Python Basics

Ragesh Hajela

Senior Programmer, Amadeus Labs

Contact : +91-9620121987, rageshhajela@gmail.com

Training Syllabus

1. **Introductory Sessions**
   a. **History of Python**
   b. **Introduction**
   c. **Starting with Python**
   d. **Execute python script**
2. **Basic Data Types**
   a. **Indentation**
   b. **Data Types and Variables**
   c. **Operators**
   d. **Lists and Strings**
   e. **List Manipulations**
   f. **Shallow and Deep Copy**
3. **Advanced Data Types**
   a. **Dictionaries**
   b. **Set and Frozen Sets**
   c. **Tuple**
   d. **Input via the keyboard**
4. **Conditional Statements**
   a. **If-else Block**
   b. **Loops, While Loop**
   c. **For Loop**
   d. **Iterators and Iterables**

Topic 1a: History of Python

Easy as ABC

- What do the alphabet and the programming language Python have in common? Right, both start with ABC.
- ABC is a general-purpose programming language and programming environment, which had been developed in the Netherlands, Amsterdam, at the CWI (Centrum Wiskunde & Informatica). The greatest achievement of ABC was to influence the design of Python.
- Python was conceptualized in the late 1980s.
- Guido van Rossum worked that time in a project at the CWI, called Amoeba, a distributed operating system.
- He experienced many frustrations with ABC and decided to try to design a simple scripting language that possessed some of ABC's better properties, but without its problems.
- He created a simple virtual machine, a simple parser, and a simple runtime. He made his own version of the various ABC parts that he liked.
- He created a basic syntax, used indentation for statement grouping instead of curly braces or begin-end blocks, and developed a small number of

powerful data types: a hash table (or dictionary), a list, strings, and numbers.

What about the name "Python"?

- Most people think about snakes, and even the logo depicts two snakes, but the origin of the name has its root in British humour.
- During Christmas vacation in 1989, he decided to write an interpreter for the new scripting language: a descendant of ABC.
- He chose Python as a working title for a "hobby" project, being in a slightly irreverent mood during Christmas'1989 (and a big fan of Monty Python's Flying Circus).

Development Steps of Python?

- Rossum published the first version of Python code in February 1991.
- Python version 1.0 was released in January 1994.
- In October 2000, Python 2.0 was introduced.
- Python flourished for another 8 years in the versions 2.x before the next major release as Python 3.0 was released in 2008.

- **Python 3 is not backwards compatible with Python 2.x**
- **Some changes in Python 3.0:**
  - **Print is now a function**
  - **Views and iterators instead of lists**
  - **The division of two integers returns a float instead of an integer. "//" can be used to have the "old" behavior.**
  - **raw_input() in Python 2.x is changed to input() & the former is discontinued.**
  - **input() in Python 2.x is changed to eval(input)**

Topic 1b: Introduction

Features

- **Multi-purpose (Web, Data Science, Automation, etc.)**
- **Procedural and Object-Oriented**
- **Interpreted**
- **Interactive Shell**
- **Strongly typed and Dynamically typed**
- **Focus on readability and productivity**
- **Cross Platform (CPython, Jython, IronPython, PyPy)**

Few Big Organizations using Python

- **Google**
- **NASA**
- **Netflix**
- **Dropbox**
- **YouTube**
- **New York University**
- **General Electric**
- **Juniper Networks**
- **Lego**
- **Nasdaq**
- **And the list goes on…**

Compiled v/s Interpreted Language

- **Many languages require you to *compile* (translate) your program into a form that the machine understands.**

compile   *execute*

source code `Hello.java`	byte code `Hello.class`	output
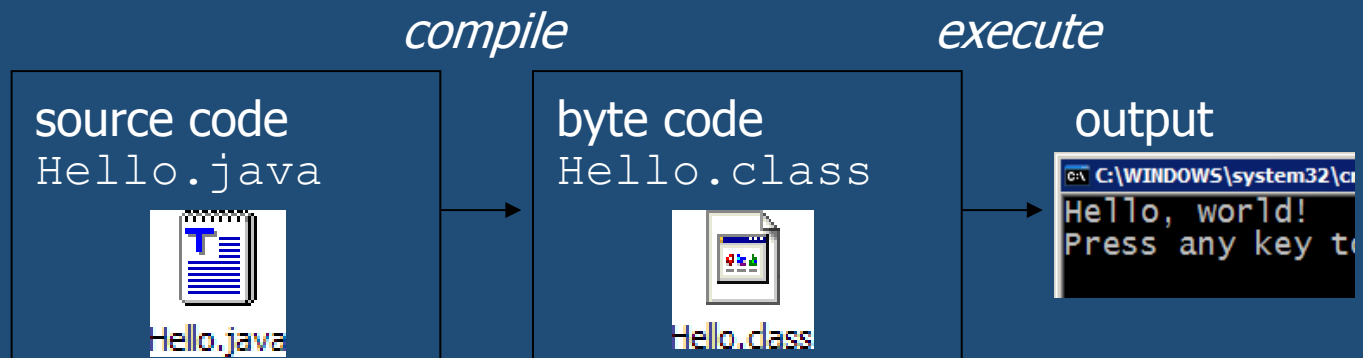
- **Python is instead directly *interpreted* into machine instructions.**

interpret

source code `Hello.py`	output
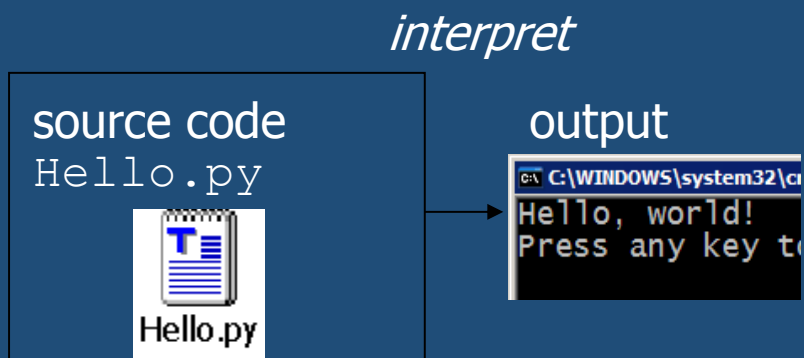
That's why Python is an interpreted language and not a compiled one. But, more insights on this topic will come in Topic 1d.

Topic 1c: Starting with Python

The Interpreter, an Interactive Shell

- The interactive shell is between the user and the operating system (e.g. Linux, Unix, Windows or others)
- The Python interpreter can be used from an interactive shell.
- The interactive shell is also interactive in the way that it stands between the commands or actions and their execution
- Python offers a comfortable command line interface with the Python shell, which is also known as the "Python interactive shell".

Setup

- We can access Python in four ways:
  - Download & install Python latest version from https://www.python.org/downloads/, and then launch IDLE. It is the interactive development environment for Python. We can access it from Windows Search Menu.
  - We can also launch Python – Command line directly after installation, instead of IDLE

- The Python interpreter can also be invoked by launching windows command prompt & typing the command "python" without any parameter followed by the "return" key at the shell prompt:

```
python
```

Python comes back with the following information

```
$ python
Python 2.7.11+ (default, Apr 17
2016, 14:00:29)
[GCC 5.3.1 20160413] on linux2
Type "help", "copyright",
"credits" or "license" for more
information.
>>>
```

This will need to set the environment variable in Advanced System Settings.
- Finally, other way is to download a Python IDE (Interactive Development Environment) such as PyCharm, Jupiter etc. These also have facilities for debugging a program using breakpoints.

- Recommended:
  - We will use IDLE for initial classes and then move to PyCharm IDE.

Let's try some simple commands

```
>>> hello
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
NameError: name 'hello' is not
defined
>>>
```

```
>>> print("Hello World")
Hello World
>>>
```

```
>>> "Hello World"
'Hello World'
>>> 3
3
>>>
```

- **How to quit the Python Shell?**
  - Exit(), or
  - Quit()

Let's try a simple calculator

```
>>> 4.567 * 8.323 * 17
646.18939699999999
>>>
```

- **Python follows the usual order of operations in mathematical expressions.**
- **The standard order of operations is expressed in the following enumeration:**
  - **exponents and roots**
  - **multiplication and division**
  - **addition and subtraction**
- **This means that we don't need parenthesis in the expression "3 + (2 * 4)"**

```
>>> 3 + 2 * 4
11
>>>
```

- **The most recent output value is automatically stored by the interpreter in a special variable with the name "_" and can be used in other expressions like any other variable**

```
>>> _ * 3
33
>>>
```

- **The underscore variable is only available in the Python shell. It's NOT available in Python scripts or programs.**
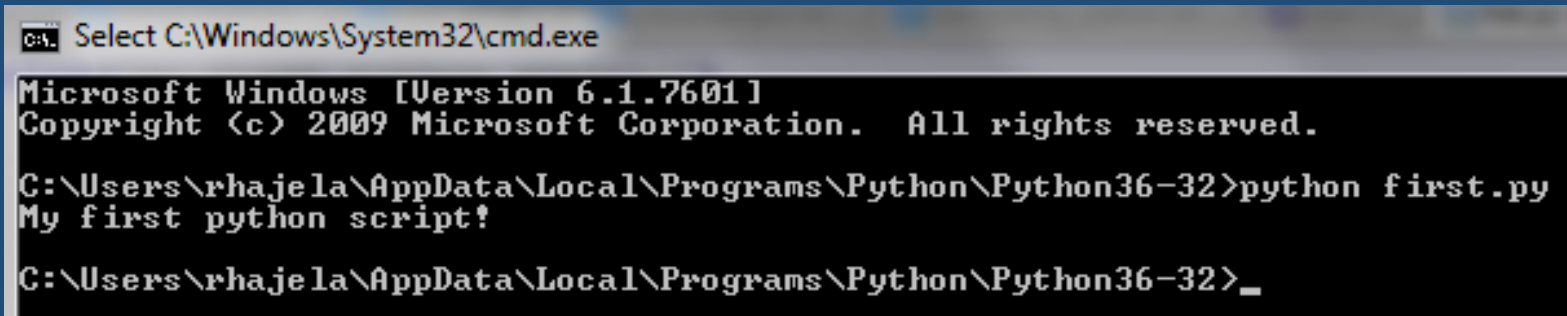
Let's try Basic Variables & Strings

```
>>> maximal = 124
>>> width = 94
>>> print(maximal - width)
30
>>>
```

```
  >>> "Hello" + " " + "World"
'Hello World'
```

```
  >>> ".-." * 4
'.-..-..-..-.'
>>>
```

