Intoduction to Computer Science

Mark Eramian

Course readings for CMPT 141

PRODUCED BY THE AUTHORS FOR STUDENTS IN CMPT 141.

CS.USASK.CA

First Edition, September 2016

Acknowledgements

The author would like to thank Brittany Chan, Michael Horsch, and Jeff Long for their invaluable contributions to this work including advice on content, organization, and help with proofreading.

We also thank the following individuals for discovering errors or otherwise contributing to the text: Chad Mckellar.

Contents

Part I

Programming in Python

1 — Algorithms and Computer Programs

Learning Objectives

After studying this chapter, a student should be able to:

- describe what an algorithm is;
- provide examples of algorithms;
- identify actions, blocks, conditionals, and repetition in algorithms;
- identify and distinguish between inputs and outputs of an algorithm;
- distinguish between algorithms written for humans and algorithms written for computers; and
- describe the process by which an algorithm becomes a computer program
- define the concepts of abstraction and refinement; and
- distinguish between problems and algorithms.

1.1 Algorithms

An *algorithm* is a list of actions that describe how to perform a task or solve a problem. A recipe for making bread is an algorithm. The recipe describes what actions you must take, and the order in which you must take them, if you want to end up with something that looks and tastes like bread. If you deviate from the algorithm, there's a good chance you end up with something quite un-bread-like. Other examples of algorithms are:

- instructions for assembling a bookshelf;
- steps to operate a coffee maker; and
- a list of things to do in case of a fire.

Here is a concrete example showing the specific steps in an algorithm to make ramen noodles:

```
Algorithm MakeRamen:

boil water
add noodles to water
wait 6-8 minutes
drain the noodles
stir in contents of flavour packet
place cooked noodles in bowl
```

This algorithm consists of six actions to solve the problem of making ramen noodles. The actions are taken in the order given, and the end result, or *output* of the algorithm is a prepared bowl of steaming hot noodles, ready to eat. The important thing to remember about algorithms is that the given actions must be taken in the given order, otherwise you are not following the algorithm and likely will not get the desired output.

1.1.1 Blocks, Repetition, and Conditionals

Algorithms can contain things other than just actions to take, such as setting conditions under which actions should be taken. Let's consider the following algorithm for playing the game Jenga. Jenga is a two-player game where you start with a tower of blocks, and players take turns making the tower taller by removing an existing block from the tower and placing it on the top of the tower without knocking it over. The player who knocks down the tower loses. For more on how to play Jenga, click here for an insightful video explanation. Here's an algorithm for playing Jenga:

```
Algorithm PlayJenga:

stack the blocks to form a tower
choose a current player
as long as the tower of blocks is still standing:
     current player takes a block from middle
     current player puts the block on top
     if the tower is still standing:
          make the other player the current player
the current player loses
```

In this algorithm we have a sequence of actions that needs to be repeated as long the tower is still standing. Notice how the actions to be repeated are indented. Such sequence of actions is called a *block*. So in this algorithm we have a block of three actions to be repeated. Notice that the third action in the block, the one that begins with `if the tower...`, is not actually an action, it is a condition that says when some other action, `make the other player the current player`, should be performed. Notice how this action is indented again, relative to the actions in the block of repeated actions. This means that the action `make the other player the current player` is, itself, a single-action block where the action is only taken if the condition `the tower of blocks is still standing` is true. A block inside of another block is called a *nested block*. Blocks can be *nested* to any number of levels. Finally, we can see that the action `the current player loses` is not one of the actions to be repeated because it is not indented.

Indentation is almost always used to denote blocks.[1] Thus, when reading algorithms, be aware that the indentation is not arbitrary or accidental, but rather conveys important meaning.

1.1.2 Variables

Algorithms often deal with numbers and are easier to write if we are allowed to give names to numbers. A *variable* is a name given to a particular numeric value.[2] We can write algorithms with variables that refer to values, change the values they refer to whenever we want, and use them to compute other values. For example, here's an algorithm for computing the average of a set of numbers that uses a variable to keep track of a running sum:

```
Algorithm Average:
Input: a set of numbers

let total = 0;
for each number x in the set of numbers:
     let total = total + x
let average = total / size of the set of numbers
```

In this algorithm the first action, `let total = 0`, means that we associate the variable name `total` with the value 0. Then we have a single-action block which is repeated for each number `x` in the input set. That action adds `x` to the existing value referred to by `total` and associates `total` with the new resulting value. Then the *output* of the algorithm, the average, is associated with another variable called `average`.

1.1.3 Input and Output

In the algorithm in the previous section, note how we explicitly specified the input to the algorithm in the second line. In general, algorithms are allowed to have any number of *inputs* (including none). Inputs are data that **come from outside the algorithm** which the algorithm uses to complete its task. An algorithm with inputs is often much more useful than an algorithm that has none. For example, here is an algorithm that computes the average of the set of even numbers between 1 and 10:

```
Algorithm AverageEven1to10:

let total = 0
for each number x in the set {2, 4, 6, 8, 10}:
     let total = total + x
average = total / 5
```

This is exactly the same algorithm as the `Average` algorithm in the previous section except that it does not have any inputs and it only works for one specific set of numbers which is written right into the algorithm. By allowing the set of numbers to be an *input* to the algorithm, we get an algorithm that is reusable in vastly more situations and can compute the average of any set of numbers instead of just one specific set.

[1] There are other ways to denote blocks in algorithms besides indentation such as enclosing a block in a pair of curly braces { ... }. But even then, indentation is almost always used as well to make algorithms more readable to humans.

[2] Later we'll learn that variables can refer to values that are things other than numbers.

Algorithms have one or more *outputs*. An *output* is data that is the result of the task that the algorithm was intended to carry out. In the `Average` algorithm from the previous section we didn't explicitly specify that the variable `average` is the algorithm's output, but we could, for example:

```
Algorithm Average:
Input: a set of numbers
Output: a variable 'average' which refers to the average
        of the numbers in the input set

let total = 0;
for each number x in the set of numbers:
     let total = total + x
let average = total / size of the set of numbers
```

Now the output of the algorithm is explicitly specified. If an algorithm has more than one input or output, then the additional inputs and outputs can be described in the same fashion.

Notice that for every algorithm that we have shown you so far, we have given it a name, and sometimes specified its input and output, and these things have appeared before the algorithm's first action. The description of an algorithm's name, inputs, and outputs, are collectively called the algorithm's *header*. The items in the header are not actions in the algorithm to be carried out, but rather describe the algorithm and its intended usage.

1.1.4 Methods of Writing Algorithms

Algorithms can be written in different forms. So far we've seen a few algorithms that are written in words. Algorithms written in words are called *pseudocode*. Pseudocode can look like the "code" we would write in a programming language, but is much more flexible because its syntax and form is not rigidly specified like that of a programming language.

The LEGO instructions and the airline safety card pictured in Figure 1.1 are examples of algorithms written using pictures. They indicate, in a step-by-step manner, what to do to complete the tasks of building the LEGO model, and escaping the aircraft in an emergency.

Words and pictures are normally how we write algorithms that are to be understood and/or carried out by humans. When we write algorithms for computers we have to use a language that a computer can understand. Computer programs are written in a *programming language*. A programming language provides instructions to the computer in a way that it can both understand them and carry them out unambiguously. For this reason, computer programs are much more strict in the syntax and style we can use to write algorithms.

If a computer program doesn't do what its programmer wants it to, it's not the computer's fault! The computer can only do exactly what the program tells it to do whether or not is what the programmer intended. This is why it is vital that a programmer understand the algorithm that he/she is trying to write. If a programmer doesn't understand the algorithm, the chance that they'll be able to tell the computer how to perform the algorithm correctly is slim to none. This is why it is advantageous for programmers to write algorithms using pseudocode first. It helps them to ensure they understand what they are about to program without having to worry about the details of the programming language syntax. Once understanding is reached via pseudocode, a programmer is much more easily able to get the details right when writing the algorithm in the precise syntax required by the programming language. Moreover, they are ready to implement the algorithm in **any**

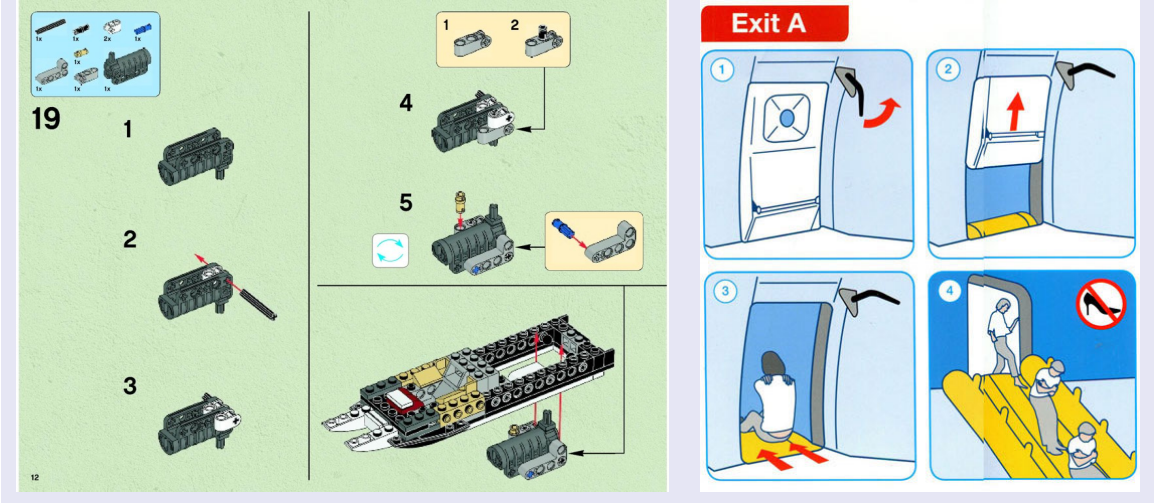


Figure 1.1: Examples of algorithms written using pictures. Left: LEGO instructions; right: aircraft safety procedures card.

programming language that they know!

In this course we will be using the Python programming language. Let's look at what the `Average` algorithm looks like when it is translated from pseudocode to Python. Don't worry if you don't understand **why** the algorithm is written the way it is in Python. We'll get to that later.

```
Algorithm Average:
Input: a set of numbers

let total = 0;
for each number x in the set of numbers:
     let total = total + x
let average = total / size of the set of numbers
```

Listing 1.1: The `Average` algorithm in pseudocode.

```
def Average(S):
     total = 0
     for x in S:
          total = total + x
     average = total / len(S)
     return average
```

Listing 1.2: The `Average` algorithm in Python.

You should be able to appreciate that the two versions of the algorithm are doing the same thing. The header in the pseudocode algorithm has been translated to the line starting with `def` in Python, so if you're thinking that this is some Python syntax for saying that we want to define an algorithm, give it a name, and say what its inputs are, then you're right. The `S` in the round brackets indicates that S is the input to the algorithm. You should also be able to appreciate that `total` and `average`

are variables in the Python version, just as they are in the pseudocode version. The repetition `for x in S` looks much the same as in the pseudocode and indicates that we do the same thing to each element of the set `S`. The `len(S)` syntax tells us how many numbers are in the set `S`. The blue words in the Python code have specific, well-defined meanings in Python. The last line contains the command `return` which tells Python that the variable `average` is the output of the algorithm.

The Python code can be understood and carried out by a computer, but the pseudocode algorithm cannot, even though it doesn't look that different. Depending on how a pseudocode algorithm is written, there may or may not be an easy, line-by-line translation of the pseudocode algorithm into Python (though in this case, it's pretty close).

1.2 Abstraction and Refinement

Algorithms can be written at different levels of detail or, as computer scientists like to say, levels of *abstraction*. Abstraction is the hiding of details that are not currently important. To illustrate this concept, let's look again at our `MakeRamen` algorithm.

```
Algorithm MakeRamen:

boil water
add noodles to water
wait 6-8 minutes
drain the noodles
stir in contents of flavour packet
place cooked noodles in bowl
```

The first instruction in this pseudocode is `boil water`. This is a great example of abstraction, because the action `boil water` glosses over all of the details of **how** to boil water. For humans, these details are pretty unimportant, because adult humans all know how to boil water. But imagine that this algorithm has to be carried out by a humanoid cooking robot. The robot does not intuitively know how to boil water. The action `boil water` is too *abstract* for it. It needs more details.

The process of describing more detail about how an instruction should be carried out is called *refinement*. For example, we might refine the `boil water` action by replacing it with a sequence of actions (shown in red text) that describe how to boil water in more detail:

```
Algorithm MakeRamen:

place pot under faucet
add water to pot
place pot on stove
turn on burner
wait until water boils
add noodles to water
wait 6-8 minutes
drain the noodles
stir in contents of flavour packet
place cooked noodles in bowl
```

Listing 1.3: MakeRamen algorithm with **refinement** of the `boil water` action.

Each of these new actions describes an action that partially carries out the original `boil water` action. But even these actions may not be detailed enough for our robot to carry out the task. At some point, the robot needs to know exactly where, and for how long to position its legs and arms to carry out these actions. This would require that we further refine the actions `place pot under faucet`, `add water to pot`, etc. to the level of detail where we tell the robot exactly where and how to move its limbs by replacing each of these actions with sequences of even more detailed actions. This is called *stepwise refinement*. We repeatedly replace actions that are too abstract with a sequence of less abstract, more detailed actions until we reach a level of detail that can be directly carried out. Each level of refinement results in actions that are at a lower level of abstraction and are closer to the individual actions that the robot (or computer) can carry out natively.

The ability to think at different levels of abstraction and mentally move between them is critical to success as a programmer and a computer scientist. We abstract away details when they are not important, and refine abstractions later when we are ready for the detail. Defining an algorithm and giving it a name is, itself, a form of abstraction. Naming an algorithm and describing its inputs and outputs allows someone to use the algorithm without knowing **how** the algorithm works. In other words, an algorithm, once written, hides the details of how the algorithm is performed, allowing it to be used to produce results without knowledge of the algorithm's details. For example, having defined the algorithm `Average`, we no longer have to remember that to compute an average, you add all the numbers up and divide by how many numbers there were. All we have to do is say "perform the algorithm `Average` on the set of numbers 1, 2, 3, 4, and 5", and we'll get the correct answer of 3.

We take abstraction for granted all the time. So many of the things we do on computers that look really simple are actually abstractions of breathtakingly complex algorithms and hardware details. These are things like Google Search, fingerprint ID on your phone, and face recognition in your digital photography software. There is a tendency for people who are used to such technology to underestimate its complexity. Keep this in mind the next time you think to yourself that it would be really "easy" to add some desired feature to your phone or a piece of software.

To summarize, abstraction allows us to think about performing higher-level, more complex actions without worrying about **how** they are performed. Abstraction doesn't mean that the lower-level details of how an abstracted algorithm is carried out don't exist or never have to be written at some point. It is just a mechanism that allows us to ignore such details when it is convenient or until they are needed.

1.3 Problems vs. Algorithms

We'll conclude this chapter by making a distinction between problems and algorithms. A *problem* is a task to be carried out. An *algorithm* is a specific set of steps for **how** to carry out a task. A problem may have more than one algorithm for solving it. A given algorithm, however, solves only one problem.

Let's consider the problem of picking up a pile of playing cards that you were just dealt, and putting them in rank-order. Some people pick up their cards one at a time and place each card into their hand at the correct position as they do so. Other people pick up all of the cards at once, find the smallest one and move that card to the left-most position, then find the next smallest and put it next to the smallest card, and so on. These are two different algorithms for solving the problem of putting a hand of cards in order. Both achieve the same result, the but algorithms themselves are fundamentally different processes.

2 — Data, Expressions, Variables, and I/O

Learning Objectives

After studying this chapter, a student should be able to:

- distinguish between atomic data and compound data;
- describe what a data type is;
- describe what a literal value is in Python;
- give examples of literal values corresponding to integer, floating-point, and string data;
- list the basic arithmetic operators in Python;
- describe what an expression is in Python;
- compose valid arithmetic expressions in Python using operators and literals;
- describe what a variable is;
- explain the naming rules for variables;
- compose valid expressions in Python using variables;
- use the `print` syntax to display literal values and the values of variables on the console; and
- use the `input` syntax to read values from the console and store the value read in a variable.

We're going to cover a lot of material fairly quickly in this chapter. But remember that CMPT 141 is is intended for students who have done some programming before. If you've programmed before, even if you didn't program in Python, then most of the concepts in this chapter should be at least a little bit familiar.

2.1 Data

Data is information. Computer programs need data to do anything useful. All input and output is data. Data can take many forms (numbers, text, pictures, etc.), but ultimately, at a low enough level of abstraction, all data is numbers because that is what computers know how to store. It is abstraction

that makes it appear that we can store things more interesting than numbers, such as images, video, text, web pages, etc. These things are all just large collections of numbers interpreted in different ways — the different interpretations are abstractions! At an even lower level of abstraction, all data is just sequences of 0's and 1's, because computer hardware stores data as binary numbers using different electric voltages to represent the binary digits 0 and 1. Fortunately, computer programmers don't have to work at such a low level of abstraction. In the rest of this section we'll look at the kinds of data we, as programmers, can use.

2.1.1 Atomic Data

Atomic Data is the smallest unit of data that a computer program can define. Atomic data is usually a single number in decimal (base 10). The word "atomic" derives from the word "atom". At one point in the history of chemistry, atoms were believed to be the smallest indivisible pieces of matter in the universe. In computer science, the word "atomic" is often used to refer to something that is indivisible or cannot be made smaller.

2.1.2 Compound Data

Compound data is data that can be subdivided into smaller pieces of data which are organized in a particular way. An example of compound data is a list. A list consists of several pieces of data which have a specific ordering. The data items that comprise a piece of compound data may themselves be either compound or atomic. For example, we could imagine a list of numbers. The list itself is compound data, while each piece of data in the list is atomic data. We could also imagine a list of lists of numbers. In such a case, the list of lists is compound data, and each piece of data in the list is itself an example of compound data whose individual pieces are, in turn, atomic data.

We will be dealing almost exclusively with atomic data until we get to Chapter 6, where we will revisit this topic. For now, the only type of compound data we will be using are *strings* (which we will define momentarily in 2.1.3).

2.1.3 Data Types

In a computer program, every piece of data, compound or atomic, has a *data type*. For one last moment, let us recall that every piece of data in a computer is, at a very low level of abstraction that we don't usually worry about, made up of binary 0's and 1's (these are referred to as *bits*). The *data type* of a piece of data tells the computer how to interpret those bits. For example, whether to interpret them as a number or a character; an integer or a fraction. In Python there are several built-in atomic and compound data types.

Atomic Data Types

In this section we describe the most commonly used atomic data types in Python.

Integer: Integer data are positive or negative whole numbers, or zero. In Python, there is no limit to the size of an integer number.

Floating-point: Floating-point data are real numbers, that is, numbers that are not necessarily whole numbers, such as the number 42.5. Floating-point data in Python (and any other language) have a limited range, and limited precision. This means that numbers with infinite representations, such as 1/3=0.33333..., $\sqrt{2}$, or π, cannot be represented exactly. In Python, floating-point numbers can range between 10^{-308} to 10^{308} (positive or negative) with at most 16 to 17 digits of precision.

Boolean: Boolean data can only be one of two values: `True` or `False`. Note that capitalization matters – `true` and `false` are not valid boolean values in Python, but `True` and `False` are.

Compound Data Types

In this section we briefly describe some of the standard compound data types that are built into Python. However, we will save the the details of most of these until later chapters.

String: Strings are sequences of characters (e.g. letters of the alphabet) and are usually used to store text. We'll say more about strings and characters later in this chapter.

List: Lists are a sequence of data items. Each item in a list can be of any data type. We'll discuss lists in more detail in Chapter 7.

Dictionary: A dictionary consists of key-value pairs in no particular order. You can look up values by their key. We'll discuss dictionaries in more detail in Chapter 8.

2.2 Expressions

Expressions in a programming language are combinations of data and special symbols called *operators*, which have specific meaning in the programming language. Operators perform computations on one or more pieces of data to produce a new piece of data. When all of the computations associated with operators in an expression have been carried out, the result is a new piece of data whose value is the result of the expression. In Python, every valid expression describes some computation that results in a value which we call the *value of the expression*.

2.2.1 Literals

A *literal* is a number or string written right into the program, that is, **literally** typed right into the program's code, such as `42`. Literals are one of the fundamental components of expressions. If we are to write more complex expressions, we first need to learn about literals.

Literals are one of the simplest forms of expressions. The value of an expression containing a single literal is the value of the literal itself. We can see this right away by running Python in interactive mode. If we start up Python from the terminal, and type in the number `42`, Python responds by telling us that the value of the expression `42` is 42.

```
iroh:CMPT141 mark$ python
Python 3.5.1 |Anaconda 2.4.1 (x86_64)| (default, Dec  7 2015, 11:24:55)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 42
42
>>>
```

Literals in Python have a data type that is inferred by the manner in which the literal is written.

Integer literals: Any number written without a decimal point is an *integer literal*. Thus, `42`, `-17`, and `65535` are integer literals.

Floating-point literals: Any number written **with** a decimal point is a *floating-point literal*. Examples are: `42.0`, `-9.8`, and `3.14159`. Note with care that even the literal `42.` (decimal point included) is a floating-point literal because it contains a decimal point. An empty sequence of digits after the decimal point is different from no decimal point at all! We can see the difference in Python; note how when we enter 42., Python responds with the value 42.0, a floating-point value:

```
>>> 42.
42.0
>>>
```

Floating-point literals can also be written in scientific notation. For example, the speed of light is 3×10^8 m/s, a quantity which can be written as the literal `3e8`. See Python's response when we enter the expression `3e8`:

```
>>> 3e8
300000000.0
>>>
```

Literals written in scientific notation are always floating point, never integers. Here are some more examples of floating-point literals:

- `6.022e23` (6.022×10^{23})
- `9.11e-31` (9.11×10^{-31})
- `1e+3` (1000)

String literals: Recall that in the previous section we said that strings are sequences of characters. A string literal is specified by enclosing a sequence of characters with a pair of single or double quotes. `"Hello world."` and `'The night is dark and full of terrors.'` are both examples of string literals. Strings can contain spaces because spaces are characters too. Any symbol that appears on the keyboard is a character.[1] Note that there is a difference between the literals `'7'` and 7; the former is a string literal and does not actually have the numeric value of 7, while the latter is an integer literal, which does. Similarly the literals `"3.14159"` and `3.14159` are different. The former is a string literal, which does not actually have the numeric value 3.14159, and the latter is a floating-point literal, which does. However, `'Bazinga!'` and `"Bazinga!"` are exactly the same, as shown if we enter them in Python:

```
>>> "Bazinga!"
'Bazinga!'
>>> 'Bazinga!'
'Bazinga!'
>>>
```

So why have two ways of writing string literals? It is so that we can conveniently include single or double quotes as part of a string. The string `'The card says "Moops".'` is enclosed in single quotes and contains two double quotes as part of the string. The single quotes are not part of the string, but the double quotes are!

```
>>> 'The card says "Moops".'
'The card says "Moops".'
>>>
```

But look what happens when we try to write the same string literal in Python instead with double quotes enclosing the whole string:

[1] Other, stranger things can be characters too, but we'll avoid that discussion for now.

```
>>> "The card says "Moops"."
  File "<stdin>", line 1
    "The card says "Moops"."
                          ^
SyntaxError: invalid syntax
>>>
```

Oh my, Python sure didn't like that. The reason this results in an error is because Python interprets the characters between the first two double quotes (the first one and the one right before the `M`) as a string literal. Then the word `Moops` makes no sense to Python because Python thinks it's not part of a string literal, so Python tries to interpret it as part of the Python language, which it isn't, so Python doesn't know what to do and gives up. Thus, you can write single quotes inside string literals enclosed in double quotes, and double quotes inside string literals enclosed in single quotes.

So should you use single or double quotes for strings? Well, there's no right or wrong answer to this question. Unless you need single or double quotes within a string literal, it doesn't matter. Normally, one chooses to use either single or double quotes as one's "default" style, and only uses the other when necessary.

String Literals in Other Languages

In most other programming languages, there is only one way to write a string literal, and single and double quotation marks have very different meanings. For example, in C, C++, and Java, strings literals **must** be enclosed in double-quotes.

2.3 Variables

If we only had literal values, we couldn't write very interesting or useful programs because the program would use the same data, and produce exactly the same results every time it is run. Variables are a way of giving names to data. Giving a name to data allows a program to operate on different data values each time a problem is run. We can then ask Python to do something to the data with a certain name. If we only had literals, we could only ask Python to do something with a specific literal data value.

2.3.1 Variable Names

Variable names (also called *identifiers*) in Python have to follow the following rules:

- may contain letters or digits, but cannot start with a digit;
- may contain underscore (_) characters and may start with an underscore; and
- may not contain spaces or other special characters.

Thus, `KyloRen`, `IG88`, and `luke_Skywalker` are valid variable names, but these are not: `Luke+Leia_4_Evar` (contains special character +), `2ManyStormTroopers` (starts with a digit), and `Han Shot First` (contains spaces).[2]

[2] Does not change the fact that Han **did** shoot first.

2.3.2 Variable Assignment

The equal sign (=) is used in Python to *assign* a variable name to a value. Unlike many other programming languages, variables in Python do **not** have to be declared before they are used. You just use them. Here are some examples of variable assignment:

```
x = 5      # assign the name x to the integer 5.
y = 42.0   # assign the name y to the floating-point number 42.0

# assign the name error_message to the string: "That didn't work!"
error_message = "That didn't work!"
```

It may seem strange to think of the name being assigned to the value. Indeed, in most other programming languages we tend to think of assigning values to variables. But in Python, it is safer, and more reflective of how Python actually works, to think of assigning variable names to values. In Python, an assignment statement causes the variable to refer to its value. If you re-assign a variable to a new value, using another assignment statement, Python changes the variable's reference, not its value. In Python, more than one variable can refer to a given value.

The data to which a variable refers always has a type, but you cannot tell from the variable name what type of data it refers to. You can even change the type that a variable refers to:

```
x = 10;      # x refers to the integer 10
x = 10.0;    # now x refers to the floating-point value 10.0
```

There are ways to determine the type of data that a variable refers to, but we'll leave that for a later discussion. For now, just be aware that there is no way to guarantee that a variable always refers to data of a specific type. You can write a program so that a variable is always **supposed** to be of a certain type, but the type might change as a result of a bug, and there is no way to force Python to notify you of this. This is a contrast to many other programming languages (e.g. C++, Java) where variables must be defined to have a specific data type, and attempting to assign a value of a different type to that variable will result in an error.

2.3.3 Variables as Expressions

Just like a single literal, a single variable is an expression. The value of such an expression is the data value that the variable refers to.

```
>>> x = 10;     # x refers to the integer 10
>>> x           # an expression
10
>>>
```

In the above example, Python tells us that the value of the expression x is 10.

2.3.4 Operators

Operators can be used to write expressions that compute new values from existing pieces of data. We say that the operator *operates* on these pieces of data. For example, the expression `2 + 3` has the value 5.

```
>>> 2 + 3
5
>>>
```

In the above example the addition operator + computes the sum of 2 and 3. Python responds with the value 5 because that is the value of the expression `2 + 3`. The data items that an operator operates on are call *operands*. Operands can be any expression. Most of the operators we will see are binary operators because they require two operands.[3] Operands need not be literals, they can be variables too:

```
>>> x = 2
>>> y = 3
>>> x + y
5
>>>
```

Since `x` refers to the integer 2, and `y` refers to the integer 3, the value of the expression `x + y` is 5.

This is also a good time to note that a variable name cannot be used in an expression if it has not been assigned to a value. For example:

```
>>> x = 2
>>> x + z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

In the above example, when we try to add together the value referred to by `x` and the value referred to by `z` (which refers to no value because none was assigned), Python cannot perform the addition operation, and issues a `NameError` which is its way of saying that the identifier `z` was never assigned to a value.

Arithmetic Operators

The basic arithmetic operators in Python are summarized in the following table:

Usage	Description	Example Expression	Value
`x ** y`	Exponentiation; `x` to the power of `y`	`2 ** 5`	32
`-x`	Negation	`-42`	-42
`x * y`	Multiplication; `x` times `y`	`6 * 7`	42
`x / y`	Division; `x` divided by `y`	`6 / 4`	1.5
`x // y`	Integer division; `x` divided by `y` rounded down	`6 // 4`	1
`x % y`	Modulo; remainder after integer division of `x` by `y`	`6 % 4`	2
`x + y`	Addition; `x` plus `y`	`3 + 6`	9
`x - y`	Subtraction; `x` minus `y`	`2 - 7`	-5

[3] Here the word "binary" only conveys that the operator requires two operands, as opposed to *unary* operators which only require one operand. Do not confuse binary operands with binary numbers — the latter are entirely different.

Now that we now all of these operators, we can use Python in interactive mode like a calculator!

```
>>> 2 + 3 * 5
17
>>> 2 ** 8 + 1
257
>>> 3.5 - 1
2.5
>>> 2 * 4 + 10 * 3
38
>>>
```

The usual order of operations applies. The operators higher in the above table are evaluated before operations lower in the table. Multiplication, division, integer division, and modulo have the same precedence and if more than one of these appears in the same expression, they are evaluated from left to right. Addition and subtraction have the same precedence (but lower than the others) and again, are evaluated from left to right. Thus, in the last expression above, `2*4` happens first, followed by `10*3`, then the values of these two expressions become the operands for the addition which results in 38.

Notice that the data type of the answer depends on whether any of the operands were floating point numbers. The first expression `2 + 3 * 5` resulted in an integer result because all of the literals in the expression were integers, and none of the operators generated any floating-point results. But the expression `3.5 - 1` resulted in a floating-point number. This is because the first operand was floating point. If any operand is floating point, the result will be too because operators must operate on operands of the same type. Much of the time, however, you can use operands of different types, and the data types will be automatically converted by Python to a common type. This is called *type coercion*. Coercion only takes place when operands of an operator are of different types, and only when the different types are compatible. Python will try its best to coerce operands of different types into a compatible type, but in some situations this isn't possible. For example, you cannot use addition with a string and an integer because a string cannot be coerced into an integer; trying to do this will result in an error.

The division operator is an exception. The result of division is **always** floating-point:

```
>>> 12 / 2
6.0
>>> 12 / 8
1.5
>>> 12 // 8
1
>>> 12.0 // 8.0
1.0
>>>
```

In the first example, division of the integers 12 and 2 results in a floating point number even though both operands are integer. But observe that integer division (and modulo) follow the usual rule where the result is only floating point if one or both of its operands are.

If you took CMPT 140 or know another programming langauge

Division in Python 3 behaves differently from division in most other languages, including Python 2. In languages like Python 2, C++, and Java, division follows the same rules as the other operators such that the result of division is only floating-point if at least one of its operands are. Remember that the situation is different in Python 3!

To conclude this section, we'll observe that, as you might expect, you can override the normal order of operations by enclosing things in parentheses:

```
>>> 2 * 4 + 10 * 3
38
>>> 2 * (4 + 10) * 3
84
>>>
```

The parenthesis have higher precedence than any of the operators. Thus, the addition occurs first, then the multiplications occur in left-to-right order. The 2 is multiplied with the value of `(4 + 10)` giving us 28, then this multiplied by 3, resulting in 84.

Operators on Strings

Some operators can be applied to string operands, but their meanings are different. The "addition" of two strings results in their concatenation. The "multiplication" of a string and a number n concatenates the string with itself n times. Here are some examples:

```
>>> 'Winter' + 'is' + 'coming!'
'Winteriscoming!'
>>> 'Na' * 8 + ' BATMAN!'
'NaNaNaNaNaNaNaNa BATMAN!'
>>>
```

2.4 Console Input and Output

The *console* is the default location for text input and output from/to the computer's user. Console output goes to the screen (usually a terminal window). Console input comes from the keyboard.

2.4.1 Outputting Text to the Screen

We have seen that when we use Python in interactive mode and enter an expression, Python responds with the value of the expression. But when we use Python in non-interactive mode, this is not the case. Suppose we put the following Python program into a file called `arithmetic.py`:

```
pi = 3.14159
r = 7
pi * r**2
2 / 7 - 12
2 / (7 - 12)
```

Then suppose we run this program:

```
iroh:CMPT141 mark$ python arithmetic.py
iroh:CMPT141 mark$
```

Nothing happened! Or at least it **appears** that nothing happened because Python didn't output any responses. All of the computations in the program **did** occur, but nothing was printed out in response. In a non-interactive program we have to explicitly ask Python to print values to the console. We do this using the `print()` syntax. Here we have modified the program to print out the values of the three expressions:

```
pi = 3.14159
r = 7
print( pi * r**2 )
print( 2 / 7 - 12 )
print( 2 / (7 - 12) )
```

To use the `print()` syntax we type the word `print`, followed by whatever expression whose value we want printed enclosed in a pair of parentheses. When we run the modified program, we now get some results:

```
iroh:CMPT141 mark$ python arithmetic.py
153.93791
-11.714285714285714
-0.4
```

You can print the values of more than one expression at once by providing a comma-separated list of expressions within the parentheses. Each value printed in this manner is separated by a space character. This allows you to combine data from several literals or variables to produce a single message. Here's a program stored in `printStuff.py`:

```
print('Two to the power of six is: ', 2 ** 6)
a = 8
b = 5
print('The remainder after dividing', a, 'by', b, 'is:', 8 % 5)
```

And here is its output:

```
iroh:CMPT141 mark$ python printStuff.py
Two to the power of six is:  64
The remainder after dividing 8 by 5 is: 3
```

If you previously took CMPT 140 or learned Python 2

The `print` syntax is different in Python 3 compared to Python 2. In Python 2 the parentheses are optional because `print` was a statement, not a function. In Python 3, they are needed because `print` is a function in Python 3.

2.4.2 Reading Strings from the Keyboard

You can ask for input from the keyboard with the `input()` syntax:

```
x = input()
```

This line will pause the program, and wait for the user to type something and press the Enter key. Whatever the user typed will be stored as a string associated with the variable `x`. You can optionally ask for Python to print out a prompt to the user by providing a string inside of the parentheses. Here's an example:

```
x = input('Please enter your name: ')
print('Hello,', x)
```

Here's what happens when we run this:

```
iroh:CMPT141 mark$ python hello.py
Please enter your name: Mark
Hello, Mark
```

The line `x = input('Please enter your name: ')` first prints the string provided, then it waits for the user to type text and press enter. The bright red text was typed by the user, and this text is given the variable name `x`. The program responds by printing out a greeting using the text that was entered.

2.4.3 Reading Numbers from the Keyboard

Reading numbers is a bit tricker because `input()` will only read strings. However, we can convert the string to a number, if the string is actually a representation of a number.

