Lecture Notes on An Introduction to Computing

Curtis Busby-Earle

June 2021

The material contained within was written and created by Curtis Busby-Earle. This includes all of the Python scripts, discourse, images, figures and tables but excludes table 1.1, figure 6.2, the cover art and page layout, all of which are freely available for use and redistribution with no attribution required. Figure 6.2 was however retrieved from Pixabay at https://pixabay.com/vectors/bank-queue-person-standing-atm-3527570/ and the cover art and page layout were retrieved from Hubspot at https://offers.hubspot.com/ebook-templates?hubs_post-cta=author

Editor: Fiona Porter-Lawson

This version was completed in June 2021

About this document

The aim of this work is to document the experience I have gained over more than a decade of what has worked best for me in delivering the introduction to computing courses. It is my hope that doing this will assist students with developing their own understanding of the material. It uses my examples that are worked through and provides code that can be directly copied and executed. Although the Python programming language is used to provide actual solutions, the problem descriptions and discussions are largely independent of that language.

Our courses follow international curriculum recommendations of the world's governing body for computing. The pair of courses that I have taught and the personal notes from which this material was created is based on the ACM/IEEE curriculum guidelines for computer science undergraduate degrees.

The content does not follow the sequence of delivery of the topics, nor does it include many exercises. These are to be followed as delivered in the courses. Use this document as a supplement towards understanding what you are learning.

Curtis Busby-Earle
PhD, Computer Science

TABLE OF CONTENTS

Chapter One

Communication and language

Language, the principal method of human communication, consisting of words used in a structured and conventional way and conveyed by speech, writing, or gesture.

Computing is all about problem solving. As computer scientists we are more interested in how to do something and less about what something is. The how-to knowledge is called imperative while the what-is is called declarative knowledge. For example, although we as computer scientists are interested in knowing what a square root of a number is (declarative), we are more interested in knowing how to calculate the square root of a number (imperative).

Computer scientists are human beings first. The ability to communicate is fundamentally important to us humans. We communicate to express among other things: how we feel, what we need and want, our thoughts, ideas and opinions. We also communicate to discuss our problems and their solutions, if at least one exists.

We communicate with language, and there are a great many. Languages have existed, evolved and died over the many centuries since societies have existed. Aramaic, Latin, Swahili, French, Mandarin, Brazilian Portuguese, Jamaican Patois, Trinidadian Creole, American Sign language are all examples of languages. Languages consist of symbols that represent something. In the case of spoken languages (as opposed to signed languages) that something is typically a subset of the sounds we can produce.

Our language is English and it includes symbols that represent sounds that we make: the symbol “a” represents the sounds “ah” and “aye”. The collection of the symbols is called an alphabet. In English the alphabet consists of the symbols we call letters and there are 26 of them. We combine letters to form words, words to form phrases and sentences, phrases and sentences to form paragraphs. The set of rules that govern how the symbols, words and phrases are to be combined so that they convey meaning (semantics) is called the syntax of the language.

In English if we combine the letters a,i,l and r in different ways we may have words that mean something or we may not: rail and lair have meaning (semantics) but ialr does not. An example of a syntax rule in English is that in a simple sentence a verb must follow a noun, 'the girl ran' conveys meaning whereas 'the ran girl' does not.

When tasked with communicating with someone who speaks a different language we have a few options. We may learn the other language, we may utilize the service of one who speaks both languages and who can therefore interpret from one language to the other, or we can now use an electronic interpreter such as Google Translate to assist. All involve taking a word or phrase in one language and translating it into the other. For example 'hello' translated into French is 'bonjour' and in Spanish is 'hola'. Programming languages have similarities and differences just like many other spoken and visual languages.

We use programming languages to communicate with a computer. Communication with a computer is primarily to instruct it on what we want it to do, and when we provide instructions to a computer, we must be *very specific*. Programming languages, like others that were mentioned earlier, consist of symbols, have a syntax and thereby able to convey some semantics. These are similarities with other languages. Recall that we combine symbols to form words, words to form sentences, and sentences to form paragraphs. Analogously, in programming languages, we typically combine symbols to form words, words to form expressions and statements, and expressions and statements to form functions and procedures.

Programming languages are not however, understood by computers and must therefore be translated. Unlike other languages that are translated into another, such as English to French as we discussed earlier, programming languages are translated into a numerical notation to enable a computer to understand what we want it to do. For all classical computers[1] that numerical notation is the binary system. Each programming language symbol has a unique numerical representation. The set of symbols and their numerical representations are then used to communicate.

1. Any computer that is not a quantum computer is a classical one. As of the writing of this book quantum computers only exist in research labs and wealthy companies. A few are publicly available for limited use via the Internet. All commercially available computers are therefore classical: smartphones, tablets, notebooks, desktops, mainframes, supercomputers etc.

For example the American Standard Code for Information Interchange (ASCII) is provided in Table 1.1. The ASCII values are the decimal equivalent of the binary representation of the characters. We want to instruct the computer on what we want it to do and that is to execute a solution to a problem. We use the programming language to write a solution to a problem, and that representation of a solution can then be executed by a computer.

Now that we know the purpose of programming languages and essentially how they work we can shift our focus to creating solutions to problems and expressing the solutions in such a way that the problems can be repeatedly solved. This is the essence of computing. We call the expression of a solution to a particular problem an algorithm. More specifically, an algorithm is a finite sequence of steps that describe how to solve a particular problem. Let's consider an example.

ASCII Character Set Table

ASCII value	Character	ASCII value	Character	ASCII value	Character
0	^@	43	+	86	V
1	^A	44	,	87	W
2	^B	45	-	88	X
3	^C	46	.	89	Y
4	^D	47	/	90	Z
5	^E	48	0	91	[
6	^F	49	1	92	\
7	^G	50	2	93	]
8	^H	51	3	94	^
9	^I	52	4	95	_
10	^J	53	5	96	`
11	^K	54	6	97	a
12	^L	55	7	98	b
13	^M	56	8	99	c
14	^N	57	9	100	d
15	^O	58	:	101	e
16	^P	59	;	102	f
17	^Q	60	<	103	g
18	^R	61	=	104	h
19	^S	62	>	105	i
20	^T	63	?	106	j
21	^U	64	@	107	k
22	^V	65	A	108	l
23	^W	66	B	109	m
24	^X	67	C	110	n
25	^Y	68	D	111	o
26	^Z	69	E	112	p
27	^[	70	F	113	q
28	^\	71	G	114	r
29	^]	72	H	115	s
30	^^	73	I	116	t
31	^_	74	J	117	u
32	[space]	75	K	118	v
33	!	76	L	119	w
34	"	77	M	120	x
35	#	78	N	121	y
36	$	79	O	122	z
37	%	80	P	123	{
38	&	81	Q	124	|
39	'	82	R	125	}
40	(	83	S	126	~
41	)	84	T	127	blank
42	*	85	U		

Table 1.1 ASCII character set

Chapter Two

Introduction to problem solving and Python

Elementary arithmetic, the simplified portion of arithmetic that includes the operations of addition, subtraction, multiplication, and division.

One of the first things we are taught to do is how to count. When we open a pod of peas, if we are lucky, we can take them out one at a time and count them as 1, 2, 3. If we are a bit unlucky, we would also be able to count the number of worms in the pod too!

The numbers we use for this purpose, as you may recall, are the counting numbers: 1,2,3...∞. If we include zero at the beginning of that sequence we call the new sequence the whole numbers. If we now include negative numbers and conceptually place them before the zero, then we call this new sequence the integers: ∞...-3,-2,-1,0,1,2,3...∞. In this chapter, we will only consider inputs that are integers for the problems that we investigate.

For our first task, let us write the steps that will result in us moving through the counting numbers in increasing order. For this, we simply want to obtain the next number in the sequence given a number: for example given the number 4, the next number is 5. Recall that we can use the word add to mean increasing from one number to another. Therefore to get the next number we just add 1 to the number we have. Here are the steps:

- get a number
- increase the number by adding 1 to it
- return the new number

Algorithm 1: Increase a counting number by 1

The steps above together form an algorithm. Algorithms are of fundamental importance in computing. Note that the algorithm has a finite number of steps (3 in this case), a specific step where it begins (step 1 above) and a specific step where it ends (step 3 above). There can be more than one step where an algorithm can end. We can now express our algorithm in a programming language as this will enable us to write our solution so it can be executed on a computer.

Unless otherwise stated, all computers and the solutions we will be writing are for execution on classical computing devices. First, let's become acquainted with entering and executing commands using a programming language.

Python

The programming language we use in this book is Python[2]. To get started with familiarizing yourself with writing code you will need to download the Python interpreter. This can be found at the website provided in the footnote on this page. Follow the instructions provided on the page for downloading and installing the interpreter. The interpreter is the code that will translate what we write into binary that the computer can then execute. Interpreters are one type of translator, the other is called a compiler.

An interpreter translates and executes the lines of code, if there are no syntax errors contained in a program (a collection of lines of code) one line at a time. A compiler translates all the lines of code in a program first, and if there are no syntax errors, then executes the code. Although lines of code may have no syntax errors, they may have logic errors and these logic errors will cause the program to abruptly abort its execution. To help programmers identify syntax errors and to make the process of writing code a bit easier, we use an integrated development environment (IDE). The IDE that comes bundled with Python is called Idle.

An IDE is a set of tools used to write, test, execute and package our solutions. IDEs are typically created for specific languages and platforms. A platform is the collection of software and hardware that our programs can be executed on. Examples of platforms are the combination of Windows 10 running on an Intel based laptop, Linux Mint running on an AMD based server and Android 11 running on a Samsung Galaxy smartphone. Idle, the IDE that is packaged/bundled with Python has versions for more than one platform. On the website it will either detect what platform you are running and present that specific version for you, or ask which platform you would like to install it on (or both). Once you have downloaded Python, open the shell; this is what enables you to type commands in Python, have the interpreter translate them and then execute them.

2. You can learn more about Python at python.org. This website is also very useful as a Python programming reference.

If your device's platform does not have a version that can be downloaded, you can find an IDE that is fully online. An example is repl.it and it can be found at https://repl.it/site/ide

"Communicating" with the interpreter

In the shell, type the digit `1` and then press the enter key on your keyboard. The digit 1 will simply be re-displayed to you or as we say in computing, echoed.

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 1
1
>>>
```

Great, no errors! The digit one is recognized by the interpreter as an integer, evaluated and printed. An integer is one of a few data types that are standard in Python, it is one of the inbuilt or primitive data types. All programming languages have inbuilt data types. A Python integer has the same connotation as we discussed earlier, the negative and positive numbers and zero. The data type is called `int` in Python. Other inbuilt data types are character, string, Boolean and float. For the interpreter to understand what we are writing, we have to follow the language's syntax, as we would need to when writing in English and other languages. We followed the syntax for integers when we typed 1, so the interpreter was able to translate, evaluate and execute. Let's enter a single letter of the alphabet as we would when we write and see the result:

```
>>> C
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    C
NameError: name 'C' is not defined
>>>
```

This may not be what you expected. That error occurred because the interpreter does not know what the letter C is. You do, but for Python to understand you want to enter the letter C, we must follow the language's syntax. The letter C is an example of the character (char) data type and in Python characters are enclosed in either single or double quotes. Refer to Table 1.1 for the list of characters that can be used.

```
>>> 'C'
'C'
>>>
```

Strings are sequences of characters. A sequence of characters from the English alphabet can form words, phrases, sentences and paragraphs. A Boolean data type can only ever have one of two values, True or False. Floats are floating point numbers, what you know as numbers with a decimal or fractional component. In Python you cannot write a fraction as you are accustomed to with one number above a short, horizontal line and another below the line, so you must write them as decimals.

```
>>> 7.2
7.2
>>>
```

Languages like English have homographs and homonyms and though there exist analogies between them and programming languages we must be very specific when coding. For greater simplicity and specificity all programming languages have a set of reserved words. These are sequences of characters that have only 1 meaning in the language and cannot be used for anything else. They are also case sensitive. Table 2.1 is a list of Python's reserved words:

and	break	def	else	False	global	in	nonlocal	or	return	with
as	class	del	except	for	if	is	None	pass	True	while
assert	continue	elif	finally	from	import	lambda	not	raise	try	yield

Table 2.1: Python's reserved words

We also however want the ability to assign different meanings or values to characters and words that we create in our code. Such characters and strings, i.e. those that can be assigned different values are called variables. When creating names in Python, including names of variables, we must again follow the language's syntax rules. The rules for naming variables, constants, functions and procedures (we will discuss those soon) are:

- reserved words cannot be used
- names cannot begin with a digit
- names are case sensitive, even if a name is spelled the same but even 1 character has a different case, it will be deemed as two different names
- names can begin with the underscore character and can contain this character anywhere else in a name
- names cannot contain spaces because if they do they will be misinterpreted

In the shell let us see how this works. We create a variable by writing a name and assigning a value. When we assign a value the variable is then created and its data type is also assigned. Below we have created a variable named `number` by assigning an integer, `0`, to it. In Python, the symbol that you usually understand to mean "is equal to" has a different meaning. Instead it is an operator that assigns the value of whatever is on the right of the symbol to whatever is on the left of the symbol.

```
>>> number=0
```

In Python variables can have different types of data assigned at different points in code. It is a dynamic language; a variable's type is checked when the code is being executed. Other languages that use compilers, for example C, must have the data type assigned once, and it cannot be changed. It is checked when the code is being compiled and if there are errors, the compile will fail. An error can occur if a variable is assigned a type of `int` and somewhere later in the code you attempt to assign a `char` to it.

Returning to Algorithm 1, we have now actually completed step 1. We can complete the steps as follows,

```
>>> number=0            #line 1
>>> number+1            #line 2
1                       #line 3
>>> number              #line 4
0                       #line 5
```

We increased the number `0` by 1 and the interpreter printed the next number, `1`. We can also see that the value of `number` was not changed. Line 1 of the code above where we created the variable number and assigned it a value is called a statement. A statement performs an action, in this case the actions are creating a variable and assigning it a value. Line 2 is called an expression. It performs an operation and returns a value. The value returned is in line 3. The operation that was performed was addition, the symbol for which is the one that you are familiar with. The symbol for subtraction is -, also something you should be familiar with.

Expressions typically do not modify values. Lines 4 and 5 demonstrate this. We see that the expression in line 2, although adding 1 to 0 (`number`) and returning 1 the value of `number` is unchanged. Although we have successfully written our solution for Algorithm 1 in Python, we should write it in such a way that we can supply our solution with any integer and have it return the next number, without having to rewrite the statement and expression each and every time. We need to group them into what is called a function.

Exercise 2.0

1) Enter a statement that creates a variable called `num` and assigns it a value of 5
2) Verify that 5 has been assigned to `num`
3) Enter an expression that will subtract 1 from the value of `num` and return the new value
4) Verify that the value of `num` has remained unchanged

Functions and procedures

When we group expressions and statements that are aimed at providing a solution, we refer to the group as either a function or a procedure. The difference between the two is that a function provides or returns a value(s) that can be used in another function, procedure or elsewhere whereas a procedure does not return any value. We can, of course, group functions and procedures themselves. In interpreted languages we tend to refer to such groups as a script, in compiled languages, a program. Let's write our first function. It will group the earlier statement and expression we wrote to get the next counting number.

```
def _next(number):                              #Line 1
    "increase a counting number by 1"           #Line 2
    return number+1                             #Line 3
```

To enter a script in IDLE, select File from the menu in the shell and then select New File. A new window will then open and this is where you will type the expression and statement. As in IDLE, the statements and expressions have different colours to help a programmer identify reserved words and other elements of code. That is the only reason for the different colours! It is also useful here for our discussion.

The words in orange are reserved words. You can refer to Table 2 and you will see them there. `def` in line 1 tells the interpreter that what follows is a group of statements and expressions that are to be considered a function or procedure. All of the statements and expressions that are a part of the procedure or function will be indented under that line that begins with `def`. This is why lines 2 and 3 do not begin directly beneath `def`. Following `def` is a space and then the name of the function. We discussed syntactically correct names earlier, so `_next` is the name of this function.

Immediately after the `t` in `_next` is a left parenthesis (no space as that would result in a syntax error) the word `number`, a right parenthesis and a colon. `number` is a variable; it corresponds with step 1 of Algorithm 1 where we "get a number". Variables that are included in the

definition of a function/procedure are also called parameters. Parameters tell the function/procedure how many inputs are to be expected. If there aren't any to be expected the syntax rules state that the parentheses must still be included but nothing will appear between them. The colon indicates the end of the definition and tells the interpreter that expressions and statements that belong to the function/procedure are to follow.

Line 2 is not required, and in fact the interpreter ignores it completely. Its sole purpose is to make notes or comments as they are properly called in our code. It is good and recommended practice to use comments when coding. Line 3 is what actually makes this group of lines of codes a function. It returns something. A procedure could still include either a `return` but with nothing following it, or it would exclude the `return` completely. Including `return` is nevertheless a good way to indicate the end of a procedure. Once the interpreter encounters a `return`, if there is an expression or statement included, it would evaluate or execute it and then end, even if there are other expressions/statements, or it will simply end if nothing follows it. By default, the interpreter evaluates the lines of code in the sequence in which they appear, just the same as we just did. We will soon see that we can force the code to be executed in different ways, through causing lines of code to be repeated and introducing a way of choosing which lines should or should not be executed.

When you have typed the function in your new window, save it (File and then Save or Save As) and then run it (Run and then Run Module). There are no syntax errors and so the IDE will then return you to the shell. Now we can test our function:

```
>>> _next(0)
1
>>> _next(-2)
-1
>>> _next(12)
13
>>>
```

These are three separate calls or invocations to our function, with three different values that are associated with the parameter `number` in the function definition. We say we passed the values to the function. The values that we pass are called arguments, so 0, -2 and 12 above are all arguments. The number of arguments passed to a function/procedure should match the number of parameters in the definition. We use "should" as Python does allow an unknown number of arguments to be passed to a function/procedure but for our discussions we will not use that feature.

►►

Exercise 2.1:

Write a function that accepts 1 argument. Write the function in the same window that you wrote _next. The function should return the previous counting number. Call the function `_previous`. It should work as shown in the following examples,

```
>>> _previous(0)
-1
>>> _previous(-2)
-3
>>> _previous(12)
11
>>>
```

Question 2.0:

What would you expect the result to be if you passed a datatype other than an integer to `_previous`? For example,

