# http://www.experts-exchange.com/articles/5354/Python-illustrated-part-1.html

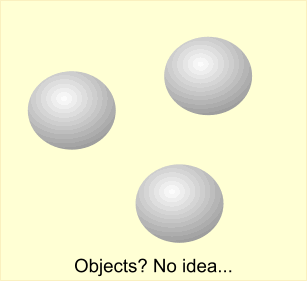
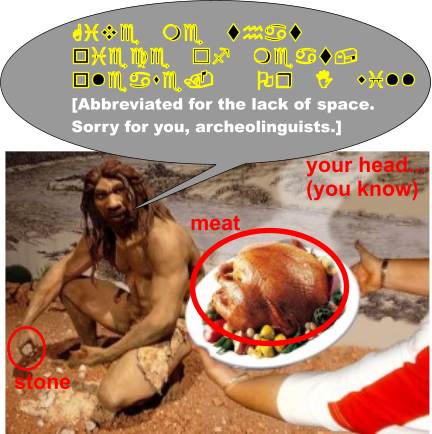
# Python illustrated (part 1)

[http://cdn.experts-exchange.com/files/3b14e350-1b90-4c69-a50a-698defc33aa9.png](http://www.experts-exchange.com/members/pepr.html)Approved Article by [pepr](http://www.experts-exchange.com/members/pepr.html) On 2011-05-07 Views: 2,903 9,104 Points

**The really strange introduction**

Once upon a time there were individuals who intentionally put the grass seeds to the soil with anticipation of solving their nutrition problems. Or they maybe only played with seeds and noticed what happened... Some years later, people chew pizza made of flour, enjoying their holiday in Italy, thinking of nothing -- definitely not about the grass seeds that were put to the soil. Still, the local bakers strive for perfection in baking *The Best Pizza in The World*, choosing carefully the kind of flour.  
  
Another time, another places, there were individuals that played with first computers. They knew *everything* about their computer -- till the last bit. (Frankly, it was *a bit easier* as there was not so much bits inside as they are now.) Often, they built the computer. Or at least they fell in love with it and knew how to build it (also in cases when they bought it), and they replayed in their imagination "how they are rebuilding the same", feeling themselves strong, self-confident in the area. They knew "much more" about "much less" in comparison with our present time.  
  
"Keep the ballance." The greatly admired people are often extreme people. They often brought us something very valuable, but they were not understood in their time, and they often *died poor* or in physical or in mental sense. In other words, being odd makes it sometimes easier to be great in something. The question is whether you want that. On the other hand, focusing on being intentionally dumb in the chosen area to keep the statistic ballance inside the society and expecting to be automatically successful in the other areas... Well, it simply does not work.  
  
"Why the hell is he writing such obvious things?" This is called a professional blindness. When focused on our professional problems, we often behave strangely. Programming is no exception. Experts often think about the users as about dumbs because the users cannot do *such obvious things*. And the pizza chewers complain like "Why the application is so complicated and still it does not do that one simple thing that I want?" And there are also people, who simply want to try programming to have some fun.  
  
The very basic things should never be underestimated or postponed for learning later. It is no accident to call such things *Basic Building Blocks* of programming. You can put something that works when interconnecting several ready-to-be-used, bigger building blocks. However, it will not make you universal. This will not help you to approach any general problem that could be solved.  
  
"One picture is worth of thousands word." Well, it depends. The truth is that "images" and "imagination" clearly have some common base. Good imaginagion is one of the key abilities of a successful programmer. In my opinion, this is the direction in which the beginners should be trained. Too many teachers are not aware of that. They may think that it is obvious and the students should already know that. The truth is that not all students are of that kind to naturally feel the same. Also, I may be wrong. Anyway, I am going to focuse on helping your imagination with images when explaining the basic building blocks of the Python programming language. The following serie (with at least one part ;) will show if the idea is good or not -- just react after reading the article.  
  
"Stop writing when it is good enough. It will never be perfect."  This is one of the golden rules.  Another one says: "Never say never."  A kind of paradox?  Maybe.  I will try to put both together.  Based on your reactions to the specific parts, I will try to simplify or explain better the basics to that level (from different angles of view) that you will find nothing else to ask.  (Just kidding.  Everything can be improved.) If you like mathematical or otherwise formal approach to programming, you may not like the article.  I will try to avoid mathematics and formalism -- not because I do not like it, but because beginners usually do not like it, and because it does not help to train in imagination if you have a poor one (no offense meant, no shame).

**Objects**

[](http://filedb.experts-exchange.com/incoming/2011/05_w19/456402/objectsNoIdea.png)The reality does not need objects.  It is our perception that tries to recognize the object.  It is our brain that is not capable to capture the reality as collection of all recognizable and possibly interrelated details.  Our brain needs simplification when thinking about complex things.  The caveman's brain also needed that.  What is *tangible* is more understandable.  "Give me that piece of meat, please.  Or I will hit your skull with that stone."  
  
[](http://filedb.experts-exchange.com/incoming/2011/05_w19/456401/caveman.jpg)[My sorry for using and modifying the copyrighted image of the caveman diet (<http://www.stayfitbug.com/wp-content/uploads/2010/09/caveman-diet-300x200.jpg>) from the stayfitbug.com article "The Caveman (No supplements) vs The 21st Century Man (Takes supplements)" <http://www.stayfitbug.com/the-fitness-bug/the-caveman-no-supplements-vs-the-21st-century-man-takes-supplements/>. No, I did not ask them for permission (you know, the time pressure, etc.).  But yes, I will remove it if asked. On the other hand, isn't this mutually beneficial?]  
  
Well, they are too complex and too distracting objects on the picture -- not good from a pedagogical point of view.  And you are capable of abstract thinking, right?  
  
  
Let's start with *variables* that are a kind of simplest version of objects in any programming language.

**Variables**

## Even if you do not like the mathematical approach to programming, you should be aware of the fact that the name *variables* comes from mathematics.  You know them from school as *letters* that are used to replace *any* number.  You have learned how to search for their values if they were part of what we call equations.  And you have learned how to manipulate symbolically the formulas so that you could get simpler formulas.  To summarize, the variables in mathematics are used as symbolic replacements for the possible values. [Mathematical formalism and variables.](http://filedb.experts-exchange.com/incoming/2011/05_w19/456403/varM.png)Now the computer-related point of view.  Any value in computers need some memory space to be stored.  One have to reserve certain amount of bytes -- the "bigger" value, the more bytes.  So, from the hardware point of view, the variable is the memory space of some size placed somewhere.  We have to know where the "where" is, and we (as humankind) decided to number the memory positions, and we name the numbers as the addresses.  The hardware does not need any "names" for variables.  The size and the placement in memory is all the processor needs to know when working with the content. [Hardware-oriented view to variables.](http://filedb.experts-exchange.com/incoming/2011/05_w19/456404/varC.png)Humans are more error prone when working with numbers (unless the number means the sallary, or the pocket money, or the number of beers, or similar counting cases that are so obvious to know them right).  When programming, we combine the mathematical thinking with the computer-related point of view.  We think in terms of memory space, but we do not like to use the numeric address when thinking about the placement of that space.  It would be too much details, and our brain would defend the situation using the "forget it" mechanism.  The name for the space (instead of the numeric address) is much more acceptable.  Our lazy brain likes abstractions.  It does not like details, it likes working with fuzzy pictures generated by our imagination.  The brain likes to search for the principles of the solution of the problem.  When programming, we try to automatize the steps leading to the solution.  Or we need to repeat the same steps many times in future (for a different input to get the wanted output), or we need to utilize the raw speed of the computer hardware and the ability to remember, to organize and to work with many boring details. [Mind picture of variables.](http://filedb.experts-exchange.com/incoming/2011/05_w19/456414/varA.png)Programming languages are here to put the humans (who design the way to get the solution) and the computers together.  A programming language is used for transformation of our abstract mental pictures of the solution to the form of a formal description of steps leading to the solution (for a computer).  Having that function, our favourite programming language must be capable to describe the abstractions that were born in our brain.  But also the reverse direction works: our favourite programming language uses constructions and data abstractions that form the abstract thinking.  We gradually tend to use the programming-language abstractions as basic blocks for our imagination.  In other words, we learn *how to program in the programming language*.  This is one of the main reasons why we have to deeply understand the basic blocks of the programming language.  We have to make the related abstract basic block for our thinking. Back to variables. Let's agree that our mind picture and the programming language abstraction of a "variable" have the common features: 1) they can be represented by the name, 2) they are capable to store the value. The programming languages introduced several ways how to put the name of the variable (called the *identifier* of the variable) with the related memory space.  We use the variable names in our "program sources" which are basically text files that are later compiled somehow to something more acceptable by the computer hardware (to be executed as steps towards the solution). Topics for the planned part 2

More about variables and about memory allocation:  
- statically compiled languages  
- statically allocated memory (compiled time)  
- dynamically allocated memory (run time)  
- dynamic languages  
- directly bound (variable) name to the statically allocated memory  
- indirectly bound name of a variable in compiled languages  
- what is a pointer (explicit dereferencing)  
- what is a reference  
- type of a variable  
- types, variables, and compiled languages  
- types, variables, and scripting languages  
- Python approach to types and variables (globals() -- dict of name->reference variables)

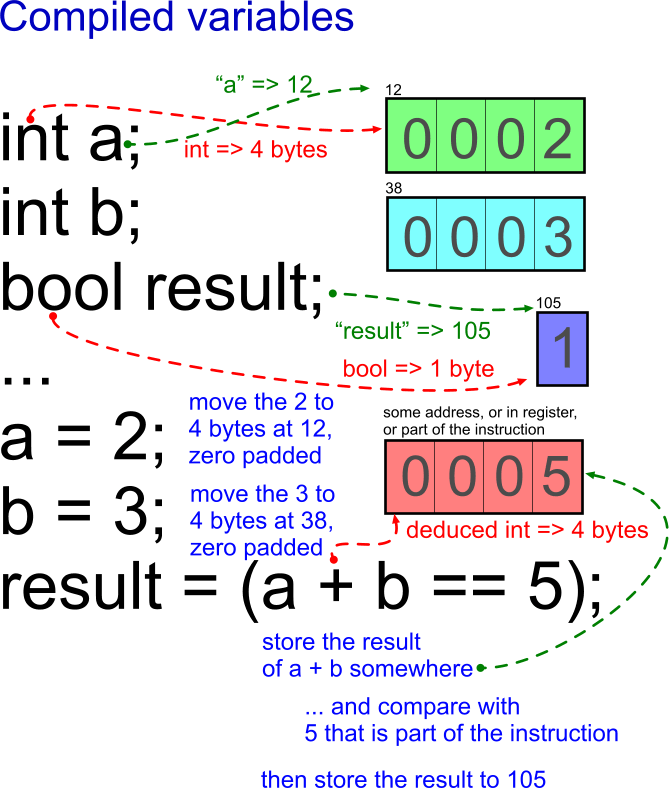
# Python illustrated (part 2)

