



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 Froszensets
- c. Tuple
- d. Input from keyboard

4. Conditional Statements

- a. If-else Block
- b. Loops, While Loop
- c. For Loop
- d. Iterators and Iterables

5. Output Formatting

- a. **Output with Print**
- b. **String Modulo**
- c. **Format Method**

6. Functions

- a. **Introduction to Functions**
- b. **Recursion and Recursive Functions**
- c. **Parameter Passing in Functions**
- d. **Namespaces**
- e. **Global and Local Variables**
- f. **Decorators**

7. File Operations

- a. **Read and Write Files**
- b. **Modules**
- c. **Packages**

8. Advanced Python Concepts

- a. **Regular Expressions**
- b. **Lambda, Filter & Map Operators**
- c. **List Comprehension**
- d. **Iterators and Generators**
- e. **Exception Handling**
- f. **Tests, DocTests & UnitTests**

9. Object Oriented Programming

- a. **Introduction**
- b. **Class and Instance Attributes**
- c. **Properties v/s getters and setters**
- d. **Inheritance**
- e. **Classes and Class Creation**
- f. **Abstract Classes**

10. Powerful Utilities

- a. **Multithreading**
- b. **Logging Facility**
- c. **Argument Parser**
- d. **Invoking Sub process**
- e. **Lincache, fnmatch etc.**
- f. **Live Project**

11. Domain Specific

- a. **Networking Automations, or**
- b. **Django Web Framework, or**
- c. **Numerical Data Analytics**
 - i. **Numpy**
 - ii. **Pandas, or**
- d. **Data Science Analytics**
 - i. **Web Scrapping & Crawling**
 - ii. **Machine Learning, or**
- e. **Automation Tooling**

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

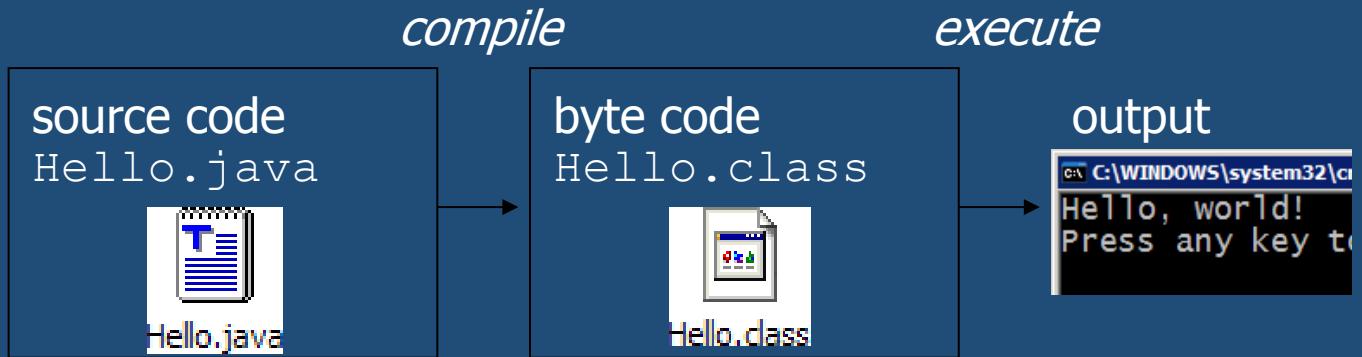
- Multi-purpose (Web, Data Science, Automation, etc.)
- Procedural and Object-Oriented
- Interpreted
- Interactive Shell
- Strongly typed and Dynamically typed
- Focus on readability and productivity
- Cross Platform (CPython, Jython, IronPython, PyPy)

Few Big Organizations using Python

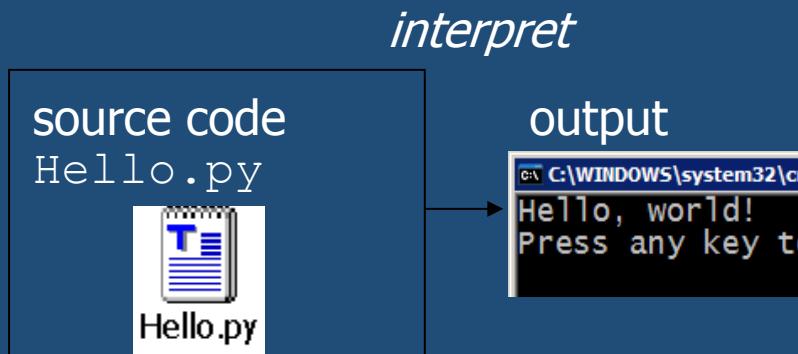
- Google
- NASA
- Netflix
- Dropbox
- YouTube
- New York University
- General Electric
- Juniper Networks
- Lego
- Nasdaq
- And the list goes on...

Compiled v/s Interpreted Language

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



- Python is instead directly *interpreted* into machine instructions.



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  
646.1893969999999  
>>>
```

- 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

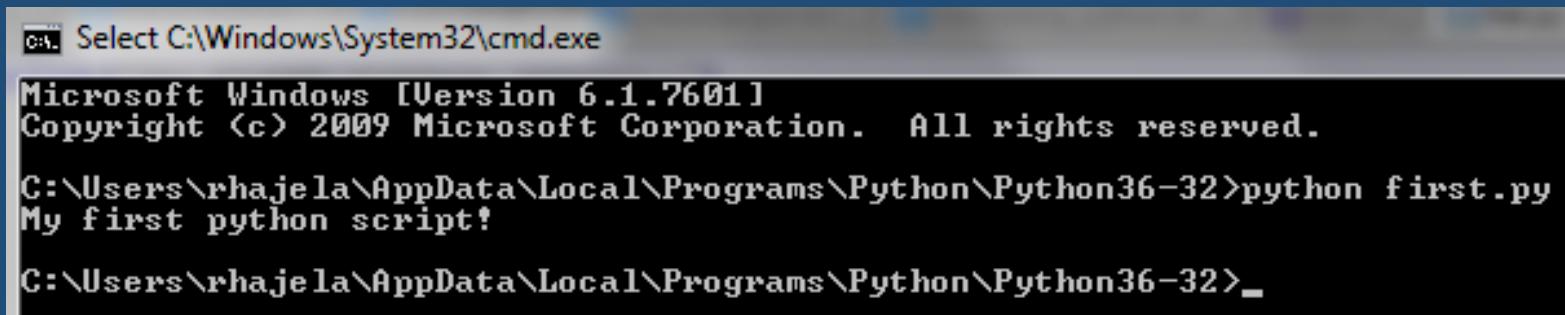
```
>>> maximal = 124
>>> width = 94
>>> print(maximal - width)
30
>>>
```

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

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

Topic 1d: Execute python script

First Python Script



A screenshot of a Windows Command Prompt window. The title bar says "Select C:\Windows\System32\cmd.exe". The window displays the following text:
Microsoft Windows [Version 6.1.7601]
Copyright © 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>_

```
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_simple_script.py')  
>>>
```

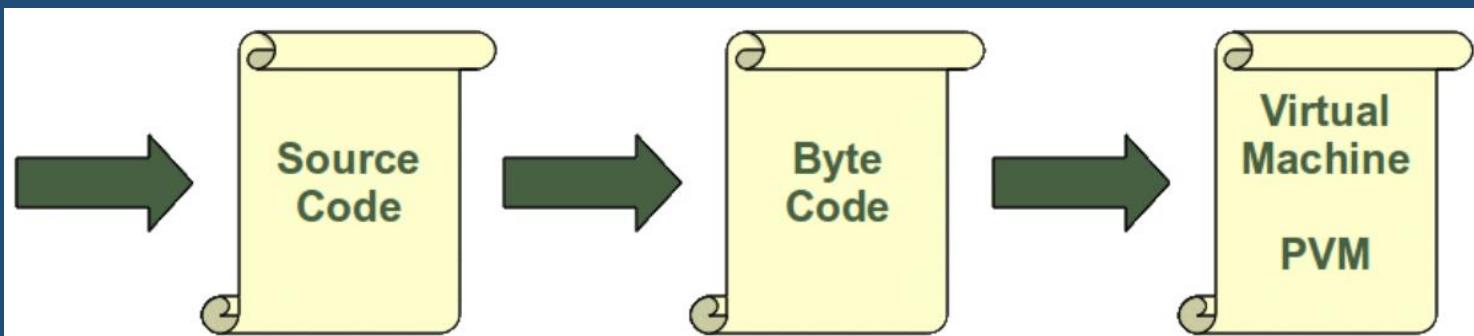
Or

```
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.