```
# read an integer
x = int(input('Enter an integer: '))

# read a floating-point number
y = float(input('Enter a floating-point number: '))
```

The text that the user typed for the first `input` will be converted to an integer and given the name `x`, if that text contains only digits. Otherwise, Python will issue an error. The second `input` will convert the typed text to a floating-point number and give it the name `y`, if the text is a valid representation of a floating point number. Again, if it isn't, Python will issue an error.

3 — Functions

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a function is;
- describe the role of a function's return value;
- describe how to call a function; and
- explain what the arguments of a function are.

In Chapter 1 we talked about giving names to algorithms, as well as algorithms receiving data as input, and generating new data as output. In Python, such named algorithms are implemented as *functions*. Functions allow us to give a name to a block of Python code. If an algorithm is implemented as a function in Python, we can run the algorithm by using the function's name in a Python program. This is called *calling* the function.

Section 3.1 of this chapter introduces functions as a method of abstraction, and their relationship to algorithms (Chapter 1). Later in Section 3.2 we will introduce the Python syntax for calling functions, as well as some of the built-in functions available in Python.

3.1 Functions and Abstraction

Functions are an example of abstraction. They allow us to refer to a block of Python code by name, and ask for that code to be executed. The code within a function can be executed without knowing what it is or how it works. Let's talk about a function that you've already seen: the `print()` function. We introduced this function in Section 2.4.1 and used it to output the values of expressions to the console, but we didn't reveal until now that it is a function!

Recall from Chapter 1 that algorithms can have input. Thus, it shouldn't surprise you that functions can have input, and indeed most functions require input. Input to functions in Python are called *arguments*. When we used the `print` function before, we provided expressions as arguments,

and their values were output to the console.

We also said in Chapter 1 that algorithms can have output. Python functions can have output too! When a Python function produces output, we say that the function *returns* a value. The *return value* of a Python function is the output value that was produced.

In this way, we can use functions by providing input values (in the form of Python expressions), and receiving back output (in the form of return values). This is a nice abstraction because we can send data to the function, the function executes, and produces its output, and we don't need to know **how** that output is arrived at. All we need to know is **what** a function does, what inputs it requires, and what it returns as a result.

3.2 Calling Functions

In Python we invoke the algorithm inside of a function by *calling* the function. To call a function, we write the name of the function followed immediately by a pair of parentheses. The *arguments* to the function (inputs!) are given as a comma-separated list within the parentheses. We illustrate this with one of the examples of the `print` function we saw in Section 2.4.1:

```
a = 8
b = 5
print('The remainder after dividing', a, 'by', b, 'is:', 8 % 5)
```

We have coloured the different parts of the function call:

Red: The function name.
Green: Parentheses enclosing the list of arguments.
Blue: The arguments (inputs) to the function. Notice how each argument is an expression.
Brown: The commas separating the arguments in the argument list.

All function calls have the same general format and look like this:

```
function_name(argument1, argument2, argument3, ...  )
```

The `print` function is a bit special in that it can accept any number of arguments. Most functions are defined to have a fixed number of arguments for providing specific inputs. The `len` function in Python is an example; it accepts exactly one argument. The `len` function can be used to find out how many characters are in a string. You provide a string that you want to know the length of as the argument, like this:

```
S = "No, I am your father!"
len(S)
len("No.  No, that's not true.  That's impossible!")
```

Here we have two calls to the function `len`. The first one returns the length of the string referred to by `S`, which is 21. The second call returns the length of the string literal `"No.  No, that's not true.  That's impossible!"`, which happens to be 45. In the next section we will discuss how to obtain and use the value returned by a function.

3.2.1 Functions as Expressions: Obtaining/Using a Function's Return Value

Function calls are expressions. Like all other expressions they have a value. The value of a function call is the return value of the function! Remember, in interactive mode, if you enter an expression,

Python prints out the value of the expression. So here's what happens when we enter the expressions in the listing from the previous section:

```
>>> S = "No, I am your father!"
>>> len(S)
21
>>> len("No.  No, that's not true.  That's impossible!")
45
>>>
```

Python prints out the return values of the function calls because the function calls are expressions whose value is the return value of the function call. Since the value of a function call is its return value, you can use a function call wherever we can use an expression! Thus, function calls can be used...

- as operands of operators:

```
>>> len(S) + len("No.  No, that's not true.  That's impossible!")
66
```

- as values in assignment statements (give a name to the return value of a function):

```
>>> L = len(S)
>>> print(L)
21
```

  The return value of `len` is 21, which gets assigned the name `L`. Since `L` now refers to the value 21, the `print` function call outputs 21 to the console.

- as arguments to other functions:

```
>>> print(len(S), len('Search your feelings!'))
21 21
>>>
```

  The return values of the `len` function are the arguments to the `print` function. The two calls `len` happen first, and their return values are used as arguments to `print`. Some people call this a *nested* function call, because a call to one function (`len`) is being made as part of a call to another function (`print`). When nested function calls are used, the calls are made in order from inner-most to outer-most.

3.2.2 Calling Functions with No Arguments

It is possible for a function to be defined to have no arguments. This means that the function doesn't have any input. To call a function with no inputs, you simply leave the space between the parentheses empty. We'll see an example of this later because right now we don't know any functions that require no arguments, and there aren't any interesting built-in Python functions that require zero arguments to use as an example!

3.2.3 Functions That Do Not Return a Value: Procedures

If a function has no output and does not return a meaningful value, then value of the function call is the special value `None`. This means that, strictly speaking, it is not possible to have a function that returns nothing, because `None` is a value!

An example is the `print` function. The `print` function doesn't need to return a meaningful value because it doesn't compute any new values, it just **does** something, namely, print its arguments to the console. The following example proves that `print` returns the value `None`:

```
>>> x = print(42)
42
>>> print(x)
None
>>> x
>>> print(y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>>
```

The first line gives the name `x` to the return value of `print(42)`. Then `print(x)` proves that the value referred to by `x` is `None`. So `print` returned `None`. The next two lines show that there is a difference between the value `None`, and no value at all. Typing just `x` is fine, because `x` refers to the value `None`. But typing `y` results in a `NameError` because `y` refers to no value at all! The variable `y` was never assigned to a value.

In mathematics, a function, by definition, always has a value. Thus, functions in a programming language that do not return a value are sometimes referred to as a *procedure* since they are not, in the strict mathematical sense, functions.

3.2.4 More Built-In Python Functions

We conclude this section with a few examples of commonly used built-in Python functions shown in the table on the next page.

Name	Number of Arguments	Description and Example

max	≥ 2	Returns the maximum value of all arguments. Accepts any number of arguments.
		`>>> largest = max(10, 15, 42, 19)`
		`>>> print(largest)`
		`42`

min	≥ 2	Works like max but returns the minimum value of all arguments. Accepts any number of arguments.

pow	2	Calling pow(x,y) returns the value of x raised to the power of y.
		`>>> print(pow(7,2))`
		`49`
		`>>> print(pow(5.0, 1/2))`
		`2.23606797749979`

type	1	Returns the data type of its argument.
		`>>> type (5)`
		`<class 'int'>`
		`>>> type (2.3)`
		`<class 'float'>`
		`>>> type('foo')`
		`<class 'str' >`

int	1	Converts the argument to the integer data type (if possible) and returns the result. Strings can be converted if they contain only digits 0–9 and possibly a decimal point.
		`>>> int(42.0)`
		`42`
		`>>> int('42')`
		`42`

float	1	Similar to int; converts the argument to the float-point data type (if possible) and returns the result.
str	1	Similar to int; converts the argument to the string data type and returns the result.
input	1 (optional)	This function may optionally be given a string argument. If given, the argument is printed as a prompt, then the function waits for the user to enter a string and press the enter key. The function returns the text entered. Yes, this is the same function we used for console input in Section 2.4.2! `x = input('Please enter your name: ' )`

4 — Creating Functions

Learning Objectives

After studying this chapter, a student should be able to:

- compose functions in Python that perform a subtask and return the result;
- compose functions in Python that accept arguments as input;
- describe the role of a function's parameters;
- distinguish between arguments and parameters;
- explain the role and behaviour of the return statement;
- differentiate between function input/output and console input/output;
- author appropriately descriptive comments to document a function;
- define the concept of generalization;
- define the concept of cohesion and explain why it is desirable for functions to have high cohesion; and
- show by example how functions with parameters can be used to achieve generalization.

The ability to create your own functions and create abstractions of your own algorithms is a tremendously powerful feature in any programming language. The main reasons for writing your own functions are abstraction and decomposition of large programs into manageable pieces. We can give names to our algorithms and abstract away their details by writing them as functions. The purpose of this section is to learn how to do this in Python.

4.1 Defining Functions and Parameters: The `def` Statement

In Python, functions are defined by writing the keyword `def`. A *keyword* is a name we give to words in a programming language that have special meaning. Generally, keywords cannot be used as variable or function names, because Python will think you mean something else! The best way to see how this works is with some examples.

4.1.1 Functions that Perform Simple Subtasks

The simplest form of function is one that takes no arguments as input (i.e. a procedure!) and returns no value. In Section 2.4.2 we saw how to ask the user to enter their name, and then respond with a greeting. Suppose this is an algorithm that we want to perform frequently. We can create an abstraction of this algorithm it by writing a function called `introductions` that performs the algorithm when we call it, allowing us the luxury of not having to remember how it works. Here's what that would look like:

```
def introductions():
    x = input('Please enter your name: ')
    print('Hello,', x)
```

Let's break down what's happening here. The first line uses the keyword `def` to define a function called `introductions`. The name of a function in a function definition must be followed by a pair of parentheses, then a colon. Notice how the rest of the lines are indented. In Chapter 1, we called this a *block*. Just like in our pseudocode, we group statements together in blocks by indenting them. All of the Python code that is part of a function has to be in the same block, so it has to be indented, and be indented by **exactly** the same amount or Python will complain thinking that some lines are not part of your function even when you want them to be.

```
# Wrong indentation; Python will think that the print function
# is not part of the function, and will just execute it.
def introductions():
    x = input('Please enter your name: ')
print('Hello,', x)

# Wrong indentation; Python will issue an error here because the
# indentation is inconsistent, and it can't figure out
# whether the print function call is part of the function or not.
def introductions():
    x = input('Please enter your name: ')
  print('Hello,', x)
```

Indentation has **specific** meaning to Python. Python is not like other languages where indentation is only cosmetic. Indentation is used to indicate blocks which specify program structure.

Now, the correctly indented function definition doesn't actually do anything other than define the function. Like any function, the code it contains only gets executed when it is **called**:

```
# defines the function only:
def introductions():
    x = input('Please enter your name: ')
    print('Hello,', x)

# this function call actually calls the function,
# which executes its code.
introductions()
```

4.1.2 Functions that Accept Arguments

We have already shown you functions that accept input using arguments. To define your own function that accepts arguments, you can add a comma-separated list of variable names between the parentheses. These are called the function's *parameters*. Parameters are the variables that the function uses to refer to the arguments it is given in a call to that function. Here we have modified our `introductions` function to have a parameter called `greeting`.

```
# defines the function only:
def introductions(greeting):
    print(greeting)
    x = input('Please enter your name: ')
    print('Hello,', x)

# this function call actually calls the function,
# which executes its code.
introductions('Welcome to my Python program!')
```

This defines a function that takes one argument when you call it; the last line of the program shows the function being called with a string as an argument. The function, internally, refers to that argument by the variable name `greeting`. We have written the function so that it assumes that the greeting is a string, which it prints out prior to asking for the user to enter their name. If we execute the above Python program, this is what we will see:

```
Welcome to my Python program!
Please enter your name: Mark
Hello, Mark
```

The bright red text was entered by the user. The string `'Welcome to my Python program!'` was used as an argument to the function `'introductions'`. The argument was then assigned the parameter name `greeting`, so that within the function, the variable `greeting` refers to the string `'Welcome to my Python program!'`, which is why this is the output we get when the function executes `print(greeting)`.

A function can have any number of parameters. Each parameter you add corresponds to an argument that must be provided when the function is called. Parameters are assigned to refer to arguments in the same order they are given. For example, if a function was defined in this way:

```
def functionWithManyArguments(a, b, c, d):
    # code for function would go here

X = False
functionWithManyArguments(42.0, 'Good Morning', 17, X)
```

then the function call would assign the parameter name `a` to refer to the argument `42.0`, the parameter name `b` to refer to the argument `'Good Morning'`, the parameter name `c` to refer to the argument `17`, and the parameter name `d` to refer to the parameter `X`. When an argument is a variable, like `X` in this example, the parameter name is assigned to refer to the value that the argument refers to, so actually, the parameter `d` ends up referring to the value `False`.

Parameters are variables that get their values from the arguments in a function call; Python does the assignment of parameter to argument behind the scenes, but it is the normal kind of assignment. A key point to understand is that a parameter always refers to a value that was created outside the function. The parameter is simply the function's name for it. The value might have other variables referring to it as well, as in the above example: the value `False` has two variables referring to it, namely `X` outside the function, and the parameter `d` for as long as the function is active. Another key point is that if you assign a parameter to a new value, you are changing what the parameter refers to, and you are not changing the old value.

4.1.3 Returning A Value

We've seen how to use parameters to define a function that accepts inputs (arguments). We've also seen built-in Python functions that return a value. So how do we have one of our own functions return a value?

The answer is pretty simple: Write the keyword word `return`, followed by an expression. The value of the expression becomes the return value for the function, and the value of the call that invoked the function. For example, we could modify our `introduction` function to return the name that the user entered, so that it can be used by the caller for future reference:

```
# defines the function only:
def introductions(greeting):
    print(greeting)
    x = input('Please enter your name: ')
    print('Hello,', x)
    return x

# this function call actually calls the function,
# which executes its code.
username = introductions('Welcome to my Python program!')
```

In this example, the return statement at the end of `introductions` causes the value referred to by `x` (the text the user entered) to be returned. The execution of the program then resumes immediately after the function call to `introductions`, and since the name `username` was assigned to the return value of the function call, it now refers to the text that the user entered. Once the function has returned, and execution has resumed after the function call, the variable `x` no longer exists. Returning a value is one way of getting data out of a function.

We will see later that functions may have more than one return statement[1]. As soon as a return statement is executed, regardless of where it appears in the function, execution of the function immediately ceases (even if there are lines of code after it!), the value of the accompanying expression is returned, and execution continues from the line immediately after the call that invoked the function (or, in some cases, the line containing the function call continues executing, e.g. if the function call was part of a variable assignment, the assignment occurs after the function call returns).

[1] Some are of the opinion that functions with more than one return statement is bad style! Some believe otherwise. We really aren't too worried about it.

4.1.4 Returning Nothing

If a function does not need to return a value, then simply do not include a return statement. After the execution of the last line of the function, the value `None` will be returned by default. As we noted before, such a function is sometimes called a *procedure*.

4.1.5 Defining Before Calling

Python functions must be defined before they are called. Thus a function definition must appear in a file prior to any calls to that function. In the following code, the first call to `introductions` would fail and cause a `NameError`. But the second call to `introductions` would work fine because its definition appears first.

```
# this fails -- function called before definition
username = introductions('Welcome to my Python program!')

def introductions(greeting):
    print(greeting)
    x = input('Please enter your name: ')
    print('Hello,', x)
    return x

# this is fine - function called after definition.
username = introductions('Welcome to my Python program!')
```

4.1.6 Summary

If we want to write a function that has inputs, we need to give it parameters. Parameters are the variable names that are used within the function to refer to the values of a function's arguments; they are given in the function's **definition**. Arguments are the input values for the function; they are provided when the function is **called**. A function can be instructed to return a value (i.e. produce an output!) using the `return` keyword. The code for the function must be indented in a *block*. Indentation has semantic meaning in Python and must be used properly and with care.

4.2 Variable Scope

The *scope* of a variable refers to the parts of the program in which a variable exists after it is assigned to a value. This sounds complicated but it's actually quite straightforward. Variables defined within a function only exist within that function. Variables defined outside any function do not exist within any functions. Here's an example:

```
def fireball_damage():
    damage = 30
    return damage

damage = 0
D = fireball_damage()
print(damage)
```

In this example, you might expect the value 30 to be printed. In fact, the value 0 is printed. To see why this is, you must realize that we have two **different** variables called `damage` in this program. The `damage` variable defined in the function `fireball_damage` only exists within the function – we say that it is a *local* variable. Likewise, the `damage` variable defined outside of the function only exists outside of the function. It's scope is outside of any function, so it is local to the main program. Thus, the `fireball_damage` function is changing only what the local variable `damage` refers to, not what the `damage` variable defined outside of the function refers to. Moreover, two different functions can use the same variable name, but they are, in fact, completely different and unrelated variables!

If you took CMPT 140

In CMPT 140 you probably encountered the Python keyword `global` which allows functions to access variables defined outside of the function. This was necessary because of the specialized Processing programming framework in which you were working. In general, we normally avoid the use of global variables because they can cause unexpected side effects and bugs when we inadvertently refer to the same variable when we don't mean to. In CMPT 141 we do not allow the use of global variables. In almost all situations where global variables seem appropriate there exists a better way.

4.3 Console I/O vs Function I/O

In computer science, we use the words *input* and *output* a lot, and we use them to mean different things. For example, the inputs and outputs of a **function** are different and distinct from *console input* and *console output*. Function inputs are the arguments of a function call assigned to function parameters; function outputs are returned from data created within a function. Console inputs are read from the keyboard; console outputs are printed to the screen rather than being sent to another part of a program.

When reading instructions it is important to be able to distinguish these forms of input and output. If you are asked to write a function that "takes" something as input, this means the function should take an argument via a parameter. If you are asked to write a function that "reads from the console", or "asks the user" for some data, this means that the function should perform console input to get this data by calling the `input` function. If you are asked to write a function that "outputs", "prints", or "displays" some data, then this should be done with console output by calling the `print` function. If you are asked to write a function that "returns" some data, then this should be done using the `return` keyword.

4.4 Documenting Function Behaviour

Python does not restrict the data type of function arguments, so you can pass an argument of any type as the argument for any parameter of a function. But usually functions expect arguments to be of a certain data type. How do we communicate these expectations to the programmer who wants to call the function?

When we write a function we should *document* what its inputs and outputs are. We can do this by writing *docstrings* in our program. Docstrings are a way of describing what the function does, what each parameter is for, the expected data type of the argument to that parameter, and what the function returns (if anything). Here's how we would do this for our `introductions` function:

```
def introductions(greeting):
    """
Greet the user and asks them for their name.

greeting: A string containing a message to greet the user
Returns: The name entered by the user.
    """
    print(greeting)
    x = input('Please enter your name: ')
    print('Hello,', x)
    return x

# this function call actually calls the function,
# which executes its code.
username = introductions('Welcome to my Python program!')
```

The docstring is enclosed in triple double-quotes and is indented with the rest of the block of code for the function.[2] The triple double-quotes are how you specify a multi-line string literal in Python. The docstring should contain a brief one-line description of what the function does, followed by a list of parameters, and what they are for, followed by a description of what the function returns. There are no particular formatting requirements for the contents of the docstring, but you should strive for something similar to the above.

If a function has a docstring, you can view it by typing `print(`*`functionname`*`.__doc__)`.

```
>>> print(introductions.__doc__)

Greet the user and asks them for their name.

greeting: A string containing a message to greet the user
Returns: The name entered by the user.
```

It also works for built-in functions, like `pow` or `max`:

```
>>> print(pow.__doc__)
Equivalent to x**y (with two arguments) or x**y % z (with three arguments)

Some types, such as ints, are able to use a more efficient algorithm when
invoked using the three argument form.
```

4.5 Generalization

Generalization of functions (or algorithms) is the process of modifying a function/algorithm that solves a specific problem so that it can solve a wider range of problems, or a larger number of instances of the same problem. Let's consider a totally imaginary video streaming service, Netflux. Suppose we want to compute how many users can simultaneously stream a movie on Netflux on

[2] The first set of triple double-quotes must be indented, but the rest of the docstring need not be because Python interprets the entire docstring as a single line of text.

an 25Mbps internet connection, and we know that each user that is streaming requires 3Mbps. We could write a function to do this:

```
def how_many_netflux_streams():
    return 25 // 3;          # use integer division since we
                             # can't have a fraction of a user
```

Now we can call this function whenever we need to know how many users can simultaneously stream Netflux, without having to remember how that is calculated. But what if some people have faster or slower internet connections? We could generalize this function to apply to those situations by making the speed of the internet connection a parameter:

```
def how_many_netflux_streams(speed):
    return speed // 3;
```

Now we have a function that can solve the same problem in a much wider range of situations! Can you think of how we might generalize this further (see the footnote for the answer!)?[3]

The more general a function is, the more re-useable it is. The more often we can re-use existing code that has been tested and proven to work, rather than write new code, the less likely we are to introduce errors into programs.

Of course, there is a limit to this. Functions that do too much or too many different things are actually bad. Thus, generalization must be tempered by another concept called *cohesion* which we discuss in the next section.

4.6 Cohesion

In software design, the term *cohesion* refers to the idea that code that is grouped together should have something in common. In terms of writing functions, functions that perform one task and one task only are said to have high cohesion. Functions with high cohesion are preferred because they increase the reusability and maintainability of software components. Our function from Section 4.5 that computes how many users can stream Netflux on an internet connection of a certain speed has high cohesion because it performs a single, well-defined task.

An example of low cohesion would be a function that, say, not only computed the number of users that can simultaneously stream on a connection but also includes a parameter that changes the user's streaming quality (e.g. standard or high definition). These are two different tasks that are entirely independent, and should be implemented in separate functions.

[3] What if Netflux improves their video quality, and each user requires instead 5Mbps to stream? We can make our function work in even more situations, including this one, by making the bandwidth needed to stream one movie a parameter as well!

5 — Objects

Learning Objectives

After studying this chapter, a student should be able to:

- describe what an object is;
- explain the role that functions play in the encapsulation of the data stored in an object; and
- author Python code that can call a function known to be defined by an object.

5.1 Objects and Encapsulation

An *object* is a combination of data and functions. The data inside of an object cannot be accessed directly. This is called *encapsulation* or *information hiding*. The data inside an object can only be manipulated by calling the object's functions. This ensures that a user cannot modify the data within an object in a way that is invalid, or that is unexpected. In some programming languages, including Python, functions defined within objects are called *methods*.

In Section 2.1.3 we told you that in Python, a *string* is a compound data type, and this remains true. But strings are **also** objects! The data stored in a string object is, as we said before, a sequence of characters. But in addition, string objects also contain methods (i.e. functions) for manipulating their data. These include methods for:

- determining if a string contains spaces;
- converting all characters in the string to lower-case;
- determining if a string contains a certain sequence of characters; and
- many others.

In the next section we'll show you how to call methods defined in objects.

If you took CMPT 140...

... you may remember that the term *encapsulation* was used to describe **procedural** encapsulation, where we "package" an algorithm as a function so that it can be re-used easily. Here we are applying the same term to the "packaging" of data as well. Objects encapsulate both data items and a group of algorithms that operate on that data.

5.1.1 Calling Methods in Objects

Let's suppose we have a variable name `s` that refers to a string. Python strings have a method called `isdigit` which returns a boolean value indicating whether the string contains **only** digits between 0 and 9. To call the `isdigit` method in the string object referred to by `s`, we write `s.isdigit()`. We call a method in an object exactly the same way we call a function, except that we have to precede the method call with the object in which we want to call the method, followed by a period. Here's a more complete example:

```
>>> s = 'This string contains 4 and 2, but not *only* digits.'
>>> t = '421'    # this string contains only digits.
>>> s.isdigit()
False
>>> t.isdigit()
True
>>>
```

The first call of `isdigit` is on the string object `s`, which is a string containing other characters in addition to digits, so it returns `False`. The second call to `isdigit` is on the string object `t`, which contains only digits, so it returns `True`. Here's another example showing how to use the `lower` method of a string to convert all of it's characters to lower-case:

```
>>> mystring = 'On the Internet CAPITAL letters mean SHOUTING!'
>>> mystring.lower()
'on the internet capital letters mean shouting!'
>>>
```

In general, if the variable `x` refers to an object which has a method named `y`, you can call the method `y` in object `x` using `x.y()`.

Just like functions, methods can be defined to require arguments. An example would be the `find` method defined by Python strings which is used to determine whether a string contains another string. The argument to the `find` function is the string we wish to find in the object in which we are calling `find`. So the method call `mystring.find('Internet')` searches for the location of the string `'Internet'` in `mystring`. Let's try it and see what the result is:

```
>>> mystring = 'On the Internet CAPITAL letters mean SHOUTING!'
>>> mystring.find('Internet')
7
```

Our method call returned 7! This is because the string `'Internet'` occurs beginning at 7 characters to the right of the first character in the string (count it!). If the `find` method does not find the given string in its object's string, it returns -1.

So to summarize, we can call a method in an object in the same way as we call functions, we just need to use the dot notation to specify which object we want to call the method on. We will be calling methods in objects quite a lot. We'll also be seeing many other kinds of objects other than strings.

5.1.2 Mutable vs Immutable Objects

We have discussed that some Python objects are immutable, e.g., strings. But there is nothing special about immutable objects apart from the kinds of methods that they have. An immutable object has no methods that allow a program to change the data stored inside the object. Immutable object methods can return new values only. In contrast, if an object has methods that allow a program to change the data inside the object, we call the object *mutable*. We'll see list and dictionaries as examples of mutable objects. The only difference between mutable and immutable objects is the behaviour of the methods!

5.1.3 Defining Our Own Objects

In case you were wondering, it is possible to create your own objects in Python, and define your own data and methods within them. But this is beyond the scope of this course. We only need to know how to use existing objects that someone else has created. Defining our own objects is a topic you will encounter if you go on to take CMPT 145.

6 — Modules

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a module is and why one would want to use one;
- identify and author Python code to make the functions in a module available to a program;
- design and author programs that make use of functions from modules; and
- be able to locate, and understand the documentation for the functions of a module.

6.1 Modules: What Are They and Why Do We Need Them?

Modules are files that contain function and object definitions that add additional, and much more powerful features to Python. The basic Python language provides only very fundamental building blocks for programs. Modules are a way for people to share functions and objects that they have written so that other people can use them in their own programs. In this respect, modules are similar to the *libraries* that are used by other programming languages such as C++ and Java. Viewed another way, modules contain *abstractions* of algorithms that we can use in our own programs without having to understand how they work.

6.2 How to Use Modules

Modules are stored in separate files from our own programs. Thus, in order to use the functions and objects defined by a module, we have to tell our own program to look in those files and read those definitions. We do this using a Python keyword called `import`.

Perhaps you noticed that Python, by default, doesn't seem to be able to compute common mathematical functions like logarithms and finding the square root of a number, or trigonometry functions such as sine, cosine, and tangent. The reason for this is that these functions are instead

defined in a module called `math`. If we want to use the functions in the `math` module, we need to `import` them from the `math` module into our program. For example:

```
vault-of-knowledge:~ mark$ python
>>> log10(1000)  # this won't work, there is no such function.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'log' is not defined

>>> import math as m   # read the definition of the log function
>>> m.log10(1000)      # now we can compute base-10 logarithms!
3.0
```

Our first call to `log10` fails, because Python does not have a built-in function called `log10`. The command `import math as m` reads the function definitions from the `math` module and creates an object called `m` that contains the functions defined by the `math` module (i.e. the functions defined by `math` become methods of `m`). Since the `log10` function is defined in the `math` module, the object `m` contains the `log10` method, and we can call it in the same way we call any method in an object using the dot notation we learned in Section 5.1.1. As we can see, above, `m.log10(1000)` returns the correct value `3.0`.

Below is a program showing some more examples of using functions from the `math` module.

