11

Computers & Computation

Home computers are being called upon to perform many new functions, including the consumption of homework formerly eaten by the dog.
Doug Larson

Computer Science is no more about computers than astronomy is about telescopes.
Edsger W. Dijkstra

What is the central core of computer science?My answer is simple -it is the art of programming a computer. It is the art of designing efficient and elegant methods of getting a computer to solve problems, theoretical or practical, small or large, simple or complex. It is the art of translating this design into an effective and accurate computer program.
C.A.R. Hoare

Opposite page: The XO Laptop
Photo courtesy of OLPC Project (www.olpc.org)

Today, there are more computers than people on most college campuses in the United States. The laptop computer shown on the previous page is the XO laptop developed by the One Laptop Per Child Project (OLPC). It is a low cost computer designed for kids. The OLPC project aims to reach out to over 2 billion children all over the world. They aim to get the XO computers in their hands to help in education. Their mission is to bring about a radical change in the global education system by way of computers. Of course a project with such a grandiose mission is not without controversy. It has had a bumpy ride since its inception. The technological issues were the least of their problems. They have had to convince governments (i.e. politicians) to buy-in into the mission. Several countries have agreed to buy-in and then reneged for all kinds of reasons. The laptops are not available for open purchase within the United States which has led to other socio-political controversies. Regardless, in the first year of its release the project aims to have over 1 million XO's in the hands of children in many developing countries. The thing about technology that has always been true is that a good idea is infective. Other players have emerged who are now developing ultra-low cost computers for kids. It will not be long before other competitive products will be available to all. The bigger questions are: What kind of change will this bring about in the world? How will these computers be used in teaching elementary and middle school children? Etc. You may also be wondering if your Scribbler robot can be controlled by the XO. It can.

You have been writing programs to control your robot through a computer. Along the way you have seen many different ways to control a robot and also ways of organizing your Python programs. Essentially you have been engaging in the activity commonly understood as *computer programming* or *computer-based problem solving*. We have deliberately refrained from using those terms because the general perception of these conjures up images of geeks with pocket-protectors and complex mathematical equations that seem to be avoidable or out of reach for most people. Hopefully you have discovered by now that solving problems using a computer can be quite exciting and engaging. Your robot may have been the real reason that motivated you into this activity and that was by design. We are confessing to you that robots were used to attract you to pay some attention to computing.

You have to agree, if you are reading this, that the ploy worked! But remember that the reason you are holding a robot of your own in your hand is also because of computers. Your robot itself is a computer. Whether it was a ploy or not you have assimilated many key ideas in computing and computer science. In this chapter, we will make some of these ideas more explicit and also give you a flavor for what *computer science* is really all about. As Dijkstra puts it, *computer science is no more about computers than astronomy is about telescopes.*

Computers are dumb

Say you are hosting an international exchange student in your home. Soon after her arrival you teach her the virtues of PB&J (Peanut Butter & Jelly) sandwiches. After keenly listening to you, her mouth starts to water and she politely asks you if you can share the recipe with her. You write it down on a piece of paper and hand it to her.

Do This: Go ahead; write down the recipe to make a PB&J sandwich.

Seriously, do try to write it down. We insist!

OK, now you have a recipe for making PB&J sandwiches.

Do This: Go ahead, use your recipe from above to make yourself a PB&J sandwich. Try to follow the instructions *exactly* as *literally* as you can. If you successfully managed to make a PB&J sandwich, congratulations! Go ahead and enjoy it. Do you think your friend will be able to follow your recipe and enjoy PB&J sandwiches?

You have to agree, writing a recipe for PB&J sandwiches seemed trivial at first, but when you sit down to write it you have no choice but to question several assumptions: does she know what peanut butter is? Should I recommend a specific brand? Ditto for jelly? By the way, did you forget to mention it was *grape jelly*? What kind of bread to use? Will it be pre-sliced? If not, you need a knife for slicing the loaf? Did you specify how thick the

slices should be? Should she use the same knife for spreading the peanut butter and the jelly? Etc. The thing is, at such a level of detail you can go on and on...does your friend know how to use a knife to spread butter or jelly on a slice? Suddenly, a seemingly trivial task becomes a daunting exercise. In reality, in writing down your recipe, you make several assumptions: she knows what a sandwich is, it involves slices of bread, spreading things on the slices, and slapping them together. There, you have a sandwich!

Think of the number of recipes that have been published in cookbooks all over the world. Most good cookbook authors start with a dish, write down a recipe, try it a number of times and refine it in different ways. Prior to publication, the recipe is tested by others. After all, the recipe is for others to follow and recreate a dish. The recipe is revised and adjusted based on feedback by recipe testers. The assumption that cookbook authors make is that you will have the competence to follow the recipe and recreate the dish to your satisfaction. It may never result in the same dish that the author prepared. But it will give you a base to improvise upon. Wouldn't it be nice if there was an exact way of following a recipe so that you end up with the exact same result as the cookbook author every time you made that dish? Would it? That may depend on your own tastes and preferences. Just for fun, here is a recipe for some yummy Saffron Chicken Kabobs.

