Introduction to Programming Concepts with Case Studies in Python

Bearbeitet von
Gokturk Ucoluk, Sinan Kalkan

1. Auflage 2012. Buch. x, 222 S. Hardcover
ISBN 978 3 7091 1342 4
Format (B x L): 15,5 x 23,5 cm
Gewicht: 514 g

Chapter 2
Data: The First Ingredient of a Program

The Von Neumann architecture has some implications even on high-level programming languages. Below is an overview of these aspects:

- The Von Neumann architecture makes a clear distinction between the processing unit, namely the CPU, and the memory.
- The content of the memory is highly mixed containing:

  The orders to the CPU about all the actions: Register ⇔ memory transfer operations; arithmetic, comparison and bitwise operations on the registers; operations effecting the execution flow.

  Adjunct information needed to carry out some instructions: Addresses for the transfer operations, constants involved in arithmetic or bitwise operations.

  Raw information to be processed: Integer or floating point values, or sequences of them; address information of such raw data.

  All these types of information live in the memory. However, still, there is a distinction between them. Anything that is stored in the memory falls into one of these categories, and an error-free machine code, when executed, will consider this distinction: Actions will be treated as actions, adjunct information as adjunct information and raw information as raw information.
- Access to the memory is strictly address-wise. If you do not know the address of what you are looking for, you cannot locate it unless you compare every memory content with what you are looking for. In other words: Content-wise addressing is not possible.
- All information subject to processing by the Von Neumann architecture must be transformed to a binary representation.

Among these implications of the Von Neumann architecture, the main implication is the distinction between 'actions' and the 'information' because this distinction affects the way we approach any World problem. Here, the term *World problem* refers to a problem of any subject domain, where a computerized solution is sought. Below are a few examples:

- Find all the wheat growing areas in a satellite image.

G. Üçoluk, S. Kalkan, *Introduction to Programming Concepts with Case Studies in Python*, DOI 10.1007/978-3-7091-1343-1_2, © Springer-Verlag Wien 2012

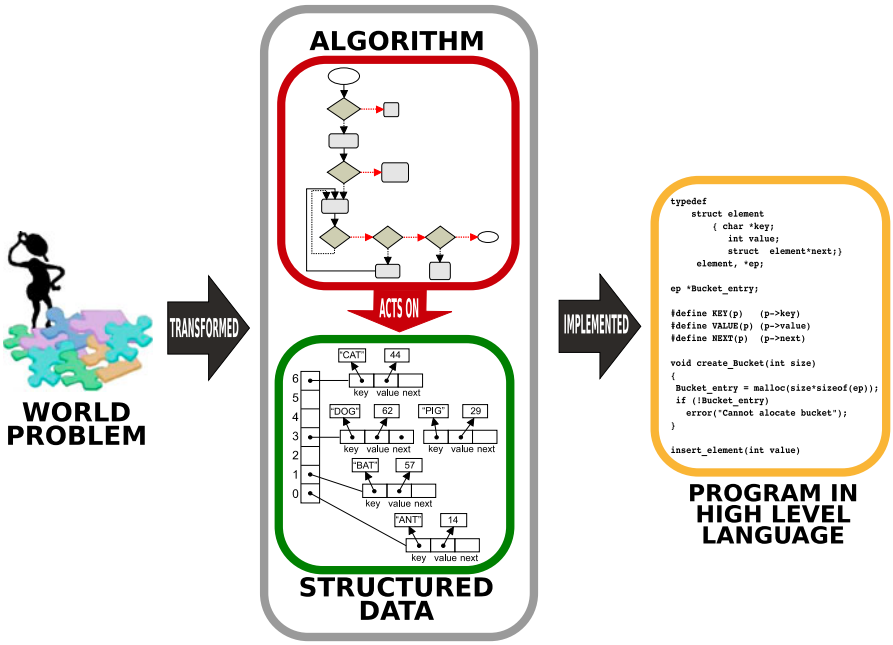

Fig. 2.1 Most of the time, a world problem is solved by a set of algorithms that act on structured data

- Given students' homework, lab and examination grades, calculate their letter grades.
- Change the amplitude of a sound clip for various frequencies.
- Predict China's population for the year 2040, based on the changes in the population growth rate up to date.
- Compute the launch date and the trajectory for a space probe so that it will pass by the outermost planets in the closest proximity.
- Compute the internal layout of a CPU so that the total wiring distance is minimized.
- Find the cheapest flight itinerary from A to B, given departure and return dates.
- Simulate a war between two land forces, given (i) the attack and the defense plans, (ii) the inventories and (iii) other attributes of both forces.

In such World problems, the first task of the programmer is to identify the information to be processed to solve the problem. This information is called *data*. Then, the programmer has to find an action schema that will act upon this data, carry out those actions according to the plan, and produce a solution to the problem. This well-defined action schema is called an *algorithm*. This separation of data and algorithm is visualized in Fig. 2.1. There can be more than one algorithm that can solve a problem and this is usually the case in practice. Actually, in Chap. 5, we will talk about the means for comparing the quality of two algorithms that solve the same problem.

2.1 What Is Data?

Practically, anything in the computer representation of a solution (for a World problem), which is not an instruction to the CPU, *i.e.*, not an action, can be called data.

Some data are directly understandable (processable) by the CPU. However, unfortunately, there are only two such data: *integers* and *floating points*. Furthermore, these data types are not full-fledged: Both integers and floating points are limited in size, which is determined by the processing capability of the CPU; *i.e.*, the range of the integer and the floating point numbers that can be represented is limited, and the limit is around 4–8 bytes at the time this book was written.

There are many more data types which are not recognized as directly processable entities by the CPU. To mention a few, we can quote fractions of numbers, real numbers of arbitrary precision, complex numbers, matrices, characters, strings of characters, distributions (statistical values), symbolic algebraic values.

Nonetheless, it is possible to write programs that implement these types of data. In fact, some of the high-level languages implement these types as part of the language: For example, high-level languages like Lisp, Prolog, Python and ML provide integers of arbitrary sizes; FORTRAN, an archaic language still in use, has support for complex numbers; BASIC supports numerical matrices; Mathematica, Matlab, Reduce and Maple provide, in addition to all of the above, symbolic algebraic quantities; and, almost all high-level languages have characters and strings.

Certainly, there are other less-demanded data types which are not provided by high-level languages but languages exist that facilitate defining new data types based on the existing ones. Therefore, it is possible, for example, to define a `color` data type with three integer attributes, `Red`, `Green` and `Blue`.

2.2 What Is Structured Data?

If the data is just a single entity then, in a low-level language, the programmer himself/herself can directly store it somewhere, a place known to him/her, in the memory. In fact, in a high-level language, there are language features that can do this automatically for the programmer. Soon, we will be looking into this subject.

What if we have more than 'one' of any type of data (*e.g.*, millions of integers)? One option is to store them in the memory consecutively, one after the other. Thus, if we want to fetch the 452389th item, we can easily compute the memory position as:

$$\begin{gathered}\langle\text{Address of the first byte of the first item}\rangle \\ + \\ (452389-1)\times\langle\text{Count of bytes occupied by a single item}\rangle\end{gathered}$$

That was easy. We call this structured data an *array*. It is a very efficient organization of the data because after we have calculated the address of the 452389th item, accessing it is just a memory fetch. This benefit is due to the Von Neumann architecture: If you know the address, the content is always provided to you in a constant and short time.

