Table of Contents

[Week1 2](#_Toc50355752)

[Week2 3](#_Toc50355753)

[Week4 3](#_Toc50355754)

[Week7 4](#_Toc50355755)

[Testing 4](#_Toc50355756)

[Black-Box testing 4](#_Toc50355757)

[Glass-Box testing 4](#_Toc50355758)

[Conducting Test 5](#_Toc50355759)

[Debugging 6](#_Toc50355760)

[Learning to Debug 7](#_Toc50355761)

[Designing the Experiments 7](#_Toc50355762)

[When the Going Gets Tough 7](#_Toc50355763)

[And When You Have Found “The” Bug 8](#_Toc50355764)

[EXCEPTIONS 9](#_Toc50355765)

[Handling Exceptions 9](#_Toc50355766)

[Exceptions as a Control Flow Mechanism 9](#_Toc50355767)

[Assertion 10](#_Toc50355768)

[Lecture8 11](#_Toc50355769)

[Lecture9 12](#_Toc50355770)

[Week10-12 13](#_Toc50355771)

[A SIMPLISTIC INTRODUCTION TO ALGORITHMIC COMPLEXITY 13](#_Toc50355772)

[Thinking About Computational Complexity 13](#_Toc50355773)

[Some Important Complexity Classes 14](#_Toc50355774)

[SOME SIMPLE ALGORITHMS AND DATA STRUCTURES 14](#_Toc50355775)

[Search Algorithms 14](#_Toc50355776)

[Sorting Algorithms 15](#_Toc50355777)

[Hash Tables 16](#_Toc50355778)

# Week1

All knowledge can be thought of as either declarative or imperative. **Declarative knowledge** is composed of statements of fact. **Imperative knowledge** is “how to” knowledge, or recipes for deducing information.

an **algorithm** is a finite list of instructions that describe a **computation** that when executed on a provided set of inputs will proceed through a set of well-defined states and eventually produce an output.

Indeed, the heart of the computer then becomes a program (called an **interpreter**) that can execute any legal set of instructions, and thus can be used to compute anything that one can describe using some basic set of instructions.

In some cases, it performs a test, and on the basis of that test, execution may jump to some other point in the sequence of instructions. This is called **flow of control**

Semantics：语义语法，计算机程序中只能有一种意思，不能存在歧义性。

A Python **program**, sometimes called a **script**, is a sequence of definitions and commands. These definitions are evaluated and the commands are executed by the Python interpreter in something called the **shell**.

A **command**, often called a **statement**, instructs the interpreter to do something.

**Objects** are the core things that Python programs manipulate. Every object has a **type** that defines the kinds of things that programs can do with objects of that type.

Types are either scalar or non-scalar. **Scalar** objects are indivisible. Think of them as the atoms of the language.7 Non-scalar objects, for example strings, have internal structure.

Objects and **operators** can be combined to form **expressions**, each of which evaluates to an object of some type.

The symbol >>> is a **shell prompt** indicating that the interpreter is expecting the user to type some Python code into the shell.

**Variables** provide a way to associate names with objects.

In Python, **a variable is just a name,** nothing more. Remember this—it is important. An **assignment** statement associates the name to the left of the = symbol with the object denoted by the expression to the right of the =.

Typing programs directly into the shell is highly inconvenient. Most programmers prefer to use some sort of text editor that is part of an **integrated development environment** (**IDE**).

Summarize：give definitions in basic computer programs.

# Week2

The kinds of computations we have been looking at thus far are called **straight- line programs**. They execute one statement after another in the order in which they appear, and stop when they run out of statements. **Branching** programs are more interesting. The simplest branching statement is a **conditional**.

When either the true block or the false block of a conditional contains another conditional, the conditional statements are said to be **nested**.

A program for which the maximum running time is bounded by the length of the program is said to run in **constant time**.

The operator + is said to be **overloaded**: It has different meanings depending upon the types of the objects to which it is applied.

That **type checking** exists is a good thing. It turns careless (and sometimes subtle) mistakes into errors that stop execution, rather than errors that lead programs to behave in mysterious ways.

**Indexing** can be used to extract individual characters from a string. **Slicing** is used to extract substrings of arbitrary length.

**Type conversions** (also called **type casts**) are used often in Python code.

A generic **iteration** (also called **looping**) mechanism is depicted in Figure 2.4. Like a conditional statement it begins with a test. If the test evaluates to True, the program executes the **loop body** once, and then goes back to reevaluate the test. This process is repeated until the test evaluates to False, after which control passes to the code following the iteration statement.

The process continues until the sequence is exhausted or a **break** statement is executed within the code block.

# Week4

By convention, Python programmers use docstrings to provide specifications of functions. These docstrings can be accessed using the built-in function **help**. The text between the triple quotation marks is called a **docstring** in Python.

**Decomposition** creates structure. It allows us to break a problem into modules that are reasonably self-contained, and that may be reused in different settings.

**Abstraction** hides detail. It allows us to use a piece of code as if it were a black box—that is, something whose interior details we cannot see, don’t need to see, and shouldn’t even want to see.21 The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. The key to using abstraction effectively in programming is finding a notion of relevance that is appropriate for both the builder of an abstraction and the potential clients of the abstraction. That is the true art of programming.

A **module** is a .py file containing Python definitions and statements.

# Week7

**Testing** is the process of running a program to try and ascertain whether or not it works as intended. **Debugging** is the process of trying to fix a program that you already know does not work as intended.

