# **Understanding data structures and algorithms**

Algorithms and data structures are the most fundamental concepts in computing. They are the building blocks from which complex software is built. Having an understanding of these foundation concepts is hugely important in software design and this involves the following three characteristics:

* How algorithms manipulate information contained within data structures
* How data is arranged in memory
* What the performance characteristics of particular data structures are

In this book, we will examine this topic from several perspectives. Firstly, we will look at the fundamentals of the Python programming language from the perspective of data structures and algorithms. Secondly, it is important that we have the correct mathematical tools. We need to understand some fundamental concepts of computer science and for this we need mathematics. By taking a heuristics approach, developing some guiding principles means that, in general, we do not need any more than high school mathematics to understand the principles of these key ideas.

Another important aspect is evaluation. Measuring an algorithms performance involves understanding how each increase in data size affects operations on that data. When we are working on large datasets or real-time applications, it is essential that our algorithms and structures are as efficient as they can be.

Finally, we need a sound experimental design strategy. Being able to conceptually translate a real-world problem into the algorithms and data structures of a programming language involves being able to understand the important elements of a problem and a methodology for mapping these elements to programming structures.

To give us some insight into algorithmic thinking, let's consider a real-world example. Imagine we are at an unfamiliar market and we are given the task of purchasing a list of items. We assume that the market is laid out randomly, and each vendor sells a random subset of items, some of which may be on our list. Our aim is to minimize the price we pay for each item as well as minimize the time spent at the market. One way to approach this is to write an algorithm like the following:

Repeat for each vendor:

1. Does the vendor have items on my list and is the cost less than a predicted cost for the item?
2. If yes, buy and remove from list; if no, move on to the next vendor.
3. If no more vendors, end.

This is a simple iterator, with a decision and an action. If we were to implement this, we would need data structures to define both the list of items we want to buy as well as the list of items of each vendor. We would need to determine the best way of matching items in each list and we need some sort of logic to decide whether to purchase or not.

There are several observations that we can make regarding this algorithm. Firstly, since the cost calculation is based on a prediction, we don't know what the real average cost is; if we underpredict the cost of an item, we come to the end of the market with items remaining on our list. Therefore, we need an efficient way to backtrack to the vendor with the lowest cost.

Also, we need to understand what happens to the time it takes to compare items on our shopping list with items sold by each vendor as the number of items on our shopping list, or the number of items sold by each vendor, increases. The order in which we search through items and the shape of the data structures can make a big difference to the time it takes to do a search. Clearly, we would like to arrange our list, as well as the order we visit each vendor, in such a way that we minimize search time.

Also, consider what happens when we change the buy condition to purchase at the cheapest price, not just the below average price. This changes the problem entirely. Instead of sequentially going from one vendor to the next, we need to traverse the market once and, with this knowledge, we can order our shopping list with regards to the vendors we want to visit.

Obviously, there are many more subtleties involved in translating a real-world problem into an abstract construct such as a programming language. For example, as we progress through the market, our knowledge of the cost of a product improves, so our predicted average price variable becomes more accurate until, by the last stall, our knowledge of the market is perfect. Assuming any kind of backtracking algorithm incurs a cost, we can see cause to review our entire strategy. Conditions such as high price variability, the size and shape of our data structures, and the cost of backtracking all determine the most appropriate solution.

# **Python for data**

Python has several built-in data structures, including lists, dictionaries, and sets, that we use to build customized objects. In addition, there are a number of internal libraries, such as collections and the math object, which allow us to create more advanced structures as well as perform calculations on those structures. Finally, there are the external libraries such as those found in the SciPy packages. These allow us to perform a range of advanced data tasks such as logistic and linear regression, visualization, and mathematical calculations such as operations on matrixes and vectors. External libraries can be very useful for an out-of-the-box solution. However, we must also be aware that there is often a performance penalty compared to building customized objects from the ground up. By learning how to code these objects ourselves, we can target them to specific tasks, making them more efficient. This is not to exclude the role of external libraries and we will look at this in [Chapter 12](https://jigsaw.vitalsource.com/books/9781786465337/epub/8162b260-0803-44a5-a4b8-b185816ca8ad.xhtml), Design Techniques and Strategies.