Saffron Chicken Kabobs

Ingredients

1 lb boneless chicken breast, cubed into 1-2 inch pieces
1 medium onion, sliced
1 tsp saffron threads
1 lime
1 tbsp olive oil
Salt and black pepper to taste

Preparation

1. Mix the chicken and onions in a non-reactive bowl.
2. With your fingers crush and add saffron threads.
3. Add the juice of the lime, olive oil, and salt and pepper.
4. Marinade in the refrigerator for at least 30 min (or overnight).
5. Preheat a grill (or oven to 400 degrees).
6. Skewer kabobs, discarding the onion slices. Or place everything in a lined baking sheet if using oven.
7. Grill/bake for 12-15 min until done.

In cooking recipes, like the ones above, you can assume many things: they will be used by people (like you and me); they will be able to follow them; perhaps even improvise. For instance, in the recipe above, we do not specify that one will need a knife to cut the chicken, onions, or the lime; or that you will need a grill or an oven; etc. Most recipes assume that you will be able to *interpret* and follow the recipe as written.

Computer programs are also like recipes, to some extent. Think of the program you wrote for choreographing a robot dance, for instance. We have reproduced the version from Chapter 3 here:

```
# File: dance.py
# Purpose: A simple dance routine
# First import myro and connect to the robot

from myro import *
initialize("com5")

# Define the new functions...

def yoyo(speed, waitTime):
       forward(speed, waitTime)
       backward(speed, waitTime)
       stop()

def wiggle(speed, waitTime):
       motors(-speed, speed)
```

```
        wait(waitTime)
        motors(speed, -speed)
        wait(waitTime)
        stop()

# The main dance program
def main():
        print "Running the dance routine..."
        yoyo(0.5, 0.5)
        wiggle(0.5, 0.5)
        yoyo(1, 1)
        wiggle(1, 1)
        print "...Done"

main()
```

In many ways, this program above is like a recipe:

To do a robot dance

Ingredients

1 function `yoyo` for the robot to go back and forth at a given speed
1 function `wiggle` that enables the robot to wiggle at a given speed

Preparation

1. yoyo at speed 0.5, wait 0.5
2. wiggle at speed 0.5, wait 0.5
3. yoyo at speed 1, wait 1
4. wiggle at speed 1, wait 1

Further, you could similarly specify the steps involved in doing the `yoyo` and `wiggle` motions as a recipe. This may seem like a trivial example, but it makes two very important points: a computer program is like a recipe in that it lays out the list of ingredients and a method or steps for accomplishing the given task; and, like a recipe, its ingredients and the steps require careful pre-

planning and thought. Importantly, computer programs are different from recipes in one aspect: they are designed to be followed by a computer!

A computer is a dumb device designed to follow instructions/recipes. We will save the technical details of how a computer does what it does for a later course. But it is almost common knowledge that everything *inside* is represented as 0's and 1's. Starting from 0's and 1's one can design encoding schemes to represent numbers, letters of the alphabet, documents, images, movies, music, etc. and whatever other abstract entities you would like to manipulate using a computer. A computer program is ultimately also represented as a sequence of 0's and 1's and it is in this form that most computers like to follow recipes. However limiting or degenerate this might sound it is the key to the power of computers. Especially when you realize that it is this simplification that enables a computer to manipulate hundreds of millions of pieces of information every second. The price we have to pay for all this power is that we have to specify our recipes as computer programs in a rather formal and precise manner. So much so that there is no room for improvisation: no pinch of salt vagaries, as in cooking recipes, is acceptable. This is where *programming languages* come in. Computer scientists specify their computational recipes using *programming languages*. You have been using the programming language Python to write your robot programs. Other examples of programming languages are Java, C++ (pron.: sea plus plus), C# (pron.: sea sharp), etc. There are well over 2000 programming languages in existence!

Do This: Can you find out how many programming languages there are? What are the ten most commonly used programming languages?

Prior to the existence of programming languages computers were programmed using long sequences of 0's and 1's. Needless to say it drove several people crazy! Programming languages, like Python, enable a friendlier way for programmers to write programs. Programming languages provide easy access to encodings that represent the kinds of things we, humans, relate to. For example, the Python statement:

```
meaningOfLife = 42
```

is a command for the computer to associate the value, 42 with the name `meaningOfLife`. This way, we can ask the computer to check that it is indeed 42:

```
if meaningOfLife == 42:
     speak("Eureka!")
else:
     speak("What do we do now?")
```

Once again, it would be good to remind you that the choice of the name, `meaningOfLife`, doesn't really mean that we are talking about *the meaning of life*. We could as well have called it `timbuktoo`, as in:

```
timbuktoo = 42
```

You see, computers are truly dumb!

It is really up to us, the programmer, to ensure that we use our names consistently and choose them, in the first place, carefully. But, by creating a language like Python, we have created a formal notation so that when translated into 0's and 1's each statement will mean only one thing, no other interpretations. This makes them different from a cooking recipe.

Robot goes to buy fresh eggs

Recipes, however, form a good conceptual basis for starting to think about a program to solve a problem. Say, you have in mind to make your favorite *Apple Strudel*. You know you will need apples. Perhaps it is the apple season that prompted the thought in the first place. You will also need pastry. But when you get down to it, you will need that recipe you got from your grandma.