[http://cdn.experts-exchange.com/files/3b14e350-1b90-4c69-a50a-698defc33aa9.png](http://www.experts-exchange.com/members/pepr.html)Awarded Article by [pepr](http://www.experts-exchange.com/members/pepr.html) On 2011-08-29 Views: 3,167 12,867 Points

## Less strange, but still introduction

This introduction was added (1st August, 2011) to reflect some reactions.  Firstly, the term *basics* in the title of the article...  As any other word, it is a symbol with meaning attached to the word by some agreement.  Still, we always have to think in some context to understand the agreement.  You may say that what is described in this and previous articles is not related to *Python basics* at all.  It could be because you expect a text written in some context that you met when reading some other "basics".  The article series could be renamed to *Python foundation* or *Python internals*.  However, the later terms are *more technical*, and I do not think it would help you to understand what I want to write about.  Instead, I am asking you to change the context of thinking about it.  Tutorials often start with the simplest example that shows a working program (say "Hello, World!" -- it is also a kind of unspoken agreement bound to the "creation of a tutorial").  This is understandable.  There are many ways of how to attract a beginner's attention.  The context of "basics" in this series is based on "what you should know to understand".  I am focusing on "mental pictures of what is done" rather than on "how to write the block of code".  For that, I need also the text of this second part.  I am aware of the situation that you may want to skip these two articles with disagreement. ("Not related to Python at all!")  In the same time I believe, that the information will be useful for those who find some knowledge gaps when reading the next part 3.  
  
The previous part 1 ([http:A\_5354-Python-basics-illustrated-part-1.html](http://http:A_5354-Python-basics-illustrated-part-1.html)) explained where the term "variable" came from, how we can think about it, what it represents, how the variables are related to mathematics on one side, and to computers on the other side.  This part (2) is focused on more details that you have to understand if you really want to be good in thinking about variables (creating a mental picture) and about using them in programming languages.  
  
For explaining principles of working with variables, we need some excursion outside the Python point of view.  This part describes variables in context of traditional compiled languages to show some details.  Let's talk about pointers and references.  Good understanding will be necessary to swallow the part 3.

## Variables in the (old) compiled languages

Let's think about the equation *a + b = 5* as about the Boolean expression that returns--for the given content of variables *a* and *b*--the Boolean value that indicates whether the values are the solution for the equation.  
  
[](http://filedb.experts-exchange.com/incoming/2011/07_w29/478313/compiledVariables.png)

This is a fragment of a source code in the C language.  The "C" is not important here.  Let it simply be the example of a classical procedural compiled language.  In such case, the source text is converted to the machine code via a compiler and a linker. The *names* of the variables are totally replaced by addresses, by register names/numbers, or the values are optimized to be the parts of the generated machine instructions.  There is no trace of strings like "a", "b", or "result" in the generated executable.  The only exception is when you generate so called *debug* version of the executable and you explicitly tell the compiler to remember the original names from the source code; however, the computer does not need that.  Only the human, the programmer wants the names be present in the debug info to make the compiled code more readable when debugging.  
  
From that point of view, *a name of a variable to a human programmer* is in the similar relation as *an address of a memory space to a processor*. The name of the variable is *directly bound to the address*.  Or YOU work with the name as with the string (in the source code) or the COMPILER works with the address that is the result of the compilation of the name of the variable (in the binary, executable code).

## Technical representation of a variable

Technically, the variable needs the memory space and the identification to be useful for a program.  The identification without the memory space makes no sense -- one cannot store anything inside.  The memory space without the identification is useless either -- one cannot access the memory space.  
  
We were speaking about *variables* in the previous part.  However, the same holds for objects, i.e. for the memory space that is used to store the data of the object.  Better to say, anything in Python is an object that can be identified, including the things that are called variables in some languages.  But stay tuned, the details will be explained later.  The topic is not that easy as you may think at first. (There are more situations like that. Say, the *strings* -- the topic "well known from the time of written history". Or not? See <http://diveintopython3.org/strings.html>).  
  
The memory address serves as a good technical identifier.  It is unambiguous. Moreover, you need no transformation to get the information where the memory is located. The major Python implementations also take that approach -- the address is equal to the technical identification of any variable (or of any object).  Having any object **obj** in Python, you can apply the built-in function **id(obj)** to get the identification.  (However, other Python implementations are free to use a different kind of identification in future. Think about a distributed computing environment where there is no single, shared memory address space. The address is ambiguous in such systems unless it is extended by some extra information about the location of the memory/computer.)  
  
What about the size of the reserved memory space?  When you need to store a Boolean value, one bit would be enough.  As you cannot address one single bit, the smallest possible piece of memory, one byte, is often used.  If you want to store an integer, you usually think about 4 or more bytes.  It depends on the programming language, on the compiler, and also on the hardware. (In Python, integer variables are not that limited.  The space for one integer value may vary depending on the actual value.)    
  
To summarize, the memory space depends on the type of the value and sometimes also on the actual value (think about a string of a different length in whatever language].  In compiled languages, the size of trivial types is known.  Therefore, the size of memory is related to the type.  However, the information about the type is used/processed only during the compilation.  Similarly to variable names, the type is used only to check statically if the things are done in a correct way.  During the compilation, the type-name information is lost, and the related size of memory is present as numbers in the executable.

## Where the memory space is located?

Think about a simple situation.  You have a normal computer with one processor and with one RAM (Random Access Memory).  The RAM address goes (for simplicity) from zero to 4.000.000.000.  The variable needs say eight bytes.  Where the eight bytes are to be located?  
  
"I don't care."  And you are right.  The machine and the compiler should care.  Anyway, the code needs to know the location.  Where the knowledge about the location is stored?  
  
The *compiled-language sources* are processed by the compilers that consume the source texts and convert them to the machine code.  In such case, the knowledge about the location must be hidden somewhere in the code.  Otherwise the code would not be able to access that portion of memory (i.e. what once was the name of the variable).  In the case, *the address* of the memory must be hidden in the code.  But how?  
  
Roughly said (i do not want to go into too much details):

The address can be a constant, and the related memory was once (in the source text) called *a static variable*. Such a variable keeps the content during the lifetime of the program and its content changes only when it is explicitly changed by the code.

The address is computed relatively -- by adding the numeric offset to some base address (in some register) of the memory subspace.  (This is done as the part of the low-level instruction behaviour -- no human-related programming like "take the register content, add 5, and assign...)  Think about a function that uses its own, local variables.  When the function is called, it gets a block of memory big enough for storing its local variables.  The local variable is located relatively to the beginning of the allocated block.  The block is released when the function returns.

The memory is allocated dynamically, during the running time via calling some function (like *malloc()* in C) or via some other action bound to the dedicated keyword (like *new*).  In such case, the address is known only after the data space was created (when the program is already running), and the address must be stored somewhere for the later reference.  No name is bound to such memory space -- even in compiled languages.  The size was deduced from the prescribed type (*new*) or it was given explicitly to the function (*malloc()*).  In the first case, the compiler keeps track about the size derived from the type.  In the later case (*malloc()*), a programmer is responsible for working correctly within the allocated space.

## Scripting languages

There was a time when so called *scripting languages* appeared (think about the Unix shell scripts or the Windows batch files).  A command processor (say *bash* or *cmd* takes the source text called *a script* line by line and interprets the commands written in the file.  These languages are also called *interpreted*.  This is because of when and how the source text is processed when the script is launched.  There are no binary native instructions stored in the executed script file.  The source text is read and interpreted immediately.  Of course, the script has to be interpreted by some binary executable program--*the interpreter*--that is capable to do the actions prescribed in the script.  (This is done via association with the script extension or via special command at the beginning of the script.) Because of the way how it works, the interpreted languages are not so fast, not so powerful in comparison with compiled languages, and also their data types are somehow more limited.  Often, the part of a name of a variable indicates also the type.  
  
Simply said, the work with variables is a bit magical in the scripting languages.  You never need to know their memory address.  As you usually write only simple scripts, you usually do not want to build more complex data structures.  You do not care how the memory is allocated.  (Anyway, the memory must be allocated dynamically.)  
  
The interpreted languages may often be *weakly typed*.  Simply said, if the string value looks like a number, it can be treated as a number.  Because of that, you can use such value with numeric operators, for example.  
  
The simpler languages often play *tit for tat*.  And we sometimes need something in between the simple scripting languages and the extremely powerful (fast running and expressive) compiled languages.  This can be done, and Python is the example of such language.  Before speaking about details of (also called) *dynamic languages*, we need to learn something more about indirect access to memory space (i.e. to the stored values).  It is a natural feature also in compiled languages; however, it is essential for dynamic languages.

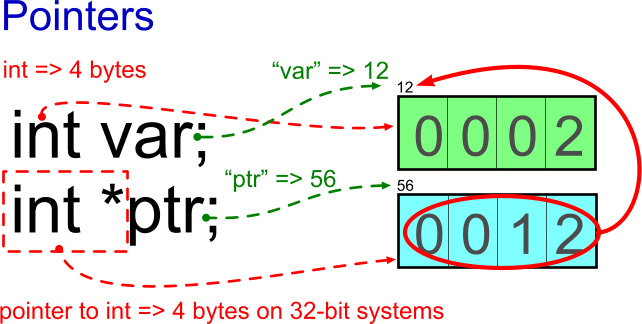
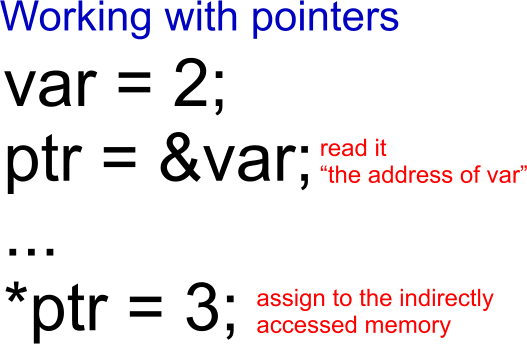
## Pointers

Oh the bloody pointers!  It is the theme where many programmers with less formal education and/or with not enough greed for knowledge fail.  When looking at, say, C++ source code without the knowledge, many students just give up and switch off their brain.  Possibly they have never got the satisfactory explanation.  Let's enhance our imagination using the following pictures.  Let's start the hard way -- from the magical C++ source code to the abstract pictures. (No problem if you do not know the C++ language.  I will explain the necessary things.)  
  