Topic 1d: Execute python script

First Python Script

```
Select C:\Windows\System32\cmd.exe

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\rhajela\AppData\Local\Programs\Python\Python36-32>python first.py
My first python script!

C:\Users\rhajela\AppData\Local\Programs\Python\Python36-32>_
```

```python
print("My first python script!")
```

Python Internals

- As we have studied earlier that Python language is an interpreted programming or a script language.
- The truth is: Python is both an interpreted and a compiled language. But calling Python a compiled language would be misleading.
- What is a compiler?
  - A compiler is a computer program that transforms (translates) source code of a programming language (the source language) into another computer language (the target language), mostly assembly or machine code.
  - An interpreter is a computer program that executes instructions written in a programming language. It can either

- - execute the source code directly or
  - translates the source code in a first step into a more efficient representation and executes this code.
- People would assume that the compiler translates the Python code into machine language. Python code is translated into intermediate code, which has to be executed by a virtual machine, known as the PVM, the Python virtual machine.
- There is even a way of translating Python programs into Java byte code for the Java Virtual Machine (JVM). This can be achieved with Jython.
- Question: Do we have to compile our Python scripts to make them faster or how can we compile them? The answer is easy: No, you don't need to do anything because "Python" is doing the thinking for you automatically.
- But, if still we want to compile, can be achieved.

```
>>> import py_compile
>>>
py_compile.compile('my_first_si
mple_script.py')
>>>
```

Or

```
python -m py_compile
my_first_simple_script.py
```

- As a result, there will be a new subdirectory "__pycache__" created, if it hasn't already existed. And also find a file "my_first_simple_script.cpython-34.pyc" in this subdirectory. This is the compiled version of our file in byte code.
- We can also compile all scripts as this:

```
monty@python:~/python$ python -
m compileall .
Listing . ...
```

- How does .pyc file generate? - If Python has write-access for the directory where the Python program resides, it will store the compiled byte code in a file that ends with a .pyc suffix. If Python has no write access, the program will work anyway. The byte code will be produced but discarded when the program exits.
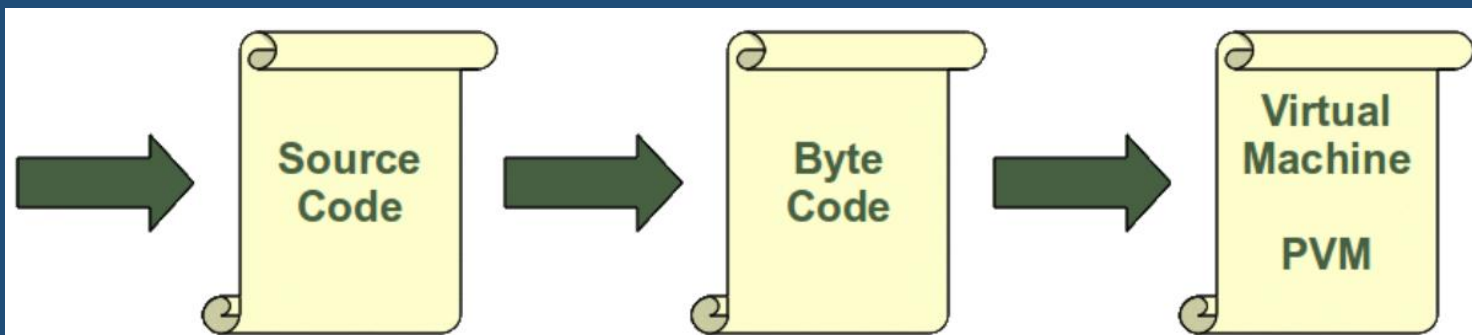
Topic 2a: Indentation

Blocks

- A block is a group of statements in a program or script. Usually it consists of at least one statement.
- A language, which allows grouping with blocks, is called a block structured language.
- Generally, blocks can contain blocks as well, so we get a nested block structure.
- A block in a script or program functions as a mean to group statements to be treated as if they were one statement.
- The first time, block structures had been formalized was in ALGOL, called a compound statement.
- Programming languages usually use certain methods to group statements into blocks:
  - Begin ... end : ALGOL, Pascal etc.
  - do ... done : Bourne and Bash Shell etc
  - Braces { ... } : C, C++, Java, Perl

Indenting Code

- Python uses a different principle. Python programs get structured through indentation, i.e. code blocks are defined by their indentation.
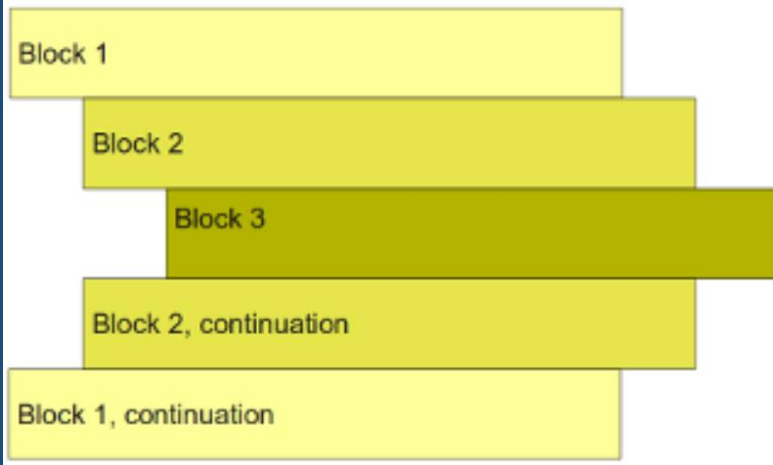
- **In Python, it's a mandatory language requirement, not a matter of style.**
- **This principle makes it easier to read and understand other people's Python code.**
- **So, how does it work? All statements with the same distance to the right belong to the same block of code.**
- **If a block has to be more deeply nested, it is simply indented further to the right.**
- **Example:**

```python
print("Start")
a = 1
if a == 1:
    print("Step1")
    b = 3
    if b == 2:
        print("step2")
    else:
        print("step3")
    print("step4")
print("end")
```

Topic 2b: Data Types & Variables

Basic Difference from other languages

If you want to use lists or an arrays in C or C++, you will have to construe the data type list or associative arrays from scratch, i.e. design memory structure and the allocation management. You will have to implement the necessary search and access methods as well.

But, Python provides power data types like lists and associative arrays, called dict in Python, as a genuine part of the language.

General Concept of Variables in other languages

- What is a variable? As the name implies, a variable is something, which can change.
- A variable is a way of referring to a memory location used by a computer program.
- In many programming languages, a variable is a symbolic name for this physical location.
- This memory location contains values, like numbers, text or more complicated types.

- We can use this variable to tell the computer to save some data in this location or to retrieve some data from this location.
- So, a variable can be seen as a container to store data.
- These concepts about variables fits best to the way variables are implemented in C, C++ or Java.

```
x = 42;
y = 42;
```



```
y = 78;
```
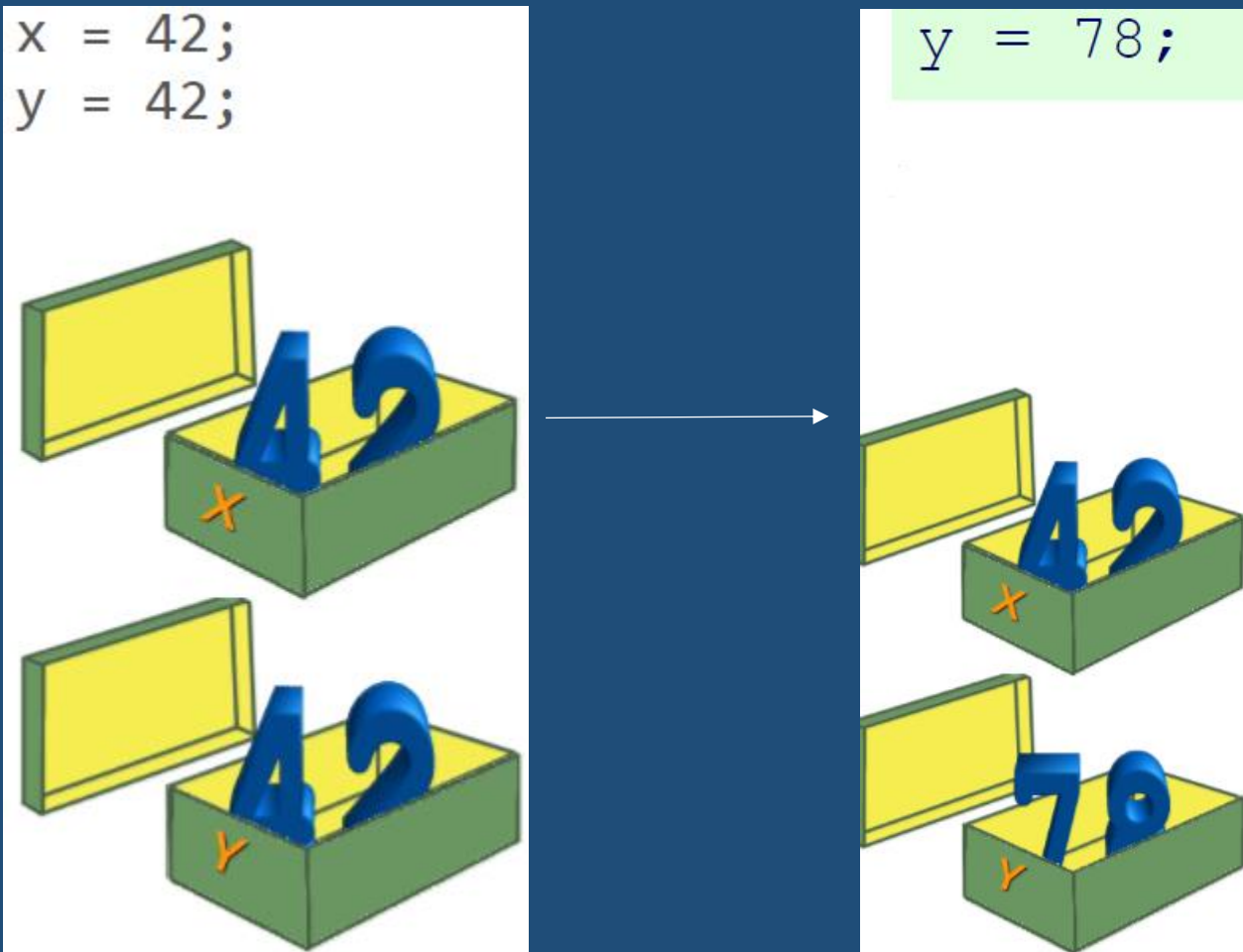
- Such declarations make sure that the program reserves memory for two variables (memory locations) with the names x and y. Putting values into the variables can be realized with assignments.
- Variables should have unique data type. E.g. if a variable is of type integer, solely integers can be saved in the variable during the duration of the program.