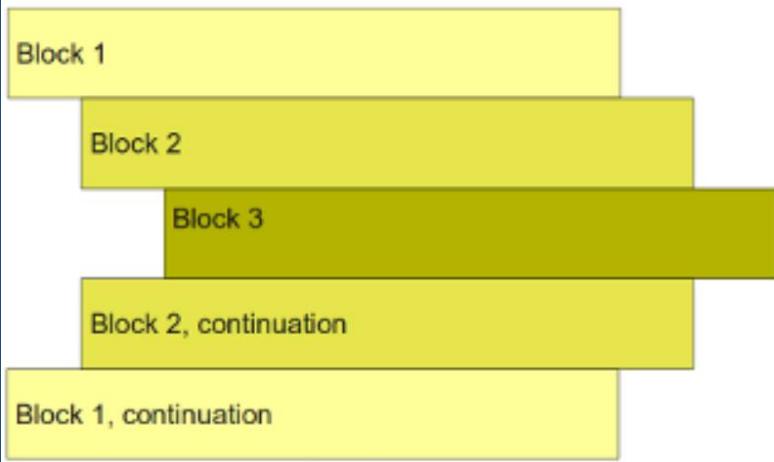
Topic 2a: Indentation

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:

```
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 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.

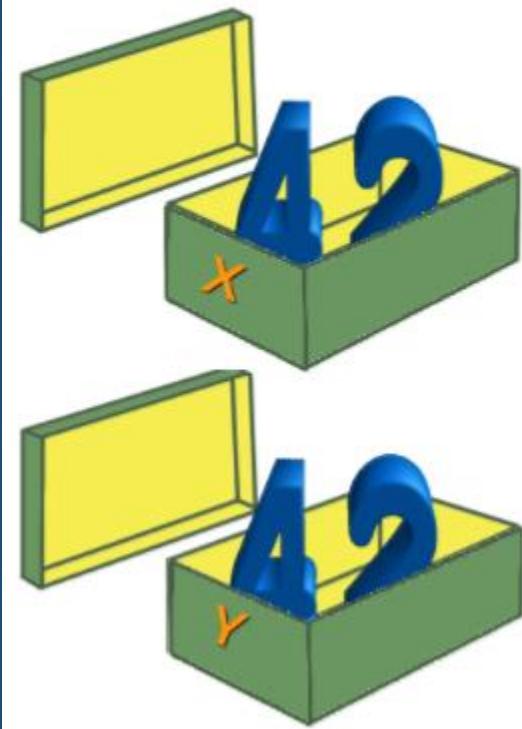
But, Python provides power data types like lists and associative arrays, called dict in Python, as a genuine part of the language.

General Concept of Variables in other languages

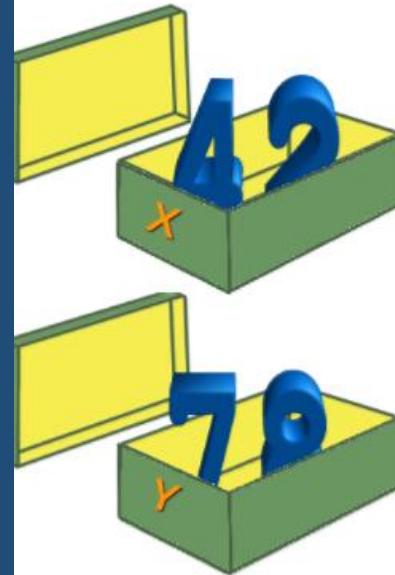
- What is a variable? As the name implies, a variable is something, which can change.
- A variable is a way of referring to a memory location used by a computer program.
- In many programming languages, a variable is a symbolic name for this physical location.
- This memory location contains values,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

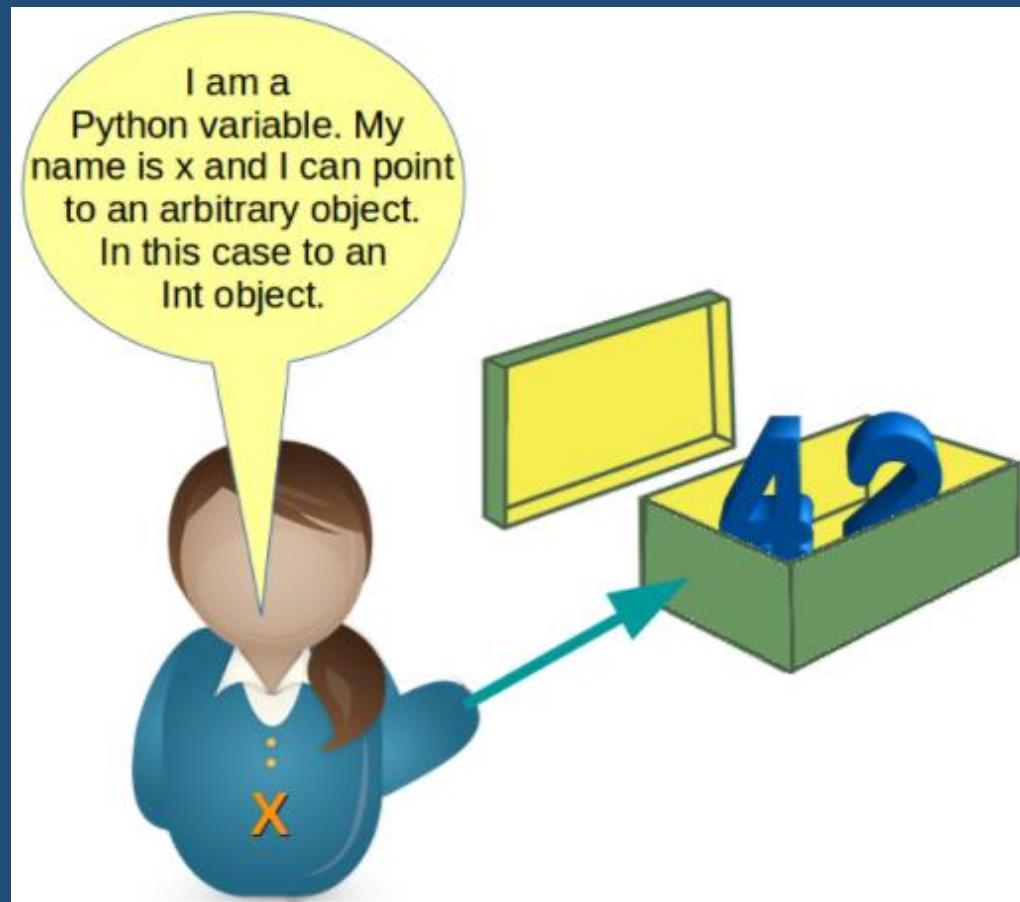
```
i = 42
```

- 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.

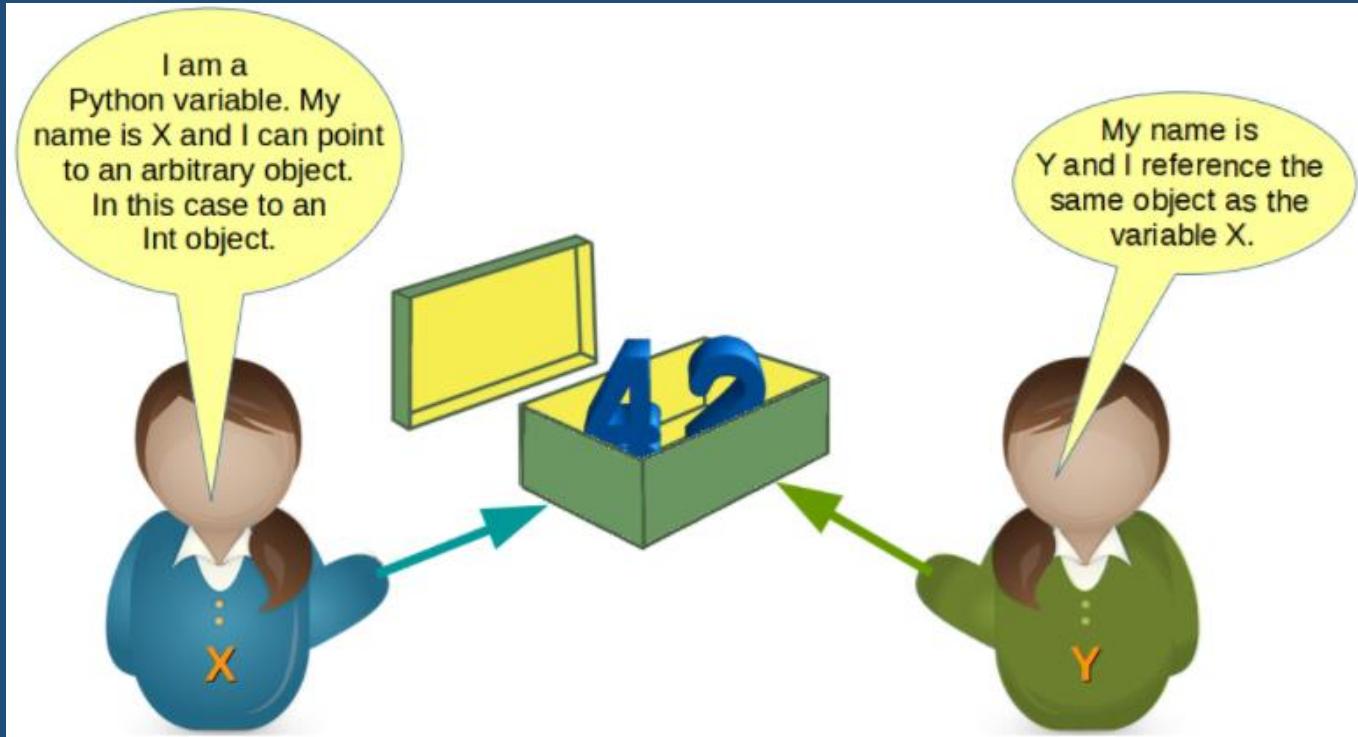


As variables are pointing to objects and objects can be of arbitrary data type, variables can't have types associated with them. This is a huge difference to C, C++ or Java, where a variable is associated with a fixed data type.

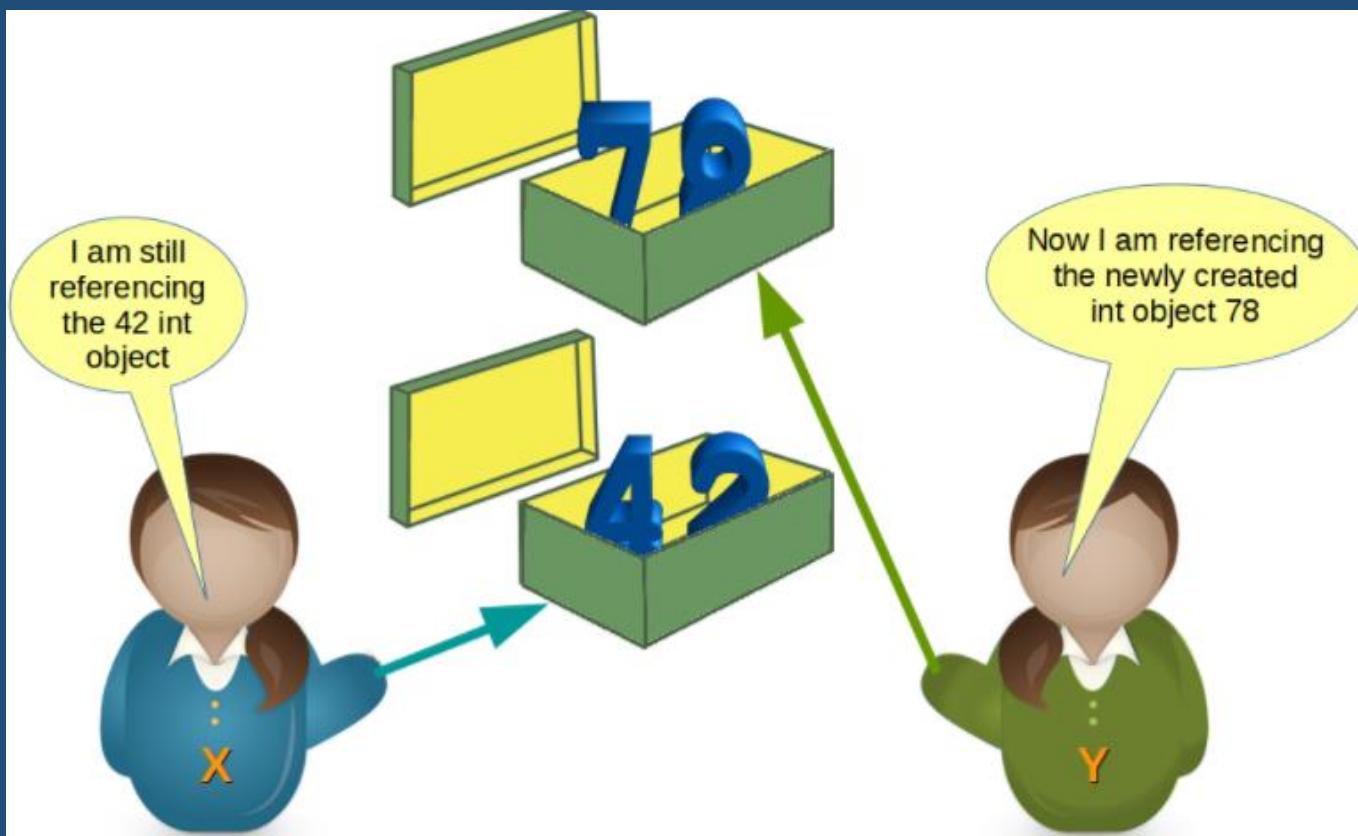
```
>>> x = 42
>>> print(x)
42
>>> x = "Now x references a
string"
>>> print(x)
Now x references a string
```

Examples

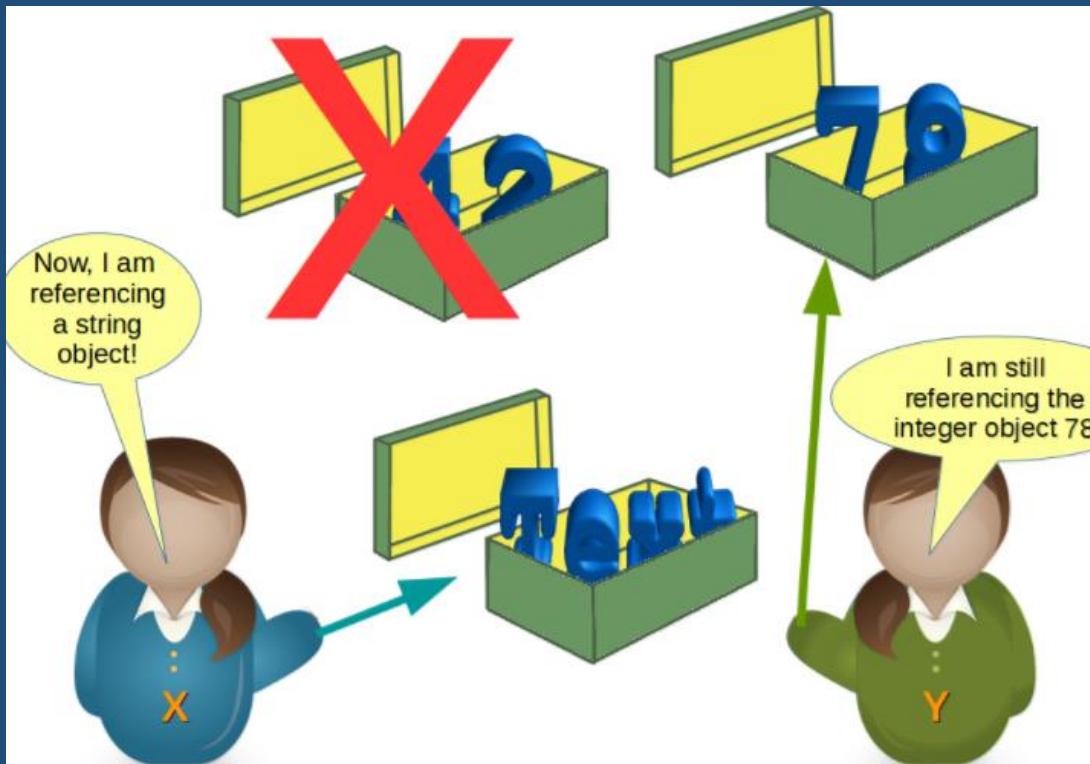
```
>>> x = 42  
>>> y = x
```



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
'0b100001'
>>> 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
005317148968569797875581092352
>>>
```