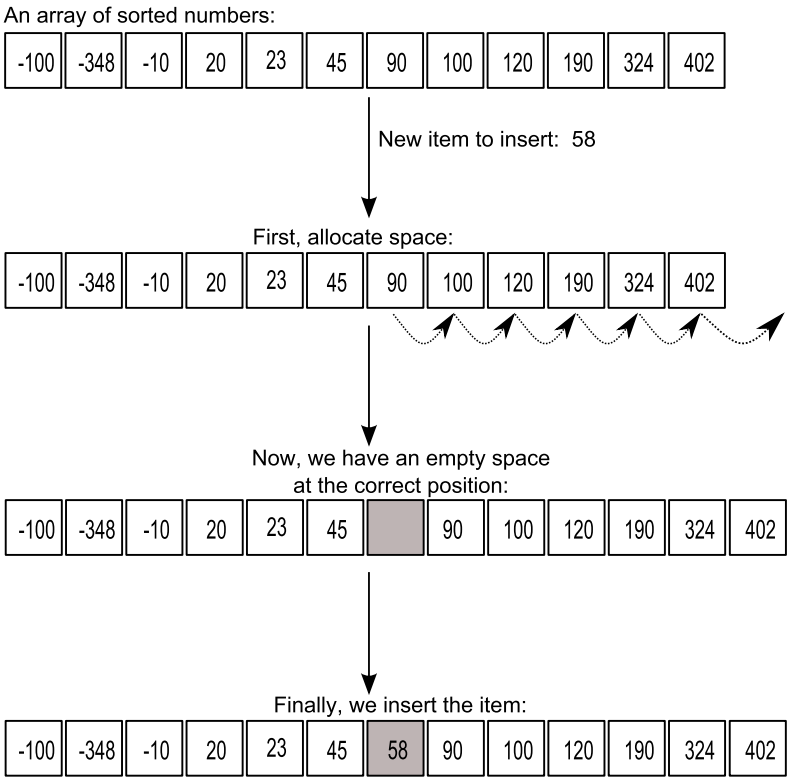

Fig. 2.2 An array becomes computationally inefficient when a new item needs to be inserted in the middle

However, what if your algorithm requires inserting some data at an arbitrary position of the array? As shown in Fig. 2.2, you would have to move all the items that are after the insertion point, and make new space for the new item. This could be the case for example if the array holds a collection of sorted numbers in order and you want to keep it sorted after the insertion. Unfortunately, the memory does not have a facility for shifting a range of its content down or up. Nonetheless, this can be performed by the CPU by shifting each item one by one. This is an extremely time consuming task especially for a large collections of data and therefore, not practicable.

Hence, we need other techniques for storing collections of data that make it easy and efficient to insert new items. The solution is to keep the data in "island"s in the memory, in a structured manner, with each island holding an information item and the addresses of the neighboring islands. In this way, data can be organized into a collection of islands, where a certain item is located by knowing how to "jump" from one island to another and by comparing the item that we are looking for with the data on the islands. This technique is extensively used in programming.

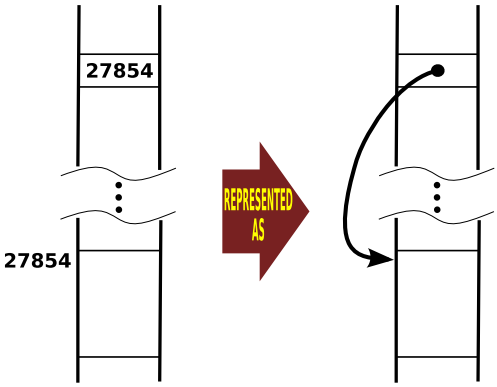

Fig. 2.3 An address reference is denoted by an *arrow*

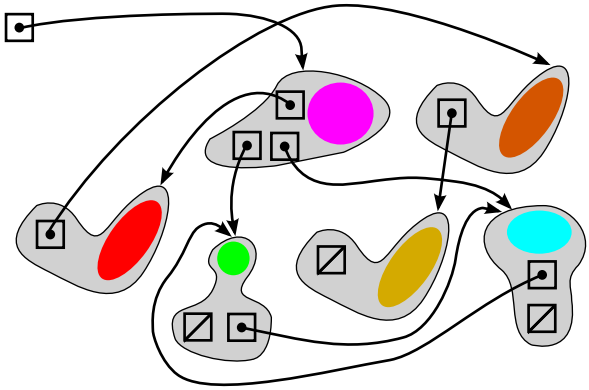

Fig. 2.4 A linked data structure *(Colors are data, arrows are links)*

Whenever, as a part of an item of data, an address is kept which is the address of another data, the address holding position is called the *pointer* (or alternatively, the *link*) to that data. Denotationally, this is represented by drawing an arrow from the address holding position to that address, as shown in Fig. 2.3.

Therefore, keeping in mind that all the data islands are stored in the memory, Fig. 2.4 provides a pictorial example of a linked data structure:

Assume that we have N-many such islands, each island keeping some information which is required for our algorithm. Having more than one link per data island and organizing the links intelligently, we can locate (search and find) any information in $\log N$ many hops (How? Think about it). Moreover, removing an island or inserting another is achieved by just modifying a couple of arrows.

Therefore, by storing and organizing the data in a particular way, we can gain efficiency in various aspects. We will see that it is possible to reduce the time spent on locating some data searched for as well as inserting or deleting a data from a vast pool of data. This is an area of Computer Science called *Data Structures*.

Some high-level languages provide mechanisms to define those islands with "linkage fields"; Prolog, Lisp and ML, for example, provide syntactic features for

automatically building and modifying such linked data structures. The 'list' construct in these languages is a good example of such embedded features.

2.3 Basic Data Types

From now on, we will use the adjective 'basic' to refer to the content of a high-level language. In this sense, 'basic' means a construct which is relatively easily implemented at the low level, *i.e.*, the machine code, and provided by a high-level language as built-in.

There are two categories of basic data:

- Numerical *(integers, floating points)*
- Symbolic (or non-numerical) *(character, boolean)*

All CPUs have support for numerical data whereas some may not support floating points. CPUs can add, subtract, multiply or divide numerical. A CPU-supported numerical data has a fixed size for representation (if the CPU uses 4 bits to represent integers, the magnitude of the maximum integer that can be represented on this CPU is around 15[1]), and this fixed size will vary from CPU to CPU. At the time of writing this book, this limit is mostly 32 bits or 64 bits.

2.3.1 Integers

Integer is the data type used in almost all discrete mathematical applications, like enumeration, counting, combinatorics, number theoretics, geometry and many more.

The most common way to representing integers is *Two's Complement* using which we can represent integers in the range of $[-2^{n-1}, 2^{n-1} - 1]$ given n bits. In Two's Complement notation, a positive number has the leading bit as `0` whereas a negative number's leading bit in Two's Complement notation is `1`. To convert a (positive or negative) decimal number x into its Two's Complement representation:

1. Convert $|x|$ into base-2, call it $b_{n-2} \dots b_1 b_0$.
2. If $x > 0$, $0b_{n-2} \dots b_0$ is the Two's Complement representation.
3. If $x < 0$,
   (a) flip each bit—*i.e.*, $c_i = 1 - b_i$ for $i = 0, \dots, n-2$.
   (b) add 1 to $c_{n-2} \dots c_0$; *i.e.*, $d_{n-2} \dots d_0 = c_{n-2} \dots c_0 + 1$.
   (c) $1d_{n-2} \dots d_0$ is the Two's Complement representation.

An important advantage of Two's Complement representation is that addition, subtraction and multiplication do not need to check the signs of the numbers. Another

[1] The exact value depends on how the CPU represents negative numbers.

important advantage is the fact that +0 and −0 have the same representations, unlike other notations.

The CPU arithmetic works rather fast (faster compared to floating points) in the integer domain. Furthermore, some obscure precision losses, which exist in representing floating point numbers, is not present for integers. Therefore, integers are favored over floating points when it is possible to choose between the two. It is frequently the case that even problems that are defined in a domain of reals (like computer graphics) is carried over to the integer domain, because of the gain in speed.

Some high-level languages do provide arbitrary precision integers, also known as *bignums* (short for big numbers), as a part of the language. In some cases, the usage is seamless and the programmer does not have to worry about choosing among the representations. Lisp, Prolog, ML, Python are such programming languages. In languages like C, Pascal and C++, facilities for bignums are available; therefore, the functionality is provided but not seamless (*i.e.*, the user has to make a choice about the representation type). As explained in the preceding section, bignums are not directly supported by the CPU; therefore, the provider has to represent them with a data structure and has to implement the algorithms himself to perform the arithmetic operations on them. Usually, bignums are represented as arrays of integers, each element of which is a digit in base-n, where n is close to the square root of the biggest integer that can be handled by the CPU's arithmetic (Why? Think about it).

It is possible that a high-level language offers more than one fixed-size integer type. Usually, their sizes are 16, 32, 64 or 128 bits. Sometimes, CPUs have support for two or three of them (having also different operations for different types of integers; for example, there are two different instructions to add integers of size 32 and 64, respectively).