Variables in Python

- It's a lot easier in Python. There is no declaration of variables required in Python.
- If there is need of a variable, you think of a name and start using it as a variable.
- Another remarkable aspect of Python: Not only the value of a variable may change during program execution but the type as well.
- You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the same variable.
- In this line, we assign the value 42 to a variable:

```
i = 42
```

- The equal "=" sign in the assignment shouldn't be seen as "is equal to". It should be read as "is set to", meaning in our example "the variable i is set to 42".

```
i = 42                          # data
type is implicitly set to
integer
i = 42 + 0.11                   # data
type is changed to float
i = "forty"                     # and
now it will be a string
```

Object References

Python variables are references to objects, but the actual data is contained in the objects.
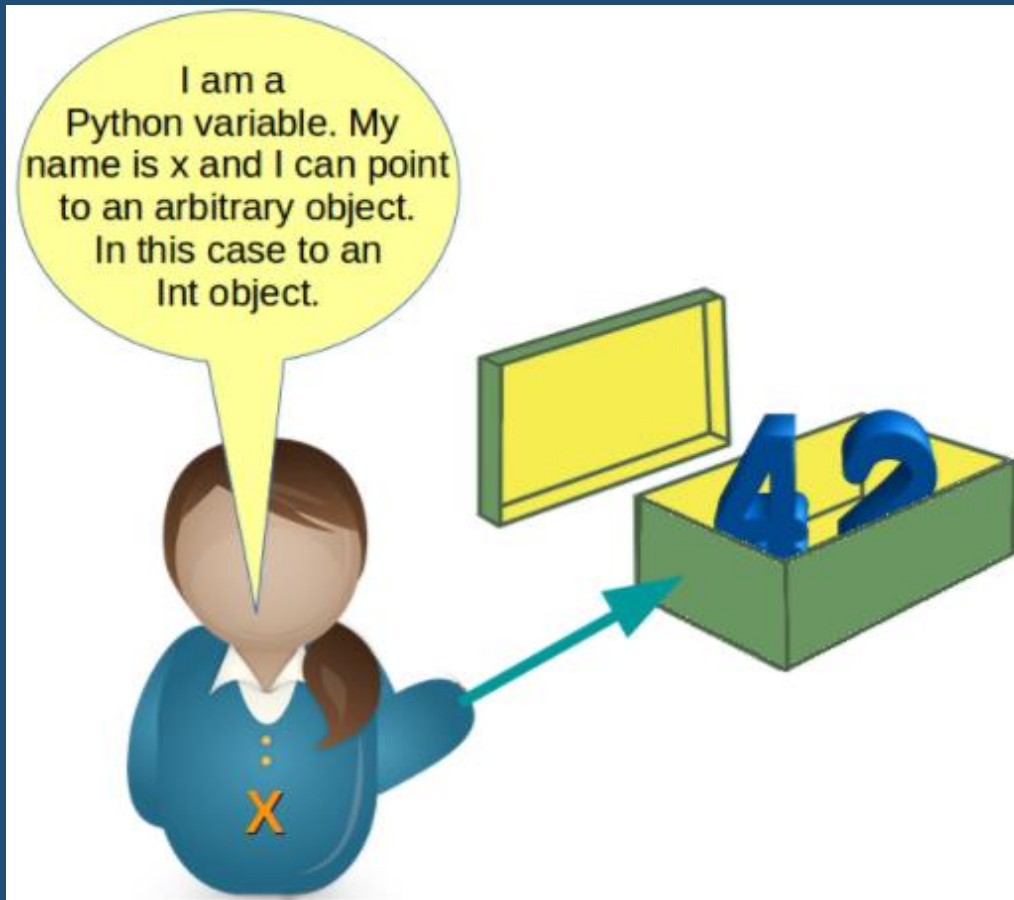
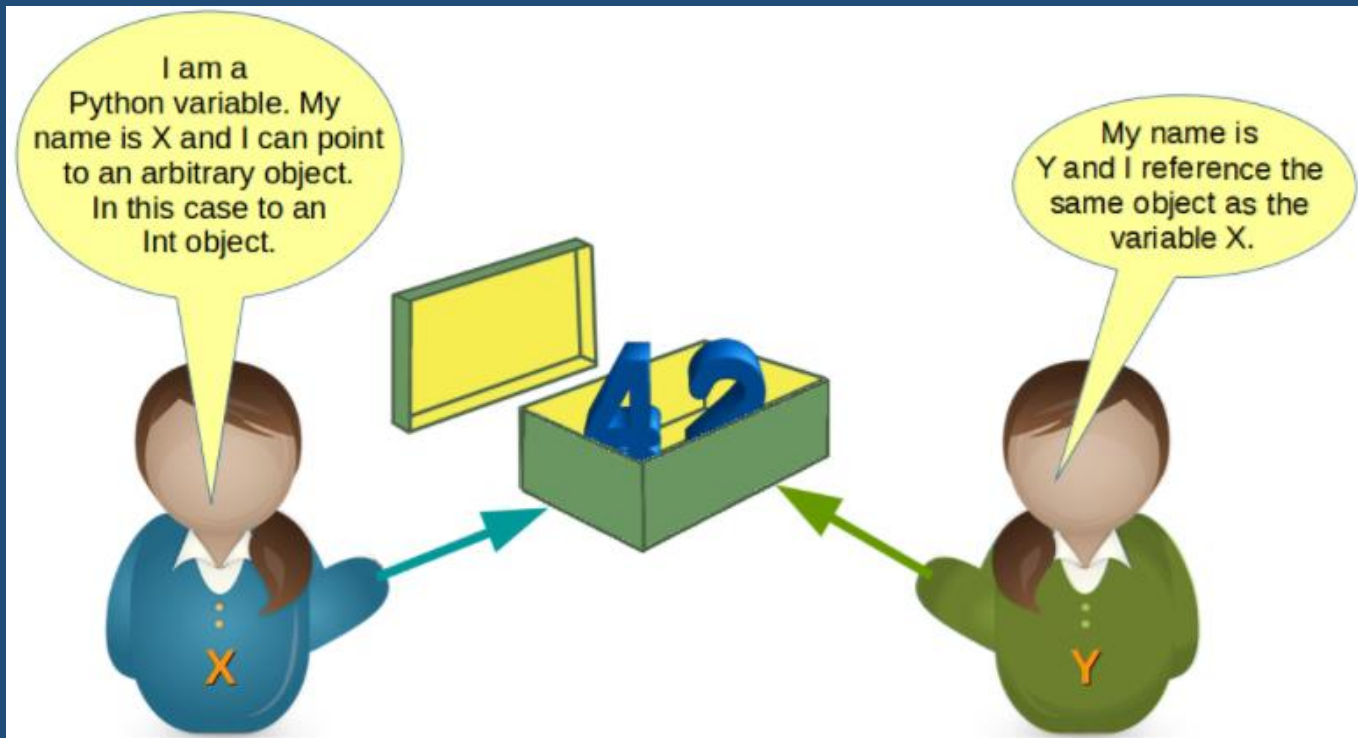As variables are pointing to objects and objects can be of arbitrary data type, variables can't have types associated with them. This is a huge difference to C, C++ or Java, where a variable is associated with a fixed data type.

```
>>> x = 42
>>> print(x)
42
>>> x = "Now x references a
string"
>>> print(x)
Now x references a string
```