- ***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  
3.333333333333335  
>>> 10.0 / 3.0  
3.333333333333335  
>>> 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!'
```

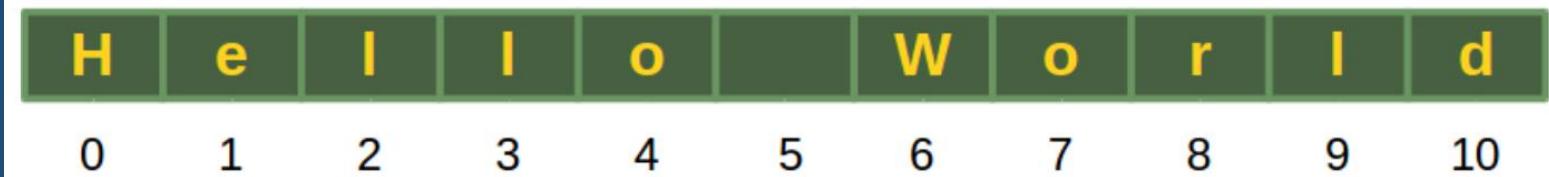
```
>>> s3 = "It doesn't matter!"
```

```
>>> 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



```
>>> 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"

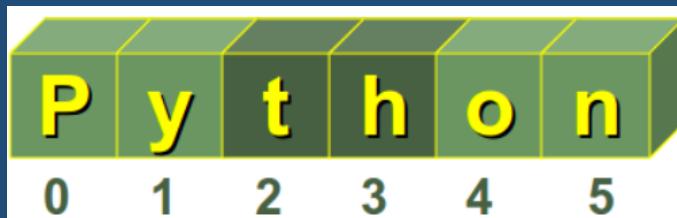
- **Repetition:**

"*_*" * 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! <pre>Import timeit print(timeit.timeit ('for x in range(1, 100):y =100 // x', number=100000))</pre>	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

```
>>> print([["London", "England", 7556900], ["Paris", "France", 2193031]])
```

[["London", "England", 7556900],
 ["Paris", "France", 2193031]]

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  
'TPoyrtohnnotno ciosu rtshees lianr gTeosrto  
nCtiot yb yi nB oCdaennasdeao'
```

```
>>> s  
'TPoyrtohnnotno 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. $s += t \rightarrow s = s + t$

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.**

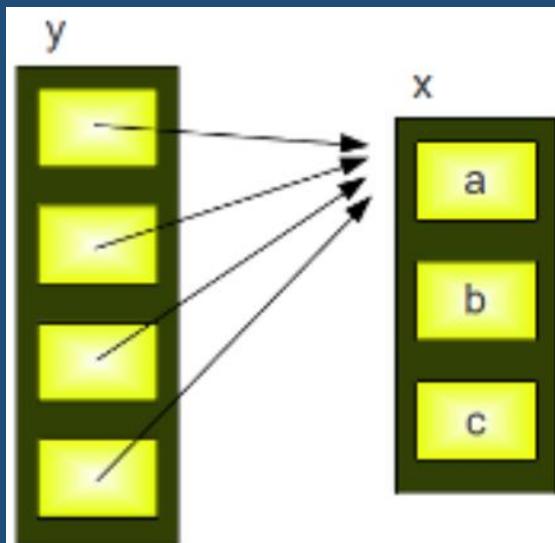
The Pitfalls of Repetitions

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.append(42)
>>> lst
[3, 5, 7, 42]
```

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

This returns “None”, so it doesn’t make sense to reassign the return value

- `s.pop(i)` : This method returns the `i`th 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

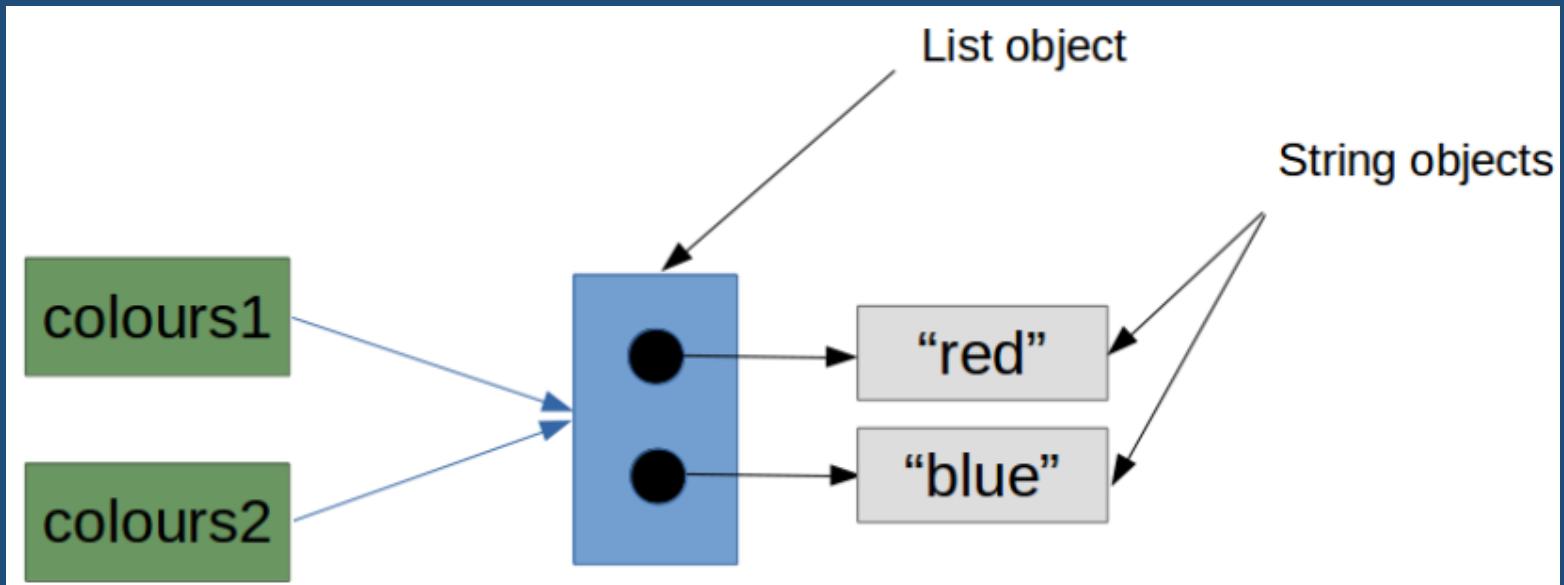
Here, Y points to the same memory location as X. When new value assigned to y, gets a separate memory location.

```
>>> 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.

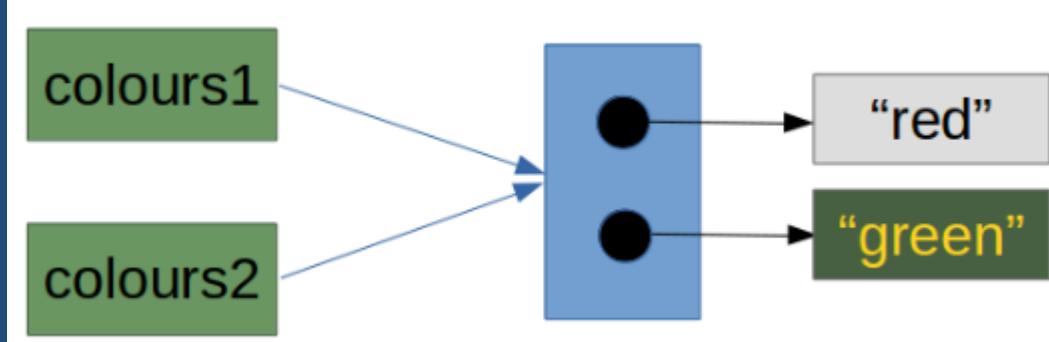
Copying a list

```
>>> 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']

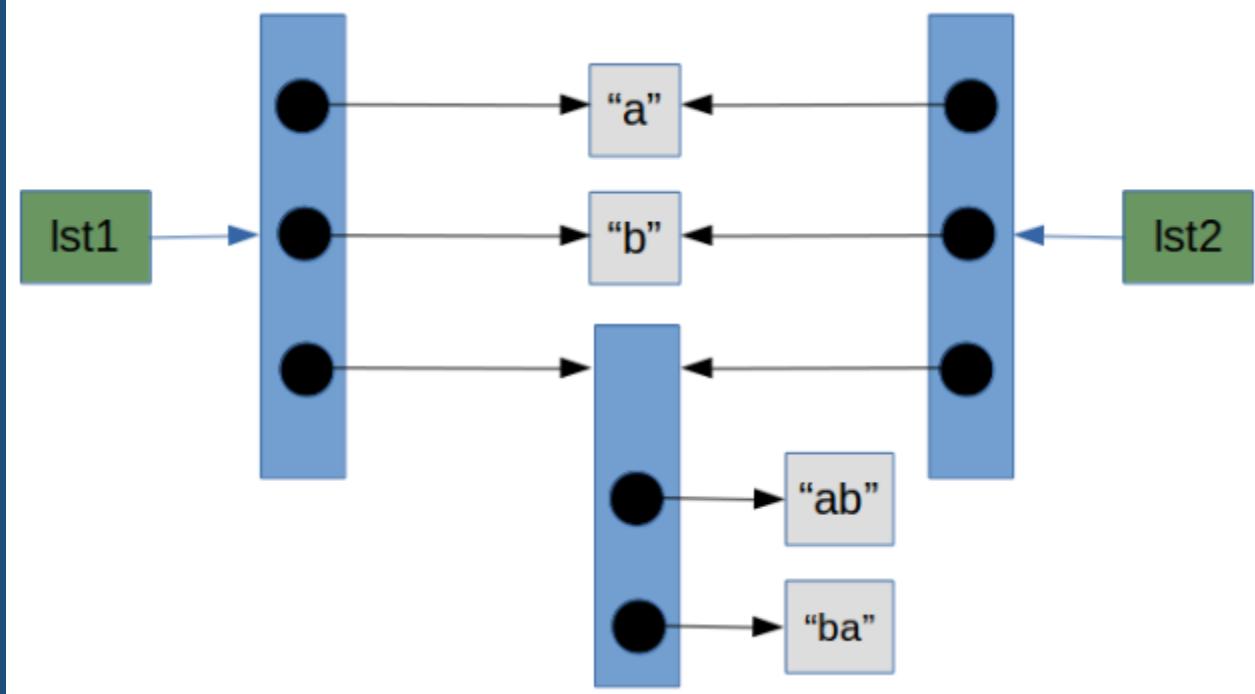
```

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.

