Reference Types and Memory models

Sergio Gutierrez-Santos and Keith L Mannock

2017

1 Introduction

Programs have lots of data, they are basically ways of getting data, transforming data, and giving data back to the world. Data types, as we have already seen, tell you which kinds of data you have in your program. Before we enter into a full discussion of so-called *reference types* let us have a short revision on numbers and the basic (primitive) types.

Computers store data in their memory in the form of bits. The word `"bit"` is a contraction of "`BI`nary digi`T`", so a bit is either `1` or `0`, `true` or `false`, `high` or `low`, `on` or `off`. Bits are organised in groups of eight called *bytes*. These bytes — groups of eight bits — are used to store any data in the memory of the computer. When you have got 1024 bytes, you have got a kilobyte or kB; when you have 1024 kB you have got a megabyte or MB, and so on for gigabytes (GB), terabytes (TB), and petabytes (PB). Note that in computing everything is measured in powers of 2 ($2^3 = 8, 2^{10} = 1024$) and not in powers of 10 as in normal life (10, 100, 1000...). That is because computers count with bits (2) and human beings count with their fingers (10)[1].

2 Simple data types

Simple data types can be thought as boxes in the computer's memory. Every time you declare a variable in your program, you can think of the computer as creating a little box in its memory to store your variable[2]. That box has two tags associated with it: one of them holds the name of the variable and the other holds the type (Figure 1)[3].

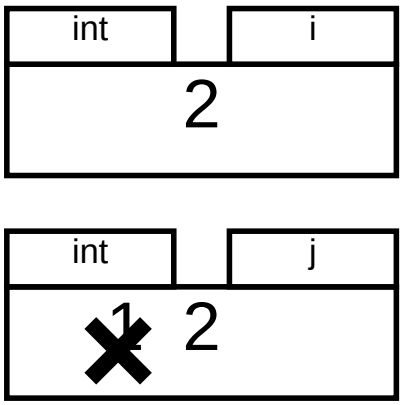
2.1 Static typing and dynamic typing

You have already seen that Python, as with most languages (including Java), puts some restrictions as to what you can use as a "name" tag for your boxes. You also know that some words cannot be used by the programmer for variables names because it would be confusing for the computer, e.g., `while`, `if`, `else`, etc.

[1] Not all humans used only the ten fingers in their hands to count. Some languages like French or Irish have remains of base-20 counting.

[2] This is only a metaphor and is not supposed to be an accurate description of how memory is managed on a modern computer. Explaining how things like the registers, the stack, and the heap work, the differences between actual machines and virtual machines, etc; are out of the scope of this document.

[3] This even applies to languages like Python where one does not declare the type of the variable.




```
i = 2;
j = 2;
```

Figure 1: Assigning a value to a variable can be seen as putting a value in the box. If there was something in the box, it is overwritten and lost forever.

Many programming languages also place one restriction on the "type" tag: once you decide the type of a variable, you cannot change it. This is like people having static opinions that they do not want to change. This type of languages are called a *statically typed language* and, contrary to people with fixed ideas, they are not necessarily bad or obnoxious. Java is an example of statically typed language.

Some programming languages allow you to change the type of your variables as you go along, so you can have a variable that sometimes is an integer and later in time is a boolean or a string. These languages are called *dynamically typed languages*. They have pros and cons compared to their statically typed counterparts.

There are many more things to know about *typing* in programming languages (*strong* vs. *weak*, *inferred* vs. *manifest*, *duck typing*, and so much more) but for now it will not be necessary to go into those details.

2.2 Most common simple types

2.2.1 Integer numbers

Integers are probably the most used simple data type, as integer numbers are used for two of the most common operations in computing: counting and indexing. Integers use 32 bits of memory, and an integer variable can hold values between -2,147,483,648 and 2,147,483,647 (inclusive). This data type is large enough for the numbers needed in 90% of programs most people write. We have already seen how to use it:

```
count = 1
```

There are two other kinds of integers for some special uses. When a program needs very large (positive or negative) values, there is a type for a *long integer*. It is called "long" because it uses 64 bits instead of 32, which means a long integer variable can hold values between $-9.22 \cdot 10^{18}$ and $9.22 \cdot 10^{18}$. There is also a "short" integer that uses only 16 bits of memory; this was sometimes useful to save memory when computers had limited storage but is hardly ever used with modern computers.