The key to testing is finding a collection of inputs, called a **test suite**, that has a high likelihood of revealing bugs, yet does not take too long to run. The key to doing this is partitioning the space of all possible inputs into subsets that provide equivalent information about the correctness of the program, and then constructing a test suite that contains one input from each partition. (Usually, constructing such a test suite is not actually possible. Think of this as an unachievable ideal.)

A **partition** of a set divides that set into a collection of subsets such that each element of the original set belongs to exactly one of the subsets.

For most programs, finding a good partitioning of the inputs is far easier said than done. Typically, people rely on **heuristics based on exploring different paths through some combination of the code and the specifications**. Heuristics based on exploring paths through the **code** fall into a class called **glass-box testing**. Heuristics based on exploring paths through the **specification** fall into a class called **black-box testing**.

## Testing

### Black-Box testing

基于文档（docstring）设计的testing

分离性，防止编程者针对自己的程序创造testing

In principle, black-box tests are constructed without looking at the code to be tested. Black-box testing allows testers and implementers to be drawn from separate populations.

Another positive feature of black-box testing is that it is robust with respect to implementation changes. Since the test data is generated without knowledge of the implementation, it need not be changed when the implementation is changed.

Boundary conditions should also be tested. When looking at lists, this often means looking at the empty list, a list with exactly one element, and a list containing lists. When dealing with numbers, it typically means looking at very small and very large values as well as “typical” values. For sqrt, it might make sense to try values of x and epsilon similar to those in the following table.

Another important boundary condition to think about is aliasing.

### Glass-Box testing

根据代码内容和细节设计的testing

Glass-box test suites are usually much easier to construct than black-box test suites. Specifications are usually incomplete and often pretty sloppy, making it a challenge to estimate how thoroughly a black-box test suite explores the space of interesting inputs. In contrast, the notion of a path through code is well defined, and it is relatively easy to evaluate how thoroughly one is exploring the space. There are, in fact, commercial tools that can be used to objectively measure the completeness of glass-box tests.

A glass-box test suite is **path-complete** if it exercises every potential path through the program. This is typically impossible to achieve, because it depends upon the number of times each loop is executed and the depth of each recursion.

Despite the limitations of glass-box testing, there are a few rules of thumb that are usually worth following:

* Exercise both branches of all if statements.
* Make sure that each except clause (see Chapter 7) is executed.
* For each for loop, have test cases in which

o The loop is not entered (e.g., if the loop is iterating over the elements of a list, make sure that it is tested on the empty list),

o The body of the loop is executed exactly once, and

o The body of the loop is executed more than once.

* For each while loop,

o Look at the same kinds of cases as when dealing with for loops, and

o Include test cases corresponding to all possible ways of exiting the loop. For example, for a loop starting with

while len(L) > 0 and not L[i] == e  
find cases where the loop exits because len(L) is greater than

zero and cases where it exits because L[i] == e.

* For recursive functions, include test cases that cause the function to return with no recursive calls, exactly one recursive call, and more than one recursive call.

### Conducting Test

Testing is often thought of as occurring in two phases. One should always start with **unit testing**. During this phase testers construct and run tests designed to ascertain whether individual units of code (e.g., functions) work properly. This is followed by **integration testing**, which is designed to ascertain whether the program as a whole behaves as intended. In practice, testers cycle through these two phases, since failures during integration testing lead to making changes to individual units.

Integration testing is almost always more challenging than unit testing. One reason for this is that the intended behavior of an entire program is often considerably harder to characterize than the intended behavior of each of its parts. For example, characterizing the intended behavior of a word processor is considerably more challenging than characterizing the behavior of a function that counts the number of characters in a document. Problems of scale can also make integration testing difficult. It is not unusual for integration tests to take hours or even days to run.

Many industrial software development organizations have a **software quality assurance (SQA)** group that is separate from the group charged with implementing the software. The mission of this group is to insure that before the software is released it is suitable for its intended purpose. In some organizations the development group is responsible for unit testing and the QA group for integration testing.

In industry, the testing process is often highly automated. Testers30 do not sit at terminals typing inputs and checking outputs. Instead, they use **test drivers** that autonomously

* Set up the environment needed to invoke the program (or unit) to be tested,
* Invoke the program (or unit) to be tested with a predefined or automatically generated sequence of inputs,
* Save the results of these invocations,
* Check the acceptability of the results of the tests, and
* Prepare an appropriate report.