```
import math as m

print(m.sqrt(-7.5))     # display square root of 7.5
print(m.exp(5))         # display e to the power of 5
print(m.log2(256))      # display the base-2 logarithm of 256
angle = m.radians(90)   # convert 90 degrees to radians.
print(m.sin(angle))     # display the sine of 90 degrees.
```

Run this program in Python, and you'll see that it produces the output described. If you're wondering why we used the `radians` function to convert the angle 90 degrees to radians, it's because the `sin` function requires that its argument be an angle in radians.

If you want to see a complete list of the functions in the `math` module, and documentation on how to use them, click on the following link, or copy it into your web browser: https://docs.python.org/3/library/math.html.

Import Syntax

In general, the syntax for importing modules is:

`import` *x* `as` *y*

x must be the name of a module, and *y* must be a valid variable name. This creates an object called *y* that contains, as methods, the functions defined in *x*.

6.3 What Other Modules Are There?

In this course, we use a Python distribution called Anaconda. Anaconda comes with a lot of modules, too many to list here. This is one of the great things about Python. There are so many modules

available for it, that there is probably a module to either do, or help you do almost anything you can think of. It is also possible to obtain and use modules that do not come with Anaconda. These take the form of Python program files (files with a `.py` extension) that can be placed in the same folder as your program, and then imported. It is also possible to write your own modules.[1]

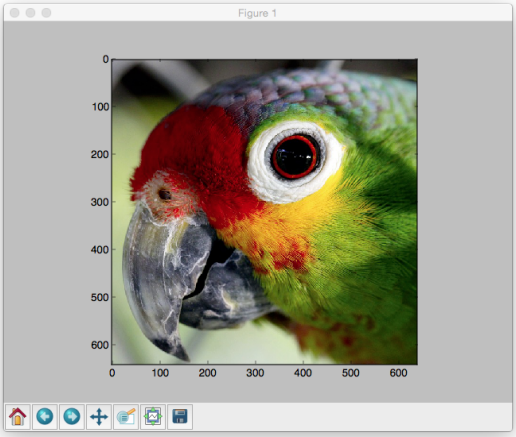
In this course we will be using several different modules that come with Anaconda, including ones that can:

- read, write, modify, and display image files;
- plot line and bar graphs; and
- draw graphics to the screen.

Let's look at one more example right now from `skimage`, otherwise known as *scikit-image*. `skimage` lets us use and/or manipulate images. The following Python code reads a JPEG image file, specified by a file name, and displays it to the screen:

```
import skimage.io as io
im = io.imread("images/parrot.jpg")
io.imshow(im)
io.show()
```

The first line of this code reads the definition of an object called `io` from the module called `skimage.io`. The second line calls the `imread` method of the `io` object which reads the given image file and returns another object (of a different kind than `io`) containing the image data from the `parrot.jpg` file[2]. The third line calls the `imshow` method of `io` which adds image `im` to a queue of images to be displayed on the screen. The fourth line calls the `show` method of the object `io`, which causes the queued image to pop up in a window. It looks something like this:



This image from the public domain was download from pixabay.com.

Look at that cute parrot. He's gorgeous! Now think about how much is actually going on behind the scenes in those few lines of code. The file on the disk has to be opened, the data has to be

[1] A module is just a `.py` file that contain only function and/or object definitions. You can write your own group of related functions and use them as a module!

[2] The image data is returned as an array object! We'll learn more about arrays and what we can do with them in a later chapter.

decompressed, decoded, and loaded into memory, and then it has to sent to the display hardware. These are all quite complex operations with many many steps. But thanks to abstraction, we accomplished all that with just three simple method calls to `imread`, `imshow`, and `show`.

We will be exploring some more of the capabilities of the `skimage` module in class.

6.4 Finding Module Documentation

At this point, I am sure you are wondering how one finds out about modules, and how to use them. The short answer is: internet search. For example, a search for "skimage display image" returns us a link to the documentation for the `skimage.io` module. That link is here: http://scikit-image.org/docs/dev/api/skimage.io.html.

While it can sometimes be hard to find documentation for modules, rest assured that, for this course, we'll always tell you how to use a module function or object that we expect you to use, or at least tell you exactly where the documentation is.

7 — Indexing and Slicing of Sequences

Learning Objectives

After studying this chapter, a student should be able to:

- be able to use indexing to obtain a desired character in a string;
- be able to use slicing to obtain substrings of a string;
- be able to use non-unit step size to select non-contiguous string characaters at regular intervals.

7.1 Sequences

A *sequence* is a compound data type consisting of one or more pieces of data in a specific linear ordering. An example is a string — it is a sequence of characters. Python defines two important operators called *indexing* and *slicing* that can be applied to sequences, including strings.

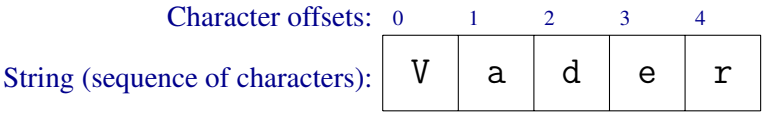
Later, we shall find out that lists and arrays in Python are also examples of sequences, and we'll be able to apply the indexing and slicing operations we learn here to those data types as well.

7.2 Indexing

Each data item in a sequence has a position. We denote that position using an integer which we call an *offset*. The item at the beginning of a sequence has offset 0. The second item in a sequence has offset 1, the third has offset 2, and so on. In general, if an item is n positions to the right of the first position, it has offset n. This is why we call it an *offset*. The second item of a sequence is offset by 1 position from the first position. The fifth item in a sequence has offset 4, because it is offset by 4 positions from the first position — if you start at the first position and move right 4 times, you'll be at the fifth item.

You can also think of it this way: if an item is the i-th item in a sequence, it has offset $i-1$. The following picture shows that the string `'Vader'` can be viewed as a sequence of characters with

offsets 0 through 4:



In Python, you can access an item in a sequence using its offset. This is done by putting the offset of the desired item inside a pair of square brackets after the sequence. The square brackets are the *indexing* operator. Since Python strings are sequences, we can use indexing to access specific characters within a string. This works with both string literals and string variables:

```
>>> 'Vader'[3]          # Get 4th character from literal
'e'
>>> s = 'Skywalker'     # Make s refer to 'Skywalker'
>>> s[0]                # Get first character from string s
'S'
>>> s[4]                # Get fifth character from string s
'a'
>>> c = s[8]            # Get 9th character from s, give it the name c
>>> print(c)            # print c (the 9th character from s).
r
>>> print(s[2])         # print the third character of s
y
>>> s[0]+s[2]+s[4]      # Concatenate the 1st, 3rd, and 5th characters
'Sya'
>>> x = 7
>>> s[x]                # Offset 7 since x refers to 7
'e'
>>> s[x+1]              # Offset 8 since x refers to 7
'r'
```

In each example above, the indexing operator obtains the character from the string at the given offset. Then we can do what we want with it: assign a variable name to it, pass it to a function, use it in an expression, etc. Also note that offsets can be integer literals, variables that refer to integers, or integer-valued expressions.

7.2.1 Offsets from the End

It is also possible to specify an offset from the **end** of the sequence using negative integers. Offset -1 is the offset of the last character. Offset -2 is the offset of the second-last character, offset -3 is the offset of the third-last character etc.. Examples:

```
>>> t = 'TARDIS'   # Make t refer to 'TARDIS'
>>> t[-1]          # Access the last character of t
'S'
>>> t[-3]          # Access the third-last character of t
'D'
```

7.2.2 Invalid Offsets

If you use an offset that does not exist or is of the wrong type, Python will issue an error. Remember that **offsets must be integers**. Positive offsets of a string `s` must be between `0` and `len(s)-1`, while

negative offsets must be be between `-len(s)` and `-1`. Here are some of the things that can go wrong if you use an offset that is out of range:

```
>>> s = 'Ice King'  # Make s refer to 'Ice King'

>>> s[9]            # Offset out of range
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range

>>> s[-10]          # Offset out of range
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range

>>> s[5.0]          # Floats cannot be offsets. Ever.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string indices must be integers
```

7.3 Slicing

Slicing is the act of selecting zero or more items of a sequence and forming them into a new sequence. Slicing is similar to indexing but it allows us to specify multiple offsets at once using a convenient syntax. The result of slicing is a new sequence consisting of the items at the specified offsets.

We can specify a contiguous range of offsets using the : operator — this is the *slicing* operator. If we write $x:y$, where x and y are integer expressions, this means the range of offsets between x and $y-1$. That's right, $y-1$. The range of offsets is **inclusive** on the lower end and **exclusive** on the upper end. Thus, `0:42` actually specifies the range of offsets $0, 1, 2, \ldots, 41$. 42 is not included!

We can use the slicing operator to specify multiple offsets for the indexing operator to obtain substrings of a string in Python:

```
>>> s = 'Skywalker'
>>> t = s[3:9]       # get the substring of s between offsets 3 and 8
>>> print (t)
walker
>>> print (s[0:3])   # get the substring of s between offsets 0 and 2
Sky
```

The exclusion of the item at the upper offset of the slicing operator in the resulting sequence probably seems strange now, but it's actually quite convenient. For example, if `s` is a string, then `s[x:len(s)]` extracts the substring beginning at offset `x` and ending at offset `len(s)-1`, which is the last valid offset.

7.3.1 Slicing with a Non-Unit Step Size

You can select every second, third, or *n*-th item between the start and end indices by specifying a second colon and a third integer:

```
s = 'Skywalker'
>>> s[0:len(s):2]     # every other character in s
'Syakr'
>>> s[2:7:3]          # every third character between offsets
                      # 2 and 6 in s.
'yl'
```

The third integer is called the *step size* for the slicing operation.

7.3.2 Slicing with Invalid Offsets

Providing an invalid offset when indexing results in an error, as we have seen. However, providing an invalid offset as the starting or ending offset of a slicing operation does **not** result in an error. The slicing operator includes in the resulting sequence all of the original sequence items that occupy valid offsets within the specified range. Invalid offsets within the specified range are ignored. Moreover, nonsensical slicing where the starting offset is to the right of the ending offset results in an empty sequence.

```
s = 'Skywalker'
>>> s[5:25]     # valid offsets between 5 and 24 (i.e. 5 through 8)
'lker'
>>> s[-55:-5]   # valid offsets between 55th last and 6th last offset.
'Skyw'
>>> s[5:3]      # nonsense results in an empty sequence
''
```

Note that in the second example, offset -5 is excluded because the ending offset is always excluded when slicing.

8 — Control Flow

Learning Objectives

After studying this chapter, a student should be able to:

- identify and define the behaviour of relational operators, logical operators, and Boolean expressions in Python;
- identify and author correct Python language syntax for branching statements: if, if-else, if-elif-else, and chained statements; and
- design and author Python programs that use if, if-else, nested if, and chained-if statements.

8.1 Relational Operators and Boolean Expressions

An operator that produces a result that is either `True` or `False` is called a *relational operator*. Relational operators are used to ask simple "true or false" questions about how one piece of data is *related* to another. Thus, relational operators always have two operands. For example, the value of the expression `2 < 4` is `True`. This is because the < operator is the “less than” operator. More generally, the expression x `<` y has the value `True` if the value of x is smaller than the value of y. The following table lists several commonly used relational operators in Python.

Operator	Meaning	Example	Result
==	are the operands equal?	42 == 42	True
!=	are the operands unequal?	42 != 42	False
<	is the first operand smaller than the second operand?	10 < 42	True
>	is the first operand larger than the second operand?	'Bill' > 'Lenny'	False
<=	is the first operand less than or equal to the second?	42 <= 42	True
>=	is the first operand greater than or equal to the second?	'R' >= 'Z'	False

Notice how the operators work with non-numeric data as well, like strings and characaters. In such cases the comparison is made lexicographically (dictionary ordering). `'Bill'` is not greater than `'Lenny'` because `'Bill'` comes before `'Lenny'` in dictionary ordering. For the same reason, the expression `'Bill' < 'Lenny'` has the value `True`.

Relational operators all have the same precedence and so, are evaluated from left-to-right. But all relational operators also have a lower precedence than all arithmetic operators, which means arithmetic operators get evaluated first. Thus the expression `5 + 5 < 10` is `False` because the addition happens first, resulting in the value 10. Since 10 is not less than 10, the < operator evaluates to `False`.

Boolean Expressions

A *Boolean expression* is any expression whose value is either `True` or `False`. Thus, all of the expressions in the third column of the above table are Boolean expressions.

8.2 Logical Operators

The operators `and`, `or`, and `not` are *logical operators* (also called *Boolean operators*). They are so-called because the operands of logical operators must be Boolean values. Thus the operands of logical operators can either be Boolean values or other Boolean expressions. We can use logical operators to ask questions about Boolean values or Boolean expressions.

All logical operators have a **lower** precedence than relational operators. So that means that relational operators always get evaluated before logical operators.

8.2.1 The `and` Operator

The expression *x* `and` *y* has a value of `True` only if both *x* and *y* are `True`. In all other cases, such an expression has a value of `False`. Remember that *x* and *y* could be Boolean literals, Boolean values, Boolean expressions, or even a function call that returns a Boolean value. Here are some examples:

Expression	Value
`1 - 1 > 0 and -2 > 0`	False
`False and 'x' < 'y'`	False
`9 >= 9 and 'FortyTwo'.isdigit()`	False
`5 < 10 and 20 != 42`	True
`len('Skywalker') > 0 and len('Skywalker') < 10 and 'Ren' < 'Rey'`	True

Note the order of operations in the first example. The subtraction happens first, because it has higher precedence than all relational and logical operators. Then the two greater-than operators are evaluated because relational operators have higher precedence than logical operators. The last thing that happens is the `and` operator. Since both > operators result in `False`, the entire expression is `False`.

In the third example, we call the `isdigit` method on the string `'FortyTwo'`. Since the string `FortyTwo` doesn't contain digits, the function returns `False`. Therefore, even though the relation `9 >= 9` is `True`, the entire expression has the value `False`.

In the last example, the two `and` operators are evaluated left-to-right. The result of the first `and` is `True`, which becomes the first operand to the second `and`, then `True and 'Ren' < 'Rey'` evaluates to `True`, so the whole expression evaluates to `True`.

8.2.2 The `or` Operator

The expression *x* `or` *y* has a value of `False` only if both *x* and *y* are `False`. In all other cases, such an expression has a value of `True`. Here are some examples of expressions using `or`:

Expression	Value
`5 < 7 or 0 == 0`	`True`
`7 < 5 or 0 == 0`	`True`
`2**5 < 16 or max(7, 42) == 7`	`False`
`'Skywalker'.find('Anakin') > -1 or 'Skywalker'.islower()`	`False`

The last example is `False` because `'Anakin'` is not a substring of `'Skywalker'` so the `find` function returns -1. -1 is not greater than -1, so the first operand to `or` is `False`. `'Skywalker'.islower()` is also `False` since `'Skywalker'` does not consist only of lowercase characters. Thus, both operands are `False`, so the `or` evaluates to `False`.

8.2.3 The `not` Operator

The `not` operator is a unary operator. It only takes one operand. The expression `not` *x* has a value of `True` only if *x* is `False`; it has a value of `False` if *x* is `True`. So `not` changes the Boolean value of its operand to the other Boolean value. Here are some examples:

Expression	Value
`not 42 < 0`	`True`
`not 6 == 6`	`False`
`not max(17, 50) > 80`	`True`

In the last example, the function call `max` has the highest precedence; it returns 50. The next highest precedence is the > operator (relational operators have higher precedence than logical operators), which results in `False` since 50 is not greater than 80, then `not False` results in `True`.

8.2.4 Mixing Logical Operators

We don't want you to get the idea that you can only use one kind of logical operator per expression. You can mix them up as much as you like, but take care — **the logical operators do not have the**

same precedence! The operator `not` has higher precedence than `and` which, in turn, has higher precedence than `or`. Take a look at these expressions:

Expression	Value
`not 5 < 7 or 0 == 0`	`True`
`not (5 < 7 or 0 == 0)`	`False`
`len('Vader') < 7 or len('Maul') < 3 and 'Vader' < 'Maul'`	`True`
`(len('Vader') < 7 or len('Maul') < 3) and 'Vader' < 'Maul'`	`False`

You might expect the first expression to have a value of `False`, because `5 < 7 or 0 == 0` is clearly `True`, and the `not` would change that to `False`. But the `not` operator has higher precedence than `or`. In this expression, the relational operators evaluate first, giving us `not True or True`. Now the `not` is applied to the first `True`, giving us `False or True`, which ends up as `True`. If we really want to apply `not` to the result of the `or`, we have to add parentheses, like in the second example. The relational operators still evaluate first, again giving us `not (True or True)`. But now, because of the parentheses, the `or` evaluates next, which gives us `not True`, and ultimately `False`.

Note how in the third and fourth examples, if we want the `or` to evaluate before the `and` we have to use parentheses around the `or` expression. You can see that it matters because we get different answers depending on which of `or` or `and` evaluates first.

8.2.5 Variables in Relational and Logical Expressions

We also don't want you to get the idea that you can't use variables with these operators. In any of the examples above where a literal appears in an expression, we could also replace the literal with a variable. For example, `a < b and c < d`. We just can't evaluate this without knowing the values of the variables. Here's a complete example where we associate the variable names with values and use them in a Boolean expression:

```
>>> a = 1
>>> b = 5
>>> c = 2
>>> d = 4
>>> a < b and c < d
True
>>>
```

8.3 Branching and Conditional Statements

Now that we know how to ask questions about data using Boolean expressions, we can use the values of Boolean expressions to get our programs to perform different actions depending on the value of a Boolean expression. This is called *branching* and it allows us to perform one block of code if an Boolean expression is `True`, and a different one if it is `False`.

In Python, we perform branching using a *conditional statement* or *if-statement*. The syntax is the word `if`, followed by a Boolean expression, followed by a colon, like this:

```
if condition:
```

The if-statement is then followed by a *block*. Remember blocks from Chapter 1? A block is a series of indented lines of code. The block of code following the if-statement is only executed if the condition in the if-statement evaluates to `True`. Let's look at an example:

```
guess = int(input('Guess a number between 1 and 100'))
if guess >= 1 and guess <= 100:
    print('That was a valid guess!')
```

Listing 8.1: A program that uses a conditional statement.

The first line of this example asks the user to input a number between 1 and 100. The name `guess` is assigned to the value entered. Then we have an if-statement. The condition of the if-statement is the Boolean expression `guess >= 1 and guess <= 100`. The value of this expression will, of course, depend on the value of `guess`. If `guess` is, in fact, between 1 and 100, the Boolean expression is `True`, and the one-line block of code consisting of the `print` call is executed. Otherwise, it is not. Here is what we see if we run the program, and enter the number 50 (green text is text entered by a user):

```
Guess a number between 1 and 100: 50
That was a valid guess!
```

Since the Boolean expression in the if-statement is `True`, the indented block consisting of the call to `print` is executed. If we enter a value that is not between 1 and 100, the `print` call will not execute and we will not see the output `That was a valid guess!`. But what if we want to print something different if the guess is not between 1 and 100? It might be natural to try this:

```
guess = int(input('Guess a number between 1 and 100: '))
if guess >= 1 and guess <= 100:
    print('That was a valid guess!')

print('That was not a valid guess.')
```

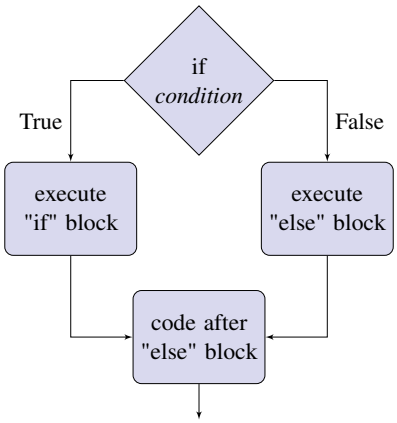
Listing 8.2: A program that uses a simple conditional statement.

But this won't work because the second call to `print` will execute regardless of whether the Boolean expression in the if-statement is `True`. What we need is a way of specifying a second block that gets executed only if the Boolean expression in the if-statement is `False`. We can do this using an *else-statement*. An else-statement is the word `else` followed by a colon:

```
guess = int(input('Guess a number between 1 and 100: '))
if guess >= 1 and guess <= 100:
    # This block executes if the condition is True
    print('That was a valid guess!')
else:
    # This block executes if the condition is False
    print('That was not a valid guess.')
```

Listing 8.3: A program that uses an if-else statement.

Now, if we enter a number that is between 1 and 100, it will execute the first block of code. Otherwise, it will execute the else statement's block of code. In general, the flow of execution for conditional statements looks like this:




```
if condition:
    # if block (indented)
else:
    # else block (indented)

# code after else block
```

Now suppose we wanted to give the user a little more information about why a guess was invalid. If the user guessed a number that was too large, we want to print out `Too high!`. If they guess too low, we want to print out `Too low!`. Otherwise, we want to print out `That was a valid guess`. Here's one way we could do that:

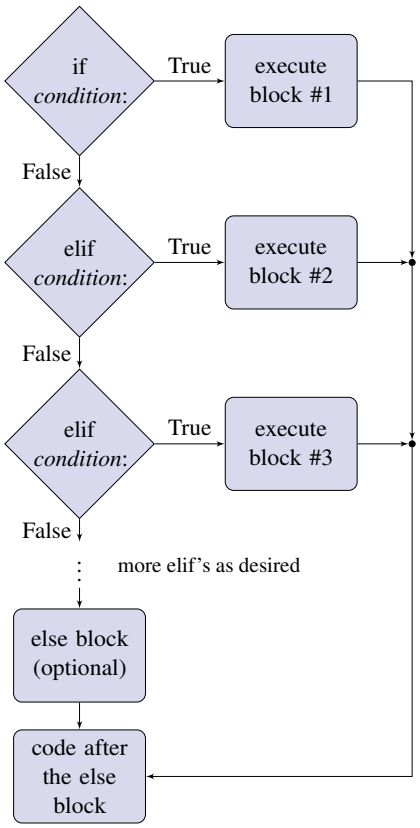
```
guess = int(input('Guess a number between 1 and 100: '))
if guess < 1:
    print('Too low!')

if guess > 100:
    print('Too high!')

if guess >= 1 and guess <= 100:
    print('That was a valid guess!')
```

But Python, and most other programming languages give us a cleaner way to do this that guarantees that only one of a series of blocks can be executed. In Python, there is an elif-statement ("elif" is short for "else if"). An elif-statement consists of the word "elif", followed by a Boolean expression, followed by a colon, followed by a block of statements to execute if the Boolean expression is `True`. An elif-statement can appear after the block associated with an if-statement or another elif-statement, but is only executed if the preceding if- or elif-statement was found to be `False`. So here's a different way we could write our program which does the same thing, but is a bit easier to read:

```
guess = int(input('Guess a number between 1 and 100: '))
if guess < 1:
    # If guess was less than one, execute this block.
    print('Too low!')
elif guess > 100:
    # Otherwise, if guess is larger than 100, do this block.
    print('Too high!')
else:
    # Otherwise, execute this block.
    print('That was a valid guess!')
```

Note that only one of the three blocks is executed. As soon as an if- or elif- statement is `True`, its block is executed and no more if- or elif- statement conditions are tested, and no more of the

blocks can execute. The final else block only executes if none of the preceding conditions were `True`. Once one of the blocks executes, the execution continues at the first line of code following the else block. Multiple elif-statements and accompanying blocks are allowed as long as the first conditional statement is an if-statement. In all cases the else statement is optional. The flow of execution in an if-elif-else chain is described by the following flowchart and code template:




```
if condition:
      # block 1 (indented)
elif condition:
      # block 2 (indented)
elif condition:
      # block 3 (indented)

# ... more elif's as desired

else:  # (optional)
      # else block

# code after the else block
```

Notice that only one of the blocks in the if-elif-elif-...-else chain can execute no matter how many elif-statements there are. Finally, remember that the blocks can consist of multiple lines of code, as long as they are all indented.

```
# suppose smaller and larger are variables referring to
# integer values
if smaller > larger:
     # swap the values referred to by the variables
     temp = smaller
     smaller = larger
     larger = temp
```

Because all three lines after the if-statement in the above code are indented, they are all part of the block, and all three only get executed if the if-statement's condition is `True` . Block indentation must be such that every line of every block is indented by the same amount, otherwise Python will not understand your program. Moreover, the indentation must be either all spaces or all tabs, you can't mix them. However, most text editors that are Python-aware (e.g. PyCharm, TextWrangler) should automatically prevent you from mixing tabs and spaces.

9 — Control Flow – Repetition

Learning Objectives

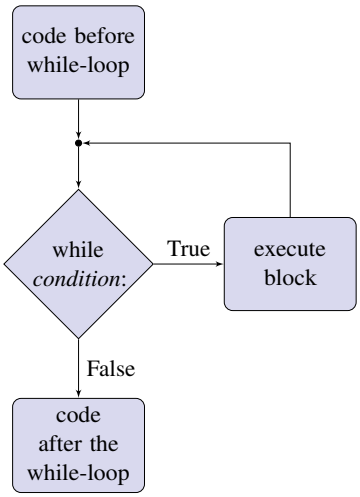
After studying this chapter, a student should be able to:

- identify and correctly author Python language syntax for repetition: while loops and for loops;
- trace by hand the flow of program execution for programs that use while-loops and for-loops;
- design and author Python programs that use one or more loops; and
- describe what is an infinite loop.

Very frequently in computer programming we would like to repeat certain actions. Sometimes we want to repeat these actions a specific number of times. Other times, we want to repeat some actions as long as some specified condition (i.e. Boolean expression) is `True`. Sometimes we'd like to repeat some actions for every element of data in some collection of data elements. In Python, we can do all of these things using *loops*.

9.1 While-Loops

While-loops work a lot like an if-statement in that they have very similar syntax — a condition followed by a block — but the block can be executed multiple times as long as the condition is `True`. While-loops consist of the word `while`, followed by a Boolean expression (the *loop condition*), followed by a colon, followed by a block of code. Below you can see the general form of a while-loop, and the corresponding flow of execution presented as a flowchart.




```
# code before while-loop

while condition:
     # block (indented)

# code after the while-loop
```

When execution of code reaches a while-loop, the loop's condition is evaluated. The condition must be a Boolean expression yielding a result of `True` or `False`. If the condition is `True`, the block of code following the while-loop's condition is repeated until the condition becomes `False`. Then the (unindented) code after the while-loop executes. Note that it is possible that the loop condition is `False` the first time it is encountered. If this is the case, then the block is never executed, and execution proceeds to the code after the while-loop.

A while-loop can help us improve our guessing game from Section 8.3. Previously, we asked the user to input a number between 1 and 100, and reported whether the guess was too high, too low, or valid. But we had no easy mechanism to ask the user for a new guess if their guess was too high or too low. With while-loops, we can repeat the actions of asking for a guess, and checking it for validity until the user enters a guess that is valid!

```
guess = int(input('Guess a number between 1 and 100: '))
while guess < 1 or guess > 100:
     if guess < 1:
          # If guess was less than one, execute this block.
          print('Too low!')
     elif guess > 100:
          # Otherwise, if guess is larger than 100, do this block.
          print('Too high!')

     # ask for a new guess
     guess = int(input('Guess a number between 1 and 100: '))

print('That was a valid guess!')
```

This program will ask the user for a guess, and then, as long as the guess is not valid, the while-loop's condition will be `True`, the reason for the guess being invalid will be printed, the user will be asked for another guess, then the loop condition will be checked again with the new guess, and so on, until the guess becomes valid. Once the guess is valid, the loop condition will be `False`, and the `print` function call after the while-loop's block will print that it was a valid guess.

Note that the block after the while-loop's condition consists of the if-elif statement, and the line that asks for another guess. The if-elif statement, in turn has its own blocks, which are indented relative to the first block. This is an example of *nested blocks*. You can nest blocks to any number

of levels so long as all of the blocks at the same level are indented by exactly the same amount throughout the entire program.

If we are to run our new guessing program, the output will be as follows (green text is text entered by the user):

```
Guess a number between 1 and 100: 125
Too high!
Guess a number between 1 and 100: 0
Too low!
Guess a number between 1 and 100: 42
That was a valid guess!
```

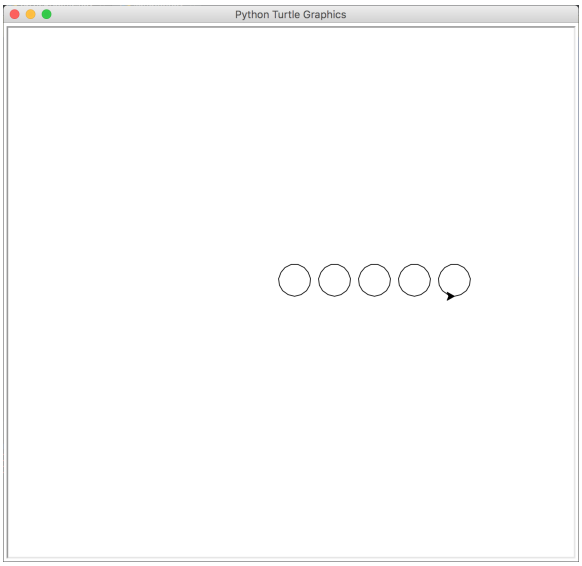
We will show you more examples of while-loops in class.

9.2 While Loops for Counting

While loops can also be used to execute a block of code a pre-determined number of times. These are called *counting loops* because an integer variable is used to count the number of times the block has executed, and the loop condition is such that the condition is `True` as long as the loop has executed fewer than the required number of times. For example, we can use a counting while-loop with the `turtle` graphics module to write a function that draws a row of *n* circles on the screen:

```
import turtle as turtle

def drawCircles(n):
    circlesDrawn = 0              # number of circles drawn so far
    while circlesDrawn < n:       # while we haven't drawn n circles
        turtle.goto(circlesDrawn*50, 0)      # move the turtle

        turtle.down()             # put the turtle's pen down
        turtle.circle(20)         # draw a circle of radius 20 pixels
        turtle.up()               # pick up the turtle's pen

        # add 1 to the number of circles drawn
        circlesDrawn = circlesDrawn + 1
```

The important things to take away from this example are that the variable `circlesDrawn` acts as a counter that keeps track of how many circles we've drawn, and that the while-loop's condition `circlesDrawn < n` causes the while-loop's block to execute until we have drawn exactly `n` circles. The last line of the block where `circlesDrawn` is increased by 1 is very important for this to work. If you leave this line out, you will get what is known as an *infinite loop* (see Section 9.6). If we were to call the `drawCircles` function with an argument of `5`, like this: `drawCircles(5)` then we'd see the following output consisting of five circles in a row:



The general form of a counting while-loop that does something *n* times

```
n = number of times you want to do something
counter = 0
while counter < n
     do the thing
     counter = counter + 1
```

Of course counting while-loops are not limited to those that count from 0 to *n*. You can count from any integer *a* to any other larger integer *b* in a similar manner by changing the initialization of the counter variable so it starts counting at *a*, and adjusting the loop condition so the loop stops repeating when the counter's value is *b*.

9.3 For-Loops

In Python, for-loops allow repetition of a block of code for each data item in a sequence (recall *sequences* from Section 7.1). Right now we know about one kind of sequence: strings. So we can use a for-loop to do something for every character in a string. In this example, we have a function that counts and returns the number of capital letters in a string:

```
def countCaps(s):
    count = 0
    for character in s:
        if character.isupper():
            count = count + 1
    return count
```

The block following the for-loop (consisting of the if-statement and its block) is executed once for each character in the string `s`; each time the block is repeated, the variable `character` refers to the next character in the string.

In general, the syntax of a for-loop consists of the word `for`, followed by a variable name, followed by the word `in`, followed by a sequence, followed by a colon, followed by a block:

```
for variable in sequence:
    # Block of code -- each time this block is repeated,
    # variable refers to the next item in the sequence.
    # Repetition stops after each item in the sequence has
    # been processed.
```

When we do something for each element of a sequence we say that we are *iterating* over the sequence. For-loops can be used to iterate over any sequence, not just strings. In the next section we will introduce another kind of sequence called a *range* which is a sequence of integers. We will learn about even more types of sequences in later chapters.

9.4 Ranges and Counting For-Loops

We can use for-loops to create counting loops just like we did with while-loops. To do so, we first need to learn about a new kind of sequence called a *range*.

A *range* is a sequence of integers that begins at an integer a (the *start*), ends **before** an integer b (the *stop*), and in which the difference between each element in the sequence, called the *step size*, is equal. Ranges are created with Python's built-in `range` function. The `range` function requires two arguments, *start* and *stop*, and can optionally accept a third argument for the step size which, if not given, defaults to 1. You may also provide just a single argument to range; `range(x)` is equivalent to `range(0, x, 1)`, and is the sequence $0, 1, 2, \ldots, x-1$. Here are some example ranges:

```
range(0,5,1)      # the sequence 0, 1, 2, 3, 4
range(5)          # the sequence 0, 1, 2, 3, 4
range(-4, 4)      # the sequence -4, -3, -2, -1, 0, 1, 2, 3
range(0, 11, 2)   # the sequence 0, 2, 4, 6, 8, 10
range(2, -3, -1)  # the sequence 2, 1, 0, -1, -2
range(0, 5, 10)   # the sequence 0

# General form:
range(start, stop, step_size)
```

Remember: the value *stop* is **not** part of the sequence.

Ranges can be used to write counting for-loops. Here is a for-loop that repeats its block exactly `N` times:

```
for i in range(N):
    # do something
```

In this loop, `i` refers to the value 0 on the first repetition, 1 on the second repetition, and so on, up to `N-1` on the last repetition. It is equivalent to the following while-loop:

```
i = 0
while i < N;
    # do something
    i = i + 1
```

9.5 Choosing the Right Kind of Loop

Generally, for-loops are what you want to use to iterate over a sequence. Both for-loops and while-loops are appropriate for simple counting loops. You may prefer using for-loops with ranges for counting purposes because it requires less typing than the equivalent while-loop. For most other non-counting loops that have complicated loop conditions and/or don't involve iterating over sequences, while-loops are likely the best choice.

9.6 Infinite Loops

Infinite loops are loops that repeat forever. A while-loop whose loop condition can never become `False` is an infinite loop. Here are a couple of examples:

```
# This counting loop is infinite because the programmer forgot
# to add the x = x + 1 line to the end of the block.  The value
# of x never changes, so the loop condition is always True.
x = 0
total = 0
while x < 10
    total = total + x
average = total / 10
```