Let's start with:

int var;

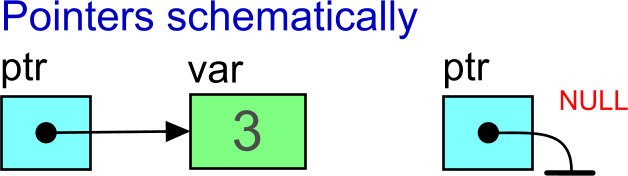
int \*ptr;

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-6589-1)

The first line declares that there will be the variable named *var* of the type *int* (means *integer*).  The second line looks almost the same.  There will be variable *ptr* somehow related to the type *int*.  But what means the star?  The star changes the meaning of the declaration of the variable.  The variable will not be of the *int*.  It will be of the *pointer to int* type, instead.  Have a look at the following picture:  
  
[](http://filedb.experts-exchange.com/incoming/2011/07_w29/478316/pointers.png)The picture shows that both *var* and *ptr* variables need their memory space.  In other words, *pointer variable* is also *a variable*.  A pointer variable also stores a value.  The value could be called a pointer value.  Basically, it is the address of another memory.  As explained earlier, any variable needs memory space and the identification.  Any pointer variable needs that much space to be able to store the address.  If the hardware is capable to address directly say 4 GB (i.e. 2^32 bytes), then the variable needs 32 bits (4 bytes) of memory to store the address.  However, if we use the 64-bit Operating System, then we say we can (theoretically) directly address 2^64 bytes of memory.  I do not believe you have that much memory in your computer.  Anyway, the pointer variables need 8 bytes in such case.  
  
Let's assign the values to the variables to see how to work with pointers:  
  
[](http://filedb.experts-exchange.com/incoming/2011/07_w29/478317/workingWithPointers.png)

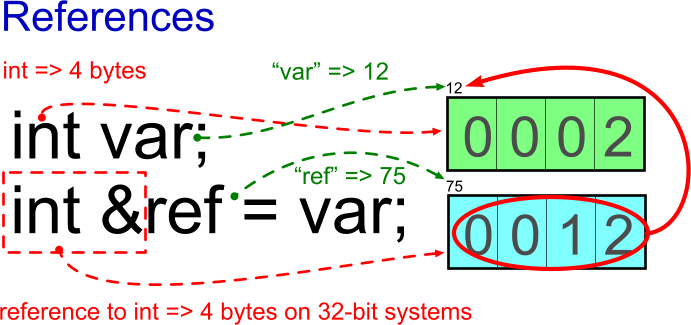
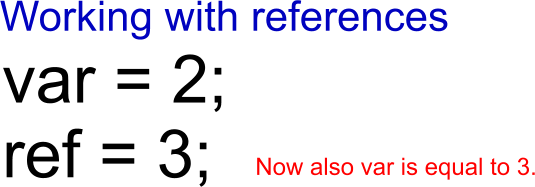
You should already be very familiar with the fist assignment of the 2 to the *var* variable.  The earlier picture shows that the four bytes at the address 12 are filled with the binary representation of the value 2.  Notice, that the variable was given the memory space at the address 12.  The notation *&var* means *the address of the variable*.  Now, the *ptr* variable is ready to store the address.  This way, the address 12 can be assigned to the pointer variable (see the second line on the picture).  
  
Having the pointer value (i.e. the address 12) inside the pointer variable that is located on address 56 (see the picture above), we can access the memory space of the *var* variable also indirectly, through the pointer.  To do that, we have to read the content of the variable *ptr* located at the address 56, and then we have to use its content (12) to locate the memory at that address.  That's all!  No extra magic.  Well, the syntax may have look magically...  
  
The *\*ptr* (with the star at in front of the name of the variable says: take the content of the ptr and use it for indirect access to another part of memory.  The process is named *dereferencing*.  The pointer variable *refers* to some other memory, and we want to access it.  As we have to explicitly use the star in front of the pointer variable, it is called *explicit dereferencing*.  
  
Having the *access to the part of memory* means also the *write access*.  The last line on the picture shows that the memory space is assigned by new value 3.  If you now read the variable *var*, you would get the value 3.  The memory space belongs to the *var* variable, but from now on it is not the only way how it can be accessed.  
  
Does the pointer theme look so difficult to you now?  Is there anything more to be said?  Thinking about the principle, then nothing.  But, well, yes.  
  
When talking about variables in compiled languages, we have said that the compiler converts the name to the address, deduces the size from the type of the variable, and uses the type for static checking during the compilation.  Is there anything about types at the last picture?  Well, yes.  
  
The *\*ptr = 3;* assigns the integer value.  The integer value can be assigned only to an integer variable.  How the compiler knows that this assignment is correct?  The answer is not that difficult.  It is more difficult to spot that the check must be done by the compiler.  
  
The *ptr* was declared as *pointer to int*.  This way, it says that it point to the memory of the size that is capable to store the int value.  In other words, pointers in compiled languages are usually bound with some type.  And again, the type is used only for checking during compilation (hence the name *statically typed languages*).  The information about the type disappears when generating the binary executable.  Anyway, the pointers are *typed* in the compiled languages.  
  
What about untyped pointers?  Are they possible?  Is it possible to have a pointer to any type?  The short answer is yes.  The pointer variable itself requires always the same amount of memory (4 bytes on 32-bit OS, 8 bytes on 64-bit OS).  It simply stores the address.  The problem is that then the compiler does not know how big is the block of memory that is pointed to.  The pointer variable stores no knowledge about the size of the target.  The programmer have to get the information from somewhere else.  One of the possibilities is to store the size of the pointed structure at the beginning of the structure.  
  
What else?  NULL.  Or the similar name.  This is the special value that can be assigned a pointer variable of any type.  This means that you can assign it also, say, to the pointer to int.  This is the only pointer-value constant available.  It says: "the pointer points to nowhere."  Naturally, you cannot de-reference such pointer.  The "nowhere" can never be accessed.  If you try, the screaming sound... kidding, not screaming actually... the error is somehow announced.

## Drawing variables and pointers symbolically

Our brain likes thinking in abstractions.  "No boring numbers, please.  I tend to forget them."  We always solve (sub)problem in our head first, and only then we write the idea down.  We use our imagination.  If it is too much to keep it in our brain, we draw it on a paper...  
  
Also, we need some graphical abstraction *to be able to share* what we mean with others.  We want to keep the picture as abstract as possible, as simple as possible, but not simpler than needed.  
  
[](http://filedb.experts-exchange.com/incoming/2011/07_w29/478318/pointersSchematically.png)

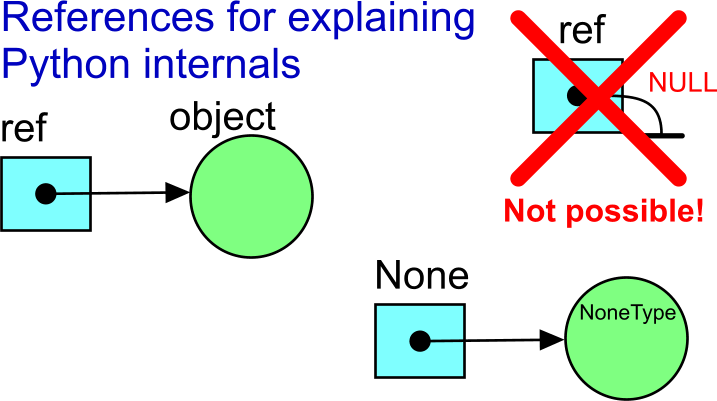
The memory space for the variable is usually drawn as rectangle.  Again, a pointer variable is also a variable -- it needs its rectangle too.  However, we want the pointer value be more abstract.  (Recall what the brain have said about the numbers.  Anyway, we do not know the exact addresses and we do not care.)  The only important thing for us is that the value points to another memory space (hence *the pointer*).  And the pointer should be pointed, right? ;)  To summarize, the pointer value is drawn as a fat dot with an arrow pointed to the other memory-space rectangle.    
  
We also do not care how the NULL is implemented.  The only interesting feature is that it points to nowhere.  The "electric ground" mark is usually used.  That mark was very usual and well known to those who worked with first computers -- not counting Charles Babbage (<http://en.wikipedia.org/wiki/Babbage>) and "World's First Computer Programmer" Ada Lovelace (<http://en.wikipedia.org/wiki/Ada_Lovelace>).  
  
Now, repeat "the pointers" until you really know what they are about.

## References

The term *references* often causes confusion.  The main reason is that it can be used in different contexts.    
  
The more abstract point of view is based on the idea that the reference value allows you to access some data (memory space) indirectly.  From that abstract point of view, the plain old pointers are also references.  
  
However, the terms references and pointers are often discussed together, and there are some differences emphasized between them in the case.  Usually, the references are said to be de-referenced automatically when used (unlike the pointers).  
  
The confusion has roots also in the fact that different programming languages think differently about references.  Some languages do not use the term references, some languages do not know pointers.  Some languages do not have any special syntax for references, and the fact of working with references internally may be completely hidden (this is also the case of Python -- see later).  
  
Back to the old compiled languages, here C++.  
  
[](http://filedb.experts-exchange.com/incoming/2011/07_w29/478319/references.png)The image shows the C++ syntax.  Ignore the syntactic details if you do not know the language.  What is important, the *ref* variable must be initialized when declared.  The reason is that it must always contain a reference to some already existing variable, i.e. to its allocated memory space.  
  
Here the *ref* variable was located by the compiler/linker/loader at the address 75 (not important).  The important is that its memory space was filled with the address of the *var* variable immediately.  The C++ restriction is that you cannot change the reference variable later.  Any usage of the *ref* has the same effect as using the variable *var*.    
  
[](http://filedb.experts-exchange.com/incoming/2011/07_w29/478320/workingWithReferences.png)

The main difference from pointers is how the references look in the source code.  Once the reference variable is set, you can use it *as if it was another name for the pointed variable*.  In the example, the reference variable is named *ref* which forces you to think this way about it.  But imagine if it was named *myVar*.  When using a reference variable, it looks the same as if you worked with a normal, simple variable.  It is because the *automatic de-reference* is done (by compiler, for you).  Anyway, the target memory space is still accessed indirectly.  
  
