common differences between problems.
3. Identifying individual elements within problems.
4. Describing patterns that have been identified.
5. Making predictions based on identified patterns.

1.50 Iterative accumulation

Iterative accumulation accumulate **target values** by iterating over them.

1.50.1 Important elements

1. Result variable to store the accumulation result.
2. A for loop.
3. A target value in each iteration to add to the result variable.

1.51 File

- A **collection of data** that is stored on **secondary storage**, like a disk.
- Accessing a file means establishing a **connection** between the **file** and the **program** and moving data between the two.
- When **opening** a file, you create a **file object** or **file stream** that is a connection between the file and the program.

1.51.1 Types of files

1. Text files
    - Organised as ASCII or Unicode characters
    - Generally human-readable, which is useful for certain file types
    - Text files are inefficient to store as each character takes up a few bits.
        - ASCII: 8 bits $\rightarrow$ 1 byte
        - Unicode: 32 bits $\rightarrow$ 4 bytes
2. Binary files
    - All the information is based on specific encodings
    - Not human-readable and contains non-readable information
    - It is a custom format that has more efficient storage

1.51.2 Current file position

- Every file maintains a **current file position**.
- It is the **current position** in the file and indicates what will be read next.
- It is set by the file mode.

1.52 File buffer

- When a file on the disk is opened, the contents of the file are **copied** into the **buffer** of the file object.
- The file object can be thought of as a very big list.
- The **current file position** is the **current index** to access the list.

1.53 Buffering

- Reading from and writing to a **disk** is **very slow**.
- Hence, a computer tries to read a lot of data from a file first.
  - If the data is needed, it will be "buffered" in the file object.
- The file object contains a copy of the information from the file, called a **cache**.
- The **file buffer** contains the information from the file and provides the information to the program, and it is located in the **file object**.

1.54 Sorting algorithms

- Sorting algorithms are algorithms that put elements in a list of a certain order.
- The most frequently used orders are **numerical** and **alphabetical orders**.
- Efficient sorting is important for optimising the efficiency of other algorithms (such as search and merge algorithms).
- Most of the primary sorting algorithms run on different space and time complexity.

1.54.1 Importance

- **Practical applications**: Sorting people by last name, countries by population, and websites by search engine relevance.
- **Sorting algorithms are fundamental to other algorithms**.

1.54.2 Trade-offs

- Different algorithms have different trade-offs.
- There is no single "best" sort for all scenarios.
- So, knowing just one way to sort is not enough.

1.55 Time complexity

Time complexity is defined to be the time the computer takes to run a program or algorithm.

1.56 Space complexity

Space complexity is defined to be the amount of memory the computer needs to run a program.

1.57 Bubble sort (Sinking sort)

- One of the simplest sorting algorithms.
- It repeatedly steps through the list to be sorted, compares each pair of adjacent items, and swaps them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list."
- It is easier to implement but slower than other sorts.

1.57.1 Overview

Bubble sort makes multiple passes through a list. For each pass, bubble sort goes through the steps below:

1. Compare the first two items in the list, and if the second item is smaller than the first, then the items are swapped.
2. Then move to the next item, compare the item and the item after it, and swap the two items if necessary.
3. Repeat the process.

1.57.2 After the first pass

Each sequence of comparison is called a pass. Once the first pass through the list has completed, the largest number has now been moved to the end of the list.

1.57.3 Start of the second pass

At the start of the second pass:

- The largest value is now in place, at the end of the list.
- There are (n - 1) items left to sort, which means there will be (n - 2) pairs.

1.57.4 Repeat the process

- Since each pass places the next largest value in place, the total number of passes necessary will be (n - 1).
- After completing the (n - 1) passes, the smallest items must be in the correct position with no further processing required.

1.58 Merge sort

- Merge sort is an example of a divide-and-conquer style of algorithm.
- A problem is repeatedly broken up into sub-problems, often using recursion, until they are small enough to be solved.
- The solutions are combined to solve the larger problem.
- Merge sort breaks the data into parts that can be sorted trivially, then combine those parts knowing that they are sorted.

1.58.1 Overview

1. Split the list into 2 parts, usually at the middle point.
2. Compare the first elements of both lists 1 by 1.
3. Move the smaller element out of the list that it was found in and add this value to the list of "sorted items".
4. Repeat the process until only a single list remains.
5. One list should still contain elements, which is sorted. Hence, the contents are moved into the result list.

1.59 Timsort

- Timsort is a hybrid sorting algorithm used by Python.
- It is derived from merge sort and insertion sort and is designed to perform well on many kinds of real-world data.
- It is invented by Tim Peters in 2002 for use in the Python programming language.
- It finds subsets of the data that are already ordered, and uses the subsets to sort the data more efficiently. This is done by merging an identified subset, called a run, with existing runs until certain criteria are fulfilled.
- Timsort has been Python's standard sorting algorithm since version 2.3.
- It is now also used to sort arrays in Java SE 7 and on the Android platform.

1.60 Searching

- Given a list of data, searching is finding the location of a particular value or reporting that the value is not present.
- It is one of the fundamental problems in computer science and programming.
- Sorting is done to make searching easier.
- There are multiple searching algorithms to solve problems.

1.61 Search key

Search key is basically the element that needs to be found in a search.

1.62 Linear search

- Linear search iterates over the sequence, one item at a time, until the specific item is found, or all items have been examined.
  - The approach is intuitive.
  - Starts at the first item.
  - Is it the one I am looking for?
  - If not, go to the next item.
  - Repeats until the item is found or all the items are checked.
- This approach is necessary if items are not sorted.

1.63 Binary search

- Binary search uses a divide-and-conquer strategy to search for an item, which divides the work in half with each step.
- However, the list of items must be sorted, otherwise this method of searching will not work.

1.63.1 Procedure

- Start at the middle of the list.
- Check if the middle item is what we are looking for.
- If it is not, check if the middle item is greater or lower than the item we are looking for.
- If it is lower, take the lower half of the list and look for the item using the same procedure above.
- If it is higher, take the higher half of the list and look for the item using the same procedure above.
- Repeat until the item is found, or the sublist is of size 0.

1.63.2 Binary search vs linear search

Algorithm	**Best-case time complexity**	**Average time complexity**	**Worst-case time complexity**	**Worst-case space complexity**
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)

Linear search	**Binary search**
Checks the item in the sequence until the desired item is found.	Checks the middle item of the list.
Often used for short lists.	Requires sorted list.
Inefficient for large sorted lists.	Repeated discarding of half of the list, which contains values that are all definitely larger or smaller than the desired value.
Simple and easy to implement, but inefficient compared to binary search.	Algorithms for binary search can be implemented in an iterative or recursive manner.

1.64 Searching algorithms

- Interpolation search
- Grover's algorithm, which requires quantum computers
- Indexed searching
- Binary search trees
- Hash table searching
- Best-first

1.65 Program complexity types

1.65.1 O(1): Constant complexity

The algorithm always uses the same amount of time to execute for all inputs.

1.65.2 O(n): Linear complexity

The algorithm's execution time increases linearly in proportion with the size of input date.

1.65.3 O(n^k): Polynomial complexity, where k is a constant

Polynomial complexity occurs for algorithm that contains nested loops.

- An example is O(n^2): Quadratic complexity, where the execution time is proportional to the square of the input data size.

1.65.4 O(log n): Logarithmic complexity

The algorithm's execution time grows as the log size of the input data.

1.65.5 O(k^n): Exponential complexity

Exponential complexity occurs for algorithm that contains recursive call.

1.66 Mainframe computers

- Mainframe computers have one machine to multiple users, and are used for applications that need to process massive amounts of data.
- Some examples include the IBM 1401 and the IBM z13.

1.67 Personal computers

- Personal computers have one machine to one user.
- Some examples include the IBM PC, the laptop and the tablet.

1.68 Mobile devices

- Mobile devices are very portable, and each user will likely have multiple devices.
- Some examples include a smartphone, a smartwatch, a health tracker and a credit card.

1.69 Ubiquitous computing

- Ubiquitous computing is when consumer electronic products and household appliances communicate with each other.
- Some examples include household appliances and a dash cam.

1.70 Internet of Things (IoT)

- The internet of things is when machines, devices and sensors are connected to the internet to exchange data, improve efficiency and reduce human error.

1.71 Cloud computing

- Cloud computing is computing in the massive servers held by big tech companies.
- They have remote computer servers accessible through the internet, called data centres.
- They provide computing resources such as to store, manage and process data.
- It is subscribed as an on-demand sharable service.
- It frees the user from maintaining the computer resources.
- Some examples include Microsoft Azure and Amazon Web Services (AWS).

1.72 Data centre

- A data centre collection of server machines at a premise.
- It provides computing resources that deal with big data, like Facebook's and Google's data centres.
- It is often used by cloud-service providers to provide cloud computing hosting services.
- Multiple data centres can be located at several geographic locations to ensure constant data availability during power outages and data centre failures.
- Cloud computing can be used to provide computing resources for IoT devices.
- However, this is not suitable for time critical applications due to network latency.

1.73 Fog computing

Fog computing refers to data processing at the network layer near the devices, such as the gateway equipment.

1.74 Edge computing

Edge computing refers to data being performed on the devices itself, or the client side.

1.75 Augmented reality

Augmented reality augments objects that reside in the real-world with computer-generated perceptual information.

1.76 Virtual reality.

- Virtual reality uses the computer technology to create a simulated environment.
- The user is immersed within the environment.

1.77 Artificial intelligence (AI)

- Artificial intelligence is to develop machines that can exhibit intelligence like humans.
- It trains itself through a machine learning algorithm.
- It is used in many applications, such as:
  - Face recognitions
  - Speech recognitions
  - Recommendation systems
  - Self-driving vehicles

2 Program translations

2.1 Interpretation approach

- Uses a program known as an interpreter
- Reads one high-level code statement at a time
  - Immediately translates and executes the statement before processing the next one

2.1.1 Examples

- Python
- R
- JavaScript
- TypeScript
- Lua
- Lisp

2.1.2 Benefits

- Very portable across different computing platforms
- Produces results almost immediately
- Easy to debug
- Program executes more slowly
- Useful for implementing dynamic, interactive features, such as those used on web pages

2.2 Compilation approach

- Uses a program called a compiler
- Reads and translates the entire high-level language program (source) code into its equivalent machine-language instructions in an executable file
- The resulting machine-language instructions can then be executed directly on the computer when the program is launched

2.2.1 Examples

- C
- C++
- Rust
- Go
- Zig
- Nim

2.2.2 Benefits

- Program runs very fast AFTER compilation
- Smaller in code size after compilation
- Must compile the entire program before execution
- Needs to be re-compiled if to be used on different computing platforms
- Used in large and sophisticated software applications when speed is of the utmost importance

2.3 Combination of compilation and interpretation

It is also possible to use the combination of both translation techniques.

2.3.1 Examples

- Java

3 Computer organisation

3.1 Central processing unit (CPU)

- Processes information
- Performs operations based on information given

3.1.1 Control unit (CU)

- Controls and coordinates the overall operation of the CPU
- Consists of decoders and logic circuits
- Controls the overall operations of the various units and modules
- Driven by a clock signal to ensure that everything happens at the correct instances and in proper sequence

3.1.2 Arithmetic/Logic Unit (ALU)

- Performs arithmetic operations as well as Boolean logic functions
- Deals with arithmetic operations like addition, subtraction, multiplication and division (if supported)
- Also performs logic operations (i.e. Boolean operations) like AND, OR, NOT, XOR etc, and bit shifting/rotation

3.1.3 Register Array

- Holds the various information used by CPU operations
- A small amount of very high speed internal storage used for frequently accessed data
- Enables data to be stored and retrieved quickly
- Some of these registers are used for more specific functions, like program counter, instruction register, stack pointer, memory address register, accumulator, etc

3.1.4 Program counter (PC) (special register)

Tells the control unit where to find the instruction in memory.

3.1.5 Instruction register (IR) (special register)

Holds the copy of the instruction to be decoded and executed by the control unit.

3.1.6 Data, Address and Control Signals

- Consists of signalling wires that are grouped into data signals, address signals, and control signals
- The signals connect the various internal functional units together
- Extend to the external system bus for other modules (memory and I/O) of the microprocessor

3.2 Memory

- Memory is used to store instructions and data for the CPU
- Consists of high speed electronics components that store the information in binary bit format
- Each location stores an 8-bit (byte) size data
- Each location is allocated a unique address
- Identified by specifying its binary pattern on the address bus
- If data is more than 8 bits in size (e.g. a 32-bit word), consecutive locations are used, and the lowest byte address is used to access the word location

3.2.1 Memory size

It is dependent on the number of bits (n) used in its address bus. Memory size = 2^n bytes.