```
# This loop is infinite because the programmer incorrectly used
# 'or' instead of 'and'.  Mathematically, the condition can
# never be False, regardless of the value referred to by x.
# Thus, the loop repeats forever.
x = -1
while x >= 0 or x <= 10:
    x = input("Enter a number that isn't between 0 and 10:")
```

It's quite difficult to accidentally write infinite for-loops because sequences are of finite length and they repeat only once for each item in the sequence.

10 — List and Tuples

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a list is;
- become familiar with the various ways in which we can access and manipulate the items in a Python list;
- describe two uses for list comprehensions;
- identify and author simple list comprehensions in Python;
- describe what a tuple is; and
- describe the similarities and differences between tuples and lists.

10.1 Lists

A *list* is a compound data type consisting of a set of data items arranged in a specific linear ordering. In Python, lists have the following properties:

- lists are sequences, and therefore support indexing and slicing (like strings);
- lists may contain items of different data types;
- lists are *mutable* sequences, meaning they can be altered after they are created (see Section 10.1.1); and
- lists are objects, and contain methods which you can call (just as strings do).

10.1.1 Mutable Sequences

In Python lists are sequences, just like strings and ranges, which means we can use indexing and slicing on them. But lists are the first sequence we have encountered that are *mutable*, which means that you can modify the contents of the sequence. Strings and ranges are *immutable sequences* — once they are created, they cannot be changed.

We will see that we can do things to mutable sequences that we cannot do to immutable sequences. For example, we can add and remove items from a list, but we can't add and remove characters from strings (they are immutable).

Sometimes the difference between mutable and immutable sequences can be confusing. When we do string concatenation using the + operator (remember this from Section 2.3.4?) it kind of looks like we're changing a string. When `a` and `b` are strings, it may seem like we're changing `a` by appending `b` to `a`, but what we are really doing is creating a new (immutable) string that is the result of the concatenation:

```
a = 'Winter is'
b = 'coming'
s = a + b
```

Here `s` is a new string. The strings `a` and `b` are not changed.

But lists are mutable. We can change them without causing a new list to be created, as we will see in subsequent sections.

10.1.2 Creating Lists

One way of creating a list is by writing a list literal. That is, literally writing a comma-separated list of expressions enclosed in a pair of square brackets:

```
x = [2, 3, 5, 7, 11]   # a list of some prime numbers

# a list of video game titles
y = ['Diablo 3', 'Path of Exile', 'Torchlight II']

# a list containing different types of data
z = ['Ultimate answer', 42.0, 6*9]
```

The multiplication of a list and an integer works the same way as multiplication of integers and strings, so we can create a list of n copies of a value by first creating a list that contains only that one value, and then multiplying it by n:

```
n = 10;
# create a list of n zeros:
zeros = [0] * n

# create a list of n empty strings:
empty_strings = [''] * n

print(zeros)          # this prints [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print(empty_strings)  # guess what this will print, then try it.
```

Creating a list of n copies of a value is useful when you want to create a list of initial values and then modify those initial values later in the course of a task. Later in this chapter we shall see how to create lists from other lists using *list comprehensions* (Section 10.3).

There are also many functions in many modules that obtain or generate data in some way and return the data items as a list. For example there are modules that contain functions for reading data from a file and returning that data in a list. We'll look at an example of this in a later chapter.

10.1.3 Accessing List Items (Indexing and Slicing)

We can use indexing and slicing on lists in exactly the same way we did for strings. Here are some examples:

```
>>> x = [2, 3, 5, 7, 11, 13, 17]
>>> y = ['Diablo 3', 'Path of Exile', 'Torchlight II', 'Grim Dawn']
>>> x[2]                  # the third item in x
5
>>> x[3:6]                # the fourth through sixth items of x
[7, 11, 13]
>>> y[-1]                 # the last item in y
'Grim Dawn'
>>> y[0:len(x):2]         # every other item of y
['Diablo 3', 'Torchlight II']
```

10.1.4 Modifying List Items

You can modify the item at a given index of the list by indexing the list, and then assigning a value to it:

```
>>> y = ['Diablo 3', 'Path of Exile', 'Torchlight II', 'Grim Dawn']
>>> y[0] = 'D3: Deluxe Edition'
>>> y
['D3: Deluxe Edition', 'Path of Exile', 'Torchlight II', 'Grim Dawn']
```

Note how the value at offset 0 was changed, and the rest of the list remained the same.

10.1.5 Determining if a List Contains a Specific Item (Membership)

The operators `in` and `not in` are boolean operators that can be used with any sequence, not just lists.

The `in` operator requires that its left operand is an expression, and its right operand is a list. It evaluates to `True` only if the value of the left operand is an item in the list given as the right operand:

```
x = [2, 3, 5, 7, 11, 13, 17]
y = ['Diablo 3', 'Path of Exile', 'Torchlight II', 'Grim Dawn']

8 in x              # False since 8 is not in the list x.
3+4 in x            # True since 7 is in the list x
'Diablo 2' in y     # False, since 'Diablo 2' is not in y.
```

The `not in` operator has the same requirements of its operands as the `in` operator, but evaluates to `True` only if the value of the left operand is **not** in the list given as the right operand:

```
>>> x = [2, 3, 5, 7, 11, 13, 17]
>>> y = ['Diablo 3', 'Path of Exile', 'Torchlight II', 'Grim Dawn']
5+4/2 not in x           # False, since 7 is in x.
'Grim Dawn' not in x     # True, since 'Grim Dawn' is not in x.
'Skyrim' not in y        # True, since 'Skyrim' is not in y.
'Diablo' not in y        # True, since 'Diablo' is not in y.
```

In the last example, you might have been tempted to think that `'Diablo' not in y` was `False`, since one of the list items contains the substring `'Diablo'`. But tests for membership always

compare the entire data item, and 'Diablo' and 'Diablo 3' are not the same strings, so it is indeed true that y does not contain the string 'Diablo'.

You can get the index of an item in a list using the list's index method:

```
>>>y = ['Diablo 3', 'Path of Exile', 'Torchlight II', 'Grim Dawn']
>>>y.index('Torchlight II')
2
```

10.1.6 Adding Items to a List

Lists have a method called append which allows you to add an item to the end of the list:

```
>>> x = [2, 3, 5, 7, 11, 13, 17]
>>> x.append(19)
>>> print(x)
[2, 3, 5, 7, 11, 13, 17, 19]
>>> x.append(23)
>>> print(x)
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

You can append a list of items to another list using a list's extend method:

```
>>> x = [2, 3, 5, 7, 11, 13, 17]
>>> x.extend([19, 23, 27])  # Add 19, 23, and 27 to end of x.
>>> print(x)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
```

x.extend([19,23, 27]) is equivalent to doing:

```
for i in [19, 23, 27]:
    x.append(i)
```

which is not the same as x.append([19,23,27])! Note the important difference between extend and append when you try to append a list:

```
>>> x = [2, 3, 5, 7, 11, 13, 17]
>>> x.append([19, 23, 27])  # append [19,23,27] as a single item of x
>>> print(x)
[2, 3, 5, 7, 11, 13, 17, [19, 23, 27]]
```

The list [19,23,27] is appended as a single item of the list x! That is, the 8-th item in the list x is not an integer, it is another list containing the items 19, 23, and 27! This is an example of a *nested list*. A nested list when you have an entire list as a single item in another list. We'll see more examples of this soon.

10.1.7 Removing Items from a List

The `remove` method of a list deletes a specified item from the list no matter what index it occupies:

```
>>> z = ['Han', 'Chewie', 'Luke', 'Leia', 'C3PO']
>>> z.remove('Luke')  # delete 'Luke' from the list
>>> print(z)
['Han', 'Chewie', 'Leia', 'C3PO']
```

If there are multiple occurrences of the specified item in the list, the occurrence with the smallest index is removed, but the other occurrences remain.

To delete the list item at a specific index (without needing to know what the item at that index is), use the `del` operator:

```
>>> z = ['Han', 'Chewie', 'Luke', 'Leia', 'C3PO']
>>> del z[1]   # delete the second item in z
>>> print(z)
['Han', 'Luke', 'Leia', 'C3PO']
```

The `del` operator can also be used with slicing to delete multiple items from a list at once:

```
>>> z = ['Han', 'Chewie', 'Luke', 'Leia', 'C3PO']
>>> del z[1:4]
>>> print(z)
['Han', 'C3PO']
```

Remember that `del` is an **operator** that takes only one operand, not a function, so round brackets, e.g. `del(z[1])`, should not be used.

10.1.8 Sorting the Items in a List

If the items in a list are all comparable with one another, the list can be sorted using the `sort` method. The `sort` method rearranges the items in the existing list and does not create a new list. Numbers are sorted in numeric order:

```
>>> import math as m
>>> numbers =  [42.0, 7, m.sqrt(12), -17, -42, m.pow(2,16)]
>>> numbers.sort()
>>> print(numbers)
[-42, -17, 3.4641016151377544, 7, 42.0, 65536.0]
```

Strings are sorted in lexicographic order (dictionary order):

```
>>> words = ['what', 'is', 'dead', 'may', 'never', 'die']
>>> words.sort()
>>> print(words)
['dead', 'die', 'is', 'may', 'never', 'what']
```

A list with both numbers and strings cannot be sorted, because strings cannot be compared to numbers; the result is a type error:

```
>>> stuff = [6, 'multiplied', 'by', 9, 'is', 42]
>>> stuff.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

10.1.9 Copying Lists

Recall that the assignment operator, =, associates a variable name (also called an *identifier*) with a piece of data. Suppose we did this:

```
x = 42   % associate the identifier x with the value 42
y = x    % associate the identifier y with the value 42
```

How many copies of the value 42 are there? Only one. We did not create two copies of 42, we simply assigned two different names to the same value. If we later associate x with a different value, it doesn't change the fact that y is still associated with the value 42, so in that sense, y's value didn't change as a result of changing x.

But things are a bit different with mutable sequences. Firstly, if we do this:

```
x = [2, 4, 6, 8, 10]
y = x
```

Then this is no different from the previous example — we have simply assigned two variable names to refer to the same list. But because lists are mutable sequences, this has some subtle consequences. What if we change the third item of x to -10?

```
x[2] = -10
```

Does this change the list referred to by y? The answer is: **yes**, because x and y refer to the same list. If we now print out the value of y[2] the value -10 will be printed, and if we print x and y, we see that they are still, indeed, the same list.

```
>>> print(y[2])
-10
>>> print(x)
[2, 4, -10, 8, 10]
>>> print(y)
[2, 4, -10, 8, 10]
```

If you want an actual **copy** of a list, you have to use its `copy` method. This will produce a second **different** list, that contains the same data items as the original list, but which can be modified without causing the original list to be changed:

```
>>> x = [2, 4, 6, 8, 10]
>>> y = [2, 4, 6, 8, 10]    # y and x refer to the same list
>>> z = y.copy()            # z refers to a copy of list y
>>> z[2] = -10              # change something in list z
>>> print(x)                # change to z does not affect list x
[2, 4, 6, 8, 10]
>>> print(y)                # or list y (x and y are the same list)
[2, 4, 6, 8, 10]
>>> print(z)                # only list z is changed since it was
[2, 4, -10, 8, 10]          # a copy of y.
```

The important thing to remember is that the assignment operator = does not make a copy of data. It only associates a new name with that data. Many mutable compound data objects, including lists, provide methods to create copies of themselves.

10.1.10 Concatenation

We saw, back in Section 2.3.4, that the + operator concatenates two strings. The + operator can actually be used as a concatenation operator with any type of sequence, including lists:

```
>>> a = [1, 3, 5, 7, 9]
>>> b = [2, 4, 6, 8, 10]
>>> c = a + b
>>> print(c)
[1, 3, 5, 7, 9, 2, 4, 6, 8, 10]
```

Earlier we saw that the `extend` method could add the items in one list onto the end of another list. So it would seem that `a + b` does the same thing as `a.extend(b)`. But be careful... they're not the same! The concatenation operator creates a **new** list that is the concatenation of its operands. To put it another way, if `a` and `b` are lists, then `c = a+b` is equivalent to:

```
c = a.copy()
c.extend(b)
```

The `extend` method does not create a new list, it just adds the items in its argument to the existing list.

10.1.11 Other Functions That Operate on Sequences

In this section, we introduce some handy built-in functions that can be used with any mutable or immutable sequence, including lists. Assuming `S` is a sequence:

- `max(S)` returns the largest item in the sequence;
- `min(S)` returns the smallest item in the sequence;
- `sum(S)` returns the sum of the items in the sequence (if the sequence items are numeric); and
- `len(S)` returns the number of items in the sequence.

10.1.12 Iterating Over the Items of a List

We often want to perform some kind of computation for every item in a list (or other sequence). This is called *iterating* over the list. We already know how to do this, with for-loops!

```
L = ['Tony was chased', 'Bruce was angered', 'Steven was scared']

# print each string in the list L
for x in L:
    print(x)
```

This form of loop is excellent if we want to use each data item in the list in some kind of action or computation.

We can also iterate over a list by iterating over its indices. This enables us to modify each item of the list:

```
L = ['Tony was chased', 'Bruce was angered', 'Steven was scared']

# Append the phrase ' by zombies' to each item of L.
for i in range(len(L)):
    L[i] = L[i] + ' by zombies.'
```

After executing the above code, the list L will have been changed to:

```
['Tony was chased by zombies.', 'Bruce was angered by zombies.',
'Steven was scared by zombies.']
```

10.2 Nested Lists

In Section 10.1.6 we encountered the concept of *nested lists*. The idea is that a data item in a list could be another list. Suppose we are designing a video game and want to store data about all the different magic items that a player might find. Further suppose that the data we need to store for each such magic item is its name, its value (in gold pieces, of course!), and the minimum level a character needs to achieve before they can use it. We could represent these three pieces of information for a single magic item as a list. For example, the list:

```
['Sword of Fighting', 1250, 10]
```

stores the data for a magic item with the name "Sword of Fighting", that is worth 1250 gold pieces, and can only be used by characters of level 10 or higher.

Now imagine we want to store data about **all** of the magic items in our game. We could do this using a list of lists, where each item in the list is another list consisting of the magic item's name, value, and minimum level. Here's an example of a list consisting of three magic items:

```
[ ['Sword of Fighting', 1250, 10], ['Scroll of Conjure Milk', 20, 5],
                    ['Yellow Wizard Robe', 100, 3] ]
```

How do we know that this a list of lists? Notice the positioning of the square brackets. There is a set of square brackets enclosing the entire thing that tells us that the whole thing is a list. Within the outer pair of square brackets, we have three more lists enclosed in pairs of square brackets, each separated by a comma. To help you see this, each item in the list has been shown in a different colour. Thus we have a list of three items, each of which is, itself, a list. We can build up fairly complicated organizations of data just by using nested lists. Lists can be nested to any depth desired.

10.3 List Comprehensions

List comprehensions offer a convenient syntax for creating more complex lists. One way we can use list comprehensions is to select some items of a list to put in a new list. This is best illustrated with an example. Suppose we had a list of magic items, like in the previous section, and we wanted to construct a new list consisting of only the magic items whose value is greater than 500 gold pieces. We could use a for-loop, like this:

```
# A list of magic items.

loot = [ ['Sword of Fighting', 1250, 10],
         ['Scroll of Conjure Milk', 20, 5],
         ['Yellow Wizard Robe', 100, 3],
         ['Orcish Rhyming Dictionary', 550, 1] ]

# an empty list for storing pricey items
expensive_loot = []

# for each magic item in the list 'loot', if it has
# a value > 500, put it in the list of expensive loot
for x in loot:
    if x[1] > 500:
        expensive_loot.append(x)
```

But we can do it even more easily with a list comprehension:

```
# A list of magic items.
loot = [ ['Sword of Fighting', 1250, 10],
         ['Scroll of Conjure Milk', 20, 5],
         ['Yellow Wizard Robe', 100, 3],
         ['Orcish Rhyming Dictionary', 550, 1] ]

expensive_loot = [ x for x in loot if x[1] > 500 ]
```

Both of these programs result in `expensive_loot` referring to the following list:

```
[['Sword of Fighting', 1250, 10], ['Orcish Rhyming Dictionary', 550, 1]]
```

The general form for using list comprehensions to select items from a list is:

```
[x for x in sequence if expression]
```

where *`sequence`* is any sequence and *`expression`* is an expression involving `x`. Each item `x` from *`sequence`* is selected and added to the resulting list if the *`expression`* involving `x` evaluates to `True`. The square brackets around the list comprehension provide visual indication that the result of the code is a new list.

Another use of list comprehensions is to apply some kind of computation to each item in a sequence and store the results in a new list. For example, suppose we want to create a list containing the square roots of the integers from 10 to 50. At this point, we hope you could see how to do this with a for-loop.[1] Here's how you would do it with a list comprehension:

[1] You would use a for loop to iterate over the sequence `range(10,51)`, take the square root of each item, and append each square root to the end of a list which is initially empty.

```
import math as m
roots = [m.sqrt(x) for x in range(10, 51)]
```

The general form for computing something for each item in a sequence and putting the results in a new list is:

```
[expression for x in sequence]
```

where *expression* is an expression involving `x` and *sequence* is any sequence. The result is a list containing the value of the expression for each item `x` in the sequence.

List comprehensions are even more versatile than what we have seen here and can be used to compactly code quite complex things. But for CMPT 141, we will be concerned mostly with relatively simple list comprehensions of the forms we have seen here. We'll look at more examples of list comprehensions in class.

10.4 Tuples

Tuples are identical to lists, with the following exceptions:

- Tuples are **immutable**.
- Tuples are written with parentheses instead of square brackets.

Other than that, they are more or less the same as lists. They can contain items of different types, they can be indexed, sliced, concatenated, and can be used with the `in` and `not in` membership operators. However, because tuples are immutable, tuple items cannot be changed, tuples cannot be used with `del`, and do not have `extend`, `append`, `copy` or `sort` methods.

So why would you use tuples instead of lists? Indeed, why would you use an immutable sequence when you could use a mutable sequence since mutable sequences have more features? To find the answer, we have to turn to the discipline of *software engineering*. Software engineering is kind of like the "science of writing good code". One principle of software engineering is: "don't write code that allows data to be modified in ways you know are not permitted". This prevents accidental modification of data in ways it should not be handled. If we have a sequence of data items that we **know** should not be modified after creation, we should use a tuple, because tuples are immutable which prevents modification of the data, either intentionally or accidentally. On the other hand, if we were writing a grocery list application, we'd want to use a list because lists are mutable, and we want people to be able to remove items from the list as they pick things off the shelf in the store.

11 — Dictionaries

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a dictionary is;
- distinguish dictionaries from lists/tuples;
- become familiar with the various ways in which we can access and manipulate data stored in dictionaries; and
- appreciate how lists, tuples, dictionaries may be combined to create more complex data structures.

11.1 Dictionaries

A *dictionary* associates pairs of data items with one another. The first item in such a pair is called a *key* and the second item is called the *value*. A dictionary stores a collection of these key-value pairs. Dictionaries allow you to look values up by their key.

Suppose we had a dictionary called *friends* containing key-value pairs where the keys are people's names, and the value associated with each key is that person's email address. We could then find out someone's email address by querying the dictionary for the value associated with a person's name. If there is a key-value pair in the dictionary `friends` whose key is `'John Smith'`, the value of `friends['John Smith']` would be the email address of John Smith. The keys of a dictionary must be unique — the same key cannot be associated with more than one value. However, the values need not be unique — different keys can be associated with the same value. Thus, there can only be one `'John Smith'` key in `friends`, but another friend with the key `'Jane Smith'` may have the same e-mail address as John Smith.

Over the next few sections, we'll see how to create such a dictionary and look up items in it.

11.1.1 Creating a Dictionary

Dictionaries can be created in a few different ways. They can be literally written out like a list, except dictionaries are enclosed in curly braces rather than square brackets. We can construct an empty dictionary using an empty pair of curly braces:

```
# associate the variable name 'friends' with an empty dictionary
friends = {}
```

We can create a non-empty dictionary by writing a comma-separated listing of key-value pairs within a pair of curly braces. Each key-value must consist of the key, followed by a colon, followed by the value. The following defines a dictionary with four key-item pairs; each pair is a name and an email address:

```
# associate 'friends' with some known key-value pairs
friends = { 'Bilbo Baggins'    : 'burglar1@theshire.net',
            'Sauron the Great' : 'greateye@mordor.gov',
            'Gandalf the White': 'whitewizard@valinor.org',
            'Saruman'          : 'entkiller@isengard.gov' }
```

We can have line breaks and line indentations between key-value pairings within the curly braces because Python ignores whitespace within curly braces (the same applies to square brackets enclosing lists as well!).

Dictionary keys may be any immutable type. Thus, numbers and strings can be dictionary keys. Even tuples can be dictionary keys so long as the tuple itself doesn't contain a mutable data type or a data item that directly or indirectly refers to a mutable type.[1] Lists and dictionaries may not be dictionary keys because they are mutable.

Dictionary values may be any type, including lists, tuples, or even another dictionary.

11.1.2 Looking Up Values by Key

Looking up values by key in a dictionary works very much like indexing a list. You write the variable name that refers to the dictionary, then a pair of square brackets enclosing the key whose value you want to look up.

```
>>> print(friends['Bilbo Baggins'])    # get Bilbo's email address
burgler1@theshire.net
>>> print(friends['Sauron the Great']) # get Sauron's email address
greateye@mordor.gov
```

If you try to look up a key that is not in the dictionary, you get a `KeyError`:

```
>>> print(friends['Tom Bombadil'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Tom Bombadil'
```

Just as well — your friend Tom Bombadil talks a lot and doesn't seem to serve any useful purpose.

[1] Thus `(1,2,'buckle_my_shoe')` could be a dictionary key because none of the items in the tuple are immutable, but `(1,2,[3,4])` could not because the third item of the tuple is a list, and lists are mutable.

11.1.3 Adding and Modifying Key-Value Pairs

You can add a key-value pair, or modify the value associated with a key using the same syntax as a lookup in conjunction with the assignment operator.

```
# add Haldir's email address to the dictionary
friends['Haldir'] = 'smug_elf_531@lothlorien.net'

# update Saruman's email address
# (This is an update since key 'Saruman' is already in
# the dictionary)
friends['Saruman'] = 'bag_end_squatter@theshire.net'
```

Adding and modifying keys look very much the same. If the key already exists in the dictionary, the existing key becomes associated with the new value on the right of the assignment operator. Otherwise, the key is added to the dictionary and becomes associated with the value to the right of the assignment operator.

11.1.4 Removing Key-Value Pairs from a Dictionary

The `del` operator, which we previously used to delete items from lists, can be used to remove a key-value pair from the dictionary.

```
# remove the pair with key 'Saruman' from the dictionary
del friends['Saruman']
```

If you want to remove all of the keys from the dictionary, call the dictionary's `clear()` method:

```
# remove everyone's email address from the dictionary
friends.clear()
```

11.1.5 Checking if a Dictionary has a Key

The `in` operator can be used to determine if a dictionary has a particular key.

```
if 'Sauron the Great' in friends:
    print('Yeah, I am friends with Sauron')
else:
    print('Sauron is not my friend.  I hope his tower collapses.')
```

11.1.6 Iterating over a Dictionary's Keys

You can iterate over all keys in a dictionary, and do something with each key's corresponding value using a for-loop.

```
spam_addresses = []
for k in friends:
    # Add all my friends email addresses to list of spam recipients.
    spam_addresses.append(friends[k])
```

It is important to note that there is no guarantee on the order in which each key of `friends` is processed in such a loop.

11.1.7 Obtaining all of the Keys or Values of a Dictionary

All dictionaries have a `keys()` method which returns a special type of sequence (not a list!) containing all of the keys from all of a dictionary's key-value pairs.

```
>>> friend_names = friends.keys()
>>> print(friend_names)
dict_keys(['Bilbo Baggins', 'Sauron the Great', 'Gandalf the White'])
```

There is no guarantee of the order in which the keys appear in the returned sequence. For example, if someone else were to call `friends.keys()` on their own Python installation, `'Gandalf the White'` may occur prior to `'Bilbo Baggins'` in the resulting sequence!

Similarly all dictionaries have a `values` method that returns a special type of sequence (again, not a list!) containing all of the values from all of a dictionary's key-value pairs.

```
>>> friend_values = friends.values()
>>> print(friend_values)
dict_values(['burgler1@theshire.net', 'greateye@mordor.gov',
             'whitewizard@valinor.org'])
```

Why would we want to use these functions? We'd want to use them to iterate over the keys or values of a dictionary using a for-loop. In order to do this, we'd need a sequence containing all of the keys or values in a dictionary. In this example, we print `friends`' key-value pairs in sorted key-order:

```
friend_names = friends.keys()
for k in sorted(friend_names):
    print(k, ':', friends[k])
```

The output of this code is:

```
Bilbo Baggins : burgler1@theshire.net
Gandalf the White : whitewizard@valinor.org
Sauron the Great : greateye@mordor.gov
```

The `sorted` function is a built-in Python function that can sort both immutable and mutable sequences. It takes a sequence as an argument and returns a **new list**[2] containing the items from its argument in sorted order. This behaviour is slightly different from the `sort` method of a list which **modifies** the existing list so that it is sorted.

11.1.8 Dictionaries vs. Lists

Dictionaries are similar to lists in the following ways:

- both are *containers* that hold a collection of data items;
- both allow storage of data items of different types; and
- both allow you to look up individual data items.

Dictionaries are different from lists in the following ways:

- there is no ordering of the key-value pairs stored in a dictionary, whereas items in a list are in a specific order; and

[2] It's always returns a list, regardless of the type of sequence being sorted.

- values in a dictionary are looked up by their key, whereas items in a list are looked up by their integer index (position in the ordering).

11.1.9 Common Uses of Dictionaries

In this section, we will discuss some common data storage patterns that can be realized with dictionaries.

Dictionaries as Mappings

Dictionaries, by definition associate keys with values. Such an association can be viewed as a mapping that translates one type of data into another. For example, we could use a dictionary to map animal species names to their taxonomical class:

```
species_to_class_mapping = {
     'red squirrel': 'Mammal',
     'komodo dragon': 'Reptile',
     'chimpanzee': 'Mammal',
     'snowy owl': 'Bird',
     'green cheeked conure' : 'Bird',
     'rainbow trout' : 'Fish'
}
```

Now we can use this mapping to look up what basic type of animal a certain species is. When a dictionary is used to store a mapping, the dictionary is viewed as a collection of many individual data items.

Dictionaries as Records

One common use of a Python dictionary is to represent a *record*. A *record* is a group of related named data elements, for example, the spaces that get filled out in a form, such as name, address, phone number, etc. Note that the term *record* is not specific to particular programming language but rather is a name for this data organization paradigm. The main purpose of a record is to store, as a group, several pieces of data that can be accessed by name.

Records are defined by the names of the data items, and the type of the data items. If we were studying the history of pirates, we might want to define a record that has five data items: given name, family name, pirate name, birth year, and death year. Such a record might be used to store and group together all of the data we want to collect about one pirate. An example of such a record might be:

given_name	Edward
family_name	Teach
pirate_name	Blackbeard
birth_year	1680
death_year	1718

Most programming languages support some way of defining and handling records. In Python, records are stored as dictionaries. The names of the data items in a record are a dictionary's keys, and a dictionary's values are the values associated with each data item. The record shown above would be stored in Python as the following dictionary:

```
pirate1 = { 'given_name': 'Edward',
            'family_name': 'Teach',
            'pirate_name': 'Blackbeard',
            'birth_year': 1680,
            'death_year': 1718 }
```

Now we can look up data about a particular pirate by name. Given the above dictionary, we could compute Blackbeard's age when he died:

```
pirate_age = pirate1['death_year'] - pirate1['birth_year']
```

When a dictionary is used as a record, the dictionary is viewed as a single data item with several properties.

Dictionaries as Databases

You can think of a *database* as a mapping that maps keys to records. When a dictionary is used as a database, the keys are often strings or numbers, and the values are records. Consider a database of customer information. The keys of such a database could be the customer's name, and the value for each key would be a record (i.e. another dictionary!) containing all of the information about that customer. This would enable us to obtain all the information about one customer by looking up their name in the dictionary to retrieve their record of information. Here is what such a dictionary might look like:

```
customer_database = {
 'Homer J. Simpson': {'first_name': 'Homer',
                      'last_name': 'Simpson',
                      'initial': 'J',
                      'address': '742 Evergreen Terrace',
                      'city': 'Springfield',
                      'state': 'Unknown',
                      'country': 'USA',
                      'phone_number': '555-555-5555'},
 'Charles M. Burns': {'first_name': 'Charles',
                      'last_name': 'Burns',
                      'initial': 'M',
                      'address': '1000 Mammon Ave.',
                      'city': 'Springfield',
                      'state': 'Unknown',
                      'country': 'USA',
                      'phone_number': '555-000-0001'},
  # More entries ...
}
```

The inner pairs of curly braces tell us that the values associated with each key are dictionaries, each of which have data items named `first_name`, `last_name`, `initial`, `address`, `city`, `state`, `country`, and `phone_number`.

We can obtain the entire record for a given person in the database by looking up their name:

```
burns_record = customer_database['Charles M. Burns']
```

Note that the variable `burns_record` now refers to **another** dictionary, specifically, the dictionary associated with the key `'Charles M. Burns'` (which is a key in the `customer_database` dictionary). Now we can find out more about Mr. Burns by looking up the data items within his record by name:

```
print('Mr. Burns lives at', burns_record['address'])
```

This prints out

```
Mr. Burns lives at 1000 Mammon Ave.
```

We can even access data items in a database record without storing it in an intermediate variable first. The following produces the same result without using the variable `burns_record`:

```
print('Mr. Burns lives at',
      customer_database['Charles M. Burns']['address'])
```

11.2 Combining Lists, Tuples, Dictionaries

We hope, at this point, that you can appreciate how lists, tuples, and dictionaries can be used to build up complex organizations of data. For example, in section 11.1.9 we saw how to make a database by making the values of a dictionary another dictionary. Indeed any of these data types can contain data items/values that are themselves lists, tuples, or dictionaries. As we proceed through this course (and subsequent courses should you continue in computer science), we'll repeatedly encounter this idea of combining data types to organize data in interesting and useful ways.

12 — File I/O

Learning Objectives

After studying this chapter, a student should be able to:

- describe some common ways in which data may be organized in a text file;
- author Python code to open and close files;
- author Python code to read a text file one line at a time;
- apply basic string processing to read numeric data from a text file containing numbers;
- author code to read a line containing multiple data from files using `split`; and
- author Python code to write data to a text file.

Up to this point, the only mechanisms we have used for data **input** into our programs is to either code the data right into our program as literal data (this is sometimes called *hard-coding* the data) or ask the user to enter input from the console. In this chapter, we look at how to obtain input data stored in files. Similarly, the only way we have seen our programs produce **output** is to print to the console. In this chapter we will also look at how to write output to a file.

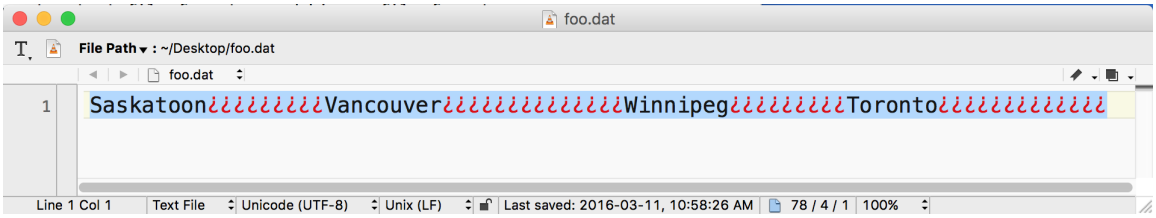
12.1 Data File Formats

The term *file format* refers to the way in which data is organized in a file. There are two main types of file formats: text file formats and binary file formats.

Text file formats are readable by humans. You can open them in any text editor and see the data inside and how it is organized. In text files, numbers are stored as strings of digits. A text file containing data about cities and their average annual high temperatures might look like this:

```
Saskatoon    9
Vancouver    14
Winnipeg     9
Toronto      13
```

Binary file formats are generally not readable by humans because the data is binary-encoded. Such files generally do not contain any meaningful whitespace such as spaces or newlines and appear as gibberish when viewed in a text editor. In a binary file format, numbers are stored in binary (base 2) format, in groups of 8-bits (a byte). A number might be comprised of the bits in one, two, or four consecutive bytes. If we stored the temperature data, above, in a binary file, it might look something like this when we load it into a text editor:



Binary files are typically more compact, use less disk space, and are used frequently in commercial applications and games. Since CMPT 141 is an introductory course, we will not be using any binary file formats, only text file formats.

12.1.1 Common Text File Formats

In this section, we review two typical ways in which we might organize data in a text file.

List Files: One Data item Per Line

A *list file* consists of one data item per line, and usually each line contains the same type of data. List files are very simple to read into a program since each line of the file contains one data item, and most programming languages have built-in functions for reading one line from a file. An example of a list file might be observations of temperature recorded over a single day:

```
-2.7
-1.8
0.3
2.4
3.5
5.9
```

Tabular Files: One Group of Related Data Items Per Line

A *tabular file* format is one where there is a fixed number of data items per line. We can think of such a file as a table, because it will have a certain number of rows (lines) and a certain number of columns (data items per line).

The data items on a line may be different types, but typically each column of data is all of the same type, that is, the n-th piece of data on each line is of the same type. Data items on a line might be separated by spaces, or another character, such as a comma. Whatever character is used to indicate separation of data items on a line of the file is called the file's *delimiter*. It delimits (separates) one data item from the next. Here is an example of a tabular text file, delimited by commas, where each line holds data from an entry in the database from Section 11.1.9:

```
Homer J. Simpson,Homer,Simpson,J,742 Evergreen Terrace,Springfield,Unknown,USA
Charles M. Burns,Charles,Burns,M,1000 Mammon Ave.,Springfield,Unknown,USA
Ned Flanders,Ned,Flanders,,744 Evergreen Terrace,Springfield,Unknown,USA
```

Each line of this file holds the data for one database entry, and contains exactly 8 data items, separated by commas. The first data item on each line is the key for a database entry, and the remaining data items are the data items in the database record associated with the key. Note how the fourth data item of the third database entry is empty since there is nothing between the commas.

If our data items themselves do not contain spaces, we can use whitespace as a delimiter, which makes the text file look more like a table. Here is an example of a tabular datafile that stores weather observations taken every four hours for different weather stations on one specific day of the year where each weather station is identified by a four-digit ID number:

```
1783 22   25   27   28   21   19
2214 -4   2    6    7    6    0
9934 -40  -32  -26  -21  -24  -32
5538 15   17   21   22   23   19
```

The first column contains the weather station ID number, and the remaining columns store temperature observations. Since observations are every four hours, there are six such columns.

Other Formats

Any format you can think of is theoretically possible, but you might have to write custom code that can process unconventional formats.

12.2 File Objects in Python – Open and Closing Files

In Python we interact with files on the disk via an abstraction. We can ask Python to return an object that allows us to interact with a data file on disk. This is called *opening* a file. We can open a file and obtain an object for that file using Python's built-in `open` function. The `open` function returns an object that contains methods that allow us to read and write data to or from a file. Suppose the table of temperature data, above, is stored in a file called `temperatures.txt`. We can open it like this:

```
f = open('temperatures.txt', 'r')
```

The first argument to `open` is a string containing the name of the file to be opened — this can be any valid pathname (see Section 12.5 for more details on pathnames). The second argument string is the *mode*. Here we are using the file mode `'r'`, to indicate that we want to **read** from the file. Later we'll see how to write to files using the `'w'` mode. Now `f` is an object that contains methods that allow us to manipulate the file. This is a nice abstraction because we can work with the file just by calling methods of `f` and we don't have to have any idea how disks and filesystems work. One other interesting thing about file objects is that they behave as sequences, which means we can use them in places where we could use sequences! We'll see how this works in the next few sections.

Before we move on, we must note that once a file is opened, it must be *closed* again when you are done with it. If `f` refers to a file object created with `open`, it is closed by calling the `close` method of `f`:

```
f.close()
```

Once you call `f.close()`, `f` can no longer be used to manipulate the file; trying to do so will result in an error message. If you forget to close a file that was opened in **read** mode, usually nothing bad will happen, although you really should always do it. If you forget to close a file that was opened in

write mode, it is possible that the data you wrote to the file will not actually be written, and that is very bad!

12.3 Reading Text Files

In the previous section, we mentioned that file objects returned by the `open` function behave like sequences. In particular, they behave like sequences of strings, where each string is a line of the file. This means that we can iterate over the lines of a file just like we can iterate over the elements of a list!

12.3.1 Reading List Files

List files are pretty easy to deal with since each line of a file contains a single data item and, as we have already mentioned, we can access each line of a file as a string easily.

Suppose we have a file called `movietitles.txt` which contains one movie title per line. We can read the movie titles from the file and store them in a Python list like this:

```
# Open the file for reading
f = open('movietitles.txt', 'r')

# create an empty list
titles = []

# iterate over each line of the file
for line in f:
    # append the next line (movie title) to the list
    titles.append(line)

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

If `movietitles.txt` contains the following data:

```
The Fellowship of the Ring
The Two Towers
The Return of the King
```

Then the above code will result in `titles` referring to the list:

```
['The Fellowship of the Ring\n', 'The Two Towers\n', 'The Return of the King\n']
```

Hey, wait, that's weird. What are those `\n`'s at the end of each string in the list? Those are *newline* characters; they are invisible characters that mark the end of each line in a text file, and therefore are included in the string that comprises a line of the file. In Python, `\n` represents the newline character. Even though it is represented by two characters, `\` and `n`, it is actually a single character. It is represented this way so that we can see it because normally it is invisible since it is not associated with any symbol.

Usually we don't want newline characters in our strings. We can remove them by calling the string method `rstrip`. If `s` refers to a string, then `s.rstrip()` returns a copy of `s` that has all of

whitespace at the end of the string, including spaces and newlines, removed. Revising our loop in the previous code to this:

```
f = open('movietitles.txt', 'r')
titles = []
# iterate over each line of the file
for line in f:
    # append the next line (movie title) to the list
    titles.append(line.rstrip())
f.close()
```

results in `titles` referring to the list:

```
['The Fellowship of the Ring', 'The Two Towers', 'The Return of the King']
```

Another way to create a list of the strings from the lines in a file is to use the `list` function to convert the sequence of lines from the file object `f` to a list. Then we can use a list comprehension to remove the newlines:

```
f = open('movietitles.txt', 'r')
titles = list(f)
titles = [ t.rstrip() for t in titles ]
f.close()
```

The result of this code is the same as the previous code listing.

What if we have a list file of numbers? This would seem to be a problem if file objects can only return each line as a string because we would want to read in a file of numbers and store them as numbers, not strings. We can use the built-in functions `int` or `float` to convert strings to numbers. For example `int('42')` returns the integer 42, and `float('64.9')` returns the floating point value 62.9. If you use `int` or `float` on a string that doesn't represent a number of the appropriate type, Python will respond with a `ValueError`. We could read the list file containing temperature data at the beginning of Section 12.1.1 and store the data as a list of floats like this:

```
f = open('temperatures.txt', 'r')
temps = []
for line in f:
    temps.append(float(line))
f.close()
```

or equivalently:

```
f = open('temperatures.txt', 'r')
temps = list(f)
temps = [float(t) for t in temps]
f.close()
```

Both programs here would cause `temps` to refer to the list:

```
[-2.7, -1.8, 0.3, 2.4, 3.5, 5.9]
```

12.3.2 Reading Tabular Files

Reading tabular files is almost the same as reading list files. The main difference is that we have to separate the data items on each line. Remember that the data items on each line are separated by a delimiter. String objects have a `split` method which returns a list of strings consisting of the individual strings that occur between a specific delimiter character. For example, the string `'The king in the north.'` can be separated into individual words like this:

```
my_string = 'The king in the north.'
words = my_string.split()
```

This results in `words` referring to the list:

```
['The', 'king', 'in', 'the', 'north.']
```

If we want to split a string based on a delimiter other than whitespace, we just pass the desired delimiter to `split` as an argument. Here's how we can obtain a list of strings from a string delimited by commas:

```
my_string = '42,38,27,99,55'
numbers = my_string.split(',')
```

This results in `numbers` referring to the list

```
['42', '38', '27', '99', '55']
```

They're still strings, but we've already seen how we can use a list comprehension to convert this to a list of integers or floats.

We can obtain the lines of a tabular data file in the same way that we obtained lines for list files, but then we have to use `split` to divide up each line into its individual data items. A common way to store the data from a tabular file in Python is a list in which each data item is another list that contains the data items from one line of the file, i.e. a list of lists. Recall the temperature data in the tabular file we saw in Section 12.1.1:

```
1783 22   25   27   28   21   19
2214 -4   2    6    7    6    0
9934 -40  -32  -26  -21  -24  -32
5538 15   17   21   22   23   19
```

The following code reads this data and stores it as a list of lists of integers:

```
f = open('temptable.txt')
stations = []
for line in f:
    stations.append([int(n) for n in line.split()])
f.close()
```

Observe how we read each line, split it (using whitespace as a delimiter), then convert the resulting list of strings into a list of integers, then append that list to the list `stations`. This causes `stations` to refer to the list:

```
[
  [1783, 22, 25, 27, 28, 21, 19],
  [2214, -4, 2, 6, 7, 6, 0],
  [9934, -40, -32, -26, -21, -24, -32],
  [5538, 15, 17, 21, 22, 23, 19],
]
```

Observe that `stations[i]` refers to the data in the `i`-th line of the file, and `stations[i][j]` refers to the item in the `j`-th column of the `i`-th line of the file. Thus, `stations[1][4]` refers to the file data found at the fifth column of the second line, which is 7.

12.4 Writing Text Files

To write to a file, you have to open it in **write** mode:

```
f = open('file_to_write.txt', 'w')
```

If a file is opened in write mode, and a file of the same name already exists, then the existing file is destroyed, and a new file of the same name replaces it. If the file opened for writing does not exist yet, it is created.

It is possible to write data at the end of an existing file without destroying it. To do so, open the file in **append** mode:

```
f = open('file_to_write.txt', 'a')
```

12.4.1 The `write()` method.

Writing data to text files is very similar to printing to the console. First you have to open a file in **write** or **append** mode. Then, instead of using the `print` function, you use the `write` method of the resulting file object. If the variable `f` refers to a file object, and the file was opened in **write** mode, then the code

```
f.write(string)
```

writes the the string *string* to the file. The `write` method does not write a newline character to the file unless the string given as an argument includes one. Note that this behaviour is different from the `print` function which, by default, always outputs a newline after printing its argument.

12.4.2 Writing List Files

List files can be written by writing each data item followed by a new line. If we have a list of strings, we can write those strings, one per line, to a file called `shoppinglist.txt` like this:

```
ingredients = ['eggs', 'milk', 'flour', 'yeast']
f = open('shoppinglist.txt', 'w')
for i in ingredients:
    f.write(i + '\n')
f.close()
```

This code iterates over each item in the list `ingredients`, and writes it to the file. Note how we concatenate each item in the list with a newline before writing it so that each string appears on its own line. The resulting file looks like this:

```
eggs
milk
flour
yeast
```

If the items we are writing are not strings, we have to convert them to strings because the `write` method can only write strings to files. We can do this using the built-in `str` function which converts its argument to a string, if possible. Here's how we would write a list of integers to a file, one per line:

```
ingredients = [99, 88, 77, 66, 55]
f = open('numbers.txt', 'w')
for i in ingredients:
    f.write(str(i) + '\n')
f.close()
```

Note how the integer `i` is converted to a string prior to concatenating it with a newline.

12.4.3 Writing Tabular Files

To write a tabular file, a typical strategy is to construct a string consisting of one line of the tabular file to be written, and then write it. This is done by combining the data items to appear on that line into a single string, separated by the appropriate delimiter. Just as we had a method, `split`, that could separate a delimited string, we have one that can construct a delimited string from a list of individual data items. String objects have a method called `join`. This method takes a list as an argument and returns a new string that consists of the items in the list separated by the original string. Remember: the string on which we call the `join` is the separator, and the list provided as an argument to `join` contains the data items to combine.

Suppose we have a list of numbers `numbers` which should all appear on one line of a tabular file, separated by commas. We can construct the appropriate string to write to the file like this:

```
numbers = [42, 24, 87, 21, 76]
line = ','.join([str(x) for x in numbers])
print(line)
```

This produces the following output **string**:

```
42,24,87,21,76
```

Look what's happening here. The list comprehension `[str(x) for x in numbers]` converts the list of integers `numbers` into a list of strings. This list is then passed to the join method of the string object ','. This causes the elements of the list to be concatenated, separated by the string `','`. The result is that `line` refers to the **string** `'42,24,87,21,76'` which is then output by the print statement.

Putting all of this together, suppose we had a list of lists. We could write all the data items of each list to a tabular file like this:

```
# a list of lists.  We've seen this temperature data before.
data = [
  [1783, 22, 25, 27, 28, 21, 19],
  [2214, -4, 2, 6, 7, 6, 0],
  [9934, -40, -32, -26, -21, -24, -32],
  [5538, 15, 17, 21, 22, 23, 19],
]
f = open('temperaturedata.txt', 'w')
for station in data:
    f.write(','.join([str(i) for i in station])+'\n')
f.close()
```

For each list of **integers** `station` in `data`, we use a list comprehension to convert the items in `station` to **strings** and put them into a new list, then `join` this list of strings into a single string with a comma as a separator, then add a newline to the end of the resulting string, and write it to the file. This results in a tabular text file that looks like this:

```
1783,22,25,27,28,21,19
2214,-4,2,6,7,6,0
9934,-40,-32,-26,-21,-24,-32
5538,15,17,21,22,23,19
```

12.5 Pathnames

Even if you've never programmed a computer before, but rather, only used one, you probably already know something about pathnames. *Pathnames* are strings that refer to files. When we use the `open` function, we said back in Section 12.2 that we need to pass a *pathname* as an argument to `open` to tell it which file to open. If you want to open a file in the same folder as your Python program, you only need to specify the file's name as a string, like `'temperatures.txt'` or `'reallycooldata.csv'`. If the file exists somewhere else you need to give a full pathname that also specifies the folder that the file resides in. The mechanism for doing this depends on your computer's operating system. On Windows, folder names are separated by a backslash, and the whole pathname might be preceded by a drive letter:

```
'C:\Users\Mark\My Documents\awesomedata.csv'
```

On Mac and Linux, folder names are separated in a pathname by a forward slash:

```
'/home/mark/Documents/awesomedata.csv'
```

These are examples of *absolute paths* because they specify the entire path to the file beginning at the root folder. You can also use *relative paths* which specify the path to a file beginning from the folder that your Python program is in, such as:

```
'../../experiment/data/specialdata.txt'
```

The folder name `'..'` means "parent folder". So the above path means go "up" two folders, then go into the `experiment/data` folder, and find `specialdata.txt` there.

The different folder separator for Windows and Linux/Mac means we have to be a little careful if we want our programs to work on **all** operating systems. Fortunately, Python has a module for that. The `os.path` module has methods for constructing pathnames using the appropriate folder separator for the operating system you are currently running on.

The method `os.path.join()` method can be used to concatenate folder and file names using the correct separator. The variable `os.sep` also refers to the correct separator. Examples:

```
import os
# An absolute path:
filename = os.path.join(os.sep, 'home', 'mark', 'Documents', 'awesomedata')
fid = open(filename)
fid.close()

# a relative path:
filename = os.path.join('..', 'experiment', 'data', 'specialdata.txt')
fid = open(filename)
fid.close()
```

Try this on different operating systems and you'll notices differences in the string referred to by `filename`. Experiment on your own with `os.path.join()` until you're comfortable with how it works.

There are lots of other ways of creating and manipulating paths in the `os.path` module that are outside the scope of the course, but you can learn more about them here if you are interested: https://docs.python.org/3.5/library/os.path.html.

Optional Trivia Challenge

Why do drive letters on Windows operating systems start at 'C' and not 'A'?

13 — Arrays

Learning Objectives

After studying this chapter, a student should be able to:

- explain how data items are organized in one- and two-dimensional arrays;
- describe the similarities and differences between arrays and lists both generally, and in Python;
- create arrays in Python;
- access important array attributes (e.g. size, shape) in Python;
- access desired items in arrays using indexing and slicing;
- perform basic arithmetic and relational operations on each item of an array using arithmetic and relational operators;
- iterate over each item in a one- or two-dimensional array using for-loops; and
- select items from a one- or two-dimensional array using local indexing.

13.1 Arrays

An array is a d-dimensional table of data items. The table of data items can be one-dimensional (like a list), two-dimensional (like a grid), or of higher dimension. Each dimension of the table is indexed by a non-negative integer. A one-dimensional (1D) array is usually visualized as a table with just one row, for example, an array of length 5 containing floating-point data items:

0	1	2	3	4
2.0	3.0	5.0	7.0	11.0

Like lists, we can access a data item in an array by specifying its index (offset). In the picture above, the value 5.0 is the data item at index 2. The indices for each position in the array are shown above

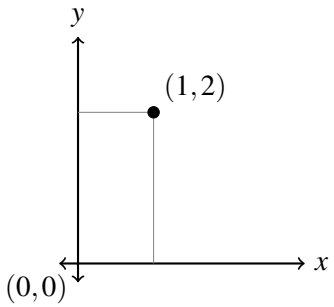
the array.

Two-dimensional (2D) arrays allow us to organize data items in a grid. An item in a 2D array can be accessed by specifying its row index and its column index. A two dimensional array is usually visualized as a table with many rows, like this:

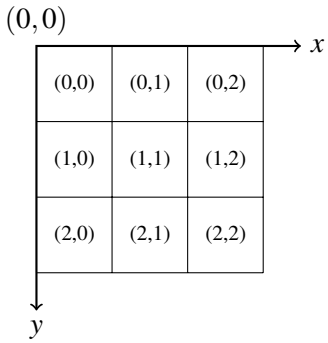
	0	1	2	3	4	5
0	38	30	80	44	82	62
1	28	27	96	62	12	72
2	5	94	10	68	58	40
3	67	23	80	16	17	88

This 2D array of integers could represent grayscale values of pixels in an image where larger numbers indicate brighter pixels. If we accessed the data item at row 2, column 0 of this array, we would find the data item 5. Thus we can think of each data item in an array as having a row-column coordinate (y,x) in a two-dimensional plane, much like we do when we plot data on an x-y axis in math class.

But... there are two important differences here. Firstly, in math, we always write the horizontal or x-coordinate first, then the vertical y-coordinate. When accessing 2D arrays, it is opposite; we write the y- or **row**-coordinate first, then the x- or **column** coordinate. Secondly, in math class when you have axes, the y-axis gets larger as you go up. In 2D arrays, larger y-coordinates access rows further **down**. There are no negative coordinates in a 2D array. Observe the difference:



Cartesian coordinate system.



2D array coordinate system.

13.1.1 Arrays vs. Lists

Recall that lists (in Python) can change in length, and can contain data items of different type. Arrays, on the other hand, cannot do either of these things. Arrays, once created, are of fixed length, and may contain data items of only one type. A list of lists that are all the same length also organizes data in much the same way as a 2D array, but without the restrictions that apply to 2D arrays.

From this perspective, there seems to be no advantage of using arrays over lists; indeed a list of lists would seem to organize data in the same way as a 2D array. But arrays and lists differ significantly in their underlying implementation.[1] Without going into too much detail, the items in a

[1] You'll learn more about this if you stick with computer science and go on to CMPT 145.

list are not necessarily stored near each other in the computer's memory, while arrays always occupy a contiguous single block of the computer's memory. Because of the way a computer's hardware is constructed, this gives arrays a speed advantage when accessing data sequentially. This speed advantage increases when the amount of data to be stored increases. For this reason, arrays are preferred over lists in circumstances where the data doesn't change frequently, the amount of data doesn't change frequently, and fast sequential access to the data is required.

Images are an excellent example of data which is better stored in an array. Images are typically quite large arrays of data, and are very frequently accessed sequentially, one row at a time.

13.2 Arrays in Python – The numpy Module

Standard Python includes a module called `array` that provides one-dimensional arrays. However, almost nobody uses it because it is not very feature-rich, and it only allows one-dimensional arrays. For this reason, almost everyone uses the `numpy` module instead.

The `numpy` module provides *n*-dimensional arrays. This means we can have one-, two-, three-, and, if we wish, higher-dimensional arrays. It is well-documented. You can find the complete `numpy` documentation at: http://docs.scipy.org/doc/numpy/ (click the blue text, this is a live link!). Don't feel you have to read it all, it's huge, and we link it only so that you know it is there in case you want to look up something specific. We may refer you to certain pages of the documentation from time-to-time that describe various functions that we want to use. And if you want to learn more about `numpy` than we will teach in CMPT 141, this is the place to go!

13.3 Programming with numpy Arrays

13.3.1 Creating Arrays

One-dimensional arrays can be created from lists or tuples by passing a list or tuple to `numpy`'s `array` function:

```
>>> import numpy as np
>>> x = np.array([1,2,3,4,5])
>>> print(x)
[1 2 3 4 5]
>>> y = np.array( ('a', 'b', 'c', 'd', 'e') )
>>> print(y)
['a' 'b' 'c' 'd' 'e']
```

Two-dimensional arrays can be created in a similar way by passing a list of lists.

```
>>> import numpy as np
>>> l = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
>>> x = np.array(l)
>>> print(x)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> # print the list l for comparison
>>> print(l)
```

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

Note how printing an array produces output that looks a lot like a list, or list of lists except that the data items are not separated by commas, and 2D arrays are arranged by rows.

`numpy` also has some functions for creating arrays of a certain size where every data item in the array is a specific value. For example, we can create arrays of a specified size containing all zeros with `numpy`'s `zeros` function. For construction of 1D arrays `zeros` requires only one argument — an integer specifying the length of the array to be created. For construction of 2D arrays, `zeros` requires a list as an argument containing two integers that represent the number of rows and columns to create respectively.

```
>>> import numpy as np
>>> # create a 1D array of 10 zeros
>>> a = np.zeros(10)
>>> print(a)
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
>>> # create a 2D array of zeros with 3 rows and 8 columns
>>> b = np.zeros([3,8])
>>> print(b)
[[ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]]
```

Observe how the dots indicate that the values are floating-point. Floating-point is the default data type for arrays in `numpy`. We'll say more about array data types shortly.

There is also a `ones` function in `numpy` that behaves the same way as `zeros` but sets each data item in the array to one instead of zero.

13.3.2 Important Array Attributes

In Python, numpy arrays are objects. We know that objects contain methods. What we haven't seen yet is that objects can also contain variables. A variable inside of an object is called an *attribute* in Python. They are accessed in the same way as methods, using the dot-notation, but without the round brackets for calling a function.

Numpy arrays contain several useful and important attributes:

Attrib. name	Description	Examples

ndim	Number of dimensions	`>>> x = np.ones(5)`
		`>>> print(x.ndim)`
		`1`
		`>>> y = np.zeros([3,5])`
		`>>> print(y.ndim)`
		`2`

size	Total number of data items	`>>> x = np.ones(5)`
		`>>> print(x.size)`
		`5`
		`>>> y = np.zeros([3,5])`
		`>>> print(y.size)`
		`15`

shape	Size of each dimension in a tuple	`>>> x = np.ones(5)`
		`>>> print(x.shape)`
		`(5,)`
		`>>> y = np.zeros([3,5])`
		`>>> print(y.shape)`
		`(3,5)`

dtype	Data type of the array items	`>>> x = np.ones(5)`
		`>>> print(x.dtype)`
		`float64`
		`>>> y = np.array( [ [1,2,3],`
		`[4,5,6] ])`
		`>>> print(y.dtype)`
		`int64`

The `dtype` attribute of an array indicates the type of the data stored in the array. The data types used by numpy are different from the data types used by Python, which can be a bit confusing. The type `float64` is essentially the same as a regular Python floating-point data item; the 64 refers to the fact that these occupy 64 bits of the computer's memory. The type `int64`, however, is different from regular Python integers. Regular Python integers are unlimited in size, but `int64` values must be between -2^{63} and $2^{63}-1$. Sometimes, especially when we work with images, arrays have a `dtype` of `uint8`. These are 8-bit integers that are *unsigned*, meaning they cannot be negative; numbers of this data type must be between 0 and 255. If you are interested, you can find a complete list of the data types that `numpy` arrays can store and the valid ranges of values of each at http://docs.scipy.org/doc/numpy-1.10.1/user/basics.types.html. The default type for integer arrays is either `int32` or `int64` depending on your computer's operating system. The default type for floating-point arrays is `float64`.

Most of time in CMPT 141 it won't be necessary for you to care about the exact data type of arrays because we won't be using numbers large enough for the limits to matter, but it is important to know that every array has a specific data type, and that the data type may limit the size of numbers you can have in the array.

13.3.3 Indexing and Slicing Arrays

One-dimensional arrays can be indexed and sliced in the same way as any other Python sequence. Here are a few examples:

```
>>> import numpy as np
>>> # create array of even numbers less than 50
>>> x = np.array([x for x in range(50) if x % 2 == 0])
>>> # get first 5 even numbers less than 50
>>> print(x[:5])
[0 2 4 6 8]
>>> # get last 5 even numbers less than 50
>>> print(x[-5:])
[40 42 44 46 48]
>>> # get every other even number from the middle
>>> print(x[4:17:2])
[ 8 12 16 20 24 28 32]
```

Two-dimensional arrays are indexed using one index per dimension, separated by a comma. If the ':' is used by itself as an index, it selects all indices for that dimension. This can be used to extract entire rows or columns of an array. Here are some examples:

```
>>> import numpy as np
>>> x = np.array( [ [1, 2, 3], [4,5,6], [7,8,9] ])
>>> # access item in the third column of the second row.
>>> print(x[1,2])
6
>>> # access third row (all columns in third row)
>>> print(x[2,:])
[7 8 9]
>>> # access first column (all rows in the first column)
>>> print(x[:,0])
[1 4 7]
```

Slicing can be used to select parts of a row or column. Suppose that x is the array:

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]
 [50 51 52 53 54 55 56 57 58 59]
 [60 61 62 63 64 65 66 67 68 69]
 [70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99]]
```

Then we can do the following:

```
>>> # access third through fifth rows
>>> print(x[2:5,:])
[[20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
```

```
 [40 41 42 43 44 45 46 47 48 49]]
>>> # access 7th and 8th columns of the 4th through 8th rows.
>>> print(x[3:8, 6:8])
[[36 37]
 [46 47]
 [56 57]
 [66 67]
 [76 77]]
>>> # access the last 3 rows
>>> print(x[-3:, :])
[[70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99]]
```

Note the difference between indexing a 2-D array and a list of lists. The 3rd item in the second row of a 2D array `x` is accessed with `x[1,2]`, but the 3rd item in the second list of a list of lists named `y` is accessed with `y[1][2]`. In the first instance, you are indexing the 2D array with a tuple, in the second instance, you are getting index 1 of a list, the result of which is another list from which we then access the item with index 2.

13.3.4 Arithmetic with Arrays

`numpy` arrays can be used as operands to the standard addition, subtraction, multiplication, division, and exponentiation operators. These operators are performed on each data item in the array. This is something we cannot do with lists. Some examples:

```
>>> import numpy as np
>>> x = np.array(range(5))
>>> y = np.array(range(5,10))
>>> print(x)
[0 1 2 3 4]
>>> print(y)
[5 6 7 8 9]
>>> # sum each item in x with the corresponding item in y.
>>> print(x+y)
[ 5  7  9 11 13]
>>> z = np.array( [ [1, 2, 3], [4,5,6], [7,8,9] ])
>>> # square each data item in the array
>>> z = z**2
>>> print(z)
[[ 1  4  9]
 [16 25 36]
 [49 64 81]]
```

The subtraction (`-`), multiplication (`*`), and division (`/`) operators function similarly with arrays, operating on individual data items in the same array positions of each operand. For this reason, only arrays of the same **shape** can be used with addition, subtraction, multiplication, and division. Thus,

you could subtract two arrays with five rows and five columns each, but you could not subtract an array with two rows and five columns from one with five rows and five columns.

An exception is that you can add/subtract/multiply/divide an array with a single number. This is the same as adding/subtracting/multiplying/dividing each data item in the array to/from/by the single number. For example: `x - 1` subtracts 1 from each data item in the array `x`; `4*x` mulitplies each data item in the array by 4. In each case the result is a new array and `x` remains unchanged. So `y = 4*x` would cause `y` to refer to a new array that contains each item of the array referred to by `x` multiplied by 4, but `x` itself would not be changed.

13.3.5 Relational Operators with Arrays

We can use relational operators on arrays too! For example, if `a` is an array of numbers, the line of code `a > 0` produces a new array of type `bool` (Boolean) which is the same size as `a` and contains `True` wherever `a` was positive, and `False` wherever `a` was not positive.

```
>>> import numpy as np
>>> a = np.array([ [0,-2,5], [-1,-8,-12],[2, 4, -9] ])
>>> z = a > 0
>>> print(a)
[[  0  -2   5]
 [ -1  -8 -12]
 [  2   4  -9]]
>>> print(z)
[[False False  True]
 [False False False]
 [ True  True False]]
>>> print(a.dtype)
int64
>>> print(z.dtype)
bool
>>>
```

Note how the original array's data type is 64-bit integers, but `z`'s data type is `bool` (short for Boolean).

13.3.6 Iterating Over Arrays

Iteration over one-dimensional arrays works very much like any other sequence. Here we print out each item of an array of strings.

```
import numpy as np
houses = np.array(['Baratheon', 'Lannister', 'Stark', 'Greyjoy'])
for actor in houses:
    print(actor)
```

This code produces the output:

```
Baratheon
Lannister
Stark
```

```
Greyjoy
```

Iteration over two-dimensional arrays can be done one row at a time. Then, you can iterate over each row as a one-dimensional array. In this example, we have a nested loop where the outer loop iterates over each row of an array, and the inner loop iterates over each item of a row.