Examples

```
>>> x = 42
>>> y = x
```



```
y = 78
```
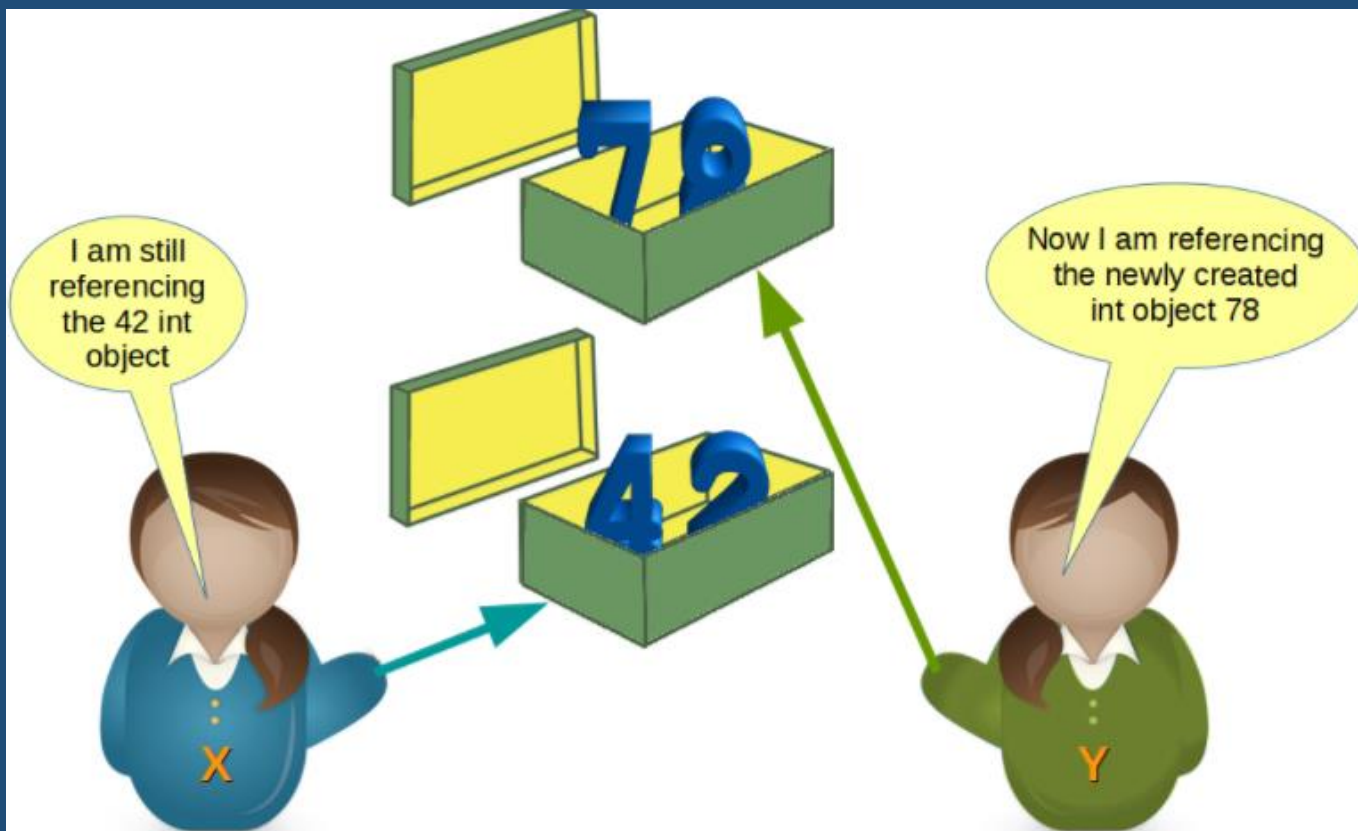
Now, let us do a string assignment to variable x.
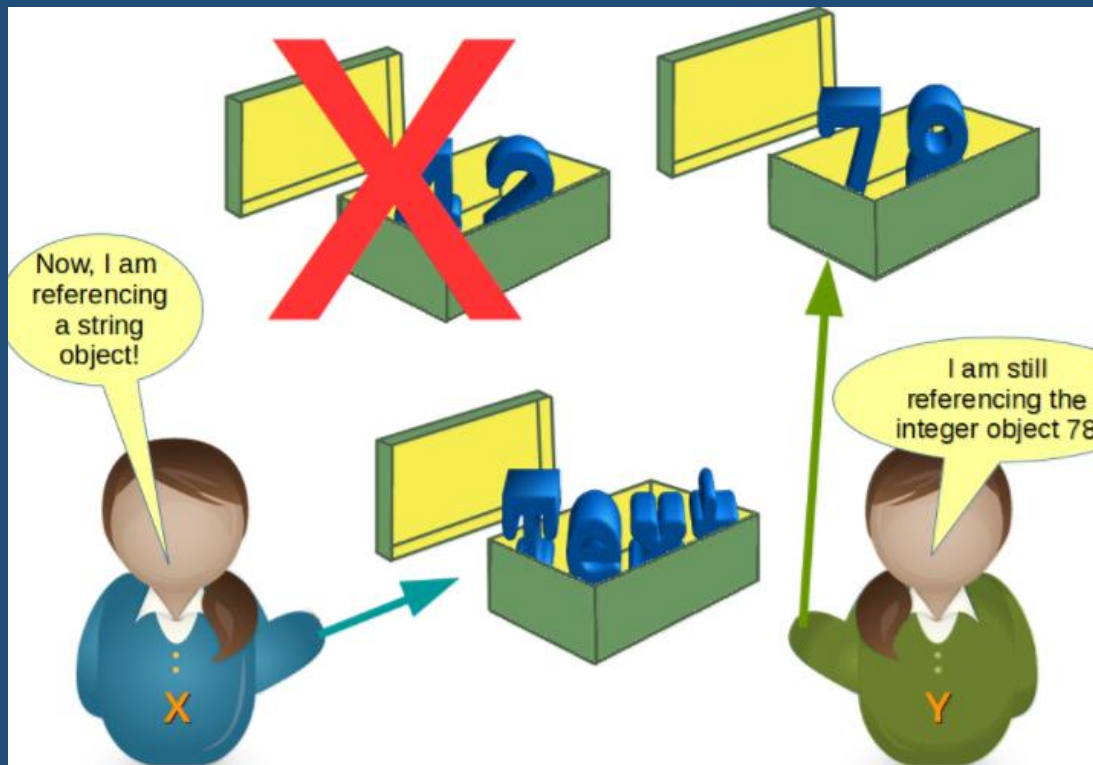
id function as a proof

The identity function id() can be used for this purpose. Every instance (object or variable) has an identity, i.e. an integer that is unique within the script or program, i.e. other objects have different identities.

```
>>> x = 42
>>> id(x)
10107136
>>> y = x
>>> id(x), id(y)
(10107136, 10107136)
>>> y = 78
>>> id(x), id(y)
(10107136, 10108288)
>>>
```

Valid Variable Names

- **A Python identifier is a name used to identify a variable, function, class, module or other object.**
- **A variable name and an identifier can consist of the uppercase letters "A" through "Z", the lowercase letters "a" through "z" , the underscore _ and, except for the first character, the digits 0 through 9.**
- **Identifiers are unlimited in length. Case is significant.**

Python Keywords

```
and, as, assert, break, class, continue, def,
del, elif, else,
except, False, finally, for, from, global, if,
import, in, is,
lambda, None, nonlocal, not, or, pass, raise,
return, True, try,
while, with, yield
```

```
>>> help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".
```

Numbers

There are four built-in data types for numbers:

- *Integer*
  - **Normal Integers e.g. 4321**
  - **Octal Literals (base 8): A number prefixed by 0o (zero and a lowercase "o" or uppercase "O") will be interpreted as an octal number**

```
>>> a = 0o10
>>> print(a)
8
```

  - **Hexadecimal literals (base 16): Hexadecimal literals have to be prefixed either by "0x" or "0X"**

```
>>> hex_number = 0xA0F
>>> print(hex_number)
2575
```

  - **Binary literals (base 2): Binary literals can easily be written as well. They have to be prefixed by a leading "0", followed by a "b" or "B"**

```
>>> x = 0b101010 >>> x 42 >>>
```

- **The functions hex, bin, oct can be used to convert an integer number into the corresponding string representation of the integer number**

```
>>> x = hex(19)
>>> x
'0x13'
>>> type(x)
<class 'str'>
>>> x = bin(65)
>>> x
'0b1000001'
>>> x = oct(65)
>>> x
'0o101'
>>> oct(0b101101)
'0o55'
>>>
```

- **Integers in Python3 can be of unlimited size**

```
>>> x =
78736609871273890324567823478235829283749
8729182728
>>> print(x)
78736609871273890324567823478235829283749
8729182728
>>> x * x * x
48812397007063821598677016210573131553882
75860919486179978711229502288911239609019
18308618286311523282239313708275589787123
0053171489685697978755581092352
>>>
```

- *Long Integers*: **In Python3 there is only one "int" type, which contains both "int" and "long" from Python2.**

- *Floating-point numbers*. For example, 42.11
- *Complex Numbers*: Complex numbers are written as <real part> + <imaginary part>j

```
>>> x = 3 + 4j
>>> y = 2 - 3j
>>> z = x + y
>>> print(z)
(5+1j)
```

Integer Division

There are two kinds of division operators:

1. "true division" performed by "/"

```
>>> 10 / 3
3.3333333333333335
>>> 10.0 / 3.0
3.3333333333333335
>>> 10.5 / 3.5
3.0
```

2. "floor division" performed by "//"

```
>>> 9 // 3
3
>>> 10 // 3
3
>>> 11 // 3
3
>>> 12 // 3
4
>>> 10.0 // 3
3.0
>>> -7 // 3
-3
>>> -7.0 // 3
-3.0
```

Strings

There are different ways to define strings in Python:

```
>>> s = 'I am a string enclosed in single
quotes.'
>>> s2 = "I am another string, but I am
enclosed in double quotes"
```

Both s and s2 of the previous example are variables referencing string objects.

Examples:

```
>>> s3 = 'It doesn\'t matter!'
```

```
>>> s3 = "It doesn't matter!"
```

```
>>> txt = "He said: \"It doesn't matter, if
you enclose a string in single or double
quotes!\""
```

```
txt = '''A string in triple quotes can extend
over multiple lines like this one, and can
contain
'single' and "double" quotes.'''
```

A string in Python consists of a series or sequence of characters - letters, numbers, and special characters. Strings can be subscripted or indexed.

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	e	l	l	o		W	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10

```
>>> s = "Hello World"
>>> s[0]
'H'
>>> s[5]
' '
```

```
>>> s[-1]
'd'
>>> s[-2]
'l'
```

```
>>> s[len(s)-1]
'd'
```

Python strings cannot be changed. Trying to change an indexed position will raise an error:

```
>>> s = "Some things are immutable!"
>>> s[-1] = "."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
assignment
```

Some Operators and functions for strings

- **Concatenation:**
  "Hello" + "World" will result in "HelloWorld"
- **Repetition:**
  "*-*" * 3 will result in "*-**-**-*"
- **Indexing:**
  "Python"[0] will result in "P"
- **Slicing:** "Python"[2:4] will result in "th"
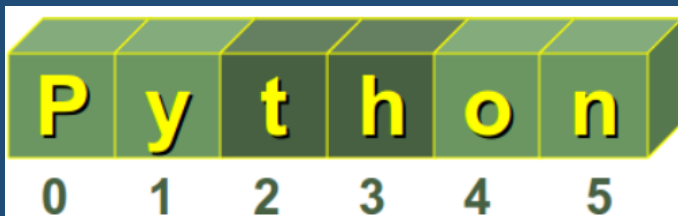
- **Size:**
  len("Python") will result in 6

A String Peculiarity

- **If both a and b are strings, "a is b" checks if they have the same identity, i.e. share the same memory location.**
- **If "a is b" is True, then it trivially follows that "a == b" has to be True as well.**
- **Yet, "a == b" True doesn't imply that "a is b" is True as well!**

Examples:

```
>>> a = "Linux"
>>> b = "Linux"
>>> a is b
True
```

```
>>> a = "Baden!"
>>> b = "Baden!"
>>> a is b
False
```

```
>>> a == b
True
```

```
>>> a = "Baden1"
>>> b = "Baden1"
>>> a is b
True
```

Escape Sequences in Strings

The backslash (\\) character is used to escape characters, i.e. to "escape" the special meaning, which this character would otherwise have.

\\\\	Backslash (\\)		
\\'	Single quote (')		
\\"	Double quote (")	\\n	ASCII Linefeed (LF)

Topic 2c: Operators

Arithmetic and Comparison Operators

These are the various built-in operators, which Python has to offer for any operation on numeric types.

Operator	Description	Example
+, -	Addition, Subtraction	10 – 3
*, %	Multiplication, Modulo	27 % 7 -> Result: 6
/	Division	10 / 3 -> Result: 3

//	Floor Division
	
	The results of int(10 / 3) and 10 // 3 are equal. But the "//" division is more than two times as fast!
	
	Import timeit
	print(timeit.timeit
	   ('for x in range(1, 100):y =100 // x',
	number=100000))
		10 // 3 -> Result: 3
		10.0 // 3 ->Result: 3.0

+x, -x	Unary Minus and Unary Plus (Algebraic signs)	-3
**	Exponentiation	10 ** 3 ->Result: 1000
or, and, not	Boolean Or, And, Not	(a or b) and c
in	"Element of"	1 in [3, 2, 1]
<, <=, >, !=	Comparison operator	2 <= 3

Topic 2d: Lists and Strings

Introduction to Lists

A list in Python is an ordered group of items or elements. These list elements don't have to be of the same type. It can be an arbitrary mixture of elements like numbers, strings, other lists and so on.