In other words, when you do not see the reference-variable declaration, you are not able to say if you work with a simple variable or (indirectly) with another variable.

## References for Python

Python uses references internally a lot.  Let's define the term *reference* the way we need for explanation of the Python internals.  
  
[](http://filedb.experts-exchange.com/incoming/2011/07_w29/478322/referencesSchematically.png)We usually do not draw the references on the paper the same way as we do with pointer-based data structures.  Anyway, we can draw them the same way when explaining the internals.  Actually, the references are used in Python the very same way as pointers are used in other languages.  However, it is hidden from the programmer.  The reference value in Python (as elsewhere) needs always the same memory space to be stored (4 bytes on 32-bit systems, 8 bytes on 64-bit system).    
  
However, Python is *a dynamic language* where memory space for the objects is always allocated in run-time, and where the name of any variable is kept intentionally in a string form inside Python's internal data structures.  
  
Why to call them *references* and not *pointers*?  This is because you will find no explicit de-referencing when working with Python.  But that's not all!  There is one more indirection level when working with variables.  The topic will be discussed more in part 3.  
  
A side note: There is nothing like NULL value for Python references.  This would break the concept of references.  However, there is the single object named *None* that plays the same role.  Whenever another object refers to *None*, it means that there is nothing more interesting there (if the fact is not interesting on its own).  
  
  
**Topics to be discussed in** [part 3](http://www.experts-exchange.com/A_7109.html)  
  
- Python as a dynamic language  
- everything in Python is an object  
- trivial Python built-in types  
- container built-in types  
- what is behind the assignment operation in Python  
- containers and references  
- Python approach variables (dict of name->reference)  
- where the types are stored in Python

# Python illustrated (part 3)

[http://cdn.experts-exchange.com/files/3b14e350-1b90-4c69-a50a-698defc33aa9.png](http://www.experts-exchange.com/members/pepr.html)Approved Article by [pepr](http://www.experts-exchange.com/members/pepr.html) On 2011-08-29 Views: 3,502 11,302 Points

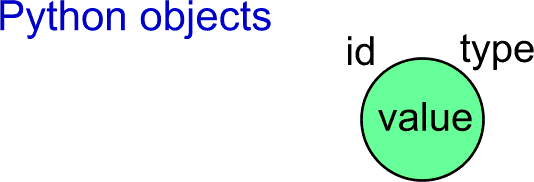
"The time has come," the Walrus said, "To talk of many things:  
Of sets--and lists--and dictionaries--  
Of variable kinks--  
And why you see it changing not--  
And why so strange are strings."

This part describes how variables and references (see parts 1 and 2) fit with Python, and how the *clearly strange* behaviour is not that strange when you know *only a little bit more* about the Python internals.  Some people may say: "Hey, they are not Python basics at all!"  The reality is that not having that little tiny bit of Python knowledge you will once stand schocked, with open mouth, staring at the behaviour of your program. Especially when you ARE a programmer (used to a compiled language).  
  
If you find anything less understandable about variables and references in this article, have a look at [Python illustrated (part 2)](http://www.experts-exchange.com/Programming/Languages/Scripting/Python/A_6589-Python-basics-illustrated-part-2.html).  Also, your feedback is warmly welcome.  It will form the problems illustrated in part 4.

## Summary for Python variables/objects

You may not like to read a lot of "theory" first, and you may want to search where the conclusion is.  "Does it make sense for me to read further?"  Good news for you -- the summary from Python point of view comes first:  
  
1. Everything in Python is an object. Any object has its unique *technical* identification (the memory address).  The information about its type is kept inside the object.  
  
2. The simplest objects are few constants, numeric values, and strings.  They are also stored as objects which means they have their own identification.  Once they are created, they cannot change their content.  They are *immutable*.  
  
3. More-complex built-in objects are lists, tuples, dictionaries, and sets.  They contain only *references* to other objects, not the objects itself.  Let's call them *containers* (as it is usual also in other languages).  The references are untyped.  The type is bound to the target object.  
  
4. Any "native" assignment in Python means assigning the reference value.  It refers the target object that represents the assigned value. In other words, there will be one more counted reference to the target object after the assignment (counted by the target object).  
  
5. If any object is named, the name is stored as a key in one of the internal Python dictionary structures. The reference to the value object is the value of the dictionary item.  
  
  
Now slowly, with pictures.

## Everything in Python is an object

Objects are not *the new thing* these days any more.  Objects as instances of their classes can be found everywhere.  Is there anything special about them in Python?  
  
Every object in Python has its unique identification, and it is of some type (i.e. instance of some class).  The object can exist without any name.  The only two requirements for its existence are: 1) it must be created, and 2) it must be accessible via at least one reference. (Nothing very special.)  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489370/01PythonObjectsIdType.png)

## Simplest Python objects

The very basic objects in Python are boolean values (True, False), numeric values (int, float, complex), strings (plus some more).  
  
The *most natural simple things in any programming language* are integer values and integer variables.  It is so natural that you can hardly imagine how the integer value could be expressed better than by simply typing the textual representation of the number in the source file.  What does it mean when we put it together with "everything in Python is an object"?  
  
Run the Python interpreter in the interactive mode (simply type **python** without arguments in you console window) and try the following:

c:\tmp\\_\_\_python\\_\_articles\03pythonVariables>python

Python 2.7.1 (r271:86832, Nov 27 2010, 17:19:03) [MSC v.1500 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> id(1)

30628552L

>>> a = 1

>>> id(a)

30628552L

>>> b = 3 - 2

>>> b

1

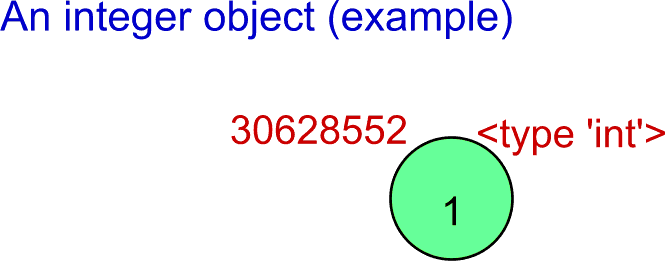
>>> id(b)

30628552L

>>> id(5 - 4)

30628552L

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-1)

[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489371/02IntegerObjectExample.png)Notice that simply typing **1** in the source code leads to the creation of the object that represents the value.  Looking better at the lines above, the *id()* function returns always the same identification independently on how you have got the value.  Then it must mean that all the cases share the same object of the integer type with the value 1.  This is a kind of optimization that does not affect the solved problem.  Actually, only some small integers have "a fixed identification".  If you try it with bigger integer values, separate objects are created. Try (or read and believe) the following:

>>> a = 1

>>> b = 1

>>> id(a)

30628552L

>>> id(b)

30628552L

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-2)

Here the identifications are the same.  But try it with bigger numbers:

>>> a = 1500000

>>> b = 1500000

>>> id(a)

36576600L

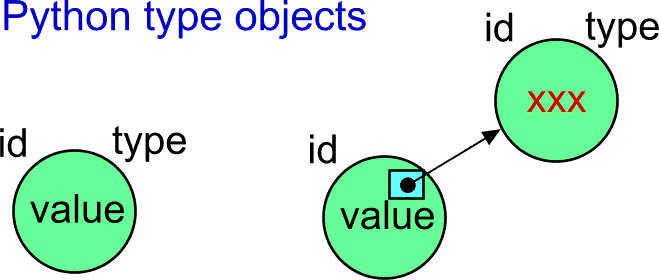
>>> id(b)

36576576L

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-3)

The identifications now differ.  The internal optimization was not used for the case.  
  
If you are interested in more details, have a look for example at Laurent Luce's Blog, "Python integer objects implementation", <http://www.laurentluce.com/posts/python-integer-objects-implementation/> (I have just found his pages, no deep opinion, but they seems to be very good).

## A note about the type

Actually, the instance of the object (its memory footprint) should be as small as possible. This means that the object itself should not store all details common for all instances of the same class.  In other words, it should NOT store also details about its type.  Technically, it is better to share a reference to another block of information -- also expressed as an object in Python, like this...  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489373/03SeparatedTypeObject.png)

There is another built-in function named *type()* in Python.  The documentation says that it that returns the reference to the related *type-object*.  When displaying the object, we can see the string like...

>>> type(1)

<type 'int'>

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-4)

The text speaks for itself.  But, hey, "... returns the reference to the related *type-object*"?  Is the type object really separated?  If yes, it must be possible to get its identification and type. Let's apply the earlier *id()* function to the type object:

>>> id(type(1))

505997744L

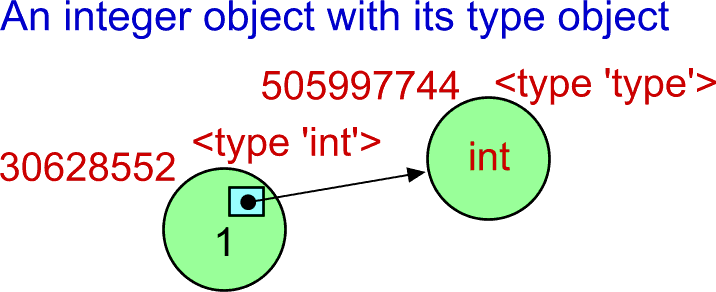
[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-5)

Well, the type objects seems to be separated. (It has its own identification.) Then, what is the type of that type object?

>>> type(type(1))

<type 'type'>

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-6)

[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489374/04IntWithTypeObject.png)

Are you confused?  If yes, then don't worry.  You will get used to.  If it is too difficult for you now, think in terms *the type is bound to the object* that represents the value.

## Some other simple Python objects/types

They are boolean constants, float numbers, complex numbers, and strings.  There is also a special type and the constant named *None*:

>>> id(True)

505930304L

>>> id(False)

505930280L

>>> type(False)

<type 'bool'>

>>> type(1.1)

<type 'float'>

>>> 1+5j

(1+5j)

>>> type(1+5j)

<type 'complex'>

>>> type('some string')

<type 'str'>

>>> None

>>> type(None)

<type 'NoneType'>

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-7)

## Values of objects of the simple types can never be changed!

When working with objects in various programming languages, it is usual to think about the object as about a data space bound to some functionality (the methods).  It is very usual to think about them as about "capsules" that can change their internal state, i.e. change the content of their internal data variables.  This is not the case of the simple-type Python objects.  
  