```
>>> _previous('C')
```

►►

Congratulations, you've written a function and saved it as a script and in so doing, written a solution that was initially expressed as an algorithm. You are well on your way to programming! With this task overcome, we can now build upon what we have learned to solve more and more challenging problems.

Jumping through...loops

We can build upon our process of increasing and decreasing counting numbers to create the operations of finding the sum and difference of two integers. Sure, we can use the symbols we learned about in the previous section, + and -, but let us create functions that do not. If we write the mathematical expression 5+3, we expect the value to be 8. If you were to enter 5+3 in the shell, you would get that expected result. Typically, when we are taught to add, we are instructed to begin with 5 and by moving along the sequence of integers one at a time, visualized by moving right along a line, we would arrive at our learned result of 8. We can use this approach to create our own function to calculate sums by using our function `_next`, and repeatedly calling it a certain number of times. We would begin with 5 in our example and call `_next` 3 times; that would give us our expected result. Here is the algorithm:

1) get the two integers to be summed
2) use one of the integer arguments as the starting value
3) add 1 to the starting value a number of times equal to the second integer argument
4) return the new value

Algorithm 2: Calculate the sum of two integers

And here is a function called `_sum` that implements Algorithm 2:

```
def _sum(int1,int2):                    #Line 1
    "find the sum of two numbers"       #Line 2
    for cntr in range(int2):            #Line 3
        int1=_next(int1)                #Line 4
    return int1                         #Line 5
```

For ease of use you should write this in the same window that you wrote `_next` and `_previous`. In this function, lines 2 and 5 are very similar to what you have already seen and written. Line 1 is similar, but it has two parameters. When we expect more than one argument, the parameters we specify in the definition are separated by commas as shown. Line 4 is an assignment but what is being assigned is a bit different to what we have previously discussed.

We stated that we would call our `_next` function to add 1 to an integer and that is precisely what we are doing on the right side of the assignment operator in line 4; we are adding 1 to whatever the current value of `int1` is. When `_next` returns the new value, we replace the current value of `int1` with this new value, hence `int1` appears on the left side of the assignment statement. Remember, what appears on the right side of the operator is assigned to what is on the left side. This line (line 4) demonstrates an important feature of many programming languages, we can invoke or call one function from within another.

Finally we wanted to call `_next` a number of times that is equal to the second integer and that is the function of Line 3. It calls `_next` the number of times that is equal to the value of `int2`. The reserved word `for` is used to repeat one or more lines of code that are indented below it and in this example it does so the number of times specified in the parentheses that follow the inbuilt Python `range` function. This is one way that we can achieve repetition or looping in code and we will see others as we go along.

The `for` is used here so we know exactly how many times we need to repeat; it is the value contained in `int2`, and nowhere in our function does that value change (as opposed to the value in `int1` which changes each time line 4 is executed). The `range` function when used the way demonstrated in this example, creates a sequence of integers beginning with 0 and ending with one less than the value stored in `int2`. Therefore if `int2` has the value `3`, it would generate the sequence `[0,1,2]`. When the interpreter encounters line 3 it evaluates the call to the range function and line 3 becomes:

```
for cntr in [0,1,2]
```

`cntr` is a variable that is assigned the values 0, 1 and 2. We say that we are iterating over the values in the range, that's 0, 1 and 2 in this case. Any syntactically correct variable can be used.

If `int1` and `int2` have been assigned the values 5 and 3 respectively, lines 3 and 4 are then executed as

```
for 0 in [0,1,2]:
  int1=_next(5) →  for 1 in [0,1,2]:
                     int1=_next(6) →  for 2 in [0,1,2]:
                                        int1=_next(7) → end loop
```

This returns 6 and stores this new value in `int1`

This returns 7 and stores this new value in `int1`

This returns 8 and stores this new value in `int1`

Further, in this example we do not need to know the actual values of `cntr`. We are only using them to count the number of times we want to perform the task(s) specified by the statement that is indented below it. Line 4 requires a little more discussion.

In line 4 we use the current value of `int1`, increase by 1 and change the current value to the new one calculated by `_next`. To understand how this works we have to talk a bit about what happens when we create a variable. Earlier we created a variable called `number` and assigned it a value of zero. When the interpreter sees such a line, it creates the variable by associating the name `number` with a location in the device's memory as displayed in Figure 1 below.

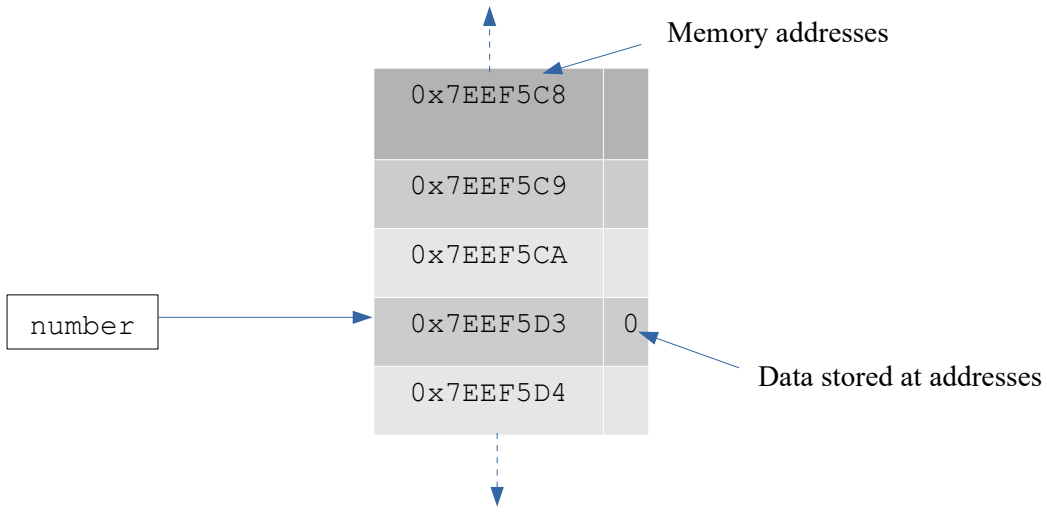


Figure 1: Variable name – memory address association

We are therefore able to read the value stored in the memory address, utilize it in our calculations, and if needed store a new value in the same memory location. In the examples we have seen thus far we stored a new value by using the assignment (=) operator. This also explains why the expression `number + 1` did not change the value stored in the variable `number`.

Exercise 2.2:

Write a function that accepts 2 arguments. Write the function in the same script as the others you have written. Call the function _difference and the parameters int1 and int2. The function should calculate the difference between int1 and int2 by repeatedly calling _previous that you wrote earlier to subtract 1 from int1 a number of times equal to int2. It should work as shown in the following examples,

```
>>> _difference(5,3)
2
>>> _difference(3,5)
-2
>>> _difference(-5,3)
-8
```

Question 2.1:

What would you expect the output to be if we invoked the function with the second argument as a negative integer? For example,

```
>>>_difference(-5,-3)
```

As the function is written, do you get the expected result?

Decisions, decisions!

When executed, `_difference(-5,-3)` also returns `-8` when it should return -2. The rules of elementary arithmetic state that `-5-(-3) = -5+3 = -2`. Similarly, we get unexpected results if for example, we invoke `_sum(1,-1)` and `_sum(-1,-1)`. There are two aspects

that we must rectify that would then have our functions work as we expect for all combinations of our integer arguments in the functions `_sum` and `_difference`.

The first issue stems from our use of the inbuilt range function. Notice that the test cases that are generating the incorrect results all have the second argument as a negative integer. If we test the range function by supplying it with a negative integer argument of say -3 it produces the following list that would be used to iterate over,

```
>>> r=range(-3)
>>> list(r)
[]
```

Recall that by default, when we provide a single integer argument the range function assumes that the values that will be iterated over will begin with 0 and end with the number just before the integer argument provided, but also, by incrementing by 1! Therefore when provided with -3 it assumes the values to be generated and iterated over will range from 0 to -3 in increasing order, but this cannot happen; it therefore cannot return a range of values with what is specified. We have to be more specific or think of another way to address this. With range we can specify a starting value other than zero and we can instruct it to decrement:

```
>>> r=range(3,-3,-1)
>>> list(r)
[3, 2, 1, 0, -1, -2]
>>>
```

In the above use of range, the starting value is 3 the end value is -3 and we instruct it to decrement by using -1. We can also use values other than 1. We would therefore need, if we decided to, to determine if the second integer argument is negative and if it is, use a different range expression. This would mean we would have two `for` loops that basically accomplish the same thing! Let us investigate the other issue that is causing our erroneous results before resorting to this solution.

The rules of elementary arithmetic state that if we are to add a negative integer to another integer (positive or negative) that actually results in a subtraction and if we are to subtract a negative integer from another (positive or negative), that results in an addition. In our functions we have not yet taken this into consideration; in `_sum` we only utilize `_next` and in `_difference` we only utilize `_previous` when in both functions we must use one or the other depending on whether or not the second integer argument passed to `_sum` and `_difference` is positive or negative.

To make a decision in code, we incorporate conditional expressions. In Python we use the reserved words `if`, `elif` and `else`. The general format for their use is:

```
if conditionA:
    statement(s)
elif conditionB:
    statement(s)
else:
    statement(s)
```

Conditions must evaluate to a Boolean result, that is they must evaluate to `True` or `False`. Inbuilt relational operators always evaluate to a Boolean and are extensively used for this purpose. Python's inbuilt relational operators are provided in Table 2.2. The statement(s) indented below the `if` and `elif` are only performed if the respective conditions evaluate to `True`. Both the `elif` and `else` are optional and the `else` is used as a "catch all" for when no other condition in an `if` or `elif` evaluates to `True` or if we explicitly want to perform one statement(s) or the other.

==	equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Table 2.2: Python's inbuilt relational operators

The second of our two issues is the "real" issue with our solution as it is an issue with our solution, our algorithm. We can address the other issue of the use of the range function by using the absolute value of the second integer argument. By using the absolute value, we can continue specifying one argument to the range function in our own functions. To determine the absolute value we can use the `math` library.

Libraries are important tools in computer programming. They provide solutions to common problems, solutions that have been tried and tested under numerous conditions. To use a library we simply use the reserved word `import` followed by a space and then the library's name. When imported the library's functions can then be used in our own code. At the top of your script include the line

```
import math
```

The improved solution for the `_sum` function is:

```
def _sum(int1,int2):
    "find the sum of two numbers"
    for cntr in range(abs(int2)):
        if int2<0:
            int1=_previous(int1)
        else:
            int1=_next(int1)
    return int1
```

Now that we have a function that provides the correct, expected result for all combinations of integer inputs, we can safely use it in other scripts if we like. Before we undertake our other problems let us take stock of what we have covered thus far.

To recap, if you have completed the exercises and written the code we have been discussing you should now know about:

- primitive/inbuilt data types
- inbuilt functions
- reserved words
- variables and their realization through the use of device memory
- expressions and statements
- inbuilt relational operators
- basic Python syntax
- the difference between a function and a procedure
- the difference between an argument and a parameter
- the assignment operator = and primitive arithmetic operators + and -
- an integrated development environment (IDE)
- writing scripts which consist of functions/procedures, which themselves consist of expressions and statements, which themselves consist of commands and variables. This is analogous to our discussion in Chapter 1
- functions, procedures, scripts and programs and that they are evaluated line by line (sequentially) by default, but also through looping(repetition) and making decisions (selection)
- the importance of testing code.

What an impressive list and you've only just begun! There is one last thing that we have been doing but have not expressly stated what it is, until now.

Our approach to creating solutions involved beginning with a smaller solution (increasing/decreasing an integer by 1) and using that solution as a component of a solution to a slightly more difficult problem (calculating the sum and difference of any two integers). This approach is a fundamental one in computing and it is how we will approach creating the vast majority of our solutions.

With the knowledge of this approach let us continue to build upon what we have done by implementing more elementary arithmetic operators.

Exercise 2.2:

Write an improved `_difference` function that corrects the errors discovered and discussed in this previous section. The corrected implementation should return results as shown below,

```
>>> _difference(5,3)
2
>>> _difference(-5,3)
-8
>>> _difference(5,-3)
8
>>> _difference(-5,-3)
-2
>>>
```

Chapter Three

Start small but think big

The name of the classic game Jenga is derived from kujenga, a Swahili word which means "to build"

In the previous chapter we created a few functions. We created a function that can add any two integers and one that can subtract any two integers. We created those two using two other, much simpler functions; one that increased an integer by 1 and one that decreased an integer by 1. This demonstrated our general approach to problem solving, break the problem into smaller problems, solve the smaller (and typically easier) problems and combine these solutions to create a solution to the original, larger one. Let us take this one, very small step further.

We created and tested `_sum` and `_difference` and showed that they could be used to find the sum and difference of any two integers. But what do we do if we want to find the sum or difference of 3, 4, 5 or any number of integers? Recall that we can call upon the "services" of one or more functions from within another as we did when we used the inbuilt range function and our user defined `_next` and `_previous` functions in `_sum` and `_difference`. We discussed that when invoked, the functions return values that could be used in the functions/procedures from which they were called. We can also call a function to calculate a value that can then be used as an argument for another. In this way, we can nest our function calls and thereby find the sum and difference of any number of integers. For example we can find the sum of the integers 3, 4 and 5 by invoking our _sum function as shown below:

```
>>> _sum(3,_sum(4,5))
12
>>>
```

Notice we have not violated the defined requirements of `_sum`. We still supply it with two arguments. The interpreter evaluates such an expression by utilizing rules that you were exposed to years ago and for which there is an acronym to aid in recollection, the order of operator precedence. The acronyms are PEMDAS or BODMAS. In the example above we therefore expect, and correctly so, that the interpreter will evaluate `_sum(4,5)`, return the value 9 and then evaluate `_sum(3,9)` and return our expected result of 12.

Take some time and try combining any number of invocations to _sum/_difference with any number of integers.

Exercise 3.1:

Using the functions `_sum` and `_difference` only and your Python interpreter, determine the results for the following. The evaluations should be processed from left to right.

9+4+3+1	(expected answer, 17)
45+21+877+2+3	(expected answer, 948)
34-13-0-2	(expected answer, 19)
68-98+15	(expected answer, -15)
786-234-45+76+34+111-1515	(expected answer, -787)

Now that we have an understanding of what we have created so far let us go a bit further. We calculated the sum and difference of integers by performing repeated increments and decrements of 1. This is the essence of what integer addition and subtraction are. Similarly, integer multiplication and division involve the repeated addition and subtraction of one integer to and from another respectively. For example:

2x3 = 2+2+2 = 6

3x2 = 3+3 = 6

6÷3 = 6-3-3 = 2 remainder 0

7÷2 = 7-2-2-2 = 3 remainder 1

With a bit of thought, we may see that integer multiplication does indeed involve repeated addition and division, repeated subtraction. We can therefore use our functions that perform those services. Here is a solution for performing integer multiplication that utilizes the services of our `_sum` function, which itself utilizes the `_next` and `_previous` functions.

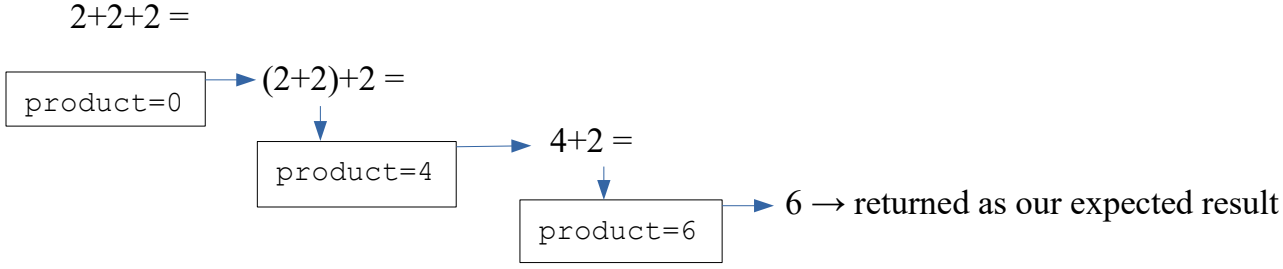
```
def _multiply(int1,int2):                          #Line 1
    "find the product of two numbers"              #Line 2
    product=0                                      #Line 3
    for cntr in range (abs(int2)):                 #Line 4
        product=_sum(product,_sum(int1,0))         #Line 5
    return product                                 #Line 6
```

All but two of the lines of code above should be very familiar to you as they are exactly the same as those we have seen in earlier solutions. The two lines that are new in some way are lines 3 and 5. Our approach to creating our solution to multiply two integers is to add the first integer (`int1`) a number of times that is equal to the second integer(`int2`). In our previous written examples, we added the integer 2, three times and then 3 twice. This is precisely what we are doing in this solution, repeatedly summing `int1`, `int2` times. The practical implementation of this may be causing some consternation but we can quickly alleviate this.

Unconsciously, when we calculate 2+2+2 we insert an intermediate step. We store the result of 2+2 in our mind and then add 2 to that result giving us 6.

2+2+2=(2+2)+2=4+2=6

The storage of the intermediate (and subsequent) result(s) is the purpose of the creation and inclusion of the variable `product` in Line 3. With this our example of the *partial* progression of the evaluation of 2x3 becomes:



We also utilize `product` because we need to keep the original values of both `int1` and `int2` as opposed to overwriting `int1` with the new values as we did in `_sum` and `_difference`. The above is a partial progression because there is something that happens in Line 5, but what happens in Line 5 is really just a further extension of this idea of creating intermediate results.

As we have defined `_sum`, we must always supply it with two arguments. In this case of calculating multiplication however, we always want to keep repeatedly adding one integer to itself. Consequently we supply `_sum` with `int1` and another integer so that we always receive `int1` as the result. This other integer must therefore be zero, as any integer summed with zero will always return the integer. This is the purpose of the expression `_sum(int1,0)` in Line 5. Further, we already demonstrated that we can nest our function calls. We therefore then find the sum of `int1` with the current value of `product` and store that new value back in `product`. This completes the progression of the evaluation process:

ITERATION 1

int1=2, int2=3, product=0

This iteration produces the statement 0 =_sum(0,2) which places the new value 2 into product after evaluating _sum(0,2)

ITERATION 2

product = _sum(product,_sum(int1,0))

substituting: 2 = _sum(2,_sum(2,0))

This iteration produces the statement 2=_sum(2,2) which places the new value 4 into product after evaluating _sum(2,2)

ITERATION 3

product = _sum(product,_sum(int1,0))

substituting: 4 = _sum(4,_sum(2,0))

This iteration produces the statement 4 =_sum(4,2) which places the new value 6 into product after evaluating _sum(4,2)

product=6

Now we are in a position to test our new function,

```
>>> _multiply(2,3)
6
>>> _multiply(-2,3)
-6
>>> _multiply(2,-3)
6
>>> _multiply(-2,-3)
-6
>>>
```

Once again, we see the value of testing our code. We are getting unexpected results when our second argument is a negative integer, and again it is due to the exclusion of elementary arithmetic rules. We still require the absolute value of the second argument for the purpose of generating the values over which we will iterate and therefore use to count the number of times we must perform our repeated additions, but we must also remember that two negative integers multiplied must return a positive integer. Further when the two integers being multiplied are of dissimilar signs, the result must be negative and in such a case it is immaterial which integer is negative. The resulting product will be negative. Therefore -2 x 3 = 2 x -3 = -6. This provides us with a fairly simple solution to correct our `_multiply` function. Before performing the repeated additions we will check that if the second argument is negative we will negate the first.