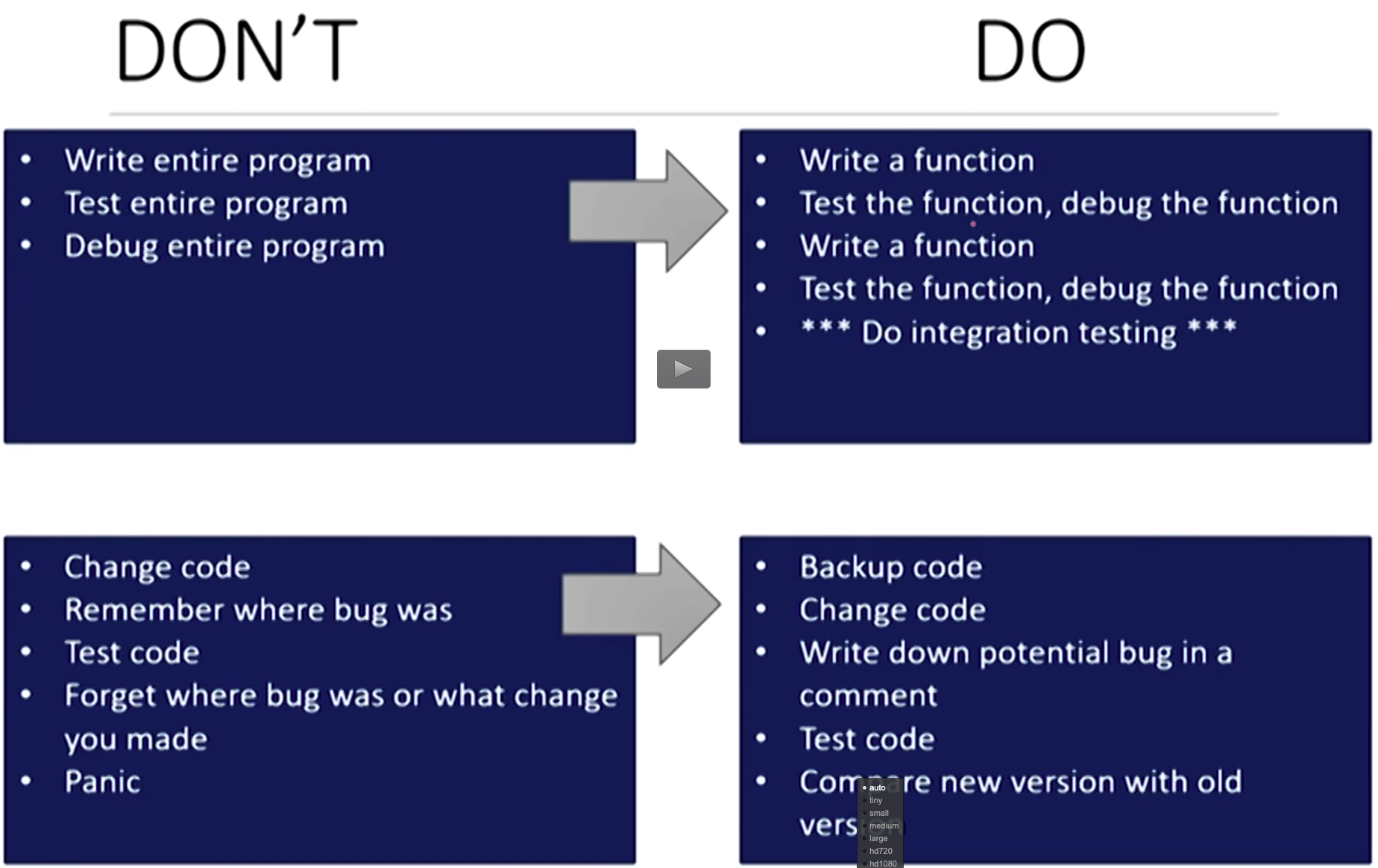
During unit testing, we often need to build **stubs** as well as drivers. Drivers simulate parts of the program that use the unit being tested, whereas stubs simulate parts of the program used by the unit being tested. Stubs are useful because they allow people to test units that depend upon software or sometimes even hardware that does not yet exist. This allows teams of programmers to simultaneously develop and test multiple parts of a system.

Ideally, a stub should

1. Check the reasonableness of the environment and arguments supplied by the caller (calling a function with inappropriate arguments is a common error),
2. Modify arguments and global variables in a manner consistent with the specification, and
3. Return values consistent with the specification.

One attraction of automating the testing process is that it facilitates **regression testing**. As programmers attempt to debug a program, it is all too common to install a “fix” that breaks something that used to work. Whenever any change is made, no matter how small, you should check that the program still passes all of the tests that it used to pass.

## Debugging



Runtime bugs can be categorized along two dimensions:

1. **Overt** →**covert**: An **overt bug** has an obvious manifestation, e.g., the program crashes or takes far longer (maybe forever) to run than it should. A **covert bug** has no obvious manifestation. The program may run to conclusion with no problem—other than providing an incorrect answer. Many bugs fall between the two extremes, and whether or not the bug is overt can depend upon how carefully one examines the behavior of the program.
2. **Persistent** →**intermittent**: A **persistent bug** occurs every time the program is run with the same inputs. An **intermittent bug** occurs only some of the time, even when the program is run on the same inputs and seemingly under the same conditions. When we get to Chapter 12, we will start writing programs of the kind where intermittent bugs are common.

Good programmers try to write their programs in such a way that programming mistakes lead to bugs that are both overt and persistent. This is often called **defensive programming**.

### Learning to Debug

Debugging starts when testing has demonstrated that the program behaves in undesirable ways. Debugging is the process of searching for an explanation of that behavior. The key to being consistently good at debugging is being systematic in conducting that search.

Start by studying the available data. This includes the test results and the program text. Study all of the test results. Examine not only the tests that revealed the presence of a problem, but also those tests that seemed to work perfectly. Trying to understand why one test worked and another did not is often illuminating. When looking at the program text, keep in mind that you don’t completely understand it. If you did, there probably wouldn’t be a bug.

Next, form a hypothesis that you believe to be consistent with all the data. The hypothesis could be as narrow as “if I change line 403 from x < y to x <= y, the problem will go away” or as broad as “my program is not terminating because I have the wrong exit condition in some while loop.”

Next, design and run a repeatable experiment with the potential to refute the hypothesis. For example, you might put a print statement before and after each while loop. If these are always paired, then the hypothesis that a while loop is causing nontermination has been refuted. Decide before running the experiment how you would interpret various possible results. If you wait until

after you run the experiment, you are more likely to fall prey to wishful thinking.