2.3.2 Floating Points

Floating point is the data type used to represent non-integer real numbers. The internal representation is organized such that the number is converted into a binary fractional part and a multiplicative exponent part (*i.e.*, $F \times 2^E$, where F is the fraction, E the exponent). After this conversion, the fractional part is truncated to a fixed length in bits, and stored along with the exponent.

You might remember from your Mathematics courses that irrational numbers do not have fractional parts that can be truncated, neither do most of the rationals. In "... the fractional part is truncated...", "truncated" means "approximated". To observe this, feel free to take the square root of (let's say) 2.0, and then square the result, in any high-level language. You will never get back the answer, 2.0.

Let us have a closer look at the internal representation of floating points to understand what is going on and where the precision gets lost. Below is the IEEE 754 binary floating point standard, converting a real number into the internal representation (see also Fig. 2.5):

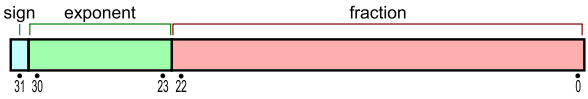

Fig. 2.5 IEEE standard for 32-bit floating point representation

1. The whole part (the value to the left of the point) and the fractional part of a real number are expressed in binary.
2. The fraction 'point' is moved to the left (or right) so that the whole part becomes exactly 1. To compensate for the move and not to alter the value of the real number, a multiplicative power of 2 is introduced. It is possible, of course, that this power is negative.
3. The binary number 1 right before the point is skipped over, and the 23 digits (following the fraction point) in the fraction part are stored as the *mantissa*.
4. 127 is added to the power of the multiplicative factor and stored as the exponent.
5. If the original real number is negative, the sign bit is set to 1, otherwise to 0.
6. The values '0', $\pm\infty$, and 'NaN' *(Not-a-Number)* are represented by some exceptional combinations of mantissa and exponent values.

Therefore, mathematically, a real number is approximated to:

$$\left(1 + \sum_{n=1}^{23} \text{bit}_{[23-n]} \times 2^{-n}\right) \times 2^{\text{exponent}-127}$$

Where exactly is the loss? The answer is as follows: if the summation were extended to infinity, then any real number could be represented precisely. However, we do not have infinite number of bits; we have only 23 of them. Therefore, the truncation after the first 23 elements in the summation causes the loss. For example, the binary representation for a simple real number 4.1 has the whole part equal to 100, and yet, the fraction part has infinitely many binary numbers: *i.e.*, 00011001100110011001 .... Hence, using only 23 bits for real numbers such as 4.1 introduces a precision loss.

Is this a big problem? Yes, indeed it is. Here are some examples of everyday problems that occur in scientific computing:

- Let us say we have 32 bits for representing floating points. Therefore, you have *only* 2^{32} real numbers that can be correctly represented. However, we know from Mathematics that even in the range [0, 1], there are infinitely many real numbers (actually, it is worse than that: to be precise, there are 'uncountably many'). In other words, uncountably many real numbers are approximated to one real number that is representable by the computer. We call this precision loss *roundoff* error.

  What makes it even worse is that we easily make wrong estimates on roundoff errors. In fact, there is no correlation between the representability in the decimal notation and representability in the binary notation. For example, to us, 0.9 might seem less prone to roundoff errors compared to 0.9375. Actually, it is just the other way around: 0.9375 is one of the rare real numbers that is represented without any loss, and 0.9, despite its innocent look, suffers from the roundoff error (take your pencil and paper and do the math! When you get tired of it, you can

go and watch the movie "Office Space" (1999), where you can learn how to make millions out of roundoff errors).

- Many numerical computations are based on multiplicative factors which are differences of two big numbers (by big numbers, we mean numbers whose whole parts $\gg 0$). The whole parts are represented in the mantissa, and as a result, the fractional parts lose precision. Therefore, for example, $(1.0023 - 1.0567)$ yields a different result from $(1000.0023 - 1000.0567)$, although, mathematically, the results should be the same (Try it!).
- The irrational number π is extensively used in scientific and engineering computations. To get a close-to-correct internal floating point representation for π, we have to type 3.1415926535897931. The sinus of π, *i.e.*, $\sin(\pi)$, should yield zero, but it does not: the result of $\sin(\pi)$ on a computer is $1.2246467991473532 \times 10^{-16}$, which is definitely a small number but not zero. Therefore, a comparison of $\sin(\pi)$ against 0.0 would fail.
- You might remember that, in your Mathematics courses, you were told that addition is associative. Therefore, $(a + b) + c$ would yield the same result as $a + (b + c)$. This is not the case with floating number computations. The losses in the intermediate computations will differ, and you will have a different result for different ways numbers are added. As an example:
  set $a = 1234.567$, $b = 45.67834$ and $c = 0.0004$:
  $(a + b) + c$ results in 1280.2457399999998,
  $a + (b + c)$ results in 1280.2457400000001.
- Precision does not mean accuracy. Assume that you are working with the (IEEE 754 standard) 32-bit float representation introduced above. You want to compute an area and you take π as 3.141. You add, multiply, divide and what you get are numbers that are precise in 7 decimal digits. However, your accuracy is not more than 4 digits in any computation that involves π, since the π value (3.141) you have taken is accurate only up to 4 digits: All those higher digits are nonsense as far as accuracy is concerned.

What is the bottom line then? Here are some rules of thumb about using floating points:

- If you can transform the problem to a problem in the integer domain, do so: As much as you can, refrain from using floating points.
- Use the most precise type of floating point in your choice of high-level language. C, for example, has `float`, `double` and `long double`, which, these days, correspond to 32, 64 and 128 bit representations, respectively.
- Use less precision floating points only when you are short of memory.
- It is very likely that you will have catastrophic roundoff errors when you subtract two floating points close in value.
- If you have two addends that are magnitude-wise incomparable, you are likely to lose the contribution of the smaller one. That will yield unexpected results when you repeat the addition in a computational loop where the looping is so much that the accumulation of the smaller addends is expected to become significant. It will not.

- The contrary happens too: Slight inaccuracies might accumulate in loops to significant magnitudes and yield non-sense values.
- You better use well-known, decent floating point libraries instead of coding floating point algorithms yourself.

2.3.3 Numerical Values in Python

Just to remind you our first interaction with Python from Chap. 1, let us have a look at a simple computation involving some numbers:

```
>>> 3+4
7
```

In this interaction, the numbers `3` and `4` are integers, and Python has a certain *name*, *i.e.*, a *type*, for all integers: `int`. If you ask Python what `int` is, it will tell you that it is a `type`:

```
>>> int
<type 'int'>
```

We could also ask the type of a constant number, or a combination of them:

```
>>> type(3)
<type 'int'>
>>> type(3+4)
<type 'int'>
```

The `int` type in Python has fixed-size representation, which depends on the CPU. If you need to work with integers that exceed the fixed-size limit of your CPU, you can use the `long` data type in Python. If you want your constant numbers to be represented as `long` integers, you need to enter the `L` letter after them:

```
>>> type(3L)
<type 'long'>
>>> type(3L+4L)
<type 'long'>
>>>
```

`int` type in Python has fixed-size representation (based on the CPU) whereas `long` type is only limited by the size of available memory.

For floating point numbers, Python has another type:

```
>>> type(3.4)
<type 'float'>
>>>
```

and similar to `int` numbers, we can do simple calculations with the `float` data:

```
>>> 3.4+4.3
7.7
>>> 3.4 / 4.3
0.79069767441860461
```

– Useful Operations on Numerical Values in Python

Let us have a look at some useful simple operations with numerical values in Python (in Chap. 3, we will look at more operations and how they are interpreted by the CPU):

- *Absolute value of a number*: The `abs(Number)` function can be used for the absolute value of a number.
- *Hexadecimal or octal representation of an integer*: The `hex()` and `oct()` functions can be used for this purpose.
- *Exponent of a number*: Typing `pow(Number1, Number2)` or `Number ** Number2` in Python results in $Number1^{Number2}$.
- *Rounding a floating point number*: The `round(Float)` function rounds the given floating point number to the closest integer.
- *Conversion between numbers*: You can use the `int(Number)`, `float(Number)` and `long(Number)` as constructors to convert a given number to `int`, `float` and `long` respectively:

  ```
  >>> long(3.6)
  3L
  >>> float(3L)
  3.0
  >>> int(3.4)
  3
  ```

  Note from the examples that converting a `float` number to an integer number results in losing the fraction part (without rounding it).