The main properties of Python lists:

- They are ordered
- They contain arbitrary objects
- Elements of a list can be accessed by an index
- They are arbitrarily nestable, i.e. they can contain other lists as sublists
- Variable size
- They are mutable, i.e. the elements of a list can be changed

List	Description
[]	An empty list
[1,1,2,3,5,8]	A list of integers
[42, "What's the question?", 3.1415]	A list of mixed data types
["Stuttgart", "Freiburg"]	A list of strings

[["London","England", 7556900], ["Paris","France",219 3031]]	A nested list
["High up", ["further down", ["and down", ["deep down", "the answer", 42]]]]	A deeply nested list

Accessing List elements

```
>>> languages = ["Python", "C", "C++", "Java",
"Perl"]
>>> print(languages[0] + " and " +
languages[1] + " are quite different!")
Python and C are quite different!
>>> print("Accessing the last element of the
list: " + languages[-1])
Accessing the last element of the list: Perl
```

Sublists

```
>>> person = [["Marc","Mayer"],["17, Oxford
Str", "12345","London"],"07876-7876"]
>>> name = person[0]
>>> print(name)
['Marc', 'Mayer']
>>> first_name = person[0][0]
>>> print(first_name)
Marc
>>> last_name = person[0][1]
>>> print(last_name)
Mayer
>>> address = person[1]
>>> street = person[1][0]
>>> print(street)
17, Oxford Str
```

```
>>> complex_list = [["a",["b",["c","x"]]]]
>>> complex_list = [["a",["b",["c","x"]]],42]
>>> complex_list[0][1]
['b', ['c', 'x']]
>>> complex_list[0][1][1][0]
'c'
```

Tuples

A tuple is an immutable list, i.e. a tuple cannot be changed in any way once it has been created.

A tuple is similar to lists, except that the set of elements is enclosed in parentheses instead of square brackets.

Where is the benefit of tuples?

- Tuples are faster than lists.
- Knowing that some data doesn't have to be changed, we should use tuples instead of lists, because this protects our data against accidental changes.
- Main advantage is that tuples can be used as keys in dictionaries, while lists can't.

```
>>> t = ("tuples", "are", "immutable")
>>> t[0]
'tuples'
>>> t[0]="assignments to elements are not
possible"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
item assignment
```

Slicing

To extract part of a string or a list, we can use the slice operator ":" in Python

```
>>> str = "Python is great"
>>> first_six = str[0:6]
>>> first_six
'Python'
>>> starting_at_five = str[5:]
>>> starting_at_five
'n is great'
>>> a_copy = str[:]
>>> without_last_five = str[0:-5]
>>> without_last_five
'Python is '
```

```
>>> cities = ["Vienna", "London", "Paris",
"Berlin", "Zurich", "Hamburg"]
>>> first_three = cities[0:3]
>>> # or easier:
...
>>> first_three = cities[:3]
>>> print(first_three)
['Vienna', 'London', 'Paris']
```

```
>>> all_but_last_two = cities[:-2]
>>> print(all_but_last_two)
['Vienna', 'London', 'Paris', 'Berlin']
```

Slicing works with three arguments as well. If the third argument is for example 3, only every third element of the list, string or tuple from the range of the first two arguments will be taken.

```
s[begin: end: step]
```

Example

```
>>> str = "Python under Linux is great"
>>> str[::3]
'Ph d n   e'
```

```
>>> s = "Toronto is the largest City in
Canada"
>>> t = "Python courses in Toronto by
Bodenseo"
>>> s = "".join(["".join(x) for x in
zip(s,t)])
>>> s
'TPoyrtohnotno  ciosu rtshees  lianr gTeosrto
nCtiot yb yi nB oCdaennasdeao'
```

```
>>> s
'TPoyrtohnotno  ciosu rtshees  lianr gTeosrto
nCtiot yb yi nB oCdaennasdeao'
>>> s[::2]
'Toronto is the largest City in Canada'
>>> s[1::2]
'Python courses in Toronto by Bodenseo'
```

Length

The length of a sequence, i.e. a list, a string or a tuple, can be determined with the function len().

For strings it counts the number of characters and for lists or tuples the number of elements are counted, whereas a sublist counts as 1 element.

```
>>> txt = "Hello World"
>>> len(txt)
```

```
11
>>> a = ["Swen", 45, 3.54, "Basel"]
>>> len(a)
4
```

Concatenation of Sequences

```
>>> firstname = "Homer"
>>> surname = "Simpson"
>>> name = firstname + " " + surname
>>> print(name)
Homer Simpson
```

```
>>> colours1 = ["red", "green","blue"]
>>> colours2 = ["black", "white"]
>>> colours = colours1 + colours2
>>> print(colours)
['red', 'green', 'blue', 'black', 'white']
```

augmented assignment "+=" e.g. s += t → s = s + t

Checking if an element is in list

```
>>> abc = ["a","b","c","d","e"]
>>> "a" in abc
True
>>> "a" not in abc
False
>>> "e" not in abc
False
>>> "f" not in abc
True
>>> str = "Python is easy!"
>>> "y" in str
True
>>> "x" in str
False
```

Repetitions

"*" operator is a kind of abbreviation for an n-times concatenation i.e. str * 4 → str + str + str + str

```
>>> 3 * "xyz-"
'xyz-xyz-xyz-'
>>> "xyz-" * 3
'xyz-xyz-xyz-'
>>> 3 * ["a","b","c"]
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

The augmented assignment for "*" can be used as well:
**s *= n is the same as s = s * n.**

The Pitfalls of Repetitions

In our last example, we applied the repetition operator on strings and flat lists. Let us apply it to nested lists as well:

```
>>> x = ["a","b","c"]
>>> y = [x] * 4
>>> y
[['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b',
'c'], ['a', 'b', 'c']]
>>> y[0][0] = "p"
>>> y
[['p', 'b', 'c'], ['p', 'b', 'c'], ['p', 'b',
'c'], ['p', 'b', 'c']]
```

We have assigned a new value to the first element of the first sublist of y, i.e. y[0][0] and we have "automatically" changed the first elements of all the sublists in y, i.e. y[1][0], y[2][0], y[3][0] .

The reason is that the repetition operator "* 4" creates 4 references to the list x: thus every element of y is changed, if we apply a new value to y[0][0].
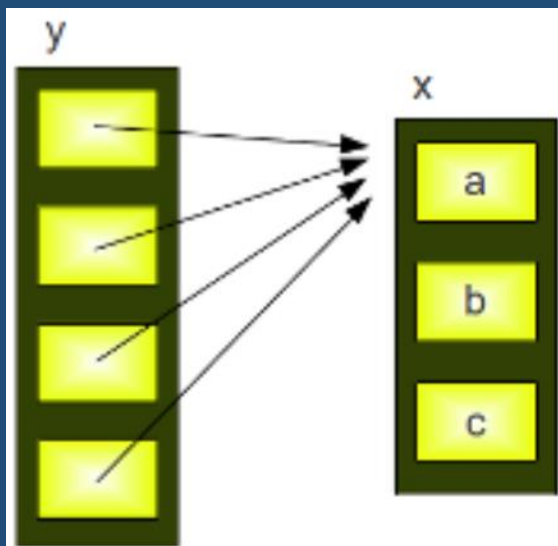
Topic 2e: List Manipulations

Here we will learn how to append and insert objects to lists and also how to delete and remove elements by using 'remove' and 'pop'.

Pop and Append

- **s.append(x) : This method appends an element to the end of the list "s".**

```
>>> lst = [3, 5, 7]          >>> lst = [3, 5, 7]
>>> lst.append(42)           >>> lst = lst.append(42)
>>> lst                       >>> print(lst)
[3, 5, 7, 42]                 None
```

  This returns "None", so it doesn't make sense to reassign the return value

- **s.pop(i) : This method returns the ith element of a list s. The element will be removed from the list as well.**

```
>>> cities = ["Hamburg", "Linz",
"Salzburg", "Vienna"]
>>> cities.pop(0)
'Hamburg'
>>> cities
['Linz', 'Salzburg', 'Vienna']
>>> cities.pop(1)
'Salzburg'
>>> cities
['Linz', 'Vienna']
```

  It raises an IndexError exception if the list is empty or the index is out of range.

- **s.pop()** : The method 'pop' can be called without an argument. In this case, the last element will be returned. So s.pop() is equivalent to s.pop(-1).

```
>>> cities = ["Amsterdam", "The Hague",
"Strasbourg"]
>>> cities.pop()
'Strasbourg'
>>> cities
['Amsterdam', 'The Hague']
```

Extend

How to add more than one element to a list? E.g., you want to add all the elements of one list to another one. Using append, the other list will be appended as a sublist.

```
>>> lst = [42,98,77]
>>> lst2 = [8,69]
>>> lst.append(lst2)
>>> lst
[42, 98, 77, [8, 69]]
```
But what we wanted to accomplish is the following:
```
[42, 98, 77, 8, 69]
```

Python provides the method 'extend'. It extends a list by appending all the elements of an iterable like a list, a tuple or a string to a list:

```
>>> lst = [42,98,77]
>>> lst2 = [8,69]
>>> lst.extend(lst2)
>>> lst
[42, 98, 77, 8, 69]
```

The argument of extend doesn't have to be a list. It can be any iterable. This means that we can use tuples and strings as well:

```
>>> lst = ["a", "b", "c"]
>>> programming_language = "Python"
>>> lst.extend(programming_language)
>>> print(lst)
['a', 'b', 'c', 'P', 'y', 't', 'h', 'o', 'n']
```

Now with a tuple:

```
>>> lst = ["Java", "C", "PHP"]
>>> t = ("C#", "Jython", "Python",
"IronPython")
>>> lst.extend(t)
>>> lst
['Java', 'C', 'PHP', 'C#', 'Jython', 'Python',
'IronPython']
```

Extending and Appending list with "+"

There is an alternative to 'append' and 'extend'. '+' can be used to combine lists.

```
>>> level = ["beginner", "intermediate",
"advanced"]
>>> other_words = ["novice", "expert"]
>>> level + other_words
['beginner', 'intermediate', 'advanced',
'novice', 'expert']
```

But be careful. Never ever do the following:

```
>>> L = [3, 4]
>>> L = L + [42]
>>> L
[3, 4, 42]
```

Even though we get the same result, it is not an alternative to 'append' and 'extend':

```
>>> L = [3, 4]
>>> L.append(42)
>>> L
[3, 4, 42]
>>>
>>>
>>> L = [3, 4]
>>> L.extend([42])
>>> L
[3, 4, 42]
```

The augmented assignment (+=) is an alternative:

```
>>> L = [3, 4]
>>> L += [42]
>>> L
[3, 4, 42]
```

Let us compare in the following example the different approaches and calculate their run times.