Whenever we are asked to solve a problem using a computer, we begin by laying out a rough plan for solving the problem. That is, sketch out a strategy. This is further refined into specific steps, perhaps even some variables are

identified and named, etc. Once you convince yourself that you have a way of solving the problem, what you have is an *algorithm*

The idea of an algorithm is central to computer science so we will spend some time here developing this notion. Perhaps the best way to relate to it is by an example. Assume that a robot goes into a grocery store to buy a dozen fresh eggs. Assuming it is capable of doing this, how will it ensure that it has selected the freshest eggs available?

Your personal robot is probably not up to this kind of task but imagine that it was. Better yet, leave the mechanics aside, let us figure out how *you* would go and buy the freshest eggs. Well, you would somehow need to know what today's date is. Assume it is September 15, 2007 (why this date? it'll become clear soon!). Now you also know that egg cartons typically carry a freshness date on them. In fact, USDA (the United States Department of Agriculture) offers voluntary, no cost, certification programs for egg farms. An egg farmer can volunteer to participate in USDA's egg certification program whereby the USDA does regular inspections and also provides help in categorizing eggs by various sizes. For example, eggs are generally classified as Grade AA, Grade A, or Grade B. Most grocery stores carry Grade A eggs. They can also come in various sizes: Extra Large, Large, Small, etc. What is more interesting is that the carton labeling system has some very useful information encoded on it.

Egg Carton Labeling

Every USDA certified egg carton has at least three pieces of information (see picture on the right): a "sell by" date (or a "use by date" or a

"best by" date), a code identifying the specific farm the eggs came from, and a date on which the eggs were packed in that carton. Most people buy eggs by looking at the "sell by" date or the "best by" date. However the freshness information is really encoded in the packed on date. To make things more confusing, this date is encoded as the day of the year.

For example, take a look at the top carton shown on the previous page. Its "sell by" date is October 4. "P1107" is the farm code. This carton was packed on the 248th day of the year. Further, USDA requires that all certified eggs be packed within 7 days of being laid. Thus, the eggs in the top carton were laid somewhere between day 241 and day 248 of 2007. What dates correspond to those dates?

Next, look at the bottom carton. Those eggs have a later "sell by" date (October 18) but an earlier packed date: 233. That is those eggs were laid somewhere between day 226 and day 233 of 2007.

Which eggs are fresher?

Even though the "sell by" date on the second carton is two weeks later, the first carton contains fresher eggs. In fact, the eggs in the upper carton were laid at least two weeks later!

The packed on date is encoded as a 3-digit number. Thus eggs packed on January 1 will be labeled: 001; eggs packed on December 31, 2007 will be labeled: 365.

Do This: Go to the USDA web site (www.usda.gov) and see if you can find out which farm the two eggs cartons came from.

For a robot, the problem of buying the freshest eggs becomes that of figuring out, given a packed on date, what the date was when the eggs were packed?

Fasten your seatbelts, we are about to embark on a unique computational voyage...

Designing an algorithm

So far, we have narrowed the problem down to the following specifications:

Input
3-digit packed on date encoding

Output
Date the eggs were packed

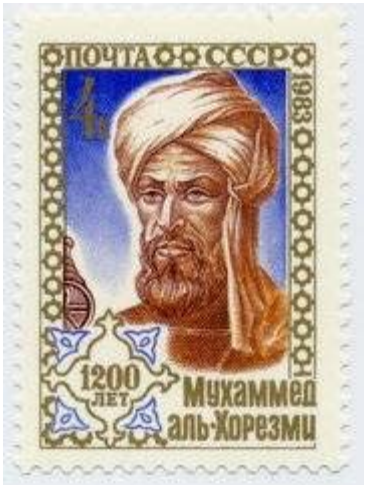
For example, if the packed on date was encoded as 248, what will be the actual date?

Well, that depends. It could be September 4 or September 5 depending on whether the year was a leap year or not. Thus, it turns out, that the problem above also requires that we know which year we were talking about. Working out one or two sample problems is always a good idea because it helps identify missing information that may be critical to solving the problem. Given that we do need to know the year, we can ask the user to enter that at the same time the 3-digit code is entered. The problem specification then becomes:

Input
3-digit packed on date encoding
Current year

The Etymology of Algorithm

The word algorithm, an anagram of logarithm, is believed to have been derived from Al-Khowarizmi, a mathematician who lived from 780-850 AD. His full name was Abu Ja'far Muḥammad ibn Mūsā al-Khwārizmī, (Mohammad, father of Jafar, son of Moses, a Khwarizmian). Much of the mathematical knowledge of medieval Europe was derived from Latin translations of his works.



In 1983, The Soviet Union issued the stamp shown above in honor of his 1200th anniversary.

Output

Date the eggs were packed

Example:

Input: 248, 2007

Output: The eggs were packed on September 5, 2007

Any ideas as to how you would solve this problem? It always helps to try and do it yourself, with pencil and paper. Take the example above, and see how you would arrive at the output date. While you are working it out, try to write down your problem solving process. Your algorithm or recipe will be very similar.

