types of hash function in python in table and dictionary
hash function is a function that can map a piece of data of any length to a fixed-length value

Hash functions have three major characteristics:

fast to compute: calculate the hash of a piece of data have to be a fast operation.

deterministic: the same string will always produce the same hash.

produce fixed-length values: it doesn't matter if your input is one, ten, or ten thousand bytes, the resulting hash will be always of a fixed, predetermined length.

hash function h maps keys of a given type to integers in a fixed interval [0, N − 1]
A hash table for a given key type consists of
Hash function h
Array (called table) of size N
When implementing a dictionary with a hash table, the goal is to store item (k, o) at index i = h (k)
A hash function is usually specified as the composition of two functions:
Hash code map: h1:
keys → integers
Compression map: h 2:
integers → [0, N − 1]
The hash code map is applied first, and the compression map is applied next on the result for
h (x) = h 2 (h1(x))

Hash Code Maps
Memory address:
Reinterpret the memory address of the key object as an integer
Integer cast:
We reinterpret the bits of the key as an integer
Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., char, short, int and float on many machines)
Component sum:
We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double on many machines)

set based on hash table in data structure
sets are unordered, we cannot access items using indexes as we do in lists.

we can move the cursor to the previous line using the “ escape sequence”. This sequence moves the cursor to the beginning of the previous line.

Seek()
This method is used to change the position of the pointer
File pointer position is always counted from the beginning
Syntax
File.seek(position)

clean code book
chapter 1
This Book is about good programming. It's about how to write good code, and how to transform bad code into good code.
The code represents the detail of the requirements and the details cannot be ignored or abstracted. We may create languages that are closer to the requirements. We can create tools that help us parse and assemble those requirements into formal structures. But we will never eliminate necessary precision.

Why write bad code?

- Are you in a rush?
- Do you try to go "fast"?
- Do not you have time to do a good job?
- Are you tired of work in the same program/module?
- Does your Boss push you to finish soon?

chapter 2
Names are everywhere in software. Files, directories, variables functions, etc. Because we do so much of it. We have better do it well.

Use Intention-Revealing Names

It is easy to say that names reveal intent. Choosing good names takes time, but saves more than it takes. So take care with your names and change them when you find better ones.
The name of a variable, function or class, should answer all the big questions. It should tell you why it exists, what it does, and how is used. If a name requires a comment, then the name does not reveals its intent

chapter 3
Functions are the first line of organization in any topic.

Small!!

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

Blocks and Indenting

This implies that the blocks within `if` statements, `else` statements, `while` statements, and so on should be one line long. Probably that line should be a function call. Not only does this keep the enclosing function small, but also adds documentary value because the function called within the block can have a nicely descriptive name.
This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This, of course, makes the functions easy to read and understand.

chapter 4
Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old comment that propagates lies and misinformation.
If our programming languages were expressive enough, or if we had the talent to subtly wield those languages to express our intent, we would not need comments very much—perhaps not at all.

Comments Do Not Make Up for Bad Code

Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

chapter 5
Code formatting is important. It is too important to ignore and it is too important to treat religiously. Code formatting is about communication, and communication is the professional developer's first order of business.

Vertical Formatting

Vertical Openness Between Concepts

This concept consist in how to you separate concepts in your code, In the next example we can appreciate it.

chapter 6

Data Abstraction

Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the essence of the data, without having to know its implementation.

Data/Object Anti-Symmetry

These two examples show the difference between objects and data structures. Objects hide their data behind abstractions and expose functions that operate on that data. Data structure expose their data and have no meaningful functions.

chapter 7
Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, but if it obscures logic, it's wrong.

Use Exceptions Rather Than Return Codes

Back in the distant past there were many languages that didn't have exceptions. In those languages the techniques for handling and reporting errors were limited. You either set an error flag or returned an error code that the caller could check

Write Your Try-Catch-Finally Statement First

In a way, try blocks are like transactions. Your catch has to leave your program in a consistent state, no matter what happens in the try. For this reason it is good practice to start with a try-catch-finally statement when you are writing code that could throw exceptions. This helps you define what the user of that code should expect, no matter what goes wrong with the code that is executed in the try.

Provide Context with Exceptions

Each exception that you throw should provide enough context to determine the source and location of an error.
Create informative error messages and pass them along with your exceptions. Mention the operation that failed and the type of failure. If you are logging in your application, pass along enough information to be able to log the error in your catch.

chapter 8
We seldom control all the software in our systems. Sometimes we buy third-party pack- ages or use open source. Other times we depend on teams

in our own company to produce components or subsystems for us. Somehow we must cleanly integrate this foreign code with our own.

Using Third-Party Code

There is a natural tension between the provider of an interface and the user of an interface. Providers of third-party packages and frameworks strive for broad applicability so they can work in many environments and appeal to a wide audience. Users, on the other hand, want an interface that is focused on their particular needs. This tension can cause problems at the boundaries of our systems.

chapter 9
Test Driven Development

The Three Laws of TDD

- First Law You may not write production code until you have written a failing unit test.
- Second Law You may not write more of a unit tests than is sufficient to fail, and not comipling is failing.
- Third Law You may not write more production code than is sufficient to pass the currently failing tests.

Clean Tests

If you don't keep your tests clean, you will lose them.
The readability it's very important to keep clean your tests.

One Assert per test

It's recomendable maintain only one asserts per tests, because this helps to maintain each tests easy to understand.

Single concept per Test

This rule will help you to keep short functions.

- Write one test per each concept that you need to verify

chapter 10

Class Organization

Encapsulation

We like to keep our variables and utility functions small, but we're not fanatic about it. Sometimes we need to make a variable or utility function protected so that it can be accessed by a test.

chapter 11

Separate Constructing a System from using It

Software Systems should separate the startup process, when the application objects are constructed and the dependencies are "wired" together, from the runtime logic that takes over after startup

Separation from main

One way to separate construction from use is simply to move all aspects of construction to `main`, or modules called by `main`, and to design the rest of the system assuming that all objects have been created constructed and wired up appropriately.
The Abstract Factory Pattern is an option for this kind of approach

chapter 12
According to Kent Beck, a design is "simple" if it follows these rules

- Run all tests
- Contains no duplication
- Expresses the intent of programmers
- Minimizes the number of classes and method

chapter 13
Concurrence is a decoupling strategy. It helps us decouple what gets fone from when it gets done. In single-threaded applications what and when are so strongly coupled that the state of the entire application can often be determined by looking at the stack backtrace. A programmer who debugs such a system can set a breakpoint, or a sequence of breakpoints, and know the state of the system by which breakpoints are hit.
Decoupling what from when can dramatically improve both the throughput and structures of an application. From a structural point of view the application looks like many lit- tle collaborating computers rather than one big main loop. This can make the system easier to understand and offers some powerful ways to separate concerns.