- When $n = 10$, $2^{10} = 1024$ bytes or 1 kilobyte (KB)
- When $n = 20$, $2^{20} = 1024 \times 1$ KB $= 1048576$ bytes or 1 megabyte (MB)
- When $n = 30$, $2^{30} = 1$ KB $\times$ 1 KB $\times$ 1 KB $= 1073741824$ bytes or 1 Gigabyte (GB)

3.3 Input/Output (I/O) Interface

- Mechanism for transferring information to and from the outside world, such as to interact with users

3.4 System Bus

- The system bus consists of groups of parallel signals that are used to transfer information between the modules in the microprocessor.
- The system bus is used to connect external devices, such as memory and I/O devices, to the functional units within the CPU.

3.4.1 Data Bus

- Conveys the information from one module to the other.

3.4.2 Control Bus

- Provides the control signals for the modules to work together, such as to determine the direction of data flow, and when each device can access the data bus and address bus.

3.4.3 Address Bus

- Used to convey the address information. The signals' pattern on the address bus lines determines the location of the source and destination of the data transfer.

4 Information in a computer

4.1 Types of information

- Instructions
- Data

4.2 Information representation

4.2.1 Binary format (base-2)

- General expression: $\mathrm{b}_{\text{n-1}} \ldots \mathrm{b}_{\mathrm{k}} \ldots \mathrm{b}_2\mathrm{b}_1\mathrm{b}_0$
- Example of an 8-bit binary number: 01001101b
- The decimal (base-10) value can be calculated as: $2^{\text{n-1}} \times \mathrm{b}_{\text{n-1}} + \ldots + 2^{\mathrm{k}} \times \mathrm{b}_{\mathrm{k}} + \ldots + 2^2 \times \mathrm{b}_2 + 2^1 \times \mathrm{b}_1 + 2^0 \times \mathrm{b}_0$
- The k^{th} bit will have a weightage of 2^{k}.

4.2.2 Hexadecimal format (base-16)

- Hexadecimal symbols: 0, 1, 2, . . . , 8, 9, A (10), B (11), C (12), D (13), E (14), F (15)
- Each hexadecimal is 4 binary bits
- The binary format data is separated into groups of 4 bits: 1011 0101 0110 1100 = B56C

5 Program execution

5.1 Instruction execution

1. On power up, the program instruction will be typically first loaded into certain default memory locations (together with data).
2. A clocking signal is also applied to the CPU (as well as to the rest of the microprocessor system).
3. Based on the rising or falling edges of the clock, the control unit (CU) will retrieve (fetch) the instruction from the default memory location.
4. The control unit (CU) of a CPU is designed to recognise its own instructions and performs the corresponding operation.

5.2 Fetch

Fetch the instruction from the memory into the instruction register (IR) using the address indicated by the program counter (PC).

5.3 Decode

Decode the machine instruction by the control unit, which is now stored in the instruction register (IR).

5.4 Execute

Execute the instruction. For example, loading the operands into the arithmetic/logic unit (ALU) and get the ALU to operate on them.

6 Ways to handle errors

6.1 Look before you leap (LBYL)

- Be very cautious!
- Check **all aspects** before execution
  - If a string is required, check that it is a string.
  - If the values should be positive, check that it is indeed positive.
- This can make code quite lengthy, which can reduce readability.

6.1.1 Example

```
if not isinstance(string, str):
    return None
elif not string.isdigit():
    return None
else:
    return int(string)
```

6.2 Easier to Ask for Forgiveness than Permission (EAFP)

- Run anything you like!
- Be ready to **clean up** in case of error
- The `try` block reflects what you want to do, and the `except` block reflects what you want to do on error.
- Cleaner separation

6.2.1 Example

```
try:
    return int(string)
except (TypeError, ValueError, OverflowError):
    return None
```

7 Program complexity evaluation

- There are multiple possible algorithms as well as implementations.
- Execution time depends on many factors, like:
  - The speed of the computer
  - The way the algorithm is implemented
    - Pre-compute lookup table
    - Loop unrolling technique
    - Input data value and input data size

7.1 Instruction steps

- Count the number of instructions it takes to execute the algorithm.
- It is independent of the computer, and more steps mean longer execution time.
- The number of steps may still depend on the data involved in the computation.

7.2 Asymptotic behaviour

- It is more important to consider the worst case scenario, which is when an item is not in a list, as the number of instruction steps will be the most.
- As the number of entries in the list increases, the number of steps under the worst case scenario also increases.
- Input data size is equivalent to the number of entries in the list.
- In time complexity analysis, the growth pattern of the number of steps as input data size increases indefinitely.
- The asymptotic behaviour of running time is given by the Big O notation.

7.3 Big O notation

- It measures and compares the time complexity of algorithms.
- It is how the execution time of the algorithm grows as the size of the input data grows.
- The execution time is in terms of the number of instruction steps for the worst case scenario.
- The Big O notation gives an upper bound on the asymptotic growth of an algorithm.

7.4 Analysis of a linear search

7.4.1 Assumption

Each line of code can be executed in one step.

7.4.2 Worst case

Item isn't in the list

7.4.3 Asymptotic behaviour

- T(n) increases proportionally with n, i.e., T(n) doubles when n is doubled.
- Growth order: f(n) = n

7.4.4 Complexity using Big O notation [O(f(n)) = O(n)]

- Linear complexity

8 Complete Binary Tree (CBT) implementation in Python

A complete binary tree can be represented as a list in Python:

```
complete_binary_tree = [left_subtree, root, right_subtree]
example = [[[7], 1, [9]], 3, [[8], 2, [4]]]
```

8.1 Total number of nodes in a complete binary tree

```
def num_of_nodes(binary_tree: list) -> int:
    "A function to return the number of nodes in a complete binary tree"

    # Gets the length of the complete binary tree
    length = len(binary_tree)

    # If the length of the binary tree is 0 or 1,
    # return the length of the binary tree
    if length <= 1:
        return length

    # Otherwise
    else:

        # Get the number of nodes in the left subtree
        # Remember that the left subtree is the first item in the list
        num_of_nodes_in_left_subtree = num_of_nodes(binary_tree[0])

        # Get the number of nodes in the right subtree
        # Remember that the right subtree is the last item in the list
        num_of_nodes_in_right_subtree = num_of_nodes(binary_tree[-1])

        # Add the number of nodes in the left and right subtrees,
        # adding one because of the middle root node,
        # and return the value
        return (num_of_nodes_in_left_subtree + num_of_nodes_in_right_subtree + 1)
```

8.2 Sum of the node values in a complete binary tree

```python
def sum_of_node_values(binary_tree: list) -> int:
    "A function to return the sum of node values in a complete binary tree"

    # Gets the length of the complete binary tree
    length = len(binary_tree)

    # If the length of the binary tree is 0
    # return zero
    if length <= 0:
        return 0

    # Otherwise, if the length of the binary tree is 1
    # return the value of the root node
    elif length == 1:
        return binary_tree[0]

    # Otherwise
    else:

        # Get the sum of the node values in the left subtree
        # Remember that the left subtree is the first item in the list
        sum_of_node_values_in_left_subtree = sum_of_node_values(binary_tree[0])

        # Get the sum of the node values in the right subtree
        # Remember that the right subtree is the last item in the list
        sum_of_node_values_in_right_subtree = sum_of_node_values(binary_tree[-1])

        # Add the sum of the node values in the left and right subtrees,
        # as well as add the value of the root node,
        # which is the second item in the list
        # and return the value
        return (
            sum_of_node_values_in_left_subtree
            + sum_of_node_values_in_right_subtree
            + binary_tree[1]
        )
```

8.3 Obtain the highest value found in the complete binary tree

```
def get_max_value(binary_tree: list) -> int:
    "A function to return the highest value found in a complete binary tree."

    # Gets the length of the complete binary tree
    length = len(binary_tree)

    # If the length of the binary tree is 0, return 0
    if length <= 0:
        return 0

    # Otherwise, if the length of the binary tree is 1
    # return the value of the root node
    elif length == 1:
        return binary_tree[0]

    # Otherwise
    else:

        # Get the maximum values of the left and right subtrees
        max_value_left_subtree = get_max_value(binary_tree[0])
        max_value_right_subtree = get_max_value(binary_tree[-1])

        # Set the max value to the value of the root node
        max_value = binary_tree[1]

        # If the left subtree has a greater max value than the root node,
        # set the max value to the one from the left subtree
        if max_value_left_subtree > max_value:
            max_value = max_value_left_subtree

        # If the right subtree has a greater max value than the left subtree
        # or the root node, set the max value to the one from the right subtree
        if max_value_left_subtree > max_value:
            max_value = max_value_right_subtree

        # Return the max value
        return max_value
```

8.4 Obtain the lowest value found in the complete binary tree

```
def get_min_value(binary_tree: list) -> int:
    "A function to return the lowest value found in a complete binary tree."

    # Gets the length of the complete binary tree
    length = len(binary_tree)

    # If the length of the binary tree is 0, return 0
    if length <= 0:
        return 0

    # Otherwise, if the length of the binary tree is 1
    # return the value of the root node
    elif length == 1:
        return binary_tree[0]

    # Otherwise
    else:

        # Get the minimum values of the left and right subtrees
        min_value_left_subtree = get_min_value(binary_tree[0])
        min_value_right_subtree = get_min_value(binary_tree[-1])

        # Set the minimum value to the value of the root node
        min_value = binary_tree[1]

        # If the left subtree has a smaller minimum value than the root node,
        # set the min value to the one from the left subtree
        if min_value_left_subtree < min_value:
            min_value = min_value_left_subtree

        # If the right subtree has a smaller minimum value than the left subtree
        # or the root node, set the minimum value to the one from the right subtree
        if min_value_left_subtree < min_value:
            min_value = min_value_right_subtree

        # Return the min value
        return min_value
```

8.5 Mirroring a complete binary tree

```python
def mirror_binary_tree(binary_tree: list) -> list:
    "Function to mirror a complete binary tree"

    # If the length of the binary tree is 0 or 1,
    # return the binary tree itself
    if len(binary_tree) <= 1:
        return binary_tree

    # Otherwise
    else:

        # Set the parent node to the root node of the binary tree
        parent_node = binary_tree[1]

        # Gets the mirrored version of the left and right subtrees
        mirrored_left_subtree = mirror_binary_tree(binary_tree[0])
        mirrored_right_subtree = mirror_binary_tree(binary_tree[-1])

        # Returns the mirrored binary tree
        return [mirrored_left_subtree, parent_node, mirrored_right_subtree]
```

8.6 Printing a complete binary tree

```python
def print_binary_tree(binary_tree: list, depth: int) -> None:
    """
    Function to print a complete binary tree.

    The depth represents how deep a node is in the binary tree,
    and affects how indented the value should be when printed.

    The binary tree is printed with the root node at the leftmost side
    of the screen, and the left subtree BELOW the root node
    and the right subtree ABOVE the root node.
    """

    # If the length of the binary tree is 0
    # don't print anything and exit the function
    if len(binary_tree) <= 0:
        return

    # If the length of the binary tree is 1
    elif len(binary_tree) == 1:

        # Print 2 spaces x the depth of the node, which is the indent,
        # before the printing the only value of the binary tree.
        # There is no need for a separator between the indent and the value
        print("  " * depth, binary_tree[0], sep="")

    # Otherwise
    else:

        # Print the right subtree of the binary tree first,
        # as it is at the top, increasing the depth by 1
        print_binary_tree(binary_tree[-1], depth + 1)

        # Print the value of the parent node
        print("  " * depth, binary_tree[0], sep="")

        # Print the left subtree of the binary tree last,
        # as it is at the bottom, increasing the depth by 1
        print_binary_tree(binary_tree[0], depth + 1)
```

9 Sorting algorithm implementation in Python

9.1 Bubble sort

This implementation **modifies** the list and **does not return** any value.

```python
def bubble_sort(list_of_items: list) -> None:
    "A function to sort a list of items using bubble sort."

    # Length of the list of items
    n = len(list_of_items)

    # Iterates from the first item to the second last item of the list
    for pass_number in range(n - 1):

        # Intialise the swapped variable to False
        swapped = False

        # Iterates over the items in the shortened list
        # The items at the back of the list is sorted and the
        # number of items at that back that are sorted
        # depends on the number of passes, so this makes the sort more efficient.
        for i in range(n - pass_number - 1):

            # Get the current item and the next item
            current_item = list_of_items[i]
            next_item = list_of_items[i+1]

            # If the current item is greater than the next item
            if current_item > next_item:

                # Swap the items and set the swapped variable to True
                list_of_items[i] = next_item
                list_of_items[i+1] = current_item
                swapped = True

        # If no swaps have been performed in the inner for loop,
        # break out of the loop as the entire list is sorted.
        # This is also to improve the efficiency of the sort.
        if not swapped:
            break
```

9.2 Merge sort

This implementation **returns a sorted list** of the items and **does not modify** the original list.