```
def _multiply(int1,int2):
    "find the product of two numbers"
    product=0
    if int2<0:
        int1=-int1
    for cntr in range (abs(int2)):
        product=_sum(product,_sum(int1,0))
    return product
```

When we test our improved implementation we find that we get expected results regardless of the signs of our integer arguments:

```
>>> _multiply(2,3)
6
>>> _multiply(-2,3)
-6
>>> _multiply(2,-3)
-6
>>> _multiply(-2,-3)
6
```

We should also test that we are able to nest our invocations to our multiplication function. Here are a few such tests, but you should conduct some of your own:

```
>>> _multiply(_multiply(-2,-3),6)
36
>>> _multiply(_multiply(-2,-3),-6)
-36
>>> _multiply(_multiply(-2,3),-6)
36
>>> _multiply(_multiply(2,3),-6)
-36
```

Let's raise the stakes

Take another look at the test values above that were used to ensure that our integer multiplication function was working as we expected. We were actually squaring the integer six as $6^2=36$. This proves that the exponentiation of one integer by a positive integer is actually a process of repeated multiplication, which we proved to be repeated addition. This means that we can write a new function that enables us to calculate exponentiation by positive integers using our previous functions. We must also include a check that if the integer that will be used in our function as the exponent is negative, the function will not return a value but will instead return an appropriate message.

An appropriate message would help the user know why the function could not perform the calculation. The code below presents our function `_raise` that accepts two integers, the second of which must be 0 or positive, and returns the result of the first integer raised to the power of the second integer:

```
def _raise(int1,int2):
    "raise int1 to the power of int2, int2 positive only"
    result=1
    if int2<0:
        return "The 2nd integer cannot be negative"
    for cntr in range(int2):
        result=_multiply(result,_multiply(int1,1))
    return result
```

The essence of the code is that of the function `_multiply` and you should therefore be able to follow what it is intended to do. If we run a few tests we see that it performs as we expect.

```
>>> _raise(3,2)
9
>>> _raise(3,0)
1
>>> _raise(3,-4)
'The 2nd integer cannot be negative'
>>> _raise(-3,4)
81
>>> _raise(-3,3)
-27
```

As usual, take a few minutes and invoke the _raise function with a few of your own arguments to test that it works as expected. It is always good and recommended practice to thoroughly test your code so that you, the developer, can detect and correct errors rather than a user unintentionally doing this for you. To complete our set of elementary arithmetic operations we must include a function that performs integer division. Let's turn our attention to this task.

Divide and conquer

Earlier in our discussion we showed that integer division is in fact a process that involves repeated subtraction, and we gave examples to support this. With integer division we are interested in determining how many times we are able to subtract one integer from another, for *positive integers only*. Now there is a subtle difference between this problem and the others we have written so far.

In the previous solutions, we always knew how many times we would perform the repetitive aspects of the processes. We standardized it as `int2` times. We multiplied by summing `int1` with itself `int2` times, we summed by incrementing `int1` by 1 `int2` times and completed similar processes when we wrote `_raise` and `_difference`. Because of this, we used the `for` loop with the `range` function which created a finite list of integers we used to iterate over to perform the repetition a finite number of times. With the division operation, we do not know beforehand how many times we will be able to subtract one integer from the other and so we should use another command that will permit us to continue iterating until a condition is reached that causes the iteration to cease. It is not the case that we cannot use a `for` loop. It is anticipated that with some more thought and after learning more about iteration and the process of positive integer division, you will be able to write a version using the `for` command.

Many languages, including Python, provide the `while` command and this enables one to perform repetitive tasks in code until a necessary condition is reached that causes the repetition to end. The condition(s) must of course evaluate to a Boolean result. As you should realize by now, the statements and expressions that are to be performed repetitively must be indented under the `while` clause and its condition expression. Let us think a bit about the process we wish to encode for our integer division operation. Looking back at the couple of examples, let us see how we will proceed:

6÷3 = 6-3-3 = 2 remainder 0

7÷2 = 7-2-2-2 = 3 remainder 1

Firstly, we must remember that division by zero is undefined as we will therefore include a condition to check for the case where zero is supplied as the value for int2. Secondly, we stated earlier that we will only be performing divisions using positive integers and will therefore also include a check to ensure we do not continue the encoded process if this is violated. Thirdly, we want to perform repeated subtractions of `int2` from `int1` so we will only perform these subtractions when `int1` is greater than `int2`. Finally we need to ensure we return the correct quotient and remainder pair as the result of our division. In summary our algorithm can be expressed as,

1) accept two integers arguments. If permitted, we will calculate the result of the 1st divided by the 2nd
2) if the 2nd integer is zero, return a message indicating that division by zero is undefined and discontinue the process
3) if either of the integers is negative, return a message indicating that only positive integers are accepted and discontinue the process
4) create and initialize variables quotient and remainder each to zero
5) while the 1st integer is greater than or equal to the 2nd, subtract the 2nd integer from the 1st, store the result of the subtraction as the new value of the 1st integer and increment the variable quotient by 1
6) when the repetition has ended, if the 1st integer is greater than zero store this value of the 1st integer as the new value of variable remainder. If both the 1st integer is zero and the quotient is zero, store the value of the 2nd integer as the new value of variable remainder
7) return the variables quotient and remainder as the results of the function.

Algorithm 3: Calculate the integer division of one integer by another

Let us apply this algorithm to examples (a) and (b) above to see how it accomplishes what we desire:

(a) 6÷3 → `int1` is assigned the value 6, `int2` is assigned the value 3

- the 2nd integer is non-zero so we proceed
- neither the 1st nor the 2nd integer is negative so we proceed

- set `quotient` = 0, `remainder` = 0
- is `int1` greater than or equal to `int2`? Yes as `int1`=6, `int2`=3
  - `int1=int1-int2` New value of `int1`=3, `int2` unchanged
  - `quotient=quotient +1` New value of `quotient`=1

At this point where we have executed the last statement that forms a part of what we wish to be repeated, we go back to the `while` clause to check if our condition still evaluates to `True`.

- is `int1` greater than or equal to `int2`? Yes as `int1`=3, `int2`=3
  - `int1=int1-int2` New value of `int1`=0, `int2` unchanged
  - `quotient=quotient +1` New value of `quotient`=2

We repeat the check of the condition in the `while`.

- is `int1` greater than or equal to `int2`? No as `int1`=0, `int2`=3

We do not continue with the repetition at this point as the `while` condition will now evaluate to `False`. We now proceed with the statement/expression that immediately follows the last statement/expression that is a part of the set belonging to the `while`.

- is the `int1` greater than the `int2`? No as `int1`=0, `int2`=3
- are both `int1` and `quotient` zero? No as `int1`=0, `quotient`=2
- the value of `remainder` was not changed during this calculation and the value of `quotient` is 2 therefore we return the result `(2,0)` signifying the expected result of 2 remainder 0

You can step through the application of the algorithm using example (b) above as an exercise.

With what you have learned, with the exception of the inclusion of the while command, you should be able to both understand the code provided below and write most of it yourself. Before moving on to the next page, write the lines of code that you believe you can. Your function in its entirety will not work without the while included, but doing this will be proof that you are understanding what has been discussed and your programming skill is improving.

```
def _divide(int1,int2):
    "divide int1 by int2 for positive integers only"
    if int2==0:
        return "Dividing by zero is undefined"
    if int1<0 or int2<0:
        return "Only positive integers are accepted"
    quotient,remainder=0,0
    while int1>=int2:
        quotient=_next(quotient)
        int1=_difference(int1,int2)
    if int1>0:
        remainder=int1
    if int1==0 and quotient==0:
        remainder=int2
    return quotient,remainder
```

And here are the results of tests conducted on the function:

```
>>> _divide(4,4)
(1, 0)
>>> _divide(4,0)
'Dividing by zero is undefined'
>>> _divide(0,4)
(0, 4)
>>> _divide(4,3)
(1, 1)
>>> _divide(4,2)
(2, 0)
>>> _divide(-4,2)
'Only positive integers are accepted'
>>> _divide(-4,-2)
'Only positive integers are accepted'
>>> _divide(4,-2)
'Only positive integers are accepted'
```

Notice that we introduced the use of two more reserved words in our function's code, `or` and `and`. These are two examples of logical operators used to combine two or more expressions. The result of the evaluation of a logical operation is a Boolean. We define these operators by what are called truth tables. The truth tables for the logical `and` and `or` operators are as follows. Given a pair of inputs, (let us call these inputs p and q) whose values are Boolean, the result of the application of the logical `and` and `or` operators are defined as:

p	q	p AND q
False	False	False
False	True	False
True	False	False
True	True	True

p	q	p OR q
False	False	False
False	True	True
True	False	True
True	True	True

There is also the logical operator `not` that "flips" a Boolean value,

p	Not p
False	True
True	False

We have now come to the end of our investigation into creating a set of operations that perform a subset of the elementary arithmetic operations. Although languages include inbuilt symbols for the associated operators[3], we took a different approach. We created functions that perform these operations, beginning with rudimentary operations that incremented and decremented integers, and ended with operations that built upon them by implementing integer exponentiation and division.

3 Python's inbuilt elementary arithmetic operator symbols are +, -, *, **, /, % for addition, subtraction, multiplication, exponentiation, division and modulus (i.e. the remainder of an integer division) respectively.

In so doing we have discussed quite a bit about the fundamental process of solving problems by breaking them up into smaller and smaller sub-problems, solving the simpler sub-problems and combining the solutions to solve the larger, more complex one. This is how you should approach solving problems in general.

We also incorporated the use of the Python programming language to express the solutions so that we can reuse them multiple times and in multiple scenarios. You will note that the focus has intentionally not been on Python. There are numerous books and other sources of information that can assist in you becoming an expert Python programmer; this is not one of them. As we progress, what you will learn applies to the discipline and can therefore be applied using any programming language. Any programming language therefore takes its true place as a tool, a you will take yours as a problem solver.

For completeness, all of the functions that we have written so far are presented in the script below:

```
import math
def _next(number):
    "increase a counting number by 1"
    return number+1

def _previous(number):
    "decrease a counting number by 1"
    return number-1

def _sum(int1,int2):
    "find the sum of two numbers"
    for cntr in range(abs(int2)):
        if int2<0:
            int1=_previous(int1)
        else:
            int1=_next(int1)
    return int1
```

```
def _difference(int1,int2):
    "subtract int1 from int2"
    for cntr in range(abs(int2)):
        if int2<0:
            int1=_next(int1)
        else:
            int1=_previous(int1)
    return int1

def _multiply(int1,int2):
    "find the product of two numbers"
    product=0
    if int2<0:
        int1=-int1
    for cntr in range (abs(int2)):
        product=_sum(product,_sum(int1,0))
    return product

def _raise(int1,int2):
    "raise int1 to the power of int2, int2 positive only"
    result=1
    if int2<0:
        return "The 2nd integer cannot be negative"
    for cntr in range(int2):
        result=_multiply(result,_multiply(int1,1))
    return result
```

```
def _divide(int1,int2):
    "divide int1 by int2 for positive integers only"
    if int2==0:
        return "Dividing by zero is undefined"
    if int1<0 or int2<0:
        return "Only positive integers are accepted"
    quotient,remainder=0,0
    while int1>=int2:
      quotient=_next(quotient)
        int1=_difference(int1,int2)
    if int1>0:
        remainder=int1
    if int1==0 and quotient==0:
        remainder=int2
    return quotient,remainder
```

With some basic concepts covered, let us move forward and investigate a few more challenging problems.

Chapter Four

Building more complex solutions

If we were only able to manipulate numbers with computers, there would be little disagreement that their use would be limited, at best. We saw in chapter 1 that the table of ASCII characters included letters of the alphabet and punctuation marks, albeit they are assigned unique numerical representations. This means that our computers are indeed capable of manipulating more than numbers. In fact we went a bit further and demonstrated that the machines can be made to recognize characters such as letters of the alphabet. In Python we showed this is accomplished by enclosing the letter in quotation marks. But alas, if this too were all that we could do, we would still have a difficult time convincing anyone of a computer's usefulness. In this chapter we investigate how we can do more.

Firstly, the curious may routinely be interested in knowing which integer is assigned to which character in the ASCII set. One could routinely check Table 1 for this but doing so would quickly become tedious. Many programming languages, including Python, provide functions that return the integer assigned to a character (its ordinal) and the character assigned to an integer if any have been. In Python the functions that provide these services are `ord` and `chr` respectively.

```
>>> ord('C')
67
>>> chr(67)
'C'
```

Try some of these on your own. Do not be discouraged if some return "strange" results based upon what you see in Table 1. For example,

```
>>> chr(12)
'\x0c'
```

does not display ^L as is shown in the table. \x0c represents a control character. Control characters are used by printers and word processors for example to format the documents you write. While typing an email you may press the enter key on the keyboard to start a new line. That key press must be saved so that it can be reproduced when displayed on the device that the receiver views it. These control or non-printable characters are also a part of the ASCII set and are also assigned ordinals. As they are non-printable, we use (readable) character combinations to represent them in the table. Therefore ^L is read as "control L" and it represents the control character that causes a printer to move to the next page and some word processors will use this when you select "insert page break" from its menu.

Another feature of word processors is enabling the conversion a letter, an entire word or phrases from lowercase to uppercase. We have enough knowledge to create this feature ourselves and then use what we create for other purposes. Using `ord` what is the difference between the ASCII integer representations of lowercase and uppercase A, B and Z? The result is the same, 32. We can use this to create a function that converts a letter to/from lowercase to/from uppercase. We can incorporate the range function to determine whether we have an upper or lowercase letter and therefore which conversion to make. The lowercase letters are in the range 97 to 122 inclusive and the uppercase letters 65-90. Our algorithm for conversion between upper and lowercase letters is:

1) get a letter
2) retrieve its ordinal
3) if its ordinal is in the range 97-122, then it is a lowercase letter. Subtract 32 form its ordinal and return the character representation of the result. It will be the uppercase equivalent
4) if its ordinal is in the range 65-90, then it is an uppercase letter. Add 32 to its ordinal and return the character representation of the result. It will be the lowercase equivalent
5) if its ordinal is not in either range return a message indicating the input was not a letter.

Algorithm 4: Convert the case of a letter

Here is the code in Python that implements our algorithm:

```
def _convertCase(char):
#Convert the case of a letter of the alphabet
    if ord(char) in range(97,123):
        return chr(ord(char)-32)
    elif ord(char) in range(65,91):
        return chr(ord(char)+32)
    else:
        return "character given is not a letter"
```

When invoked, it returns our expected results:

```
>>> _convertCase('c')
'C'
>>> _convertCase('C')
'c'
>>> _convertCase('[')
'character given is not a letter'
```

Looking back at the code for `_convertCase`, although what it is doing is evident because we know what `ord`, `chr` and the ASCII letter ranges are, it is not very "readable". Determining whether or not a letter is upper or lower case is quite useful, we may need to do so often in the code we write. It is therefore a very good candidate to be extracted and written as a pair of standalone functions.

If we thoughtfully choose names for these functions our code becomes:

```
def _isLowerCase(char):
    return ord(char) in range(97,123)

def _isUpperCase(char):
    return ord(char) in range(65,91)

def _convertCase(char):
    #Convert the case of a letter of the alphabet
    if _isLowerCase(char):
        return chr(ord(char)-32)
    elif _isUpperCase(char):
        return chr(ord(char)+32)
    else:
        return "character given is not a letter"
```

At a glance, we can readily see that `_convertCase` determines whether a character is lower or uppercase and then changes them accordingly. There are a couple of other subtle uses of parameters and commands that may introduce doubt in your understanding of what our code does, but this is expected and natural.

In chapter 2 we mentioned homonyms in our introduction of keywords. A homonym is a word that is spelled and pronounced the same but which can have different meanings, depending on the context in which it is used. An example is the word *lie* which, depending on context could mean to be dishonest or to place one's body on a surface in a horizontal position.

The keyword in was used earlier as a part of the for construct to generate a list used to iterate over to perform a repetitive task, but here we are using it to determine membership. In the functions _isLowerCase and _isUpperCase the range function generates the lists of integers as expected and in determines whether the ordinal value of the character is one of the integers in the generated list. If it is, the functions return True and if not, they return False. Here we see the use of in as a predicate, it returns a Boolean as a result of its operation thus affirming or denying membership.

As we progress we will encounter other commands and operators that appear to be “homonyms”. In particular we will discuss the overloading feature that is common to many programming languages.

You would likely have noticed that we assigned the same name to be used in multiple functions that are included in one script. Examples of such parameter names we used are `int1`, `int2` and `char`. But how is this achieved without “confusing” the interpreter and computer? It is accomplished through the use of scope.

At one time or another we have misplaced an item, a favourite pen for example. We may look for it in the book in which it was most recently used. If we don’t find it we may look in the knapsack we used to carry the book and other items. If it is not in the knapsack we may look on the desk upon which the knapsack and book rest, and if not on the desk, we may search the room where the desk is situated. In looking for the favourite pen we incrementally widen the scope of our search until we find it or reach a conclusion that we may have lost it.

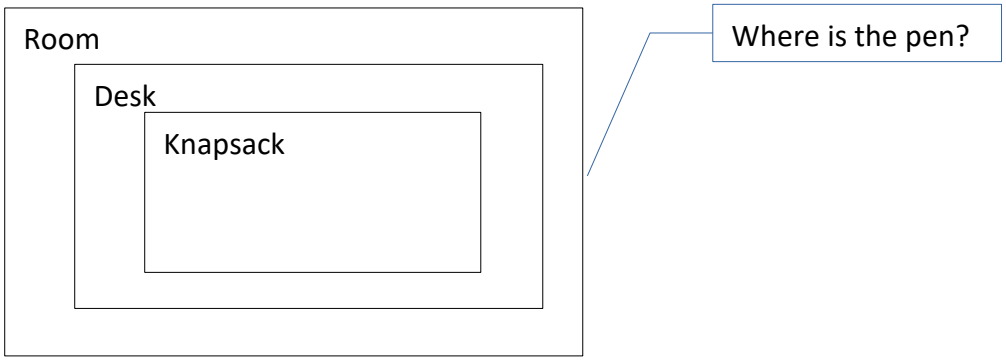
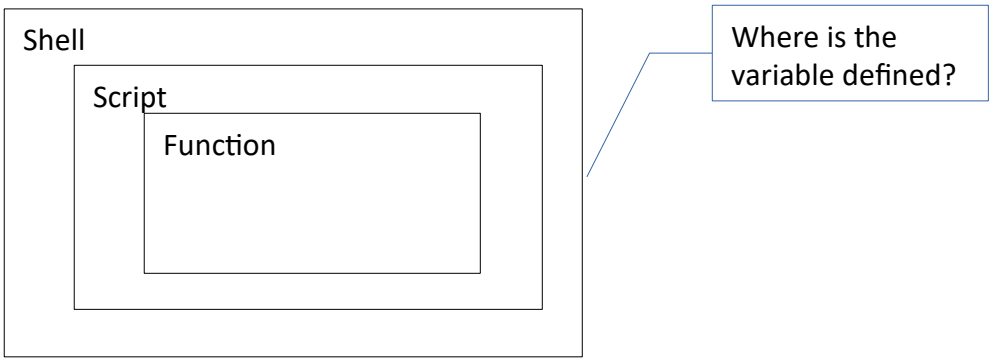


Figure 4.1: Scoping considerations in search of the pen

Furthermore the knapsack is a sort of black box as we may not know its contents because it is opaque and we cannot see inside it.

When we define a function/procedure, it becomes a knapsack of sorts. Anything written within it is shared and known to everything else that is also written within it but is otherwise unknown to everything else outside of it.

In our code that implements the character conversion algorithm each parameter `char`, defined in each of `_isLowerCase`, `_isUpperCase` and `_convertCase` are assigned to different memory locations. They are each different parameters that have the same name. Their respective values are each known to the functions within which they are defined and used but unless that value is exchanged with another function, it remains unknown to them.



As each of the three functions are defined at the same level of indentation and none defined within the other within the script, the shell knows they exist and we can invoke each of them independently.

```
>>> _convertCase('c')
'C'
>>> _isLowerCase("c")
True
```

If we were to rewrite our solution so that we define `_isLowerCase` and `_isUpperCase` within `_convertCase` then the first two functions become known only to the last, and we can define `char` as a parameter once in `_convertCase`.

```
def _convertCase(char):
    #Convert the case of a letter of the alphabet
    def _isLowerCase():
        return ord(char) in range(97,123)                    # Line A

    def _isUpperCase():
        return ord(char) in range(65,91)                     # Line B

    if _isLowerCase():                                       # Line C
        return chr(ord(char)-32)
    elif _isUpperCase():                                     # Line D
        return chr(ord(char)+32)
    else:
        return "character given is not a letter"
```

In this version, when the interpreter encounters the variable `char` in lines A and B it is unknown in its present, local scope as there is no creation or definition of `char` anywhere within these two functions. The interpreter then looks at the immediate, outer scope which is the function `_convertCase`, the function within which these two are now defined. It finds that it is defined as a parameter and therefore uses this `char` and its value. Had it not found a `char` there or defined in the script, it would have returned an error indicating that `char` is undefined. If we were to invoke either _isLowerCase or _isUpperCase we would receive an error indicating it is undefined.

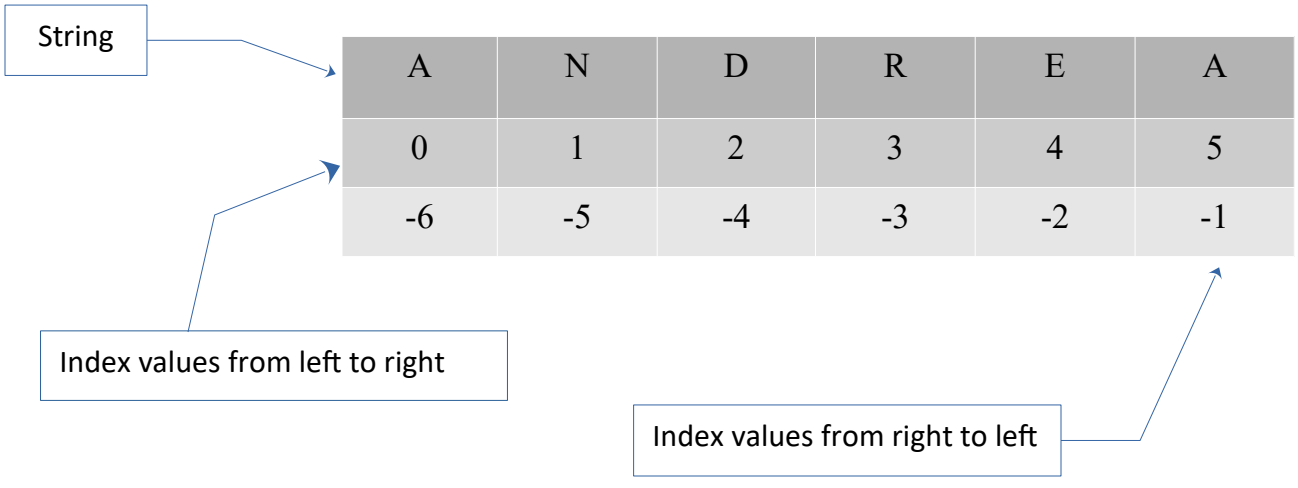
```
>>> _isLowerCase()
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    _isLowerCase()
NameError: name '_isLowerCase' is not defined
```

This is because these functions' scope is limited to that of the function within which they are defined, `_convertCase`, and therefore their invocation must occur from within `_convertCase` as shown in lines C and D.

String along a while longer

So once again we find that we can do a bit more, but still maybe not as much as we would like. We can use and manipulate a character, integers and booleans, individually but let us continue our general approach and build upon what we know and can do to enable us to do more.

Although we know and can manipulate a character, we have the basic component of words, phrases and the like. A word is a sequence or string of characters and many programming languages provide an inbuilt data type called a string. The name Francine can be represented as a string by enclosing it in quotes, just as we did with a character. Anything we enclose in quotations will be considered a string, thus, "Mona, Kingston 7" is also a string. That string includes upper and lowercase letters, an integer, a comma and two blank spaces. A string is immutable; it cannot be changed once created. We can however access its constituent characters. In Python, the string's constituents can be accessed from left to right using an integer index beginning with zero or accessed from right to left beginning with an integer index of negative one. For example:



We can therefore retrieve any character from a string using its index. Below is an example of how we would create the string Andrea and retrieve the uppercase letter A:

```
>>> name="Andrea"
>>> name[0]
'A'
>>>
```

We can also iterate over a string using a for loop. Using our `_convertCase` we could iterate over our sting and swap the case of the characters but this would be whimsical. Instead we can write a new function that converts all lower case letters to upper case, a feature included in many word processors. The function `_lowerToUpper` could be written as:

```
def _lowerToUpper(string):
    # convert all lower case letters to uppercase
    allUpper=''
    for char in string:
        if _isLowerCase(char):
            allUpper=allUpper+chr(ord(char)-32)
        if _isUpperCase(char):
            allUpper=allUpper+char
    return allUpper
```

When tested we see that it produces expected results:

```
>>> _lowerToUpper("Andrea")
'ANDREA'
```

A few points to note. `allUpper` is created and initialized as an empty string, denoted as a pair of quotations with nothing between them. The `char` variable in the `for` statement is not used as a counter in this form of the `for` statement; it is used as a placeholder. The loop iterates over the contents of `string` and assigns each element one at a time in succession to `char`, beginning with the element in position 0. In our example the element in position 0 is A.

The + operator is used here to perform a concatenation, not an addition as adding two characters has no logical meaning.

The + operator is an example of an overloaded operator, it performs a different function depending on the operands it is presented with. It is a binary operator and therefore requires two operands. When presented with numeric operands such as integers and floating point numbers, it performs the addition operation. When presented with characters or strings it performs the concatenation operation. What do you think will be the result if you were to present it with two operands where one operand is a character/string and the other numeric?

Using what we now know we can create an even more useful function. Many applications require passwords that are considered more challenging to guess. Such passwords are required to be of a minimum length, contain at least 1 uppercase character, at least 1 lowercase character, at least 1 decimal digit and at least 1 printable character that is not a letter of the alphabet. To do this we would take a password as a string, iterate over it character by character and check each to see if it satisfies one of the above criteria. If all the criteria for a more secure password are satisfied we will return True to indicate the password is such or False otherwise.

The code for `_isSecurePassword` is located on the following page. All the code contained in the functions has been used and introduced to you before with two exceptions. The `len` function is inbuilt and returns the length of its argument. In this case it provides the length of the string provided, i.e. the password. The backslash is simply used to allow a line of code to span multiple lines in the editor's window without causing a syntax error. Take a bit of time and work through the code provided. A couple of tests are also included on the following page.

```
def _isSecurePassword(password):
    # is the provided password secure?
     hasLower,hasUpper,hasDigit,hasPrintable,isMinLen=False,False,False,\
     False,False
    def _isLowerCase():
        return ord(char) in range(97,123)

    def _isUpperCase():
        return ord(char) in range(65,91)

    def _isDigit():
        return ord(char) in range(48,58)

    def _isPrintable():
        return ord(char) in range(33,48) or ord(char) in range(58,65)\
               or ord(char) in range(91,97) or ord(char) in range(123,128)

    if len(password)>7:
        isMinLen=True
    for char in password:
        if _isLowerCase():
            hasLower=True
        if _isUpperCase():
            hasUpper=True
        if _isDigit():
            hasDigit=True
        if _isPrintable():
            hasPrintable=True
    return hasLower and hasUpper and hasDigit and hasPrintable and isMinLen
```

```
>>> _isSecurePassword('Andrea')
False
>>> _isSecurePassword('P@s$w0rd')
True
```

Other more complex, inbuilt, data types

We should now have some momentum. We have some knowledge of an approach to problem solving and a few, basic, programming tools in our "pockets". We have combined these in a few different ways and have seen that we can create solutions to more challenging and interesting problems.

We have either mentioned or discussed some primitive data types and have used some of them in our code. Examples include integers, floating point numbers, characters, booleans and strings. We saw that a string is a sequence of characters and that we could do interesting things with them. There are other data types provided by programming languages that can be composed of other primitive data types, but what is provided by a language as inbuilt differs from one language to another. Some of these composite data types are essentially the same but are assigned different names. Here is an enumeration of a few that can be found:

- array
- list
- linked list
- tuple
- dictionary
- record
- set

In the programming language C, you will find an inbuilt, composite data type called an array but you will not find a data type with that name in Python. In Python, you will however find an inbuilt, composite data type called a list. Lists and arrays have many similarities so learning about one will provide you with an ability to use the other. In Python, the inbuilt, composite data types that are included in the above enumeration are the list, tuple, dictionary and set.

Table 4.1 below provides a summary of the features of Python's inbuilt composite data types.

Feature	String	List	Tuple	Dictionary	Set
Mutable	N	Y	N	Y	N
Indexed	Y	Y	Y	N	N
Keyed	N	N	N	Y	N
Ordered	Y	Y	Y	N	N
Duplicate elements	Y	Y	Y	N	N
Created using	' ' or " "	[ ]	( )	{:}	{ }
Heterogeneous	N	Y	Y	Y	Y

Table 4.1: Python's inbuilt composite data types

Here is an example of the word Andrea stored using each of the types in Table 4.1 with each letter as 1 element, a character:

```
>>> name='Andrea'
>>> name
'Andrea'
>>> name=['A','n','d','r','e','a']
>>> name
['A', 'n', 'd', 'r', 'e', 'a']
>>> name=('A','n','d','r','e','a')
>>> name
('A', 'n', 'd', 'r', 'e', 'a')
>>> name={1:'A',2:'n',3:'d',4:'r',5:'e',6:'a'}
>>> name
{1: 'A', 2: 'n', 3: 'd', 4: 'r', 5: 'e', 6: 'a'}
>>> name={'A','n','d','r','e','a'}
>>> name
{'A', 'd', 'a', 'r', 'n', 'e'}
```

You should note that:

1) all but strings require the use of a comma to separate elements when there is more than one element
2) the colon is required to separate the key (the digits in our example above) from the data and the key should itself be an immutable data type
3) the echoed result of the set shows that unordered data types can appear in a different sequence each time they are used
4) the row "Created using" also shows the representation for the respective empty data types
5) the types that are indexed follow the same indexing rules as the string data type that were discussed earlier.

The power of these data types, as we will see later, is that we can combine them all in different ways. It provides tremendous flexibility and allows us to create our own data types. This power along with the creation of functions and procedures permits us to solve a great many problems. An alternative way of storing the string `'Andrea'` could be a list, tuple, dictionary or set with one element.

```
>>> name=["Andrea"]
>>> name
['Andrea']
>>> name=("Andrea")
>>> name
'Andrea'
>>> name={'first':'Andrea'}
>>> name
{'first': 'Andrea'}
>>> name={'Andrea'}
>>> name
{'Andrea'}
```

Because of the heterogeneity feature lists, tuples and dictionaries can themselves be composed of any combination of other data types. Below we have a list with an integer, string, Boolean, float, dictionary, tuple, set and list components:

```
>>>anycombo=[1,3.4,True,'Chase',('Cakes'),{4:'Kat'},{1,5,7},[7,4.1]]
>>> anycombo
[1, 3.4, True, 'Chase', 'Cakes', {4: 'Kat'}, {1, 5, 7}, [7, 4.1]]
```

A set however, though heterogeneous cannot be composed with mutable composite data types i.e. lists and dictionaries.

```
>>> name={'Andrea',(1,2)}
>>> name
{'Andrea', (1, 2)}
>>> p={[1,2,3],1}
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    p={[1,2,3],1}
TypeError: unhashable type: 'list'
```

Having expanded our tool set by including the ability to create (abstract) representations for a great many things using our composite data types we are now in a position to explore more complex approaches and techniques to expressing our solutions with code and thereby solve even more challenging problems.

Chapter Five

Building upon the foundation

Computer programming is an art. We each see and understand a problem differently. Each person will also express a solution in their own way. We each have our own style of writing. As you continue to develop your programming skill you will also develop your own style of writing code. The tools that a programming language provides also enable creative expression because we have a number of ways that we can write a solution to the same problem. Let us look at a small example.

In the previous chapter we developed a bit of code that was used to determine the strength of passwords. As numerous applications require the use of passwords, the idea behind that bit of code is quite useful. We can extend the thinking behind that "problem" a bit further so that we may develop a deeper understanding of what is involved in implementing one of the most widely used components in modern devices and services.

In addition to testing the strength of a password, a password is not typically stored on a device. What is stored is an encrypted form of it. By encrypting it, we are attempting to make it unintelligible to anyone who reads it. Passwords are after all, supposed to be kept secret and encryption is a fundamental component of another art form, cryptography. One of the earliest ways of maintaining secrecy has been attributed to the Roman General Julius Caesar. The Caesar cipher takes the letters of words and scrambles them, making them appear nonsensical. The word 'computer' for example could be encrypted using the Caesar cipher as "lxvydcna". The Caesar cipher algorithm is shown below. We will only use lowercase letters for this version.

1) Get the word to be encrypted.
2) Assign a unique integer to each lowercase letter, integers must form a sequence.
3) Choose a random letter of the lowercase alphabet. This letter will be called the key.
4) Convert when necessary, uppercase letters to lowercase.
5) Encrypt the word, one letter at a time by finding the sum of the integer assigned to a letter and the key and replacing the letter with the letter assigned to the sum.
6) Return the new sequence of letters.

Algorithm 5: A version of the Caesar cipher

Upon close inspection of the algorithm you will notice that we have already created solutions for the majority of its steps. In fact, we already know how to do everything except how to choose a random letter. This is yet another example of the power of our general approach to problem solving. The problem has been broken up into smaller ones which have been solved, so we can easily solve the more challenging one.

The vast majority of programming languages offer some way of generating pseudo-random numbers. They are called pseudo-random because they are not *truly* random. As one of the steps of the Caesar cipher involves the assignment of unique integers to letters, we can randomly generate an integer that is within the range assigned to our letters and thereby randomly choose a key. With Python, one option that provides the service of generating random numbers is the `random` library. Here is a solution that implements our algorithm for the Caesar cipher:

```
def _genKey():
    return random.randint(0, 100)%26                    #Line 1
def _isUpperCase(char):
    return ord(char) in range(65,91)
def _encode(pTxt,k=_genKey()):                          #Line 2
    cTxt=''
    for i in pTxt:
        if _isUpperCase(i):
            i=chr(ord(i)+32)
        cTxt+=chr((((ord(i)-97) + k)%26)+97)            #Line 3
    print ('Using ', chr(k+97),' the ciphertext is ',cTxt)
```

When we test with the word *computer* and the randomly selected letter j we obtain,