```

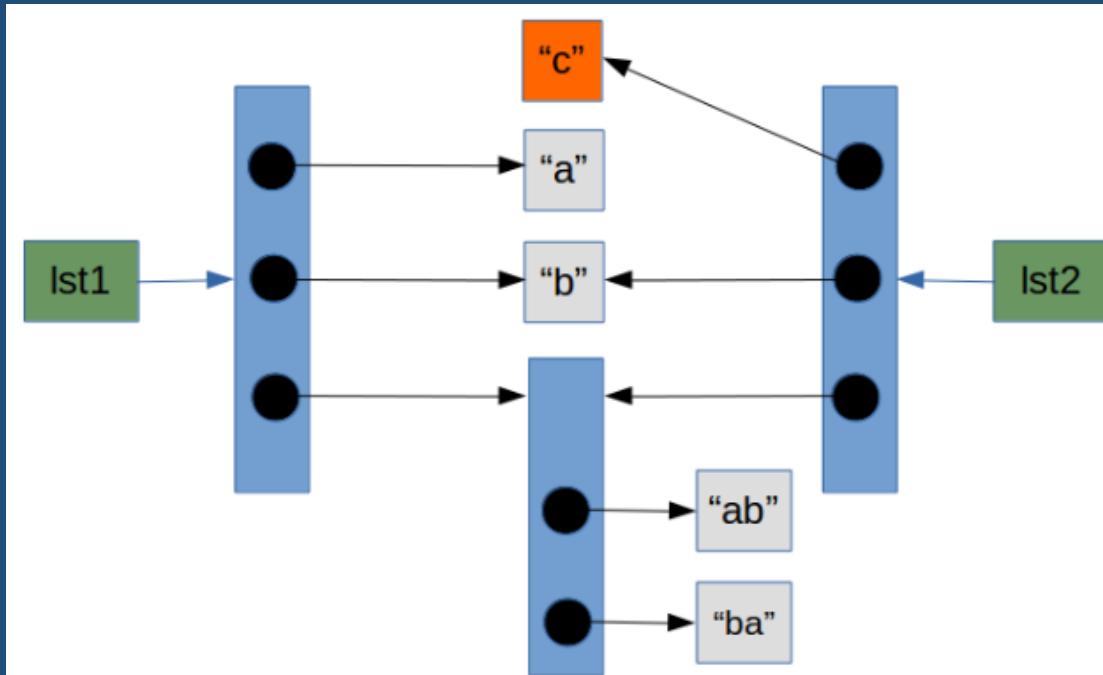
>>> lst1 = ['a', 'b', ['ab', 'ba']]
>>> lst2 = lst1[:]

```



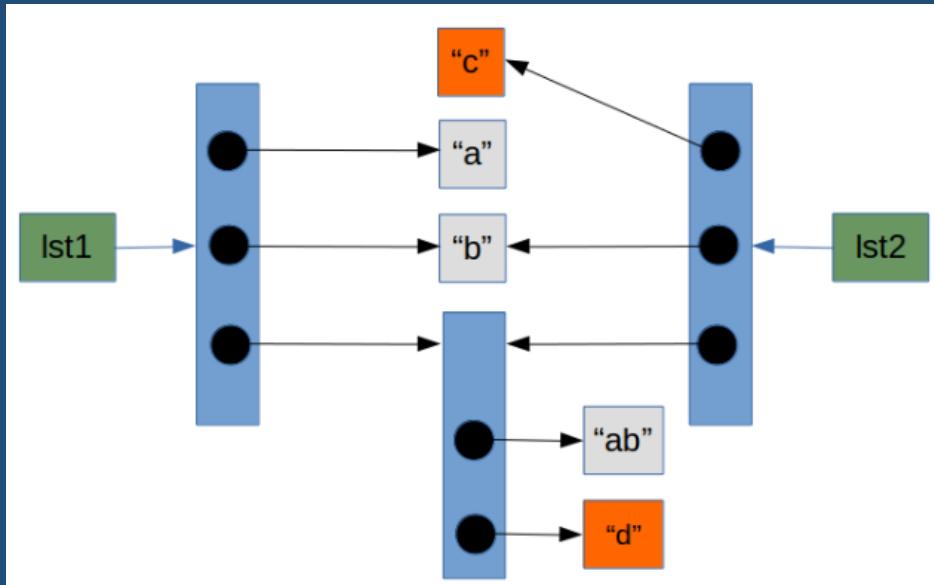
If you assign a new value to the 0th or the 1st index of one of the two lists, there will be no side effect.

```
>>> 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']]
```

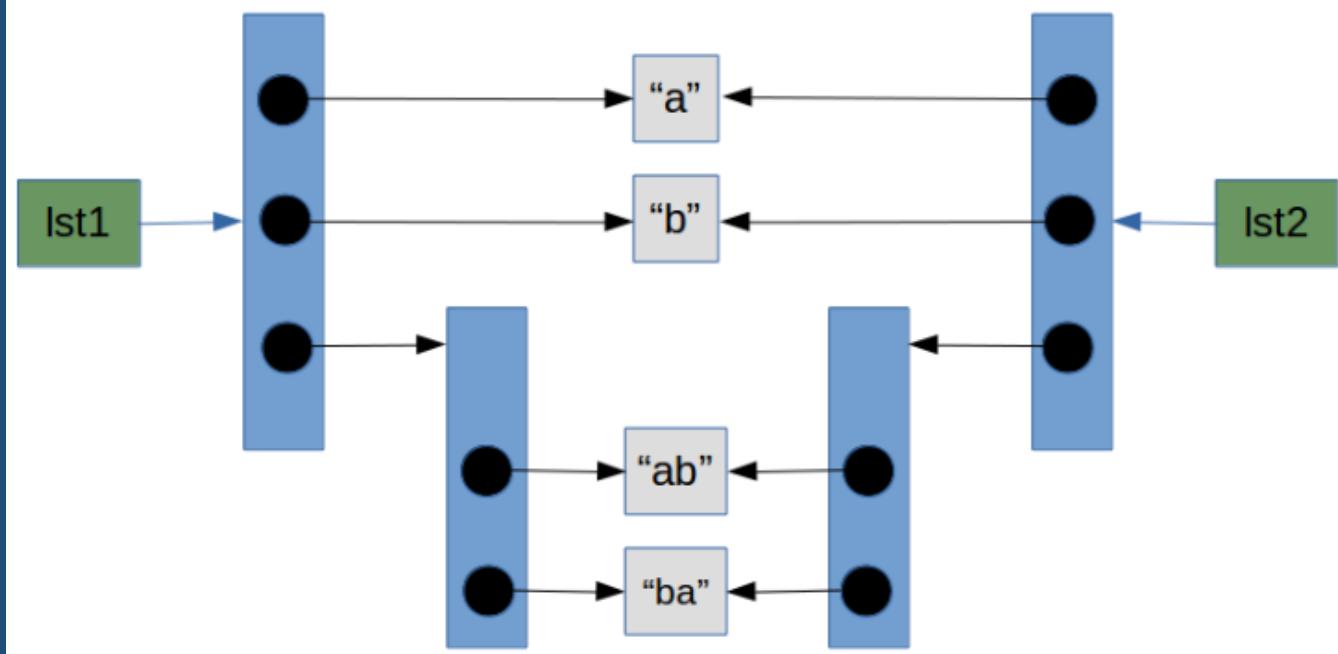


Using Method deepcopy from Module copy

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

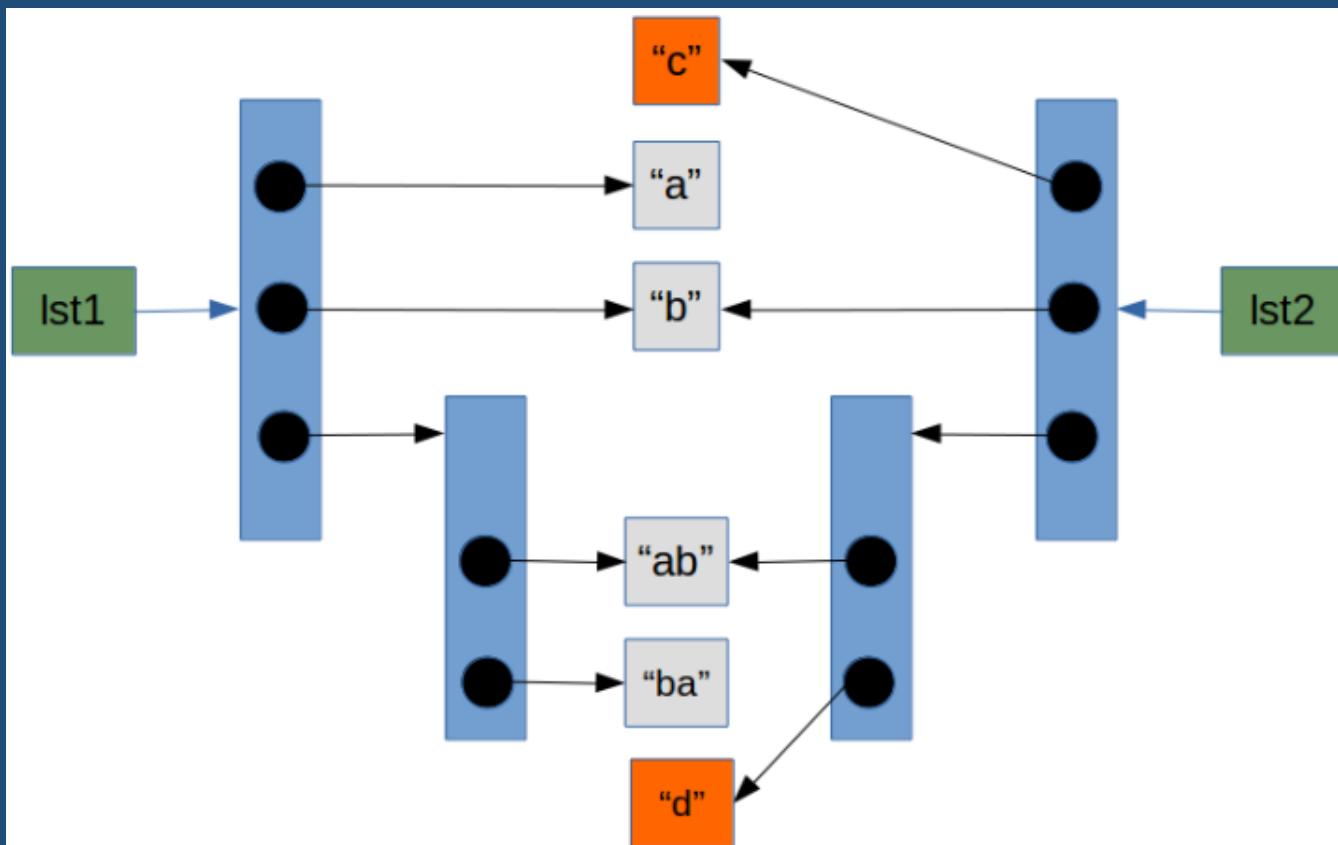
```
>>> from copy import deepcopy  
>>>  
>>> lst1 = ['a', 'b', ['ab', 'ba']]  
>>>  
>>> lst2 = deepcopy(lst1)  
>>>  
>>> lst1  
['a', 'b', ['ab', 'ba']]  
>>> lst2  
['a', 'b', ['ab', 'ba']]
```

```
>>> 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

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

2. Next, German to French dictionary

```
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

```
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:

```
>>> 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

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

We can also create a dictionary of dictionaries

```
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].

```
>>> 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():
...     print(key)
...
b
c
a
d
```

```
>>> for value in d.values():
...     print(value)
...
34
304
123
99
```

The above-right loop is equivalent to the following:

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

Lists from Dictionaries

keys() creates a list, which consists of the keys of the dictionary. **values()** produces a list consisting of the values. **items()** creates a list consisting of 2-tuples of (key,value)-pairs:

```
>>> w = {"house": "Haus", "cat": "", "red": "rot"}
>>> items_view = w.items()
>>> items = list(items_view)
>>> items
[('house', 'Haus'), ('cat', ''), ('red', 'rot')]
>>>
>>> keys_view = w.keys()
>>> keys = list(keys_view)
>>> keys
['house', 'cat', 'red']
>>>
>>> values_view = w.values()
>>> values = list(values_view)
>>> values
['Haus', '', 'rot']
>>> values_view
dict_values(['Haus', '', 'rot'])
>>> items_view
dict_items([('house', 'Haus'), ('cat', ''), ('red', 'rot')])
>>> keys_view
dict_keys(['house', 'cat', 'red'])
```

Turn Lists into Dictionaries

We have two lists, one containing the dishes and the other one the corresponding countries:

```
>>> dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
>>> countries = ["Italy", "Germany", "Spain", "USA"]
```

Now we will create a dictionary, which assigns a dish to a country. For this purpose we need the function `zip()`. The result is a list iterator. This means that we have to wrap a `list()` casting function around the `zip` call to get a list:

```
>>> country_specialities_iterator = zip(countries, dishes)
>>> country_specialities_iterator
<zip object at 0x7fa5f7cad408>
>>> country_specialities = list(country_specialities_iterator)
>>> print(country_specialities)
[('Italy', 'pizza'), ('Germany', 'sauerkraut'), ('Spain', 'paella'),
 ('USA', 'hamburger')]
```

```
>>> country_specialities_dict =
dict(country_specialities)
>>> print(country_specialities_dict)
{'USA': 'hamburger', 'Germany': 'sauerkraut',
 'Spain': 'paella', 'Italy': 'pizza'}
```

But it's inefficient, we created a list of 2-tuples to turn list into dict. This can be done directly by applying `dict` to `zip`:

```
>>> dishes = ["pizza", "sauerkraut", "paella",
"hamburger"]
>>> countries = ["Italy", "Germany", "Spain",
"USA"]
>>> dict(zip(countries, dishes))
{'USA': 'hamburger', 'Germany': 'sauerkraut',
 'Spain': 'paella', 'Italy': 'pizza'}
```

Topic 3b: Set and Frozensets

Introduction

A set contains an unordered collection of unique and immutable objects. To create a set, use the built-in set function with a sequence or another iterable object:

```
>>> x = set("A Python Tutorial")
>>> x
{'A', ' ', 'i', 'h', 'l', 'o', 'n', 'P', 'r',
'u', 't', 'a', 'y', 'T'}
>>> type(x)
<class 'set'>
```

```
>>> x = set(["Perl", "Python", "Java"])
>>> x
{'Python', 'Java', 'Perl'}
```

```
>>> cities = set(("Paris", "Lyon",
"London", "Berlin", "Paris", "Birmingham"))
>>> cities
{'Paris', 'Birmingham', 'Lyon', 'London',
'Berlin'}
```

Sets doesn't allow mutable objects.

```
>>> cities = set(((("Python", "Perl"), ("Paris",
"Berlin", "London"))))
>>> cities = set(([["Python", "Perl"], ["Paris",
"Berlin", "London"]]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

To define a set, we can only use curly braces also:

```
>>> adjectives =  
{"cheap", "expensive", "inexpensive", "economical"}  
>>> adjectives  
{'inexpensive', 'cheap', 'expensive',  
'economical'}
```

Frozensests

Though sets can't contain mutable objects, sets are mutable:

```
>>> cities = set(["Frankfurt",  
"Basel", "Freiburg"])  
>>> cities.add("Strasbourg")  
>>> cities  
{'Freiburg', 'Basel', 'Frankfurt',  
'Strasbourg'}
```

Frozensests are like sets except that they cannot be changed, i.e. they are immutable:

```
>>> cities = frozenset(["Frankfurt",  
"Basel", "Freiburg"])  
>>> cities.add("Strasbourg")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'frozenset' object has no  
attribute 'add'
```

Set Operations

- *add(element)*

A method which adds an element, which has to be immutable, to a set.