Suppose we are trying to decode the input 248, 2007. If you were to do this by hand, using a pen and paper, the process might go something like this:

```
The date is not in January because it has 31 days and 248 is
much larger than 31.
Lets us subtract 31 out of 248: 248 - 31 = 217

217 is also larger than 28, the number of days in February,
2007.
So, let us subtract 28 from 217: 217 - 28 = 189

189 is larger than 31, the number of days in March.
Subtract 31 from 189: 189 - 31 = 158

158 is larger than 30, the number of days in April.
So: 158 - 30 = 128

128 is larger than 31, the number of days in May.
Hence: 128 - 31 = 97

97 is larger than 30, the number of days in June.
97 - 30 = 67

67 is larger than 31, the number of days in July.
67 - 31 = 36
```

```
36 is larger than the number of days in August (31).
36 - 31 = 5

5 is smaller than the number of days in September.
Therefore it must be the 5th day of September.

The answer is: 248th day of 2007 is September 5, 2007.
```

That was obviously too repetitious and tedious. But that is where computers come in. Take a look at the process above and see if there is a pattern to the steps performed. Sometimes, it is helpful to try another example.

Do This: Suppose the input day and year are: 56, 2007. What is the date?

When you look at the sample computations you have performed, you will see many patterns. Identifying these is the key to designing an algorithm. Sometimes, in order to make this easier, it is helpful to identify or name the key pieces of information being manipulated. Generally, this begins with the inputs and outputs identified in the problem specification. For example, in this problem, the inputs are: day of the year, current year. Begin by assigning these values to specific variable names. That is, let us assign the name `day` to represent the day of the year (248 in this example), and `year` as the name to store the current year (2007). Notice that we didn't choose to name any of these variables `timbuktu` or `meaningOfLife`!

Also, notice that you have to repeatedly subtract the number of days in a month, starting from January. Let us assign a variable named, `month` to keep track of the month under consideration.

Next, you can substitute the names `day` and `year` in the sample computation:

```
Input:
   day = 248
   year = 2007

# Start by considering January
month = 1
```

```
The date is not in month = 1 because it has 31 days and 248 is
much larger than 31.
day = day - 31

# next month
month = 2

day (= 217) is also larger than 28, the # of days in month = 2
day = day - 28

# next month
month = 3
day (= 189) is larger than 31, the # of days in month = 3.
day = day - 31

# next month
month = 4
day (= 158) is larger than 30, the # of days in month = 4.
day = day - 30

# next month
month = 5
day (= 128) is larger than 31, the # of days in month = 5.
day = day - 31

# next month
month = 6
day (= 97) is larger than 30, the # of days in month = 6.
day = day = 30

# next month
month = 7
day (= 67) is larger than 31, the # of days in month = 7.
day = day - 31

# next month
month = 8
day (= 36) is larger than the # of days in month = 8.
day = day - 31

# next month
month = 9
day (= 5) is smaller than the # of days in month = 9.
```

```
Therefore it must be the 5th day of September.

The answer is: 9/5/2007
```

Notice now how repetitious the above process is. The repetition can be expressed more concisely as shown below:

```
Input:
   day
   year

# start with month = 1, for January
month = 1
repeat
   if day is less than number of days in month
      day = day – number of days in month
      # next month
      month = month + 1
   else
      done

Output: day/month/year
```

It is now starting to look like a recipe or an algorithm. Go ahead and try it with the sample inputs from above and ensure that you get correct results. Additionally, make sure that this algorithm will work for boundary cases: 001, 365.

Thirty days hath September

We can refine the algorithm above further: one thing we have left unspecified above is the computation of the number of days in a month. This information has to be made explicit for a computer to be able to follow the recipe. So, how do we compute the number of days in a month? The answer may seem simple. Many of you may remember the following poem:

Thirty days hath September
April, June, and November
All the rest have thirty-one
Except for February alone
Which hath twenty-eight days clear
And twenty-nine in each leap year

From a design perspective, we can assume that we have an ingredient, a function in this case, called `daysInMonth` that, given a month and a year will compute and return the number of days in the month. That is, we can refine our algorithm above to the following:

```
Ingredients:
   1 function daysInMotnh(m, y): returns the number of days in
                                 month, m in year y.

Input:
   day
   year

# start with month = 1, for January
month = 1
repeat
   if day is less than number of days in month
      day = day - daysInMonth(month, year)
      # next month
      month = month + 1
   else
      done

Output: day/month/year
```

Now, we do have to solve the secondary problem:

Input
   month, M
   year, Y

Output
   Number of days in month, M in year, Y

On the surface this seems easy, the poem above specifies that April, June, September, and November have 30 days, and the rest, with the exception of February have 31. February has 28 or 29 days depending upon whether it falls in a leap year or not. Thus, we easily elaborate a recipe or an algorithm for this as follows:

```
Input:
   m, y

if m is April (4), June(6), September(9), or November (11)
   days = 30
else if m is February
         if y is a leap year
              days = 29
         else
              days = 28
else
   (m is January, March, May, July, August, October, December)
    days = 31

Output:
   days
```

This still leaves out one more detail: how do we tell if `y` is a leap year?

First, try and answer the question, *what is a leap year?*

Again, we can refine the algorithm above by assuming that we have another ingredient, a function: `leapYear`, that determines if a given year is a leap year or not. Then we can write the algorithm above as:

```
Ingredients:
   1 function leapYear(y)
     returns True if y is a leap year, false otherwise
```

```
Input:
   m, y

if m is April (4), June(6), September(9), or November (11)
   days = 30
else if m is February
         if leapYear(y)
              days = 29
         else
              days = 28
else
   (m is January, March, May, July, August, October, December)
    days = 31

Output:
   days
```

Most of us have been taught that a leap year is a year that is divisible by 4. That is the year 2007 is not a leap year, since 2007 is not divisible by 4, but 2008 is a leap year, since it is divisible by 4.