```
>>> encode('computer')
Using  j  the ciphertext is  lxvydcna
>>>
```

`_genKey` is our function that we use to (pseudo) randomly select a letter of the alphabet. In Line 1 we use the `randint` method of the random library to generate a number between 0 and 100. The choice of this range is arbitrary. For ease of calculation, we mentally assign the lowercase letters of the alphabet the integers 0-25. The actual assignment of the unique integers to the lowercase letters has already been done; we will simply use the ASCII code. Because we are using a randomly chosen letter as our key we ensure that the randomly generated number we choose will always be an integer in the range 0-25 by finding the remainder when the number generated by `randint` is divided by 26. Python's inbuilt symbol for the modulus operator is %.

Line 2 is interesting because we are specifying two parameters, the word to be encrypted and the key to be used. However, the key will be generated by `_genKey` and the value generated becomes the argument that is assigned and bound to the parameter `k`. Recall that when we introduced functions, we stated that we can call upon a function anywhere from within another function or script. This further demonstrates this and as we will discuss a bit later on, we can extend this idea even further by passing a function *itself* as an argument to another function. Because the function `_encode` generates the argument to be bound to `k` itself, the user is neither expected nor required to enter anything but the word to be encrypted.

Line 3 is the bit of code that actually 'scrambles' the letters as stated in Algorithm 5. We first convert the integer value to one that is within the range 0-25 (remember 97 is the ASCII code for lowercase a), find the sum of this number and the key k, ensure that the sum is itself within the range 0-25 (i.e. the sum mod 26), and finally determine the ASCII value in the range assigned to the lowercase letters by adding 97 to this result.

The other unfamiliar bit of Line 3 is the symbol +=. This is Python 'shorthand'. If we have a variable `z` to which we want to add/concatenate some other value `y` and place the result back into `z`, we may write that statement as either `z=z+y` or `z+=y`. Similar symbols can be used for the other arithmetic operators.

Programming is an art though, and as we discussed earlier, we can write multiple solutions for the same problem. Below is a slightly different version called `_encodeV2` that uses a while loop and an operation called slicing that we can perform on indexed data types:

```
#_genKey and _isUpperCase are also to be included here
def _encodeV2(pTxt,k=_genKey()):
    cTxt=''
    while pTxt!='':
        i=pTxt[0]                                          #Line 1
        if _isUpperCase(i):
            i=chr(ord(i)+32)
        cTxt+=chr(((ord(i)-97) + k)%26)+97)
        pTxt=pTxt[1:]                                      #Line 2
    print ('Using ', chr(k+97),' the ciphertext is ',cTxt)
```

The slicing operator is presented in the line of code labeled Line 2 in `_encodeV2` above. A slice is a sub set. The syntax is 1 or 2 colons enclosed within a pair of square brackets:

[starting index of slice: ending index of slice: step]

We can begin a slice from the left or right of our indexed data type. In our code we are taking slices of a string i.e. sub-strings beginning at position 0 of the string. The code also demonstrates a general procedure for processing indexed data types:

1) If the indexed data item is not empty, take the element in position 0, otherwise stop.
2) Perform the procedure we wish with this element.
3) Remove the element that is in position 0 i.e. take the slice [1:] of the indexed data item.
4) Repeat beginning at step 1.

In line 1 we "extract" the element in position 0 by assigning a copy of it to the variable `i`. As you know, every element of the string is a letter so as the algorithm described, we are encrypting one letter of a word at a time. We perform the encryption exactly as we did in the previous version and then we take a slice of the word. The slice we take has starting index of 1 and no specification of and end, so Python defaults to the end of the string. There is also no specification of a step so the Python default is 1. It therefore traverses the string by incrementing the index by 1 when taking the slice from starting index to ending index. As an example of this procedure, let us look at a couple of rounds using the string `'computer'` and `k=j`

(a) `pTxt='computer'`
(b) `i=pTxt[0], therefore i='c'`
(c) `encrypt 'c' using j therefore c is encrypted as l. cTxt='l'`
(d) `pTxt=pTxt[1:], we replace pTxt with the string that excludes 'c'`
`therefore pTxt='omputer'`
(e) `i=pTxt[0], therefore i='o'`
(f) `encrypt 'o' using j therefore o is encrypted as x. cTxt='lx'`
(g) `pTxt=pTxt[1:], we replace pTxt with the string that excludes 'o'`
`therefore pTxt='mputer'`
(h) `we continue until we get the result 'lxvydcna'`

It is important to remember this general procedure for processing indexed data. It will be very useful and will be applied when processing the other indexed data we have discussed. In addition to strings which we just covered, these were lists and tuples.

Encryption, when applied to the problem we just discussed, is not intended to be a one-way function. What we encrypt we are desirous of decrypting. With the Caesar cipher, the procedure for decrypting is the exact opposite of the encryption procedure. I invite you to write a function `_decode` that reverses the procedure implemented in `_encode`/`_encodeV2`. Your decode function must accept 2 arguments, the encrypted word, and the letter that was used as the key to encrypt it.

It will not be expected to reconvert any previously converted letters back to uppercase as the encode functions did not retain that information. The pair of functions would work as follows:

```
>>> _encode('slice')
Using  n  the ciphertext is  fyvpr
>>> _decode('fyvpr','n')
Using  n  the plaintext was  slice
>>> _encodeV2('Slice')
Using  x  the ciphertext is  pifzb
>>> _decode('pifzb','x')
Using  x  the plaintext was  slice
>>>
```

Our final bit of code that we will implement will be to encode an entire sentence using our functions. The sentences we will encrypt will consist of only lowercase letters and a single space to separate them. We will incorporate the inbuilt string method split that takes a string and separates it using a delimiter. The delimiter we will use is the space. Our encode and decode functions are provided on the following page with a few minor modifications to the functions we wrote earlier. In both, we also demonstrate slicing from the right by specifying that we wish to take all elements but the last, the trailing space. The slice is specified as [:-1]

I hope you found this to be both interesting and fun. However, these functions nor the Caesar cipher are by no measure considered secure. To make life a little more difficult for someone who is completely unaware of cryptography or who is not interested in decoding, or for information that is by no means private they may work, but otherwise they are very useful for this educational purpose.

We will continue the trend of undertaking ever more challenging and interesting topics. In the next chapter we will investigate how we can create our own data types.

```
#remember to include _genKey in your script
def _encodeSentence(sentence,k=_genKey()):
#encode a sentence using the Caesar cipher
    def _isUpperCase(char):
        return ord(char) in range(65,91)

    def _encode(pTxt):
        cTxt=''
        for i in pTxt:
            if _isUpperCase(i):
                i=chr(ord(i)+32)
            cTxt+=chr((((ord(i)-97)+ k)%26)+97)
        return cTxt

    encrypted_sentence=''
    word_list=sentence.split(' ')
    for word in word_list:
        encrypted_sentence+=(_encode(word)+' ')
    return encrypted_sentence[:-1],chr(k+97)
```

```
def _decodeSentence(sentence,k):
#decode a sentence that was encoded using _encodeSentence
    def _decode(cTxt):
        ky=ord(k)-97
        pTxt=''
        for i in cTxt:
            pTxt+=chr((((ord(i)-97)- ky)%26)+97)
        return pTxt

    decrypted_sentence=''
    word_list=sentence.split(' ')
    for word in word_list:
        decrypted_sentence+=(_decode(word)+' ')
    return decrypted_sentence[:-1]

>>> _encodeSentence('this is the end of this chapter')
('lzak ak lzw wfv gx lzak uzshlwj', 's')
>>> _decodeSentence('lzak ak lzw wfv gx lzak uzshlwj', 's')
'this is the end of this chapter'
```

Chapter Six

Abstract data types

Abstraction, the process of considering something independently of its associations or attributes.

Not everything in life can simply be represented as a composite data type. There are numerous elements of one's reality that do not, but, the beauty and power of our programming languages allow us to create our own data types and thereby allow us to represent just about anything. This notion of representation provides a delightful segue into a brief discussion on one of the tenets of software creation - abstraction.

When investigating and attempting to understand a problem that we wish to solve, a question that often arises is "is what is being presented the *real* problem or is it an example of it?" This is an important question as it helps to direct our focus to the core of what we are trying to develop a solution for. We must determine if what is being presented as a problem is indeed an example of a more fundamental one, a family of problems as such. If we can understand what the real problem is, we may then be able to create a solution for this and in so doing, we will have created a solution for all other members or examples of the problem. Focusing our attention in this way is the essence of abstraction: developing an understanding or representation of some element of one's reality, tangible or intangible, reduced to its essential form.

When developing code, many languages thus provide the ability to create user defined data types. A user-defined data type is an abstract data type or ADT. We create ADTs by combining inbuilt data types and even other ADTs together to form new data types. We use ADTs to represent objects that exist in our reality. In this chapter we will construct three ADTs commonly used in computer programming: stacks, queues, binary trees.

Stack

There is a very widely available infant toy where an infant piles one disc on top of another. One disc that is placed underneath another cannot be removed without first removing those above it. A depiction of such a toy is shown in Figure 6.1; the blue disc cannot be removed without first removing the other five discs above it. The yellow and smallest disc is the last disc to be placed on the tower and will therefore be the first one to be removed.

This toy is a visual representation of the programming data structure we call a stack. A stack's principal attribute is that when used, its data is stored and accessed just as the discs are placed and removed on the toy, last in, first out or LIFO.

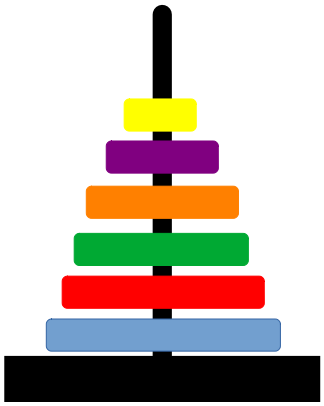


Figure 6.1: A stack-able tower

In creating an ADT that will be used to implement some data structure, there are two components that we must always include: a representation of the ADT and a set of functions that provide the means by which we may use the ADT. In developing a representation our ability to create an abstraction of it comes into play. We cannot create a tower and discs nor can we physically add and remove discs. Reduced to its "essential form", however, the toy illustrates addition and removal of discs from only one end of the tower. The top of the tower permits discs to be added and removed while the tower's base prevents both actions. The tower then, is our stack while the discs are our data. We add and remove data from only the "end" of a stack. This structure can be represented using data types that we have already encountered. It is up to us to decide which one to use.

As we will want to use a stack to store most any type of data, we cannot use the primitive data types. We can, however, use one of the composite data types, but it should be one that permits heterogeneity and order (the discs are added and removed sequentially). Dictionaries and sets do not maintain order; strings are not heterogeneous. This leaves us with lists and tuples. We may also want to modify stack elements as addition and removal of stack elements is intrinsic to its behaviour, therefore a list seems to be the best fit as lists are mutable whereas tuples are not. A stack can therefore be represented as a list, but we impose (through the associated functions we will create) a restriction that list elements can only be added and removed from one side of the list.

Using a list in this way, however, does not distinguish it from any other list so it is useful to include the name of our ADT within the ADT's representation itself. The representation of our stack then, consists of a name tag or just tag, and a list. Furthermore, once created we do not want the stack itself to be modified, only its contents. The use of a tuple can also be incorporated in our stack representation for this purpose. Our stack representation is,

```
('Stack',[])
```

This takes care of our first component of our stack ADT - its representation. We must now create the functions that will enable its use in the manner in which we have conceptualised it. The stack ADT functions are provided below. Be sure to read the comments included in the code.

```
def new_Stack():
# create a new, empty stack ADT using our representation
    return ('Stack',[])

def is_Stack(sk):
# verify object is a stack. It must be a tuple with 2 elements,
# the 1st must be the string 'Stack', the 2nd must be a list
    return type(sk)==tuple and len(sk)==2 and sk[0]=='Stack' and\
               type(sk[1])==list

def stack_Contents(sk):
# retrieve the contents of the provided stack, but 1st check that
# the object conforms with our stack ADT
    if is_Stack(sk):
        return sk[1]
    else:
        raise TypeError('Argument is not a stack')
```

```
def empty_Stack(sk):
# determine if the stack is empty, but 1st check that
# the object conforms with our stack ADT
    if is_Stack(sk):
        return stack_Contents(sk)==[]
    else:
        raise TypeError('Argument is not a stack')

def stack_Top(sk):
# retrieve the value of the data element at the top of the stack,
# but 1st check that the object conforms with our stack ADT and that
# if it does conform that it is not an empty stack
    if is_Stack(sk):
        if not empty_Stack(sk):
            #top of stack is in position -1, i.e. right end of list
            return stack_Contents(sk)[-1]
        else:
             raise Exception('Cannot provide top element of an empty
stack')
    else:
        raise TypeError('Argument is not a stack')
```

```
def push_Stack(sk,el):
# add a new element to the stack, but 1st check that
# the object conforms with our stack ADT. This action is called
# pushing a stack.
    if is_Stack(sk):
        stack_Contents(sk).append(el)
    else:
        raise TypeError('Argument is not a stack, cannot push')

def pop_Stack(sk):
# remove the data element at the top of the stack,
# but 1st check that the object conforms with our stack ADT and that
# if it does conform that it is not an empty stack.
    if is_Stack(sk):
        if not empty_Stack(sk):
            stack_Contents(sk).pop(-1)
        else:
         raise Exception('Cannot remove top element of an empty\
              stack')
    else:
        raise TypeError('Argument is not a stack, cannot pop')
```

The combination of representation and function set completes our definition of our stack ADT. Note that we included the `raise` keyword. It is used in error handling and we use it here to make our ADT error handling features more robust. When the Python interpreter encounters the `raise`, it stops program execution and prints the customised error included. In our code, we included specific error types (Exception and TypeError) to more accurately reflect the cause of an error and thereby be of greater assistance to a potential user of our ADT. We also incorporated the use of the inbuilt list methods `append` and `pop` which add an element to the right end of a list, and remove an element specified by its index from a list, respectively. Below we provide a few lines of code entered at the shell that demonstrate the basic use of our stack ADT:

```
>>> stack=new_Stack()
>>> stack
('Stack', [])
>>> is_Stack(stack)
True
>>> empty_Stack(stack)
True
>>> push_Stack(stack,'1st element')
>>> push_Stack(stack,'2nd element')
>>> empty_Stack(stack)
False
>>> stack
('Stack', ['1st element', '2nd element'])
>>> stack_Contents(stack)
['1st element', '2nd element']
>>> stack_Top(stack)
'2nd element'
>>> pop_Stack(stack)
>>> stack
('Stack', ['1st element'])
```

Once the ADT's functions have been developed and tested, we should always use them when using the ADT. Although we could, for example access the top of a stack by specifying the index position, there is a function that provides this service. If we were to specify the index and not use the function, we would be violating the abstraction, and this is not very good programming practice.

Consider a scenario where we used a stack as a core component in a very large program and at the end of writing the code, decided we preferred to make the left end of the stack its top and not its right as we did here. If we violated the abstraction, then we would be required to search through the code and find every instance where the stack was pushed, popped and the top retrieved and modify the code. If we do not violate the abstraction we would simply modify the code in only 3 places: the functions that push, pop and retrieve the top of the stack.

Stacks are quite useful. We will see in a later chapter how stacks are incorporated to provide another means of performing repetition in our code. Stacks are also used in many day-to-day activities. Having developed a stack ADT, we can use the same approach to develop *any* ADT. Determine the essence of the object's structure and operation, then conceptualise and implement a representation and an associated set of functions that enable its use based upon the chosen implemented representation.

Queue

A queue is something we encounter in numerous scenarios. When we go to a concert, bank or supermarket, we typically end up joining a line. Such a line is a queue; the person who joined it first will leave it first and who joined it last will leave it last, once no one skips the line!

A queue is very similar to a stack, except that it permits data to be added to one end only and permits data to removed from the opposite end only. It is a FIFO structure, or first in first out. Figure 6.2 depicts a typical queue.

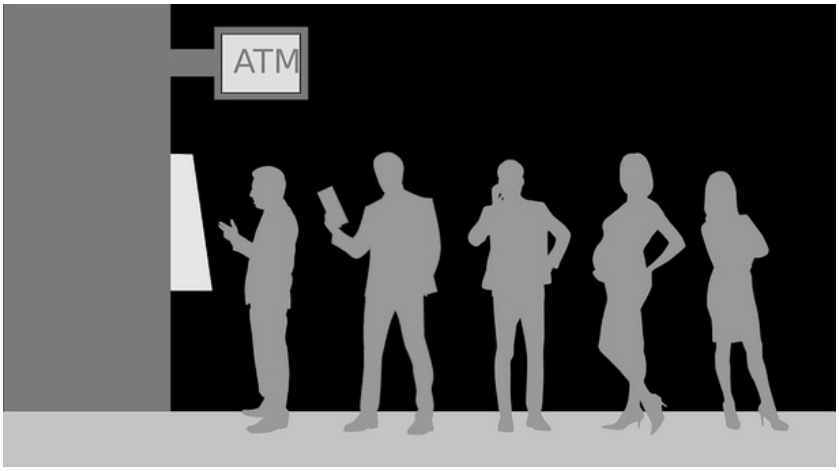


Figure 6.2: A typical queue at an ATM

Our thoughts on the essence of a queue are very similar to that of a stack in terms of the heterogeneity of data we would like to store, the mutability of both the data and the ADT itself - and the sequence in which the data is added and removed. The major difference is that with a queue we must add data to one end only and remove data from the opposite end. The representation of a queue can therefore be:

```
('Queue',[])
```

As with our stack, we will add or append to the right end of the list and therefore we must remove from its left end. The queue ADT functions are provided below:

```
def new_Queue():
    return ('Queue',[])

def is_Queue(q):
    return type(q)==tuple and len(q)==2 and q[0]=='Queue' and\
           type(q[1])==list
```

```
def queue_Contents(q):
    if is_Queue(q):
        return q[1]
    else:
        raise TypeError('Argument is not a queue')

def empty_Queue(q):
    if is_Queue(q):
        return queue_Contents(q)==[]
    else:
        raise TypeError('Argument is not a queue')

def queue_Front(q):
    if is_Queue(q):
        if not empty_Queue(q):
    # front of the queue is position 0, i.e. the left end of the list
            return queue_Contents(q)[0]
        else:
          raise Exception('Cannot provide front element\
                  of an empty queue')
    else:
        raise TypeError('Argument is not a queue')
```

```
def enqueue(q,el):
    if is_Queue(q):
        queue_Contents(q).append(el)
    else:
        raise TypeError('Argument is not a queue, cannot enqueue')

def dequeue(q):
    if is_Queue(q):
        if not empty_Queue(q):
            queue_Contents(q).pop(0)
        else:
         raise Exception('Cannot remove front element of\
                  an empty queue')
    else:
        raise TypeError('Argument is not a queue, cannot dequeue')
```

Here are a few examples of the use of our queue ADT.