Any Python object of a simple type cannot change its value!  Once the object was created, it stores its initial value until the object is destroyed.  In other words, the object shown in the upper examples will always behave as constants. **The objects of the simple types represent the captured value.  They never (logically) act as a memory space that could be reused for different values.**  In other words, having reference value to some simple object, you can be sure that you always get the same value.  
  
This fact may be surprising for many programmer, and it brings a whole lot of questions.  The true beginners just do not care.  Actually, is it that unnatural?  When working with the symbol **1** (one) in mathematics, you would never expect that the same symbol changed its internal value to 5.  
  
There are reasons for the "strange decisions" of the authors of Python.  The reasons may not be immediately apparent.  One of the reasons is that the simple-type objects are to be shared heavily, because assigning the value means actually assigning the reference to the object.  It will be more apparent later.  
  
There are also consequences of the constant nature of the objects of the simple types.  The consequences could be quite disturbing at first.  For example, ignoring the fact may lead to the very inefficient string processing in the sense of time/space complexity of operations.  

## Built-in containers

The more-complex built-in types of objects are lists, tuples, dictionaries, and sets.  They are designed to *contain* other objects.  Because of that we call them *containers* (as it is usual also in other programming languages).  
  
A list allows you to store and update a sequence of other objects (the order is preserved).  Tuple-type objecs are similar to lists, but they cannot be modified after creation. A dictionary stores pairs *key, value* where the key is used to access the value part (i.e. associative access; also known as a *map* or a *hash table*).  A set allows you to capture a set of objects. Sets can be tested, modified and otherwise manipulated using operations that are known from the set theory (mathematics).  
  
It is extremely important for understanding how Python works, that all container objects always contain only the *references* to the objects representing the values.

## Tuples and lists

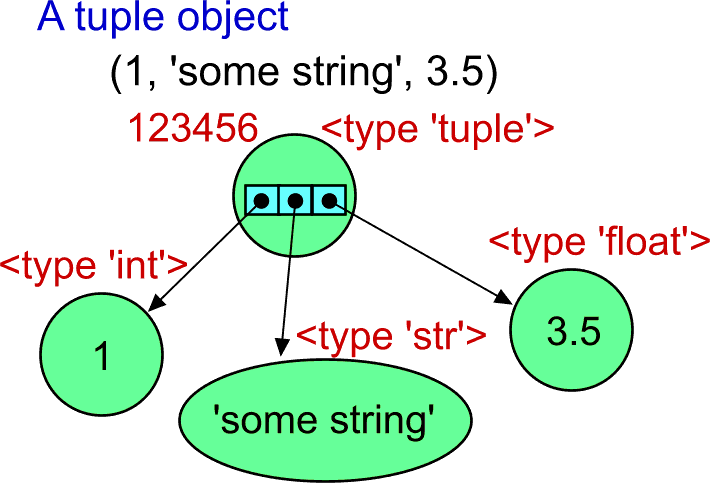
Tuples came from mathematics.  They contain certain number of elements in some order. A single tuple in Python can contain elements of more types.

>>> t = (1, 'some string', 3.5)

>>> t

(1, 'some string', 3.5)

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-8)

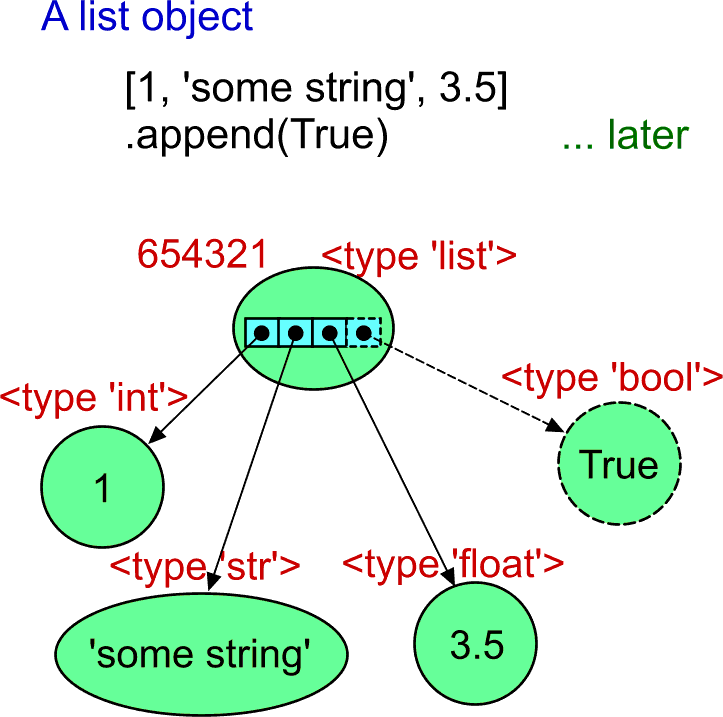
Now, how should we imagine the tuple object?  How it looks inside?  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489376/05TupleExample.png)Surprised?  The tuple object does not contain the elements inside.  The element objects are located outside, and they are bound to the tuple object only via refrences.  The value part of the tuple object from the example is actually only the array of three references.  In other words, a tuple stores the array of references of fixed size.  It means that the tuple object with three elements has always the same memory footprint -- it uses the same amount of memory independently on how big are the element objects.  
  
References in Python are always untyped -- they are references to *any type*.  This means that the element of a tuple can be the object of *any type/class*.  
  
There is one important thing to be mentioned.  The tuple object cannot be changed after it was created.  This means that its value is constant.  What does it *really* mean?  The static array of references cannot be changed.  If the refered element objects are also constant, then the logical value of the tuple remains the same "forever".  
  
What about lists?

>>> lst = [1, 'some string', 3.5]

>>> lst

[1, 'some string', 3.5]

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-9)

How can we imagine the list object?  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489377/06ListExample.png)

The list object is very similar to the tuple objects.  But there is one significant difference.  The array of references (that act as the value of the object) is a dynamic array. It can change the number of elements.  Being capable of growing and/or shrinking, it also means that it makes also sense to be able to change the value of the existing references (to assign another reference value).  This means that the list object with the given identity can change completely during its lifetime.    
  
So far, I tried to avoid using the variable names.  Why? It will be clear at the end of this part.  Anyway, it is neccessary to use the variable now for demonstration of the difference between the list and the tuple.  The *lst* variable was assigned the list object.  The same variable can be used later to call the *append()* method of the object.  The list object gets one more element (it becomes bigger; see the dashed-line parts of the above image).

>>> lst = [1, 'some string', 3.5]

>>> lst

[1, 'some string', 3.5]

>>> lst.append(True)

>>> lst

[1, 'some string', 3.5, True]

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-10)

When trying to do the same with the tuple object...

>>> t = (1, 'some string', 3.5)

>>> t

(1, 'some string', 3.5)

>>> t.append(True)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'tuple' object has no attribute 'append'

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-11)

... not only that the tuple class does not define the *append()* method, but there is also no other way to modify the tuple.  You can think about the tuple object as about some *frozen list*.  The references inside the object cannot be assigned.  Also, the array of the references cannot grow.  There will always be three and only the three references inside, in this case.  
  
I know I am repeating what was said above.  But we have to be careful what does it mean.  The same tuple can actualy represent something that changes!  Let's use the list as another element of the tuple.  Of course, the list object has to be created first.  (Note: I have to use the named variables again even though they will be explained later.  They will be drawn as separated references with the name on the image.  From now on, I am also going to use simplified visualization of the type information of the objects if it is apparent.)

>>> lst = [1, 'some string', 3.5]

>>> lst

[1, 'some string', 3.5]

>>> t = (1, 'some string', 3.5, lst)

>>> t

(1, 'some string', 3.5, [1, 'some string', 3.5])

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-12)

The list object referenced by the *lst* variable was included as the fourth element of the tuple. (It is a different tuple than in the earlier example.  It has four elements now.)  But the *lst* variable is still available, and it can be used to call the method of the referenced list object:

>>> lst.append(True)

>>> lst

[1, 'some string', 3.5, True]

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-13)

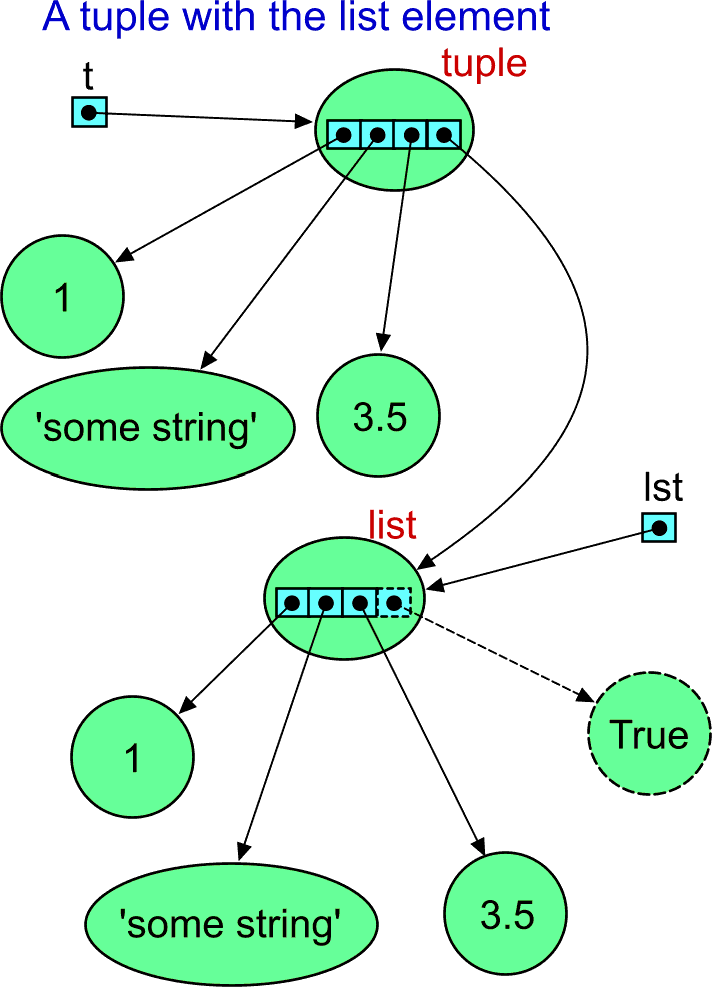
The list object changed.  What about the tuple that you may consider to be independent on the situation?