9.2.1 Merge function (used in the merge sort function)

```
def merge(left_list: list, right_list: list) -> list:
    "A function to merge the left list and the right list for merge sort."

    # Gets the length of the sorted list
    sorted_list_length = len(left_list) + len(right_list)

    # Initialise a list of Nones with the length of the sorted list
    sorted_list = [None] * sorted_list_length

    # Initialise the iterating variables
    left_list_index, right_list_index, sorted_list_index = (0, 0, 0)

    # While there are items in both the left and right lists
    while left_list_index < len(left_list) and right_list_index < len(right_list):

        # If the value of the left item is less than the right item,
        # put the value from the left list into the sorted list
        # and increment the index of the left list by 1
        if left_list[left_list_index] < right_list[right_list_index]:
            sorted_list[sorted_list_index] = left_list[left_list_index]
            left_list_index += 1

        # Otherwise, put the value from the left list into the sorted list
        # and increment the index of the left list by 1
        else:
            sorted_list[sorted_list_index] = right_list[right_list_index]
            right_list_index += 1

        # Always increment the index of the sorted list by 1
        sorted_list_index += 1

    # Continued on the next page...
    # The following blocks of code are NOT inside the while loop above.
    # They are in the main block of the function,
    # which is the same indentation level
    # as the "Continued on the next page..." comment

    # If there are still items in the left list
    while left_list_index < len(left_list):

        # Add the item to the sorted list
        sorted_list[sorted_list_index] = left_list[left_list_index]

        # Increment the indexes for both lists by 1
        sorted_list_index += 1
        left_list_index += 1

    # If there are still items in the right list
    while right_list_index < len(right_list):

        # Add the item to the sorted list
        sorted_list[sorted_list_index] = right_list[right_list_index]

        # Increment the indexes for both lists by 1
        sorted_list_index += 1
        right_list_index += 1

    # Return the sorted list
    return sorted_list
```

9.2.2 Merge sort function (the actual sorting function)

This is **the function to use** to sort a list using the merge sort algorithm.

```python
def merge_sort(list_of_items: list) -> list:
    "A function to sort the list of items using merge sort."

    # Get the length of the list
    length = len(list_of_items)

    # If the length of the list is less than 2, then return the list of items.
    # This is the base case.
    if length < 2:
        return list_of_items

    # Get the middle of the list.
    # Use the floor division operator to get an integer for lists of odd length.
    middle = length // 2

    # Split the list into 2, and get the left and right list.
    left_list = list_of_items[:middle]
    right_list = list_of_items[middle:]

    # Recursively sort the left and right list
    sorted_left_list = merge_sort(left_list)
    sorted_right_list = merge_sort(right_list)

    # Merge the two lists together and return the sorted list
    return merge(sorted_left_list, sorted_right_list)
```

10 Binary search implementation in Python

10.1 Iterative binary search

```python
def iterative_binary_search(sorted_items: list | tuple, target) -> bool:
    "Function to binary search a sorted collection of items using a loop."

    # Get the index of the lowest item and the highest item
    lowest_item_index = 0
    highest_item_index = len(sorted_items) - 1

    # Iterate while the lowest item index is less than the highest item index
    while lowest_item_index < highest_item_index:

        # Get the middle of the list
        middle_item_index = (lowest_item_index + highest_item_index) // 2

        # Get the middle item of the list
        middle_item = sorted_items[middle_item_index]

        # If the middle item of the list is the target value, return True
        if middle_item == target:
            return True

        # Otherwise, if the target value is less than the middle item,
        # set the highest item index to the middle item index - 1
        elif target < middle_item:
            highest_item_index = middle_item_index - 1

        # Otherwise, if the target value is more than the middle item,
        # set the lowest item index to the middle item index + 1
        else:
            lowest_item_index = middle_item_index + 1

    # If the item is still not found, then return False
    return False
```

10.2 Recursive binary search

```python
def recursive_binary_search(sorted_items: list | tuple, target) -> bool:
    "Function to binary search a sorted collection of items using recursion."

    # Get the index of the lowest item and the highest item
    lowest_item_index = 0
    highest_item_index = len(sorted_items) - 1

    # If the index of the highest item is less than or equal to
    # the index of the lowest item, return False
    if highest_item_index <= lowest_item_index:
        return False

    # Gets the middle item index
    middle_item_index = (lowest_item_index + highest_item_index) // 2

    # Gets the middle item
    middle_item = sorted_items[middle_item_index]

    # If the middle item is the target value, return True
    if middle_item == target:
        return True

    # Otherwise, if the target value is less than the middle item,
    # call the function with the sorted list being truncated to
    # the list before the middle item and return the result
    elif target < middle_item:
        return recursive_binary_search(sorted_items[:middle_item_index], target)

    # Otherwise, if the target value is more than the middle item,
    # call the function with the sorted list being truncated to
    # the list after the middle item and return the result
    else:
        return recursive_binary_search(sorted_items[middle_item_index + 1:], target)
```

11 Data types in Python

Python uses duck-typing to figure out the type of a variable.

- Python does not have variable declaration, like Java or C, to announce or create a variable.
- A variable is created by just assigning a value to it and the type of the value defines the type of the variable.
- If another value is re-assigned to the variable, its type can change.

11.1 Why differentiate between data types?

1. The underlying representations for different data types are different.
2. Different data types support different operations, and for these operations to work, we need to supply them with variables of the correct types.
3. The objects we wish to represent in a computer program are of different types in nature and require different data types.

11.2 String (`str`)

- It is a sequence, typically a sequence of characters delimited by single quotes `'` or double quotes `"`.
- The sequence of characters is important and is maintained.
- Use either single or double quotes to create a string.
- Do not use both single and double quotes to create a string.
- Escape a quote by using the backslash character "\".

11.2.1 Index (`[]`)

- Characters in a string are in a sequence
- We can identify each character with a unique index (a position in the sequence).
- We can index a character from either end of the sequence.
  - Non-negative values: counting from left, starting at 0
  - Negative values: counting from right, starting at -1
- The index operator is always at the end of the expression and is preceded by something, either a variable or a sequence.

We can use `[]` to access particular characters in a string.

```
string = "Hello World"
print(string[0])
print(string[1])
print(string[-2])
print(string)
print("omg wow"[4])
```

Output:

```
H
e
l
Hello World
w
```

11.2.2 Slice ([:])

`[start : end : step]`

- `start` is the index of the start of the subsequence.
- `end` is the index of the end of a subsequence (not included).
- `step` specifies the step size to jump along the sequence.

```
string = "Hello World"
print(string[1:4])
print(string[::2])
print(string[::-2])
print(string[0:14:-1])
print("omg wow"[2:6])
```

Output:

```
ell
HloWrd
drWolH

g wo
```

11.2.3 Length (len)

```
string = "Hello World"
print(len(string))
```

Output:

```
11
```

11.2.4 Concatenation (+)

```
string = "Hello World"
print(string + "!")
```

Output:

```
Hello World!
```

11.2.5 Repeat (*)

```
string = "Hello World"
print(string * 3)
```

Output:

```
Hello WorldHello WorldHello World
```

11.2.6 Comparison

The ASCII or Unicode code obtained using the `ord` function is used to compare strings.

```
print("a" == "a")
print("a" < "b")
print("1" < "9")
print("a" < "B")
```

Output:

```
True
True
True
False
```

11.2.7 Membership (`in`)

`a in b` is True if string `a` is contained in string `b`.

```
string = "abcdef"
print("c" in string)
print("cde" in string)
print("cef" in string)
print(string in string)
```

Output:

```
True
True
False
True
```

11.2.8 Immutability

- Strings are immutable, which means you cannot change a string after it has been created.

  ```
  string = "spam"
  string[1] = "l"  # Error
  ```

- However, you can use it to make another string:

  ```
  string = "spam"
  new_string = string[:1] + "l" + string[2:]
  print(new_string)
  ```

  Output:

  ```
  slam
  ```

11.3 Lists (`list`)

- A list is an **ordered sequence of items**.
- As with all data structures, lists have a **constructor** that is the same name as the data structure.
- Lists are delimited with square brackets (`[]`).

11.3.1 Creation

Constructing a list or initialising a list both mean creating a list.

- Creating an empty list

  ```
  l = list()
  l = []
  ```

- Creating a list from an iterable data structure, like a set, a tuple, or a string

  ```
  print("List from tuple: ", list((1, 2, 3, 4)))
  print("List from set: ", list({1, 2, 3, 4}))
  print("List from string: ", list("1234"))
  ```

  Output:

  ```
  List from tuple:  [1, 2, 3, 4]
  List from set:  [1, 2, 3, 4]
  List from string:  ['1', '2', '3', '4']
  ```

- Creating a list with predefined items

  ```
  l = [1, 2, 3, 4]
  l = [1, 3.14159, "a", True]
  ```

11.3.2 Index ([])

- Items in a list are in a sequence
- We can identify each item in the list with a unique index (a position in the sequence).
- We can index an item from either end of the sequence.
  - Non-negative values: counting from left, starting at 0
  - Negative values: counting from right, starting at -1
- The index operator is always at the end of the expression and is preceded by something, either a variable or a sequence.

We can use `[]` to access particular items in a list.

```python
l = [1, "omg wow", 420.69, False]
print(l[0])
print(l[1])
print(l[-2])
print(l)
print(["hey", 5, True][0])
```

Output:

```
1
omg wow
420.69
[1, 'omg wow', 420.69, False]
hey
```

11.3.3 Slice ([:])

`[start : end : step]`

- `start` is the index of the start of the subsequence.
- `end` is the index of the end of a subsequence (not included).
- `step` specifies the step size to jump along the sequence.

```
l = [1, "omg wow", 420.69, False]
print(l[1:4])
print(l[::2])
print(l[::-2])
print(l[0:3:-1])
print(["hey", 5, True, "lovely"][2:4])
```

Output:

```
['omg wow', 420.69, False]
[1, 420.69]
[False, 'omg wow']
[]
[True, 'lovely']
```

11.3.4 Length (len)

```
l = [1, "omg wow", 420.69, False]
print(len(l))
```

Output:

```
4
```

11.3.5 Concatenation (+)

```
l_1 = [1, "omg wow", 420.69, False]
l_2 = ["Finger licking good!", True, "I'm loving it!"]
print(l_1 + l_2)
```

Output:

```
[1, 'omg wow', 420.69, False, 'Finger licking good!', True, "I'm loving it!"]
```

11.3.6 Repeat (*)

```
l = [1, "omg wow", 420.69, False]
print(l * 2)
l = [3]
print(l * 5)
```

Output:

```
[1, 'omg wow', 420.69, False, 1, 'omg wow', 420.69, False]
[3, 3, 3, 3, 3]
```

11.3.7 Comparison

- The items of a list are compared starting from the first item.
- This means for the greater than, or less than operators, the list that has the higher value at the front of the list is considered to be greater than the list with the same value at the back of the list.
- The boolean `True` is changed into 1 and the boolean `False` is changed to 0 comparing lists.
- For two lists to be equal, all items in both lists must be the same, and have the same order.

```
print([4, 3, 2, 1] > [1, 2, 3, 4])
print([4, 3, 2, 1] == [1, 2, 3, 4])
print([True, 3, 2, 1] >= [1, 2, 3, 4])
print([False, 2, 3, 4] <= [1, 2, 3, 4])
print(["a", "list", "of", "strings"] <= ["string", "list"])
```

Output:

```
True
False
True
True
True
```

11.3.8 Membership (`in`)

`a in b` is True if `a` is contained in the list `b`.

```
l = [1, "omg wow", 420.69, False]
print(1 in l)
print(420 in l)
print("omg wow" in l)
print("omg" in l)
```

Output:

```
True
False
True
False
```

11.3.9 Minimum (`min`)

- Gets the smallest element in the list.
- Only applicable to **lists containing strings only** or **lists containing numeric values (`int` and `float`) and boolean values**.
- The boolean `True` is changed into 1 and the boolean `False` is changed to 0 when evaluating the minimum of a list.

```
l_1 = [1, 2, 420.69, 699]
l_2 = [1, 2, True, False]
l_3 = ["ooo", "a", "list", "of", "Only", "Strings"]
print(min(l_1))
print(min(l_2))
print(min(l_3))
```

Output:

```
1
False
Only
```

11.3.10 Maximum (`max`)

- Gets the largest element in the list.
- Only applicable to **lists containing strings only** or **lists containing numeric values (`int` and `float`) and boolean values**.
- The boolean `True` is changed into 1 and the boolean `False` is changed to 0 when evaluating the maximum of a list.

```
l_1 = [1, 2, 420.69, 699]
l_2 = [1, 2, True, False]
l_3 = ["ooo", "a", "list", "of", "Only", "Strings"]
print(max(l_1))
print(max(l_2))
print(max(l_3))
```

Output:

```
699
2
ooo
```

11.3.11 Sum (`sum`)

- Gets the sum of the elements of the list.
- Only applicable to lists containing numeric values (`int` and `float`) and boolean values.
- The boolean `True` is changed into 1 and the boolean `False` is changed to 0 when evaluating the maximum of a list.

```
l_1 = [1, 2, 420.69, 699]
l_2 = [1, 2, True, False]
print(sum(l_1))
print(sum(l_2))
```

Output:

```
1122.69
4
```

11.3.12 Mutability

Lists are mutable, which means you can change them after you create them.

```python
l = [1, "omg wow", 420.69, False]
l[1] = "changed the list hehe"
print(l)
```

Output:

```
[1, 'changed the list hehe', 420.69, False]
```

11.3.13 List of lists

A list can be nested inside another list.

```python
l = [1, [1, "a", "list", True], 420.69, False]
print(l[1])
print(l[2])
print(l[1][2])
l[1][3] = "hehe changed the inner list"
print(l)
```

Output:

```
[1, 'a', 'list', True]
420.69
list
[1, [1, 'a', 'list', 'hehe changed the inner list'], 420.69, False]
```

11.3.14 Differences from strings

- Lists can contain a **mixture of Python objects (types)** while strings can only hold characters. (This is not recommended though, your list should only hold items of the same type, otherwise it makes things very difficult to work with.)
- Lists are mutable, which means you can change them.
- Lists are designated with square brackets (`[]`), with elements separated by commas (`,`), while strings use double quotes `"` or single quotes `'`.