2.3.4 Characters

As stated earlier in this book, we attempt to generate computer solutions for some of our world problems. These problems are not always about numbers, they can also

be about words or sentences of a natural language. The written language is made up of fundamental units called *graphemes*. Alphabetic letters, Chinese/Japanese/Korean characters, punctuation marks, numeric digits are all graphemes, and there is a demand to have these represented on the computer. In addition to graphemes, certain basic actions of the computer-external world interactions need also to be represented on the computer: "make a beep sound", "eject the half printed paper from the printer", "end a line that is being printed and start a new one", "exit", "up", "down", "left", "right" are all such interaction actions. We call them *unprintables*. Since the semantics of graphemes is very different from numbers and numerical values, they are not at the center of CPU design. The CPUs nowadays do not have any built-in feature for graphemes.

Whenever there is no built-in representation provided by the CPU architecture, then programmers are free to make their own choice. In the case of representing graphemes, we have several options which all involve constructing a mapping from the set of graphemes and unprintables to binary representations. In more practical terms, being a computer manufacturer, you make a table of two columns: one column is for the graphemes and unprintables and the other is for the binary representations that you choose for the graphemes. You manufacture your interaction devices, the so-called input/output devices, to represent graphemes and interpret the binary representations of graphemes according to this table. A grapheme is represented in the memory in binary. For the CPU, the representation does not have any meaning—it is just a set of binary numbers. At most, without interpreting it, the CPU can aid to copy such binary data from one memory position to another or just compare whether two such binary representations are identical.

Throughout the history of computers, there has been many such tables that link graphemes with binary representations. Almost all large computer manufacturers had their own tables. Later, local authorities responsible for standardization began to standardize those tables. Table 2.1 shows the ASCII (American Standard Code for Information Interchange) table, which was defined by the American Standards Association, in 1963.

As you may have observed, the ASCII table uses 7 bits. The first 32 entries are the unprintables. For decades, the USA was the dominant computer manufacturer and this table affected the way programs were written for years. Even today, the ASCII table is still extensively used world wide, although it is solely organized to reflect a subset of the local needs of the USA. The ASCII table includes the dollar sign ($), however, apparently, it falls short to satisfy the needs of foreign trade; the symbols for sterling (£) and yen (¥) are absent. Furthermore, none of the diacritics which are widely used in European languages are included in the ASCII table; *e.g.*, the table does not have Ö, Ü, Ç, Å or È letters. The punctuation characters suffer, too. It is difficult to understand why the tilde is there but the section sign is not.

For these reasons, many countries had to extend the ASCII table and add their additional characters using the 8th bit set. In such extensions, the letter Ç has a numerical value that is not between the numerical values of letter C and D, making alphabetical sorting a problem. A solution to this problem is having another table

Table 2.1 The ASCII (American Standard Code for Information Interchange) table

NUL	00000000	SYN	00010110	,	00101100	B	01000010	X	01011000	n	01101110
SOH	00000001	ETB	00010111	-	00101101	C	01000011	Y	01011001	o	01101111
STX	00000010	CAN	00011000	.	00101110	D	01000100	Z	01011010	p	01110000
ETX	00000011	EM	00011001	/	00101111	E	01000101	(	01011011	q	01110001
EOT	00000100	SUB	00011010	0	00110000	F	01000110	\	01011100	r	01110010
ENQ	00000101	ESC	00011011	1	00110001	G	01000111	)	01011101	s	01110011
ACK	00000110	FS	00011100	2	00110010	H	01001000	^	01011110	t	01110100
BEL	00000111	GS	00011101	3	00110011	I	01001001	_	01011111	u	01110101
BS	00001000	RS	00011110	4	00110100	J	01001010	`	01100000	v	01110110
HT	00001001	US	00011111	5	00110101	K	01001011	a	01100001	w	01110111
LF	00001010	SPC	00100000	6	00110110	L	01001100	b	01100010	x	01111000
VT	00001011	!	00100001	7	00110111	M	01001101	c	01100011	y	01111001
FF	00001100	"	00100010	8	00111000	N	01001110	d	01100100	z	01111010
CR	00001101	#	00100011	9	00111001	O	01001111	e	01100101	}	01111011
SO	00001110	$	00100100	:	00111010	P	01010000	f	01100110	|	01111100
SI	00001111	%	00100101	;	00111011	Q	01010001	g	01100111	}	01111101
DLE	00010000	&	00100110	<	00111100	R	01010010	h	01101000	~	01111110
DC1	00010001	'	00100111	=	00111101	S	01010011	i	01101001	DEL	01111111
DC2	00010010	(	00101000	>	00111110	T	01010100	j	01101010		
DC3	00010011	)	00101001	?	00111111	U	01010101	k	01101011		
DC4	00010100	*	00101010	@	01000000	V	01010110	l	01101100		
NAK	00010101	+	00101011	A	01000001	W	01010111	m	01101101		

for the ordering of these letters. However, this makes working with such extension tables inherently slower. It is an undeniable fact that since many of such existing mappings are limited in size and scope, incompatible with multilingual environments, and cause programmers no end of trouble.

Years later, in the late 80's, a nonprofit organization, the Unicode Consortium, was formed with a goal to provide a replacement for the existing character tables that is also (backward) compatible with them. Their proposed encoding (representation) scheme is called the *Unicode Transformation Format (UTF)*. This encoding scheme has variable length and can contain 1-to-4 8-bit-wide components (in the case of UTF-8), or 1-to-2 16-bit-wide components (in the case of UTF-16). Gaining wide popularity, UTF is now becoming part of many recent high level language implementations such as Java, Perl, Python, TCL, Ada95 and C#.

Characters in Python

Python has a data type for a collection of characters (called 'string') and does not have a separate data type for individual character. However, taking a character as a subset of the string class (*i.e.*, a character is a string of length one), you can write programs making use of characters. We will come back to characters in Python when we introduce strings.

Although characters are not explicitly represented with a type in Python, we have the following functions that allows us to work with characters:

- The function `chr(ascii)` returns the one-character string corresponding to the ASCII value `ascii`.
- The `ord(CharString)` is the reverse of `chr(ascii)` in that it returns the ASCII value of the one character string `CharString`.

2.3.5 Boolean

Boolean is the type of data that represents the answer to questions like $3.4 > 5.6$ or $6/2 \stackrel{?}{=} 3$. CPUs has built-in support for asking such questions and acting accordingly to the answers. Therefore, the concept of *True* and *False* must be understood by CPUs. CPUs recognize 0 as the representation of *False* (*falsity*). Mostly, any value other than 0 stands for *True* (*truthness*). If the CPUs process any boolean calculation, the result will be either 0 or 1; 1 representing the *True* truth value that CPUs generate.

High-level languages have a similar mapping. Generally, they implement two keywords (like `TRUE, FALSE` or `T, F`) to represent the two boolean values.

When the interpreter or the compiler sees these keywords, it translates them into internal representations of values 0 or 1. High-level languages make additional checks to ensure that those values are created

- either by the programmer entering the keyword, or
- as a result of a comparison operation.

Although they will be represented as 1s and 0s internally, on the surface (*i.e.*, in the high-level language) these keywords will not be treated as integers.[2]

Boolean Values in Python

You can check yourself whether or not Python has a separate data type for boolean values by asking it a simple comparison:

```
>>> 3 > 4
False
>>> type(3 > 4)
<type 'bool'>
```

Therefore, Python has a separate data type for boolean values and that data type is called `bool`. The `bool` data type can take only two values: `True` and `False`.

We can use the `not` keyword for negating a boolean value. For example, `not True`, and `not 4 > 3` are `False`.

Since everything is internally represented as binary numbers, we can check for the mapping of the `True` and `False` values to the integers:

```
>>> int(False)
0
>>> int(True)
1
```

which tells us that `False` is internally represented as `0` (zero), and `True` as `1` (one).