Do This: How do you determine if something is divisible by 4? Try your solution on the year 1996, 2000, 1900, 2006.

Leap Years: Papal Bull

To design a recipe or an algorithm that determines if a number corresponding to a year is a leap year or not is straightforward if you accept the definition from the last section. Thus, we can write:

Input

y, a year

Output

True if y is a leap year, false otherwise

Method

```
 if y is divisible by 4
     it is a leap year, or True
  else
     it is not a leap year, or False
```

However, this is not the complete story. The western calendar that we follow is called the *Gregorian Calendar* which was adopted in 1582 by a Papal Bull issued by Pope Gregory XIII. The Gregorian Calendar defines a leap year, by adding an extra day, every fourth year. However, there is a 100-year correction applied to it that makes the situation a little more complicated: Century years are not leap years except when they are divisible by 400. That is the years 1700, 1800, 1900, 2100 are not leap years even though they are divisible by 4. However, the years 1600, 2000, 2400 are leap years. For more information on this, see the exercises at the end of the chapter. Our algorithm for determining if a year is a leap year can be refined as shown below:

```
Input
   y, a year

   if y is divisible by 400
      it is a leap year, or True
   else if y is divisible by 100
      it is not a leap year, or False
   else if y is divisible by 4
      it is a leap year, or True
   else
      it is not a leap year, or False
```

Finally, we have managed to design all the algorithms or recipes required to solve the problem. You may have noticed that we used some familiar constructs to elaborate our recipes or algorithms. Next, let us take a quick look at the essential constructs that are used in expressing algorithms.

Essential components of an algorithm

Computer scientists express solutions to problems in terms of algorithms, which are basically more detailed recipes. Algorithms can be used to express any solution and yet are comprised of some very basic elements:

1. Algorithms are step-by-step recipes that clearly identify the inputs and outputs
2. Algorithms name the entities that are manipulated or used: variables, functions, etc.
3. Steps in the algorithm are followed in the order they are written (from top to bottom)
4. Some steps can specify decisions (if-then) over the choice of some steps
5. Some steps can specify repetitions (loops) of steps
6. All of the above can be combined in any fashion.

Computer scientists claim that solutions/algorithms to *any* problem can be expressed using the above constructs. You do not need any more! This is a powerful idea and it is what makes computers so versatile. From a larger perspective, if this is true, then these can be used as tools for thinking about any problem in the universe. We will return to this later in the chapter.

Programming Languages

Additionally, as you have seen earlier, in writing Python programs, programming languages (Python, for example) provide formal ways of specifying the essential components of algorithms. For example, the Python language provides a way for you to associate values to variables that you name, it provides a sequential way of encoding the steps, it provides the if-then conditional statements, and also provides the while-loop and for-loop constructs for expressing repetitions. Python also provides means for defining functions and also ways of organizing groups of related functions into libraries or modules which you can import and use as needed. As an example,

we provide below, the Python program that encodes the `leapYear` algorithm shown above:

```
def leapYear(y):
   '''Returns true if y is a leap year, false otherwise.'''

   if y %400 == 0:
      return True
   elif y % 100 == 0:
      return False
   elif y % 4 == 0:
      return True
   else:
      return False
```

The same algorithm, when expressed in C++ (or Java) will look like this:

```
bool leapYear(int y) {
   // Returns true if y is a leap year, false otherwise.
   if (y % 400 == 0)
      return true
   else if (y % 100 == 0)
      return false
   else if (y % 4 == 0)
      return true
   else
      return false
}
```

As you can see, there are definite syntactic variations among programming languages. But, at least in the above examples, the coding of the same algorithm looks very similar. Just to give a different flavor, here is the same function expressed in the programming language CommonLisp.

```
(defun leapYear (y)
   (cond
     ((zerop (mod y 400)) t)
     ((zerop (mod y 100)) nil)
     ((zerop (mod y 4)) t)
     (t nil)))
```

Again, this may look weird, but it is still expressing the same algorithm.

What is more interesting is that given an algorithm, there can be many ways to encode it, even in the same programming language. For example, here is another way to write the `leapYear` function in Python:

```
def leapYear(y):
   '''Returns true if y is a leap year, false otherwise.'''

   if ((y % 4 == 0) and (y % 100 != 0)) or (y % 400 == 0):
      return True
   else:
      return False
```

Again, this is the same exact algorithm. However, it combines all the tests into a single condition: y is divisible by 4 or by 400 but not by 400. The same condition can be used to write an even more succinct version:

```
def leapYear(y):
   '''Returns true if y is a leap year, false otherwise.'''

  return ((y % 4 == 0) and (y % 100 != 0)) or (y % 400 == 0)
```

That is, return whatever the result is (`True`/`False`) of the test for `y` being a leap year. In a way, expressing algorithms into a program is much like expressing a thought or a set of ideas in a paragraph or a narrative. There can be many ways of encoding an algorithm in a programming language. Some seem more natural, and some more poetic, or both, and, like in writing, some can be downright obfuscated. As in good writing, good programming ability comes from practice and, more importantly, learning from reading well written programs.