To begin, we will take an overview of some of the key language features that make Python such a great choice for data programming.

# **The Python environment**

A feature of the Python environment is its interactive console allowing you to both use Python as a desktop programmable calculator and also as an environment to write and test snippets of code. The read-evaluate-print loop of the console is a very convenient way to interact with a larger code base, such as to run functions and methods or to create instances of classes. This is one of the major advantages of Python over compiled languages such as C/C++ or Java, where the write-compile-test-recompile cycle can increase development time considerably compared to Python's read - evaluate - print loop. Being able to type in expressions and get an immediate response can greatly speed up data science tasks.

There are some excellent distributions of Python apart from the official CPython version. Two of the most popular are Anaconda (<https://www.continuum.io/downloads>) and Canopy (<https://www.enthought.com/products/canopy/>). Most distributions come with their own developer environments. Both Canopy and Anaconda include libraries for scientific, machine learning, and other data applications. Most distributions come with an editor.

There are also a number of implementations of the Python console, apart from the CPython version. Most notable amongst these is the Ipython/Jupyter platform that includes a web-based computational environment.

# **Variables and expressions**

To translate a real-world problem into one that can be solved by an algorithm, there are two interrelated tasks. Firstly, select the variables, and secondly, find the expressions that relate to these variables. Variables are labels attached to objects; they are not the object itself. They are not containers for objects either. A variable does not contain the object, rather it acts as a pointer or reference to an object. For example, consider the following code:



Here we have created a variable, a, which points to a list object. We create another variable, b, which points to this same list object. When we append an element to this list object, this change is reflected in both a and b.

Python is a dynamically typed language. Variable names can be bound to different values and types during program execution. Each value is of a type, a string, or integer for example; however, the name that points to this value does not have a specific type. This is different from many languages such as C and Java where a name represents a fixed size, type, and location in memory. This means when we initialize variables in Python, we do not need to declare a type. Also, variables, or more specifically the objects they point to, can change type depending on the values assigned to them, for example:



# **Variable scope**

It is important to understand the scoping rules of variables inside functions. Each time a function executes, a new local namespace is created. This represents a local environment that contains the names of the parameters and variables that are assigned by the function. To resolve a namespace when a function is called, the Python interpreter first searches the local namespace (that is, the function itself) and if no match is found, it searches the global namespace. This global namespace is the module in which the function was defined. If the name is still not found, it searches the built-in namespace. Finally, if this fails then the interpreter raises a NameError exception. Consider the following code:

a=10; b=20

def my\_function():

global a

a=11; b=21

my\_function()

print(a) #prints 11

print(b) #prints 20

Here is the output of the preceding code:



In the preceding code, we define two global variables. We need to tell the interpreter, using the keyword global, that inside the function, we are referring to a global variable. When we change this variable to 11, these changes are reflected in the global scope. However, the variable b we set to 21 is local to the function, and any changes made to it inside the function are not reflected in the global scope. When we run the function and print b, we see that it retains its global value.

**Flow control and iteration**

Python programs consist of a sequence of statements. The interpreter executes each statement in order until there are no more statements. This is true if both files run as the main program as well as files that are loaded via import. All statements, including variable assignment, function definitions, class definitions, and module imports, have equal status. There are no special statements that have higher priority than any other and every statement can be placed anywhere in a program. There are two main ways of controlling the flow of program execution, conditional statements and loops.

The if, else, and elif statements control the conditional execution of statements. The general format is a series of if and elif statements followed by a final else statement:

x='one'

if x==0:

print('False')

elif x==1:

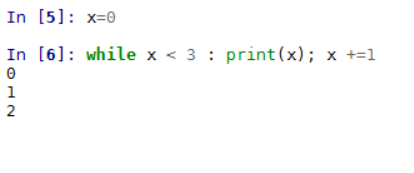
print('True')

else: print('Something else')

#prints 'Something else'

Note the use of the == operator to test for the same values. This returns true if the values are equal; it returns false otherwise. Note also that setting x to a string will return something else rather than generate a type error as may happen in languages that are not dynamically typed. Dynamically typed languages such as Python allow flexible assignment of objects with different types.