Finally, it’s important to keep a **record** of what experiments you have tried. When you’ve spent many hours changing your code trying to track down an elusive bug, it’s easy to forget what you have already tried. If you aren’t careful, it is easy to waste way too many hours trying the same experiment (or more likely an experiment that looks different but will give you the same information) over and over again. Remember, as many have said, “insanity is doing the same thing, over and over again, but expecting different results.”34

### Designing the Experiments

Think of debugging as a search process, and each experiment as an attempt to reduce the size of the search space. One way to reduce the size of the search space is to design an experiment that can be used to decide whether a specific region of code is responsible for a problem uncovered during integration testing. Another way to reduce the search space is to reduce the amount of test data needed to provoke a manifestation of a bug.

Often the best way to do this is to conduct a **binary search**. Find some point about halfway through the code, and devise an experiment that will allow you to decide if there is a problem before that point that might be related to the symptom. (Of course, there may be problems after that point as well, but it is usually best to hunt down one problem at a time.) In choosing such a point, look for a place where there are some easily examined intermediate values that provide useful information. If an intermediate value is not what you expected, there is probably a problem that occurred prior to that point in the code. If the intermediate values all look fine, the bug probably lies somewhere later in the code. This process can be repeated until you have narrowed the region in which a problem is located to a few lines of code.

### When the Going Gets Tough

This subsection contains a few pragmatic hints about what do when the debugging gets tough.

* *Look for the usual suspects*. E.g., have you

o Passed arguments to a function in the wrong order,

o Misspelled a name, e.g., typed a lowercase letter when you should have typed an uppercase one,

o Failed to reinitialize a variable,

o Tested that two floating point values are equal (==) instead of nearly equal (remember that floating point arithmetic is not the same as the arithmetic you learned in school),

o Tested for value equality (e.g., compared two lists by writing the expression L1 == L2) when you meant object equality (e.g., id(L1) == id(L2)),

o Forgotten that some built-in function has a side effect,

o Forgotten the () that turns a reference to an object of type function into a function invocation,

o Created an unintentional alias, or

o Made any other mistake that is typical for you.

* ***Stop asking yourself why the program isn’t doing what you want it to. Instead, ask yourself why it is doing what it is****.* That should be an easier question to answer, and will probably be a good first step in figuring out how to fix the program.
* ***Keep in mind that the bug is probably not where you think it is****.* If it were, you would probably have found it long ago. One practical way to go about deciding where to look is asking where the bug cannot be. As Sherlock Holmes said, “Eliminate all other factors, and the one which remains must be the truth.”37
* ***Try to explain the problem to somebody else****.* We all develop blind spots. It is often the case that merely attempting to explain the problem to someone will lead you to see things you have missed. A good thing to try to explain is why the bug cannot be in certain places.
* ***Don’t believe everything you read****.* In particular, don’t believe the documentation. The code may not be doing what the comments suggest.
* ***Stop debugging and start writing documentation****.* This will help you approach the problem from a different perspective.
* ***Walk away, and try again tomorrow***. This may mean that bug is fixed later in time than if you had stuck with it, but you will probably spend a lot less of your time looking for it. That is, it is possible to trade latency for efficiency. (Students, this is an excellent reason to start work on programming problem sets earlier rather than later!)

### And When You Have Found “The” Bug

When you think you have found a bug in your code, the temptation to start coding and testing a fix is almost irresistible. It is often better, however, to slow down a little. **Remember that the goal is not to fix one bug, but to move rapidly and efficiently towards a bug-free program.**

Ask yourself if this bug explains all the observed symptoms, or whether it is just the tip of the iceberg. If the latter, it may be better to think about taking care of this bug in concert with other changes

Before making any change, try and understand the ramification of the proposed “fix.” Will it break something else? Does it introduce excessive complexity? Does it offer the opportunity to tidy up other parts of the code?

**Always make sure that you can get back to where you are.** There is nothing more frustrating than realizing that a long series of changes have left you further from the goal than when you started, and having no way to get back to where you started. Disk space is usually plentiful. Use it to store old versions of your program.

Finally, if there are many unexplained errors, you might consider whether finding and fixing bugs one at a time is even the right approach. Maybe you would be better off thinking about whether there is some better way to organize your program or some simpler algorithm that will be easier to implement correctly.

## EXCEPTIONS

### Handling Exceptions

When an exception is raised that causes the program to terminate, we say that an **unhandled exception** has been raised.

An exception does not need to lead to program termination. Exceptions, when raised, can and should be **handled** by the program. Sometimes an exception is raised because there is a bug in the program (like accessing a variable that doesn't exist), but many times, an exception is something the programmer can and should anticipate.

If you know that a line of code might raise an exception when executed, you should handle the exception. In a well-written program, unhandled exceptions should be the exception.

It would have been better to have written something along the lines of

try:  
successFailureRatio = numSuccesses/float(numFailures) print 'The success/failure ratio is', successFailureRatio

except ZeroDivisionError:  
print 'No failures so the success/failure ratio is undefined.'

print 'Now here'

Upon entering the **try block**, the interpreter attempts to evaluate the expression numSuccesses/float(numFailures). If expression evaluation is successful, the program assigns the value of the expression to the variable successFailureRatio, executes the print statement at the end of the try block, and proceeds to the print statement following the try-except. If, however, a ZeroDivisionError exception is raised during the expression evaluation, control immediately jumps to the **except block** (skipping the assignment and the print statement in the try block), the print statement in the except block is executed, and then execution continues at the print statement following the try-except block.

### Exceptions as a Control Flow Mechanism

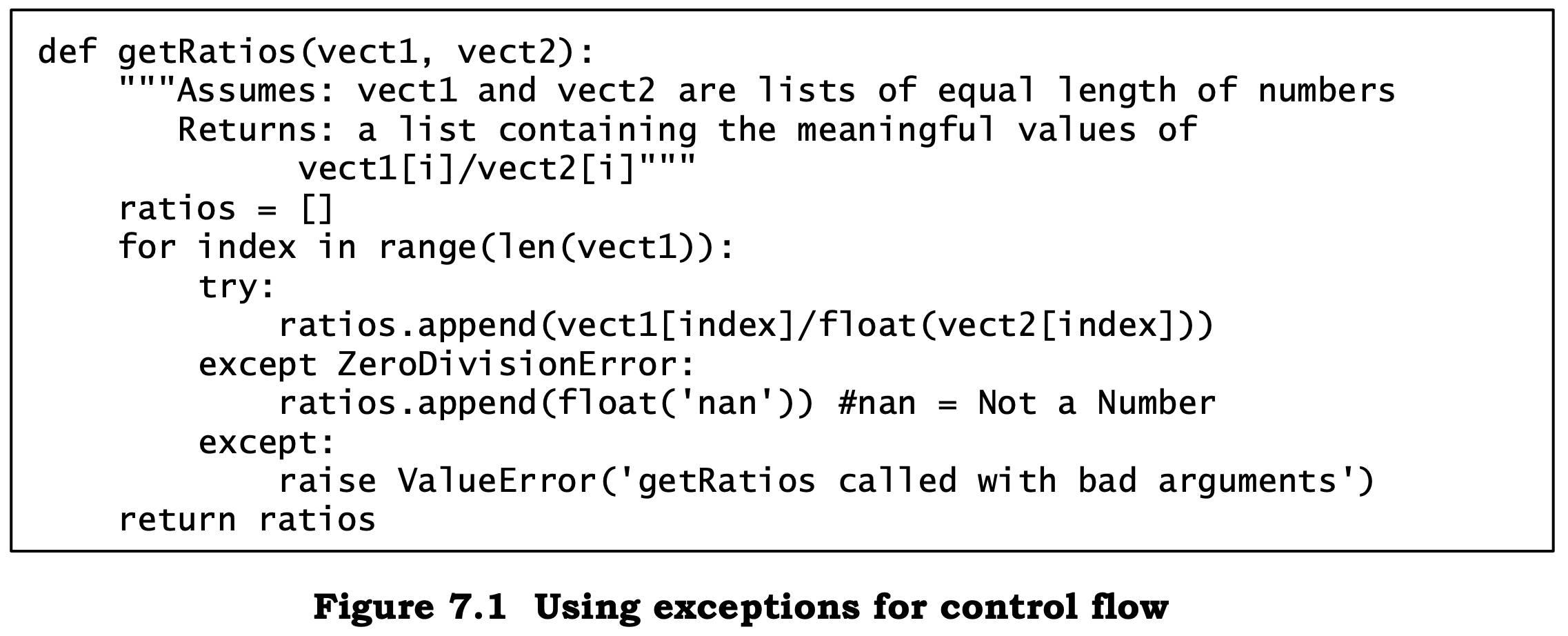
Don’t think of exceptions as purely for errors. They are a convenient flow-of- control mechanism that can be used to simplify programs.

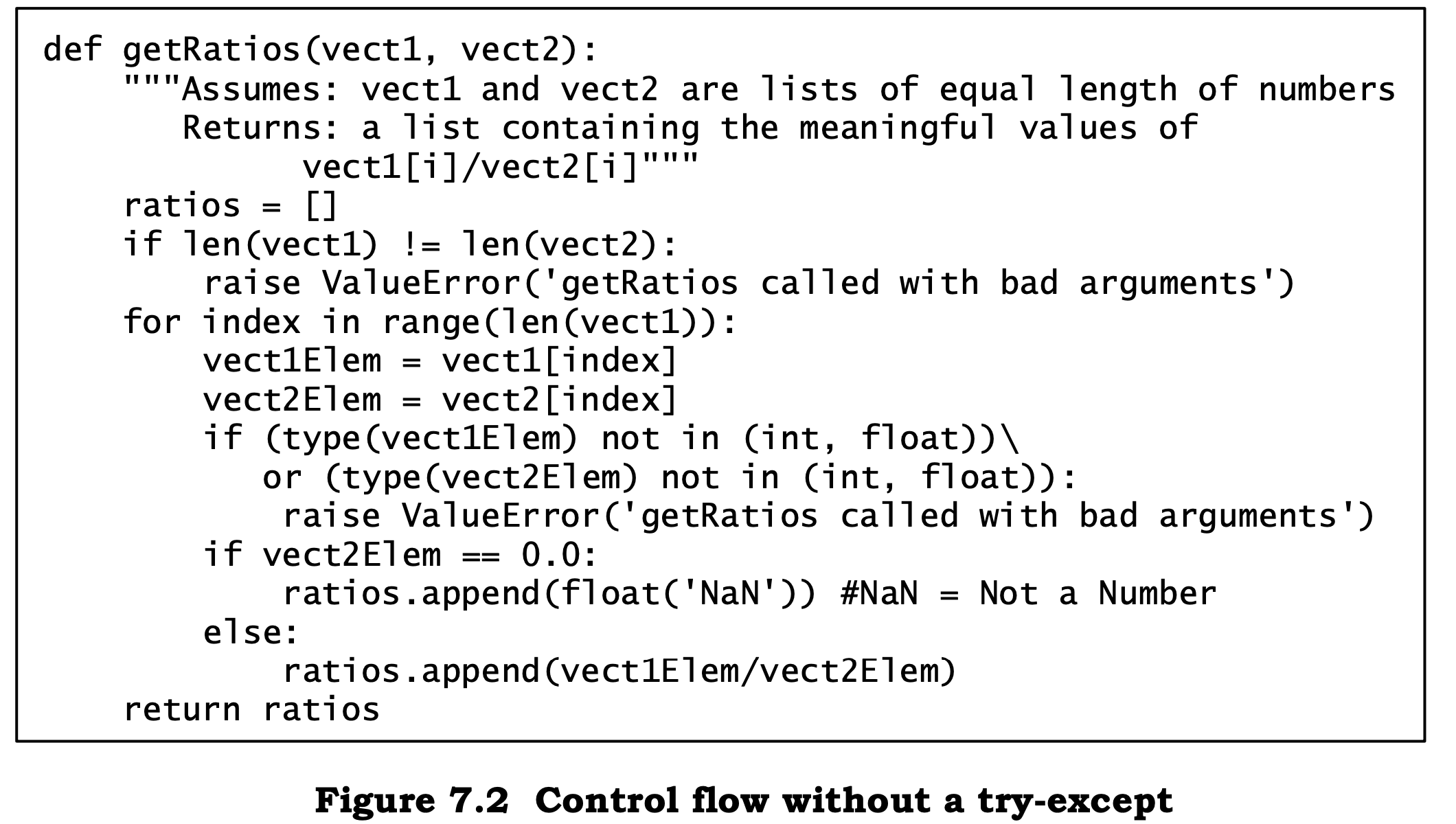
The Python **raise** statement forces a specified exception to occur. The form of a raise statement is

raise *exceptionName*(*arguments*)

The *exceptionName* is usually one of the built-in exceptions, e.g., ValueError.

However, programmers can define new exceptions by creating a subclass (see Chapter 8) of the built-in class Exception. Different types of exceptions can have different types of arguments, but most of the time the argument is a single string, which is used to describe the reason the exception is being raised.

书记figure7.1和7.2利用try-exception进行流程控制，可以减少代码量，增加易读性，并且减少了计算负载



The code in Figure 7.2 is longer and more difficult to read than the code in Figure 7.1. It is also less efficient. (The code in Figure 7.2 could be slightly shortened by eliminating the local variables vect1Elem and vect2Elem, but only at the cost of introducing yet more inefficiency by accessing each element repeatedly.)

## Assertion

The Python assert statement provides programmers with a simple way to confirm that the state of the computation is as expected. An assert statement can take one of two forms:

assert *Boolean expression*

or

assert *Boolean expression*, *argument*

Assertions are a useful defensive programming tool. They can be used to confirm that the arguments to a function are of appropriate types. They are also a useful debugging tool. The can be used, for example, to confirm that intermediate values have the expected values or that a function returns an acceptable value.

# Lecture8

Classes can be used in many different ways. In this book we emphasize using them in the context of **object-oriented programming**. The key to object- oriented programming is thinking about objects as collections of both data and the methods that operate on that data.

The notion of an abstract data type is quite simple. An **abstract data type** is a set of objects and the operations on those objects. These are bound together so that one can pass an object from one part of a program to another, and in doing so provide access not only to the data attributes of the object but also to operations that make it easy to manipulate that data.

The specifications of those operations define an **interface** between the abstract data type and the rest of the program. The interface defines the behavior of the operations—what they do, but not how they do it. The interface thus provides an **abstraction barrier** that isolates the rest of the program from the data structures, algorithms, and code involved in providing a realization of the type abstraction.

***Programming is about managing complexity in a way that facilitates change***. There are two powerful mechanisms available for accomplishing this: **decomposition** and **abstraction**. ***Decomposition creates structure in a program, and abstraction suppresses detail***. The key is to suppress the appropriate details. This is where data abstraction hits the mark. One can create domain- specific types that provide a convenient abstraction. Ideally, these types capture concepts that will be relevant over the lifetime of a program. If one starts the programming process by devising types that will be relevant months and even decades later, one has a great leg up in maintaining that software.

In Python, one implements data abstractions using **classes**

When a function definition occurs within a class definition, the defined function is called a **method** and is associated with the class. These methods are sometimes referred to as **method attributes** of the class.

Classes support two kinds of operations:

* **Instantiation** is used to create instances of the class. For example, the statement s = IntSet() creates a new object of type IntSet. This object is called an **instance** of IntSet.
* **Attribute references** use dot notation to access attributes associated with the class. For example, s.member refers to the method member associated with the instance s of type IntSet.

A class should not be confused with instances of that class, just as an object of type list should not be confused with the list type. Attributes can be associated either with a class itself or with instances of a class:

* Method attributes are defined in a class definition, for example IntSet.member is an attribute of the class IntSet. When the class is instantiated, e.g., by s = IntSet(), instance attributes, e.g., s.member, are created. Keep in mind that IntSet.member and s.member are different objects. Whiles.memberisinitiallyboundtothemembermethoddefined in the class IntSet, that binding can be changed during the course of a computation. For example, you could (but shouldn’t!) write s.member = IntSet.insert.
* When data attributes are associated with a class we call them **class variables**. When they are associated with an instance we call them **instance variables**. For example, vals is an instance variable because for each instance of class IntSet, vals is bound to a different list. So far, we haven’t seen a class variable. We will use one in Figure 8.3.

Data abstraction achieves representation-independence. Think of the implementation of an abstract type as having several components:

* Implementations of the methods of the type,
* Data structures that together encode values of the type, and
* Conventions about how the implementations of the methods are to use the data structures. A key convention is captured by the representation invariant.