11.4 Tuples (`tuple`)

- Tuples are immutable lists, or lists that cannot be changed.
- They are designated by **commas (`,`)**, **NOT** round brackets (`()`).
- Round brackets are often used to make tuples more readable, and used to group the items (like in nested tuples), but they are not what makes a tuple.

11.4.1 Creation

Constructing a tuple or initialising a tuple both mean creating a tuple.

- Creating an empty tuple

  ```
  t = tuple()
  ```

- Creating a tuple from an iterable data structure, like a set, a list, or a string

  ```
  print("Tuple from list: ", tuple([1, 2, 3, 4]))
  print("Tuple from set: ", tuple({1, 2, 3, 4}))
  print("Tuple from string: ", tuple("1234"))
  ```

  Output:

  ```
  Tuple from list:  (1, 2, 3, 4)
  Tuple from set:  (1, 2, 3, 4)
  Tuple from string:  ('1', '2', '3', '4')
  ```

- Creating a tuple with predefined items

  ```
  t = (1, 2, 3, 4)
  t = (1, 3.14159, "a", True)
  t = (1, )
  t = 1,
  ```

11.4.2 Index ([])

- Items in a tuple are in a sequence
- We can identify each item in the tuple with a unique index (a position in the sequence).
- We can index an item from either end of the sequence.
  - Non-negative values: counting from left, starting at 0
  - Negative values: counting from right, starting at -1
- The index operator is always at the end of the expression and is preceded by something, either a variable or a sequence.

We can use [] to access particular items in a tuple.

```
t = (1, "omg wow", 420.69, False)
print(t[0])
print(t[1])
print(t[-2])
print(t)
print(("hey", 5, True)[0])
```

Output:

```
1
omg wow
420.69
(1, 'omg wow', 420.69, False)
hey
```

11.4.3 Slice ([:])

`[start : end : step]`

- `start` is the index of the start of the subsequence.
- `end` is the index of the end of a subsequence (not included).
- `step` specifies the step size to jump along the sequence.

```
t = (1, "omg wow", 420.69, False)
print(t[1:4])
print(t[::2])
print(t[::-2])
print(t[0:3:-1])
print(("hey", 5, True, "lovely")[2:4])
```

Output:

```
('omg wow', 420.69, False)
(1, 420.69)
(False, 'omg wow')
()
(True, 'lovely')
```

11.4.4 Length (len)

```
t = (1, "omg wow", 420.69, False)
print(len(t))
```

Output:

```
4
```

11.4.5 Concatenation (+)

```
t_1 = (1, "omg wow", 420.69, False)
t_2 = ("Finger licking good!", True, "I'm loving it!")
print(t_1 + t_2)
```

Output:

```
(1, 'omg wow', 420.69, False, 'Finger licking good!', True, "I'm loving it!")
```

11.4.6 Repeat (*)

```
t = (1, "omg wow", 420.69, False)
print(t * 2)
t = (3,)
print(t * 5)
```

Output:

```
(1, 'omg wow', 420.69, False, 1, 'omg wow', 420.69, False)
(3, 3, 3, 3, 3)
```

11.4.7 Comparison

- The items of a tuple are compared starting from the first item.
- This means for the greater than, or less than operators, the tuple that has the higher value at the front of the tuple is considered to be greater than the tuple with the same value at the back of the tuple.
- The boolean `True` is changed into 1 and the boolean `False` is changed to 0 comparing tuples.
- For two tuples to be equal, all items in both tuples must be the same, and have the same order.

```
print((4, 3, 2, 1) > (1, 2, 3, 4))
print((4, 3, 2, 1) == (1, 2, 3, 4))
print((True, 3, 2, 1) >= (1, 2, 3, 4))
print((False, 2, 3, 4) <= (1, 2, 3, 4))
print(("a", "tuple", "of", "strings") <= ("string", "tuple"))
```

Output:

```
True
False
True
True
True
```

11.4.8 Membership (`in`)

`a in b` is True if `a` is contained in the tuple `b`.

```
t = (1, "omg wow", 420.69, False)
print(1 in t)
print(420 in t)
print("omg wow" in t)
print("omg" in t)
```

Output:

```
True
False
True
False
```

11.4.9 Minimum (`min`)

- Gets the smallest element in the tuple.
- Only applicable to **tuples containing strings only** or **tuples containing numeric values (`int` and `float`) and boolean values**.
- The boolean `True` is changed into 1 and the boolean `False` is changed to 0 when evaluating the minimum of a tuple.

```
t_1 = (1, 2, 420.69, 699)
t_2 = (1, 2, True, False)
t_3 = ("ooo", "a", "tuple", "of", "Only", "Strings")
print(min(t_1))
print(min(t_2))
print(min(t_3))
```

Output:

```
1
False
Only
```

11.4.10 Maximum (`max`)

- Gets the largest element in the tuple.
- Only applicable to **tuples containing strings only** or **tuples containing numeric values (`int` and `float`) and boolean values**.
- The boolean `True` is changed into 1 and the boolean `False` is changed to 0 when evaluating the maximum of a tuple.

```
t_1 = (1, 2, 420.69, 699)
t_2 = (1, 2, True, False)
t_3 = ("ooo", "a", "tuple", "of", "Only", "Strings")
print(max(t_1))
print(max(t_2))
print(max(t_3))
```

Output:

```
699
2
tuple
```

11.4.11 Sum (`sum`)

- Gets the sum of the elements of the tuple.
- Only applicable to tuples containing numeric values (`int` and `float`) and boolean values.
- The boolean `True` is changed into 1 and the boolean `False` is changed to 0 when evaluating the maximum of a tuple.

```
t_1 = (1, 2, 420.69, 699)
t_2 = (1, 2, True, False)
print(sum(t_1))
print(sum(t_2))
```

Output:

```
1122.69
4
```

11.4.12 Immutability

- Tuples are immutable, which means you cannot change a tuple after it has been created.

  ```
  t = (1, "omg wow", 420.69, False)
  t[1] = "tuple"   # Error
  ```

- However, you can use it to make another tuple:

  ```
  t = (1, "omg wow", 420.69, False)
  new_tuple = t[:1] + ("tuple",) + t[2:]
  print(new_tuple)
  ```

  Output:

  ```
  (1, 'tuple', 420.69, False)
  ```

11.4.13 Tuple of tuples

A tuple can be nested inside another tuple.

```
t = (1, (1, "a", "tuple", True), 420.69, False)
print(t[1])
print(t[2])
print(t[1][2])
```

Output:

```
(1, 'a', 'tuple', True)
420.69
tuple
```

11.5 Dictionaries (`dict`)

- A dictionary is an associative array, or associative list, or a map.
- You can think of it as a list of pairs.
  - The key, which is first element of the pair, is used to retrieve the second element, which is the value.
- Hence, we map a key to a value in a dictionary.
- The key acts as a "lookup" to find the associated value.
- Just like a dictionary, you look up a word by its spelling to find the associated definition.
- A dictionary can be searched to locate the value associated with a key.
- The key of the dictionary must be immutable, so strings, integers and tuples are allowed, but **lists are NOT**.
- The value can be any Python object.

11.5.1 Creation

Constructing a dictionary or initialising a dictionary both mean creating a dictionary.

- Creating an empty dictionary

```
dic = dict()
dic = {}
```

- Creating a dictionary with predefined items

```
dic = {"omg": "lol"}
dic = {1: "lol"}
dic = {5: 42069}
```

11.5.2 Index ([])

Items in a dictionary can be indexed using its respective key.

We can use [] to access particular values in a dictionary.

```
dic = {
    "bill": 25,
    "tax": 3
}
print(dic["bill"])
```

Output:

```
25
```

11.5.3 Adding items ([])

Items can also be added to the dictionary using the indexing operator.

```
dic = {
    "bill": 25,
    "tax": 3
}
dic["petrol"] = 10
print(dic)
```

Output:

```
{'bill': 25, 'tax': 3, 'petrol': 10}
```

11.5.4 Removing items (del)

Items can be removed from the dictionary using `del` keyword with the indexing operator.

```
dic = {
    "bill": 25,
    "tax": 3
}
del dic["tax"]
print(dic)
```

Output:

```
{'bill': 25}
```

11.5.5 Length (`len`)

```
dic = {
    "bill": 25,
    "tax": 3
}
print(len(dic))
```

Output:

```
2
```

11.5.6 Membership (`in`)

`a in b` is True if the key `a` is contained in the dictionary `b`.

```
dic = {
    "bill": 25,
    "tax": 3
}
print("bill" in dic)
print(25 in dic)
print("3" in dic)
print("tax" in dic)
```

Output:

```
True
False
False
True
```

11.5.7 For loop (`for`)

A `for` loop iterates over the **keys** of the dictionary.

```
dic = {
    "bill": 25,
    "tax": 3
}
for key in dic:
    print(key)
```

Output:

```
bill
tax
```

11.6 Integers (`int`)

- Integers are like whole numbers, including the negative numbers. They can be negative (-1, -2, -3, etc), positive (1, 2, 3, etc) or zero (0).
- The largest n-bit integer is given by 2^n - 1. For example, the largest 16-bit integer is 2^{16} - 1 = 65535.

11.7 Floats (`float`)

- Floats represent real numbers and have a decimal point, like 2.8, 7.1 and 9.0001.
- When writing them down, they **must always have the decimal point**, so 2 should be represented as 2.0.

11.8 Boolean (`bool`)

In most computer programming languages, a Boolean data type is a data type with only two possible values, either True or False.

12 Python syntax

12.1 Statements

Each line of code in a Python program is called a **statement**. Python interprets and runs the statements one by one.

Python is sensitive to the end of line in text files, which marks the end of a statement. In text editors, we press "Enter".

12.1.1 Continuation of a statement

- The symbol "\" is used to continue a statement with the next line so that the two lines can be joined as one statement.
- It improves readability in the text editor.

12.2 Comments

- The pound sign "#" in Python indicates a comment.
- **Anything after "#"** is ignored during interpretation.

12.3 `if` statements

- A colon must be used to mark the start of a block
- An indentation must be used for the entire block

12.3.1 Example

```python
a = 5
b = 1
if a > b:
    print("a > b")
```

12.3.2 `if-else` statements

```python
a = 5
b = 1
if a > b:
    print("a > b")
else:
    print("b < a")
```

12.3.3 `if-elif-else` statements

```python
a = 5
b = 10
if a > b:
    print("a > b")
elif a > 4:
    print("a > 4")
else:
    print("b < a")
```

12.4 `while` loops

- The `while` statement allows repetition of a group of Python code as long as a condition (Boolean expression) is True.
- It is structurally similar to an if statement but repeats the block until the condition becomes False.
- When the condition becomes False, repetition ends and control moves on to the code following the repetition.

12.4.1 Syntax

```python
count = 0
while count < 10:
    count += 1
```

12.5 `while-else` loops (avoid using as much as possible)

- The `while` loop can have an associated `else` statement.
- The `else` block is executed when the loop finishes under normal conditions. It is the last thing the loop does as it exits.
- The `else` block is entered after the `while` loop's Boolean expression becomes False.
- This occurs even when the expression is initially False and the `while` loop has never run.
- It is a handy way to perform some final tasks when the loop ends normally.

12.5.1 Syntax

```python
count = 0
while count < 10:
    count += 1
else:
    print("Done looping")
```

12.6 `break` statement

- The `break` statement can be used to **immediately** exit the execution of the current loop and skip past all the remaining parts of the loop.
- Note that "skip past" also means that the `break` statement also **skips** the `else` block as well.
- The `break` statement is useful for stopping computation when the "answer" has been found or when continuing the computation is otherwise useless.

12.6.1 Syntax

```python
count = 0
while count < 10:
    count += 1
    if count == 8:
        break

# The else clause is skipped in this case
# as the loop is broken out of when the count hits 8
else:
    print("Done looping")
```

12.7 `continue` statement

- The `continue` statement skips some portion of the `while` block that we are executing and have control flow back to the beginning of the `while` loop.
- Exit early from this iteration of the loop (not the loop itself), and keep executing the `while` loop.
- The `continue` statement continues with the next iteration of the loop.