```
>>> que=new_Queue()
>>> que
('Queue', [])
>>> is_Queue(que)
True
>>> empty_Queue(que)
True
>>> enqueue(que,(11674,'Platinum'))
>>> enqueue(que,(25819,'Gold'))
>>> enqueue(que,(44388,'Standard'))
>>> enqueue(que,(90083,'Black'))
```

```
>>> que
('Queue', [(11674, 'Platinum'), (25819, 'Gold'), (44388, 'Standard'),
(90083, 'Black')])
>>> queue_Front(que)
(11674, 'Platinum')
>>> queue_Contents(que)
[(11674, 'Platinum'), (25819, 'Gold'), (44388, 'Standard'), (90083,
'Black')]
>>> dequeue(que)
>>> que
('Queue', [(25819, 'Gold'), (44388, 'Standard'), (90083, 'Black')])
```

In this example, we are simulating a line at a bank that is servicing its credit card customers. The customer is identified in the queue as a tuple with the tuple's first element being the card number, and the second, the type of card. In our queue ADT, we add or enqueue at the right end of the list and remove or dequeue from the left end.

Remember, the queue (and stack) is itself a tuple. We can neither add nor remove its components i.e. its string nor its list. We can, however, modify its list component because a list, by definition is mutable. This is why we provide a function that can access the content. Both our ADTs thus far also provide examples of the four types of functions that we can create as components of an ADT:

- constructor – creates an instance of the ADT e.g. `new_Stack` and `new_Queue`
- selector – returns a component(s) of the ADT e.g. `stack_Top` and `queue_Front`
- predicate – tests (True/False) properties of the ADT e.g. `empty_Stack` and `empty_Queue`
- mutator – modifies a component of the ADT e.g. `push_Stack` and `enqueue`

Thus all ADTs will have some combination of these types of functions.

Binary Tree

A binary tree is a data structure that is (conceptually) composed of nodes and edges. The data we wish to store is saved within a node, and we link the data contained within nodes using edges. A graphical representation of a binary tree is shown in Figure 6.3:

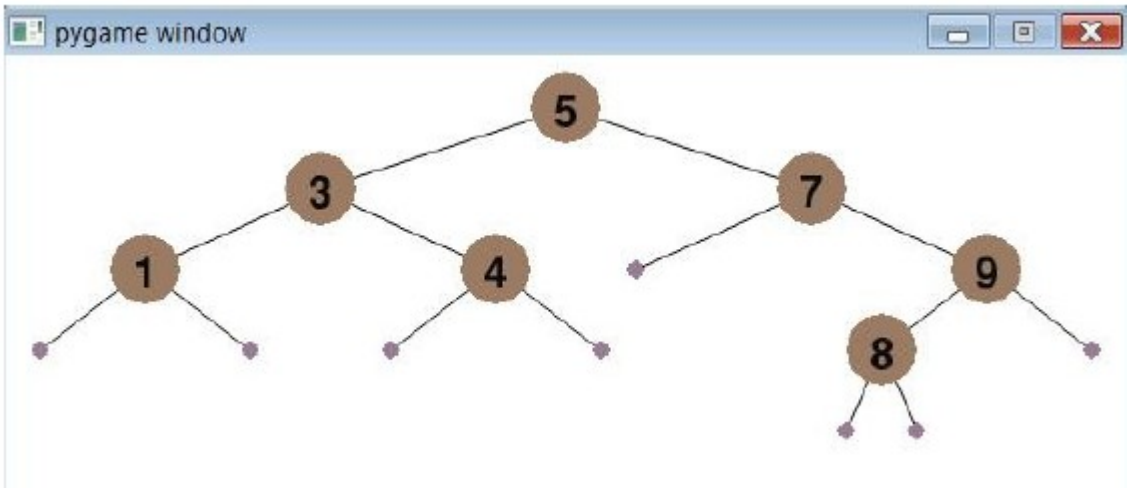


Figure 6.3: An example of a binary tree

In a binary tree, each node can have at most two sub-nodes to which it is directly connected. We refer to a node and its sub-nodes as the parent and children respectively. More specifically, the children are called a node's left and right child. The children of a node must themselves conform with the binary tree structure i.e. the must also be binary trees. A sequence of nodes and edges, or a path, is terminated by a node that does not have any data stored within it. Such an empty node is called a leaf. Within a tree data structure we therefore can have only one of two types of nodes. There can either be an empty or a non-empty node. Further, as every non-empty node must itself be a binary tree, we call a non-empty node a root. Binary trees are very useful. They enable significant advantages in sorting and searching processes. Take another look at Figure 6.3.

If we stored the set of numbers in ascending order using a tuple, we would have (1,3,4,5,7,8,9). If we were searching for the integer 1 using the tuple and beginning our search from the left, we would find it after making only 1 comparison. If we were to begin our search from the right, however, we would need to traverse the entire list, a total of 7 comparisons. The same will apply if we were to search for the integer nine and beginning our search from the right. With the binary tree in Figure 6.3 however, in *both* instances, we would make 3 comparisons.

With our tree, in this scenario we have halved the number of searches we would need to make. If we apply a tree structure to large data stores, where the location of specific data items is unknown, the advantage and usefulness of the tree structure becomes even more apparent.

To create a binary tree based on the aforementioned description we must define a structure that will enable:

- data to be stored in its root nodes
- root nodes to have at most two sub children, a left and a right which must themselves be binary trees if they are non-empty
- nodes to be empty, called leaf nodes
- logical connection of nodes.

The second point clearly illustrates the fact that a binary tree is a recursive structure; we repeatedly apply and ensure that a non-empty node must itself be the root of a binary tree. We find a bit of data within a tree by "visiting" nodes and looking at the data stored within. If we find what we are looking for we end our search; if not, we look at either the left or right child and repeat the process. We stop searching when we encounter a leaf, at which point we either look elsewhere in the tree or end the search completely if we are certain what we are looking for does not exist within the tree. In both cases, looking elsewhere or abandoning the search depends on how the tree was constructed.

Because we may change what is stored in a tree, we need a data type that will permit modification. This requirement eliminates the immutable types. We must also be able to impose a sequence among nodes. This will represent the edges that connect the nodes in our tree. The list seems to be a very good fit. The code for a representation and a few functions of a binary tree ADT are presented below. Based on our previous discussions on the artistic nature of programming, using a list and the code provided below are not the only ways that the binary tree ADT can be created.

```
def metr():
# make an empty binary tree. This will also represent a leaf node
    return ('bTree',[])
```

```
def isBtree(tree):
# does the structure conform to the binary tree representation?
  return type(tree)==tuple and tree[0]=='bTree' and type(tree[1])==list\
          and len(tree)==2

def isEmpty(btree):
# is the tree empty?
    if isBtree(btree):
        return btree==('bTree',[])
    else:
        raise TypeError('Argument is not a binary tree')

def mtr(r,lt,rt):
# make a tree with a non-empty root and (possibly) non-empty children
    return('bTree',[r,lt,rt])

def getRoot(btree):
# retrieve the root of a tree
    if isBtree(btree):
       return btree[1][0]
    else:
        raise TypeError('Argument is not a binary tree')

def getLT(btree):
# retrieve the left child of a tree
    if isBtree(btree):
        return btree[1][1]
    else:
        raise TypeError('Argument is not a binary tree')
```

```
def getRT(btree):
# retrieve the right child of a tree
    if isBtree(btree):
        return btree[1][2]
    else:
        raise TypeError('Argument is not a binary tree')
```

With the binary tree, we have the opportunity to demonstrate the effect of abstraction violation. In the examples shown below, you will notice that when we build a tree, we must remember its recursive nature. The code will permit a tree to be created without such a structure but the ADT breaks down when we attempt to use it based on the binary tree's definition. In the invocations below, note what happens when we define the binary `tr` and attempt to access the data stored in one of its children without violating the abstraction:

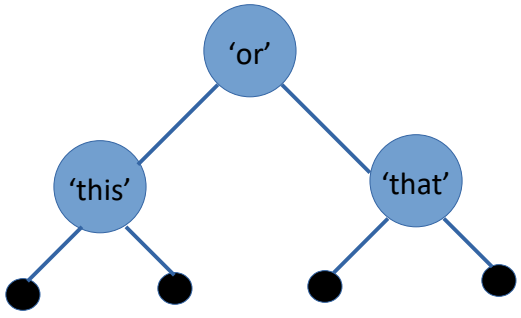
```
>>> egTree=mtr(5,mtr(3,metr(),metr()),mtr(7,metr(),metr()))
>>> egTree
('bTree', [5, ('bTree', [3, ('bTree', []), ('bTree', [])]),
('bTree', [7, ('bTree', []), ('bTree', [])])])

>>> isBtree(egTree)
True
>>> isEmpty(egTree)
False
>>> getRoot(egTree)
5
>>> getLT(egTree)
('bTree', [3, ('bTree', []), ('bTree', [])])
>>> getRT(egTree)
('bTree', [7, ('bTree', []), ('bTree', [])])
>>> getRoot(getRT(egTree))
7
```

```
>>> tr=mtr(5,3,7)
>>> tr
('bTree', [5, 3, 7])
>>> getRoot(tr)
5
>>> getRT(tr)
7
>>> getRoot(getRT(tr))
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    getRoot(getRT(tr))
  File "C:\scripts\btreeADT.py", line 38, in getRoot
    raise TypeError('Argument is not a binary tree')
TypeError: Argument is not a binary tree
```

We are not limited to what we can store in our tree as defined. We can therefore store strings if we need to. We are only limited by the underlying list data type.

```
>>>strTree=mtr('or',mtr('this',metr(),metr()),mtr('that',metr(),metr()))
>>> strTree
('bTree', ['or', ('bTree', ['this', ('bTree', []), ('bTree', [])]),
('bTree', ['that', ('bTree', []), ('bTree', [])])])
```

Finally, `strTree` can be graphically represented as:



Having investigated data abstraction and taken a look at a recursive structure, we will now investigate recursive procedures. Recursion in code, as we will see, is simply another tool we can use to implement repetition in our programs.

Chapter Seven

Recursion

Fractal - a never ending pattern that repeats itself at different scales and is made by repeatedly performing a simple process.

Nature is magnificent. As has happened numerous times over many centuries, we may find solutions to many problems by querying nature. The complexity that is a living thing is fascinating. We are only just beginning to scratch the surface in our understanding of not just that trees may communicate with each other, but how. Using things such as pheromones and underground fungal networks, trees communicate with each other.

Figure 7.1 is a photograph that I took of a small fern that grows in my garden. Notice how the branch's leaves form a pattern that is a smaller version of the branch itself?

Figure 7.1: Recursion in nature

This is very similar to what we discussed in the previous chapter when we investigated binary trees. Recall that a non-empty child node is itself a binary tree. On the fern, the branch's leaves are themselves miniature branches. The fern has solved its "problem" of building itself by repeatedly following the same basic pattern, with each iteration of the repetition producing a smaller version of the pattern until it produces a true, single leaf. This approach is not unlike our general approach to problem solving.

As we discussed earlier, we take a problem, break it up into smaller sub-problems, solve the smaller sub-problems and combine the solutions to solve the original large problem. A recursive process adopts the same approach.

With a recursive process we take a problem and develop a general procedure towards solving it. We repeat the general procedure each time with a smaller subset of the initial problem until we arrive at a subset for which a solution is known. The known solution in our recursive process is analogous to our fern's leaf, or when we search a binary tree and either find the value we are searching for or arrive at a leaf node. Let us now consider a small example.

The factorial of a positive integer is the product of all the positive integers less than the integer. The product also includes the integer itself. We use the exclamation symbol (!) after a positive integer to denote that we wish to calculate its factorial. Examples of factorials are:

1! = 1

2! = 2x1 = 2

3! = 3x2x1 = 6

4! = 4x3x2x1 = 24

N! = N x (Nx1) x (N-2) x...x 1

In the last example, N can be any positive integer. Upon inspection, calculating the factorial of a positive integer is a recursive process. Consider 4! again.

4!=4x3x2x1

The right side of the equation can be rewritten as seen below without altering the result in any way as multiplication is associative:

4!=4x(3x2x1)

But what we have now enclosed in parentheses is actually 3! and when we repeat the insertion of parentheses on the calculation of 3!, we find it becomes 3x(2x1) and what is now enclosed is in fact 2!. We will stop when we arrive at 1! as we know that the solution here is 1.

In this case, 1 is our “terminal leaf”. The pattern that emerges here provides us with a general procedure for calculating the factorial of any positive integer N.

N! = N x (N-1)!

The recursive nature of the procedure is evident with this equation as we define the calculation by stating that it involves calculating smaller and smaller “versions” of itself until, again in this case, we arrive at 1! This is intriguing as this suggests that were we to implement this directly in code as a function then the function must, by definition of our general procedure for calculating the factorial, invoke itself from within its own code. This ability of a function or procedure to call itself from within its own code is included in many programming languages including Python. To successfully implement recursion in our code we must obey the following. There must be:

1) code that implements a condition(s) that will cause the recursion to end. Such code is referred to as the recursive procedure’s base case(s).
2) code that implements the recursive call(s) within the procedure itself. Such code is referred to as the recursive procedure’s recursive case(s).
3) a change in the argument(s) that are processed by the recursive case(s) so that they eventually satisfy a base case.

If we implement a procedure or function that does not satisfy condition 2 above, it is not recursive. It we implement a recursive procedure or function that does not satisfy condition 3 above, the repetition created by the recursive calls will not end. In such a scenario, the IDE may terminate the program or the program will continue to loop until the device on which it is executing runs out of available memory and causes the device to stop responding to user interaction i.e. it “crashes” or “freezes” the device.

Here is a recursive function that implements the calculation of the factorial of a positive integer.
We cannot use the same exclamation symbol nor can we place a symbol after an integer as we wrote it earlier because of Python’s syntax rules.

```
def _factorial(N):
    if N==1:
        return 1
    else:
        return N*_factorial(N-1)

>>> _factorial(4)
24
>>> _factorial(5)
120
```

It is useful to know the basics of how recursion is implemented in a language. We have already discussed two main components of such an implementation, device memory and a stack.

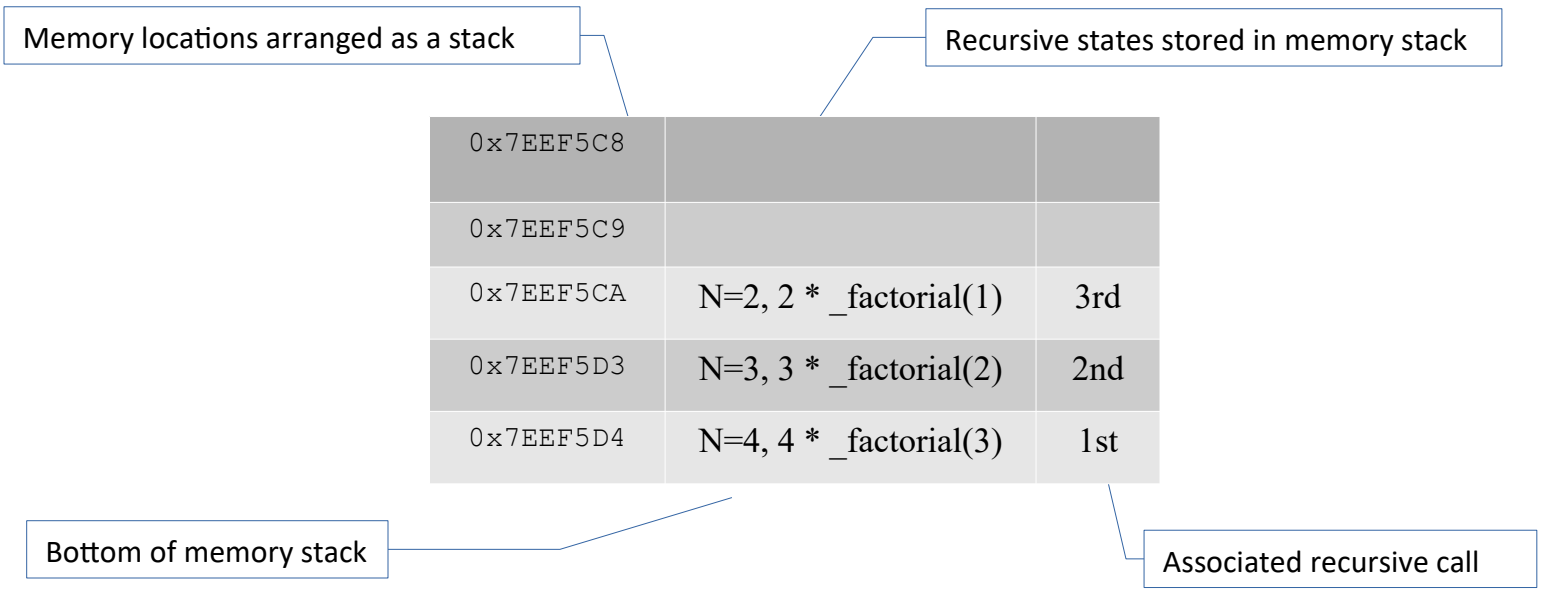


Figure 7.2: How recursion works

Let us step through how `_factorial` accomplishes its task.