In Python, like many high-level languages, numerical values (other than `0` and `1`) have a mapping to boolean values as well:

```
>>> not 0
True
>>> not 1
False
>>> not 2.5
False
```

[2]Except in C and its descendants. C does not provide a distinct boolean type and assumes that the integers `1` and `0` play the True/False role.

From this interaction, we understand that Python interprets any numerical other than `0` as `True`. Moreover, any non-empty instance of a container data type, which we will see in the remainder of this chapter, is interpreted as `True`; and otherwise, any data is `False`. For example:

```
>>> not ""
True
>>> not "This is some text"
False
>>> not ()
True
>>> not (1, 2, 3)
False
```

2.4 Basic Organization of Data: Containers

In Sect. 2.2, we introduced the necessity of storing collection of data in the memory and retrieving it later. Furthermore, we argued that storing the data in an intelligently organized way can provide efficiency and flexibility.

In computer science, a data structure with the sole purpose of storing elements is called a *container*. If a high-level language implements a container type, we expect the following functionalities from it:

- Construct a container from a set of elements and store them.
- Destruct a container and free the associated memory.
- Access the stored elements for use.

In addition to those aspects, if we have the right to set/delete/change the individual elements, then the container is said to be *mutable*. Otherwise, it is *immutable*.

As far as the internal representation is concerned, the simplest way to implement a container is to store the elements in adjacent memory locations. If the elements are homogeneous, *i.e.*, of the same type, then we call this an *array* representation. If they are heterogeneous, *i.e.*, of different types, this is a *tuple* representation.

In the following subsections, we will introduce three basic containers:

- Strings (mutable or immutable)
- Tuples (immutable)
- List (mutable)

2.4.1 Strings

Strings are the containers to store a sequence of characters. The need for strings is various; world problems intensively contain textual information and Computer

Science itself relies heavily on strings. For example, the implementations of programming languages themselves, compilers as well as interpreters, is partly a string processing task.

The internal representation of strings is based on storing the binary representations of characters that make up the string, scanned from left to right, in adjacent memory cells in the increasing address order. The count of characters in a string is not fixed: *i.e.*, we can have a string instance with three characters or a thousand characters. The count of characters that make up the string is named as the *length* of the string. Zero-length strings, *i.e. empty strings*, are also allowed.

Since there is a flexibility in the length of a string, there is a problem of determining where the string ends in the memory. There are two solutions for this representational problem:

1. Store at a fixed position (usually just before the string starts), the length of the string as an integer.
2. Store at the end of the string a binary code which cannot be a part of any string. In other words, put an ending marker which is not a character that could take a place in a string.

Both of these approaches do exist in high-level language implementations. For example, Pascal takes the first approach, whereas C takes the second. Both approaches have their pros and cons. The first approach limits the length of a string to be at most the biggest integer that can be stored in that fixed position which is a disadvantage. On the other hand, in this approach directly reaching the end of string is extremely easy. You just add the length of the string to the starting address of the string and you get the address of the last character in the string, that is an immense advantage. The second approach has the advantage that there is no limitation on the length of a string, however, the disadvantage is that to reach to the end of the string, you have to undertake a sequential search through all the characters in the string for the binary code of the ending-mark.

Strings are usually immutable. Inserting new characters and/or deleting characters is not possible. Altering of individual characters is technically doable but many high-level languages depreciate it. However, high-level languages provide syntactic features to form new strings by copying or concatenating (*i.e.*, appending) two or more strings. You must keep in mind that such processes are not carried out *in place* and are costly. In other words, when you concatenate a string S_1 with another string S_2 to form a new string S_3, first, the lengths of both S_1 and S_2 are obtained; then, a fresh memory space that will hold $length(S_1) + length(S_2)$ many characters is allocated. All bytes that make up S_1 are copied to that fresh memory location, then, when the end of S_1 is reached, the copying process continues with the characters of S_2. Therefore, a simple concatenation costs a memory allocation plus copying each of the characters in the strings to some new location. If the end of a string is marked by a terminator then, in addition, the length calculation requires access to each member of both S_1 and S_2.

Strings in Python

Python provides the `str` data type for strings:

```
>>> "Hello?"
'Hello?'
>>> type("Hello?")
<type 'str'>
```

The simplest thing we can do with strings is to get their lengths, for which we can use the `len(string)` function:

```
>>> len("Hello?")
6
```

– Accessing Elements of a String

Using brackets with an index number, *i.e.*, `[index]`, after a string, we can access the character of the string residing at the given index number. For example, `"Hello?"[0]` returns the character `'H'`, and `"Hello?"[4]` the character `'o'`. Note that indexing the elements of a string starts at zero and not at one. This means that to access the last element of the string, we would need to provide `len(string)-1` as the index.

Luckily, we don't need to use `len(string)-1` to access the last element of a string since Python provides a very useful tool for that: *negative indexing*. In other words, if the index number is negative, the indexing starts from the end of the string; *i.e.*, `"Hello?"[-1]` gives us `'?'` and `"Hello?"[-2]` the character `'o'`. Note that negative indexing starts with one (as the last element in the list) and in general, an index of `-i` refers to the character of the string at position `len(string) - i` in *positive indexing*.

Python provides additional tools to access a subset of a string using *ranged indexing*; *i.e.*, `[start:end:step]`, where `start` is the starting index, `end` is the ending index, and `step` specifies that every `step`th element of the string until the index `end` will be returned. For example:

```
>>> "Hello?"[0:4:2]
'Hl'
>>> "Hello?"[2:4]
'll'
>>> "Hello?"[2::2]
'lo'
>>> "Hello?"[::2]
'Hlo'
```

As seen in these examples, we can skip any of the `start`, `end` and `step` values in `[start:end:step]` (*i.e.*, we can just write `[start::step]`,

`[start::]`, `[::step]` *etc.*), and in these cases, Python will assume that the following default values are given: `start`: 0, `end`: the end of the string and `step`: 1. However, if a negative `step` value is specified, then the `start` corresponds to the last element, and `end` corresponds to the first element. In other words:

```
>>> "ABC"[::-1]
'CBA'
```

If the subfields of `[start:end:step]` are omitted, Python assumes default values for them (see the text for some examples).

In Python, strings, lists and tuples have similar representations and therefore, as we will see later, they have similar means for accessing their elements.

– Constructing Strings

We can get a string in either of the following ways:

- Enclosing a set of characters between quotation marks, like `"Hello?"`, in the code.
- Using the `str()` function as the constructor and supplying data of other types. For example, `str(4.5)` constructs a string representation of its floating point argument `4.5` and returns `'4.5'`.
- Using the `raw_input()` function as follows to get a string from the user:

```
>>> a = raw_input("--> ")
--> Do as I say
>>> a
'Do as I say'
>>> type(a)
<type 'str'>
```

– Useful Operations on Strings

- *The length of a string*: As we shown above, we can use the `len(string)` function to find the number of elements of a string: For example, `len("Hello?")` returns 6 since there are six characters in the string.
- *Searching for a substring in a string*: To do this, we can use the `string.index()` function, for example: `"Hello?".index("o?")` returns `4` since the substring `'o?'` starts at index four.

- *Counting the occurrence of a given substring in a string*: Can be achieved by using the `string.count()` function, for example: `"Hello?".count('lo')` returns `1` since there is only one occurrence of the substring `'lo'`.
- *Concatenating two strings*: Python provides the plus sign `+` to concatenate two strings: `"Hell" + "o"?` yields `"Hello?"`.
- *Making the string lowercase or uppercase*: We can use the `string.upper()` and the `string.lower()` functions respectively to make a string uppercase or lowercase. For example, for a string `"Hello?"`, the call to `"Hello?".upper()` result in `'HELLO?'` whereas the `"Hello?".lower()` would give us `'hello?'`.
- *Checking the characters in a string*: The function calls `string.isalpha()`, `string.isdigit()` and `string.isspace()` respectively check whether all the characters in a string are alphabetical, digital or whitespace (*i.e.*, space, newline and tab characters).
- *Splitting a string into substrings*: Often, we will need to split a string into two or more substrings based on a given separator. We can use the `string.split([string])` function for that (`[string]` denotes that the `string` argument is optional. Test it on your Python interpreter and see what happens when you don't supply a string argument to the `split()` function). For example, `"Hello?".split("e")` gives us a list of two substrings: `['H','llo?']`.
- *Checking for membership*: We can use the `in` and `not in` operators to check whether a substring occurs in a string: `substring in string`, `substring not in string`.
- *Minimum and maximum characters*: The functions `min(string)` and `max(string)` return the characters in the string that have respectively the minimum and the maximum ASCII values. For example, for the string `"Hello?"`, `min("Hello?")` and `max("Hello?")` respectively give us `'?'` and `'o'` since with ASCII values 63 and 111, `'?'` and `'o'` respectively have the minimum and the maximum ASCII values in the string.
- *Repeating a string*: We can multiply a string with a number, *i.e.*, `string*number`, and the result will be the concatenation of `number` many copies of the `string`. For example, `"Dan"*3` is `'DanDanDan'`.