```
>>> colours = {"red", "green"}  
>>> colours.add("yellow")  
>>> colours  
{'green', 'yellow', 'red'}  
>>> colours.add(["black", "white"])  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unhashable type: 'list'
```

Of course, an element will only be added, if it is not already contained in the set. If it is already contained, the method call has no effect.

- *clear()*

All elements will be removed from a set.

```
>>> cities = {"Stuttgart", "Konstanz",  
"Freiburg"}  
>>> cities.clear()  
>>> cities  
set()
```

- *copy()*

Creates a shallow copy, which is returned.

```
>>> more_cities =  
{"Winterthur", "Schaffhausen", "St.  
Gallen"}  
>>> cities_backup = more_cities.copy()  
>>> more_cities.clear()  
>>> cities_backup  
{'St. Gallen', 'Winterthur',  
'Schaffhausen'}
```

- ***difference()***

This method returns the difference of two or more sets as a new set.

```
>>> x = {"a", "b", "c", "d", "e"}  
>>> y = {"b", "c"}  
>>> z = {"c", "d"}  
>>> x.difference(y)  
{'a', 'e', 'd'}  
>>> x.difference(y).difference(z)  
{'a', 'e'}
```

```
>>> x - y  
{'a', 'e', 'd'}  
>>> x - y - z  
{'a', 'e'}
```

- ***difference_update()***

This method removes all elements of another set from this set. *x.difference_update(y)* is same as "*x = x - y*"

```
>>> x = {"a", "b", "c", "d", "e"}  
>>> y = {"b", "c"}  
>>> x.difference_update(y)  
>>>  
>>> x = {"a", "b", "c", "d", "e"}  
>>> y = {"b", "c"}  
>>> x = x - y  
>>> x  
{'a', 'e', 'd'}
```

- *discard(el)*

An element el will be removed from the set, if it is contained in the set. If el is not a member of the set, nothing will be done.

```
>>> x = {"a", "b", "c", "d", "e"}  
>>> x.discard("a")  
>>> x  
{'c', 'b', 'e', 'd'}  
>>> x.discard("z")  
>>> x  
{'c', 'b', 'e', 'd'}
```

- *remove(el)*

works like `discard()`, but if el is not a member of the set, a `KeyError` will be raised.

```
>>> x = {"a", "b", "c", "d", "e"}  
>>> x.remove("a")  
>>> x  
{'c', 'b', 'e', 'd'}  
>>> x.remove("z")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'z'
```

- *union(s)*

This method returns the union of two sets as a new set, i.e. all elements that are in either set.

```
>>> x = {"a", "b", "c", "d", "e"}  
>>> y = {"c", "d", "e", "f", "g"}  
>>> x.union(y)  
{'d', 'a', 'g', 'c', 'f', 'b', 'e'}
```

This can be abbreviated with the pipe operator "|":

```
>>> x = {"a", "b", "c", "d", "e"}  
>>> y = {"c", "d", "e", "f", "g"}  
>>> x | y  
{'d', 'a', 'g', 'c', 'f', 'b', 'e'}
```

- *intersection(s)*

Returns the intersection of the instance set and the set s as a new set. In other words: A set with all the elements which are contained in both sets is returned.

```
>>> x = {"a", "b", "c", "d", "e"}  
>>> y = {"c", "d", "e", "f", "g"}  
>>> x.intersection(y)  
{'c', 'e', 'd'}
```

```
>>> x = {"a", "b", "c", "d", "e"}  
>>> y = {"c", "d", "e", "f", "g"}  
>>> x & y  
{'c', 'e', 'd'}
```

- *pop()*

pop() removes and returns an arbitrary set element.

The method raises a **KeyError** if the set is empty

```
>>> x = {"a", "b", "c", "d", "e"}  
>>> x.pop()  
'a'  
>>> x.pop()  
'c'
```

Topic 3c: Tuples

Introduction

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, tuples cannot be changed unlike lists and tuples use parentheses instead of square brackets.

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup1 = ();
```

To write a tuple containing a single value you have to include a comma – `tup1 = (50,);`

Tuples Operations

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print "tup1[0]: ", tup1[0];
print "tup2[1:5]: ", tup2[1:5];
```

Python Expression	Results	Description
<code>len((1, 2, 3))</code>	3	Length
<code>(1, 2, 3) + (4, 5, 6)</code>	<code>(1, 2, 3, 4, 5, 6)</code>	Concatenation
<code>('Hi!',) * 4</code>	<code>('Hi!', 'Hi!', 'Hi!', 'Hi!')</code>	Repetition
<code>3 in (1, 2, 3)</code>	True	Membership
<code>for x in (1, 2, 3): print x,</code>	1 2 3	Iteration

cmp(tuple1, tuple2)

Compares elements of both tuples.

len(tuple)

Gives the total length of the tuple.

max(tuple)

Returns item from the tuple with max value.

min(tuple)

Returns item from the tuple with min value.

tuple(seq)

Converts a list into tuple.

Topic 3d: Input from Keyboard

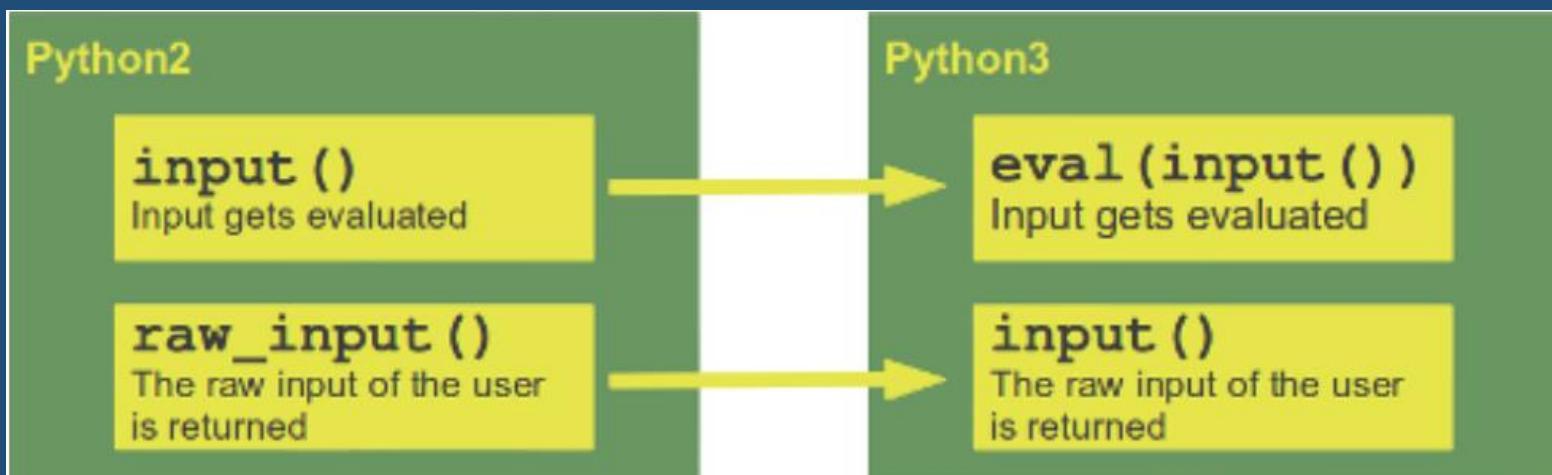
Introduction

For accepting input from user, Python provides the function `input()`. If the `input` function is called, the program flow will be stopped until the user has given an input and has ended the input with the return key.

```
name = input("What's your name? ")
print("Nice to meet you " + name + "!")
age = input("Your age? ")
print("So, you are already " + age + " years
old, " + name + "!")
```

The input of the user will be returned as a string without any changes. If this raw input has to be transformed into another data type needed by the algorithm, we can use either a casting function or the `eval` function.

Differences to Python2



Topic 4a: If-else Block

Introduction

Imagine a situation of events / decisions like this:

If it rains tomorrow, I will do the following:

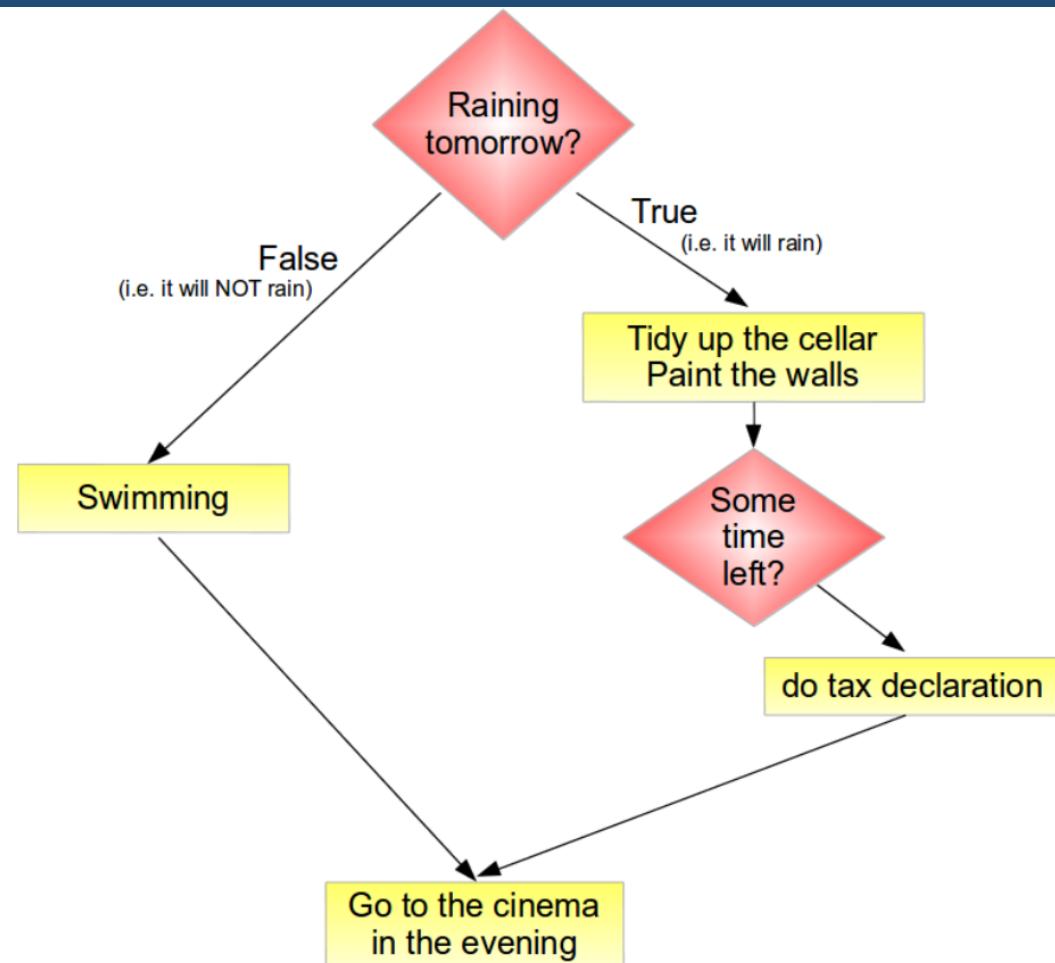
- tidy up the cellar
- paint the walls
- If there is some time left, I will
 - do my tax declaration

Otherwise, I will do the following:

- go swimming

go to the cinema with my wife in the evening

Such a work flow is often formulated in the programming environment as a so-called flow chart or programming flowchart:



How would other programming language pseudo code be:

```
if (raining_tomorrow) {  
    tidy_up_the_cellar();  
    paint_the_walls();  
    if (time_left)  
        do_taxes();  
} else  
    enjoy_swimming();  
    go_cinema();
```

The code may be all right for the interpreter or the compiler of the language, but it can be written in a way, which is badly structured for humans.

In Python, blocks are created with indentations. We could say "What you see is what you get!"