```
>>>_factorial(4)
def _factorial(N):
    if N==1:  ← this is the base case
        return 1
    else:
        return N*_factorial(N-1)← this is the recursive case
```

Upon invocation the argument 4 is bound to the parameter N. In this 1st pass through the code N does not equal to 1 so the program execution jumps to the else, the recursive call. Substituting 4 for N the expression becomes,

4 * _factorial(3)

There is no solution *at this point* in execution for _factorial(3). The evaluation of 4 * _factorial(3) is therefore put on hold or deferred until there is a solution for _factorial(3). This is accomplished by saving the current values of the variables and other information in this pass of the function i.e. the function's state. The state is saved by pushing the associated data onto a memory stack as depicted in Figure 7.2. The function then calls itself with a new argument, 3. We can already see that this continued reduction in the value of N will eventually result in its attaining the value of 1 and satisfying the base case's condition. At this point we have shown that the 3 rules for successfully writing a recursive process have been obeyed: we have a base case, a recursive case and change in the argument such that it will satisfy the base case's condition.

On the 2nd pass substituting 3 for N the expression becomes,

3 * _factorial(2)

Again there is no solution at this point in execution for _factorial(2), and this expression is another deferred operation. The state of the function in this call is pushed onto the memory stack and the function calls itself again with a new argument, 2. On the 3rd pass substituting 3 for N the expression becomes,

2 * _factorial(1)

This is yet another deferred operation. The steps of pushing the state onto the stack and calling the function with a new argument are repeated.

On the 4th pass substituting 1 for N we satisfy the base case's condition. This provides the first real result in our calculations. At this point we have three deferred operations stored in the memory stack awaiting results of the evaluation of calls to the recursive function. We can also see how recursion provides another method that we can use to achieve repetition in our code and how we incorporate our problem solving approach. We began with the "large problem" of calculating the factorial of the integer 4 and have repeatedly made it smaller until we have arrived at a solution we know, the factorial of 1. Follow all of the blue arrows first then a green followed by a red one in figure 7.3.

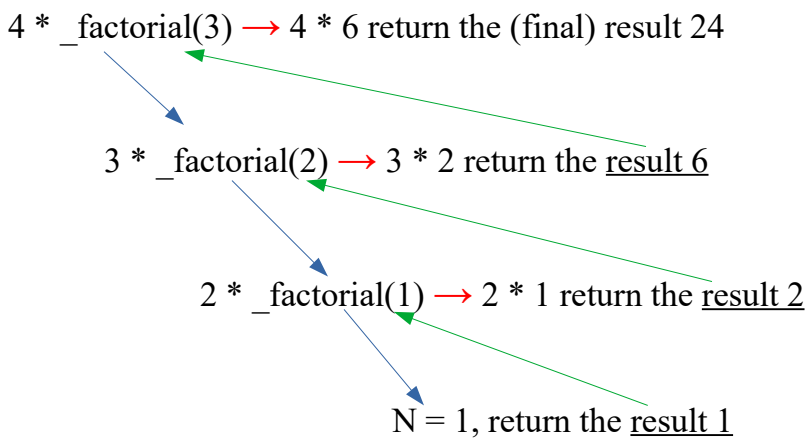


Figure 7.3: Stepping through an augmenting recursion process

As the green arrows in figure 7.3 show, the results are each returned to the invocation of the function that awaits that resides in the memory stack. As we expect based on our discussion of stacks, we pop the stack one element at a time. The result 1 is therefore returned as the result of the invocation _factorial(1) that is a part of the deferred expression 2 * _factorial(1) stored in the memory stack. This continues until all stack elements are popped and the final result of 24 is returned as expected.

```
>>> _factorial(4)
24
```

The factorial function then, is inherently recursive. Nevertheless there is a drawback. The number of memory locations used in the stack and the number of evaluations will increase as the integer N increases. In our small example, we depicted the use of N-1 memory locations in the stack. With this we can estimate for example that the recursive calculation of 100! would require 99 memory locations and therefore the rate of growth in memory usage grows linearly as N does.

When we have a recursive call with a deferred operation, we refer to the recursion as augmenting. We have seen more efficient ways of performing iterations using for and while loops. Using a for loop for example we could write a solution as shown below. In this solution, regardless of the size of `N` we only utilise two memory locations and repeatedly change the values stored in them, `result` and `intgr`. This solution is superior in its use of memory resources than the augmenting recursion, but writing it is not as intuitive as was the case with the recursive form.

```
def _factFL(N):
    result=1
    for intgr in range(1,N+1):
        result*=intgr
    return result
```

This is a distinct benefit in having multiple ways of achieving the same task, the task in this case being to achieve repetition in our programs. We can write a solution using what is most intuitive or natural to us and then rewrite it using a method that uses a device's resources more efficiently. We can in fact make a small improvement in our recursive solution by eliminating the deferred operation. To do this we use a parameter to maintain the state of the calculation instead of pushing a deferred operation to a memory stack.

To accomplish this we define a function with local scope that performs the recursion and use its enclosing function to invoke the locally defined recursive function and provide initial arguments. With this approach we eliminate the steps of popping the deferred operations from the memory stack and evaluating these associated expressions and in so doing, reduce the overall number of steps involved in the augmenting recursive form.

In Figure 7.3 we will eliminate the steps depicted with the green arrows. We refer to this improved method of recursion as tail recursion. The function `_factTR` below implements tail recursion in its solution to calculating the factorial of a positive integer.

```
def _factTR(N):
    # calculate the factorial using tail recursion
    def loop(intgr,result):                          #Line 1
        if intgr==1:
            return result                            #Line 2
        else:
            return loop(intgr-1,result*intgr)        #Line 3
    return loop(N,1)                                 #Line 4
```

In `_factTR` our locally defined recursive function is called `loop`. `loop` is invoked in line 4 and this invocation also provides an initial pair of arguments to `loop`, the value of `N` and the value 1 which we use to initialise the parameter `result`. `result` is the parameter we use to maintain the state of our calculation (line 1). Notice that in this solution there is always a value for N, `intgr` and `result`. There are no expressions that would result in deferred operations. We still obey the rules for successful recursion however. We have a base case, a recursive case and `intgr` changes so that it will eventually satisfy the base case's condition.

We update the values stored in `intgr` and `result` in the recursive call, line 3. This should not be foreign to you as you should recall that in an earlier discussion we not only assigned a value to a parameter in a function's definition, we did so by invoking another function. Remember `_genKey` in our solution for the Caesar cipher? In line 3 then, we are simply modifying the values stored in the variables `intgr` and `result` and using these as the new arguments to successive calls to `loop`. When we satisfy the base case, we will already have the final answer stored in `result` and this answer is what is returned (line 2) at which point the recursive process ends. On the following page we have four solutions to providing the answer for the factorial of a positive integer.

Two functions provide recursive solutions, two provide iterative solutions. When we utilize non-recursive forms of repetition we refer to them as iterative. When we use each of them with integers of increasing size we get an appreciation for the relative performance of the solutions. This provides an insight into the relative efficiency in utilisation of a device's resources of space (memory and storage) and time(number of steps). When we consider resource utilisation and efficiency we speak about orders of growth.

We mentioned this in our discussion without referring to it by name. The order of growth in time, or its time complexity, by the augmenting recursive factorial function is linear. The rate of growth in the number of steps (and memory usage) is directly proportional to the growth in the input, N. We use Big O notation to express this. In this case the order of growth is $\Theta(n)$.

```
def _factAR(N):
    # calculate the factorial using augmenting recursion
    if N==1:
        return 1
    else:
        return N*_factAR(N-1)

def _factFL(N):
    # calculate the factorial using a for loop
    result=1
    for intgr in range(1,N+1):
        result*=intgr
    return result
```

```
def _factTR(N):
    # calculate the factorial using tail recursion
    def loop(intgr,result):
        if intgr==1:
            return result
        else:
            return loop(intgr-1,result*intgr)
    return loop(N,1)

def _factWL(N):
    # calculate the factorial using a while loop
    result=1
    while N>=1:
        result*=N
        N-=1
    return result
```

Because the efficiency in execution is dependent on the device, how each of these performs will differ based on what device they run on. On my device, both recursive versions produce results up to N=1000. Thereafter the amount of memory allocated by the IDE for the functions is consumed.

The error “maximum recursion depth exceeded” is returned. With the iterative versions however we get the anticipated improvement in efficiency. In both iterative versions we still receive a result with N=10000. In both cases, the result is a number that spans hundreds of lines on the device’s display.

Try these out for yourself and see how your device performs. In addition to any recursion depth errors take note of any delay in the time between invoking one of the functions and the return of the result.

In closing this discussion the fact that we were able to write both recursive and iterative versions of a solution is not coincidental. In fact, whatever we can express recursively can be expressed using iteration and vice-versa. It supports the statement that we write a solution in a form that comes naturally to us and implement it in this natural form or, if via investigation of the solution's order of growth, using a more efficient means.

It is only fitting to end this chapter as we began, marveling at nature. We mentioned trees and in an earlier chapter the art of computer programming. Here is a small program I wrote that draws a tree fractal, using recursion. Sometimes, although there was no problem that presented itself, we go looking for one to solve.

The tree fractal displayed in Figure 7.4 was created by invoking the function below with,

```
    >>>drawTreeFractal(12,30,200)
```

```
"""
Last executed using Python version 3.9
"""

import turtle
import math

def drawTreeFractal(height,angle,limblength):
    turtle.showturtle()
    turtle.penup()
    base=(0,-400)
    turtle.goto(base)
    moveleft=0
    moveright=1
    turtle.setheading(90)
    angularDisp=turtle.heading()
```

```
def drawTree(height,limblength,base,direction,angularDisp):

    def drawLimb(limblength,direction):
        turtle.setheading(angularDisp)
        if height<=1:
            turtle.pencolor("green")
        else:
            turtle.pencolor("brown")
        turtle.pensize(height)
        if direction==moveleft:
            turtle.left(angle)
            moveTurtle(limblength)
        elif direction==moveright:
            turtle.right(angle)
            moveTurtle(limblength)
        else:
            moveTurtle(limblength)

    if height==0:
        return
    else:
        drawLimb(limblength,direction)
        base=turtle.pos()
        angularDisp=turtle.heading()
        limblength-=int(limblength*0.25)
        drawTree(height-1,limblength,base,moveleft,angularDisp)
        turtle.goto(base)
        drawTree(height-1,limblength,base,moveright,angularDisp)
        turtle.goto(base)
        return
```

```
def moveTurtle(limblength):#same level of indentation as drawTree
    turtle.pendown()
    turtle.forward(limblength)
    turtle.penup()

drawTree(height,limblength,base,2,angularDisp)
turtle.done()
turtle.hide()
```

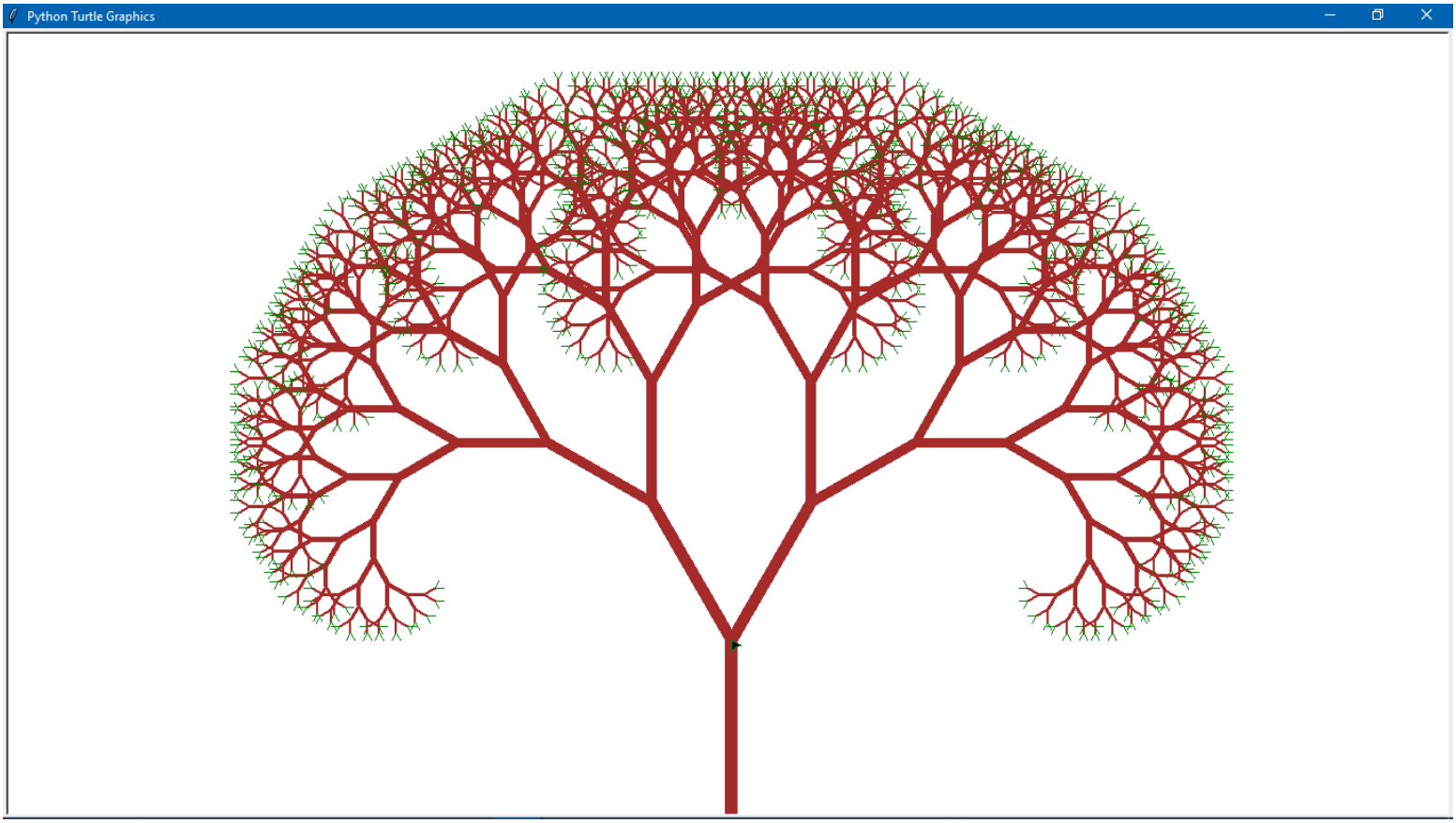


Figure 7.4: A tree fractal

Exercise 7.1

Rewrite the functions

`_multiply`

`_raise`

from chapter 3 as recursive functions.

Exercise 7.2

Write 2 functions that perform what the function `_divide` provided from chapter 3. The 1st function `_divideV2` should return the quotient of the division and the function `_modulus` should return the remainder of the division. Both `_divideV2` and `_modulus` must be recursive functions.

Exercise 7.3

We have used the inbuilt function `len` to return the length of composite data types string, list and tuple. Write a recursive function that returns the length of any string.

Chapter Eight

Higher order procedures

Throughout these discussions we have focused on solving problems by breaking them up into smaller sub-problems and solving these. We also discussed the abstraction of problems. Towards providing a more complete implementation of tools that permit a programmer to create abstractions that form the foundation of solutions to (potentially) families of problems, we need to be able to more that create data abstractions as we did when we investigated abstract data types. We also need to be able to create procedural abstractions.

Let us revisit our initial pair of functions, `_next` and `_previous`. Although you were asked to write your own version of `_previous` I have included my own. Here they are:

```
def _next(number):
    "increase a counting number by 1"
    return number+1

def _previous(number):
    "decrease a counting number by 1"
    return number-1
```

When we look at these two functions, it is quite obvious that they are quite different. `_next` adds 1 to a number while `_previous` subtracts 1 from a number. They are "moving" in completely opposite directions on the number line. Further, their names are different as are their comments. But are they *really* that different? Let us look at them from another perspective, one that sees their similarities. Both functions accept a single, integer argument. Both functions also change their argument by 1. These two features are the core of what both processes are accomplishing. The process of changing an integer by 1 is the procedure we wish to implement in code, and what differs is whether we wish to increase or decrease the integer.

To accomplish the abstraction of our procedure i.e. calling directly upon one function to change an integer by 1 regardless of whether we wish to increase or decrease by 1, we need an ability to extend the fundamental idea of a variable as something that changes to include not just data but a procedure. In this example there are two things that change, the value of the integer and whether or not that value should be increased or decreased by 1. Here is the code for such a function:

```
def _iod(number,operator):
     "either increase or decrease a number"
     return operator(number)
```

In `_iod`, the parameter `operator` is being used as a function. It is invoked with one argument, `number`. Because `_iod` accepts a function as one of its arguments, we call it a higher-order procedure (HOP). For `_iod` to successfully complete its task, the value we pass to it to be used as the operator must exist. We still cannot violate the language's rules, however, and we therefore cannot use the symbols + or – as this will generate an error.

```
>>> _iod(0,-)
SyntaxError: invalid syntax
>>> _iod(0,+)
SyntaxError: invalid syntax
```

Furthermore, the + operator is a binary one i.e., it requires two operands and therefore cannot be used in our line of code where operator is applied, operator(number). We do however already have a pair of functions where one increases a number by 1(_next) and the other decreases it by 1(`_previous`), and it was upon these that `_iod` is based. If we want to increase an integer by 1, we therefore invoke `_iod` as follows:

```
>>> _iod(1,_next)
2
```

To decrease an integer by 1 we invoke `_iod` as follows:

```
>>> _iod(1,_previous)
0
```

Formally, an HOP is one that either accepts at least one procedure or function as an argument or returns a procedure or function as a result. `_iod` above is an HOP because it accepts one function as one of its arguments.

Take another look at `_iod`. You will notice that based upon its definition we are not limited to only increasing or decreasing by 1 as we previously did. In fact, we can perform *any valid* operation on its parameter `number` that satisfies the application of its other parameter `operator`. We could square, halve and cube an integer for example. To further generalize the solution that is `_iod` it would be rather useful if we could use something that permits us to dynamically create a function!

In Python and other languages we can indeed create functions. In Python, we can use the lambda keyword to create a function that has no name, an anonymous function, and that can be defined at the point where arguments are being provided. The syntax for creating a lambda function is the keyword `lambda`, followed by one or more parameters separated by commas, a colon and then what the function should do. For example an anonymous function that squares a number can be written as `lambda x:x**2` and one that adds two numbers (or concatenates two characters or strings) can be written as `lambda x,y:x+y`. Therefore to square a number using `_iod` we can invoke it with:

```
>>> _iod(2,lambda x:x**2)
4
```

HOPs frequently used with lists

Now that we can develop both data and procedural abstractions in our code, combining the two provides us with tremendous improvement in the types of solutions we may create. There are a few procedures that are used quite often and learning about them will be useful.

Mapping

When we map, we apply one function to each element of a list and we produce a new list with the result of each of the function applications. For example, if we had a list of integers and we wanted to obtain the square of each of them, we can map a square function over the list.

- List → [1,2,3,4,5]
- Square function written using lambda → lambda x:x**2
- mapping procedure → [1,2,3,4,5]

(lambda x:x**2)

- New list with results → [1,4,9,16,25]

The mapping procedure is therefore a HOP. It accepts a list and a function as its arguments, applies the function to each element of the list and returns a new list as its result. Below is a recursive solution for a mapping function:

```
def _map(lst,fn):
"map fn onto lst"
     if lst==[]:
          return []
     else:
          return [fn(lst[0])]+_map(lst[1:],fn)
```

When we invoke _map with the arguments provided in the example above we get the expected result:

```
>>> _map([1,2,3,4,5],lambda x:x**2)
[1, 4, 9, 16, 25]
```

Filtering

There are occasions when we wish to select elements of a list that satisfy a condition. The selection or filtering of list elements can be achieved through the use of a predicate function. In a manner that is similar to mapping, the predicate is applied to each element of a list, but unlike mapping, only those list elements that satisfy the predicate are included in the new list that is produced. For example, given the same list [1,2,3,4,5] we used to introduce mapping, if we wanted to filter only those elements that are prime numbers, we could accomplish this by creating a predicate function that determines if its argument is prime, and map the predicate over the list elements. Here is a solution with a locally defined recursive function that determines whether or not its integer argument is prime. It works fairly well for relatively small integers.