```
import time

n= 100000

start_time = time.time()

l = []

for i in range(n):

    l = l + [i * 2]

print(time.time() - start_time)

start_time = time.time()

l = []

for i in range(n):

    l += [i * 2]

print(time.time() - start_time)

start_time = time.time()

l = []

for i in range(n):

    l.append(i * 2)

print(time.time() - start_time)
```

We can see that the "+" operator is too much slower than the append method. If we use the append method, we will simply append a further element to the list in each loop pass. In l = l + [i * 2], the list will be copied in every loop pass, the new element will be added to the copy of the list and result will be reassigned to the variable l. After this the old list will have to be removed by Python, because it is not referenced anymore.

Removing an element with remove

It is possible to remove with the method "remove" a certain value from a list without knowing the position.

```
s.remove(x)
```

This call will remove the first occurrence of x from the list.

```
>>> colours = ["red", "green", "blue",
"green", "yellow"]
>>> colours.remove("green")
>>> colours
['red', 'blue', 'green', 'yellow']
>>> colours.remove("green")
>>> colours
['red', 'blue', 'yellow']
>>> colours.remove("green")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Find the position of an element

The method "index" can be used to find the position of an element within a list:

```
s.index(x[, i[, j]])
```

It returns the first index of the value x. A ValueError will be raised, if the value is not present. If the optional parameter i is given, the search will start at the index i. If j is also given, the search will stop at position j.

```
>>> colours = ["red", "green", "blue", "green", "yellow"]
>>> colours.index("green")
1
>>> colours.index("green", 2)
3
>>> colours.index("green", 3,4)
3
>>> colours.index("black")
Traceback (most recent call last):
  File "", line 1, in
ValueError: 'black' is not in list
```

Insert

We need also a way to add elements to arbitrary positions inside of a list. This can be done with the method "insert":

```
s.insert(index, object)
```

An object "object" will be included in the list "s" and placed before the element s[index]

```
>>> lst = ["German is spoken", "in Germany,",
"Austria", "Switzerland"]
>>> lst.insert(3, "and")
>>> lst
['German is spoken', 'in Germany,', 'Austria',
'and', 'Switzerland']
```

The functionality of the method "append" can be simulated with insert in the following way:

```
>>> abc = ["a","b","c"]
>>> abc.insert(len(abc),"d")
>>> abc
['a', 'b', 'c', 'd']
```

Topic 2f: Shallow and Deep Copy

Here, Y points to the same memory location as X. When new value assigned to y, gets a separate memory location.

```
>>> x = 3
>>> y = x
>>> print(id(x), id(y))
9251744 9251744
>>> y = 4
>>> print(id(x), id(y))
9251744 9251776
>>> print(x,y)
3 4
```

Let us have a look at few crucial problems, which can occur when copying mutable objects, lists or dictionaries.

Copying a list

```
>>> colours1 = ["red", "blue"]
>>> colours2 = colours1
>>> print(colours1)
['red', 'blue']
>>> print(colours2)
['red', 'blue']
>>> print(id(colours1),id(colours2))
43444416 43444416
>>> colours2 = ["rouge", "vert"]
>>> print(colours1)
['red', 'blue']
>>> print(colours2)
['rouge', 'vert']
>>> print(id(colours1),id(colours2))
43444416 43444200
```
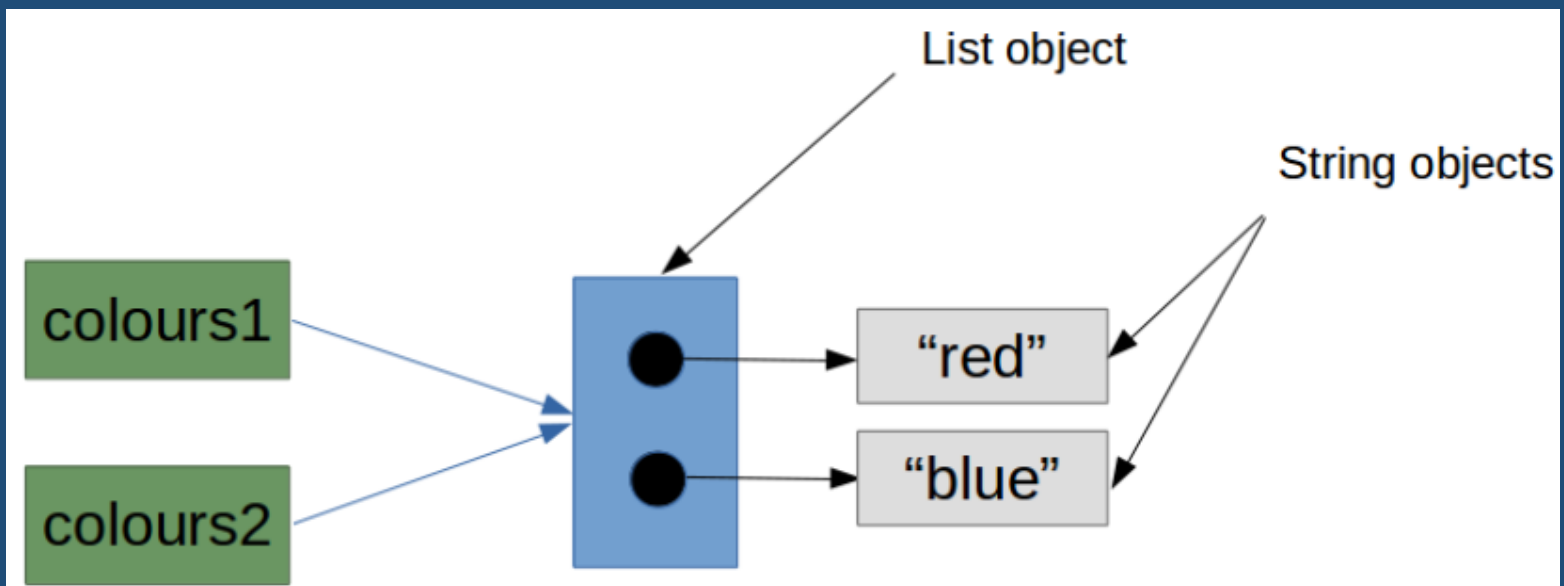
A simple list is assigned to colours1. This list is a so-called "shallow list", because it does not have a nested structure, i.e. no sublists are contained in the list. The id() function shows us that both variables point to the same list object, i.e. they share this object.

```
>>> colours1 = ["red", "blue"]
>>> colours2 = colours1
>>> print(id(colours1),id(colours2))
14603760 14603760
>>> colours2[1] = "green"
>>> print(id(colours1),id(colours2))
14603760 14603760
>>> print(colours1)
['red', 'green']
>>> print(colours2)
['red', 'green']
```
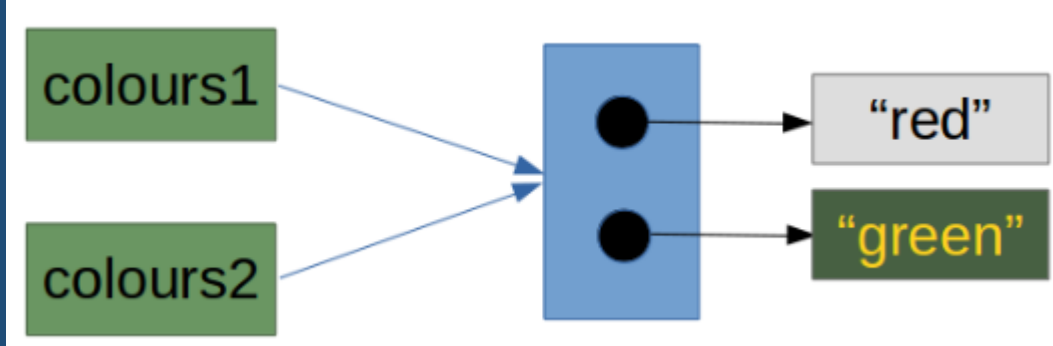
We assigned a new value to the second element of colours2, i.e. the element with the index 1. The list of colours1 has been "automatically" changed as well. This happened because we have two names for the same list.

Copying with the Slice Operator

It's possible to completely copy shallow list structures with the slice operator without having any side effects

```
>>> list1 = ['a','b','c','d']
>>> list2 = list1[:]
>>> list2[1] = 'x'
>>> print(list2)
['a', 'x', 'c', 'd']
>>> print(list1)
['a', 'b', 'c', 'd']
```
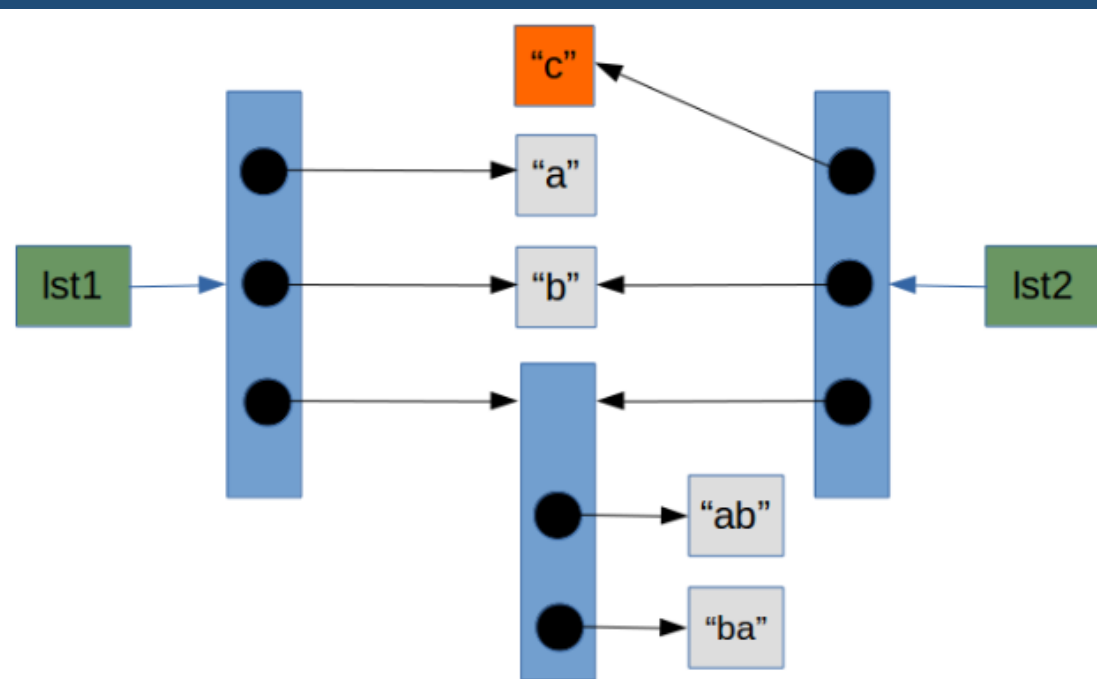
When a list contains sublists, another difficulty: The sublists are not copied but only the references to sublists. We create a shallow copy with the slicing operator.