```
if raining_tomorrow:  
    tidy_up_the_cellar()  
    paint_the_walls()  
    if time_left:  
        do_taxes()  
else:  
    enjoy_swimming()  
go_cinema()
```

Conditional Statements in Python

if-statements are used to change the flow of control in a Python program. This makes it possible to decide at run-time whether or not to run one block of code or another.

```
if condition:  
    statement  
    statement  
    # ... some more indented statements if  
necessarily
```

The indented block of code is executed only if the condition "condition" is evaluated to True, meaning that it is logically true.

Examples:

```
person = input("Nationality? ")  
if person == "french":  
    print("Préférez-vous parler français?")
```

```
person = input("Nationality? ")  
if person == "french" or person == "French":  
    print("Préférez-vous parler français?")
```

```
person = input("Nationality? ")  
if person == "french" or person == "French" :  
    print("Préférez-vous parler français?")  
if person == "italian" or person == "Italian"  
:  
    print("Preferisci parlare italiano?")
```

```
person = input("Nationality? ")
if person == "french" or person == "French" :
    print("Préférez-vous parler français?")
elif person == "italian" or person ==
"Italian":
    print("Preferisci parlare italiano?")
else:
    print("You are neither Italian nor
French,")
    print("so we have to speak English with
each other.")
```

So, the general form of if statement in Python looks like:

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
...
elif another_condition:
    another_statement_block
else:
    else_block
```

The ternary if

```
max = a if (a > b) else b
```

Expression is "max shall be a if a is greater than b else b".

```
max = (a if (a > b) else b) * 2.45 - 4
```

**Here if is used in an expression, which can be used within
another expression**

Examples:

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

if x > y and x > z:
    maximum = x
elif y > x and y > z:
    maximum = y
else:
    maximum = z

print("The maximal value is: " + str(maximum))
```

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

if x > y:
    if x > z:
        maximum = x
    else:
        maximum = z
else:
    if y > z:
        maximum = y
    else:
        maximum = z

print("The maximal value is: " + str(maximum))
```

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

maximum = max((x, y, z))

print("The maximal value is: " + str(maximum))
```

Topic 4b: Loops

Introduction

Python supplies two kinds of loops:

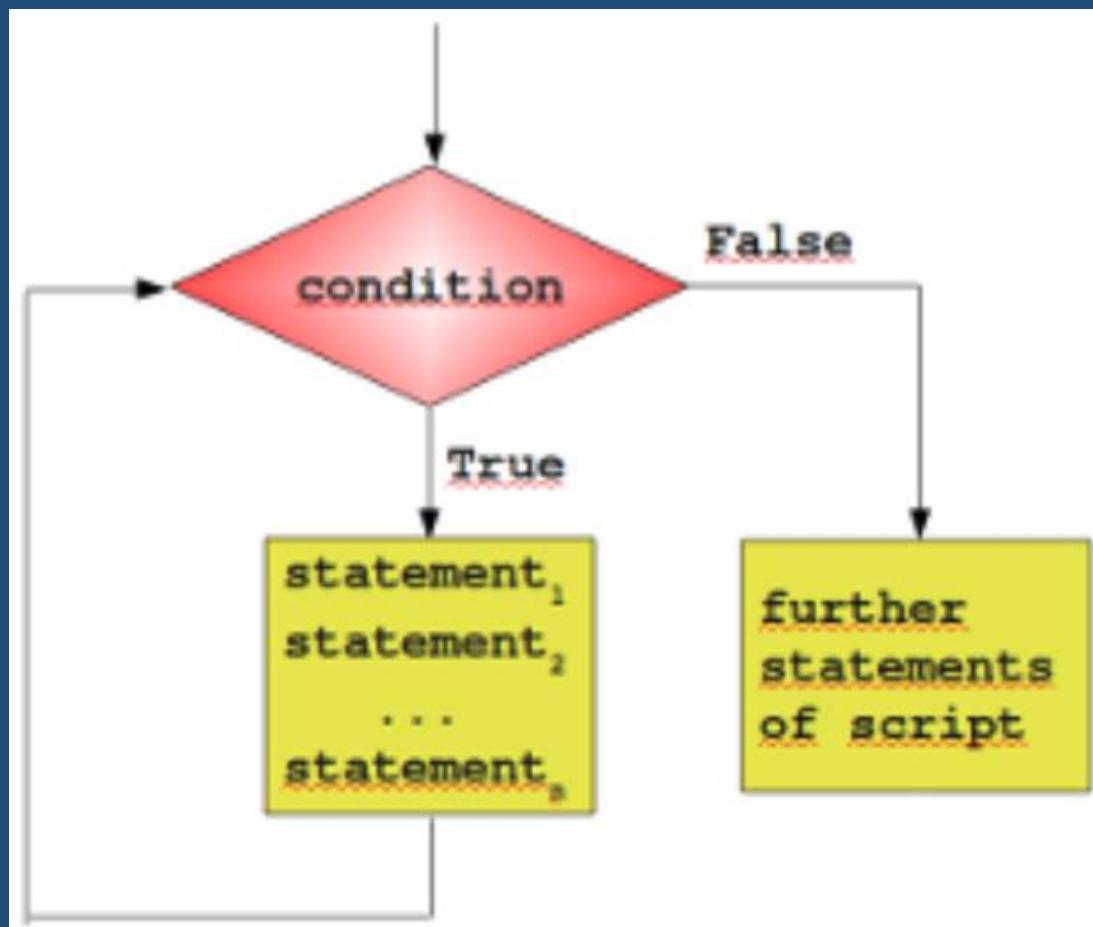
1. the while loop (condition-controlled loop)

A loop will be repeated until a given condition changes, i.e. changes from True to False or from False to True, depending on the kind of loop.

2. the for loop (collection-controlled loop)

This is a special construct which allow looping through the elements of a "collection", which can be a dictionary, list or other sequence.

While Loop



Examples:

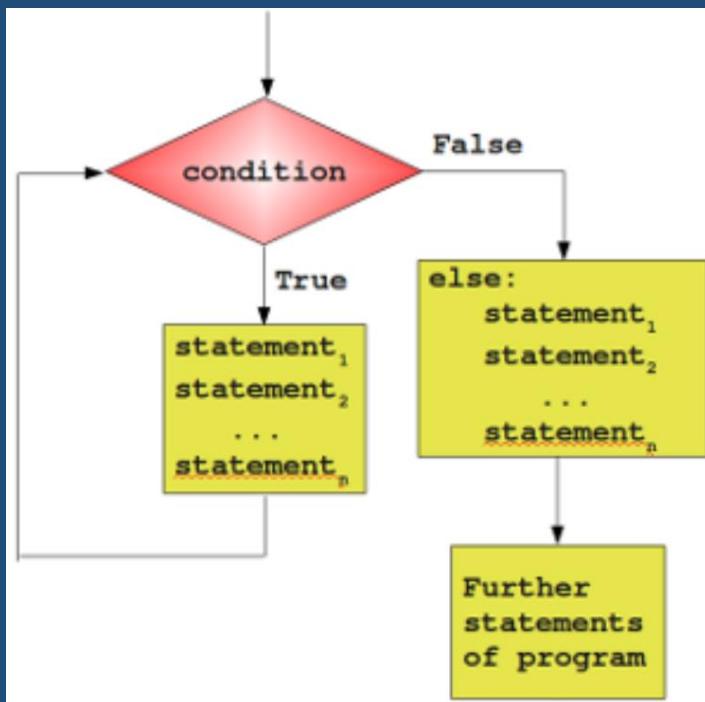
```
n = 100  
  
s = 0  
counter = 1  
while counter <= n:  
    s = s + counter  
    counter += 1
```

```
print("Sum of 1 until %d: %d" % (n,s))
```

```
import sys  
  
text = ""  
while 1:  
    c = sys.stdin.read(1)  
    text = text + c  
    if c == '\n':  
        break  
  
print("Input: %s" % text)
```

While with Else Part

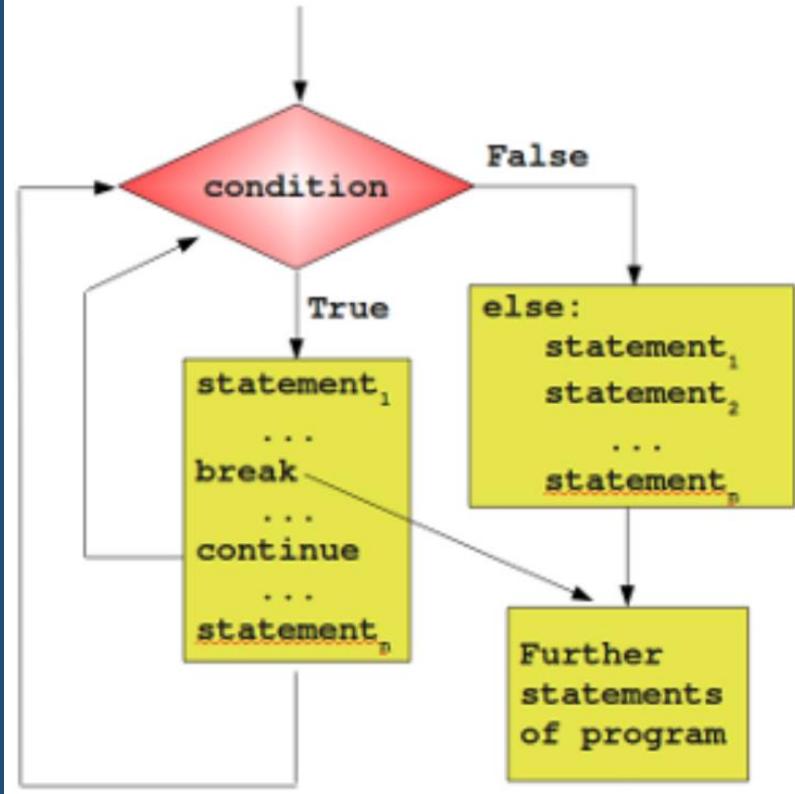
Similar to the if statement, the while loop of Python has also an optional else part.



```
while condition:  
    statement_1  
    ...  
    statement_n  
  
else:  
    statement_1  
    ...  
    statement_n
```

Premature Termination of a While Loop

So far, a while loop only ends, if the condition in the loop head is fulfilled. With the help of a break statement a while loop can be left prematurely, i.e. the loop will be immediately left.



```

import random
n = 20
to_be_guessed = int(n *
random.random()) + 1
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess > 0:
        if guess > to_be_guessed:
            print("Number too large")
        elif guess < to_be_guessed:
            print("Number too small")
        else:
            print("Sorry that you're giving up!")
            break
    else:
        print("Congratulation. You made it!")
  
```

"break" shouldn't be confused with continue statement.

"continue" stops the current iteration of the loop and starts the next iteration by checking the condition.

Note: If a loop is left by break, the else part is not executed.

Topic 4c: For Loop

Introduction

Python for loop is an iterator based for loop. It steps through the items of lists, tuples, strings, the keys of dictionaries and other iterables. General Syntax is:

```
for <variable> in <sequence>:  
    <statements>  
else:  
    <statements>
```

The items of the sequence object are assigned one after the other to the loop variable. For each item the loop body is executed.

Examples:

```
>>> languages = ["C", "C++", "Perl", "Python"]  
>>> for x in languages:  
...     print(x)  
...  
C  
C++  
Perl  
Python
```

If a break statement has to be executed in the program flow of the for loop, the loop will be exited and the program flow will continue with the first statement following the for loop, if there is any at all.