[1, 'some string', 3.5, True]

>>> t

(1, 'some string', 3.5, [1, 'some string', 3.5, True])

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-14)

Ooops!  The tuple is different now!  And they say the tuple cannot be changed!  And this is true.  The tuple object value did not change because the references in the internal array are the same.  However, there is no way to display symbolicaly the list object other than using its representation.  The image explains it better:  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489379/07TupleWithListElement.png)

If you payed attention, you may ask: "Where is the sharing of the **1** object?"  We can check the identifications (the numbers changed as they are usually different for each run of Python):

>>> id(lst[0])

31021656L

>>> id(t[0])

31021656L

>>> id(lst[1])

31160288L

>>> id(t[1])

30854384L

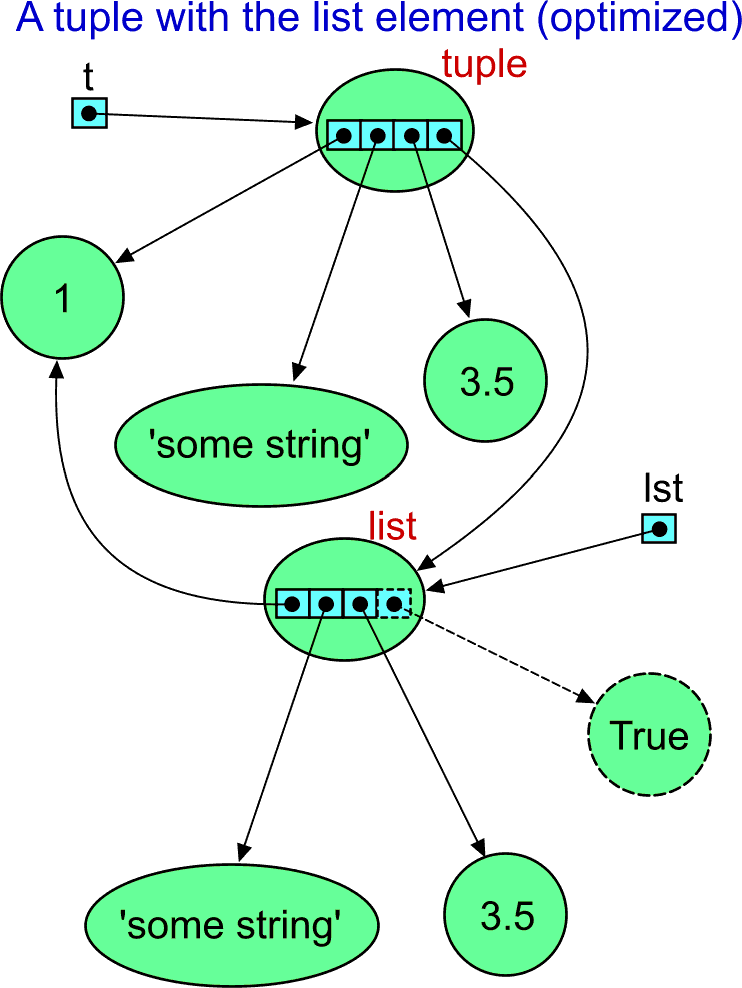
>>> id(lst[2])

31082424L

>>> id(t[2])

31082400L

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-15)

Notice that both the list and the tuple can be indexed (zero based) as if they were arrays.  Now you already know why!  It comes almost for free in the Python language, because both tuples and lists store the array of references inside.  Indexing here means getting the reference from the array on the index.  It is automatically dereferenced, and you have the access to the element object.  
  
And yes, the identification of the integer objects from the list and from the tuple are the same.  It means that the object is the same, identical.  The object is shared even though there is no apparent explicit reason.  Recall that it is one of the Python interpreter optimizations.  Then the image should look like that:  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489380/08TupleWithListElementOptimized.png)The most important conclusion so far is that containers always contain references to the objects with the value.  They do not store the value explicitly inside itself.  The "strange behaviour" usually means that the person who complains is not aware of the usage of references instead of target-object values.  Once you know that, you also know the positive consequences.  Working with the container of certain size take the same time independently on how complex are the element objects.

## Sets

The documentation says: "A *set* object is an unordered collection of distinct *hashable* objects."  Unlike a tuple of a list object, a set object does not capture the order of collected elements.  Try the following:

>>> mySet = set([1, 'some string', 3.5])

>>> mySet

set(['some string', 1, 3.5])

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-16)

Well, the strange syntax... The set type was added in Python version 2.4, which means quite late.  Actually, the above code does not use any special syntax.  You can even (re)define the *class set* that would be used the same way.  As we need to pass some initial elements when the object is created, we need some other container that contains them.  The authors of Python prefer a list.  However, you can pass also other iterable objects.  Try it with the tuple:

>>> set( (1, 'some string', 3.5) )

set(['some string', 1, 3.5])

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-17)

As for other types, if possible, the representation of the object has the form that could be copy/pasted to a source file to get the object with the same content.  Even though we passed a tuple, the representation shows the list as the argument.  This is because there is no captured knowledge what was passed to the constructor.  Notice also that the order of the elements in the sequence has changed.  This is because the set object does not caputre the initial order of elements either.  
  
Newer versions of Python support also the new syntax for creation of a set object it uses curly braces -- as usual in mathematics.  There is one exception.  The empty set cannot be expressed as **{}** because it is already used for empty dictionary object.  We must use **set()** or **set([])** instead:

>>> {1, 'some string', 3.5}

set([3.5, 'some string', 1])

>>> set()

set([])

>>> type(set())

<type 'set'>

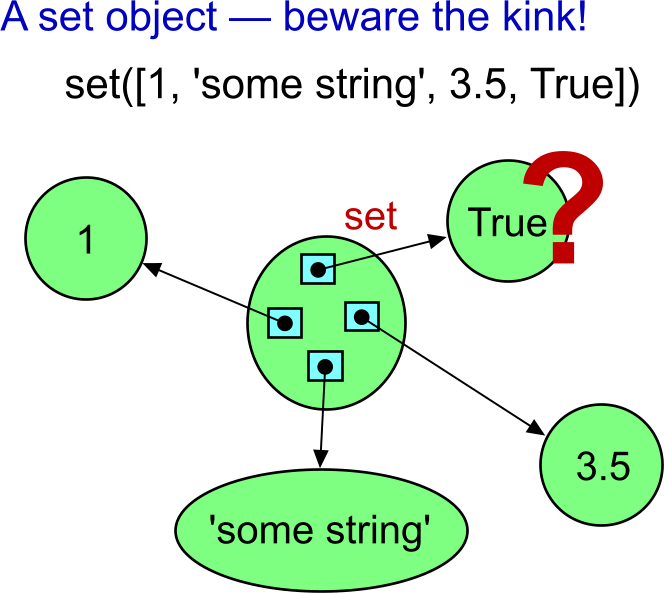
>>> {}

{}

>>> type({})

<type 'dict'>

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-18)

"OK! Now I know how Python set work internally!"  Well, there are some kinks that you would not expect.  You may want to create the set like this:  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489408/09SetObjectExampleKink.png)

"No problem!"

>>> set([1, 'some string', 3.5, True])

set(['some string', 1, 3.5])

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-19)

"Oops! Where is my **True**?"  Well, this may be quite surprising.  To explain that behaviour, we have to talk about hash values.  Probably the simplest way to demonstrate is to use the built-in function named...

## hash()

The *hash(arg)* built-in function returns so called *hash value* of the passed argument.  
  
Where the name *hash* came from... Do you now that food made of chopped meat?  Even when the resul looks differently than the raw meat, you can still guess that it was made of meat.  When you chop some vegetables, the result will look differently.  But if you chop another portion of the same kind of meat, the sample will look the same as the other meat sample.  You can take only small amount of the result of chopping, and you can still guess if it belongs to the chopped meat or to the chopped vegetable.  You can decide using the small sample, even though the samples differ from the originals.    
  
The *hash()* function does the same with the object passed as the argument.  The sample has the form of an integer number.  When you get the same number, the source was somehow ekvivalent. Try:

>>> hash(1)

1

>>> hash('some string')

-604248944

>>> hash(3.5)

1879113728

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-20)

You can see that it always return some integer value.  However, not every object is *hashable*.  Try:

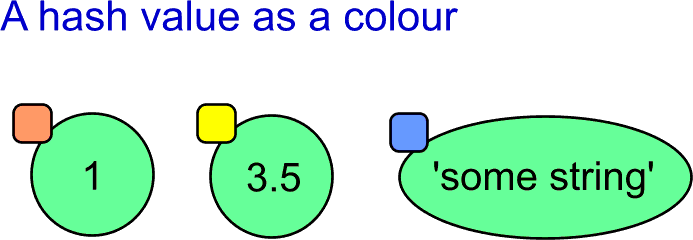
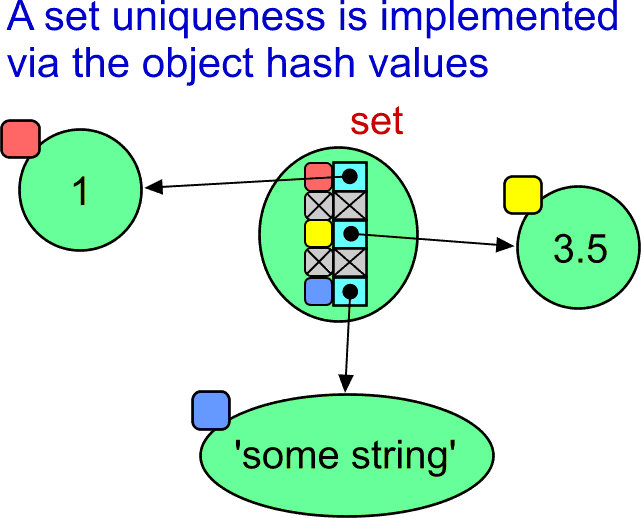
>>> hash([1, 2, 3])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unhashable type: 'list'

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-21)

In this case, the list is not hashable because it can change its content.  This means that the next time the object would return a different hash number.  Because of this, the concrete hash number cannot always represent the content of the content of the list object.  
  
I will use the small square with a distinct colour to show the value of the hash of the object.  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489383/10hashAsColour.png)Now back to the set type.  The set always contains one value only once.  In other words, it contains unique values.  On the other hand, the set in Python can be used as a container for objects of different types.  We need some general mean to decide whether the object is inside the set or not.  Here comes the hash value of the object.  
  