From algorithms to a working program

To be able to solve the fresh eggs problem, you have to encode all the algorithms into Python functions and then put them together as a working program. Below, we present one version:

```
# File: fresheggs.py

def leapYear(y):
   '''Returns true if y is a leap year, false otherwise.'''

   return ((y % 4 == 0) and (y % 100 != 0)) or (y % 400 == 0)

def daysInMonth(m, y):
   '''Returns the number of days in month, m (1-12)
      in year, y.'''

   if (m == 4) or (m == 6) or (m == 9) or (m == 11):
      return 30
   elif m == 2:
      if leapYear(y):
         return 29
      else:
         return 28
   else:
      return 31

def main():
   '''Given a day of the year (e.g. 248, 2007),
      convert it to the date (i.e. 9/5/2007)'''

   #Input: day, year
   day, year = input("Enter the day, year: ")

   # start with month = 1, for January
   month = 1

   while day > daysInMonth(month, year):
      day = day - daysInMonth(month, year)

      # next month
      month = month + 1

   # done, Output: month/day/year
   print "The date is: %1d/%1d/%4d" % (month, day, year)

main()
```

If you save this program in a file, `fresheggs.py`, you will be able to run it and test it for various dates. Go ahead and do this. Here are some sample outputs:

```
Enter the day, year: 248, 2007
The date is: 9/5/2007

>>> main()
Enter the day, year: 12, 2007
The date is: 1/12/2007

>>> main()
Enter the day, year: 248, 2008
The date is: 9/4/2008

>>> main()
Enter the day, year: 365, 2007
The date is: 12/31/2007

>>> main()
Enter the day, year: 31, 2007
The date is: 1/31/2007
```

All seems to be good. Notice how we tested the program for different input values to confirm that our program is producing correct results. It is very important to test your program for a varied set of input, taking care to include all the *boundary* conditions: first and last day of the year, month, etc. Testing programs is a fine art in itself and several books have been written about the topic. One has to ensure that all possible inputs are tested to ensure that the behavior of the program is acceptable and correct. You did this with your robot programs by repeatedly running the program and observing the robot's behavior. Same applies to computation.

Testing and Error Checking

What happens, if the above program receives inputs that are outside the range? What if the user enters the values backwards (e.g. 2007, 248 instead of 248, 2007)? What if the user enters her name instead (e.g. Paris, Hilton)? Now

is the time to try all this out. Go ahead and run the program and observe its behavior on some of these inputs.

Ensuring that a program provides acceptable results for all inputs is critical in most applications. While there is no way to avoid what happens when a user enters his name instead of entering a day and a year, you should still be able to safeguard your programs from such situations. For example:

```
>>> main()
Enter the day, year: 400, 2007
That corresponds to the date: 14/4/2007
```

Obviously, we do not have a month numbered 14!

The thing that comes to rescue here is the realization that, it is your program and the computer will only carry out what you have expressed in the program. That is, you can include *error checking* facilities in your program to account for such conditions. In this case, any input value for day that is outside the range 1..365 (or 1..366 for leap years) will not be acceptable. Additionally, you can also ensure that the program only accepts years greater than 1582 for the second input value. Here is the modified program (we'll only show the `main` function):

```
def main():
   '''Given a day of the year (e.g. 248, 2007), convert
      it to the date (i.e. 9/5/2007)'''

   #Input: day, year
   day, year = input("Enter the day, year: ")

   # Validate input values...
   if year <= 1582:
       print "I'm sorry. You must enter a valid year
              (one after 1582). Please try again."
       return
   if day < 1:
       print "I'm sorry. You must enter a valid day
             (1..365/366). Please try again."
```

```
        return
    if leapYear(year):
        if day > 366:
            print "I'm sorry. You must enter a valid day
                  (1..365/366). Please try again."
            return
    elif day > 365:
        print "I'm sorry. You must enter a valid day
              (1..365/366). Please try again."
        return

    # input values are safe, proceed...
    # start with month = 1, for January
    month = 1

    while day > daysInMonth(month, year):
       day = day - daysInMonth(month, year)

       # next month
       month = month + 1

    # done, Output: month/day/year
    print "The date is: %1d/%1d/%4d" % (month, day, year)

main()
```

Here are the results of some more tests on the above program.

```
Enter the day, year: 248, 2007
The date is: 9/5/2007

>>> main()
Enter the day, year: 0, 2007
I'm sorry. You must enter a valid day (1..365/366). Please try
again.

>>> main()
Enter the day, year: 366, 2007
I'm sorry. You must enter a valid day (1..365/366). Please try
again.

>>> main()
Enter the day, year: 400, 2007
```

```
I'm sorry. You must enter a valid day (1..365/366). Please try
again.

>>> main()
Enter the day, year: 248, 1492
I'm sorry. You must enter a valid year (one after 1582).
Please try again.

>>> main()
Enter the day, year: 366, 2008
The date is: 12/31/2008
```

Starting from a problem description it is a long and carefully planned journey that involves the development of the algorithm, the encoding of the algorithm in a program, and finally testing and improving the program. In the end you are rewarded not just by a useful program, you have also honed your general problem solving skills. Programming forces you to anticipate unexpected situations and to account for them prior to encountering them which itself can be a wonderful life lesson.