12.7.1 Syntax

```
count = 0
while count < 10:
    count += 1
    if count == 3:
        print("It's a three!")
        continue

    # This print statement is skipped when the count is 3
    print("count is", count)
```

12.8 `for` loops

- `for` loops have the ability to iterate over the items of any sequence, such as a list or a string.
- Each item in the sequence is assigned to the iterating variable, which can be any variable.
- The `for` loop completes when the last of the elements has been assigned to the iterating variable, or when the entire sequence is exhausted.

12.8.1 Syntax

```python
for i in range(10):
    print("i is", i)

for char in "Hello World!":
    print("Current char is", char)

for drink in ("coffee", "tea", "milo"):
    print("Current drink is", drink)
```

12.8.2 `for-else`

```python
for i in range(10):
    print("i is", i)
else:
    print("Done looping")
```

12.8.3 `for-else-break`

```python
for i in range(10):
    if i == 5:
        break
    print("i is", i)

# This else statement is skipped as
# the loop is broken out of when i is 5
else:
    print("Done looping")
```

12.8.4 `for-else-break-continue`

```python
for i in range(10):
    if i == 5:
        break
    if i == 3:
        print("It's a three!")
        continue

    # This print statement is skipped when i is 3
    print("i is", i)

# This else statement is skipped as
# the loop is broken out of when i is 5
else:
    print("Done looping")
```

12.9 pass statement

- The `pass` statement has no effect (it does nothing) but it helps in indicating an **empty** statement, suite, or block.
- Use the `pass` statement when you have to put something in a statement (syntactically, you cannot leave it blank or Python will complain), but what you really want is nothing.
- Python has the syntactical requirement that code blocks after `if`, `for`, `while`, `except`, `def`, `class`, etc cannot be empty.
- It can be used to **test a statement**, like opening a file or iterating through a collection to see if it works.
- You can also use `pass` as a placeholder.

12.9.1 Syntax

```python
for i in range(10):
    pass
```

12.10 List comprehension

- List comprehensions are a Python syntactic structure to construct lists concisely.
- The first variable before the `for` is what is placed into the list.
- The `for` loop is just a regular for loop.
- Any `for` loops after the first for loop will be **nested inside** the for loop before it. If there is a `if` statement at the end of the previous for loop, the next `for` loop will be placed inside the `if` statement. (This is absolutely not recommended as it makes code impossible to read.)
- Any `if` statements after the `for` loop will determine what will be added to the list. Basically, if the item doesn't meet the condition in the `if` statement, it's not added to the list.

12.10.1 Syntax

```python
print([i for i in range(11)])
print([i for i in range(21) if i % 2 == 0])
print([i for i in range(21) if i % 2 == 0 and i >= 10])

# Please don't do this, just use a regular for loop instead
print([x + y for x in range(9) if x % 2 == 0 for y in range(9) if y % 2 != 0])
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
[10, 12, 14, 16, 18, 20]
[1, 3, 5, 7, 3, 5, 7, 9, 5, 7, 9, 11, 7, 9, 11, 13, 9, 11, 13, 15]
```

12.10.2 Equivalent code using for loops

1. The first list comprehension:

   ```
   l = []
   for i in range(11):
       l.append(i)
   print(l)
   ```

   Output:

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

2. The second list comprehension:

   ```
   l = []
   for i in range(21):
       if i % 2 == 0:
           l.append(i)
   print(l)
   ```

   Output:

   ```
   [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
   ```

3. The third list comprehension:

   ```
   l = []
   for i in range(21):
       if i % 2 == 0 and i >= 10:
           l.append(i)
   print(l)
   ```

   Output:

   ```
   [10, 12, 14, 16, 18, 20]
   ```

4. The fourth list comprehension:

```
l = []
for x in range(9):
    if x % 2 == 0:
        for y in range(9):
            if y % 2 != 0:
                l.append(x + y)
print(l)
```

Output:

```
[1, 3, 5, 7, 3, 5, 7, 9, 5, 7, 9, 11, 7, 9, 11, 13, 9, 11, 13, 15]
```

12.11 Functions

12.11.1 Defining a function

- Functions in Python are defined using the `def` keyword.
- After the `def` keyword comes the name of the function, which has the same rules as Python identifiers. Refer to the section on Python identifiers for the rules.
- After the name of the function, there are parentheses (`()`) to indicate the function parameters, which are the inputs to be passed to a function.
- Each parameter inside the parentheses (`()`) is separated with a comma (`,`).
- If there are no parameters, an empty pair of parentheses (`()`) is fine as well.
- After the parameter list in parentheses, a colon (`:`) must be present at the back of the function definition.
- The line after the colon (`:`) must be indented, and is the start of the function body.

12.11.2 Function definition example

```python
# Function with parameters
def function_definition(parameter_1, parameter_2):

    # Function body...
    print("Just defined a function!")
    print("Printing arguments...")
    print("parameter_1: {}".format(parameter_1))
    print("parameter_2: {}".format(parameter_2))

    # Optional return statement (can be omitted).
    # When the return statement is omitted,
    # the function returns None
    return parameter_1 + parameter_2

# Function without parameters
def no_args_function():

    # Use pass to skip defining the function
    # This function will return None
    pass
```

12.11.3 Using a function

- Using a function is also called invoking a function or calling a function.
- To use a function, type out the function name and put the parentheses (`()`), as well as the desired arguments behind if there are any.

```python
# No arguments
print()

# With arguments
print("Hi there!")
```

12.12 Exception handling (`try-except` block)

12.12.1 `try` block

- The `try` block contains code that **we want to monitor** for errors during execution.
- If an error occurs anywhere in that `try` block, Python looks for a **handler** that can deal with the error.
- If no specific handler exists, Python handles it.
  - The program halts with an error message.

12.12.2 `except` block

- The `except` block is associated with a `try` block.
- A `try` block can have **multiple** `except` blocks after it.
- Each `except` block names a type of exception it is monitoring for and can **handle**.
- If the error occurring in the `try` block matches the type of exception, then the **first** `except` block is activated.

12.12.3 `try-except` block

- If no exception is in the `try` block, skip to the next line of code after the `try-except` block.
- If an error occurs in a `try` block, look for the correct exception handler in the `except` blocks.

12.12.4 Syntax

```python
try:
    do_something()
    do_something_else()
    do_something_with_error()
except TypeError:
    handle_type_error()
except IndexError:
    handle_index_error()
```

12.12.5 Example

```
try:
    print("Entering the try block")
    dividend = float(input("Enter the dividend: "))
    divisor = float(input("Enter the divisor: "))
    result = dividend / divisor
    print("The result is ", result)

except ZeroDivisionError:
    print("Can't divide by 0!")

except ValueError:
    print("Couldn't convert your input to a float")

print("Continuing with the rest of the program...")
```

12.12.6 Built-in exceptions

- Python has a list of exceptions that are built-in.
- To find an exception that you're interested in, you can try it in the Python interpreter.
- You can also look at the Python documentation on exceptions.

12.12.7 `else` block

The `else` block is used to execute code when **no exception** occurs.

```
try:
    result = do_something_with_error()
except IndexError:
    handle_index_error()
else:
    do_something_with_result(result)
```

12.12.8 `finally` block

- The `finally` block is used to execute code at the end of a `try-except` block, regardless of whether an error has occurred.
- The `finally` block must be placed **after all the other blocks** in the `try-except` code block.

```
try:
    do_something_with_error()
except TypeError:
    handle_type_error()
finally:
    always_do_this_regardless_of_error()
```

12.12.9 `try-except-else-finally`

Note that the `else` block must always come **before** the `finally` block.

```
try:
    result = x / y
except ZeroDivisionError:
    print("Divide by zero!")
else:
    print(result)
finally:
    print("Goodbye!")
```

12.13 Method in general

```
object.method()
```

We say that `object` is calling the method `method`.

12.14 Whitespace

Python counts the "Tab", "Space bar" and "Enter" as white spaces.

- The purpose of whitespace is to separate words in a statement.
- For the most part, you can place white spaces anywhere in your program to make the code more readable.

12.15 Indentation

An indentation is a leading whitespace at the start of a statement. In Python, a group of indented statements is called a **suite** or a **block**. A **compound statement** is a set of statements being used as a group.

Purpose of indentation:

- Makes the code more readable
- For grouping, particularly for control flow such as branching and looping

12.16 Arithmetic operators

Operator	Meaning
`+`	Add two operands or unary plus
`-`	Subtract the right operand from the left or unary minus
`*`	Multiply two operands
`/`	Floating point division: divide the left operand by the right one (always results in a `float`)
`%`	Modulus: remainder of the division of the left operand by the right
`//`	Floor division (integer division): the resultant value is a whole integer, although the result's type is not necessarily `int`
`**`	Exponent: the left operant is raised to the power of the right operand

12.17 Augmented assignment operators

Shortcut	Equivalent
`x += 2`	`x = x + 2`
`x -= 2`	`x = x - 2`
`x /= 2`	`x = x / 2`
`x *= 2`	`x = x * 2`
`x %= 2`	`x = x % 2`

12.18 "Truthy" and "falsy" values

A "truthy" value is a value that will satisfy the check performed by `if` or `while` statements, while "falsy" values will not. "Truthy" and "falsy" are used to differentiate from the `bool` values `True` and `False`.
All values are considered "truthy", except for the following, which are considered "falsy":

- `None`
- `False`
- `0`
- `0.0`
- `0j`
- `Decimal(0)`
- `Fraction(0, 1)`
- `[]` - an empty `list`
- `{}` - an empty `dict`
- `tuple()` - an empty `tuple`
- '' - an empty `str`
- `b''` - an empty `bytes`
- `set()` - an empty `set`
- An empty range, like `range(0)`
- Objects for which:
  - `obj.__bool__()` returns `False`
  - `obj.__len__()` returns `0`

12.19 Chained comparisons

In Python, chained comparisons work just like you would expect in a mathematical expression.

12.19.1 Examples

- `0 <= x <= 5` is the same as `0 <= x and x <= 5`.
- `0 <= x <= 5 > 10` is the same as `0 <= x and x <= 5 and x > 10`.

13 Documentation for Python functions

13.1 Floor division operator (//)

- The floor division operator takes the floor of the left number divided by the right number. Flooring a number means to round the number down to the integer. For example, flooring 5.987 will result in 5.0, and flooring -3.14 will result in -4.0.
- Take note that flooring a floating point number (`float`), will result in a **floating point number (`float`)** being returned, **NOT** an integer (`int`).
- To get an integer type back, use the `math.floor` function instead, which returns an integer instead of a float when flooring a floating point number.

13.1.1 Syntax

```
x // y
```

13.2 Modulus operator (remainder operator) (%)

The modulus operator takes the remainder of the left number divided by the right number.

13.2.1 Syntax

```
x % y
```

The remainder of `x / y`.

13.2.2 Formula used to calculate the remainder

```
x % y = x - y * (x // y)
```

13.2.3 Example

```
print(8 % 3)
```

Output:

```
2
```

13.3 Boolean AND operator (`and`)

Return the first **"falsy"** value if there are any, else return the **last** value in the expression. Refer to the section on "truthy" and "falsy" values for an explanation of "falsy".

13.3.1 Syntax

```
a and b
```

13.3.2 Example

```
a = 1
b = 5
print(a > 1 and b < 10)
print(a == 1 and b > 3)
print(a and b)
print(False and "something")
print(1 and 5 and 6 and 0 and 10 and 9)
```

Output:

```
False
True
5
False
0
```

13.4 Boolean OR operator (`or`)

Return the first **"truthy"** value if there are any, else return the **last** value in the expression. Refer to the section on "truthy" and "falsy" values for an explanation of "truthy".

13.4.1 Syntax

```
a or b
```

13.4.2 Example

```
a = 1
b = 5
print(a > 1 or b < 10)
print(a == 1 or b > 3)
print(a or b)
print(False or "something")
print(1 or 5 or 6 or 0 or 10 or 9)
print("" or "default value")
print(False or "" or 0 or [] or "wow")
```

Output:

```
True
True
1
something
1
default value
wow
```

13.4.3 In combination with the boolean AND operator (`and`)

```
a = 5
b = 10
print((a == 5 or b > 30) and (a < 2 or b > 5))
print(a >= 3 and b < 15 or a < 3 and b > 8)
print(a and b or b or a)
print(a and b or b and a or a or b or a)
print(0 and 5 or 100)
```

Output:

```
True
True
10
10
100
```

13.5 `input`

`input` is a built-in function in Python to get an input.

- It prints the message string on the screen and waits until the user types anything and presses "Enter".
- It returns a **string** no matter what is given, even a number.

13.5.1 Usage

```
user_input = input("Please enter an input: ")
```

`user_input` will be a **string**.