2.4.2 Tuples

The term *tuple* is borrowed from *set theory*, a subfield of mathematics. A tuple is an ordered list of elements. Different from tuple in Mathematics, in Computer Science a tuple can have elements of heterogeneous types. A very similar wording will be

given in the following subsection for the definition of *lists*. Having almost the same definition for lists and tuples could be somewhat confusing but this can be resolved by understanding their difference in use and their function in life. Tuples are used to represent a *static form* of aggregation. There is a known prescription and you, the programmer, will bring together the components that make up that prescribed aggregate.

For example, in a computer simulation of a physical event, you will extensively be dealing with coordinates, *i.e.*, a 3-tuple of floating point numbers. You will send 'coordinates' to functions, and form variables of them. However, they will neither become a 2-tuple nor a 4-tuple; like the 3-musketeers, the grouping is unique. Similarly, assume you want to represent an 'employee record' which may consist of six fields of information:

ID, FIRSTNAME, LASTNAME, SALARY, ADDRESS, STARTDATE

As you can see, the types of the fields are not at all homogeneous. ID is an integer; FIRSTNAME, LASTNAME and the ADDRESS are three strings, whereas SALARY is a floating point, and STARTDATE is likely to be a 3-tuple. Since after setting the structure of the aggregate information once (in the example that is what makes up an employee record), it is illogical to change it in the subparts of the program, thus, you make it a tuple.

We will see that lists, too, serve the purpose of grouping information but they are dynamic; *i.e.*, the information in a list can be altered, new elements can be added and removed, *etc*.

As far as high-level language implementations are concerned, the immutability constraint is sometimes turned into an advantage when it comes to compilation. Compared to lists, tuples lead to fast and simple machine code. However, apart from this, the only use of tuples in favor of lists is to prevent programming errors.

Tuples in Python

Python provides the `tuple` data type for tuples. You can provide any number of elements of any type between parentheses `()` to create a tuple in Python:

```
>>> (1, 2, 3, 4, "a")
(1, 2, 3, 4,'a')
>>> type((1, 2, 3, 4, "a"))
<type 'tuple'>
```

– Accessing the Elements of a Tuple

Tuples have exactly the same indexing functionalities as strings to access the elements; *i.e.*, you can use the following that was introduced for strings:

- Positive Indexing: *e.g.*, `(1, 2, 3, 4, "a")[2]` returns `3`.
- Negative Indexing: *e.g.*, `(1, 2, 3, 4, "a")[-1]` yields `'a'`.
- Ranged Indexing, *i.e.*, `[start:end:step]`: *e.g.*, `(1, 2, 3, 4, "a")[0:4:2]` leads to `(1, 3)`.

Strings, Tuples and Lists in Python have the same indexing mechanisms.

– Constructing Tuples

There are several ways to constructing tuples:

- Writing a set of elements within parentheses: *i.e.*, `(1, 2, 3)`.
- Using the `tuple()` function as a constructor and supplying a string or a list as argument: `tuple("Hello?")` gives us the tuple `('H', 'e', 'l', 'l', 'o', '?')`, and supplying the `[1, 2, 3]` to `tuple()` function results in the `(1, 2, 3)` tuple (Note the difference between the brackets and the parentheses).
- Using the `input()` function as follows to get a tuple from the user:

```
>>> a = input("Give me a tuple:")
Give me a tuple:(1, 2, 3)
>>> a
(1, 2, 3)
>>> type(a)
<type 'tuple'>
```

– Useful Operations on Tuples

Like strings, tuples have the following operations:

- *The number of elements of a tuple*: We can use the `len(tuple)` function to find the number of elements in a tuple: For example, `len((1,2,3,4,'a'))` returns 5.
- *Searching for an element in a string*: One can use the `index()` function `location = my_tuple.index(element)` as follows, for example: `(1,2,3,4,'a').index(4)` returns `3`. Note, that unlike stings, you cannot use the `index()` function to find a subtuple within a tuple.
- *Counting the occurrence of a given element in a tuple*: One can use the `count()` function `location = my_tuple.count(element)` as follows, for example: `(1,2,3,3,'a').count(3)` returns `2`. Note,

that unlike strings, cannot use the `count()` function for counting the occurrence of a subtuple within a tuple.
- *Concatenating two tuples*: Python provides the plus sign `+` to concatenate two tuples: `(1, 2) + (3, 4)` yields `(1, 2, 3, 4)`.

2.4.3 Lists

Similar to tuples, lists store a sequence of ordered information. From the programmers point of view, the only difference between tuples and lists is that lists are mutable and tuples are not. You can extend a list by inserting new elements and shrink it by removing elements. However, this ability has deep implementational impacts. Once a tuple is created every member's position in the memory is fixed but this is not the case for a list. In lists, both the size and content can be dynamically adjusted. This brings the problem of structuring the data (the elements of the list) so that these changes to the size and content can be easily and efficiently performed. There exists various implementation alternatives for lists, which fall into two categories:

(a) Dynamic array solutions.
(b) Linked data structures solutions.

In type (a) implementations, reaching any member takes a constant number of steps (*i.e.*, reaching the first, the middle, the end of the list, or any other member in the list requires the same number of steps) which is better than type (b). However, in type (b), the insertion or deletion of elements takes a constant time which is not so for type (a). High-level languages are divided on this subject; for example, Lisp and Prolog implement lists as linked structures whereas the Python implementation is based on dynamic arrays.

The list as a container concept is quite often confused with the linked list data structure. Linked list data structures can serve as implementations of the list abstract data type (the container). As stated, there can be other types of implementations of lists.

Lists are heterogeneous, *i.e.*, the members of a list do not have to be of the same type. Mostly, lists themselves can be members of other lists, which is extremely powerful and useful for some problems. When you have this power at hand, all structures that are not self-referential[3] become representable by means of lists. Some languages, like Lisp, allow even self reference in lists. Therefore, complex structures like graphs become efficiently representable.

[3]Structures are self-referential if they are (recursively) defined in terms of themselves. We will come back to self-referential structures in Chap. 4.

Lists in Python

Tuples are immutable, *i.e.*, unchangeable, sets of data. For mutable sets of data, Python provides the `list` data type:

```
>>> [1, 2, 3, 4, "a"]
[1, 2, 3, 4,'a']
>>> type([1, 2, 3, 4, "a"])
<type 'list'>
```

Note that, unlike tuples, a `list` in Python uses brackets.

– Accessing the Elements of a List

Lists have exactly the same indexing functionalities as strings and tuplesto access the elements; *i.e.*, you can use the following that we already introduced for strings and tuples:

- Positive Indexing: `[1, 2, 3, 4, "a"][2]` returns `3`.
- Negative Indexing: `[1, 2, 3, 4, "a"][-1]` returns `'a'`.
- Ranged Indexing, *i.e.*, `[start:end:step]`: `[1, 2, 3, 4, "a"][0:4:2]` leads to `[1, 3]`.