```
import math
def _isPrime(p):
    def findDiv (n):
        if p%n==0:
            return False
        elif n > int(math.sqrt(p)):
            return True
        else:
            return findDiv(n+1)
    if p<=0:
        raise TypeError("primality is undefined for provided argument")
    elif p==1 or p==2:
        return True
    else:
       return findDiv(2)
```

When supplied with a few test cases it works as we expect:

```
>>> _isPrime(841237794)
False
>>> _isPrime(1000)
False
>>> _isPrime(997)
True
>>> _isPrime(0)
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    _isPrime(0)
    File  "C:\Users\ccrbu\Documents\Professional\UWI\Python  scripts\
_isPrime.py", line 11, in _isPrime
    raise TypeError("primality is undefined for provided argument")
TypeError: primality is undefined for provided argument
```

We must now develop a filter function that takes two arguments: a list and a function. Our filter function must select and return only those list elements that satisfy the predicate. We will not include a test that the function provided as an argument is a predicate.

```
def _filter(fn,lst):
    if lst==[]:
        return []
    elif fn(lst[0]):
        return [lst[0]]+_filter(fn,lst[1:])
    else:
        return _filter(fn,lst[1:])
```

In this recursive solution the predicate function `fn`, is used to test each element of the list, `lst`. If the element satisfies the predicate, it is included in the new list that will be returned as the result. If it does not, the element is ignored and the filter function continues with the next element in the list. It ends when there are no more elements to process i.e. the list becomes empty. We are now in a position to combine these two functions to produce a list of prime numbers from our list [1,2,3,4,5]:

```
>>> _filter(_isPrime,[1,2,3,4,5])
[1, 2, 3, 5]
```

Folding

If we take a closer look at our filter and map procedures, we will notice that there are similarities. Whenever we notice similarities we may also see opportunities to generalize the procedures through abstraction. The similarities in mapping and filtering are hinting that there may be an opportunity to create a generalized list processing procedure. This is interesting!

Earlier, when we investigated our `_next` and `_previous` functions we realised that their essence was changing a number by one. That is what was similar between the two. Their difference was in whether the number was increased or decreased by one. This was then captured in our HOP `_iod`. What then is the essence of map and filter? They both take a list and a function and they both return a new list. If you realised these similarities, well done. But we can generalise further.

The essence of both procedures is taking each element of a list and doing *something* with it until there are no more elements to process, and then return a result. We can capture this process in an algorithm,

1) If there are no more elements, stop the process and return the result, otherwise continue with step 2
2) Take the first element of the list
3) Perform the required task with this element
4) Adjust the final result if necessary with the intermediate result of performing the task on this element
5) “Discard” this element and continue with step 1

The steps above provide a generalized approach for processing any list. If we can capture this in a bit of code, we will have achieved our goal of creating a solution for the family of problems that involve list processing and not just its individual members such as map and filter.

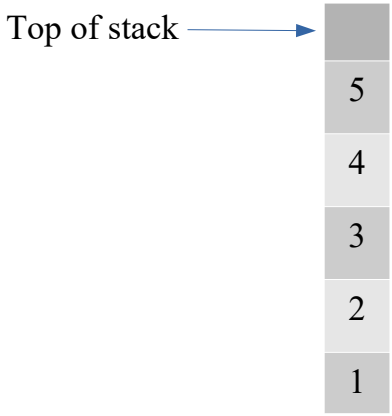
The steps seem naturally recursive. Step 1 provides a base case for our generalized procedure. Check if we have any more elements to process. The recursive process involves doing something with list one element at a time. What will differ from list to list will be the type of elements and therefore what we return when we reach the base case, and what is the *something* we do when we process each element.

Our general list processing procedure will therefore have three parameters i.e. three items that will vary from list to list when processed. What should be returned when we reach the stopping condition, the list itself and a function that will perform the task we want on each element of the provided list. We refer to this generalized process as folding a list. Recall that we can traverse a list beginning from either the left or the right and as a consequence we may fold from either the left or the right. Let us consider an example.

We would like to find the sum of a list of integers. Let us continue using our earlier list [1,2,3,4,5], the sum of which is 15. We will process this list by beginning our traversal from the left. Let us refer to the list as lst. The first pass of this process progresses as follows:

1) The list is non-empty so we continue
2) The 1st element is in position 0, lst[0] =1
3) We will add this value to a running total. As we are processing recusively, we will utilise a deferred operation to store this running total
4) We now ignore the list element 1 by passing the remaining list elements as the new argument to a recursive call. The list argument to this recursive call is [2,3,4,5]
5) Repeat, beginning at step 1.

The recursive line to calculate the sum could be partially written as lst[0] + fn(lst[1:]), where fn is whatever name we would assign to the recursive function we are conceptualizing to calculate the sum of the list elements. After the pass that extracts the value 5 from the list, the memory stack used in the recursive solution would include the five elements. They could be visualized as below:



and the set of deferred operations can be visualised as

```
1 + fn([2,3,4,5])                  deferred operation 1
     → 2 + fn([3,4,5])             deferred operation 2
          → 3 + fn([4,5])          deferred operation 3
               → 4 + fn([5])       deferred operation 4
                    → 5 + fn([])   deferred operation 5
```

At this point there are no more elements in the list. What should be returned? As we are finding the sum, we would like a value that can be added to the others without altering the final result. When performing addition, that (identity) value is zero, so this is what we should return in this *particular* case when we arrive at the base case. It is at this point when the folding of the list truly begins. The result of fn([]) should therefore be zero. By substituting zero for fn([]) in deferred operation 5 and performing repeated backward substitutions, we have:

```
5 + 0 = 5
     → 4 + 5 = 9
          → 3 + 9 = 12
               → 2 + 12 =14
                    → 1 + 14=15
```

This is our expected result. The trick in writing solutions using a fold is therefore to realise that the processing of elements does not actually begin until we combine the value returned as the base case result with the last element of the list. In this example those values are 0 and 5 respectively.

When we fold, we must therefore give some thought as to how we combine elements and a good way of doing this is to think about this at the point when the fold actually begins. Further, in this example we began folding with the rightmost element 5 and as such we say we folded from the right. Had we begun traversing from the right and began the actual fold with the leftmost element, we would say we folded from the left.

Taking the three parameters into consideration and that we always combine two elements at time during our folding procedure we can write a fold function. This is a function that implements a fold from the right:

```
def _foldr(fn, base, lst):
"fold a list from the right"
    if lst == []:
        return base
    else:
        return fn(lst[0],_foldr(fn, base, lst[1:]))
```

To utilise our `_foldr` function to find the sum of our list [1,2,3,4,5], our combiner function will take two values and add them. Our base value will be 0 and the list provided will be [1,2,3,4,5]. We can use lambda to define a function to be used as the combiner.

```
>>> _foldr(lambda x,y:x+y,0,[1,2,3,4,5])
15
```

Note that as defined in the recursive `_foldr` statement, we assign x and y from our lambda function to `lst[0]` and `_foldr(fn,base,lst[1:])` respectively. It is important to note therefore that when conceptualizing how the list elements will be combined/processed that the lambda parameter x will always be assigned the values of lst[0] in succession and that the first value of the lambda parameter y will always be that of the `_foldr` parameter base. With this in mind, consider using `_foldr` to determine the length of any list, regardless of the types of elements one may contain.

In this case:

1) We are not concerned *at all* with what the *value* of a list's elements are
2) We are concerned with the fact that an element exists, and each time we encounter one we will add 1 to a total
3) When we get to the stopping clause, as we are finding a sum we do not want the base value to alter the final sum. We should therefore return 0 as the base value
4) Based on item 1 above, our lambda variable x will not be used in our processing and the lambda variable y's initial value will be 0

Using `_foldr`, we can therefore determine the length of a list as below:

```
_foldr(lambda x,y:y+1,0,lst)
```

When invoked with our list [1,2,3,4,5] produces expected results

```
>>> _foldr(lambda x,y:y+1,0,[1,2,3,4,5])
5
```

Finally, here are `_map` and `_filter` rewritten by incorporating a call to the `_foldr` function

```
def _map(f,lst):
    return _foldr(lambda x,y:[f(x)]+y,[],lst)

def _filter(f,lst):
    return _foldr(lambda x,y:[x]+y if f(x) else y,[],lst)
```

This version of `_filter` incorporates the use of an inline if/else statement in the lambda function and is read as, if f(x) is True then [x]+y else y.

List comprehension

When we specifically need to produce a new list based upon the elements of another list as was the case with mapping and filtering we have another option available in some programming languages. The approach is based on mathematical notation and unsurprisingly the term list comprehension was first associated with and implemented in, functional programming languages. An example of a functional programming language is Haskell. Although Python is not a functional language it does provide the means to write list comprehensions. It does this by using syntax and a special form of a `for` loop. The Python list comprehension takes the following form,

```
new list = [expression for element in list if condition]
```

The if statement is optional. List comprehensions provide a concise syntax for producing new lists from another list. We can rewrite the solution to map the square function over our example list [1,2,3,4,5] as:

```
>>> [x**2 for x in [1,2,3,4,5]]
[1, 4, 9, 16, 25]
```

or using the range function:

```
>>> [x**2 for x in range (1,6)]
[1, 4, 9, 16, 25]
```

and rewrite the solution to filter the prime numbers from the same list as:

```
>>> [x for x in [1,2,3,4,5] if _isPrime(x)]
[1, 2, 3, 5]
```

where `_isPrime` is the function that was written in the earlier section where we introduced filtering and we could also have used the range function to generate the list [1,2,3,4,5]. This brings to a close our notes on an introduction to computing. With the ideas, methods and tools discussed you will be well on your way to developing solutions to many problems you will encounter.

The discipline of computing is a truly rewarding one to engage in. Have fun!

Chapter Nine

A brief introduction to quantum computing

Throughout these notes the problems we have looked at were relatively small and well defined. From the very beginning we also introduced algorithms and stated that they are fundamental to the discipline of computing. When we investigated recursion we also looked at the relative efficiency of solutions. This may lead you to ask a very important question, can we develop efficient algorithmic solutions for every known problem?

It turns out that many problems can be solved by developing an algorithm, but not all. Furthermore, even those that can be solved with an algorithm can necessarily be computed efficiently. A well-known example of this is finding the prime factors of a large number. There are algorithms that exist that can be used to solve this problem but using the most powerful computers cannot do so in this and many future lifetimes. This makes the solutions inefficient. Problems that can be solved quickly using an algorithm are members of a class of problems called P, because they can be solved in polynomial time.

The problem of calculating prime factors is not in P. This problem is at the heart of a well-known, widely used algorithm for providing security in many domains. The algorithm is called RSA. Because of its wide usage it is almost constantly under “attack”. As computers become more powerful, the numbers that a solution that implements the RSA algorithm uses must grow. At the time of writing these notes, the main output of an RSA implementation, a key, must contain at least 2048 binary digits to be considered secure.

The issue of developing algorithms for solutions for problems that are not in P has existed for decades. Many of the solutions are important and if we could find more efficient ways of computing them, we could make astonishing breakthroughs. Enter quantum computing.

The concept of a quantum computer is fundamentally different from a contemporary one. At the core of the differences is the representation of data. In all contemporary computers all data is represented using the binary number system. A single binary digit called a bit, can be used to store the value of 0 or 1, nothing else. These values are typically represented by two different voltages in the circuits we use to realise the computers that we use. There is never any other value a bit can store; it is *always either* 0 or 1.

In a quantum computer, the data is represented using a single quantum bit, called a qubit. A qubit, unlike a bit, can be used to store a 0, a 1 or anything in between those two values. This is the power behind a quantum computer. To date, we use one of a few types of quantum particles as qubits, for example electrons and photons. More specifically, we use certain properties of these particles. With electrons we use a property called its spin and with photons we use a property called polarization. Let us consider how we can use a photon as a qubit.

Photons are the particles of which light is composed. Roughly speaking, a photon has a property that tells us the direction in which it is vibrating. That property, its polarization, can be linear. You can think of linear polarization as being analogous to the x and y axes on a plane. The x and y axes are perpendicular to each other. With this property we can assign the value of 1 if a photon's vibration is oriented in the direction of the x axis and 0 if it is oriented in the direction of the y axis. This representation of data is exactly the same as a bit so how does this give us the power we expect from a quantum computer?

It lies in the fact that a photon can be placed in a state that lies "somewhere between" 0 and 1. That state actually represents a probability of whether the photon will be in the 0 or 1 state when the photon is measured to determine its polarization. We refer to and represent this state mathematically as a superposition of the two states 0 and 1 and refer to the states 0 and 1 as the basis states. It should be noted that these are not the only possible basis states. The superposition of states is one of three very important and fundamental concepts in quantum computing. The other two are entanglement and interference.

Whereas we use the digits 0 and 1 to represent the value of a bit, we cannot use the same to represent the value of a qubit as there can be a superposition of those two states. Instead, we use vectors. A qubit's states 0 and 1 can thus be represented by the following column vectors:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

The way we write $|0\rangle$ and $|1\rangle$ is using the Dirac notation for vector representation and we call them ket zero and ket 1 respectively. We may completely represent the state $|\psi\rangle$ of a single qubit by the superposition expressed mathematically using this notation and the square of the probability amplitudes α and β associated with each basis state as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \;\; where\ \alpha + \beta = 1$$

Thus, if the probability of measuring the photon in state 0 is 0.5, then the probability of measuring it in state 1 is also 0.5 as the sum of the probabilities must be 1. Remember, the square of a probability amplitude provides the probability of a state. The often related story of Schrodinger's cat has its roots in superposition but don't let the possibility of a cat being both alive and dead at the same time confuse you. It is a way of stating that at a quantum level we do not know the state of a particle, which can be accurately represented as a superposition, until we measure it. With a single qubit thus far all is well but as we mentioned in an earlier chapter, nature is magnificent!

We can completely and independently represent the quantum state of a qubit using a superposition and this holds true for systems with more than one qubit. There is, however, an exception. It turns out that we can place a pair of qubits in states such that each qubit's state in the pair cannot be independently represented i,e., one cannot be expressed without the other. For example in a 2 qubit system the combined state,

$$\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

represents such a state where each qubit in the pair cannot be expressed independently of the other. This is an example of a special state called a Bell state and provides an example of quantum entanglement. When entangled, when one qubit of the entangled pair is altered in any way, so is the other instantaneously, regardless of the distance between the two. With entanglement we will be able to overcome a number of challenges including the creation of completely secure ways of exchanging information using insecure networks.

Finally, there is interference and this is likely something we have all experienced at some point in time. When a stone is thrown into a pond or puddle of still water, it likely causes the water at the surface to ripple. Those ripples are waves and are but one example of the phenomenon. When we hear music it reaches our ears because it travels as a wave. Light also travels as a wave. Think about a room that is closed with two speakers placed along one wall of the room and you are at the wall parallel to this one as depicted in figure 9.1 below.

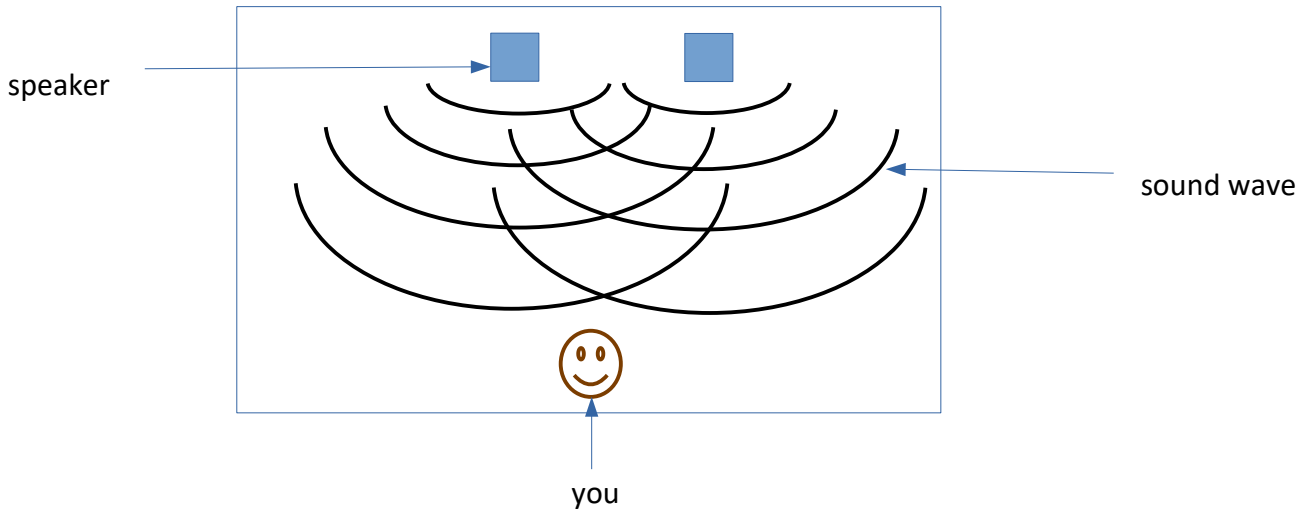


Figure 9.1 : The interference of sound waves

Depending on where along the wall you are you will either experience sound that is either slightly louder or slightly softer in volume. This is caused by the sound waves interfering with each other, for example where the lines that depict the sound waves intersect in figure 9.1. In quantum computing when we allow quantum particles to interfere, we are able to explore many outcomes. This is not practically possible using bits.

Superposition, entanglement, interference and the laws of quantum mechanics thus provide us with the potential ability to efficiently perform computations of algorithms that are not considered to be in P. There is a great deal of information on quantum mechanics and now, there are quite a bit of resources, research and effort focused on developing quantum computers. IBM has made its Qiskit platform available. This platform allows code to be written in Python and executed on one of its quantum computers that resides in its labs. Other companies that are in this domain include Google. Universities are therefore also expending their resources into the domain of quantum computing. Delft University of Technology in the Netherlands is one such example.

This brings us to the conclusion of these notes. I hope you have found them helpful and useful in your quest to learn how to solve problems, write coded solutions in a programming language and know what lies ahead in the field of computing.

--oo--