2.2.2 Floating-point (decimal/rational) numbers

Not all numbers are integer. Examples of common non-integer numbers include the result of a division of two integers where one is not a multiple of the other, and real-measurements like your height, your weight, and the distance between your work-place and your home (unless you work at home). In maths, these are called *real* numbers. In computing they are usually called *floating-point* numbers and are represented as a list of significant numbers and an exponent. The term "floating-point" refers to the fact that the decimal point can "float", that is, it can be placed anywhere in the number as long as the the exponent is changed accordingly.

$$1.23 \cdot 10^{-3} = 123 \cdot 10^{-5} = 0.00123 \cdot 10^{0}$$

We cannot use super-index notation in a plain-text file, so we need a special way of writing these numbers. Those three ways of depicting the number above can be written as `1.23E-3`, `123E-5`, and `0.00123`, and the three are equivalent.

In many programming languages real numbers are usually represented with the simple data type that uses 64 bits. There is also a 32-bit version but, as with a short integer, it is seldom used today.

Important note. Floating-point numbers do *not* have infinite precision, and operating with them can cause rounding errors. There is a special type for representing real values where precision is paramount (like in banking); more of which later.

Equality. Due to rounding errors, it does not make sense to test for equality among real numbers as we can do with integers. Two numbers could be notionally the same but be different due to rounding errors, so they are never compared with equality with == (or shouldn't be). Instead, what is usually done is test whether the difference is less that some precision limit appropriate for the application (e.g. 1.0E-6), as in the example below. Note: `abs()` returns the absolute value of the number inside the brackets, e.g. `abs(-3)` returns 3.

```
d1 = (0.1+0.1)/0.3
d2 = 2.0* 1000.0/9000.0*3.0
# WRONG!
if (d1 == d2):
        # this is not printed due due to rounding errors
        print("They are the same (wrong comparison)")

# RIGHT!
if (abs(d1 - d2) < 10E-6):
        print("They are the same (right comparison)")
```

2.2.3 Boolean (binary) values

This simple data type represents one bit of information. It can hold the values `true` and `false`.

2.2.4 Characters

Text is composed of characters: 'a', 'b', 'c'... The `char` simple type is used to represent characters. It uses 16 bits, meaning it can represent any of 65,536 different characters.

Actually, the 16 bits of a char represent a Unicode symbol. Unicode is a computing industry standard for the consistent encoding, representation and handling of text expressed in most of the world's writing systems. It includes symbols from most writing systems in the world, including alphabets like Latin, Cyrillic, Arabic, or Hebrew; syllabaries like Japanese katakana and hiragana, or Cherokee; and many more.

You may have noticed that we have not mentioned strings yet. This is because strings are a complex, or reference type.

3 Complex or reference types

Complex types are types of data that do not fit in a box, not even in one of the big 64-bit boxes used for double precision numbers. As they do not fit in the "boxes", computers have to store them somewhere else. However, they also need to know where they are... and that is what the boxes are used for.

Modern computers have *a lot* of memory. Long forgotten are the days when Bill Gates said:

> "640kB of memory should be enough for everything"

Part of a computer's memory is used for the boxes (in a part of memory called the *stack*) and most of the rest is used for everything else, including complex data (that part is called, quite unceremoniously, the *heap*).

When your Python or Java code uses some complex data, the computer stores that data in some region of the heap — identified by a *memory address*; then it stores the address in a box in the stack, much in the way it stores integers and booleans. This looks similar to Figure 2. The memory address in the box can be seen as "pointing to" the place in memory where the real data is stored. For this reason, we will call it a *pointer*.

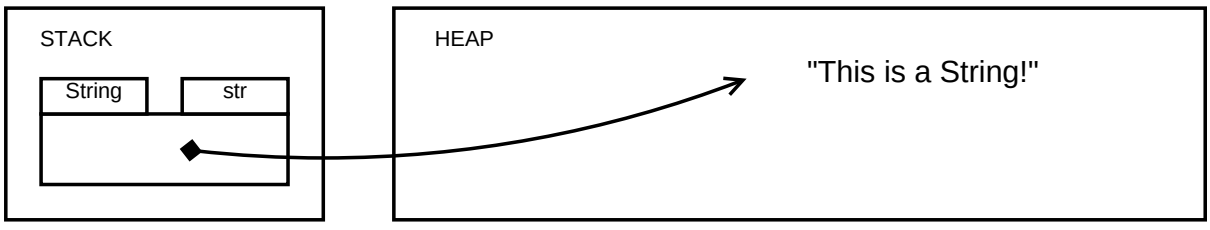


Figure 2: A String is a type of complex data type. The data itself is stored in the heap and its address is stored in box in the stack —pointing to it.

Using complex data types is different from using simple types in several ways. For starters, as complex types are not stored in the stack but in the heap, you have to *allocate* memory-space in the heap to store the data. In some languages this will require the programmer to use a specific keyword, e.g., `new`, in others the space will *auto-magically* be reserved. We are now going to examine several examples, starting with strings.