– Constructing Lists

There are several ways of constructing Lists:

- Writing a set of elements within brackets: *i.e.*, `[1, 2, 3]`.
- Using the `list()` function as a constructor and supplying a string or a tuple as argument: `tuple("Hello?")` gives us the list `['H', 'e', 'l', 'l', 'o', '?']`.
- Using the `range()` function: `range([start,] stop[, step])`
- Using the `input()` function as given below, to get a list from the user:

```
>>> a = input("Give me a list:")
Give me a list:[1, 2, "a"]
>>> a
[1, 2, 'a']
>>> type(a)
<type 'list'>
```

– Modifying a List

You can substitute an element at a given index with a new one: *i.e.*, `List[index] = expression`. Alternatively, Python allows changing a subset of the list with new values:

```
>>> L = [3, 4, 5, 6, 7, '8', 9, '10']
>>> L[::2]
[3, 5, 7, 9]
>>> L[::2] = [4, 6, 8, 10]
>>> L[::2]
[4, 6, 8, 10]
>>> L[]
[4, 4, 6, 6, 8, '8', 10, '10']
```

Note that, in the above example, we have given the name `L` to the list in the first line. We call `L` *a variable*, and later in this chapter we will discuss in detail giving names to data (*i.e.*, variables).

Python provides the `L.append(item)` function to add one item to the end of the list `L`:

```
>>> L =  [4, 4, 6, 6, 8, '8', 10, '10']
>>> L.append("a")
>>> L
[4, 4, 6, 6, 8, '8', 10, '10', 'a']
```

Alternatively, you can add more than one item to a list using the `L.extend(seq)` function:

```
>>> L.extend(["a", "b"])
>>> L
[4, 4, 6, 6, 8, '8', 10, '10', 'a', 'a', 'b']
```

If you want to add a new element in the middle of a list, you can use the `L.insert(index, item)` function:

```
>>> L=[1, 2, 3]
>>> L
[1, 2, 3]
>>> L.insert(1, 0)
>>> L
[1, 0, 2, 3]
```

To remove elements from a list, you can use either of the following:

- `del` *statement*: `del L[start:end]`

  ```
  >>> L
  [1, 0, 2, 3]
  ```

```
>>> del L[1]
>>> L
[1, 2, 3]
```

- `L.remove()` *function*: `L.remove(value)`

```
>>> L
[2, 1, 3]
>>> L.remove(1)
>>> L
[2, 3]
```

- `L.pop()` *function*: `L.pop([index])`

```
>>> L=[1,2,3]
>>> L.pop()
3
>>> L
[1, 2]
>>> L.pop(0)
1
>>> L
[2]
```

List modification is meaningful only for named lists; *i.e.*, if you run (for example) the `append()` function like `[1, 2, 3].append(4)`, you will not observe an effect, although the list `[1, 2, 3]` might have been changed.

– Useful Operations on Lists

- *Reversing a list*: `L.reverse()` function
  A `list` can be reversed using the `reverse()` function.

```
>>> L = [1, 2, 3]
>>> L.reverse()
>>> L
[3, 2, 1]
>>> L[::-1]
[1, 2, 3]
>>> L
[3, 2, 1]
```

In this example, we also see the difference between the `reverse()` function and the `[::-1]` indexing (which gives the elements of a list in reverse order): `[::-1]` does not change the list whereas the `reverse()` function does (*i.e.*, it is *in-place*)!

- *Searching a list*: `index = L.index(item)` function
  The `index()` function returns the index of the `item` in the list `L`. If `item` is not a member of `L`, an error is returned.
- *Counting the occurrence of an element in a list*: `n = L.count(item)` function
  The `count()` function counts the number of occurrence of the `item` in the list `L`.
- *Range of a list*: `value = min(L)` and `value = max(L)` functions
  We can easily find the minimum- and maximum-valued items in a list using the `min()` and `max()` functions. For example, `min([-100, 20, 10, 30])` is `-100`.
- *Sorting a list*: `L.sort()` or `L2 = sorted(L)` functions
  There are two options for sorting the elements of a list: `L.sort()` or `L2 = sorted(L)`. They differ in that `sort()` modifies the list (*i.e.*, it is *in-place*) whereas `sorted()` does not.

2.5 Accessing Data or Containers by Names: Variables

We are mostly used to the word 'variable' from our Mathematics classes. Interestingly, the meaning in programming is quite different from the mathematical one. In Mathematics, 'variable' stands for a value which is unknown or undetermined. In programming though, it refers to a value that is at hand and hence, is very well-known. Literary speaking, a variable, in programming, is a storage which can hold a data element of the high-level language and in the program, is referred to by a symbolic name, given by the programmer.

2.5.1 Naming

The symbolic name is the handle that the programmer uses for reaching the variable. What this name can consist of differs from language to language. Mostly, it is an identifier which starts with a letter from the alphabet and continues either with a letter or a digit. Examples of possible names are `x, y, x1, xmax, xmin2max, a1a, temperature`. Whether there is a distinction between upper case and lower case letters, and the inclusion of some of the punctuation characters in the alphabet (*e.g.* '_', '-', '$', ':') is dependent on the language. Moreover, some languages limit the length of the identifiers whereas others do not.

2.5.2 Scope and Extent

The way variables are created differ among high-level languages. Some need a declaration of the variable before it is used whereas some do not. Some high-level languages allow the creation and annihilation of a variable for subparts of a program where some do this automatically. The subparts of a program can exist in various forms. Although not restricted to this, they are mostly in the form of subroutines, a part of the program which is designed to carry out a specific task and is, to a great extent, independent of other parts of the code. Subroutines are functional units that receive data (through variables that are named as parameters) process it and as a result, produce action(s) and/or data.

In addition to the parameters, many high level languages also allow the definition of local variables in subroutines. The situation can become quite complex when a high-level language (like Pascal) allows subroutine definitions inside subroutine definitions. This is done with an intention of limiting the subroutine's existence and usability to a locality.

All this creation and annihilation of variables, the nest-ability, both definition-wise and usage-wise, in subparts of a program result in the emergence of two important properties of variables:

- Scope
- Extent (or lifetime)

Scope is about where the language allows or disallows accessing a variable by referring to its name; *i.e.*, scope determines the program parts in which a variable is usable. Disallowing mostly occurs when a subpart (*e.g.* a subroutine) defines a variable with the exact same name of a variable that already exists in the global environment (the part of the program which is exterior to that subpart but is not another (sibling) subpart). The variable of the global environment is there, lives happily but is not visible from the subpart that has defined a variable with the same name.

Extent or so called *lifetime* is the time span, from the creation to the annihilation of the variable, during the flow of the program execution.

2.5.3 Typing

Depending on the high-level language, variables may or may not be defined with a restriction on the type of data they will hold. The difference stems from the language being;

- statically typed, or
- dynamically typed.

In the *statically-typed* case, the language processor (the compiler or interpreter) knows exactly for what kind of data the variable has been established. It is going to

hold a single type of data, and this is declared by a statement in the program (prior to using the variable). Therefore, whenever a reference in the program is made to that variable, the interpreter or compiler knows exactly what type of content is stored in the corresponding memory location. This is the case in C, Pascal and Java.

If it is a *dynamically-typed* language, the content of the variable can vary as far as types are concerned. For example, the same variable can first store an integer then, in accordance with the programmer's demand, it can store a floating point. In this case, every piece of data in the memory is tagged (preceded) with a byte that specifies the type of the data. Lisp, Perl and Python are such languages.

Statically-typed languages are relatively faster, since at run time (*i.e.*, when the program is running) there is no need to perform a check on the type of the data and make a decision about how to proceed with an action (*e.g.* to decide whether a multiplication instruction is that of a floating point or an integer type).[4]

2.5.4 What Can We Do with Variables?

The answer is simple: you can do whatever you wanted to do with the 'data' stored in the variable. You can replace the data, or use the data. Certain uses (syntaxes) of a variable (name) have the semantic (meaning) of "I want to replace the content of the variable with some new value (data)". All other uses of the variable (name) have the semantics of "I want to have the data (or sometimes a copy of it) inserted at this position". An assignment instruction example, common to many high-level imperative languages, would read:

```
average = (x + y) / 2.0
```

Here, `average`, `x` and `y` are three distinct variables. The use of `average` is syntactically different from the use of `x` and `y`. Being on the left of the assignment operator,[5] the equal sign, the data stored in the variable `average` is going to be replaced with a new value. This new value is the result of the calculation on the right side of the assignment operator and that calculation also makes use of two variables: `x` and `y`. Since they appear on the right-hand side of the assignment operator this time, the semantic is different, meaning *"use in the calculation the values stored in those two variables"*. In the course of the evaluation, those two values will be added then the sum will be divided by two.