13.6 `print`

`print` is another built-in function in Python that displays related messages and data on the shell screen. It uses a **comma** to separate the elements.

```
a = 1
b = 2
print("a is", a, "and b is", b)
```

Output:

```
a is 1 and b is 2
```

13.6.1 `sep` parameter (optional)

`sep` takes a **string** that specifies what **string** to use to separate the objects passed to the function, if there is more than one. It defaults to a space character, `" "`.

```
a = 1
b = 2
print("a is", a, "and b is", b, sep=" @ ")
```

Output:

```
a is @ 1 @ and b is @ 2
```

13.6.2 `end` parameter (optional)

`end` takes a **string** that specifies what the end of the printed string should be. It defaults to a new line `"\n"`.

```
print("Hello world!")
print("My first program.")
print("Hello world!", end=" ")
print("My first program.")
```

Output:

```
Hello world!
My first program.
Hello world! My first program.
```

13.6.3 `file` parameter (optional)

- `file` takes a **file object** as the file to print to.
- Nothing will be printed to the screen when this parameter is given.
- Instead, the file will be written to with the items given to the `print` function.

```
file = open("temp.txt", "w")
print("First line", file=file)
print("Second line", file=file)
print("Third line", file=file)
file.close()
```

13.7 `range`

`range([start], end[, step])`

- `start` is the starting number of the sequence. The default value is 0.
- `end` is the number to generate numbers up to, but not including this number.
- `step` is the difference between each number in the sequence. The default value is 1.

13.7.1 `range(end)`

- Python sets `start` to 0 and `step` to 1.

```
print(list(range(11)))
```

Output:

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

13.7.2 `range(start, end)`

- Python sets `step` to 1.

```
print(list(range(1, 11)))
```

Output:

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

13.7.3 `range(start, end, step)`

```
print(list(range(1, 11, 2)))
print(list(range(11, 2, -2)))
```

Output:

```
[1, 3, 5, 7, 9]
[11, 9, 7, 5, 3]
```

13.8 `ord`

Takes a character as input and returns the Unicode of the character. For standard symbols, this is the same as in ASCII.

```
print(ord("a"))
```

Output:

```
97
```

13.9 `chr`

Takes an ASCII or UTF-8 code and returns the corresponding character.

```
print(chr(97))
```

Output:

```
a
```

13.10 `sorted`

- This function takes a collection, like a **list**, **tuple**, **dictionary**, or a **set**, and returns a **sorted list** of the items in the collection.
- This function does not modify the original collection, and returns a **new** list.
- This function uses a sorting algorithm called Timsort.

```
num_list = [10, 6, 3, 1, 9]
num_tuple = (10, 6, 3, 1, 9)
num_set = {10, 6, 3, 1, 9}
num_dict = {
    10: "ten",
    6: "six",
    3: "three",
    1: "one",
    9: "nine"
}

print(sorted(num_list))
print(sorted(num_tuple))
print(sorted(num_set))
print(sorted(num_dict))
```

Output:

```
[1, 3, 6, 9, 10]
[1, 3, 6, 9, 10]
[1, 3, 6, 9, 10]
[1, 3, 6, 9, 10]
```

13.10.1 `key` parameter (optional)

- `key` takes a function that acts on the **items** in the collection that is being sorted.
- The sorting algorithm uses the **result** of that function to sort the values.

```python
num_list = [10, 6, 3, 1, 9]
num_dict = {
    10: "ten",
    6: "six",
    3: "three",
    1: "one",
    9: "nine"
}

def get_dict_value(dict_item: tuple):
    "Function to get the value from the dictionary when calling dict.items()"
    return dict_item[1]

def alternate(num: int) -> int:
    """
    Function to change the sign of the number based on its value.

    For example, for 10, this function will return:
    result = 10 * (-1) ** (10 + 1)
    result = 10 * (-1) ** 11
    result = 10 * (-1)
    result = -10
    """
    return num * (-1) ** (num + 1)

# Sort based on the dictionary value, which is to sort using alphabetical order
print(sorted(num_dict.items(), key=get_dict_value))

print(sorted(num_list, key=alternate))
```

Output:

```
[(9, 'nine'), (1, 'one'), (6, 'six'), (10, 'ten'), (3, 'three')]
[10, 6, 1, 3, 9]
```

13.10.2 `reverse` parameter (optional)

- `reverse` takes a boolean that tells the sorted function to reverse the list.
- The value of `True` will **reverse** the list.
- The value of `False` will **not** reverse the list.
- The default value is `False`.

```
num_list = [10, 6, 3, 1, 9]

print(sorted(num_list))
print(sorted(num_list, reverse=True))
```

Output:

```
[1, 3, 6, 9, 10]
[10, 9, 6, 3, 1]
```

13.11 String method: `string.upper`

This method will output a new string, which is the same as the string on which it was called, except all letters in the string will now be in uppercase.

```
string = "Shouting!"
print(string.upper())
```

Output:

```
SHOUTING!
```

13.12 String method: `string.lower`

This method will output a new string, which is the same as the string on which it was called, except all letters in the string will now be in lowercase.

```
string = "WHISPERING"
print(string.lower())
```

Output:

```
whispering
```

13.13 String method: `string.title`

This method will output a new string, which is the same as the string on which it was called, except that the string will be in title case, which means that every word in the string will be capitalised.

```
string = "john f. kennedy"
print(string.title())
```

Output:

```
John F. Kennedy
```

13.14 String method: `string.capitalize`

- This method will output a new string, which is the same as the string on which it was called, except that the first letter of the first word of the string will be capitalised.
- Take note that capitalize is using the American spelling with a "z", instead of the British spelling with a "s".

```
string = "this is a sentence."
print(string.capitalize())
```

Output:

```
This is a sentence.
```

13.15 String method: `string.find`

- This method takes a single character and outputs the **index** of the character (first seen from left to right).
- If the character is not found, `-1` is returned.

```
string = "Find me!"
print(string.find("m"))
print(string.find("a"))
```

Output:

```
5
-1
```

13.16 String method: `string.index`

- This method takes a single character and outputs the **index** of the character (first seen from left to right).
- If the character is not found, a `ValueError` is thrown.
- This method is essentially the same as the above `string.find` method, but it throws an error when the value isn't found.
- In a way, it's a worse version of the `string.find` method.

```
string = "Find me!"
print(string.index("m"))
print(string.index("a"))    # Error
```

Output:

```
5
```

13.17 String method: `string.count`

This method counts the number of occurrences of the substring you pass to the method.

```
string = "This is an pretty long sentence."
print(string.count("i"))
print(string.count("t"))
```

Output:

```
2
3
```

13.18 String method: `string.join`

This method takes a **string** to be joined and outputs a new string where the base string joins the target string.

```
string_1 = "1234"
string_2 = "abcd"
print(string_1.join(string_2))
print(string_2.join(string_1))
```

Output:

```
a1234b1234c1234d
1abcd2abcd3abcd4
```

13.19 String method: `string.split`

- This method takes a **string** and generates a list of strings by splitting the string at the given string.
- The default character the string splits at is **whitespace**.
- This method returns a list of strings.

```
l = "oooo a list of strings"
print(l.split())
print(l.split("o"))
print(l.split("st"))
```

Output:

```
['oooo', 'a', 'list', 'of', 'strings']
['', '', '', '', ' a list ', 'f strings']
['oooo a li', ' of ', 'rings']
```

13.20 String method: `string.isalnum`

Returns `True` if all characters in the string are **alphanumeric**, which means all the letters of the alphabet and the numbers 0 to 9.

```
print("abcdef12345".isalnum())
print("abcdef12345_".isalnum())
print("abcdef12345_(*&#)".isalnum())
print("-1".isalnum())
print("1.5".isalnum())
print("-1.5".isalnum())
```

Output:

```
True
False
False
False
False
False
```

13.21 String method: `string.isalpha`

Returns `True` if all characters in the string are in the **alphabet**.

```
print("abcdef".isalpha())
print("abcdef12345".isalpha())
print("abcdef12345_(*&#)".isalpha())
```

Output:

```
True
False
False
```

13.22 String method: `string.isascii`

Returns `True` if all characters in the string are **ASCII characters**.

```
print("abcdef".isascii())
print("abcdef12345".isascii())
print("abcdef12345_(*&#)".isascii())
```

Output:

```
True
True
True
```

13.23 String method: `string.isdecimal`

Returns `True` if all characters in the string are **decimals (0 to 9)**.

```
print("12392483".isdecimal())
print("sldkfjd".isdecimal())
print("abcdef12345".isdecimal())
print("abcdef12345_(*&#)".isdecimal())
print("-1".isdecimal())
print("1.5".isdecimal())
print("-1.5".isdecimal())
```

Output:

```
True
False
False
False
False
False
False
```

13.24 String method: `string.isdigit`

- Returns `True` if all characters in the string are **digits**.
- Exponents like the superscript 2 in 1^2 are also considered digits.
- `"-1"`, `"1.5"` and `"-1.5"` are **not** considered digits as all the characters in the string must be a digit. `"-"` and `"."` are not digits.

```
print("12392483".isdigit())
print("sldkfjd".isdigit())
print("abcdef12345".isdigit())
print("abcdef12345_(*&#)".isdigit())
print("-1".isdigit())
print("1.5".isdigit())
print("-1.5".isdigit())
```

Output:

```
True
False
False
False
False
False
False
```

13.25 String method: `string.isidentifier`

Returns `True` if the string is **a valid Python identifier**. Basically, it checks if the string is a valid Python variable name.

```
print("12392483".isidentifier())
print("varname".isidentifier())
print("_private".isidentifier())
print("string1".isidentifier())
print("omg!".isidentifier())
```

Output:

```
False
True
True
True
False
```

13.26 String method: `string.islower`

Returns `True` if all characters in the string are **lower case**.

```
print("12093".islower())
print("whispering...".islower())
print("SHOUTING!".islower())
print("39485943wow".islower())
print("39485943WOW".islower())
```

Output:

```
False
True
False
True
False
```

13.27 String method: `string.isnumeric`

- Returns `True` if all characters in the string are **numeric**.
- Exponents like the superscript 2 in 1^2, as well as fractions, are also considered numeric.
- `"-1"`, `"1.5"` and `"-1.5"` are **not** considered numeric as all the characters in the string must be numeric. `"-"` and `"."` are not numeric characters.

```
print("12392483".isnumeric())
print("sldkfjd".isnumeric())
print("abcdef12345".isnumeric())
print("abcdef12345_(*&#)".isnumeric())
print("-1".isnumeric())
print("1.5".isnumeric())
print("-1.5".isnumeric())
```

Output:

```
True
False
False
False
False
False
False
```

13.28 String method: `string.isprintable`

Returns `True` if all characters in the string are **printable**.

```
print("12392483".isprintable())
print("sldkfjd".isprintable())
print("abcdef12345".isprintable())
print("abcdef12345_(*&#)".isprintable())
```

Output:

```
True
True
True
True
```

13.29 String method: `string.isspace`

Returns `True` if all characters in the string are **white spaces**.

```
print("      ".isspace())
print("  alkdfjlas".isspace())
print("    abc   def   12345    ".isspace())
print("".isspace())
```

Output:

```
True
False
False
False
```

13.30 String method: `string.istitle`

Returns `True` if the string is in **title case**.

```
print("Wow This Is Cool".istitle())
print("wow this is cool".istitle())
print("WOW THIS IS COOL".istitle())
print("Wow this is cool".istitle())
```

Output:

```
True
False
False
False
```

13.31 String method: `string.isupper`

Returns `True` if all characters in the string are **upper case**.

```
print("12093".isupper())
print("whispering...".isupper())
print("SHOUTING!".isupper())
print("39485943wow".isupper())
print("39485943WOW".isupper())
```

Output:

```
False
False
True
False
True
```

13.32 String method: `string.format`

- This method takes any number of arguments and keyword arguments which are substituted into the string that the format method is being called on.
- The string that calls this method should have placeholders inside curly brackets {}.
- The placeholders can be either be identified using named indexes `{price}`, or numbered indexes `{0}`, or empty placeholders `{}`.

```
print("{} is a nice number.".format(4200.696969))
print("{meh:.2f} is a meh number.".format(meh=4200.696969))
print("{0:.4f} is a meh number.".format(4200.696969))
print("{:_.6f} is a nice number.".format(4200.696969))
print("{nice:,.6f} is a nice number.".format(nice=4200.696969))
```

Output:

```
4200.696969 is a nice number.
4200.70 is a meh number.
4200.6970 is a meh number.
4_200.696969 is a nice number.
4,200.696969 is a nice number.
```

13.32.1 :< formatting type

Left aligns the result (within the available space given).

```
print("{:<20} 20 spaces".format(4200.696969))
print("{:*<15} 15 stars".format(4200.696969))
print("{:#<10} 10 hashes".format(4200.696969))
```

Output:

```
4200.696969          20 spaces
4200.696969**** 15 stars
4200.696969 10 hashes
```

13.32.2 :> formatting type

Right aligns the result (within the available space given).

```
print("{:>20} 20 spaces".format(4200.696969))
print("{:*>15} 15 stars".format(4200.696969))
print("{:#>10} 10 hashes".format(4200.696969))
```

Output:

```
         4200.696969 20 spaces
****4200.696969 15 stars
4200.696969 10 hashes
```