3.1 Declaration and initialisation

When working with simple types, the memory is always used as soon as you declare the variable. In other words, your program will use the same amount of memory if you type `int i` and if you type `int i = 1`: it always uses 32 bits of your computer's memory.

This is not true with complex types where the computer only reserves the box for the pointer. In languages like Python though, which require initialisation with a value to determine the type, the pointer and the value are allocated. So,

```
str = "A string"
```

notionally reserves some space for the string AND space for a box with the pointer to the string in it.

Please note that the difference between simple and complex/reference types is huge. Boxes are 32-bit or 64-bits long, but there is no limit to the size of a complex type: it could be several kB or even MB (types so big are a rarity,though).

Sometimes the pointer will not point to any address in memory. This can be useful in some cases that will become clearer as we learn more about programming, including error detection and the release of memory that is no longer needed (remember that complex types can use a lot of memory). In some languages, e.g., Java, to make a pointer point to nowhere, we use the reserved word `null` (with non-capital letters). In Python we don't have this *feature*, well, not readily.

A pointer pointing to null is called a *null pointer*. This basically means that the address in the box is zero (rather than an obscure hexadecimal number like `0x1a3ec74`); the computer knows there is never anything at address zero, so it knows a null pointer is not being used to point to any real data (Figure 3).

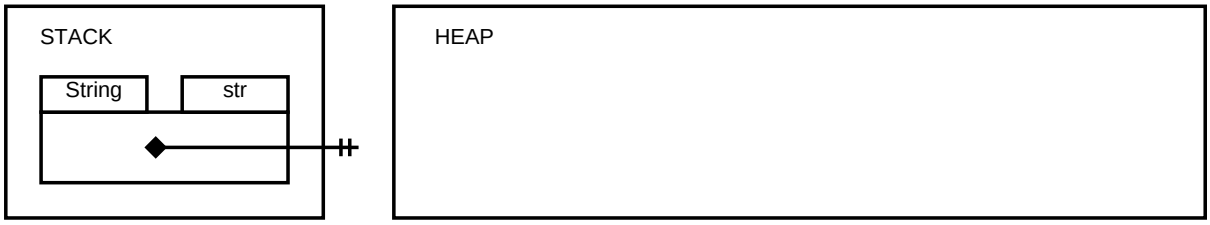


Figure 3: A null pointer is basically a zero address, pointing nowhere.

As a null pointer does not point to any real data, if you try to access a variable that is pointing to null, the computer will complain with an exception.

3.2 String

Strings are everywhere. Every program, except the most trivial, uses strings: user names, passwords, addresses, configuration options, data input from the keyboard, a webpage read through the Wi-Fi connection... almost anything is a string. It is the most widely used complex type.

At their most basic, strings are sequences of characters. You already know how to read a piece of text from the user:

```
name = input("What is your name? ")
```

If you want to create a string in your program without reading it from the user, the basic form of creating a string is as follows:

```
str = "This is a String"
```

but you already know that! It is important to note that a string with only one character is still a string.

You already know that you can do some things with strings using various *functions* and *operators*, e.g., slicing and indexing. Therefore we won't go into any further detail here. Please refer to the notes on string if you need to remind yourself as to how to manipulate them.

3.3 You own structures: classes

Although string is the most common complex type, it is not the only one. As a matter of fact, programmers can create their own complex types very easily. We are jumping a bit ahead of ourselves here but the question does arise:

> "How do I create my own data types?"

To create new types of complex data, we use the classes (more of which in a later session and section of the notes). You can think of it as creating new classes of data.

A new class of data must have a name and be defined. Usually a complex type is composed of several other types, simple of complex. The same rules for representation still apply though; simple values are stored in boxes, and complex types are *linked* using pointers (memory references).

3.4 A final note on terminology

If you read other books or web pages about Python or Java, you may notice that they use terms that are different from the ones we have used in this section. We go through some of them here for the sake of clarity.

Simple type: A type that is stored in its own box, usually 32-bit or 64-bit long. These are usually called *primitive* types in the Java world. I prefer to call them simple data types to distinguish them clearly from complex data types.

Complex type: A type that has two parts: the box contains a pointer, and it points to a place in memory where the actual value is stored. In the Java world, these are simply called *classes* and *objects*, which is precise but fails to be explicit on the difference between simple and complex data types, and this can be a source of confusion.

Pointer: The content of the box in a complex type, a memory address where actual data is stored. This is sometimes called a *reference* or a *handle* to prevent confusion with pointers of other languages (like C) that behave in a slightly different way. I think many programming languages have constructs that share the same name and behave in slightly different ways (Strings is a major example) so this is not of much concern. Besides, it looks to me like very confusing to access a null "reference" or a null "handle" and get a Null**Pointer**Exception as a consequence.