[4] Processors have different sets of instructions for floating point and integer arithmetic.

[5] The assignment operator exists in all imperative languages. '=', '`:=`' are the most common notations used for assignment. '`<-`', '«', '`=:`', '`:`' are also used, but less frequently.

2.5.5 Variables in Python

As discussed above, programming languages almost always provide a means for naming the locations of the stored information, and these names are called *variables*. In Python, to create a variable is as easy as writing a name and assigning a value to that name:

```
>>> a = 4
>>> b = 3
>>> c = a + b
>>> a
4
>>> b
3
>>> c
7
```

One of the goodies of Python is visible in the previous example: we used the variables `a`, `b` and `c` without first defining them, or declaring what they are; Python can evaluate the right side of the equal sign and discover the type of the value. If the variable has already been created, the existing variable is used and the value stored in the variable is over-written. If the variable has not already been created, a new one is created with the type of the expression on the right-hand side of the assignment operator (*i.e.*, =).

The left-hand side of the assignment operator has to be a valid variable name; for example, you cannot type `a+2 = 4`, (we will come back to what can be put on the left hand side of an assignment operator in Chap. 3).

– Variable Naming in Python

Like in other programming languages, in Python only certain combinations of characters can be used as variable names:

- Variable names are case sensitive, therefore the names `a` and `A` are two different variables.
- Variable names can contain letters from the English alphabet, numbers and an underscore _.
- Variable names can only start with a letter or an underscore. So, `10a`, `$a`, and `var$` are all invalid whereas `_a` and `a_20`, for example, are valid names in Python.

Since the following keywords are already used by Python, you cannot use them as variable names:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

– Scope and Extent of a Variable in Python

The scope and the extent of a variable in Python depends on where it is defined. Since we will cover more complex Python constructs later in the book, we will return to the topic of scope and the extent of variables in the following chapters.

– Typing of Variables in Python

As discussed above, variables can be either statically or dynamically typed. In Python, variables are dynamically-typed. The following example displays this property:

```
>>> a = 3
>>> type(a)
<type 'int'>
>>> a = 3.4
>>> type(a)
<type 'float'>
```

That means, in Python, (i) you do not need to specify the type of the variable and (ii) you can change the type of the variable by assigning new data of a different type.

– Using Variables in Python

You can use variables in Python as you would use data: you can assign a tuple, a string or a list to a variable and manipulate it in any way that you would manipulate the data, for example:

```
>>> a = (1, 2, 3, 'a')
>>> type(a)
<type 'tuple'>
>>> a[1]
2
>>> a[-1]
'a'
```

– Variables, Values and Aliasing in Python

Each data (or object) in Python is assigned a unique identifier (basically, an integer) which can be accessed by the `id()` function. Having unique identifiers, Python manages memory space such that multiple occurrences of the same data are stored only once whenever possible. For example:

```
>>> a = 1
>>> b = 1
>>> id(1)
135720760
>>> id(a)
135720760
>>> id(b)
135720760
```

Here, we see that the data '1' and the variables `a` and `b` all hold the same content; *i.e.*, the data '1' is represented once and all these three cases make use of only one stored '1'.

However, it is safe to change the content of one variable without affecting the content of the other (following the previous interaction):

```
>>> a = 2
>>> b
1
>>> id(a)
135720748
>>> id(b)
135720760
```

In the following example for lists (which are mutable containers), however, the variables are *linked* to the same memory location and changing one variable means changing the other one, due to mutability:

```
>>> a = ['a', 'b']
>>> b = a
>>> id(a)
3083374316L
>>> id(b)
3083374316L
>>> b[0] = 0
>>> a
[0, 'b']
```

In this example, although we did not explicitly access the list pointed to by the variable `a`, we could change it since variable `b` became an *alias* to the variable `a` in the assignment `b = a`.

In Python, mutable data types (we have only seen lists up to now) are affected by aliasing. Although this can be a beneficial property at times, it is often dangerous to assign variables of mutable content to each other such as, `a = b`.

2.6 Keywords

The important concepts that we would like our readers to understand in this chapter are indicated by the following keywords:

Structured Data	Tuples
Basic Data	Variables
Integers	Static Typing
Floating Points	Dynamic Typing
Characters	Mutable Types
Boolean Values	Immutable Types
Strings	Aliasing
Lists	Scope&Extent of Variables

2.7 Further Reading

For more information on the topics discussed in this chapter, you can check out the sources below:

- *Two's Complement*:
  http://en.wikipedia.org/wiki/Two%27s_complement
- *IEEE 754 Floating Point Standard*:
  http://en.wikipedia.org/wiki/IEEE_754-2008
- *ASCII*:
  http://en.wikipedia.org/wiki/ASCII
- *UTF—UCS Transformation Format*:
  http://en.wikipedia.org/wiki/UTF-8
- *Mutable and Immutable Objects*:
  http://en.wikipedia.org/wiki/Mutable_object
- *For naming conventions and the reserved keywords in Python*:
  http://docs.python.org/reference/lexical_analysis.html#identifiers
- *Aliasing*:
  http://en.wikipedia.org/wiki/Aliasing_%28computing%29

2.8 Exercises

1. Find the Two's Complement representation of the following numbers: 4, −5, 1, −0, 11. How many bits do you require for all these numbers?
2. Find the IEEE 754 32-bit representation of the following floating points: 3.3, 3.37, 3.375.
3. You can use either a comma `,` or a plus sign `+` to combine different values or variables in a print statement. Experiment with the Python interpreter to see the difference.
4. Write a Python code that inputs two numbers from the user and then displays their arithmetic, geometric and harmonic means. Do you see any relation between the arithmetic mean, geometric mean and the harmonic mean? (Hint: For a set of numbers $x_0, \ldots, x_N$, arithmetic, geometric and harmonic means are respectively defined as: $1/N \sum_{x_i} x_i$, $1/N \prod_{x_i} x_i$ and $N/\sum_{x_i} 1/x_i$.)
5. Write a Python code that inputs the coefficients of two lines $y = a_1x + b_1$ and $y = a_2x + b_2$ from the user and finds their intersection. (1st Hint: You can assume that the lines are not parallel. 2nd Hint: The x coordinate of the intersection can be found by equaling $a_1x + b_1$ to $a_2x + b_2$. Putting that x coordinate in the equation of one of the lines leads us to the y coordinate of the intersection.)
6. Write a Python code that inputs the coordinates of the corners of a triangle (*i.e.*, (x_0, y_0), (x_1, y_1), (x_2, y_2)) and compute the area of the triangle.
7. What happens when you call the `append()` function on a list with another list as the argument? (For example: For a list `L = [1, 2, 3]`, what is the output of `L.append([4, 5])`?)
8. What happens when you call the `extend()` function on a list with another list as the argument? For example: For a list `L = [1, 2, 3]`, what is the output of `L.extend([4, 5])`?
9. Write a piece of Python code that takes the first half of a list, reverses it and appends it to the end of the second half. For example: For a list `L = [1, 2, 10, 20]`, the result of your Python code should be `[10, 20, 1, 2]`.
10. Write a piece of Python code that checks whether a word is palindrome. A palindrome is a word or phrase or number which reads the same forward and backwards. Examples are; *radar, racecar, wasitaratisaw*. (Hint: You can make use of the == operator to check the equality of two sequences.)
11. Write the following in Python and study their outputs: `help(help)`, `help(type)`, `help(int)`, `help(long)`, `help(float)`, `help(bool)`, `help(str)`, `help(list)`.
12. From the help page of the `list` type, find out how to calculate the number of bytes that a list occupies.
13. What happens when you try to index a string with a number greater than the length of the string? Do you get the same result for tuples and lists?
14. What happens when you try to change the elements of a tuple? For example: for a tuple `T = (1, 2, 3)`, what is the result of `T[0] = 3` ?
15. Do the `sort()` and the `sorted()` functions sort in increasing or decreasing order? How can you change the sorting order? (Hint: Use the help page of the `list` data type or the `sorted()` function.)

16. Type `import sys` and `help(sys)` into your Python interpreter and look for how you can access the following:
    (a) The maximum `int`, `float` and `long` that can be represented by your Python interpreter on your machine.
    (b) The version of the Python interpreter.
    (c) The name of the platform that the interpreter is running on.