13.32.3 :^ formatting type

Centre aligns the result (within the available space given).

```
print("{:^20} 20 spaces".format(4200.696969))
print("{:*^15} 15 stars".format(4200.696969))
print("{:#^10} 10 hashes".format(4200.696969))
```

Output:

```
    4200.696969      20 spaces
**4200.696969** 15 stars
4200.696969 10 hashes
```

13.32.4 := formatting type

Places the sign to the left most position instead of just left of the number when there is padding.

```
print("{:20} 20 spaces".format(-4200.696969))
print("{:=20} 20 spaces".format(-4200.696969))
```

```
        -4200.696969 20 spaces
-        4200.696969 20 spaces
```

Output:

```
4200.696969 positive number
-4200.696969 negative number
0 zero
```

13.32.5 `:+` formatting type

Use a plus sign or a minus sign to indicate if the result is positive or negative respectively.

```
print("{:+} positive number".format(4200.696969))
print("{:+} negative number".format(-4200.696969))
print("{:+} zero".format(0))
```

Output:

```
+4200.696969 positive number
-4200.696969 negative number
+0 zero
```

13.32.6 `:-` formatting type

- Use a minus sign for negative values only.
- This is the default behaviour.

```
print("{:-} positive number".format(4200.696969))
print("{:-} negative number".format(-4200.696969))
print("{:-} zero".format(0))
```

Output:

```
4200.696969 positive number
-4200.696969 negative number
0 zero
```

13.32.7 `:<space>` formatting type

- Replace the `<space>` with an actual space character.
- Use a space to insert an extra space before positive numbers (and a minus sign before negative numbers).

```
print("{: } positive number".format(4200.696969))
print("{: } negative number".format(-4200.696969))
print("{: } zero".format(0))
```

Output:

```
 4200.696969 positive number
-4200.696969 negative number
 0 zero
```

13.32.8 `:,` formatting type

Use a comma as a thousand separator.

```
print("{:,} positive number".format(4200.696969))
print("{:,} negative number".format(-4200.696969))
print("{:,} zero".format(0))
```

Output:

```
4,200.696969 positive number
-4,200.696969 negative number
0 zero
```

13.32.9 `:_` formatting type

Use an underscore as a thousand separator.

```
print("{:_} positive number".format(4200.696969))
print("{:_} negative number".format(-4200.696969))
print("{:_} zero".format(0))
```

Output:

```
4_200.696969 positive number
-4_200.696969 negative number
0 zero
```

13.32.10 `:b` formatting type

Binary format, only applicable to integers.

```
print("{:b} positive number".format(42069))
print("{:b} negative number".format(-42069))
print("{:b} zero".format(0))
```

Output:

```
1010010001010101 positive number
-1010010001010101 negative number
0 zero
```

13.32.11 `:c` formatting type

Converts the value into the corresponding Unicode character. Only applicable to positive integers.

```
print("{:c} Unicode character 100".format(100))
print("{:c} Unicode character 69".format(69))
```

Output:

```
d Unicode character 100
E Unicode character 69
```

13.32.12 `:d` formatting type

Decimal format.

```
print("{:d} positive number".format(42069))
print("{:d} negative number".format(-42069))
print("{:d} zero".format(0))
```

Output:

```
42069 positive number
-42069 negative number
0 zero
```

13.32.13 `:e` formatting type

Scientific format, with a lower case e.

```
print("{:e} positive number".format(4200.696969))
print("{:.2e} negative number".format(-4200.696969))
print("{:.4e} zero".format(0))
```

Output:

```
4.200697e+03 positive number
-4.20e+03 negative number
0.0000e+00 zero
```

13.32.14 `:E` formatting type

Scientific format, with an upper case E.

```
print("{:E} positive number".format(4200.696969))
print("{:.2E} negative number".format(-4200.696969))
print("{:.4E} zero".format(0))
```

Output:

```
4.200697E+03 positive number
-4.20E+03 negative number
0.0000E+00 zero
```

13.32.15 `:f` formatting type

Fix point number format.

```
print("{:f} positive number".format(4200.696969))
print("{:.2f} negative number".format(-4200.696969))
print("{:.4f} zero".format(0))
```

Output:

```
4200.696969 positive number
-4200.70 negative number
0.0000 zero
```

13.32.16 `:F` formatting type

Fix point number format, in uppercase format (show inf and nan as INF and NAN).

```
print("{:F} positive number".format(4200.696969))
print("{:.2F} negative number".format(-4200.696969))
print("{:.4F} zero".format(0))
```

Output:

```
4200.696969 positive number
-4200.70 negative number
0.0000 zero
```

13.32.17 `:g` formatting type

General format.

```
print("{:g} positive number".format(4200.696969))
print("{:.2g} negative number".format(-4200.696969))
print("{:.4g} zero".format(0))
```

Output:

```
4200.7 positive number
-4.2e+03 negative number
0 zero
```

13.32.18 `:G` formatting type

General format (using an upper case E for scientific notations).

```
print("{:G} positive number".format(4200.696969))
print("{:.2G} negative number".format(-4200.696969))
print("{:.4G} zero".format(0))
```

Output:

```
4200.7 positive number
-4.2E+03 negative number
0 zero
```

13.32.19 `:o` formatting type

Octal format, only applicable to integers.

```
print("{:o} positive number".format(42069))
print("{:o} negative number".format(-42069))
print("{:o} zero".format(0))
```

Output:

```
122125 positive number
-122125 negative number
0 zero
```

13.32.20 `:x` formatting type

Hex format, lowercase.

```
print("{:x} positive number".format(42069))
print("{:x} negative number".format(-42069))
print("{:x} zero".format(0))
```

Output:

```
a455 positive number
-a455 negative number
0 zero
```

13.32.21 `:X` formatting type

Hex format, uppercase.

```
print("{:X} positive number".format(42069))
print("{:X} negative number".format(-42069))
print("{:X} zero".format(0))
```

Output:

```
A455 positive number
-A455 negative number
0 zero
```

13.32.22 `:n` formatting type

Number format.

```
print("{:n} positive number".format(4200.696969))
print("{:n} negative number".format(-4200.696969))
print("{:n} zero".format(0))
```

Output:

```
4200.7 positive number
-4200.7 negative number
0 zero
```

13.32.23 :% formatting type

Percentage format.

```
print("{:%} positive number".format(4200.696969))
print("{:.2%} negative number".format(-4200.696969))
print("{:.4%} zero".format(0))
```

Output:

```
420069.696900% positive number
-420069.70% negative number
0.0000% zero
```

13.33 List method: `list.append`

- This method takes an element, which can be any Python object, and adds it to the end of the list.
- Note that this method **changes the list** in place and does **not** return any value.

```
l = [1, "omg wow", 420.69, False]
l.append("lol")
print(l)
l.append(True)
print(l)
l.append(69)
print(l)
```

Output:

```
[1, 'omg wow', 420.69, False, 'lol']
[1, 'omg wow', 420.69, False, 'lol', True]
[1, 'omg wow', 420.69, False, 'lol', True, 69]
```

13.34 List method: `list.extend`

- This method takes a **list** and adds the elements of the given list to the end of the original list.
- This method cannot take an element to add to the back of the original list, use `list.append` for that.
- Using this method to add a string to the list will add all the characters in the string into the list instead of the string as one item.
- Note that this method **changes the list** in place and does **not** return any value.

```python
l_1 = [1, "omg wow", 420.69, False]
l_2 = ["extended l_1 hehe", "lovely", 6969, True]
l_1.extend(l_2)
print(l_1)
l_1.extend(5)     # Error
```

Output:

```
[1, 'omg wow', 420.69, False, 'extended l_1 hehe', 'lovely', 6969, True]
```

13.35 List method: `list.pop`

- This method takes an index and removes the item corresponding to that index from the list.
- Note that this method **changes the list** in place and does **not** return any value.

```python
l = [1, "omg wow", 420.69, False]
l.pop(2)
print(l)
```

Output:

```
[1, 'omg wow', False]
```

13.36 List method: `list.insert`

- This method takes an index and an element, which can be any Python object, and adds the element at the given index in the list.
- The element that was in the given index will be pushed back.
- Essentially, the given index will be the same index you can use to get the added element back from the list.
- Note that this method **changes the list** in place and does **not** return any value.

```
l = [1, "omg wow", 420.69, False]
l.insert(3, 6969)
print(l)
print(l[3])
l.insert(0, "I'm at the front")
print(l)
print(l[0])
```

Output:

```
[1, 'omg wow', 420.69, 6969, False]
6969
["I'm at the front", 1, 'omg wow', 420.69, 6969, False]
I'm at the front
```

13.37 List method: `list.remove`

- This method takes an element and removes it from the list.
- Note that this method **changes the list** in place and does **not** return any value.

```
l = [1, "omg wow", 420.69, False]
l.remove(420.69)
print(l)
```

Output:

```
[1, 'omg wow', False]
```

13.38 List method: `list.sort`

- This method sorts the list lexicographically.
- This means that everything is converted to a number if possible and the list can only be either a **list of strings** or a **list of numeric (`int` and `float`) and boolean values**.
- Strings will be compared using their ASCII character code.
- The boolean `True` is changed into 1 and the boolean `False` is changed to 0 when sorting a list.
- Note that this method **changes the list** in place and does **not** return any value.

```
l = [1, True, 420.69, False]
l.sort()
print(l)
l = ["ooo", "a", "list", "of", "Only", "Strings"]
l.sort()
print(l)
```

Output:

```
[False, 1, True, 420.69]
['Only', 'Strings', 'a', 'list', 'of', 'ooo']
```

13.39 List method: `list.reverse`

- This method reverses the list in place.
- This means that the method will **change the list** in place and **not** return any value.

```
l = [1, "omg wow", 420.69, False]
l.reverse()
print(l)
```

Output:

```
[False, 420.69, 'omg wow', 1]
```

13.40 Dictionary method: `dict.items`

This method returns all the key value pairs in a list-like structure called `dict_items`, containing the key value pairs as tuples.

```
dic = {
    "bill": 25,
    "tax": 3
}
print(dic.items())
```

Output:

```
dict_items([('bill', 25), ('tax', 3)])
```

13.41 Dictionary method: `dict.keys`

This method returns all the keys of the dictionary in a list-like structure called `dict_keys`.

```
dic = {
    "bill": 25,
    "tax": 3
}
print(dic.keys())
```

Output:

```
dict_keys(['bill', 'tax'])
```

13.42 Dictionary method: `dict.values`

This method returns all the values of the dictionary in a list-like structure called `dict_values`.

```
dic = {
    "bill": 25,
    "tax": 3
}
print(dic.values())
```

Output:

```
dict_values([25, 3])
```

13.43 Dictionary method: `dict.clear`

- This method clears the **dictionary**, which means the dictionary becomes an empty dictionary again.
- This method **changes** the dictionary in place and **does not** return any value.

```
dic = {
    "bill": 25,
    "tax": 3
}
dic.clear()
print(dic)
```

Output:

```
{}
```

13.44 Dictionary method: `dict.update`

- This method takes a dictionary and updates the original dictionary with the keys and values from the new dictionary.
- The **value in the new dictionary** is used if the new dictionary also has keys that are in the original dictionary
- This method **changes** the dictionary in place and **does not** return any value.

```
dic = {
    "bill": 25,
    "tax": 3
}
new_dic = {
    "bill": 50,
    "petrol": 20
}
dic.update(new_dic)
print(dic)
```

Output:

```
{'bill': 50, 'tax': 3, 'petrol': 20}
```

13.45 `open`

- Opens a file located on the disk.
- The function takes two strings, the first string contains either the name of the file or the file path.
- The second string tells Python what mode to open the file in.
- The default mode is read-only mode, or `"r"`.