The other way of controlling program flow is with loops. They are created using the while or for statements, for example:



# **Overview of data types and objects**

Python contains 12 built-in data types. These include four numeric types (int, float, complex, bool), four sequence types (str, list, tuple, range), one mapping type (dict), and two set types. It is also possible to create user-defined objects such as functions or classes. We will look at the string and the list data types in this chapter and the remaining built-in types in the next chapter.

All data types in Python are objects. In fact, pretty much everything is an object in Python, including modules, classes, and functions, as well as literals such as strings and integers. Each object in Python has a type, a value, and an identity. When we write greet = "hello world" we are creating an instance of a string object with the value "hello world" and the identity of greet. The identity of an object acts as a pointer to the object's location in memory. The type of an object, also known as the object's class, describes the object's internal representation as well as the methods and operations it supports. Once an instance of an object is created, its identity and type cannot be changed.

We can get the identity of an object by using the built-in function id(). This returns an identifying integer and on most systems this refers to its memory location, although you should not rely on this in any of your code.

Also, there are a number of ways to compare objects, for example:

if a== b: #a and b have the same value

if a is b: # if a and b are the same object

if type(a) is type(b): # a and b are the same type

An important distinction needs to be made between mutable and immutable objects. Mutable object's such as lists can have their values changed. They have methods, such as insert() or append(), that change an objects value. Immutable objects, such as strings, cannot have their values changed, so when we run their methods, they simply return a value rather than change the value of an underlying object. We can, of course, use this value by assigning it to a variable or using it as an argument in a function.

# **Strings**

Strings are immutable sequence objects, with each character representing an element in the sequence. As with all objects, we use methods to perform operations. Strings, being immutable, do not change the instance; each method simply returns a value. This value can be stored as another variable or given as an argument to a function or method.

The following table is a list of some of the most commonly used string methods and their descriptions:

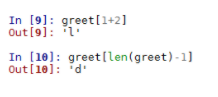
|  |  |
| --- | --- |
| Methods | Descriptions |
| s.count(substring, [start,end]) | Counts the occurrences of a substring with optional start and end parameters. |
| s.expandtabs([tabsize]) | Replaces tabs with spaces. |
| s.find(substring, [start, end]) | Returns the index of the first occurrence of a substring or returns -1 if the substring is not found. |
| s.isalnum() | Returns True if all characters are alphanumeric, returns False otherwise. |
| s.isalpha() | Returns True if all characters are alphabetic, returns False otherwise. |
| s.isdigit() | Returns True if all characters are digits, returns False otherwise. |
| s.join(t) | Joins the strings in sequence t. |
| s.lower() | Converts the string to all lowercase. |
| s.replace(old, new [maxreplace]) | Replaces old substring with new substring. |
| s.strip([characters]) | Removes whitespace or optional characters. |
| s.split([separator], [maxsplit]) | Splits a string separated by whitespace or an optional separator. Returns a list. |

Strings, like all sequence types, support indexing and slicing. We can retrieve any character from a string by using its index s[i]. We can retrieve a slice of a string by using s[i:j], where i and j are the start and end points of the slice. We can return an extended slice by using a stride, as in the following: s[i:j:stride]. The following code should make this clear:

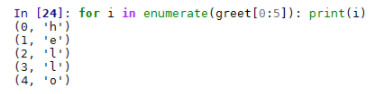


The first two examples are pretty straightforward, returning the character located at index 1 and the first seven characters of the string, respectively. Notice that indexing begins at 0. In the third example, we are using a stride of 2. This results in every second character being returned. In the final example, we omit the end index and the slice returns every second character in the entire string.

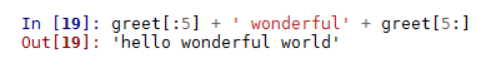
You can use any expression, variable, or operator as an index as long as the value is an integer, for example:



Another common operation is traversing through a string with a loop, for example:



Given that strings are immutable, a common question that arises is how we perform operations such inserting values. Rather than changing a string, we need to think of ways to build new string objects for the results we need. For example, if we wanted to insert a word into our greeting, we could assign a variable to the following:



As this code shows, we use the slice operator to split the string at index position 5 and use + to concatenate. Python never interprets the contents of a string as a number. If we need to perform mathematical operations on a string, we need to first convert them to a numeric type, for example:

