5 Python Concepts That Are Misunderstood by Beginners

Python is easy — but it can still be tricky when you are starting out

Fabio Veronese

Photo by Siora Photography on Unsplash

Python is great for mid-level and experienced programmers, and also ideal for computer engineers. But if you have just learned to program or perhaps have no academic background there may be a few tricky things with no intuitive explanation.

TL;DR: Python is a high-level interpreted language. Many lower-level gauges and knobs are hidden — by design. Not knowing what happens under the hood will lead to misunderstanding and to learning the wrong lessons.

1. Assignment

The assignment is when a variable is given a value. There are several aspects to be taken into account when analyzing assignment. Being a variable a space in memory whose content represents its value, in reality, the assignment may be one of several operations.

In the case of a literal, what happens is that the value on the right of the equal sign is given to the variable on the left.

In other cases, there are great differences between Python and other programming languages. The assignment between two variables of a base type represents a copy of the value. Thus the values of strings, ints, floats, booleans, and few other types are copied from right-hand variables to left-hand ones.

However, the other types are objects, which are considered more complex and detailed. So, instead of copying their structure going deep in their levels recursively, Python always uses reference assignments. It means that the space in memory remains the same, but from the assignment on the two variables are stored in the

same place — just as if they were aliases of the same instance in memory. This has some funny consequences if you have no idea of what happens in memory.

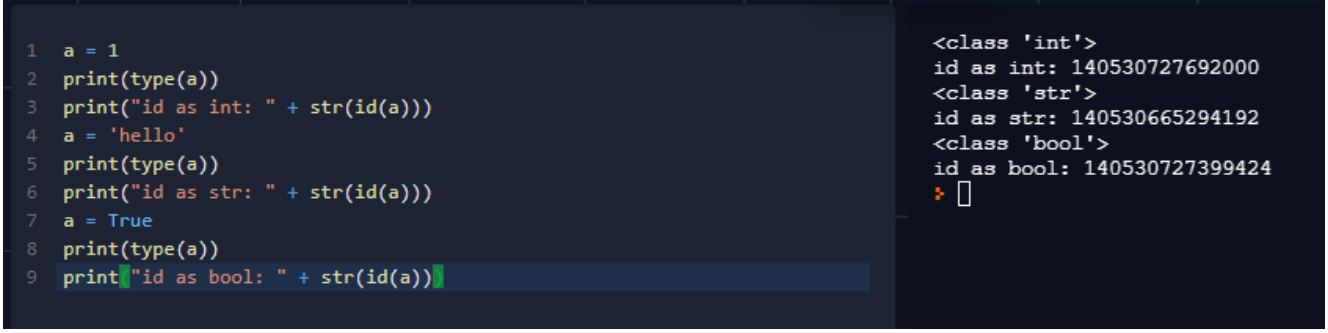
High-level programming languages like Java have the same policy, while C has explicit management of memory, pointers (i.e. locations in memory), etc. It is far more complex, but it represents a true image of computer reality.

2. Declaration and Instantiation

The declaration is a statement requiring the existence of a variable. This usually is associated with instantiation, i.e. the creation of space in memory to hold the variable value. Moreover, this is commonly done with a declaration of the type of the variable (numeric, string, etc.).

In Python, there is no need to explicitly declare a variable. It has declaration and instantiation at assignment, implicitly, without declared type.

One of the consequences is that you have no evidence when trying to re-declare a variable. It seems the variable changes type and value, but actually it is a completely different variable .



The function `id` gives you evidence of the raw identifier of the memory space assigned to the variable (sort of). So as you can read the three declarations imply three different variables.

Continuing with the side effects of this policy, using a variable with the same name as a function argument is allowed. But then, are you sure you are not messing with the original variables used as arguments?

Finally, no declared type may result in variable misuse (wrong datum or algorithm error).

3. Typing

Typing refers to the space to hold data in memory and the characteristics of the variable derive from its type. At instantiation, the program requests the allocation of a memory block to hold the variable, so that information is then used.

Python uses duck-typing. This means that a variable has a specific type, inferred at the assignment, but when using a given variable, no specific type is requested. The important thing to note is that the operation performed is

legal. If it quacks like a duck and it walks like a duck, then it is probably a duck — and so Python treats it as such.