Opening a file in the **current program directory** on Windows:

```
file = open("A_text_file.txt")
file = open(".\A_text_file.txt")
```

Opening a file in the **parent directory** on Windows:

```
file = open("..\A_text_file.txt")
```

Opening a file using its **absolute path** on Windows:

```
file = open("C:\Users\Draykoth\Documents\A_text_file.txt")
```

13.45.1 File modes

Mode String	Mode	File exists	File doesn't exist
"r"	Read-only	Opens the file.	Error
"w"	Write-only	Clears the file contents.	Creates and opens a new file
"a"	Write-only	The file contents are left intact and new data is appended at the end of the file.	Creates and opens a new file
"r+"	Read and write	Reads and overwrites from the beginning of the file.	Error
"w+"	Read and write	Clears the file contents.	Creates and opens a new file
"a+"	Read and write	The file contents are left intact and reading and writing starts at the end of the file.	Creates and opens a new file

13.45.2 `encoding` parameter (optional)

- `encoding` takes a **string** that specifies the file encoding for reading the file.
- The default value is `"UTF-8"`, which is for text files.
- Binary files, depending on language or operating system, will have **different** encodings, and hence the encoding for binary files needs to be specified explicitly, like the example below.
- For more information about encodings, visit this website.

```
open("table.csv", "r", encoding="windows-1252")
```

13.45.3 Reading a file

```
# Opens the file in read-only mode
file = open("A_text_file.txt", "r")

# Iterates over the file and prints the lines.
# No end character as the lines in the file already
# include the new line character.
for line in file:
    print(line, end="")

# Close the file
file.close()
```

Output:

```
First line: €
Second line
Third line
Fourth line
Fifth line
```

13.46 File method: `file.close`

- This method closes the file.
- It is important to call this method once you're done with the file to save your changes to the disk.
- The information in the file buffer is "flushed" out of the buffer and into the file on the disk when this method is called.
- If you don't call this method, your changes WILL NOT be saved to the disk.

```
file.close()
```

13.46.1 Automatically closing a file

- The file is opened at the **`with`** statement.
- The file mode is read-only, and the file type is text in the example below.
- The file is automatically closed at the end of the **`with`** block.

```python
with open("A_text_file.txt", "r") as file:
    for line in file:
        print(line)
```

Output:

```
First line: €

Second line

Third line

Fourth line

Fifth line
```

13.47 File method: `file.readline`

- This method returns the next line as a **string**.
- This method also moves the current file position forward.
- When the current file position reaches the end of the file, every read will yield an empty string `""`.
- Using the `seek` method is required to read the file again.
- Closing and reopening the file also works, but is far less efficient.

```
file = open("A_text_file.txt", "r")

# No end character as the lines in the file already
# include the new line character
print(file.readline(), end="")
print(file.readline(), end="")
print(file.readline(), end="")
file.close()
```

Output:

```
First line: €
Second line
Third line
```

13.48 File method: `file.readlines`

- This method returns a list of all the lines from the file.
- This method also moves the current file position forward.
- When the current file position reaches the end of the file, every read will yield an empty string `""`.
- Using the `seek` method is required to read the file again.
- Closing and reopening the file also works, but is far less efficient.

```
file = open("A_text_file.txt", "r")
print(file.readlines())
file.close()
```

Output:

```
['First line: €\n', 'Second line\n', 'Third line\n', 'Fourth line\n', 'Fifth line\n']
```

13.49 File method: `file.read`

- This method takes an **integer**, which represents how many **characters** is to be read from the file.
- If this integer is not given, then the entire file is read and returned as a single string.
- This method also moves the current file position forward.
- When the current file position reaches the end of the file, every read will yield an empty string `""`.
- Using the `seek` method is required to read the file again.
- Closing and reopening the file also works, but is far less efficient.

```
file = open("A_text_file.txt", "r")
print(file.read(5))
file.close()
```

Output:

```
First
```

```
file = open("A_text_file.txt", "r")
print(file.read())
file.close()
```

Output:

```
First line: €
Second line
Third line
Fourth line
Fifth line
```

13.50 File method: `file.write`

This method takes a **string** and writes it to the file.

```
file = open("temp.txt", "w")
file.write("First line\n")
file.write("Second line\n")
file.close()
```

13.51 File method: `file.writelines`

This method takes a **list of strings** and writes it to the file.

```
file = open("temp.txt", "w")
lines = ["First line\n", "Second line\n"]
file.writelines(lines)
file.close()
```

13.52 File method: `file.tell`

- This method tells returns an **integer** representing the **current file position**.
- The positions are in **bytes** from the **beginning** of the file.
- This is not necessarily the same as the number of characters, as it depends on encoding.
- Some characters may take multiple bytes, like the euro symbol in the example below.

```
file = open("A_text_file.txt", "r")
print(file.tell())
print(file.read(11))
print(file.tell())
print(file.read(1))
print(file.tell())
print(file.read(1))
print(file.tell())
file.close()
```

Output:

```
0
First line:
11

12
€
15
```

13.53 File method: `file.seek`

This method takes an **integer** representing the offset in **bytes** from the **beginning** of the file and updates the current file position that position.

```
# Seek to the beginning of the file
file.seek(0)

# Seek to 100 bytes from the beginning of the file
file.seek(100)
```

13.53.1 Example

```
file = open("A_text_file.txt", "r")
print(file.read(12))
print(file.tell())
print(file.read(1))
print(file.tell())
print(file.seek(6))
print(file.read(4))
file.close()
```

Output:

```
First line:
12
€
15
6
line
```

13.53.2 `whence` parameter (optional)

- `whence` takes an **integer**, either `0`, `1` or `2`, and represents file positioning.
- This parameter defaults to `0`, which is to count from the beginning of the file.
  - `0` means to count from the beginning of the file.
  - `1` means to count from the current position in the file.
  - `2` means to count from the end of the file. This option is usually paired with a negative offset.
- In text files, only seeks relative to the beginning of the file are allowed.
- An exception to this is the seek to the end of the file, like `file.seek(0, 2)`.

```
# Go to 100 bytes before the end of the file
file.seek(-100, 2)

# Go to 5 bytes from the current file position
file.seek(5, 1)
```

13.54 `math` module method: `math.floor`

- This method takes a number and returns an **integer (`int`)** rounded **down** to the nearest integer.
- It works the same as the floor division operator (//), except that it always returns an **integer (`int`)**.
- The `math` module must be **imported before** using this function.

```python
import math
print(math.floor(6))
print(math.floor(6.9))
print(math.floor(420.42))
```

Output:

```
6
6
420
```

13.55 `math` module method: `math.sin`

- This method takes an angle in **radians** and returns the sine of the angle.
- In general, all trigonometric functions in the `math` module take an angle in **radians**.
- The `math` module must be **imported before** using this function.

```python
import math
print(math.sin(math.pi / 2))
print(math.sin(0))
print(math.sin(90))
```

Output:

```
1.0
0.0
0.8939966636005579
```

13.56 `math` module method: `math.cos`

- This method takes an angle in **radians** and returns the cosine of the angle.
- In general, all trigonometric functions in the `math` module take an angle in **radians**.
- The `math` module must be **imported before** using this function.

```
import math
print(math.cos(math.pi / 2))
print(math.cos(0))
print(math.cos(90))
```

Output:

```
6.123233995736766e-17
1.0
-0.4480736161291701
```

13.57 `operator` module method: `operator.itemgetter`

- This method takes **indexes** for a list or tuple, or **keys** for a dictionary and returns a **function**.
- When there is **only one** index or key given, the **value** is returned.
- When there is **more than one** index or key given, a **tuple** of the values is returned.
- This method can be used with the `sorted` function to get the value for sorting.

```python
import operator

num_list = [10, 6, 3, 1, 9]
num_dict = {
    10: "ten",
    6: "six",
    3: "three",
    1: "one",
    9: "nine"
}

# Print the second item of the list
print(operator.itemgetter(1)(num_list))

# Sort the dictionary based on the values of the dictionary,
# which is to sort by alphabetical order
print(sorted(num_dict.items(), key=operator.itemgetter(1)))
```

Output:

```
6
[(9, 'nine'), (1, 'one'), (6, 'six'), (10, 'ten'), (3, 'three')]
```

14 Operator precedence

Precedence	**Associativity**	**Operator**	**Description**
18	Left-to-right	`()`	Parentheses (grouping)
17	Left-to-right	`f(args...)`	Function call
16	Left-to-right	`x[index:index]`	Slicing
15	Left-to-right	`x[index]`	Indexing an array
14	Right-to-left	`**`	Exponentiation
13	Left-to-right	`~x`	Bitwise not
12	Left-to-right	`+x`	Positive
12	Left-to-right	`-x`	Negative
11	Left-to-right	`*`	Multiplication
11	Left-to-right	`/`	Division
11	Left-to-right	`%`	Modulo (Remainder operator)
10	Left-to-right	`+`	Addition
10	Left-to-right	`-`	Subtraction
9	Left-to-right	`<<`	Bitwise left shift
9	Left-to-right	`>>`	Bitwise right shift
8	Left-to-right	`&`	Bitwise AND
7	Left-to-right	`^`	Bitwise XOR
6	Left-to-right	|	Bitwise OR
5	Left-to-right	`in`, `not in`, `is`, `is not`	Membership
5	Left-to-right	`<`, `<=`, `>`, `>=`	Relational
5	Left-to-right	`<>`, `==`	Equality
5	Left-to-right	`!=`	Inequality
4	Left-to-right	`not x`	Boolean NOT
3	Left-to-right	`and`	Boolean AND
2	Left-to-right	`or`	Boolean OR
1	Left-to-right	`lambda`	Lambda expression

15 ASCII Table

DEC	OCT	HEX	BIN	Symbol	Description
32	040	20	00100000	SP	Space
33	041	21	00100001	!	Exclamation mark
34	042	22	00100010	"	Double quotes (or speech marks)
35	043	23	00100011	#	Number sign
36	044	24	00100100	$	Dollar
37	045	25	00100101	%	Per cent sign
38	046	26	00100110	&	Ampersand
39	047	27	00100111	’	Single quote
40	050	28	00101000	(	Open parenthesis (or open bracket)
41	051	29	00101001	)	Close parenthesis (or close bracket)
42	052	2A	00101010	*	Asterisk
43	053	2B	00101011	+	Plus
44	054	2C	00101100	,	Comma
45	055	2D	00101101	-	Hyphen-minus
46	056	2E	00101110	.	Period, dot or full stop
47	057	2F	00101111	/	Slash or divide
48	060	30	00110000	0	Zero
49	061	31	00110001	1	One
50	062	32	00110010	2	Two
51	063	33	00110011	3	Three
52	064	34	00110100	4	Four
53	065	35	00110101	5	Five
54	066	36	00110110	6	Six
55	067	37	00110111	7	Seven
56	070	38	00111000	8	Eight
57	071	39	00111001	9	Nine
58	072	3A	00111010	:	Colon
59	073	3B	00111011	;	Semicolon
60	074	3C	00111100	<	Less than (or open angled bracket)
61	075	3D	00111101	=	Equals
62	076	3E	00111110	>	Greater than (or close angled bracket)
63	077	3F	00111111	?	Question mark
64	100	40	01000000	@	At sign
65	101	41	01000001	A	Uppercase A
66	102	42	01000010	B	Uppercase B

Continued on next page

Continued from previous page

DEC	OCT	HEX	BIN	Symbol	Description
67	103	43	01000011	C	Uppercase C
68	104	44	01000100	D	Uppercase D
69	105	45	01000101	E	Uppercase E
70	106	46	01000110	F	Uppercase F
71	107	47	01000111	G	Uppercase G
72	110	48	01001000	H	Uppercase H
73	111	49	01001001	I	Uppercase I
74	112	4A	01001010	J	Uppercase J
75	113	4B	01001011	K	Uppercase K
76	114	4C	01001100	L	Uppercase L
77	115	4D	01001101	M	Uppercase M
78	116	4E	01001110	N	Uppercase N
79	117	4F	01001111	O	Uppercase O
80	120	50	01010000	P	Uppercase P
81	121	51	01010001	Q	Uppercase Q
82	122	52	01010010	R	Uppercase R
83	123	53	01010011	S	Uppercase S
84	124	54	01010100	T	Uppercase T
85	125	55	01010101	U	Uppercase U
86	126	56	01010110	V	Uppercase V
87	127	57	01010111	W	Uppercase W
88	130	58	01011000	X	Uppercase X
89	131	59	01011001	Y	Uppercase Y
90	132	5A	01011010	Z	Uppercase Z
91	133	5B	01011011	[	Opening bracket
92	134	5C	01011100	\	Backslash
93	135	5D	01011101	]	Closing bracket
94	136	5E	01011110	ˆ	Caret - circumflex
95	137	5F	01011111	_	Underscore
96	140	60	01100000	‘	Grave accent
97	141	61	01100001	a	Lowercase a
98	142	62	01100010	b	Lowercase b
99	143	63	01100011	c	Lowercase c
100	144	64	01100100	d	Lowercase d
101	145	65	01100101	e	Lowercase e
102	146	66	01100110	f	Lowercase f
103	147	67	01100111	g	Lowercase g

Continued on next page

Continued from previous page

DEC	OCT	HEX	BIN	Symbol	Description
104	150	68	01101000	h	Lowercase h
105	151	69	01101001	i	Lowercase i
106	152	6A	01101010	j	Lowercase j
107	153	6B	01101011	k	Lowercase k
108	154	6C	01101100	l	Lowercase l
109	155	6D	01101101	m	Lowercase m
110	156	6E	01101110	n	Lowercase n
111	157	6F	01101111	o	Lowercase o
112	160	70	01110000	p	Lowercase p
113	161	71	01110001	q	Lowercase q
114	162	72	01110010	r	Lowercase r
115	163	73	01110011	s	Lowercase s
116	164	74	01110100	t	Lowercase t
117	165	75	01110101	u	Lowercase u
118	166	76	01110110	v	Lowercase v
119	167	77	01110111	w	Lowercase w
120	170	78	01111000	x	Lowercase x
121	171	79	01111001	y	Lowercase y
122	172	7A	01111010	z	Lowercase z
123	173	7B	01111011	{	Opening brace
124	174	7C	01111100	|	Vertical bar
125	175	7D	01111101	}	Closing brace
126	176	7E	01111110	~	Equivalency sign - tilde
127	177	7F	01111111	DEL	Delete