The **representation invariant** defines which values of the data attributes correspond to valid representations of class instances. The representation invariant for IntSet is that vals contains no duplicates. The implementation of \_\_init\_\_ is responsible for establishing the invariant (which holds on the empty list), and the other methods are responsible for maintaining that invariant. That is why insert appends e only if it is not already in self.vals. <https://en.wikipedia.org/wiki/Class_invariant（wikipedia> for explaining class invariant）

***Data abstraction encourages program designers to focus on the centrality of data objects rather than functions***. Thinking about a program more as a collection of types than as a collection of functions leads to a profoundly different organizing principle. Among other things, it encourages one to think about programming as a process of combining relatively large chunks, since data abstractions typically encompass more functionality than do individual functions. This, in turn, leads us to think of the essence of programming as a process not of writing individual lines of code, but of composing abstractions.

***The availability of reusable abstractions not only reduces development time, but also usually leads to more reliable programs, because mature software is usually more reliable than new software***. For many years, the only program libraries in common use were statistical or scientific. Today, however, there is a great range of available program libraries (especially for Python), often based on a rich set of data abstractions, as we shall see later in this book.

# Lecture9

Many types have properties in common with other types. For example, types list and str each have len functions that mean the same thing. **Inheritance** provides a convenient mechanism for building groups of related abstractions. It allows programmers to create a type hierarchy in which each type inherits attributes from the types above it in the hierarchy.

In the jargon of object-oriented programming, MITPerson is a **subclass** of Person, and therefore **inherits** the attributes of its **superclass**. In addition to what it inherits, the subclass can:

* Add new attributes. For example, MITPerson has added the class variable nextIdNum, the instance variable idNum, and the method getIdNum.
* **Override** attributes of the superclass. For example, MITPerson has overridden \_\_init\_\_ and \_\_lt\_\_.

Overiride（重载）过程

When it attempts to evaluate the expression str(p1), the runtime system first checks to see if there is an \_\_str\_\_ method associated with class MITPerson. Since there is not, it next checks to see if there is an \_\_str\_\_ method associated with the superclass, Person, of MITPerson.

# Week10-12

## A SIMPLISTIC INTRODUCTION TO ALGORITHMIC COMPLEXITY

### Thinking About Computational Complexity

For simplicity, we will use a **random access machine** as our model of computation. In a random access machine, steps are executed sequentially, one at a time.43 A **step** is an operation that takes a fixed amount of time, such as binding a variable to an object, making a comparison, executing an arithmetic operation, or accessing an object in memory.

In general, there are three broad cases to think about:

* The best-case running time is the running time of the algorithm when the inputs are as favorable as possible. I.e., the **best-case** running time is the minimum running time over all the possible inputs of a given size. For linearSearch, the best-case running time is independent of the size of L.
* Similarly, the **worst-case** running time is the maximum running time over all the possible inputs of a given size. For linearSearch, the worst- case running time is linear in the size of the list.
* By analogy with the definitions of the best-case and worst-case running time, the **average-case** (also called **expected-case**) running time is the average running time over all possible inputs of a given size. Alternatively, if one has some *a priori* information about the distribution of input values (e.g., that 90% of the time x is in L), one can take that into account.

People usually focus on the worst case. All engineers share a common article of faith, Murphy’s Law: If something can go wrong, it will go wrong. The worst-case provides an **upper bound** on the running time.

**Asymptotic Notation**

We use something called **asymptotic notation** to provide a formal way to talk about the relationship between the running time of an algorithm and the size of its inputs. The underlying motivation is that almost any algorithm is sufficiently efficient when run on small inputs. What we typically need to worry about is the efficiency of the algorithm when run on very large inputs. As a proxy for “very large,” asymptotic notation describes the complexity of an algorithm as the size of its inputs approaches infinity.

This kind of analysis leads us to use the following rules of thumb in describing the asymptotic complexity of an algorithm:

* If the running time is the sum of multiple terms, keep the one with the largest growth rate, and drop the others.
* If the remaining term is a product, drop any constants.

The most commonly used asymptotic notation is called “**Big O**” notation.44 Big O notation is used to give an **upper bound** on the asymptotic growth (often called the **order of growth**) of a function. For example, the formula f(x) ∈O(x2 ) means that the function f grows no faster than the quadratic polynomial x , in an asymptotic sense.

We, like many computer scientists, will often ***abuse*** Big O notation by making statements like, “the complexity of f(x) is O(x2).” By this we mean that in the worst case f will take O(x2 ) steps to run. The difference between a function being **“in O(x2)”** and **“being O(x2)”** is subtle but important. Saying that f(x) ∈O (x2) does not preclude the worst-case running time of f from being considerably less that O(x2). When we say that f(x) is O(x2), we are implying that x2 is both an upper and a **lower bound** on the asymptotic worst-case running time. This is called a **tight bound**.45

### Some Important Complexity Classes

Some of the most common instances of Big O are listed below. In each case, *n* is a measure of the size of the inputs to the function.

* O(1) denotes **constant** running time.
* O(log n) denotes **logarithmic** running time.
* O(n) denotes **linear** running time.
* O(n log n) denotes **log-linear** running time**.**
* O(nk) denotes **polynomial** running time. Notice that *k* is a constant.
* O(cn ) denotes **exponential** running time. Here a constant is being raised to a power based on the size of the input.

The impact of space complexity is harder to appreciate than the impact of time complexity. Whether a program takes one minute or two minutes to complete is quite visible to its user, but whether it uses one megabyte or two megabytes of memory is largely invisible to users. This is why people typically give more attention to time complexity than to space complexity. The exception occurs when a program needs more space than is available in the main memory of the machine on which it is run.