```
import numpy as np
crop_yield = np.array([ [29.2, 36.8, -10.5],
                        [-9.5, 3.9, -5.8],
                        [45.7, 21.2, 8.1] ])
improved = 0
for row in crop_yield:
     for col in row:
          if col > 0.0:
               improved = improved + 1

pct_improved = improved/crop_yield.size*100
print(pct_improved, 'percent of combinations improved the yield.');
```

Suppose the array `crop_yield` represents the improvement in crop yields for different combinations of herbicides and pesticides. Each row is the result from using one kind of herbicide, and each column is the result of using one kind of pesticide. An entry at row `i`, column `j` is the percentage change in crop yield when using the `i`-th herbicide with the `j`-th pesticide. The program computes the percentage of herbicide/pesticide combinations that resulted in an improvement in crop yield.

13.3.7 Logical Indexing

Logical indexing is a way of selecting items that meet certain requirements out of an array. It works on arrays of any dimension. Suppose we have an array `A` of any data type, and an array `B` that is the same size as `A` and has data type `bool`. If we index an array `A` with `B` (i.e. `A[B]`), the result is that the items of `A` for which the corresponding items of `B` are `True` are extracted and stored in a new one-dimensional array (even if the original array `A` is of higher dimension). It is a very powerful feature that we cannot use with lists! We can see how this works in an example:

```
import numpy as np
crop_yield = np.array([ [29.2, 36.8, -10.5],
                        [-9.5, 3.9, -5.8],
                        [45.7, 21.2, 8.1] ])
positive_yields = crop_yield > 0.0
positive_percents = crop_yield[positive_yields]
pct_improved = positive_percents.size / crop_yield.size * 100
print(pct_improved, 'percent of combinations improved the yield.')
```

`positive_yields = crop_yield > 0` creates an array the same size as `crop_yield` that contains `True` wherever the `crop_yield` array was greater than 0.0, and `False` everywhere else. It looks like this if you print it out:

```
>>> print(positive_yields)
[[ True  True False]
 [False  True False]
 [ True  True  True]]
```

Then on the next line we use `positive_yields` to index `crop_yield`. This give us a new one-dimensional array containing only the items in `crop_yield` for which the corresponding items in `positive_yields` were `True`. The array `positive_percents` looks like this:

```
>>> print(positive_percents)
[ 29.2  36.8   3.9  45.7  21.2   8.1]
```

The remaining lines compute and display the percentage of herbicide/pesticide combinations that improved crop yield from the sizes of `positive_percents` and `crop_yield` (which are 6, and 9, respectively). Note that logical indexing allowed us to solve the same problem as in the previous section but without using iteration! While it is important to know how to iterate over arrays, there are many ways (including logical indexing) of using operations on entire arrays to avoid iteration. Doing so is usually a little faster and results in less code compared to using iteration, but otherwise there is nothing wrong with iteration over arrays using loops.

13.3.8 Copying Arrays

Like lists, arrays are mutable. When you modify an array, all variables that refer to that array reflect the change. If you have an array `A` and then write `B=A`, then any changes you make to `A` are also made to `B` because `A` and `B` refer to the same array. If you really want `B` to be a distinct copy of `A`, you need to use the array's `copy` method:

```
import numpy as np
A = np.array([1,2,3,4,5])
B = A.copy();
```

13.3.9 Passing Arrays to Functions

You can pass arrays to functions just as you can any other data. But since arrays are mutable, any changes made to the array by the function are also reflected outside of the function. We observed the same behaviour when we passed lists to functions because lists are also mutable. Thus, if you pass an array `A` to a function `f` which has a parameter called `B` and the function `f` modifies `B`, the changes will also be made to `A` because `A` and the parameter `B` refer to the same object.

Part II

Topics in Computer Science

14 — Recursion

Learning Objectives

After studying this chapter, a student should be able to:

- explain the difference between a recursive and a non-recursive function;
- explain the purpose of the base and recursive cases of a recursive function;
- identify the base and recursive cases of a recursive function;
- understand how recursive formulations for algorithms are derived; and
- explain recursion in terms of the *delegation* metaphor.

14.1 Introduction

Recursion is a form of repetition that uses function calls instead of loops. A *recursive function* is a function which contains one or more calls to itself. This allows for repetition of the instructions in the recursive function. One should not find this type of *self-reference* disconcerting. It's not much different from a self-referencing definition like "the sum of n numbers is equal to the sum of the first $n-1$ numbers added to the last number", which defines a sum in terms of another sum. In any case, it is this notion of *self-reference* that makes a function recursive. Functions that do not call themselves are *non-recursive*.

A recursive function solves one and only one problem, but can solve most or all *instances* of that problem. "Add up the first N positive integers" is a problem; doing so for a specific value of N is an *instance* of that problem.

When given an instance of a problem, a recursive function typically solves a slightly smaller or easier instance of the problem by calling itself and providing the slightly smaller problem instance as input, then uses the solution to the slightly smaller problem instance to, usually quite trivially, solve the original problem instance. Thus, a recursive function has to be able to solve any size instance of a particular problem.

Many problems lend themselves to recursive solutions. For example, consider the problem of determining how many direct ancestors you have at the n-th generation. You are the 0-th generation, your parents are the 1st generation, your grandparents are the 2nd generation, your great-grandparents the 3rd, and so on. So how many ancestors do you have at the n-th generation? If you knew how many ancestors you had at the $n-1$-th generation then you could figure out the answer easily, because each of the ancestors at the $n-1$-th generation has two ancestors at the n-th generation. So if you know that you have k ancestors at the $n-1$-th generation, then you must have $2k$ ancestors at the n-th generation. Thus, we could characterize the solution to the problem like this:

$$ancestors(n) = \begin{cases} 1 & \text{if n = 0;} \\ ancestors(n-1) * 2 & otherwise. \end{cases}$$

We would read this as follows: "The number of ancestors at the n-th generation ($ancestors(n)$) is equal to 1 if n is zero, otherwise, it's twice the number of ancestors in the previous generation ($ancestors(n-1)$). So we could write the following function for calculating the number of ancestors at the n-th generation:

```
def ancestors(n):
    # we want to determine number of ancestors at the n-th
    # generation.  If n is 0, we know the answer immediately.
    if n == 0:
        return 1
    else:
        # otherwise, we determine how many ancestors there are at
        # generation n-1.
        # (solve a slightly smaller instance of the problem!)

        k = ancestors(n-1)

        # Now we know how many ancestors there are at generation
        # n-1, if we double that, we have the correct number of
        # ancestors at generation n.
        return 2 * k
```

Notice how this algorithm *delegates* the problem of determining how many ancestors there are at generation $n-1$ to a function call. The fact that we are calling the same function should not alarm you. We just assume that the function call does what it is supposed to, solves the problem of how many ancestors are at generation $n-1$, and returns the right answer. That function call, in turn, will solve the problem of how many ancestors are at generation $n-1$ by first solving the problem of how many ancestors are at generation $n-2$ (by making another recursive call!), then doubling that answer, and so on. So a whole sequence of n recursive calls will be made, each trying to solve a smaller version of the problem, but this has to stop at some point. When the problem instance is the smallest possible, we can just return the solution immediately without delegating the computation of part of the solution to another function call. Thus, when n is zero, we can immediately return 1, because there is no smaller version of the problem.

Anything that can be done with a loop can also be done with recursion, and vice versa. The following function computes the solution to the same number of ancestors problem, but without using recursion.

```
def ancestors(n):
    k = 1
    i = 0
    while i < n:
        k = k * 2
        i = i + 1
    return k
```

The variable k is initalized to 1 and then k is multiplied by 2 exactly n times. You should convince yourself that, ultimately, the recursive solution does the same thing! It just does it with function calls instead of a loop.

You may wonder why we bother with recursion, since we already know loops. The answer is that a loop is a special case of recursion, and an introductory course in computer science is incomplete without taking a look at recursion. As well, some algorithms can be written far more elegantly and with much less code using recursion, and are far more difficult to write using a loop. As we learn the basics of recursion, we won't see many of these more difficult cases. We hope you'll take our word for it that they exist.

14.2 Recursion Terminology

All recursive functions have the same general structure. At first glance, they are functions containing a conditional (if-else statement, or a variation). At least one of the branches of the if-statement gives a simple answer to a very simple task. This is called the *base case*. The base case is the smallest or simplest possible instance of the problem. The other branch of the if-statement solves the problem by transforming it into one or more sub-problems, and then combines the solutions of the subproblem(s) to form the result of the main problem. This is called the *recursive case*. The recursive case always makes a recursive function call (usually just one, but sometimes more) to the function being defined.

Here is another example. Here, we are adding up all the squares of integers from 0 to N, that is, it calculates $0^2 + 1^2 + 2^2 + 3^2 + \cdots + N^2$.

```
def sum_squares(N):
    if N <= 0:
        # base case
        return 0
    else:
        # recursive case
        return (N*N) + sum_squares(N-1)
```

The base case occurs when $N = 0$; this is the simplest instance of the problem. When $N = 0$, the sum of squares of integers from 0 to 0 is equal to 0, so if N is zero or less, the function immediately returns 0.[1] The base case for a recursive function is almost always so simple that it requires very little problem solving. Usually, common sense tells us the answer without much thought. When writing your own recursive functions, you might be concerned because the base case seems too easy.

[1] In our base case, we also included $N < 0$, which is a practical design, in case someone calls the function with a negative integer. It gives the wrong answer for negative input, but that's OK, because there is no correct answer if N is negative—negative N doesn't define a valid problem instance.

Don't be. It is supposed to be extremely easy. The recursive case of our function occurs when N is larger than 0. The recursive case is found in the `else` block of the if-statement. It tells us that we can calculate the required sum by first solving the sub-problem of finding the sum of squares from 0 to $N-1$ using a recursive call. We can then turn that into a solution for the sum of the first N squares by adding N^2 to the solution to the sub-problem.

The simplest recursive functions for easier problems usually consist of one base case and one recursive case. For more complicated problems, there may be more than one base case, or more than one recursive case. You really need to understand a problem very well to decide if you need multiple base cases or recursive cases to solve it. The number of these cases is determined by the problem you are solving, not the recursive function you are writing.

14.3 More Examples

Simple Sum

Consider the task of adding up all the integers from 1 to N, where N is any positive number. In other words, we want to calculate $1+2+3+\cdots+N$. This could easily be done with a loop, but let's derive a recursive solution. The key idea for this task is to recognize that it does not matter which order you add the numbers. One way to transform the task is to group all but the last number into a sum, and then add the last N to it:

$$1+2+3+\cdots+N = (1+2+3+\cdots+(N-1))+N$$

In other words, we have broken the task of adding numbers from 1 to N into 2 steps: first, add the numbers 1 to $(N-1)$; second, add N to the result of the first step. Clearly, adding up the numbers from 1 to $(N-1)$ is a smaller version of the **same** task of adding the number from 1 to N. If we had some way to do that calculation, all we'd have to do is add N to the result. Fortunately, the function we are writing is just such a function!

Let's use the notation $sum(N)$ to represent the sum of integers from 1 to N. We have to be careful to use this notation only for $N > 0$, because otherwise it doesn't make sense. With this notation, we can also say $sum(N-1)$ is the sum of integers from 1 to $N-1$ (as long as $N-1 > 0$). By the property we observed above, we can say that $sum(N) = sum(N-1)+N$, as long as $N-1 > 0$. From here it is a short exercise to write a recursive function in Python:

```
def sum(N):
    if N == 1:
        # base case
        return 1
    else:
        # recursive case
        return sum(N-1) + N
```

The base case comes from the knowledge that $sum(1) = 1$; the recursive case comes from the equation $sum(N) = sum(N-1)+N$.

Notice that we were primarily engaged in understanding the task (summing a bunch of integers), and we used a bit of basic math to describe the properties of the task. When we finished, we translated the math into Python. The recursive function, therefore, describes a mathematical truth about the task. One only has to understand the language of Python and the nature of addition to see that the program is correct.

Even or Not?

Here's a slightly different example. We can write a recursive function to determine whether a given positive integer is even or not. There are better ways to do this task, but it provides an interesting example for us to discuss.

Suppose we are given a positive integer X, and suppose for the sake of example that $X > 2$ (we ignore negative numbers). There is a property of even numbers that is extremely useful: if X is an even number, then so is $X - 2$; likewise, if X is not even, then $X - 2$ is also not even. In other words, we have identified a relationship between the numbers X and $X - 2$: they are either both even, or both odd.

We can make this relationship work for us in the form of a recursive function. Our function will return the boolean value true if a given X is even, and false if X is odd. We also know that two is an even number, but one is not even:

```
def is_even(X):
    # first base case
    if X == 1:
        return False
    # second base case
    elif X == 2:
        return True
    # recursive case
    else:
        return is_even(X-2)
```

This example has two base cases and one recursive case. Notice that `X` is the input to the function, and that the recursive step transforms the task into a subtask about the value `X-2`. The recursive case decides whether `X-2` is even or not, and there is no combination here because the answer for `X-2` is the same as the answer for `X`.

Again, we motivated the function by explaining a relationship between X and $X - 2$. The function describes this relationship in Python. It is a matter of understanding something about numbers, and a little Python to see that the program is correct.

100 Bottles of Beer on the Wall

Consider the task of singing about some number of bottles of beer that happen to be on the wall:

```
def drinking_song(N):
    # base case
    if N <= 0:
        print("All gone!")
    # recursive case
    else:
        # display one verse on the console
        print(N, 'bottles of beer on the wall,')
        print(N, 'bottles of beer.')
        print('Take one down, pass it around,')
        print(N-1, 'bottles of beer on the wall!')
        print()

        #sing the rest of the song
        drinking_song(N-1)
```

The base case when there are zero or fewer bottles of beer on the wall (`N <= 0`) is very simple: there is no beer left, so we simply display `All gone!` and do nothing more.

In the recursive case we do several things, in sequence. First, display one verse about the current number of beers on the wall; this is done with some `print` statements. Then decrease the number of beers on the wall by one, and recursively call `drinking_song` to display the rest of the song, starting with number of bottles that still remain. The recursive call performs a simpler task, in the sense that it displays the verses for a smaller number of beers.

14.4 How to Design a Recursive Function

There are two steps to designing a recursive function.

1. Determine the base case(s):
   - Determine the smallest instance(s) of the problem.
   - Write one or more base cases to return solutions to those specific instances. These take the form of if-statements, checking whether the problem instance is a base case and, if it is, returning the appropriate answer.
2. Write the recursive case – this usually appears in an else-block following the base cases. Establish the mathematical relationship between the main task and the smaller sub-task(s) before you start coding.
   - Transform the problem instance that you start with (we will call it the "main task" of the function) into a smaller or simpler instance of the same problem (we will call it the "sub-task" implied by the transformation).
   - Obtain a solution to the subtask using a recursive call.
   - Combine the solution of the "sub-task" with some information you have about the "main task" to create a solution for the "main task."

We will demonstrate these ideas using the drinking song example. First we need to identify the base case, which is when there are zero beers remaining on the wall, in which case, thankfully, no more verses need to be sung. In our program we test whether `N <= 0` and if true, print "All Done". This is the only base case.

To write the recursive case, we need to identify the main task. The "main task" is to display the verses of a song; the number of verses depends on the integer N that is input to the function `drinking_song()`. We can transform the "main task" into a simpler one by "drinking" one of the beers on the wall. This transformation gives us a "sub-task" in which we have to display $N-1$ verses, because now there are only $N-1$ bottles of beer.[2] While it may seem that $N-1$ verses is not a lot simpler than N verses, it is a step in the right direction, and it's really all we need. We can make a recursive call to perform the "sub-task" and we can assume this is done correctly.[3] If we can assume that the $N-1$ verses will be correctly displayed, then we can solve the "main task" by displaying exactly one verse **before** we display the $N-1$ verses. We are combining the printing out of one verse with the solution to the "sub-task" (the printing out of the remaining verses) to produce a solution to the "main task".

You can always get a start on defining a recursive program by typing the following template:

```
def <function name>( <parameters> ):
   if <base case test>:
      <return or perform solution to base case>
   else:
     <combine recursive call with something about
      the data to return or perform solution to ‘‘main task’’>
 }
```

Then you fill in the blanks. Not all at once, and maybe you will make some revisions as you go.

14.5 The Delegation Metaphor

It is helpful to use a metaphor to remove confusion. Think of a function call as *delegation*. That is, no matter what function is involved, a function call is like calling in an assistant, giving him a task, and some room to work, and waiting for him to come back with a result (or the task complete).

With this metaphor, it doesn't matter if you give your assistant the instructions for the same function you are working on, or if you give your assistant instructions for a different function. It's all the same. The assistant takes only the data you give him, and works completely independently. He may create variables with the same name as the ones you created, but they are different variables. He may also call an assistant to help with a subtask.

14.6 Common Pitfalls

Some believe that recursive functions are easier to write than repetition using loops. The reason is that every part of a recursive function describes a property of the task being defined. If you understand the task well enough, you can inspect the recursive function and verify each part independently: the

[2] This is not a suggestion that every task can be made simpler by drinking a bottle of beer. But the idea has been tried on many occasions.

[3] The word "assume" is not used here in the sense that we don't know, or can't prove something. We use "assume" to mean that we will get around to making sure it is true, after we finish what we are currently doing.

base case test, the base case task, and the recursive case task. Correctness comes from the task, not from Python.

14.6.1 Confusion About Self-Reference

The most common pitfall is to focus on recursion as something confusing. Some books actually want you to see recursion as mystical or mysterious. Don't fall for that sophistry. Focus on the delegation metaphor. Think about how you can represent two similar tasks, like $sum(N)$ and $sum(N-1)$, with the same sort of notation. This seems like self-reference, defining something in terms of itself. But this particular kind of self-reference is not problematic. A recursive call is just saying "To do this task, use a copy of the instructions you are reading now to solve a slightly smaller version of the problem."

14.6.2 Infinite Recursion

Infinite recursion is like an infinite loop: the function keeps calling itself without ever reaching the base case. Sometimes, it's because the test to identify the base case is incorrect. Sometimes, it's because the recursive call was incorrectly written to solve the main task, not a smaller or simpler sub-task. Sometimes, especially in examples like `is_even()`, the subtasks might not be correct.

14.6.3 Incorrect Answers

Suppose your recursive function stops, and gives an answer, but it's the wrong answer. The most obvious place to look is the combination of the subtask with the information from the main task. However, in this case, any of the components could be incorrect.

15 — Testing and Debugging

Learning Objectives

After studying this chapter, a student should be able to:

- explain the difference between testing and debugging;
- explain what a *fault* is;
- explain what a test case is and list its components;
- generate test cases using the *black-box method*;
- generate test cases using the *white-box method*;
- write *test drivers* that implement test cases and report faults; and
- identify three methods for *debugging*.

15.1 What are Testing and Debugging?

Testing and debugging are processes to ensure that a program is correct in the sense that it behaves as expected, producing the correct outputs for given inputs. If a program does not behave as expected, this is called a *fault* (also called an *error* or *bug*).

Testing is a **proactive** process where one specifically chooses inputs or designs usage scenarios meant to determine if the program behaves correctly under those conditions. For example, if we have a program that takes a floating-point value r as input and is expected to output the area of a circle of radius r, then we might choose to test it by inputting a value of 4 and checking whether the output is correct (it should be about 50.27). Or we might input a negative value for r to test whether the program behaves reasonably when we give it unreasonable input. In other words, we use testing to **detect** faults, hopefully before the software is released and faults are detected by customers in the course of normal usage!

Debugging is a **reactive** process where, having detected a fault, we attempt to determine the reason for the fault and **repair** the fault. For example, in our program that computes areas of circles,

the expected output if a negative number is entered would be an error message indicating that the input radius needs to be positive. But if we input -4 and get an answer of -50.27 instead then our testing detected a fault — the program didn't respond to the invalid input as expected! We now have to determine why the fault occurred, and repair it. Sometimes this can just be done by looking at the program and noticing where we made a mistake. All too often, however, the reason for a fault occurring in a program is not obvious. The larger and more complex a program is, the less obvious it becomes what the cause of a fault is likely to be. We will look at various *debugging* techniques that can help us find and repair faults.

15.2 Testing

The goal of the testing process is to detect all of the faults in some code. To achieve this goal, we start by coming up with a set of *test cases*. A test case consists of a specific input to the code, or a usage scenario performed under specific conditions. When we test code, we want to generate a set of test cases that is:

1. as small as possible; and
2. has a very high likelihood of uncovering every fault that might exist in the code.

It is worth noting at this point that it is rare to test an entire program all at once with a single set of test cases. More frequently, we generate test cases for an individual function we have written to make sure it is correct before moving on and writing other code that uses that function. This is because trying to test an entire program all at once is quite unmanageable (the set of test cases becomes much too large) for all but the smallest programs. If we test a program one function at a time, we can assume that previously written functions are correct when testing our most recently written function, which speeds test case generation. That said, the process of testing is essentially the same whether we are testing just one small function of a larger program, or an entire program.

15.2.1 Standard Form of Test Cases

A test case is defined by determining the following items:

1. the input(s) for the test case;
2. the expected output(s) for the given input(s); and
3. the reason for the test case

When writing test cases, we will use the following standard form:

Input(s):	Description of program or function required inputs.
Output(s):	Description of expected program or function outputs.
Reason:	Description of reason for test case.

This still leaves the question of how to actually identify test cases for a program or function. There are two approaches we can use to generate test cases: *white-box testing*, and *black-box testing*.

15.2.2 Test Case Generation: Black-Box Testing

If we generate test cases for an algorithm/program/function using knowledge of only the expected behaviour of the program or function, and without knowledge of the actual code, this is known as *black-box testing*. The name is a metaphor: you imagine the code is in an opaque black box. You

may test it by feeding input into the box, and receiving output from the box and checking whether it is correct, but you cannot see the code inside the box. Test cases are generated by considering the different inputs that might be provided to the code including common, rare, unusual, and erroneous inputs.

Let's consider the following function:

```
def is_divisible_by_7(numbers):
    """
    This function returns true if the list numbers
    contains a number that is divisible by 7,
    and returns false otherwise.
    numbers: list of numbers to check
    return: if list contains a number divisible by 7
    """
    for i in numbers:
        if i % 7 == 0:
            return True

    return False
```

As we have said, in black-box testing we are not supposed to look at the code for the algorithm when generating test cases. So we must generate test cases using only the following knowledge:

```
def is_divisible_by_7(numbers)
    """
    This function returns true if the list numbers
    contains a number that is divisible by 7,
    and returns false otherwise.
    numbers: list of numbers to check
    return: if list contains a number divisible by 7
    """
```

We now write test cases by considering typical, rare, and erroneous inputs and their expected outputs. Below is a list of test cases we might come up with. Though it is not normally necessary, we have tagged each test case with the label "common", "rare", or "erroneous" so you can better understand our thinking.

Input(s):	[1,2,7,3,5]
Output(s):	`True`
Reason:	Test when there is one element divisible by 7 in the middle of a list with many elements. [Common]

Input(s):	[1,2,4,3,7]
Output(s):	`True`
Reason:	Test when there is one element divisible by 7 at the end of a list with many elements. [Rare]

Input(s):	[14,2,4,3,6]
Output(s):	`True`
Reason:	Test when there is one element divisible by 7 at the beginning of a list with many elements. [Rare]

Input(s):	[13,2,4,3,6]
Output(s):	`False`
Reason:	Test when there is no element divisible by 7 in a list with many elements. [Common]

Input(s):	[7]
Output(s):	`True`
Reason:	Test when the list has only one element that is divisible by 7. [Rare]

Input(s):	[9]
Output(s):	`False`
Reason:	Test when the list has only one element that is not divisible by 7. [Rare]

Input(s):	[]
Output(s):	`False`
Reason:	Test when the list is empty [Rare]

Input(s):	[2, 4, 'seventeen', 14]
Output(s):	`False`
Reason:	Test when the list contains something that is not a number. [Erroneous]

Notice that none of the test cases rely on knowing the implementation of the function, only its header, and its docstring description.

We could probably come up with more test cases, but we'll stop here. Writing test cases requires effort, and that effort has diminishing returns. That is, after a certain point, more test cases are less and less likely to uncover new faults. It is often more of a priority to make sure one tests all of the rare cases than to exhaustively test the more common cases since the rare cases are more likely to require special cases in the code, and special cases in the code are more likely to harbour faults. The goal of test case generation is not to make enough of them to **guarantee** that the software is bug-free, but to do just enough testing that the probability of a bug remaining is extremely low. Believe it or not, it requires vastly much more work to provide a guarantee or proof of correctness than it does to provide a very low probability!

15.2.3 Test Case Generation: White-Box Testing

When we generate test cases for the code we are testing, we call this *white-box testing*. It is a metaphor similar to black-box testing, but in this case we imagine that the code is in a transparent box, and we can see everything inside.

In white-box testing we examine the code and try to think about all the different paths of execution through the code such as true and false branches of an if-statement, or loops that could execute different numbers of times. Then we identify test cases that cause the execution of each of

those paths at least once. If our code contains an if-statement, then we should write at least one test case that causes the if-statement's condition to be true, and one test case that causes it to be false. If we have a loop in our code, we should write a test case that causes the loop to execute zero times, another that causes it to execute one time, one that causes it to execute many times, and one that causes it to execute the maximum number of times (if applicable).

Let's write some test cases for the `is_divisible_by_7` function from the previous section. Here's the code again, followed by the test cases:

```
def is_divisible_by_7(numbers):
    """
    This function returns true if the list numbers
    contains a number that is divisible by 7,
    and returns false otherwise.
    numbers: list of numbers to check
    return: if list contains a number divisible by 7
    """
    for i in numbers:
        if i % 7 == 0:
            return True

    return False
```

Input(s):	[]
Output(s):	False
Reason:	Cause the for-loop to be executed 0 times.

Input(s):	[7]
Output(s):	True
Reason:	Cause the for-loop to be executed one time; cause the if-statement to be true.

Input(s):	[1,2,7,3,5]
Output(s):	True
Reason:	Cause the for-loop to be executed many times; cause the if-statement to be true and false (on different loop iterations).

Input(s):	[1,2,4,3,7]
Output(s):	True
Reason:	Cause the for-loop to be executed the maximum number of times (for the given input); cause the if-statement to be true and false (on different loop iterations).

Notice how some test cases cover multiple testing criteria (in this case, the number of loop iterations and whether the if-statement condition is true or false)! This is completely fine, and is encouraged, because it amounts to less work.

Also notice that all of the test cases we identified using the white-box method were also identified using the black-box method in terms of just the inputs and expected outputs. It is well-known that white-box and black-box testing are complementary methods. One will often identify the same test

cases using either method, however, sometimes one method will helps us discover good tests that the other method does not. There is no one right test case generation method to use; we can use one, or the other, or both. In any case, the goal is to generate a good set of tests that is highly likely to find all of the faults that might be hiding in the code.

15.2.4 Implementing Tests

Once we have a suitable set of test cases, we have to write (i.e. *implement*) code that actually runs the tests so that we can see if the tests detect any faults. This takes the form of a program that contains the implementation of the test cases. Such a program is called a *test driver*.

Recalling that each test case lists an input and an expected output, we implement each test case by invoking the code to be tested with the listed input, and checking whether the received output is the expected output. A fault is detected when a test case does not produce its expected output. If a test case implementation detects a fault, the fault must be reported by printing a message to the console. If a test case implementation does not detect a fault, it should output nothing. In this way, **a test driver only reports on failed test cases.** Results of successful test cases are not reported — this makes it easy to see if any faults were detected after running the test driver. If you see output, there's a problem!

Example Test Case Implementations

Implementation of test cases proceeds in the same way regardless of which method (white-box or black-box) was used to identify the test cases. Here we will demonstrate the implementations of some test cases that we identified with white-box testing in Section 15.2.3 for the `is_divisible_by_7` function. Each test case is shown, and then followed by its implementation.

Input(s):	[]
Output(s):	False
Reason:	Cause the for-loop to be executed 0 times.

```
# call with empty list argument
result = is_divisible_by_7([])
# expected output: False
if result == True:
    print('Error: returned True when given empty list.')
    print('(no items divisible by 7)')
```

Input(s):	[7]
Output(s):	True
Reason:	Cause the for-loop to be executed one time; cause the if-statement to be true.

```
# call with single-item list containing one element divisible by 7
result = is_divisible_by_7([7])
# expected output: True
if result == False:
    print('Error: returned False when given [7] (divisible by 7)')
```

Input(s):	[1,2,7,3,5]
Output(s):	True
Reason:	Cause the for-loop to be executed many times; cause the if-statement to be true and false (on different loop iterations).

```
# call with many-item list containing one element divisible by 7
result = is_divisible_by_7([1,2,7,3,5])
# expected output: True
if result == False:
    print('Error: returned False when given [1,2,7,3,5]')
    print('(3rd item divisible by 7)')
```

Notice that when faults are reported, we always indicate what the fault was and why it was wrong. Also notice that nothing is reported when faults are not detected. The only exception is that we might print a "test complete" message when the test driver is concluded so that we can be certain that the test driver ran to completion and reported no faults.

15.3 Debugging

Once we have identified a fault, we have to correct it. We shall briefly discuss three *debugging* strategies here. These strategies can be used regardless of how the fault was detected, whether it was through the normal course of use of the program, or through a more formal testing process like we have described in the previous section.

15.3.1 Debugging by Inspection

Sometimes the cause of a fault is obvious once the existence of the fault is detected. In such cases, once can just inspect the code and make the necessary adjustments to repair the fault.

Sometimes if it is obvious where the fault is, but not **why** the fault occurred, it is useful to output to the console (using `print`) the values of variables/data that are used in the code causing the fault. All too often a fault occurs at one line of code because some data was computed or stored incorrectly at an earlier point and it is necessary to trace the cause of the fault back to that point by inspecting the data along the way.

15.3.2 Debugging by Hand-Tracing Code

If inspection of the code and printing out data relevant to the fault does not help you fix it, then we can resort to more formal debugging techniques. For small pieces of code, hand-tracing the execution of the code will usually uncover the cause of a fault.

In hand-tracing we simulate the execution of a program or function on paper. We manually step through the code one line at a time and record the value of each program variable after every line of code, essentially executing the program "on paper". This helps us identify the exact point in the execution at which the fault occurs and incorrect data is generated, and usually gives us insight into why the fault occurred and how to fix it.

It is very hard to demonstrate this process in a reading, but you will see a demonstration during class time.

15.3.3 Integrated Debuggers

For programs/functions that are larger and/or manipulate a very large amount of data, hand-tracing can become impractical. In such case, we can turn to an *integrated debugger* for help. An integrated debugger is a feature of a code editor (like PyCharm) which allows you to have **the computer** step through a program one line at a time. The computer still performs all of the execution of the program as normal, but you get to watch it happen one line at a time. Again, this is very difficult to demonstrate in a reading, so we'll do a demonstration during class. For now we will briefly describe three main features of integrated debuggers that we will demonstrate in class:

Stepping

Integrated debuggers display the program's code, the current value of all variables defined at that point in time, as well as which line of code is about to be executed next. At your own speed you can repeatedly tell the debugger to execute the next single line of code. You can also choose to step inside of function calls, or to execute an entire function call without stopping to look inside.

Inspection

Inspection allows you to dig deeper into the data associated with a particular variable. For example, the variable inspection window allows you to look inside sequences and see, for example, what data items are stored inside of them.

Breakpoints

Breakpoints are useful for larger programs where it is impractical to step through every single line of the program. If you tag a line of code as a breakpoint, then normal program execution will pause when execution reaches that line, and allow you to then inspect variables and/or begin stepping a line at a time. You can also tell a debugger to resume uninterrupted execution of a program until the next breakpoint is encountered. This allows you to quickly run parts of a program you aren't interested in (because you know the fault occurs elsewhere).

15.4 Summary

Of course, testing and debugging doesn't end when the faults are fixed. Code that has been modified to fix faults should be tested again to make sure no new faults were introduced in the course of fixing the previously detected faults.[1] This makes testing and debugging an iterative process (illustrated in Figure 15.1) and is also why investing the time to write a good test driver will save you time in the long run, since, if done right, re-testing is a simple matter of re-running the existing test driver.

One final note: make sure you come to class to see the demonstrations of hand-tracing and integrated debuggers! These are highly interactive processes that we cannot easily show using text and static pictures.

[1] You'd be surprised how often this happens!

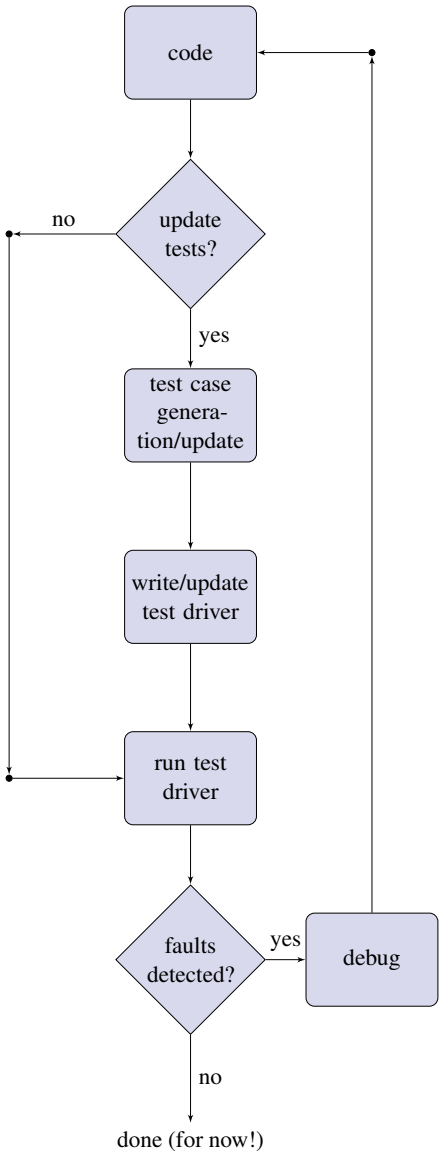


Figure 15.1: The testing and debugging process.

16 — Search Algorithms

Learning Objectives

After studying this chapter, a student should be able to:

- list and describe the possible goals of a searching algorithm;
- describe, in plain English, the linear search algorithm;
- demonstrate the linear search algorithm by hand, identifying which data items are inspected by the algorithm for a given search key;
- describe, in plain English, the binary search algorithm; and
- demonstrate the binary search algorithm by hand, identifying which data items are inspected by the algorithm for a given search key.

16.1 Fundamentals of Searching

One of the most fundamental problems in computing is that of searching a collection of data items for a particular desired data item. This problem is solved by *search algorithms*. In this section we will introduce some basic concepts and terminology surrounding searching. In the following sections we will present two basic search algorithms — linear search, and binary search — which are two very different algorithms for solving the same problem.[1]

16.1.1 Collections

The term *collection* refers to any data structure that stores one or more data items — this includes lists, dictionaries, and arrays. We say that we perform a search on a collection to find a specific item.

[1] Do you remember the difference between *problems* and *algorithms* from Section 1.3? If not, go back and remind yourself.

Type	Search Key	Example
integer	itself	the search key for data item `42` is `42`
float	itself	the search key for data item `42.0` is `42.0`
string	itself	the search key for `'Marvin'` is `'Marvin'`
dictionary	value corresponding to a predetermined dictionary key	the search key for the dictionary `{'name':'Marvin', 'description':'The paranoid android.'}` could be chosen to be the value of either the `name` or `description` fields, i.e. either `'Marvin'` or `'The paranoid android'`.
list	N/A	We do not normally search for a specific list in a collection of lists.
array	N/A	We do not normally search for a specific array in a collection of arrays.

Table 16.1: The search keys typically designated for different types of data.

16.1.2 Search Keys

When searching for an item, we must be able to identify it. Therefore, in a collection, every data item is identified by a *search key*. The type of the data items in the collection being searched plays a role in determining the search key of each data item, as shown in Table 16.1.

The search key of an atomic data item is itself. The search key of a compound data item is usually a specific atomic data item that forms part of the compound data item, for example, a specific field of a record. In the case of Python, records are implemented using dictionaries. The search key of a dictionary can be the value of the key-value pair for any of that dictionary's keys. For example, if we have a collection of dictionaries with keys `'movie_title'`, `'release_year'`, and `'average_viewer_rating'`, we would have to select which of these dictionary keys' corresponding values would be used as the search keys for the dictionaries in that collection. Normally, when searching a collection of records, all of the records have the same set of field names and the search key for each dictionary is the value of the same field (i.e. the value of the same key-value pair for each dictionary).

Note the distinct difference between the terms *search key* and *dictionary key*. Any data item of any type, including dictionaries, can have a search key. Only dictionaries have dictionary keys. The search key of a dictionary is a **value** associated with one of the dictionary's dictionary keys.

Search keys need not be unique, but we often want them to be. If search keys are not unique, then a search for a particular search key could yield multiple matching data items or only the first data item that matches. If we want a unique search key for our collection of movie rating dictionaries, we would have to use the values of the `movie_title` dictionary keys as the search keys since ratings and release year of movies are not unique. The implementation of a particular search algorithm for a particular collection is influenced by whether or not the search keys of a collection are unique.

16.1.3 The Target Key

We search a collection by choosing a *target key* and then look for the item in the collection whose search key matches the target key. For example, if we wanted to search for the movie "Dracula:

Dead and Loving It" in our collection of movie ratings, we would use the string `'Dracula: Dead and Loving It'` as our target key and then determine which (if any) data items in the collection have the target key as their search key (which is the value of their `'movie_title'` dictionary key).

16.1.4 Search Goals

A search can have different goals. We'll discuss two such possible goals here.

Membership

We might perform a search only to determine whether or not there is at least one data item in the collection whose search key matches the target key. The result of such a search is a Boolean value — true or false. Either the collection contains a data item whose search key matches the target key, or it doesn't. This type of search is called a *membership search*. We want to know whether there is a member of a collection that matches the target key.

Retrieval (Look-up)

On the other hand we might perform a search because we want to retrieve the actual data item(s) in the collection whose search key(s) match(es) the target key. This type of search is called a *retrieval search* or a *look-up*. In this case, the result of the search is a new collection (e.g. a list) that contains only the data items from the original collection whose search keys match the target key.

16.2 Linear Search

The *linear search* algorithm is the simplest, but often least efficient search algorithm. Linear search simply examines each data item in the collection, one by one, comparing each data item's search key to the target key. When you find an item whose search key matches the target key, whether or not you can stop depends on whether the search keys are unique, and whether it is a membership search or a look-up search. If it is a membership search, you can always stop as soon as you find a matching data item, and return true. If it is a lookup search, you can only stop if the search keys are unique; if they are not, then there could be more matching items that have not yet been examined.

The name *linear search* comes from the fact that to find something in the collection, the bigger the collection is, the more items you have to look at to find it, and that this relationship between collection size, and number of items examined is a linear relationship. If you were to plot, on a graph, the number of items in the collection against the number of items looked at by the search, then you would see a trend that could be summarized by a straight line.

A linear look-up search can be described in a language-independent way by the following pseudocode:

```
1  Algorithm LinearSearch( C, target_key )
2
3  # a linear retrieval search for all instances of target key
4  # C - a collection of data items
5  # target_key - the target key for the search
6  # Returns: a collection containing items from C whose search
7  #          keys match target_key
8
9  matches = an empty collection
10 for each item i in C:
```

```
     s = search key of i
     if s == target_key:
          add i to matches

return matches
```

This version assumes that keys are not unique. It would still work if keys were unique, but think about how we could improve this algorithm if keys were, in fact, unique.

The following Python code implements a linear look-up search which works when the collection being searched is a sequence (i.e. list, tuple, array) of numbers or strings, and when the data items themselves are the search keys (i.e. when the data items are numbers or strings), again assuming that keys are not unique.