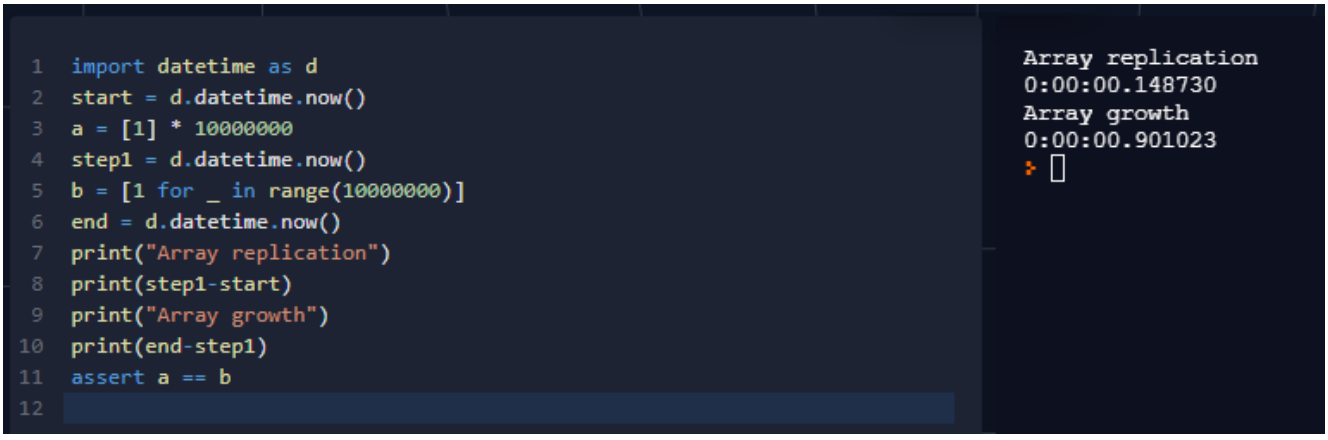
Coupled with declaration at assignment, duck typing has the funny consequence that reassigning/redeclaring a variable apparently changes its type. No way. That's not what happens, in reality, another space in memory with the same alias is reserved for the new value to be stored. So it is a brand new variable, just with the same name.

Moreover, you may mess with pieces of your own code, feeding it the wrong type variable, just because it *quacks*.

4. Memory Management

As mentioned in the assignment, Python has automatic policies for memory management. They include implicit memory allocation, manipulation, and release.

As often happens with automatic and implicit things, this can lead to funky consequences. To make a brief (and silly) example, if you allocate an int array in different ways you get different timings.



So `a` takes 0.14s to be generated while `b` 0.90s (almost 10 times more). This difference (apparently not clear) is due to memory management: in the first case, the whole memory is reserved at once. In the second, all the results of the call to `range` are stored in memory, and (looping on them) the memory for `b` is allocated piece by piece: this implies a double overhead.

Moreover, the garbage collector frees unused variables when the scope is completed. So it may happen that some useless variables (maybe very big ones) still stand in the memory and you experience an out-of-memory issue when instantiating an apparently harmless small variable.

5. Looping, Iterators, and Generators

Without entering in the details of the`yeld` command, Python has only two types of loops: `while` and `for ... in`. Still, they generate great confusion when compared with the other programming languages. Compared with C, `while` has the same behavior, it checks a Boolean condition and exits as soon as it is not matched. However, `for` is radically different: in C the code block is repeated based on an initial condition, executing an increment each step, as long as a boolean condition is met; in Python it executes the code block on a sequence of elements, exiting when they are over.

This discrepancy has consequences greater than you may think. First, clearly, `for` iterates objects, so even if submitted a list it may include repeating entries and all sort of things. Moreover, there is no need of an actual list as long as you are given a function, whose invocation returns a valid “index” and raises an exception (the one used when the list elements are out). That’s a generator, and its usage has plenty of features and issues completely different and much more complex than C-style for looping.

Wrapping Up

As a former teaching assistant for C programming lab for engineering B.Sc. I’ve seen many errors by inexperienced programmers. Difficulties with pointers, loops, forks, memory management, and scopes are very frequent. So, if you’re learning, and you go for Python beware of its peculiarities, and use a proper course as a guide, you may avoid the effort of understanding low-level issues. But it’s very likely you will pay the price later, when you become more experienced and do more complex things.