**keep in mind that in most practical situations, O(n log(n)) is fast enough to be useful.**

## SOME SIMPLE ALGORITHMS AND DATA STRUCTURES

What we do instead is learn to reduce the most complex aspects of the problems with which we are faced to previously solved problems. More specifically, we

* Develop an understanding of the inherent complexity of the problem with which we are faced,
* Think about how to **break that problem up into subproblems**, and
* **Relate** those subproblems to other problems for which efficient algorithms already **exist**.

### Search Algorithms

A **search algorithm** is a method for finding an item or group of items with specific properties within a collection of items. We refer to the collection of items as a **search space**. The search space might be something concrete, such as a set of electronic medical records, or something abstract, such as the set of all integers. A large number of problems that occur in practice can be formulated as search problems.

This example（列表索引） illustrates one of the most important implementation techniques used in computing: **indirection**.49 Generally speaking, indirection involves accessing something by first accessing something else that contains a reference to the thing initially sought. This is what happens each time we use a variable to refer to the object to which that variable is bound. When we use a variable to access a list and then a reference stored in that list to access another object, we are going through two levels of indirection

(**Binary Search**)Here we rely on the **assumption** that the list is ordered.

The idea is simple:

1. Pick an index, i, that divides the list L roughly in half.
2. Ask if L[i] == e.
3. If not, ask whether L[i] is larger or smaller than e.
4. Depending upon the answer, search either the left or right half of L for e.

### Sorting Algorithms

Does this mean that binary search is an intellectual curiosity of no practical import? Happily, no. Suppose that one expects to search the same list many times. It might well make sense to pay the overhead of sorting the list once, and then **amortize** the cost of the sort over many searches. If we expect to search the list k times, the relevant question becomes, is (sortComplexity(L) + k\*log(len(L))) less than k\*len(L)? As k becomes large, the time required to sort the list becomes increasingly irrelevant.

(**Selection Sort**)We use induction to reason about loop invariants.

* Base case: At the start of the first iteration, the prefix is empty, i.e., the suffix is the entire list. The invariant is (trivially) true.
* Induction step: At each step of the algorithm, we move one element from the suffix to the prefix. We do this by appending a minimum element of the suffix to the end of the prefix. Because the invariant held before we moved the element, we know that after we append the element the prefix is still sorted. We also know that since we removed the smallest element in the suffix, no element in the prefix is larger than the smallest element in the suffix.
* When the loop is exited, the prefix includes the entire list, and the suffix is empty. Therefore, the entire list is now sorted in ascending order.

Fortunately, we can do a lot better than quadratic time using a **divide-and- conquer algorithm**. The basic idea is to combine solutions of simpler instances of the original problem. In general, a divide-and-conquer algorithm is characterized by

1. A threshold input size, below which the problem is not subdivided,
2. The size and number of sub-instances into which an instance is split, and
3. The algorithm used to combine sub-solutions.

The threshold is sometimes called the **recursive base**. For item 2 it is usual to consider the ratio of initial problem size to sub-instance size. In most of the examples we’ve seen so far, the ratio was 2.

**Merge sort** is a prototypical divide-and-conquer algorithm

1. If the list is of length 0 or 1, it is already sorted.
2. If the list has more than one element, split the list into two lists, and use merge sort to sort each of them.
3. Merge the results.

This improvement in time complexity comes with a price. Selection sort is an example of an **in-place** sorting algorithm. Because it works by swapping the place of elements within the list, it uses only a constant amount of extra storage (one element in our implementation). In contrast, the merge sort algorithm involves making copies of the list. This means that its space complexity is O(len(L)). This can be an issue for large lists.53

The sorting algorithm used in most Python implementations is called **timsort**.54 The key idea is to take advantage of the fact that in a lot of data sets the data is already partially sorted. Timsort’s worst-case performance is the same as merge sort’s, but on average it performs considerably better.

Both the list.sort method and the sorted function provide **stablesorts**. This means that if two elements are equal with respect to the comparison used in the sort, their relative ordering in the original list (or other iterable object) is preserved in the final list.

### Hash Tables

When we introduced the type dict in Chapter 5, we said that dictionaries use a technique called hashing to do the **lookup in time that is nearly independent of the size of the dictionary**. The basic idea behind a **hash table** is simple.

Which gets us to the subject of hash functions. A **hash function** maps a large space of inputs (e.g., all natural numbers) to a smaller space of outputs (e.g., the natural numbers between 0 and 5000). Hash functions can be used to convert a large space of keys to a smaller space of integer indices.

Since the space of possible outputs is smaller than the space of possible inputs, a hash function is a **many-to-one mapping**, i.e., multiple different inputs may be mapped to the same output. When two inputs are mapped to the same output, it is called a **collision**—a topic which we will to return shortly. A good hash function produces a **uniform distribution**, i.e., every output in the range is equally probable, which minimizes the probability of collisions.



Designing good hash functions is surprisingly challenging. The problem is that one wants the outputs to be uniformly distributed given the expected distribution of inputs. The basic idea is to represent an instance of class intDict by a list of **hash buckets**, where each bucket is a list of key/value pairs. By making each bucket a list, we handle collisions by storing all of the values that hash to the same bucket in the list.

The hash table works as follows: The instance variable buckets is initialized to a list of numBuckets empty lists. To store or look up an entry with key dictKey, we use the hash function % to convert dictKey into an integer, and use that integer to index into buckets to find the hash bucket associated with dictKey. We then search that bucket (which is a list) linearly to see if there is an entry with the key dictKey. If we are doing a lookup and there is an entry with the key, we simply return the value stored with that key. If there is no entry with that key, we return None. If a value is to be stored, then we either replace the value in the existing entry, if one was found, or append a new entry to the bucket if none was found.