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
```
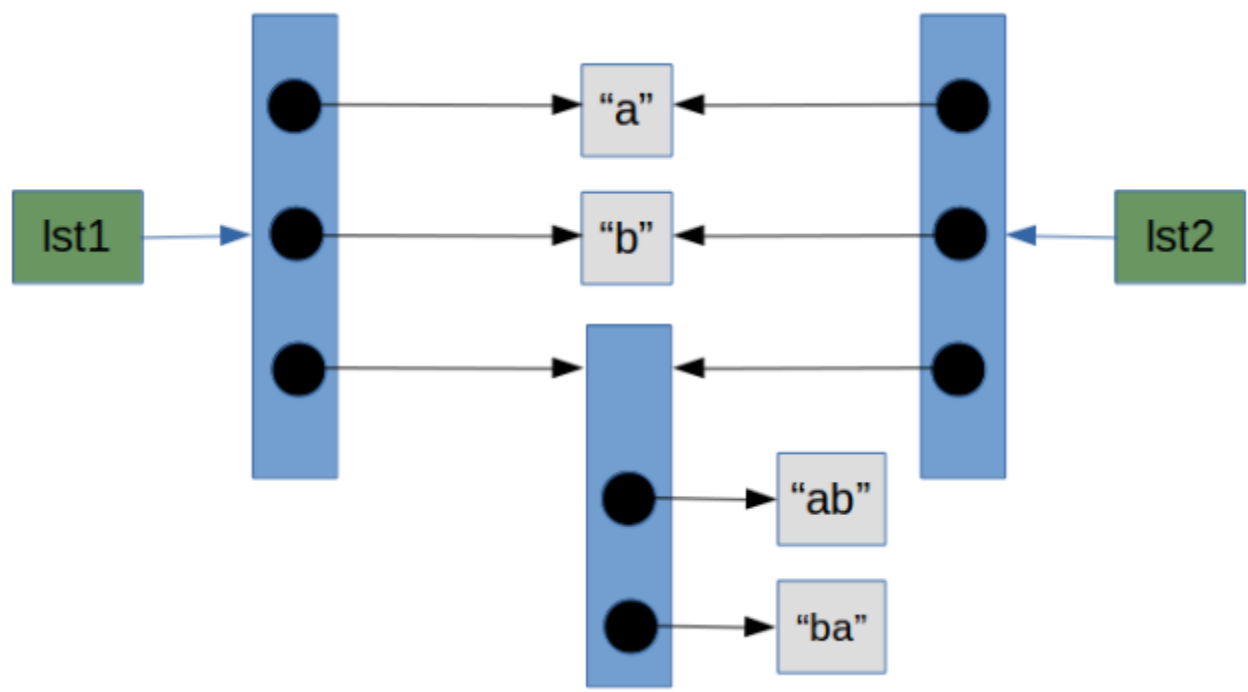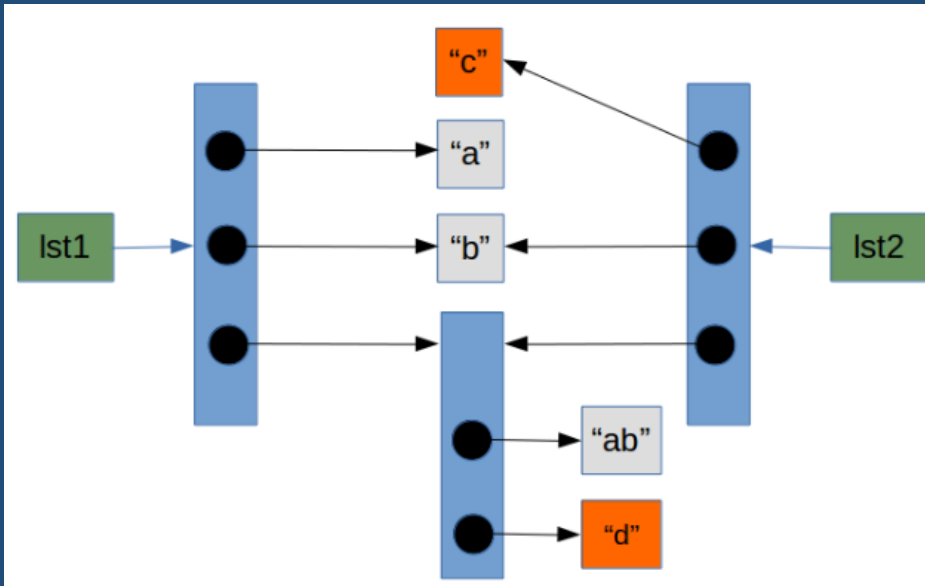
If you assign a new value to the 0th or the 1st index of one of the two lists, there will be no side effect.

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
>>> lst2[0] = 'c'
>>> print(lst1)
['a', 'b', ['ab', 'ba']]
>>> print(lst2)
['c', 'b', ['ab', 'ba']]
```

Problem arise if you change one of the elements of sublist

```
>>> lst2[2][1] = 'd'
>>> print(lst1)
['a', 'b', ['ab', 'd']]
>>> print(lst2)
['c', 'b', ['ab', 'd']]
```


Using Method deepcopy from Module copy

It allows a complete or deep copy of an arbitrary list, i.e. shallow and other lists.

```
>>> from copy import deepcopy
>>>
>>> lst1 = ['a','b',['ab','ba']]
>>>
>>> lst2 = deepcopy(lst1)
>>>
>>> lst1
['a', 'b', ['ab', 'ba']]
>>> lst2
['a', 'b', ['ab', 'ba']]
```

```
>>> id(lst1)
139716507600200
>>> id(lst2)
139716507600904
>>> id(lst1[0])
139716538182096
>>> id(lst2[0])
139716538182096
>>> id(lst2[2])
139716507602632
>>> id(lst1[2])
139716507615880
```
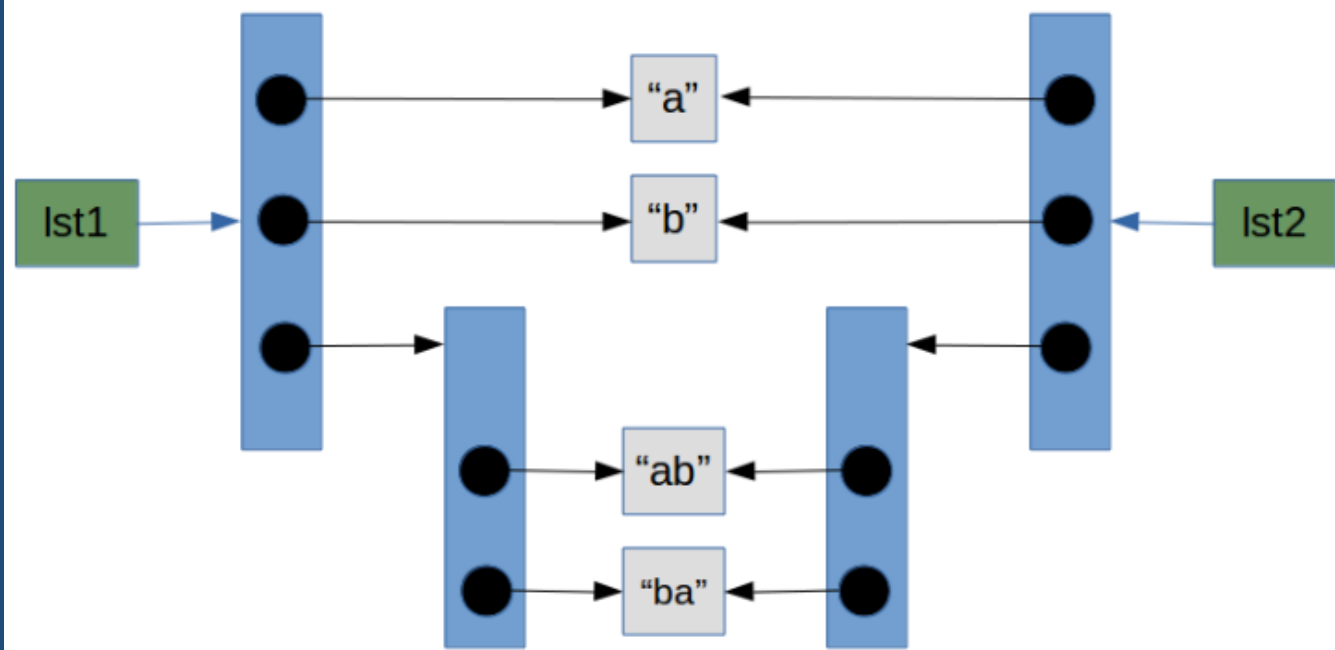
```
>>> lst2[2][1] = "d"
>>> lst2[0] = "c"
>>> print(lst1)
['a', 'b', ['ab', 'ba']]
>>> print(lst2)
['c', 'b', ['ab', 'd']]
```