When testing whether the object is inside the set or not, we calculate *hash(obj)* first.  Then we can check, whether the object with the same hash value is inside.  The hash function has also one important feature.  It can be easily transformed to the index of an array element.  When being lucky, the element contains information about the object with the same hash value.  When being not so lucky, there is a conflict (two hash values converted to the same index).  The conflict must be resolved by some additional mechanism.  You can find the reference to more detailed article below in the section describing dictionaries.  The internal array must be bigger than the number of elements to minimize the conflicts.  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489384/11SetHashAndUniqueness.png)However, there is one kink with Python hash values.  Sometimes, different objects may have the same hash value:

>>> hash(1)

1

>>> hash(1.0)

1

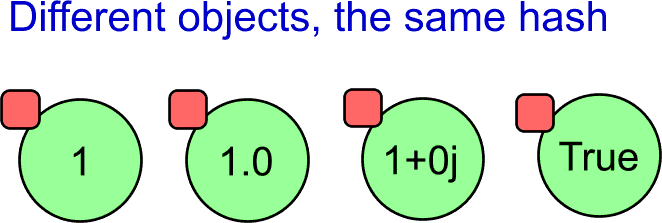
>>> hash(1+0j)

1

>>> hash(True)

1

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-22)

[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489385/12sameHash.png)This is the reason, why the *True* was not inserted into the set in the earlier example.  When searching for the index in the internal array, the hash value of the True object was already found at the index.  The insertion algorithm decides only based on the hash value, even though the object present in the set represents the *1* (of the integer type).  Because of that simplification, Python interpreter decided not to insert *object with the same value* again. There are always some tradeoffs when you look deep enough.  Everywhere.  
  
A final note related to hashability.  An object of the same type can be sometimes hashable and sometimes not.  For example a tuple object:

>>> type( (1, 'some string', 3.5) )

<type 'tuple'>

>>> hash( (1, 'some string', 3.5) )

1562160248

>>> type( (1, 'some string', 3.5, []) )

<type 'tuple'>

>>> hash( (1, 'some string', 3.5, []) )

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unhashable type: 'list'

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-23)

If the tuple contains only hashable items, then the whole tuple is hashable.  This means that we can get the number that can be used as a signature of the object.  However, when the tuple contains a list, it is not hashable any more.  This is because the content of the list object can be modified, and there is no fixed number that could represent the tuple content.

## Mutable and immutable... What does it mean?

When reading the Python documentation or articles, you will definitely find the terms *mutable* or *immutable*.  Fancy words, simple meaning.  A mutable object can be changed (its content) during lifetime.  An immutable object cannot change its content after it was created.  In other words, immutable means *with the constant content*, mutable means *with the content that can be modified*.  But remember, that the content of the container is the array of references.  If the container is immutable (e.g. of the tuple type), then only the references are constant.  Some or all of the refered objects may still be mutable.  
  
So, what are the immutable types that we have mentioned so far?  All simple built-in types are immutable: integer, float, complex, string.  The tuple type is immutable.  There is also the immutable version of the set type.  The type is called *frozenset*.  
  
Why so much about hashability and mutability?  We will use that term later.  There is one type heavily used in Python as another type that is based on hashing.  The type is named *dict*.

## Dictionaries

Dictionary type is one of the major data types in Python.  It must be implemented very efficiently not only because *you* want to use it, but also because the Python interpreter uses it even when you are not looking.  
  
What the doc says about the dictionary type...

A *mapping* object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. [...]  
  
A dictionary’s keys are almost arbitrary values. Values that are not hashable [...] may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry.

As for the mentioned numeric types for keys, we already know why it is so.  This is because the hash value of the keys plays the role when accessing the items of a dictionary.  If you understand the above *set* type, it will not be difficult to understand the *dict* type for you.  

>>> dict()

{}

>>> {}

{}

>>> type({})

<type 'dict'>

>>> type(dict())

<type 'dict'>

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-24)

A dictionary object can be created the same way as any other object -- via calling the class name as if it were a function (see the *dict()*).  (You may prefer to name this as "using the class constructor", but there area some differences in Python, and it does not define the term *constructor* explicitly.  
  
However, there is also a special syntax for creating a dictionary object in Python.  It uses curly braces.  Using empty curly braces mean creating an empty dictionary.  (A side note: This works in Python from early days.  Do you remember the new syntax for creating the set objects?  Here is the reason why the empty set cannot be created this way.)  
  
Usually, dictionary objects are created and initialized using the special syntax with curly braces.  In some cases, we need to prescribe the content of the dictionary explicitly in the source code.

>>> {'a': 1, 'bb': 'some string', 5: 3.5, 'xxx': True}

{'a': 1, 'xxx': True, 5: 3.5, 'bb': 'some string'}

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-25)

Notice that the representation of the just created dictionary has the same form.  Notice where the spaces are placed and where they are not.  This may be a minor issue.  Still, keeping the style helps to make your programs more readable.  If you get used to, you will also more easily swallow the sources written by the others.  
  
Notice also, that the representation of the above dictionary content shows the items in a different order.  The reason is the same as in the set type.  A dictionary organizes the items inside differently than say a list type.  There is no way to get the original order.  
  
Sometimes we get data from other sources in another form, and we want to fill the dictionary with them.  The following example show the list of tuples that is used for initialization:

>>> dict([('a', 1), ('bb', 'some string'), (5, 3.5), ('xxx', True)])

{'a': 1, 'xxx': True, 5: 3.5, 'bb': 'some string'}

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-26)

The *dict()* constructor is capable to consume whatever sequence of tuples.  This means that it can be a tuple of tuples instead of the list of tuples.  It can also be set of tuples.  The rule is that the dictionary initialize have to be able iterate through the items and then the item must contain two ordered elements -- the key and the value:

>>> dict( (('a', 1), ('bb', 'some string'), (5, 3.5), ('xxx', True)) )

{'a': 1, 'xxx': True, 5: 3.5, 'bb': 'some string'}

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-27)

Notice a kind of strange example that shows a tuple of strings with two characters.  It also meets the rules.  The two-char string is split to two chars.  The first one is used as the key, the other as the value.  (But this is purely for illustration of how it works.  Do not search for any meaningful usage of the case.)

>>> dict( ('aA', 'bB', 'cC') )

{'a': 'A', 'c': 'C', 'b': 'B'}

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-28)

A dictionary works as an associative array.  Therefore the syntax directly reflects the idea.  You can use a key value as if it was an index.  You can read the existing value, and you can assign both the existing item (i.e. modification) or the new item (i.e. creation).  We can also delete an element using the *del* statement.  (We need the variable name for that.  The variables will be explained a bit later, as said earlier.)

>>> d = {'a': 1, 'bb': 'some string', 5: 3.5, 'xxx': True}

>>> d['a']

1

>>> d['bb']

'some string'

>>> d[5]

3.5

>>> d['xxx']

True

>>> d[0]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 0

>>> d[0] = 'new string'

>>> d[0]

'new string'

>>> d

{'a': 1, 0: 'new string', 'xxx': True, 5: 3.5, 'bb': 'some string'}

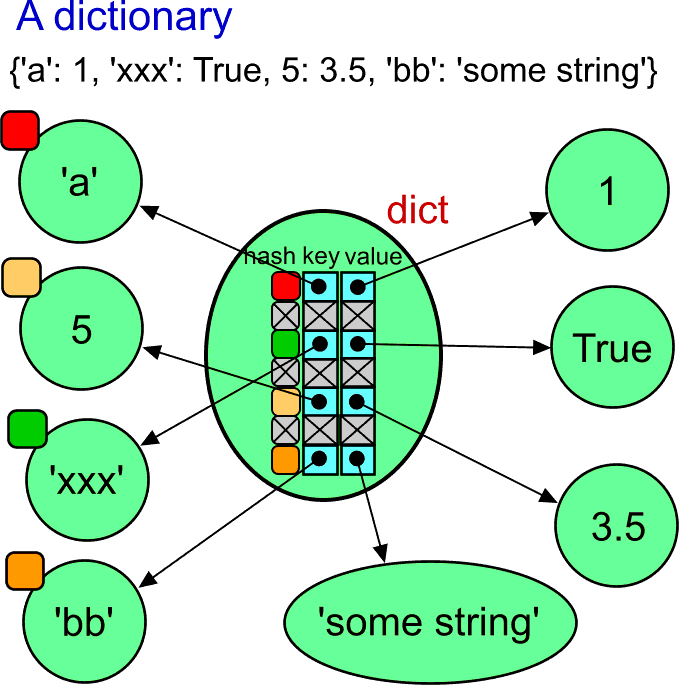
>>> del d[0]

>>> d

{'a': 1, 'xxx': True, 5: 3.5, 'bb': 'some string'}

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-29)

## Dictionary illustrated

The time has come to have a look inside.  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489386/13dict.png)There is plenty of objects here.  However, keep in mind that the dictionary contains only references to them.  Whatever you do with the dictionary, you modify only the internal array of references.  
  
When searching associatively for a value, the hash value of the key is calculated.  The hash value is transformed into the index to the internal hash table.  If there is no conflict, you get the reference to the value object very quickly.  
  
The hash value of the value object does not play any role.  Therefore, it is not illustrated at the value objects.  Actually, the value object need not to be hashable and it may not be possible to calculate their hash value.  For example, the value object could be a list or a dictionary.  
  
If you are interested in implementation of dictionaries, you can find more detais at <http://www.laurentluce.com/posts/python-dictionary-implementation/>.

## Variables

Now it comes.  What really are the variables in Python?  Variables in Python are named references.  The name is stored in memory during runtime.  It is not lost during compilation as in classical compiled languages.  A Python variable has no type associated to the name (unlike in compiled languages).  This is because a Python variable stores always the same type of value.  It stores a reference to the target object.  
  
And how the names and the references are bound together.  You may have already guessed.  The variable names are keys to a hidden dictionary (used by the Python interpreter for the purpose).  The references to the targets are at the value side of that dictionary.  
  
Actually, the hidden dictionaries are not that hidden.  There are functions that gives you the access to the dictionaries.  One of them is the built-in function named *globals()*.  
  