```
def linear_search(C, target_key):
     """
     a linear retrieval search for all instances of target key
     C: a sequence of numbers or strings
     target_key: the target key for the search
     Returns: a list containing items from C whose search
             keys match target_key
     """
     matches = []
     for i in C:
          if i == target_key:
               matches.append(i)

     return matches
```

The key thing to remember (no pun intended) is that a linear search might examine every data item in the collection, for example, in the case where the collection contains no data item whose search key matches the target key! Linear search might stop early if it finds what it is looking for, but that does not affect whether it is a linear search or not. If there is a loop that examines every data item in the collection, then you've got a linear search.

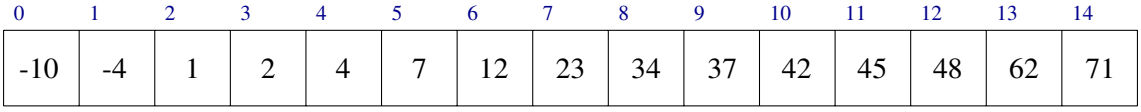
The advantage of linear search is that it is easy to write, and will work on any collection for which you can write a loop to look at each data item. Unfortunately, it is also one of the slowest searches that we know of. So, unless speed is not a concern (it almost always is, in practice), then smarter searches that perform less work are desired.

16.3 Binary Search

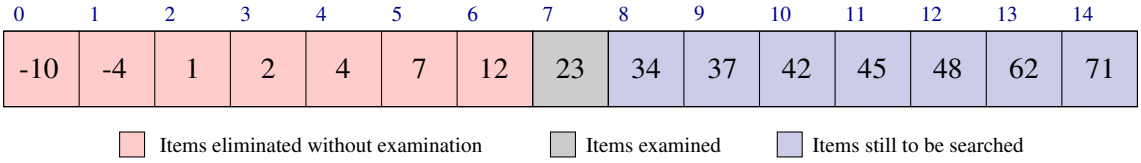
Binary search is one of the most powerful search algorithms. However, binary searches can only be performed on collections where the data items are **sorted** in order by their search keys. So if we want to use binary search on our list of movie rating records from the previous section, then the records need to be sorted in order by movie title (alphabetic ordering). If we want to use binary search on a sequence (array, list, tuple) of numbers, then the numbers have to be sorted in increasing order.

The fundamental concept that makes binary search work is that because the data items to be searched are in sorted order, we can examine one data item, and immediately remove half of the

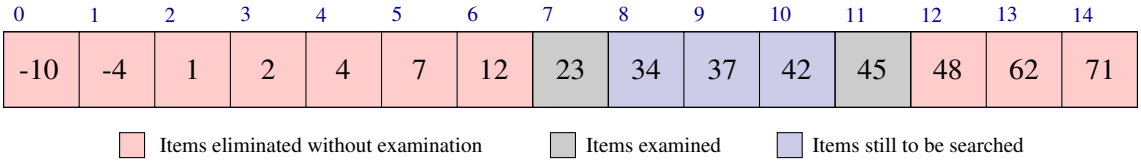
remaining data items from consideration without examining them. How is this possible? Consider the following array of numbers in which we want to search for the number 42:



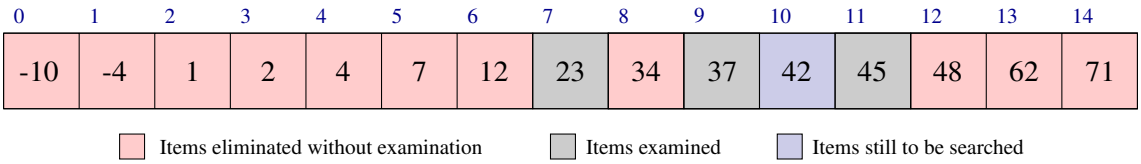
In binary search, the first thing we do is examine the middle data item of the sorted sequence of items — in this case the data item at array offset 7. We examine the data item at offset 7 and find that it is 23. Drat... this isn't the number we are looking for. But, because the array is sorted, we immediately know that the data items at offsets 0 through 6 also cannot be 42, because they must all be smaller than 23! So we need continue searching only in the right half of the array because 42 is larger than 23, and therefore must be somewhere between offsets 8 and 14 (if it is in the array at all):



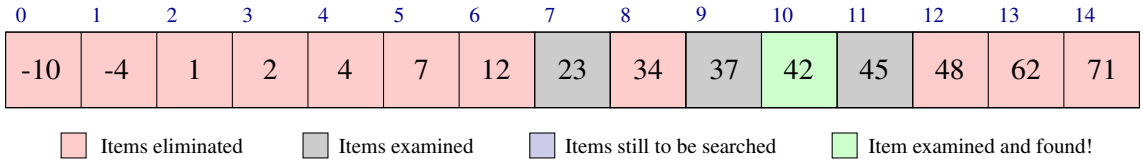
To continue the search, we examine the middle item of those items that are still to be searched (those items shaded in blue, above). The middle item of those still to be searched is at offset 11. We examine it, and find that it is the number 45. This, again, isn't what we were looking for. But, because the array is sorted, we immediately know that if 42 is in the array, it cannot be between offsets 12 and 14, so we have eliminated half of the remaining data items by examining the number at offset 11. Now the only offsets that need to be searched are offsets 8 through 10:



From this point, the search continues in the same way — we examine the middle item of those items that still need to be searched. At this time, that item is the one at offset 9. We examine offset 9 and find that it is the number 37. Still not what we're looking for. But since the array is sorted, we know that data item 42 cannot be at offset 8, because 42 is larger than 37. So now things look like this:



We again continue by searching the middle item of those that still need to be searched. This time there is only one such item, the item at offset 10. We examine the item at offset 10, find that it is equal to our target key, and the search is complete!



Now think about this — something really interesting has happened here. Because the items were sorted we were able to find an item matching the target key by examining only four out of the 15 items (the items shaded red were **never** looked at)! Because we are able to eliminate half of the remaining items every time we examine **one** data item, binary search is extremely fast. We will discuss this in more detail in class, but, by way of example, if you have a collection with 1,000,000 data items, then a binary search will look at **no more than 20** of the data items in the collection. Compare that to a linear search which may have to look at all 1,000,000 items!

The most natural way to write the binary search algorithm is as a recursive algorithm. Here is the pseudocode for a binary search for membership:

```
1  Algorithm BinarySearch(S, target, start, end)
2
3  # a binary search for membership
4  # S: a collection of data items ordered by their search keys
5  # target: the target key
6  # start: first offset of S to be searched
7  # end: last offset of S to be searched
8  # return: true if S contains an item whose search key
9  #    matches the target, false otherwise
10
11 if(end < start):
12      # base case #1: the range of offsets to be searched
13      # has no items in it, so we must conclude the target
14      # is not in the collection.
15      return false
16
17 # find the middle of the array offsets to be searched
18 mid = (start + end) // 2     # note: integer division!
19
20 if search key of S[mid] == target:
21      # base case #2: item was found!
22      return true
23
24 else if search key of S[mid] < target:
25      # if item examined is smaller than target key, it must
26      # be in the right half of the remaining items, so
27      # recursively search there.
28      return BinarySearch(S, target, mid+1, end)
29
30 else:
31      # otherwise the item examined is larger than the
32      # target key, so search the left half of the remaining items.
33      return BinarySearch(S, target, start, mid-1)
```

If we want to search an entire collection, we would invoke the algorithm with an argument of 0 for `start`, and an argument of $N-1$ for `end`, where N is the number of items in the sequence. In this algorithm, `start` and `end` keep track of the offset for the start of the range to be searched (the area shaded blue in the previous example), and the end of the range to be searched, respectively. The variable `mid` is always calculated to be the midpoint between `start` and `end`. Finally, there is an if-statement to compare the search key of the item at offset `mid` against the target key to disqualify half of the current offset range from consideration in the search. It is the splitting of the list in half, and disqualifying half of the list, that makes this a **binary** search. Also note that the size of the collection does not change as the algorithm proceeds, only the set of items in the collection under consideration changes (the items between offsets `start` and `end`, inclusive).

The algorithm as presented assumes that the collection is sorted in *increasing* order. If the collection were sorted in decreasing order, then the only change required is that the less-than operators would have to be changed to greater-than operators.

Finally, here is the binary search for membership implemented in Python. It will work when `C` is any sequence of numbers or strings:

```
def bin_search(C, target_key, start, end):
    """
    a binary search for membership
    S: a collection of data items ordered by their search keys
    target: the target key
    start: first offset of S to be searched
    end: last offset of S to be searched
    return: true if S contains an item whose search key
        matches the target, false otherwise
    """
    if end < start:
        return False

    mid = (start + end) // 2

    if C[mid] == target_key:
        return True
    elif C[mid] < target_key:
        return bin_search(C, target_key, mid+1, end)
    else:
        return bin_search(C, target_key, start, mid-1)
```

16.4 Comparison and Summary of Linear Search and Binary Search

Linear search steps through every data item in the collection one at a time looking for items whose search key matches the target key. Linear search might need to look at every data item in the collection, in particular, in the case where the collection contains no item whose search key matches the target key. For linear search it does not matter if the collection is sorted or not.

Binary search requires that the items in the collection be in sorted order. Binary search reduces the number of items to be searched by half with each item examined, resulting in the examination of

only a very few items, even if there are no items matching the target key.

Comparison of Linear and Binary Search			
Algorithm	Items Examined	Sorted Collection	Effect of examining one item
Linear Search	Possibly All	Optional	Remaining items to be searched reduced by one.
Binary Search	Very Few, Never All[2]	Required	Remaining items to be searched reduced by half.

[2]To be absolutely correct, we must say that binary search will never examine all of the items for collections with three or more data items. For collections of only one item, obviously that one item will have to be examined. For collections of two items, it is possible that both items are examined. For three items or more, at least one item will be eliminated without examination.

17 — Sorting Algorithms

Learning Objectives

After studying this chapter, a student should be able to:

- name three different sorting algorithms;
- list, in plain English, the way that the *insertion sort* algorithm sorts a sequence;
- demonstrate the execution of the insertion sort algorithm by hand, identifying the order of the data items in a given sequence after each iteration of the algorithm;
- explain the *divide and conquer* approach to problem solving;
- describe, in plain English, the algorithm for merging two sorted sequences into a single sorted sequence;
- describe, in plain English, the merge sort algorithm, and identify the *divide* and *conquer* portions of the algorithm;
- describe, in plain English, the *partition* algorithm used by quick sort;
- describe, in plain English, the *quick sort* algorithm, and identify the *divide* and *conquer* portions of the algorithm;
- describe, in general terms, the relative strengths and weaknesses of insertion sort, merge sort, and quick sort.

17.1 Introduction

Any sequence of data items (e.g. Python lists, arrays) where a greater-than/less-than/equal relationship can be established between each pair of items can be sorted. Sorting a sequence means to rearrange the sequence, or create a new sequence, so that all of the data items in the original sequence are in increasing (or decreasing) order. There are many cases when designing a program that you may need to sort a list of things, for example, sorting a list of names into alphabetical order, or sorting expenditures in decreasing order of cost.

Again we have to remember the important distinction between *problems* and *algorithms*. There are dozens of different sorting algorithms, each of which solves the **same** problem — that of putting the items in an input sequence in sorted order.

Sorting algorithms are widely implemented as part of most programming languages or are available as add-ons (libraries, modules). It is rare for us to have to write a sorting algorithm by ourselves. Instead we usually just call the appropriate existing sorting function. However, we still study sorting algorithms because each one has different strengths and weaknesses. Thus, it is important to understand how the different sorting algorithms work, which sorting algorithm is implemented by the sorting function being used, and whether it is appropriate for your data.

If you only care about having a sorted list, and not the speed at which that list is sorted, then it does not matter which sorting algorithm you choose; in the end, they will all sort the list. But if your data has special properties, there may be sorting algorithms that work faster on it than others, or, perhaps more importantly, that you will want to avoid because they are really inefficient for your data. In this course, we will cover three sorting algorithms: insertion sort, merge sort, and quick sort.

Insertion sort is a basic sorting algorithm that is very fast for very short sequences. Merge sort is a recursive algorithm that is a good general-purpose sort in that it performs well in all situations, but is not optimal for every situation. Quick sort is another recursive algorithm that outperforms merge sort **most** of the time, but is **much worse** in a few specific situations. Both merge sort and quick sort are examples of algorithms that were designed using the *divide and conquer approach* to problem solving, which we will introduce prior to discussing the details of the algorithms.

17.2 Insertion Sort

The basic idea behind insertion sort is as follows: if you had a small sorted sequence, you can add a new data item to the sequence, and keep the sequence sorted, by searching through the sequence to find the right place to put the new data item. If you've ever played games with standard playing cards, you've probably used insertion sort. When you are dealt some cards, you pick up the cards one by one, and, as you do so, you put each into your hand in the right spot so that the cards in your hand remain in rank-order. That's insertion sort!

The insertion sort algorithm maintains a sequence S of data items that are already sorted, and a sequence U of data items that have yet to be sorted. At each step of the insertion sort algorithm, one of the items from the sequence U is removed from U, and inserted into the correct position of S so that S, including the new item, remains sorted. This increases the size of the sorted portion by one. It stops only when the entire sequence has been sorted. The sort derives its name from the repeated insertion of a value into the sorted portion of the sequence S.

Let's consider an example of sorting a short sequence of integers. Initially we have the unsorted sequence U, and the empty sorted sequence S:

We now repeatedly remove the first data item in U and insert it into the appropriate spot in S, beginning with the 2:

Then the 5:

Now the 4:

Note how the 5 was shifted to the right to make room for the 4 in S. This is important to note because it takes time. It's not a big deal here because only one data item had to be shifted, but consider a different scenario where we insert a negative number into an S that contains one million positive numbers — you'd have to shift all one million positive numbers over to make room for the negative number! The sort continues by removing the 1 from U and inserting it into S:

Observe how all of the data items already in S had to be moved over to make room for the 1. Finally, we insert the 3, and the sorting is complete:

Remember we said earlier that insertion sort is good for **short sequences**? This is precisely because of the work involved in shifting data items to make room for new ones. For longer sequences, potentially many more data items need to be shifted when small data items are inserted than for short sequences.

Here's one way of implementing insertion sort in Python:

```
def insertion_sort(U):
    """
    creates new sequence containing sorted data of U where
    data is sorted using insertion sort
    U: sequence to sort
    return: new sequence where U is sorted
    """

    # create an empty sequences for S
    S = []

    # insert each item in U into S
    for item in U:
        i = 0

        # search for the offset at which to insert the
        # item into S.
        while i < len(S) and item >= S[i]:
            i = i + 1

        # insert the item immediately before the item at offset i.
        # this also shifts the items at offset i and higher
        # to the right to make space for the new item.
        S.insert(i, item)

    return S
```

Insertion sort is very fast on really short sequences compared to other sorts. However, in general, for arbitrary-length sequences, it is one of the slower sorts. Merge sort and quick sort, which we will discuss in the following sections, are generally much faster sorts.

In-place Insertion Sort (optional reading, not covered on the exam)

Most sorts are more efficient when implemented using arrays rather than lists. It is possible, for example, to implement insertion sort where the input sequence is an array, and no second array is used for S. This is called an *in-place* sort. Instead of using separate arrays for S and U, we store S on the left side of the input array, and U on the right side of the input array. This works because the sum of the lengths of S and U is always constant — whenever an item is removed from U, it is added to S. Initially the entire array is U (because S is empty). As S grows, more of the array is used for S and less for U, until finally, at the end, the entire array is S. The other thing we would have to do is to manually shift items one-by-one by when necessary, instead of relying on the `insert` method of a list to do this for us. See if you can write an insertion sort where the input is an array without using any other arrays. Do you accept the challenge?

17.3 Divide-and-Conquer Sorts

There are two fairly efficient sorts that work on the principle of *divide-and-conquer*. The *divide-and-conquer* approach to problem solving is this:

- **Divide:** If the problem size is so small that the answer is trivial, just return the answer (this is the base case of the recursion). Otherwise, divide the problem to be solved into two smaller sub-problems whose solutions will help you solve the original problem.
- **Recurse:** Recursively solve the sub-problems (possibly by dividing them into even smaller problems in the process).
- **Conquer:** Take the solutions of the sub-problems and combine them into a solution of the original problem.

The divide-and-conquer approach is not specific to solving the problem of sorting, rather, it is a general problem solving approach. We will look at two divide-and-conquer sorting algorithms: *merge sort* and *quick sort*.

17.3.1 Merge Sort

Suppose we are sorting a sequence of data items S containing n items. The merge sort algorithm uses the following divide-and-conquer approach:

- **Divide:** If S has zero or one items, return S immediately, because it is already sorted (these are the base cases!). Otherwise, divide S into S_1 and S_2 such that S_1 contains the first $\frac{n}{2}$ items of S and S_2 contains the remaining $\frac{n}{2}$ (rounding up and down respectively).
- **Recurse:** Recursively sort S_1 and S_2 using merge sort.
- **Conquer:** Obtain a sorted version of S by merging the sorted sequences S_1 and S_2.

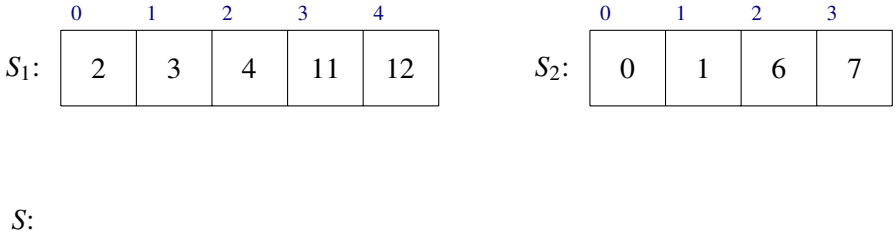
Observe that the "divide" step is trivial — we just chop the array in half and recursively sort each half. All of the work is done in the "conquer" step. The mathematical truth behind this recursive algorithm is that:

$$sorted(S) = merge(sorted(S_1), sorted(S_2))$$

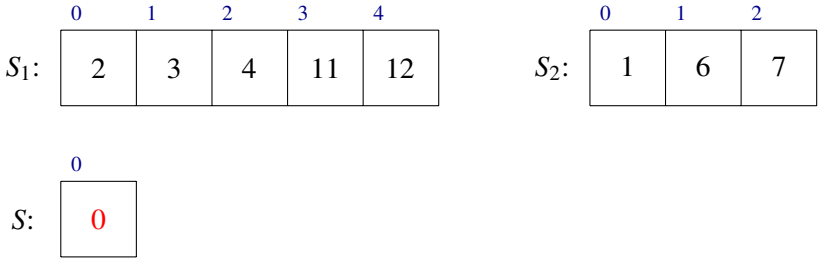
Let's turn this into pseudocode:

```
Algorithm mergeSort(S)
# sorts sequence S using merge sort
# S - a sequence of data items
# return: new sorted sequence of S
if S contains 0 or 1 data items:
     return S

# divide
S1 = first half of S
S2 = second half of S

# recursively sort S1 and S2
S1 = mergeSort(S1)
S2 = mergeSort(S2)

# conquer!!!
S = merge(S1, S2)
return S
```

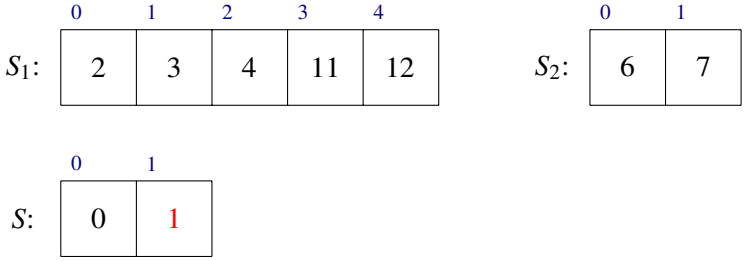
Now it should be easy to see that everything hinges on what is going on in the `merge` function of the "conquer" step (the details of which we abstracted away in the above pseudocode). Intuitively, if we have sorted sequences `S1` and `S2` we can "merge" them together so that we obtain a sorted version of the original sequence `S`. We will now see an example of merging two sorted sequences S_1 and S_2 into a single sorted sequence S. Suppose $S_1 = [2,3,4,11,12]$, $S_2 = [0,1,6,7]$, and S is empty:



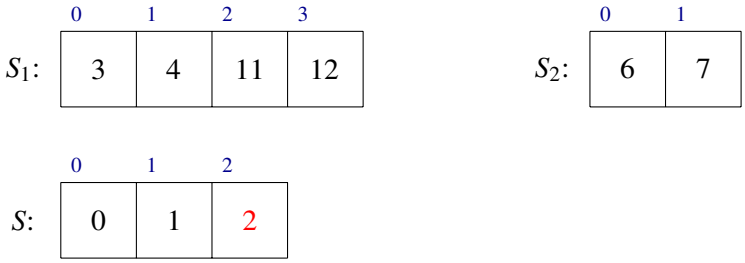
Since S_1 and S_2 are already sorted, we know that the first item of S_1 is the smallest item in S_1 and the first item in S_2 is the smallest item in S_2. That means that one of these two items must be the first item in S, in particular, the one that is the smallest, which happens to be the 0 from S_2. So the first step of the merge operation is to remove 0 from S_2 and append it to the end of S:



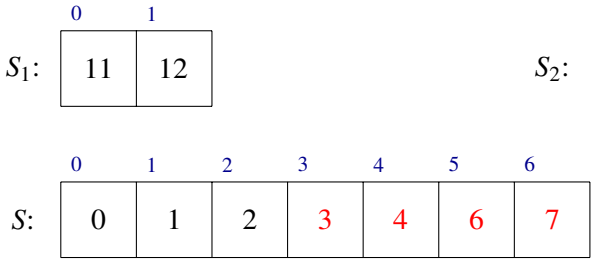
The next item of S must, again, be the smaller of the first item of S_1 and the first item of S_2. This is the 1 at the start of S_2. We remove the 1 from S_2 and append it to S:



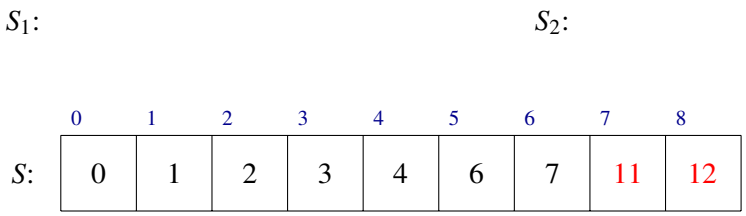
Once again, the next item of S must be the smaller of the first item of S_1 and the first item of S_2. This is the 2 at the start of S_1. We remove the 2 from S_1 and append it to S:



We continue in this fashion, repeatedly moving the smaller of the items at the start of S_1 and S_2 to the end of S until one of S_1 or S_2 is empty. In this example, this occurs after moving 3, 4, 6, and 7 to S:



At this point, we simply append the remainder of whichever of S_1 or S_2 is non-empty to the end of S:



Now S contains all of the data items in S_1 and S_2, in sorted order! Here is how we might implement the merge algorithm in Python:

```
def merge(S1, S2):
    """
    combines two sorted sequences into single sorted sequence
    S1: sorted sequence to combine
    S2: other sorted sequence to combine
    return: single sorted sequence of S1, S2 combined
    """
    # let S be an empty sequence
    S = []

    # repeatedly move the smallest item to S
    while len(S1) > 0 and len(S2) > 0:
        if S1[0] < S2[0]:
            S.append(S1[0])
            del S1[0]
        else:
            S.append(S2[0])
            del S2[0]

    # once one of S1 or S2 is empty, append the remaining
    # non-empty sequence to S.
    if len(S1) > 0:
        S.extend(S1)
    else:
        S.extend(S2)

    return S
```

17.3.2 Quick Sort

Quick sort is another divide-and-conquer approach to sorting in which all of the hard work is done in the "divide" step. Contrast this with merge sort where all of the hard work is done in the conquer step. The approach used by quick sort to sort a sequence of data items S is:

- **Divide**: If S has zero or one items, it is already sorted, so just return S (these are the base cases!). Otherwise, select some item p from S, called the *pivot*. Remove each item from S and place them in one of three sequences:
  - L (for items less than p)
  - E (for items equal to p)
  - G (for items greater than p)
- **Recurse:** Recursively sort L and G using quick sort (E is already sorted by virtue of all of its items being equal).
- **Conquer:** Concatenate the (already sorted) items of L, E, and G. This results in a sorted version of the original S since everything in L is smaller than everything in E, which, in turn, are smaller than everything in G, due to the divide step. The mathematical truth implemented by the recursive quick sort algorithm is:

$$sorted(S) = sorted(L) + E + sorted(G)$$

where $+$ stands for concatenation. Let's express these ideas in pseudocode:

```
Algorithm quickSort(S)
# sorts sequence S using quick sort
# S - array of data items to be sorted
# return: sorted sequence of S

# divide into sub-problems
pivot = any item of S                # e.g. the last item of S
L = items in S smaller than pivot    # these items may be in any order
G = items in S larger than pivot     # these items may be in any order
E = items in S equal to the pivot

# recursively solve the sub-problems of sorting L and G
quickSort(L)
quickSort(G)

# L and G are now sorted

# conquer!!!
S = L + E + G  // (where + represents concatenation)
return S
```

We can already see that the conquer step is easy; we just need to paste together the already-sorted sequences L, E and G because we know that everything in L is smaller than E and that everything in E is smaller than G. Most of the work in quick sort is done in creating L, E, and G, in the first place. Conceptually, this is very easy. We can initialize L, G, and E to be empty sequences, choose an item of S (say, the first item) to use as the pivot p, and then just append each item of S to the appropriate

sequence. Here is how we might do that in Python[1]:

```
# divide step of quick sort
L = []      # for items smaller than the pivot
E = []      # for items equal to the pivot
G = []      # for items greater than the pivot
p = S[0]    # choose the first item as the pivot

for x in S:
    if x < p:
        L.append(x)
    elif x > p:
        G.append(x)
    else:
        E.append(x)
```

Now all we need to do is recursively sort L and G, then concatenate L, E, and G to obtain a sorted sequence.

Quick sort is usually faster than merge sort and insertion sort (except for very short sequences where insertion sort wins). In class we will examine the type of inputs for which the performance of quick sort becomes poor.

[1] In practice, this is not a very efficient implementation of quick sort, but an efficient implementation using arrays is beyond the scope of this course. If you decide to major in computer science, you'll encounter the efficient implementation in second year.

18 — Binary Number Systems and Logic

Learning Objectives

After studying this chapter, a student should be able to:

- describe what the binary number system is, using concepts of digits and bases;
- perform addition and multiplication on binary numbers;
- convert binary numbers to decimal numbers, and vice versa;
- understand the connection between binary numbers, circuits, and logic; and
- recognize the importance of logic to computer science.

18.1 Introduction

The design of electronic computers uses electronic circuitry to make the computer work. As we said in an earlier reading, data is represented as numbers, and that is true at a certain level of abstraction. Electronic engineers found it convenient and robust to represent data using voltages in electronic circuits: a voltage is considered "high" if it is above a certain threshold, and "low" if it is below the same threshold. The threshold itself depends on the design of the electronic device. This itself is an abstraction that allows us to ignore exact voltage values.

The binary number system is built on the abstraction that the low voltage can represent the quantity 0, and the high voltage can represent the quantity 1. This electronics design convention is exposed to programmers in some languages by equating 0 with the boolean value false, and 1 with the boolean value true. As a result of all these common ideas, the notions of computer programming (data), digital circuit design (voltages) and Boolean logic are all intertwined.

The question for electronics designers is "how to design a circuit that implements data operations?" We know that computers can do things like arithmetic, but how do circuits carrying voltages actually do that? In essence, there are circuit components (transistors) that affect voltages in circuits in ways very similar to the kinds of operations we know as Boolean operators. So, for the electronic

engineer, a knowledge of Boolean logic directly assists circuit design. Of course, logic is not the only concern for electronics design: the physical constraints of electronic materials plays a big role too: circuits are usually laid out on a 2D surface, and these circuits generate heat, which must be managed. We won't say more about this here, but that's not because there's nothing more to say.

The question for programmers is "How much do I need to know about Boolean logic to enhance my programming skills?" as well as "What do I need to know about how the computer works to make sure I am not missing anything important about my work?" Boolean logic is used in programs in conditional statements (if-statements) and repetition constructs (loops, recursion), so it's obvious that we need them. Boolean logic is also the first and simplest approach to a branch of mathematics that's very important to Computer Science: formal logic. We need formal logic to clarify our thinking about algorithms. In an introductory course such as CMPT 141, we don't really need such tools, but they become crucial as we move from introductory concepts to advanced concepts. Formal logic is used in the design of digital circuits, the verification of software correctness, and as an approach to formal reasoning by computers in artificial intelligence. There is also a programming language called Prolog ("PROgramming in LOGic") built entirely from the idea that formal logic is actually a model of computation.

18.2 Binary Numbers

In this section we introduce the *binary* representation of numbers.

18.2.1 Numbers vs Numerals

First, we have to make a comment about the difference between a number and a numeral. This is a subtle distinction, but it will help us a bit to talk about binary numbers unambiguously. A *number* is quantity, an amount. A *numeral* is a symbol used in the representation of a number. The number associated with the English word "three" is a quantity which can be expressed in a variety of ways, that is, through various representations. For example we could represent the number "three" in base 10 (decimial) with the Arabic numeral **3**. The ancient Romans used a system of number representation with numerals such as `I`, `V`, `X`, in which the number "three" is represented by the string of three numerals `III`. The important thing in the following discussion is to not confuse the **representation** of a number with its **quantity**. We'll use English words for smallish numbers, and use Arabic digits for numerals, until we think the point is clear.

18.2.2 Representation of Binary Numbers

We'll build our understanding of binary numbers from the existing understanding we have of decimal numbers and we'll focus solely on positive integer numbers. Decimal numbers are expressed using ten different numerals: 0,1,2,3,4,5,6,7,8, and 9. The fact that there are 10 numerals is the reason why another name for decimal numbers is *base 10 numbers*. This is beacuse each numeral in a decimal number represents a multiple of a power of 10. The right-most numeral denotes a multiple of 10^0 or one, the second-right most numeral denotes a multiple of 10^1 or ten, the third right-most numeral denotes a multiple of a power of 10^2 or one hundred.[1] That's why these numeral positions are respectively called the ones, tens and hundreds columns. Thus, the decimal number 8209 represents 9 ones, plus 0 tens, plus 2 hundreds plus 8 thousands. In other words:

$$8209 = 8 \times 10^3 + 2 \times 10^2 + 0 \times 10^1 + 9 \times 10^0.$$

[1] See how the number ten is the base for the exponent in each case? That's *base-10*!!!

Notice that there is no numeral for the number ten. In fact, we don't need one because we can represent the quantity ten as a combination of more than one numeral, namely, with 1 tens plus zero ones: $10 = 1 \times 10^1 + 0 \times 10^0$.

Notice that integer division by 10 is easy in decimal: we simply remove the numeral on the right. Try it! Multiplying by ten is equally easy: we append the numeral 0 onto the right end of the number. Try this too! Remember, this only works for **integer division**, but it's a handy trick to know.

There's no real need to represent numbers with exactly ten numerals. It is simply convenient for us humans because we happen to have ten fingers. Had things turned out differently, humans might have six fingers on each hand, and might have developed a number system with twelve numerals. It would be no more difficult for us than what we have now (though it might be a bit more trouble if we were descended from centipedes or millipedes). In fact, English words "eleven" and "twelve" are evidence that at least some cultures used a base-twelve number system in the past. In French, linguistic evidence points to a base-16 number system (numbers represented with 16 numerals).

The binary number system, also known as the base-2 number system, uses only two numerals, namely the Arabic symbols 0 and 1. In computers, binary numerals are called *bits*, which is short for *binary digit*. Because these same numerals are used in base-10, this immediately causes confusion. How do we tell the difference between the base-10 number 111 (the quantity one hundred and eleven) and the base-2 binary number 111 (the quantity seven)? To avoid this confusion, we will adopt the following convention. Decimal numbers are written normally; binary numbers will be written with "0b" as a prefix. Using this convention, 111 is the base-10 number one hundred and eleven, and 0b111 is the binary number seven, and 7 is the base-10 number seven. The "0b" prefix contributes no quantitative information to the number; it is simply an annotation to remind us we're dealing with a binary number.

A binary number like 0b1101 represents a quantity in a sequence of binary digits. Literally, this sequence means "1 eight plus 1 four plus 0 twos plus 1 one". The words "eight, four, two, one" are powers of two, and the bigger a number is the more powers of two we will need to represent it.[2] Notice that this is **no different** from base-10 except that there are fewer numerals, and the **base** of the powers that are associated with each numeral's position is changed from 10 to 2. The powers of two that we use for each position, starting with the right-most position are $2^0 =$ (one), 2^1 (two), 2^2 (four), 2^3 (eight), and so on. For a positive integer with k binary digits, we'll need the powers of two from 2^0 to 2^{k-1}.

And now for a little test. If at this point you understand the difference between numerals and numbers, and between decimal (base-10) and binary (base-2) number representations, you will appreciate this joke:

> "There are 10 kinds of people in the world: those who know binary and those who don't."

Of course, we would have written "0b10 kinds of people," but that obliterates the tiny amount of humour in the joke.

In the same way that multiplying and dividing by ten was easy for decimal numbers, multiplying and dividing by two is easy in binary. To divide by two, just remove the numeral on the right. For example, if we take the number thirteen and divide by two (integer division) then the answer is six. Let's see how this works in binary. In binary, the number thirteen is 0b1101, that is, 1 eight, 1 four, 0 twos, and 1 one. If we remove the right-most numeral, we are left with 0b110, which is 1 four, 1 two and 0 ones, which adds up to six! Multiplying by two is equally easy: we append the digit zero to

[2] The base of the powers is 2, hence *base-2*!!

the right hand side. For example, we can multiply the number 0b101 (five) by two by appending a zero to get 0b1010, which is 1 eight, 0 fours, 1 two, and 0 ones, which adds up to ten.

Table 18.1 lists binary and decimal representations for a small set of integer quantities.

Decimal	Binary	Decimal	Binary	Decimal	Binary	Decimal	Binary
0	0000 0000	1	0000 0001	2	0000 0010	3	0000 0011
4	0000 0100	5	0000 0101	6	0000 0110	7	0000 0111
8	0000 1000	9	0000 1001	10	0000 1010	11	0000 1011
12	0000 1100	13	0000 1101	14	0000 1110	15	0000 1111
16	0001 0000	17	0001 0001	18	0001 0010	19	0001 0011
20	0001 0100	21	0001 0101	22	0001 0110	23	0001 0111
24	0001 1000	25	0001 1001	26	0001 1010	27	0001 1011
28	0001 1100	29	0001 1101	30	0001 1110	31	0001 1111

Table 18.1: Some 8 bit binary numbers. The space between the sets of 4 bits is used to help readability, much the same way that commas are used in decimal numbers.

18.2.3 Converting from Binary to Decimal

Computers represent numbers in binary. Almost always, when we ask the computer to display a number by printing it on the console or other output device, it automatically converts its internal binary representation into decimal. This is very easy to do, and, in fact, we already did it a number of times in the previous section when we added up the multiples of the powers of two that the binary number represents.

Example: The binary number 0b1010110 has 7 bits. To convert this to decimal, we need to add up the corresponding multiples of the powers of two from $2^0 = 1$ to $2^6 = 64$. Reading the binary number from left to right, we have 1 sixty-four plus 0 thirty-twos plus 1 sixteen plus 0 eights plus 1 four plus 1 two plus 0 ones. Or, more simply:

$$1 \times 64 + 0 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 86$$

Note that as a shortcut, you can ignore the bits that are zero:

$$1 \times 64 + 1 \times 16 + 1 \times 4 + 1 \times 2 = 86$$

The only real difficulty here is calculating the powers of 2, but the first eight or nine powers of two are memorized easily enough.

18.2.4 Addition of Binary Numbers

For addition of binary numbers, there are four basic addition facts that must be memorized:

1. $0b0 + 0b0 = 0b0$
2. $0b0 + 0b1 = 0b1$
3. $0b1 + 0b0 = 0b1$
4. $0b1 + 0b1 = 0b10$

The first three facts are very straightforward because they involve adding something to zero. Zeroes are zeroes in every number representation system, and adding zero leads to no change, just like in decimal. The last fact is more interesting. We get two bits as the result, because of *carrying*.

In decimal $1+1=2$, and does not cause a *carry* into the tens column like, say, $6+6=12$ would. But in binary, $0b1 + 0b1 = 0b10$ and **does** cause a carry. There's a carry of 1 to the next numeral position (the twos column, in this case). Carrying in binary works **exactly** the same way it does in decimal.

The algorithm for addition in binary is **exactly** the same as the one you learned in third grade for addition in decimal. The only difference is that in decimal you had to memorize a lot more facts (55 of them: $1+1=2$, $1+2=3$,$1+3=4$,$1+4=5$, ...).

When we add binary numbers, we line the numbers up so that the right most digit is in the same column. Then we do addition in columns from right to left, using the 4 binary addition facts above. If we carry a 1 to the next column, we include it in the addition of that column, as normal.

Here's an example binary addition, we just dropped the 0b annotations here to reduce clutter. The columns being added in each step are shown in red, carries are shown in blue.

1		11		111		111	
111		111		111		111	
+ 11	⇒	+ 11	⇒	+ 11	⇒	+ 11	
0		10		010		1010	
one's column		two's column		four's column		eight's column	

In this next example we show an example of decimal addition and an example of binary addition where the pattern of adding and carrying is identical. This illustrates that addition in both systems of number representation is identical, except for the particular set of addition facts that are used. As in the previous example, the columns being added in each step are shown in red, and carries are shown in blue.

decimal:	1		1		1 1		1 1	
	628		628		628		628	
	+ 507	⇒	+ 507	⇒	+ 507	⇒	+ 507	
	5		35		135		1135	
binary:	1		1		1 1		1 1	
	101		101		101		101	
	+ 101	⇒	+ 101	⇒	+ 101	⇒	+ 101	
	0		10		010		1010	
	carry		no carry		carry		no carry	

18.2.5 Multiplication of Binary Numbers

The algorithm for multiplication in binary is the same as the one you learned for multiplication in decimal. You need to know binary addition (see previous section) and you need the following 4 facts about multiplication of single-numeral binary numbers:

1. $0b0 \times 0b0 = 0b0$
2. $0b0 \times 0b1 = 0b0$
3. $0b1 \times 0b0 = 0b0$
4. $0b1 \times 0b1 = 0b1$

Here's an example (again, we drop the 0b flag to reduce clutter). The numerals being multiplied at each step are shown in red. The final addition is shown in blue.

$$\begin{array}{r} 110 \\ \times\ 11 \\ \hline 0 \\ + \\ \hline \end{array} \Rightarrow \begin{array}{r} 110 \\ \times\ 11 \\ \hline 10 \\ + \\ \hline \end{array} \Rightarrow \begin{array}{r} 110 \\ \times\ 11 \\ \hline 110 \\ + \\ \hline \end{array} \Rightarrow$$

$$\begin{array}{r} 110 \\ \times\ 11 \\ \hline 110 \\ +\ 0 \\ \hline \end{array} \Rightarrow \begin{array}{r} 110 \\ \times\ 11 \\ \hline 110 \\ +\ 10 \\ \hline \end{array} \Rightarrow \begin{array}{r} 110 \\ \times\ 11 \\ \hline 110 \\ +\ 110 \\ \hline \end{array} \Rightarrow \begin{array}{r} 110 \\ \times\ 11 \\ \hline 110 \\ +\ 110 \\ \hline 10010 \end{array}$$

18.2.6 Subtraction and Division

We won't cover this directly, but you already know enough to work these out. They are not significantly different from the decimal versions. Long division in binary is significantly easier than in decimal. Seriously. Try it!

18.2.7 Converting from Decimal to Binary

Converting from decimal to binary is likely the only thing in this chapter that you will find that is truly new in the sense that we need an algorithm that you probably don't already know. Before we begin, we want to remind you that we are only changing the representation of the number when we convert from binary to decimal. The quantity of the number does not change when we change representations.

Suppose we want to convert a number, let's call it x, from decimal to binary. The conversion algorithm repeatedly divides x by 2 using integer division, and the remainders from the integer division at each step are collected into a sequence that forms the binary representation of the original decimal number. The result of each division is used in the next division. The algorithm finishes when the result of the division is zero. Here is the algorithm presented as pseudocode:

```
Algorithm DecimalToBinary(x)
x: a decimal number to be converted to binary
Returns: binary representation of x

Let b be an empty string
while x > 0:
     r = x % 2  # r is a remainder, either 0 or 1
     add r to the left side of the string b
     x = x // 2            # integer division!

if b is still an empty string: # (i.e., x was 0 to begin with)
     return "0"
else:
     return b
```

The sequence `b` is built-up from right to left from the remainders resulting from each division. The variable `r` holds the remainder for the current division. The following table shows the values of `x`, `r`, and `b` at the end of each iteration of the while loop in the algorithm when it is given an initial value of $x = 27$.

x	r	b	Comments
27	–	–	Initial value of x before the while loop
13	1	0b1	After 1st loop iteration; $27/2 = 13$, remainder 1.
6	1	0b11	After 2nd loop iteration; $13/6 = 6$, remainder 1.
3	0	0b011	After 3rd loop iteration; $6/2 = 3$, remainder 0.
1	1	0b1011	After 4th loop iteration; $3/2 = 1$, remainder 1.
0	1	0b11011	After 5th loop iteration, $1/2 = 0$, remainder 1.

At the conclusion of the loop, since `b` is not empty, the answer is `b` = 0b11011. It's easy enough to check our answer by converting 0b11011 back to decimal:

$$\begin{aligned} 0b11011 &= 1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0 \\ &= 16+8+2+1 \\ &= 27 \end{aligned}$$

This algorithm can be adapted to convert numbers in **any** base back to decimal. Simply change the algorithm so that the base of the input number is used in the division and remainder operations in place of the number 2.[3]

18.2.8 Binary Addition and Multiplication: Connections with Logic

You may have noticed that the facts for adding and multiplying single-numeral binary numbers look suspiciously similar to Boolean AND and OR operations. This is no coincidence and it has some

[3] This is a *generalization* of our algorithm! We changed it from working for only one specific base to working for **any** base. Remember we talked about generalization way back in Section 4.5?

interesting implications. For one, it means that computer hardware can be engineered to perform arithmetic using AND and OR operations, which are precisely the kinds of things that fundamental electronic components are good at! In the following tables, we illustrate the similarities between addition/OR and multiplication/AND, respectively (all table numbers are binary, 0b prefix omitted).

x	y	$x+y$	$x \times y$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	10	1

x	y	x OR y	x AND y
false	false	false	false
false	true	true	false
true	false	true	false
true	true	true	true

In examining the above table, note the correspondence between the binary value 0b0, and the Boolean value *false*; likewise the binary value 0b1 and the value *true*.

18.2.9 Going Further with Number Representations

Positive integers are easy. If you want to represent negative integers *and* positive integers, it takes a bit more work. In normal arithmetic using decimal numbers, we represent a negative by a dash in front of the number. We could do the same in binary, giving us simple extensions to all the familiar concepts.

But computers don't do that. Ultimately, if a computer is to do arithmetic, we have to think about how to represent the dash as a circuit. So far, we've been writing down binary numbers using only as many digits as are needed. But you may remember that computers have finite memory, and that bits (binary digits or numerals) are grouped together into bytes, words, and so forth. So let's think about numbers that are represented by a fixed number of bits, say 8 bits (one byte)[4]. With 8 bits, every number will always have exactly 8 numerals, using leading zeros on the left to fill the spots that aren't really needed to represent the number. For example, the number 0b101 would be 0b00000101 as an 8-bit number. The leading zeros have no effect on the quantity of the number. With 8 bits, we can represent all of the positive integers from 0 to 255. It is not possible to represent a larger positive number using only 8 bits.

To represent negative numbers, computers could use one bit of a number to indicate whether it is positive or negative, and the remaining bits to represent the magnitude of the number. This approach is called the *signed magnitude* representation. It's a good place to start learning, but it's not used in modern systems, for reasons we'll mention later. Using a bit *inside the number itself* means we don't have to add circuitry, which is good, but it does mean we have to be careful about how to interpret numbers for use in calculations. Suppose we decide to let the left-most bit represent whether the number is negative. If the left-most bit is 1, this will mean that the number is negative. Under this scheme, 0b10000101 is a negative integer, and 0b00000101 is a positive integer. Notice that we have only 7 bits to represent the magnitude of the number, from 0 to 127, so we can represent numbers between -127 and $+127$ (the 7-bit magnitude can be a quantity between 0 and 127, and the left-most *sign bit* determines the sign).

The signed magnitude representation has a little problem: there are two representations for zero! Specifically there are 0b00000000 and 0b10000000 which are $+0$ and -0, respectively. This causes a whole bunch of special cases to arise in the circuitry when you want to check whether a number is equal to zero, which happens really really often! To avoid this, modern computer design uses a clever

[4]Modern computers typically use 32 bits, 64 bits, or even 128 bits to represent integers.

variation on "signed magnitude" to represent positive and negative integers. If you are interested in reading further about such representations, Google the "one's complement" and "two's complement" representations. The two's complement representation is what is used to represent integers in most modern-day laptop and desktop computers. This is covered in later computer science courses, too.

But wait... you said... (optional reading, not covered on the exam)

Yes, we said here that integers are represented in computers by sequences of bits with fixed length, which means that we cannot represent numbers larger than a certain quantity, **and** we said back in Section 2.1.3 that there is no limit to the quantity that we can store as a Python integer. Here's the thing: both are true. Python integers are not stored with a fixed number of bits, but instead use an entirely different implementation called *arbitrary precision integers* and arithmetic is performed using *arbitrary precision arithmetic*.

The basic idea of *arbitrary precision integers* is that numbers are represented as strings of characters `'0'` through `'9'`. The advantage of this is obvious: any number can be stored no matter how big it is because strings can be any length we want. The disadvantage is that arithmetic cannot be performed using the computer's built-in arithmetic hardware, and instead has to be done in software, which results in it being a little slower. Most common programming languages (Python being a notable exception) store integers using a fixed number of bits.

One final note: integers stored in `numpy` arrays are not like standard Python integers. Because of the nature of arrays, these are stored using a fixed number of bits, which is why numpy arrays have `dtype`'s like `int64` (64-bit integers), `int32` (32-bit integers), etc.. This is one (but not the only) reason why `numpy` arrays offer a speed advantage over lists.

18.3 From Boolean Operators to Propositional Logic and Beyond

We are familiar with Boolean values, and Boolean operators, because they help us express conditions in if-statements and repetition constructions like loops and recursion. They are essential for expressing contextual information within an algorithm. And now that we've looked at how numbers are represented in binary, you should almost be able to see that numerical operations can be expressed by electronic circuits; you have enough information to work this out yourself, but we won't say any more about it in this book. There are other courses for that!

The next step from Boolean logic is called *propositional* logic. It adds two fundamental ideas to the mix. First: knowledge can be represented in a form that's not too far from what we have learned from programming with Boolean operations. Second: proofs (that is, a sequences of verbal or symbolic reasoning steps in support of a claim) can be constructed for some statements in this form that are unequivocally valid and correct. A statement that can be supported with a proof cannot be doubted, and that's a very powerful concept.

Propositional logic is one step removed from a more expressive language loftily called "the predicate calculus". It's used to formalize a wider variety of arguments, for wider purposes. The predicate calculus is powerful enough to express useful properties of interest to philosophers, scientists, engineers, and software developers, much the same way that a programming language lets us express computations. Formal logics are enhanced by formal reasoning techniques, sometimes expressed as strategies, and sometimes as algorithms, with which the properties can be established

with a formal, valid proof. It is worthwhile to mention that the predicate calculus, in combination with a specific formal reasoning technique, can be used as the basis of a programming language, as capable as Python or any other programming language.

The main point of this section is to point out that there are very deep ideas below the surface of the material we've been studying. These are covered in later computer science courses.

18.4 Common Pitfalls

When you see a binary number, like 0b1101011, you might not see it as a recognizable quantity, and that could be confusing. But try not to think of it as a word that you don't understand. Think of it only as a quantity whose magnitude you have not yet determined. Large decimal numbers are accessible to us only because of the amount of time we have already spent using them.

19 — Computer Architecture

Learning Objectives

After studying this chapter, a student should be able to:

- describe the differences between a fixed function and a von Neumann architecture computer;
- provide definitions for *bit* and *byte*;
- describe the organization of main memory in a computer;
- explain what a *memory address* is, and what it is used for;
- describe the basic function of the central processing unit, arithmetic logic unit, and control unit of a computer;
- explain the purpose of the instruction pointer register and instruction register; and
- describe the steps of the machine cycle algorithm.

19.1 Introduction: The von Neumann Architecture

Computer architecture is the conceptual design and operational structure of a computer system. When discussing a computer architecture, we talk about the parts of a computer (the hardware) and how those parts operate together to run computer programs (software).

Prior to the mid-1940s all computers were *fixed function* computers. That is, they were built to perform a single specific task (adding numbers; or calculating ballistic trajectories; or calculating the values of the logarithm function) and could perform no task other than the one they were built for. Such computers were built using a combination of mechanical and electronic parts.

John von Neumann changed all that in 1945. The key innovation of the von Neumann architecture is the idea that computer programs can be stored in computer memory just like data! In other words, von Neumann realized that the instructions for carrying out a task could be treated like data in the sense that they can be stored in the computer's memory right alongside the data to be used while

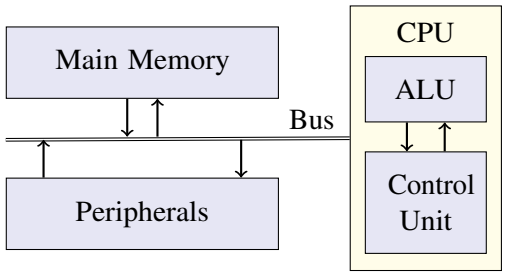


Figure 19.1: Conceptual diagram of the von Neumann architecture. We'll talk about all of the components shown here – main memory, CPU, and peripherals – in sections 19.2 through 19.6.

performing the task. Remember that, prior to this innovation, computers were hard-wired to perform only one task; essentially computer programs were **hardware**. By treating computers programs as data instead, and storing them in the same way and in the same place as data, a von Neumann architecture computer is able to perform different tasks simply by storing different programs in memory as **software**. It seems obvious to us now, but back in 1945, this was a seriously big deal.

Figure 19.1 shows a conceptual diagram of the von Neumann architecture. In this diagram the lines (with and without arrows) indicate *bus* lines. A *bus* is a set of wires connecting the components of a computer; a *bus* is used to send data (in the form of electrical impulses along wires) between the components. You know all those thin lines you see on your computer's motherboard? Some of those, at least, are the bus.

19.2 Main Memory

The *main memory* is the central storage centre for the computer; this is where your program and program data are stored when the program is running. The basic unit of your computer's main memory is a simple circuit for storing 1 bit of data; a *bit*, or binary digit, is just a single 0 or 1 value. To represent a bit using an electronic circuit, engineers define a threshold voltage for the circuit; if the voltage of the circuit is above the threshold then the circuit is holding a bit with value 1, otherwise the bit has value 0. These 1-bit circuits are organized into groups of 8 called a *byte*. A byte is the smallest accessible memory unit on a computer. All of the data in your computer — program instructions, numbers, letters, pictures, etc. — are stored as one or more bytes.

Memory, as a whole, is organized as a very large sequence of bytes (see Figure 19.2). Each byte in the sequence has a numeric *address* that is used by the computer to access the data stored in the byte very quickly. This *address* is not actually stored in memory; it is implicit in the design of the memory circuitry. The *address* of the first byte in the sequence is 0, the second is 1, the third is 2, and so on; think of the address of a byte in memory as its offset from the start of the sequence (the first byte in the sequence is offset 0 from the start of the sequence). Viewed in this way, the main memory of a computer is a very large one-dimensional sequence of bytes.

The maximum amount of memory that can be used in a computer is dictated by the number of bits used by one of its addresses; an N-bit computer (ex: 8-bit, 32-bit, 64-bit) uses N bits for its memory addresses, and can have an absolute maximum of 2^N bytes of memory installed in it (in practice this number is lower due to other contraints, but it is in the same ballpark).

Data is divided into byte-sized pieces in part because it allows hardware designers to move data in parallel. Typically, an N-bit computer can move N-bits ($N/8$ bytes) at the same time (rather than

Address	Data
000	1101 1001
001	0010 0101
002	1001 0000
003	0000 0000
004	1100 0010
005	1100 1100
006	0101 0101
007	1010 1010
	⋮

Figure 19.2: Organization of main memory.

one at a time, which would be N times slower, at least).

19.3 Central Processing Unit

The *central processing unit* (CPU), often called the "brain" of the computer[1]; is the circuitry that does most of the actual work of computation. The CPU is made up of an *arithmetic logic unit* (ALU) and a *control unit*.

The ALU is where all of the circuits are located for data manipulation. For example, the ALU knows how to add together two numbers that are represented in binary as a sequence of bits and give the result, also as a binary number. It also knows how to do other simple arithmetic operations on integers and floating point numbers, comparisons to see which of two numbers is larger, and boolean operations like AND, OR, and NOT.

The control unit contains the circuitry for coordinating execution of a computer program; it uses the bus to fetch data and program instructions from main memory as needed, feeds data to the ALU to perform bits of computation, and obtains the results of such computations, possibly also storing them back into main memory.

A very small amount of data can be stored right inside the CPU in extremely high-speed memories called *registers*. Because of the laws of physics, communication between the CPU and main memory is relatively slow: the CPU and ALU can do hundreds or thousands of other things in the same amount of time it takes the CPU to communicate data to or from main memory! On the other hand, storing data in CPU registers is relatively fast. Registers can be used as temporary storage for small amounts of data that will be used in the very immediate future to avoid the relatively high time costs of communication with main memory. This is especially useful if some piece of data will be needed several times in successive calculations.

Different CPU designs have different numbers and sizes of registers, but there are some special-purpose registers that are common to all CPUs (though they may go by different names). One of these common registers is the *instruction pointer* (IP); also called the *program counter* (PC). Another

[1] While metaphorically called a "brain", the CPU is actually pretty stupid in the sense that it can only do exactly what it is told to. It performs the same very simple sequence of steps over and over (see Section 19.5).

is the *instruction register* (IR). Both of these registers are key to the function of the CPU and we will describe them in the next section.

19.4 Machine Instructions

A computer application consists of a sequence of machine instructions, represented as bit-sequences. Each instruction is typically very simple. For example, a *load* instruction retrieves data from a given address in main memory, and stores it in a given register. An *add* instruction adds the data in 2 given registers together, storing the result in a third register. A *store* instruction sends data from a given register out to a given address in main memory.

The design of the CPU determines the number and kinds of instructions that the CPU can perform. Most CPUs have similar kinds of instructions, and some are specialized in some way or another. One thing to keep in mind is that the instructions are very mechanical. Another thing is that the instructions are very limited, relative to the way people think about doing calculations. Finally, there's no mind inside the CPU monitoring the activities. The CPU really is like a machine filled with cogs and springs and axles, and the machine instructions only affect which cogs and wheels are activated at any time.

Miraculously, we can organize a sequence of machine instructions so that the machine does useful things for us. In an introductory course, we don't usually work directly with machine instructions. The machine instructions were designed for the machine, not for humans. An easier place to start is to work with a language that was designed for humans (like Python!). But it is helpful to remember that the language we will study gets converted to machine instructions.

19.5 Fetch-Decode-Execute

Programs that are running are stored in main memory. Because the program's instructions are in memory, the CPU's job is very simple. The CPU just repeatedly performs the *machine cycle*. This is an algorithm that has three steps: (1) Fetch; (2) Decode; and (3) Execute. A CPU performs these three steps over and over and over again to run a program.

The instruction pointer (IP) that we mentioned in the previous section is a special CPU register that contains the memory address of the next program instruction to be executed. In the *fetch* step, the CPU will send a signal to main memory asking for whatever value is stored at the memory address contained in the IP register. That value is sent back from main memory and stored in the CPU's instruction register (IR). At the same time, the value in the IP is updated to contain the memory address of the next instruction in the program, so it can be retrieved in the next repetition of the machine cycle. In this way, the IP always contains the memory address of the next instruction to be fetched.

In the *decode* step, the CPU decodes the instruction stored in the IR. Decoding means that the instruction is used to enable appropriate circuits in the CPU, and sometimes the ALU, that are required to perform the instruction. Different instructions are encoded using different sequences of bits. For example, the addition operation is represented by one sequence of bits, and the subtraction operation by another. The sequence of bits in the IR will also contain an encoding of which registers should be used in the operation (if any). For example, if the instruction is an addition instruction, there will be bits in the instruction stored in the IR that indicate which CPU registers contain the numbers to be added.[2]

[2]You might be wondering how such numbers would get into the CPU registers in the first place. A *load* operation

The *execute* step simply performs the operation indicated by the decoded instruction, by activating the circuits enabled in the decode step. If the instruction requires that data be sent to, or retrieved from, main memory then the instruction will be performed by the control unit of the CPU. Otherwise, the instruction is mathematical or logical in nature and is performed by the arithmetic logic unit. In the case of the example of an addition instruction in the previous paragraph, the control unit of the CPU would activate circuits to send the contents of the two registers indicated in the IR to the ALU and ask it to add them. Then the ALU would send the result back to the CPU which would store it in a third register. A subsequent instruction might then cause the result of the addition to be moved back to main memory, or the result might be held in the register to be used later by a subsequent instruction.

19.6 Peripheral Devices

Peripheral devices include any hardware that you would attach to a computer, and that the computer would communicate with. This includes input devices that feed data in to the computer, such as a keyboard, mouse, and/or touchpad; output devices that output information from the computer like your display and sound card; and combination, input/output, devices that do both, such as a network card or modern printer (the printer might send status codes back to the computer – this counts as input). At one time, peripheral devices were frequently seen outside the computer case. As our technology increases, especially with the development of notebook computers, tablets and smartphones, peripheral devices are very often being enclosed inside the case.

For peripheral devices to be useful, data must be transferred between main memory and peripheral devices. One way to accomplish this transfer would be to use the CPU. For example, the CPU could fetch data from an input peripheral to be stored momentarily in a register; the CPU would then store the data from the register to main memory. The drawback here is that the data travels across the bus twice, existing only temporarily in a CPU register, and the CPU would not be able to do anything else while the data was transferred across the bus. This approach was used in early computer design, and fit quite nicely in the von Neumann architecture.

A better approach is to allow peripherals and main memory to communicate directly, leaving the CPU free to do other things while this is occurring. This is accomplished by an innovation called *direct memory access* (DMA). The idea here is that all peripherals have bus lines that connect them to a central *direct memory access controller*, which has a dedicated bus that connects it to main memory. The DMA controller is an integrated circuit like the CPU, but much simpler because it has to do one thing and one thing only: move data between peripherals and main memory. The CPU can initiate a *DMA transfer* for I/O between a peripheral and main memory by sending a signal (via the bus) to the DMA controller, which initiates and completes the data transfer. This approach results in a faster transfer of the data between the peripheral and main memory because it doesn't have to go through the CPU. Also, the CPU isn't forced to be idle while the data is transferred.

19.7 Concluding Remarks

In this chapter, we have explained some basic computer architecture concepts and hopefully given you a sense of how the computer executes programs, and the interaction between hardware and software. Much of what we have discussed has been greatly simplified and the reality in the computer is much more complicated and subtle. We don't require that you understand that additional

would first be used to copy the two numbers that are to be added from main memory into registers.

complexity and subtlety at this point, but we want you to know that there is much more to it, which you can and will study if you continue into second and third year computer science classes.