Now the data structure looks like this:
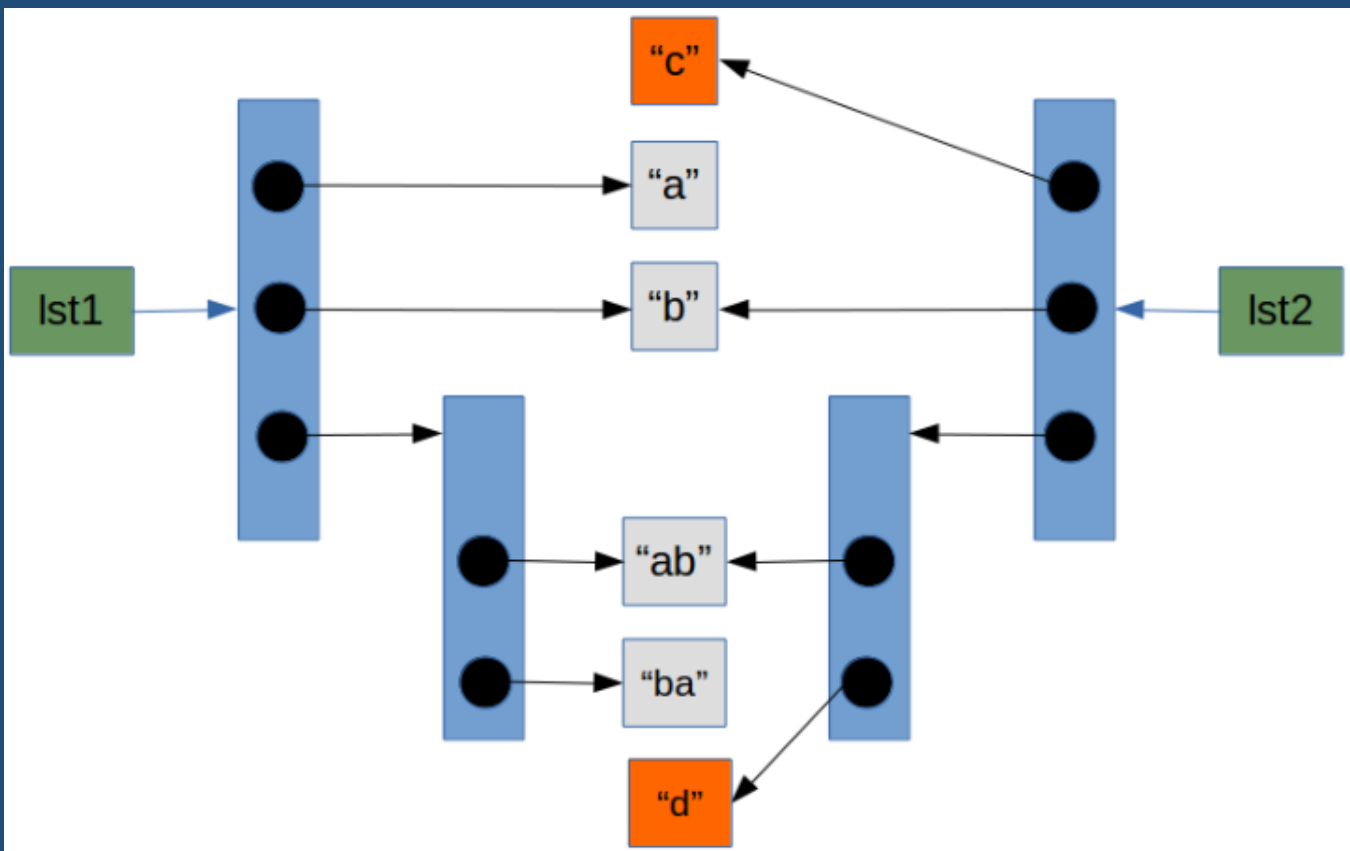
Topic 3a: Dictionaries

Introduction

Difference between lists and dictionaries? A list is an ordered sequence of objects, whereas dictionaries are unordered sets. The main difference is : items in dictionaries are accessed via keys and not their position.

Examples:

```
>>> city_population = {"New York
City":8550405, "Los Angeles":3971883,
"Toronto":2731571, "Chicago":2720546,
"Houston":2296224, "Montreal":1704694,
"Calgary":1239220, "Vancouver":631486,
"Boston":667137}
```

If we want to get the population of one of those cities, all we have to do is to use the name of the city as an index:

```
>>> city_population["New York City"]
8550405
>>> city_population["Toronto"]
2731571
>>> city_population["Boston"]
667137
```

If we try to access a key which is not contained in the dictionary, KeyError is raised

```
>>> city["Detroit"]
Traceback (most recent call last):
  File "<stdin>", line 1, in
KeyError: 'Detroit'
```

There is no ordering in dictionaries. That's why the output of the dictionary, doesn't reflect the "original ordering"

```
>>> city
{'Toronto': 2615060, 'Ottawa': 883391, 'Los
Angeles': 3792621, 'Chicago': 2695598, 'New
York City': 8175133, 'Boston': 62600,
'Washington': 632323, 'Montreal': 11854442}
```

Therefore it is neither possible to access an element of the dictionary by a number, like we did with lists:

```
>>> city[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in
KeyError: 0
```

It's very easy to add another entry to existing dictionary

```
>>> city["Halifax"] = 390096
>>> city
{'Toronto': 2615060, 'Ottawa': 883391, 'Los
Angeles': 3792621, 'Chicago': 2695598, 'New
York City': 8175133, 'Halifax': 390096,
'Boston': 62600, 'Washington': 632323,
'Montreal': 11854442}
```

So, it's possible to create a dictionary incrementally by starting with an empty dictionary. How to define an empty dictionary?

```
>>> city = {}
>>> city
{}
```

Now, let us create English to French dictionary:

1. **English to German dictionary**

```python
en_de = {"red" : "rot", "green" : "grün",
"blue" : "blau", "yellow":"gelb"}
print(en_de)
print(en_de["red"])
```

2. **Next, German to French dictionary**

```python
de_fr = {"rot" : "rouge", "grün" : "vert",
"blau" : "bleu", "gelb":"jaune"}
```

3. **Now it's possible to translate English to French, even though we don't have English-French-dictionary**

```python
print("The French word for red is: " +
de_fr[en_de["red"]])
```

We can use arbitrary types as values in a dictionary, but there is a restriction for the keys. Only immutable data types can be used as keys, i.e. no lists or dictionaries can be used. If you use a mutable data type as a key, you get an error message:

```python
>>> dic = { [1,2,3]:"abc"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

Tuple as keys are okay

```python
>>> dic = { (1,2,3):"abc", 3.1415:"abc"}
>>> dic
{3.1415: 'abc', (1, 2, 3): 'abc'}
```

We can also create a dictionary of dictionaries

```python
en_de = {"red" : "rot", "green" : "grün",
"blue" : "blau", "yellow":"gelb"}
de_fr = {"rot" : "rouge", "grün" : "vert",
"blau" : "bleu", "gelb":"jaune"}

dictionaries = {"en_de" : en_de, "de_fr" :
de_fr }
print(dictionaries["de_fr"]["blau"])
```

Operators on Dictionaries

Operator	Explanation
len(d)	returns the number of stored entries, i.e. the number of (key,value) pairs.
del d[k]	deletes the key k together with his value
k in d	True, if a key k exists in the dictionary d
k not in d	True, if a key k doesn't exist in the dictionary d

pop()

D.pop(k) removes the key k with its value from the dictionary D and returns this key's value, i.e. D[k].

```python
>>> en_de = {"Austria":"Vienna",
"Switzerland":"Bern", "Germany":"Berlin",
"Netherlands":"Amsterdam"}
>>> capitals = {"Austria":"Vienna",
"Germany":"Berlin", "Netherlands":"Amsterdam"}
>>> capital = capitals.pop("Austria")
>>> print(capital)
Vienna
>>> print(capitals)
{'Netherlands': 'Amsterdam', 'Germany':
'Berlin'}
```

If the key is not found a KeyError is raised:

```
>>> capital = capitals.pop("Switzerland")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Switzerland'
```

To prevent these errors, there is an elegant way. The method pop() has an optional second parameter, which can be used as a default value:

```
>>> capital = capitals.pop("Switzerland",
"Bern")
>>> print(capital)
Bern
>>> capital = capitals.pop("France", "Paris")
>>> print(capital)
Paris
>>> capital = capitals.pop("Germany",
"München")
>>> print(capital)
Berlin
```

popitem()

popitem() is a method of dict, which doesn't take any parameter and removes and returns an arbitrary (key,value) pair as a 2-tuple. If popitem() is applied on an empty dictionary, a KeyError will be raised.

```
>>> capitals = {"Springfield":"Illinois",
"Augusta":"Maine", "Boston": "Massachusetts",
"Lansing":"Michigan", "Albany":"New York",
"Olympia":"Washington", "Toronto":"Ontario"}
>>> (city, state) = capitals.popitem()
>>> (city, state)
('Olympia', 'Washington')
>>> print(capitals.popitem())
('Albany', 'New York')
>>> print(capitals.popitem())
('Boston', 'Massachusetts')
>>> print(capitals.popitem())
('Lansing', 'Michigan')
>>> print(capitals.popitem())
('Toronto', 'Ontario')
```

Accessing Non Existing Keys

If you try to access a key which doesn't exist, you will get an error message:

```
>>> locations = {"Toronto" : "Ontario",
"Vancouver":"British Columbia"}
>>> locations["Ottawa"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Ottawa'
```

You can prevent this by using the "in" operator:

```
>>> if "Ottawa" in locations:
print(locations["Ottawa"])
...
>>> if "Toronto" in locations:
print(locations["Toronto"])
...
Ontario
```

Another method to access the values via the key consists in using the get() method. get() is not raising an error, if an index doesn't exist. In this case it will return None. It's also possible to set a default value, which will be returned, if an index doesn't exit:

```
>>> proj_language = {"proj1":"Python",
"proj2":"Perl", "proj3":"Java"}
>>> proj_language["proj1"]
'Python'
>>> proj_language["proj4"]
Traceback (most recent call last):
  File "<stdin>", line 1, in
KeyError: 'proj4'
>>> proj_language.get("proj2")
'Perl'
>>> proj_language.get("proj4")
>>> print(proj_language.get("proj4"))
None
>>> # setting a default value:
>>> proj_language.get("proj4", "Python")
'Python'
```

Important Methods

A dictionary can be copied with the method copy():

```
>>> w = words.copy()
>>> words["cat"]="chat"
>>> print(w)
{'house': 'Haus', 'cat': 'Katze'}
>>> print(words)
{'house': 'Haus', 'cat': 'chat'}
```

This copy is a shallow and not a deep copy.

The content of a dictionary can be cleared with the method clear(). The dictionary is not deleted, but set to an empty dictionary:

```
>>> w.clear()
>>> print(w)
{}
```

Update: Merging Dictionaries

update() method merges keys and values of one dictionary into another, overwriting values of same key:

```
>>> knowledge = {"Frank": {"Perl"}, "Monica":
{"C","C++"}}
>>> knowledge2 = {"Guido":{"Python"}, "Frank":
{"Perl", "Python"}}
>>> knowledge.update(knowledge2)
>>> knowledge
{'Frank': {'Python', 'Perl'}, 'Guido':
{'Python'}, 'Monica': {'C', 'C++'}}
```

Iterating over a Dictionary

No method is needed to iterate over keys of a dictionary:

```
>>> d = {"a":123, "b":34, "c":304, "d":99}
>>> for key in d:
...       print(key)
...
b
c
a
d
```

```
>>> for key in d.keys():    >>> for value in d.values():
...     print(key)          ...     print(value)
...                         ...
b                           34
c                           304
a                           123
d                           99
```

The above-right loop is equivalent to the following:

```
for key in d:
        print(d[key])
```
but less efficient.