Start the Python interpreter in interactive mode from scratch and try the following:

Python 2.7.1 (r271:86832, Nov 27 2010, 17:19:03) [MSC v.1500 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> d = {'a': 1, 'bb': 'some string', 5: 3.5, 'xxx': True}

>>> d

{'a': 1, 'xxx': True, 5: 3.5, 'bb': 'some string'}

>>> var1 = 4

>>> var2 = 'whatever string'

>>> var3 = False

>>> g = globals()

>>> type(g)

<type 'dict'>

>>> g

{'var1': 4,

'd': {'a': 1, 'xxx': True, 5: 3.5, 'bb': 'some string'},

'var3': False,

'var2': 'whatever string',

'\_\_builtins\_\_': <module '\_\_builtin\_\_' (built-in)>,

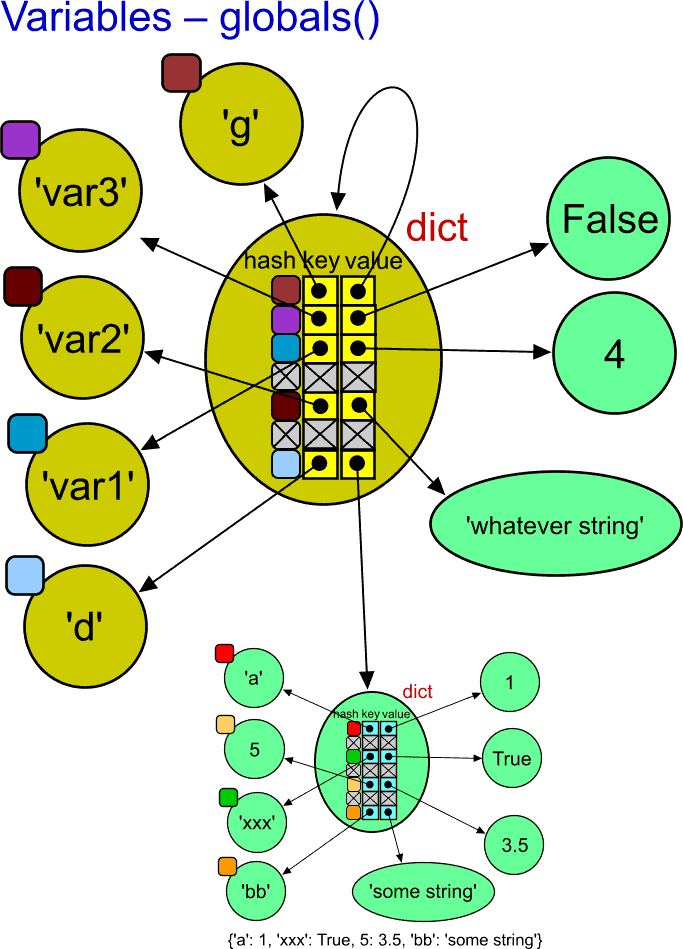
'\_\_package\_\_': None,

'g': {...},

'\_\_name\_\_': '\_\_main\_\_',

'\_\_doc\_\_': None}

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-30)

Notice the last variable named *g*, that was assigned the result of the *globals()*.  Next, we have tried it is of the *dict* type.  In other words, it is a normal dictionary, indistinguishable from other dictionary objects.  When displaying its representation, we can see all the defined variables inside, plus some more.  (The dictionary representation was manually formatted to make the dictionary items more visible.)  
  
Notice that even the *g* name is inside.  But its value cannot be easily represented as it points to itself.  Let's illustrate it!  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489388/14globals.png)The reason for using the dark yellow colour of the object is to emphasize that they were created by the Python interpreter for its internal purpose.  Otherwise, the object are quite normal objects of their type.    
  
There are some restictions for identifiers in any programming language. They usually must be compound of letters, numerals, and some special characters (e.g. undercore).  To make it short, the keys in the hidden dictionary must be of the string type.

## Assignment operation

Now it should be clear that the assignment to a variable means that the string form of its name is or found in the appropriate internal dictionary as a key, or the item for the key is created.  The assigned value is always some object.  The reference to the object is used as the value part of the dictionary.

## More variables

What about the dirty idea to create dynamically a variable that was not given its identifier in the source code.  Say, its name could be dynamically created, read from a text file or whatever.  Here we simulate it by assigning the *s* variable.

>>> s = 'myVariable'

>>> g[s] = 12345

>>> g

{'var1': 1,

'd': {'a': 1, 'xxx': True, 5: 3.5, 'bb': 'some string'},

'var3': False,

'var2': 'whatever string',

'\_\_builtins\_\_': <module '\_\_builtin\_\_' (built-in)>,

'\_\_package\_\_': None,

'myVariable': 12345,

'g': {...},

's': 'myVariable',

'\_\_name\_\_': '\_\_main\_\_',

'\_\_doc\_\_': None}

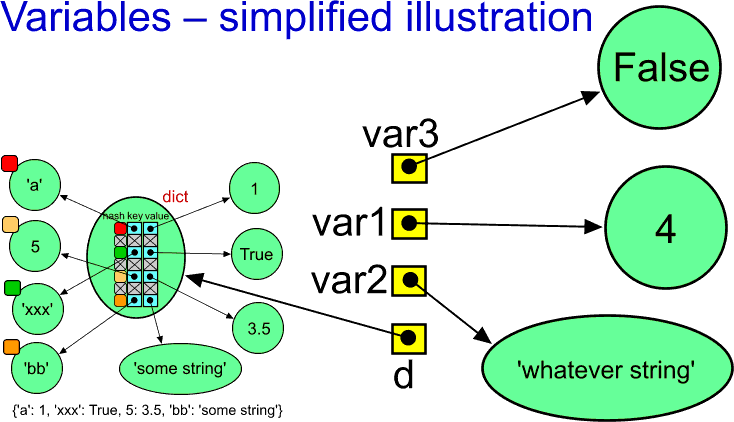
[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-31)

See, the *myVariable* with the content 12345 is there!  Can we use the newly created variable name as if it was defined in the source text?

>>> myVariable

12345

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-32)

It works!  Notice that it is very different in programming languages to write an identifier and write a string literal with the same name inside.  
  
Warning: the *globals()* name suggests that it stores global variables.  However, the dictionary stores only top-level variables *inside a module*.  There is that many of such dictionaries, how many modules is used in your program.  There also is the built-in *locals()* function that returns the dictionary of local variables.  However, you should never modify it.  
  
The above illustration shows how variables are implemented in Python.  However, it looks quite complex on its own.  To simplify the illustration of used variables, I will use pictures like the following.  It illustrates the same case -- except the *g* variable.  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489391/15variablesSimplified.png)

## Some final notes

"And why you see it changing not-- And why so strange are strings."

"Oh, now I know! Assigning a variable means sharing the object via another reference..."

>>> lst1 = ['first', 'list']

>>> lst2 = lst1

>>> lst1

['first', 'list']

>>> lst2

['first', 'list']

>>> lst2.append('extended')

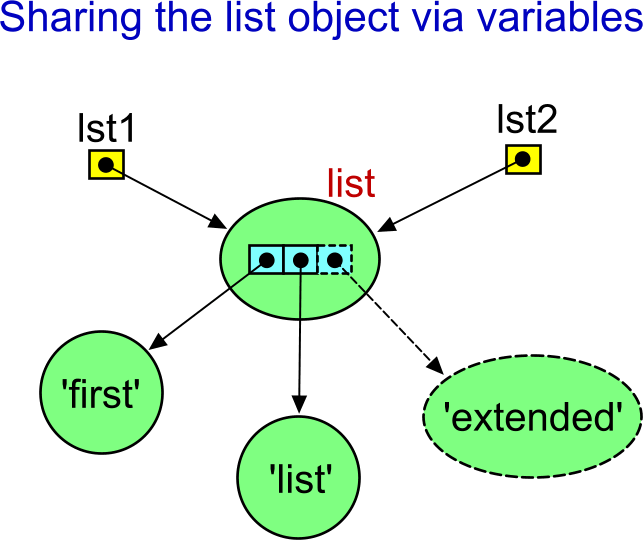
>>> lst2

['first', 'list', 'extended']

>>> lst1

['first', 'list', 'extended']

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-33)

[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489668/16sharedList.png)"Oh, yes.  I have told you!"  What about this?

>>> s1 = 'first string'

>>> s2 = s1

>>> s1

'first string'

>>> s2

'first string'

>>> s2 = s2 + ' extended'

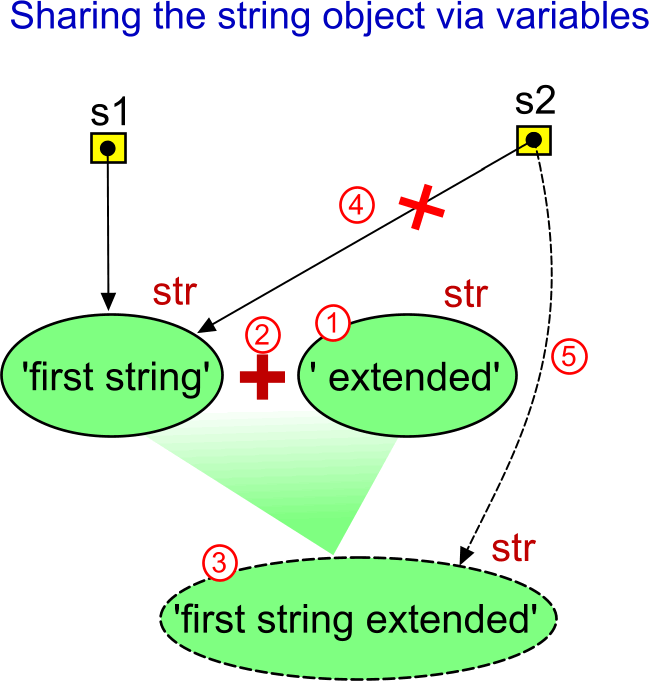
>>> s2

'first string extended'

>>> s1

'first string'

[Open in new window](http://www.experts-exchange.com/viewCodeSnippet.jsp?codeSnippetId=30-7109-34)

Why the list object behaves as expected (when knowing we work via references), and why the string behaves so strangely?  Is there any exception?  Do string variables also use references to the string object?  Aren't they implemented somehow differently?  
  
[](http://filedb.experts-exchange.com/incoming/2011/08_w34/489670/17modifySharedString.png)I hope it is the "Aha!" problem for you.  The key difference is that the list objects are mutable.  This means that the *.append()* method is able to modify the same object.  However, strings are immutable.  The concatenation operator (+) cannot modify the original string object.  The string object with the value *' extended'* must be created first, then the + operator puts the values of the two strings together and creates another string object.  Only after that the *s2* variable is assigned the reference to the newly created object.  Therefore the *s1* and *s2* cannot point to the same object.  (The string object with the value *' extended'* exists only temporarily.  It is not referenced later, and it will be destroyed by the garbage collector.)  
  
If you want to learn more about implementation of strings, see  
<http://www.laurentluce.com/posts/python-string-objects-implementation/>.  
  
(The end of the part 3)  
  
  
---------------------------------------------------  
Part 4 will reflect your questions...  
+ some consequences of the above,  
+ user defined classes.