Modules to organize components

Often, in the course of designing a program, you end up designing components or functions that can be used in many other situations. For example, in the problem above, we wrote functions `leapYear` and `daysInMonth` to assist in solving the problem. You will no doubt agree that there are many situations where these two functions could come in handy (see Exercises below). Python provides the *module* facility to help organize related useful functions into a single file that you can then use over and over whenever they are needed. For example, you can take the definitions of the two functions and put them separately in a file called, `calendar.py`. Then, you can *import* these functions whenever you need them. You have used the Python `import` statement to import functionality from several different modules: `myro`, `random`, etc. Well, now you know how to create your own. Once you create the `calendar.py` file, you can import it in the `fresheggs.py` program as shown below:

```
from calendar import *

def main():
   '''Given a day of the year (e.g. 248, 2007), convert
      it to the date (i.e. 9/5/2007)'''

   #Input: day, year
   day, year = input("Enter the day, year: ")

   # Validate input values...
   if year <= 1582:
       print "I'm sorry. You must enter a valid year
             (one after 1582). Please try again."
       return
   if day < 1:
       print "I'm sorry. You must enter a valid day
             (1..365/366). Please try again."
       return
   if leapYear(year):
       if day > 366:
           print "I'm sorry. You must enter a valid day
                 (1..365/366). Please try again."
           return
   elif day > 365:
       print "I'm sorry. You must enter a valid day
             (1..365/366). Please try again."
       return
   # input values are safe, proceed...
   # start with month = 1, for January
   month = 1

   while day > daysInMonth(month, year):
      day = day - daysInMonth(month, year)

      # next month
      month = month + 1

   # done, Output: month/day/year
   print "The date is: %1d/%1d/%4d" % (month, day, year)

main()
```

Homer contemplates P = NP?

The second equation on the right is Euler's Equation.

It may have also occurred to you by now that for any given problem there may be many different solutions or algorithms. In the presence of several alternative algorithms how do you decide which one to choose? Computer Scientists have made it their primary business to develop, analyze, and classify algorithms to help make these decisions. The decision could be based on ease of programming, efficiency, or the number of resources it takes for a given algorithm. This has also led computer scientists to create a classification of problems: from easy to hard, but in a more formal sense. Some of the hardest open questions in the realm of problems and computing lie in this domain of research. We will not go into details here, but these questions have even shown up in several popular TV shows.We will attempt to give you a flavor of this next.

Space & Time Complexity

Let us start with another problem: You have to travel from Washington State to Washington DC in the United States of America. To make things interesting, lets us add a restriction that you can only travel through states whose names begin with the letters in the word "woman". That is, it is OK to go from Washington to Oregon since both "W" and "O" are in the word "woman" but it is not OK to go from Washington to California. Is this feasible? If it is, how many ways are possible? Which one goes through the least/most number of states? Etc.

If you are thinking about how to solve this, you have to rely on your geographic knowledge of the United States. Alternately, you can Google a

state map and then figure out a solution. But, in doing so, you have stumbled upon the two key ingredients of computing: data & algorithm.

Data gives you a representation of the relevant information which, in this case is a way of knowing which states adjoin which other states. The algorithm gives you a way of finding a solution: Start in Washington, look at its neighbors. Pick a neighbor whose name satisfies the constraint, then look at that state's neighbors, and so on. Additionally, an algorithm also forces you to make certain choices: if there is more than one eligible neighbor, which one do you pick? What if you end up in a dead end, how do you go back to the choices you left behind to explore alternative paths? Etc. Depending on these decisions, you are likely to end up with many different algorithms: one that may suggest exploring all alternatives simultaneously; or one that forces you to choose, only to return to other choices in case of failure. Each of these will further impact on the amount of data you will need to store to keep track of your progress.

Developing computer programs to solve any problem requires one to design data representations and to choose among a set of alternative algorithms. Computer scientists characterize choices of data representations and algorithms abstractly in terms of the computer resources of *space* and *time* needed to implement those choices. Solutions vary in terms of the amount of space (or computer memory) and time (seconds, minutes, hours, days, years) required on a computer. Here are some examples:

- To compute the product of two numbers requires constant time: to a computer there is no difference between multiplying 5 by 2 or 5,564,198 by 9,342,100. These are called *constant time* algorithms.
- To find a number in a list of *N* unordered numbers takes time proportional to *N*, especially if the number you are looking for is not in there. These are called *linear time* algorithms.
- To find a number in a list of N ordered numbers (say, in ascending order) requires at most $\log_2 N$time. How? Such algorithms are called *logarithmic time* algorithms.

- To transform a *N*x*N* pixel camera image by manipulating all its pixels takes time proportional to N^2 time. These are called *quadratic algorithms*.
- To find a path from a state to another, in a map of N states, given certain constraints, can take time proportional to b^d where b is the average number of neighbors of each state and d is the number of states that make up the solution. In general, many problems fall into the category N^k where N is the size of the problem. These are called *polynomial time* algorithms.
- There are also several unsolvable problems in the world.

In Chapter 9, when doing image transformations, we restricted ourselves to fairly small sized images. You may have noticed that the larger the image, the longer it takes for the transformation. This has to do with the speed of the computer you are using as well as the speed of the programming language implementation. For example, a computer running at 4GHz speed is capable of doing approximately 1 billion arithmetic operations in one second (or 10^9 operations/second). A program written in the C programming language might be able to give you a speed of ½ billion operations per second on the same computer. This is due to extra operations required to implement the C language and additional tasks the operating system is carrying out in the background. Python programs run approximately 20 times slower that C programs. That is, a Python program running on the same computer might give you 25 million operations per second at best. A typical transformation, say computing the negative, of an image of size `WxH` pixels would require the following loop:

```
for x in range(W)
   for y in range(H):
      pixel = getPixel(myPic, x, y)
      r, g, b = getRGB(pixel)
      setRGB(pixel, (255-r,255-g,255-b))
```

If the image is 1000x1000 pixels (i.e. W=1000 and H=1000), each of the three statements in the loop is executed 1 million times. Each of those statements in turn requires an average of 8-16 low-level computing operations: the actual

calculations, plus calculations to locate pixels in memory, etc. Thus, the transformation above would require over 24-48 million operations. As you can imagine, it will take a few seconds to complete that task.

In the computing industry, computing speeds are classified based on official benchmark computations that calculate speeds in terms of the number of floating-point operations per second, or *flops*. A typical laptop or a desktop these days is capable of delivering speeds between ½ to 1 Gflops (Giga flops). The world's fastest computer can deliver computing speeds as fast as 500 Tflops (Terra flops). That is, it is about a million times faster (and costs many millions to make as well). However, if you stop and think about the Chess playing program we mentioned in Chapter 10 that would require approximately 10^{65} operations before making a single move, even the world's fastest computer is going to take gazillion years to finish that computation! Such a problem would be considered *uncomputable*.

Computer scientists have developed an elaborate vocabulary for discussing problems and classifying them as *solvable* or *unsolvable*, *computable* or *uncomputable*, based on whether there are known models to solve a given problem and whether the models are solvable, computable, etc. There are also hierarchies of problem solutions, from simple to hard; constant time to polynomial time and longer; and equivalence classes implying the same algorithm can solve all problems in a an equivalence class, etc. It is indeed amazing to conceptualize an algorithm that is capable of solving many unrelated problems! For example, the algorithms that optimize shipping and delivery routes can also be used in determining protein folding structures for DNA molecules. This is what makes computer science intellectually interesting and exciting.

Summary

In this chapter we have tied together many fundamental ideas in computing and computer science. While our journey started in Chapter 1 with playing with personal robots, we have, in the process, acquired a wealth of fundamental concepts in computing. As you can see, computing is a rich,

diverse, and deeply intellectual discipline of study that has implications for all aspects of our lives. We started this chapter by pointing out that there are now more computers on a typical college campus in the United States than the number of people. It probably will not be long before there are more computers than people on this entire planet. Yet, the idea of an *algorithm* that is central to computing is barely understood by most users despite its simple and intuitive constituents. Many computer scientists believe that we are still sitting at the dawn of *the age of algorithm* and that there are much bigger intellectual and societal benefits yet to be realized, especially if more people were aware of these ideas.

Myro review

No new Myro features were introduced in this chapter.

Python Review

The only new Python feature introduced in this chapter was the creation of modules. Every program you create can be used as a library module from which you can import useful facilities.

Exercises

1. To compute the number of days in a month, we used the following:

```
def daysInMonth(m, y):
   '''Returns the #of days in month, m (1-12) in year, y.'''

   if (m == 4) or (m == 6) or (m == 9) or (m == 11):
      return 30
   elif m == 2:
      if leapYear(y):
         return 29
      else:
         return 28
   else:
      return 31
```

You can further simplify the writing of the condition in the first if-statement by using lists:

```
if m in [4, 6, 9, 11]:
   return 30
…
```

Rewrite the function to use the above condition.

2. Define a function called `valid(day, month, year)` so that it returns true if the `day`, `month`, and `year` conform to a valid date as defined in this chapter. Use the function to rewrite the program as follows:

```
def main():
   '''Given a day of the year (e.g. 248, 2007), convert
      it to the date (i.e. 9/5/2007)'''

   #Input: day, year
   day, year = input("Enter the day, year: ")

   # Validate input values...
   if not valid(day, month, year):
      print “Please enter a valid date.”

   # input values are safe, proceed...
   # start with month = 1, for January
   month = 1

   while day > daysInMonth(month, year):
      day = day - daysInMonth(month, year)

      # next month
      month = month + 1

   # done, Output: month/day/year
   print "The date is: %1d/%1d/%4d" % (month, day, year)

main()
```

Rewrite the program as shown above to use the new function. Besides developing a correct algorithm it is also important to write programs in a way that makes them more readable for you.

3. Find out the how fast your computer is by noting the clock speed of the CPU. Based on that estimate how many arithmetic operations it will be able to perform in 1 second. Write a program to see how many operations it actually performs in one second.

4. Do a web search for “Chazelle age of algorithm”. You will be rewarded with a link to an essay written by Prof. Chazelle. Read the essay and write a short commentary on it.

5. What is the fastest computer in use today? Can you find out? How fast is it compared to the computer you use? 100 times? 1000 time? 100,000 times?

6. What is/was the “Y2K” problem?