```
edibles = ["ham", "spam", "eggs", "nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

```
edibles = ["ham", "spam", "eggs", "nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        continue
    print("Great, delicious " + food)
    # here can be the code for enjoying our
food :-)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

The range() function

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions:

Example:

```
range(begin, end, step)
```

```
>>> list(range(4, 50, 5))
[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]
```

```
>>> list(range(42, -12, -7))
[42, 35, 28, 21, 14, 7, 0, -7]
```

```

fibonacci = [0,1,1,2,3,5,8,13,21]
for i in range(len(fibonacci)):
    print(i,fibonacci[i])
print()

```

The following program calculates all pythagorean numbers less than a maximal number. Three integers satisfying $a^2+b^2=c^2$ are called Pythagorean numbers.

```

from math import sqrt
n = int(input("Maximal Number? "))
for a in range(1,n+1):
    for b in range(a,n):
        c_square = a**2 + b**2
        c = int(sqrt(c_square))
        if ((c_square - c**2) == 0):
            print(a, b, c)

```

List Iteration with Side Effects

If loop over a list, avoid changing the list in the loop body.

```

colours = ["red"]
for i in colours:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print(colours)

```

```

colours = ["red"]
for i in colours[:]:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print(colours)

```

Using slicing operator, even if changing list, the elements to be looped will remain the same during the iterations.

Topic 4d: Iterators & Iterables

Introduction

We have seen that we can loop or iterate over various Python objects like lists, tuples and strings for example.

```
for city in ["Berlin", "Vienna", "Zurich"]:  
    print(city)  
for city in ("Python", "Perl", "Ruby"):  
    print(city)  
for char in "Iteration is easy":  
    print(char)
```

This form of looping can be seen as iteration. Iteration is not restricted to explicit for loops. If you call function sum e.g. on a list of integer values, - you do iteration as well.

So what is the difference between an iterable and an iterator?

- In one perspective they are the same: You can iterate with a for loop over iterators and iterables.
- Every iterator is also an iterable, but not every iterable is an iterator.
- E.g. a list is iterable but a list is not an iterator!

An iterator can be created from an iterable by using the function 'iter'. Iterators are objects with a '`__next__`' method, which will be used when the function 'next' is called.

So what is going on behind the scenes, when a for loop is executed?

- The for statement calls iter() on the object (which should be a so-called container object), which it is supposed to loop over.
- If this call is successful, the iter call will return return an iterator object that defines the method __next__() which accesses elements of the object one at a time.
- The __next__() method will raise a StopIteration exception, if there are no further elements available.

```
cities = ["Berlin", "Vienna", "Zurich"]
iterator_obj = iter(cities)
print(iterator_obj)
print(next(iterator_obj))
print(next(iterator_obj))
print(next(iterator_obj))
```

The following function 'iterable' will return True, if the object 'obj' is an iterable and False otherwise.

```
def iterable(obj):
    try:
        iter(obj)
        return True
    except TypeError:
        return False

for element in [34, [4, 5], (4, 5), {"a":4},
"dfsdf", 4.5]:
    print(element, "iterable: ",
iterable(element))
```

Topic 5a: Output with Print

Introduction

One of the most frequently occurring errors will be related to print, because print is a function in Python 3.x

```
>>> print 42
      File "", line 1
          print 42
                  ^
SyntaxError: invalid syntax
```

```
>>> print(42)
42
```

The arguments of the print function are the following:

```
print(value1, ..., sep=' ', end='\n',
      file=sys.stdout, flush=False)
```

The print function can print an arbitrary number of values ("value1, value2, ..."), which are separated by commas. These values are separated by blanks.

Examples:

```
>>> print("a", "b")
a b
>>> print("a", "b", sep="")
ab
>>> print(192, 168, 178, 42, sep=". ")
192.168.178.42
>>> print("a", "b", sep=":-) ")
a:-) b
```

```
>>> print("a = ", a)
a = 3.564
>>> print("a = \n", a)
a =
3.564
```

By default, A print call is ended by a newline. To change this behaviour, we can assign an arbitrary string to the

keyword parameter "end". This string will be used for ending the output of the values of a print call:

```
>>> for i in range(4):  
...     print(i)  
...  
0  
1  
2  
3
```

```
>>> for i in range(4):  
...     print(i, end=" ")  
...  
0 1 2 3 >>>  
>>> for i in range(4):  
...     print(i, end=" :-) ")  
...  
0 :-) 1 :-) 2 :-) 3 :-) >>>
```

The output of the print function is send to the standard output stream (sys.stdout) by default. By redefining the keyword parameter "file" we can send the output into a different stream e.g. sys.stderr or a file:

```
>>> fh = open("data.txt", "w")  
>>> print("42 is the answer, but what is the question?", file=fh)  
>>> fh.close()
```

We don't get any output in the interactive shell. Output is sent to the file "data.txt". It's also possible to redirect the output to the standard error channel this way:

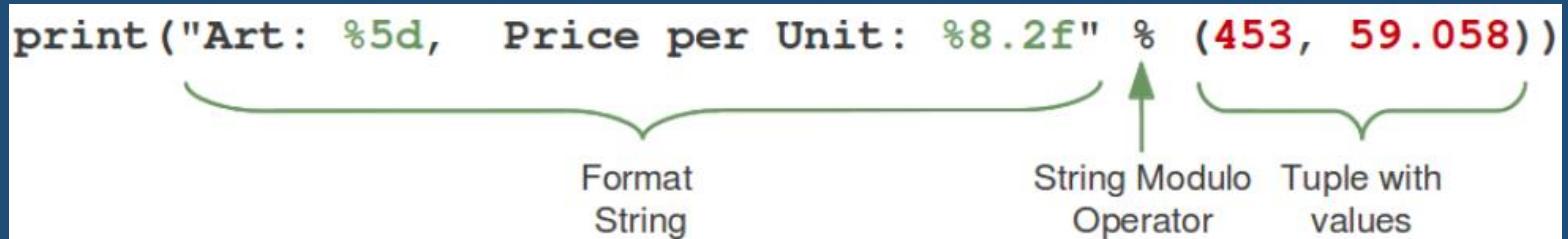
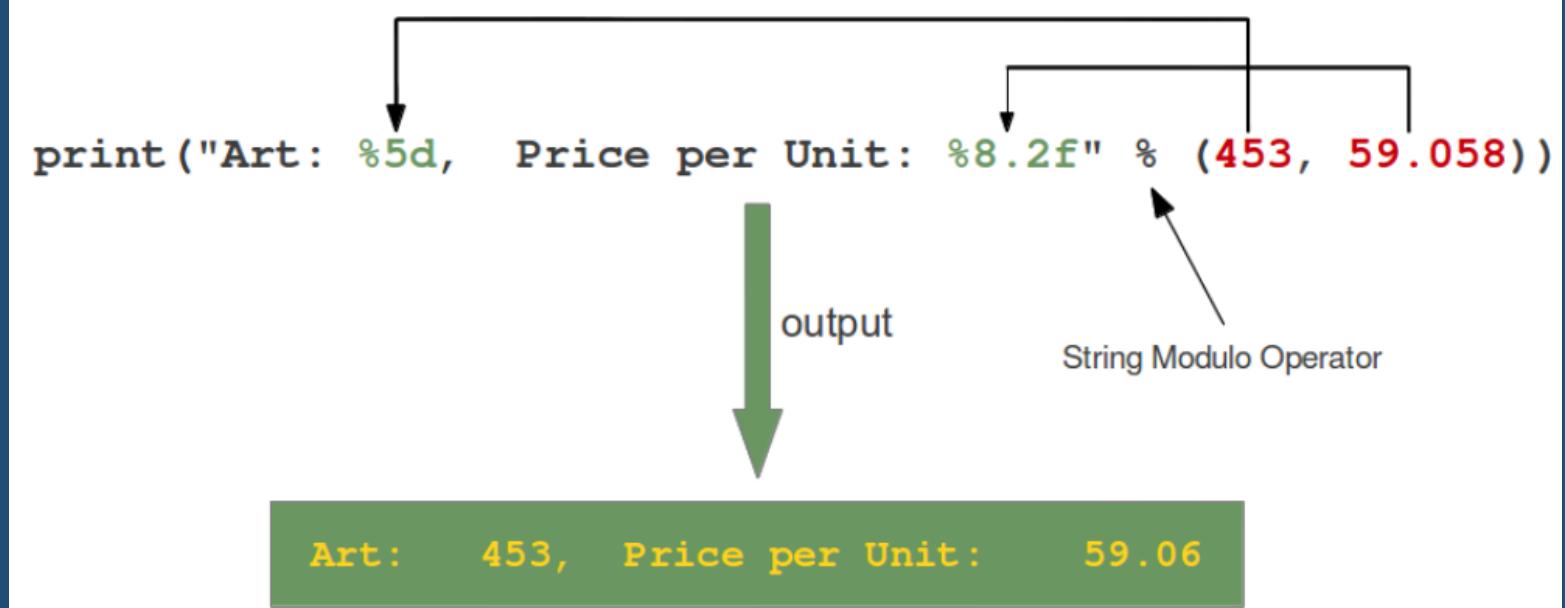
```
>>> import sys  
>>> # output into sys.stderr:  
...  
>>> print("Error: 42", file=sys.stderr)  
Error: 42
```

Topic 5b: String Modulo

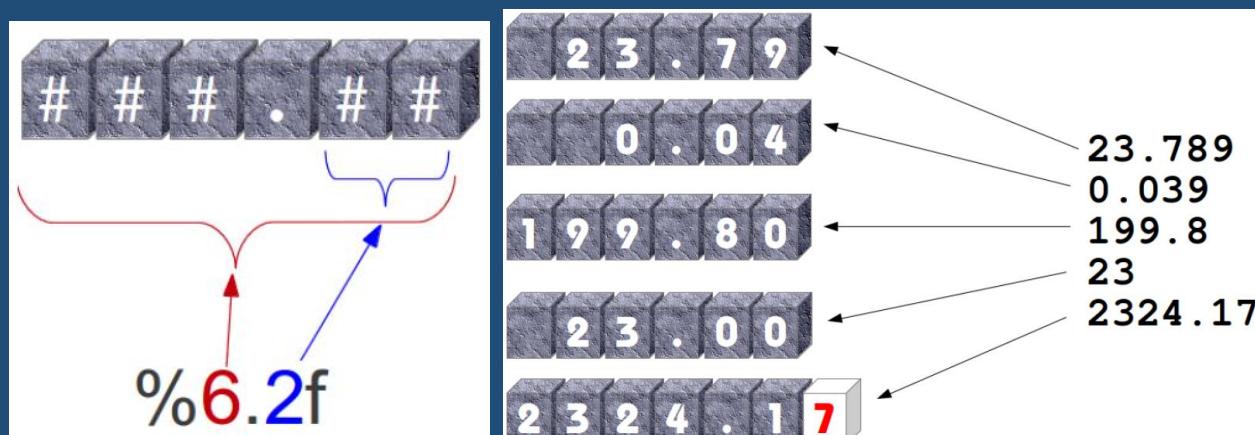
Introduction

String Modulo “%” is the string formatter in Python.

For the same purpose str.format() can also be used.



The format string contains placeholders. There are two of those in our example: "%5d" & "%8.2f". The general syntax for format placeholder is: `%[flags] [width] [.precision] type`



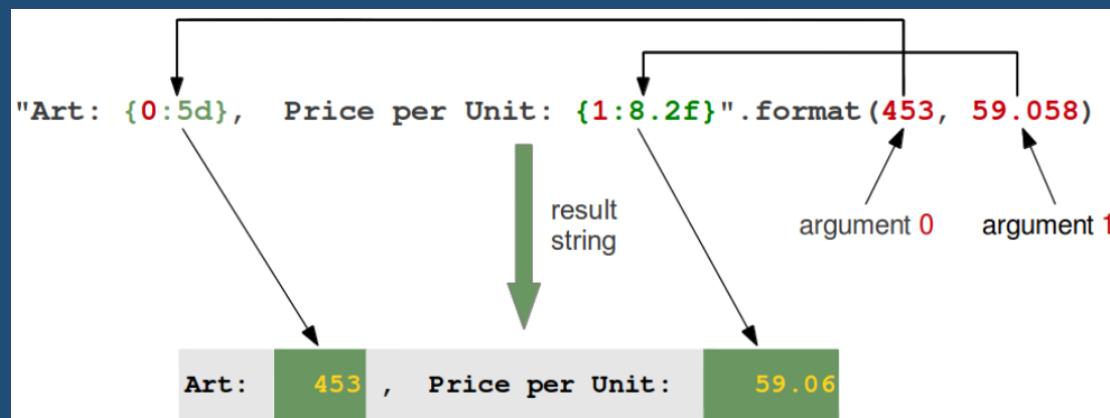
Topic 5c: Format Method

Introduction

The format method was added in Python 2.6. The general form of this method looks like this:

```
template.format(p0, p1, ..., k0=v0, k1=v1,  
...)
```

The template (or format string) is a string which contains one or more format codes (fields to be replaced) embedded in constant text. The "fields to be replaced" are surrounded by curly braces {}.



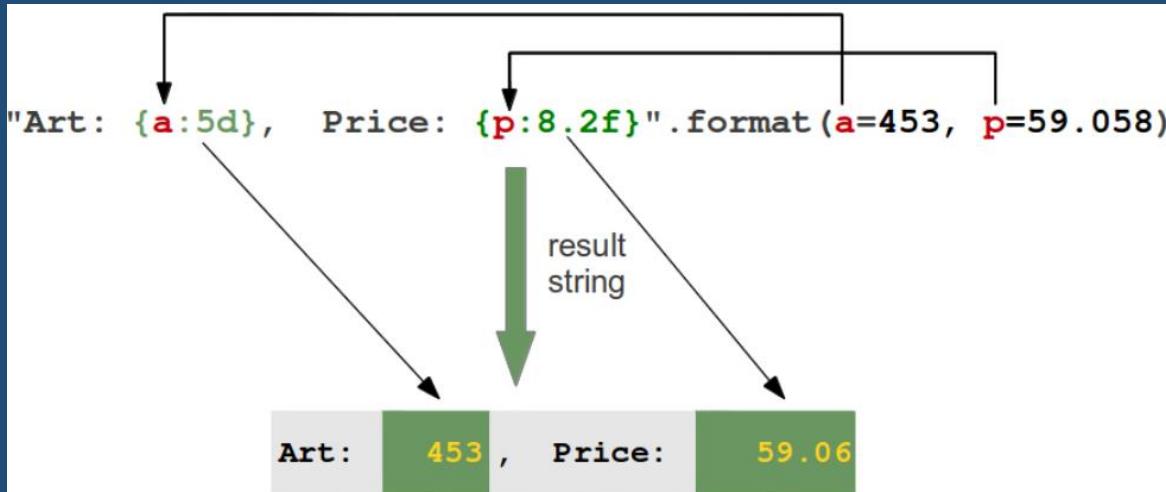
Examples of positional parameters:

```
>>> "First argument: {0}, second one:  
{1} ".format(47,11)  
'First argument: 47, second one: 11'  
>>> "Second argument: {1}, first one:  
{0} ".format(47,11)  
'Second argument: 11, first one: 47'  
>>> "Second argument: {1:3d}, first one:  
{0:7.2f} ".format(47.42,11)  

```

Keyword parameters used with the format method:

```
>>> "Art: {a:5d}, Price:  
{p:8.2f} ".format(a=453, p=59.058)  
'Art: 453, Price: 59.06'
```



It's possible to left or right justify data with this method:

```
>>> "{0:<20s} {1:6.2f} ".format('Spam & Eggs:',  
6.99)  
'Spam & Eggs: 6.99'  
>>> "{0:>20s} {1:6.2f} ".format('Spam & Eggs:',  
6.99)  
'          Spam & Eggs: 6.99'
```

Dictionaries in “format”

```
>>> print("The capital of {0:s} is  
{1:s} ".format("Ontario", "Toronto"))  
The capital of Ontario is Toronto
```

```
>>> print("The capital of {province} is  
{capital} ".format(province="Ontario", capi  
tal="Toronto"))  
The capital of Ontario is Toronto
```

```
>>> print("The capital of {province} is  
{capital} ".format(province="Ontario", capi  
tal="Toronto"))  
The capital of Ontario is Toronto
```

```
>>> data =  
dict(province="Ontario",capital="Toronto")  
>>> data  
{'province': 'Ontario', 'capital': 'Toronto'}  
>>> print("The capital of {province} is  
{capital}".format(**data))  
The capital of Ontario is Toronto
```

The double "*" in front of data turns data automatically into the form 'province="Ontario",capital="Toronto"'.

```
capital_country = {"United States" :  
"Washington",  
                   "US" : "Washington",  
                   "Canada" : "Ottawa",  
                   "Germany": "Berlin",  
                   "France" : "Paris",  
                   "England" : "London",  
                   "UK" : "London",  
                   "Switzerland" : "Bern",  
                   "Austria" : "Vienna",  
                   "Netherlands" :  
"Amsterdam"}  
  
print("Countries and their capitals:")  
for c in capital_country:  
    format_string = c + ": {" + c + "}"  
  
print(format_string.format(**capital_country))
```

Using local variable names in “format”

"locals" is a function, which returns a dictionary with the current scope's local variables, i.e- the local variable names are the keys of this dictionary and the corresponding values are the values of these variables:

```
>>> a = 42  
>>> b = 47  
>>> def f(): return 42  
...  
>>> locals()  
{'a': 42, 'b': 47, 'f': <function f at  
0xb718ca6c>, '__builtins__': <module  
'builtins' (built-in)>, '__package__': None,  
'__name__': '__main__', '__doc__': None}
```

Continuing with above example, we can create following output, in which we use the local variables a, b and f:

```
>>> print ("a={a}, b={b} and f=\n{f} ".format(**locals()))\na=42, b=47 and f=<function f at 0xb718ca6c>
```

Formatted String Literals

Python 3.6 introduces formatted string literals. They are prefixed with an 'f'. The formatting syntax is similar to the format strings accepted by str.format(). Like the format string of format method, they contain replacement fields formed with curly braces. The replacement fields are expressions, which are evaluated at run time, and then formatted using the format() protocol.

```
>>> price = 11.23\n>>> f"Price in Euro: {price}"\n'Price in Euro: 11.23'\n>>> f"Price in Swiss Franks: {price * 1.086}"\n'Price in Swiss Franks: 12.195780000000001'\n>>> f"Price in Swiss Franks: {price *\n1.086:5.2f}"\n'Price in Swiss Franks: 12.20'\n>>> for article in ["bread", "butter", "tea"]:\n...     print(f"{article:>10}:")\n...\nbread:\nbutter:\nteat:
```

Topic 6a: Functions

Introduction

A Function is a structuring element in programming languages to group a set of statements so they can be utilized more than once in a program. A function in Python is defined by a def statement. The syntax is:

```
def function-name(Parameter list):  
    statements, i.e. the function body
```

- The parameter list consists of none or more parameters & called arguments if function is called.
- Function bodies can contain one or more return statement. They can be situated anywhere in the function body.
- A return statement ends the execution of the function call and "returns" the result.
- If the return statement is without an expression or no return statement, the special value None is returned.

```
def fahrenheit(T_in_celsius):  
    """ returns the temperature in degrees  
Fahrenheit """  
    return (T_in_celsius * 9 / 5) + 32  
  
for t in (22.6, 25.8, 27.3, 29.8):  
    print(t, ":", fahrenheit(t))
```

Doc String

The first statement in the body of a function is usually a string which can be accessed with `function_name.__doc__`. This statement is called Docstring.

```
def Hello(name="everybody"):  
    """ Greets a person """  
    print("Hello " + name + "!")
```

```
print("The docstring of the function Hello: "  
+ Hello.__doc__)
```

Examples

```
def sumsub(a, b, c=0, d=0):  
    return a - b + c - d  
  
print(sumsub(12, 4))  
print(sumsub(42, 15, d=10))
```

```
def no_return(x, y):  
    c = x + y  
  
res = no_return(4, 5)  
print(res)
```

```
def return_sum(x, y):  
    c = x + y  
    return c  
  
res = return_sum(4, 5)  
print(res)
```

```
def empty_return(x, y):  
    c = x + y  
    return  
  
res = empty_return(4, 5)  
print(res)
```

```
def Hello(name="everybody"):  
    """ Greets a person """  
    print("Hello " + name + "!")  
  
Hello("Peter")  
Hello()
```

```
def fib_intervall(x):  
    """ returns the largest fibonacci  
    number smaller than x and the lowest  
    fibonacci number higher than x"""  
  
    if x < 0:  
        return -1  
  
(old,new, lub) = (0,1,0)  
  
    while True:  
  
        if new < x:  
            lub = new  
  
            (old,new) = (new,old+new)  
  
        else:  
            return (lub, new)  
  
    while True:  
  
        x = int(input("Your number: "))  
  
        if x <= 0:  
            break  
  
(lub, sup) = fib_intervall(x)  
  
print("Largest Fibonacci Number smaller than x: " + str(lub))  
print("Smallest Fibonacci Number larger than x: " + str(sup))
```

Local and Global Variables

Variable names are by default local to the function, in which they get defined.

```
def f():
    s = "Perl"
    print(s)
s = "Python"
f()
print(s)
```

```
def f():
    s = "Perl"
    print(s)
s = "Python"
f()
print(s)
```

```
def f():
    print(s)
    s = "Perl"
    print(s)
s = "Python"
f()
print(s)
```

```
def f():
    global s
    print(s)
    s = "dog"
    print(s)
s = "cat"
f()
print(s)
```

In the third example above, variable `s` is ambiguous in `f()`, i.e. first `print` in `f()` the global `s` could be used with the value "Python". After this we define a local variable `s` with the assignment `s = "Perl"`.

Now, in the fourth example, we made the variable s global inside of the script on the left side. Therefore anything we do to s inside of the function body of f is done to the global variable s outside of f.

More Examples

```
def arithmetic_mean(first, *values):
    """ This function calculates the
arithmetic mean of a non-empty
    arbitrary number of numerical values
"""

    return (first + sum(values)) / (1 +
len(values))

print(arithmetic_mean(45,32,89,78))
print(arithmetic_mean(8989.8,78787.78,3453,787
78.73))
print(arithmetic_mean(45,32))
print(arithmetic_mean(45))
```

This is great, but if we have a list of numerical values, we can use “*” to unpack the list when calling the function.

```
x = [3, 5, 9] arithmetic_mean(x)

arithmetic_mean(x[0], x[1], x[2])

arithmetic_mean(*x)
```

```
>>> def f(**kwargs):
...     print(kwargs)
...
>>> f()
{}
>>> f(de="German", en="English", fr="French")
{'fr': 'French', 'de': 'German', 'en':
'English'}
```

Topic 6b: Recursive Function

Recursion is a method of programming in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call.

```
def factorial(n):
    print("factorial has been called with n = "
" + str(n) )
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, "
* factorial(" ,n-1, "): ",res)
        return res

print(factorial(5))
```

Exercise:

Create a deeply nested sublist and print each and every element using simple function or a recursive function.

Hint – use `isinstance()` built-in function

Topic 6c: Parameters & Arguments

Call by Value v/s Call by Reference

The evaluation strategy for arguments, i.e. how the arguments from a function call are passed to the parameters of the function, differs in each programming language. The most common evaluation strategies are "call by value" and "call by reference":

- **Call by Value:**

The argument expression is evaluated, and the result of this evaluation is bound to the corresponding variable in the function. So, its value will be assigned (copied) to the corresponding parameter. This ensures that the variable in the caller's scope will be unchanged when the function returns.

- **Call by Reference:**

Here, function gets an implicit reference to the argument, rather than a copy of its value. Thus, the function can modify the argument, i.e. the value of the variable in the caller's scope can be changed.

We save computation time and memory space, as arguments do not need to be copied. But need to be careful as that variables can be "accidentally" changed in the function call.

And what about Python?

Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".

```
def ref_demo(x) :  
    print("x=",x," id=",id(x))  
    x=42  
    print("x=",x," id=",id(x))
```

```
>>> x = 9  
>>> id(x)  
9251936  
>>> ref_demo(x)  
x= 9  id= 9251936  
x= 42  id= 9252992  
>>> id(x)  
9251936
```

Examples:

```
>>> def no_side_effects(cities):  
...     print(cities)  
...     cities = cities + ["Birmingham",  
"Bradford"]  
...     print(cities)  
...  
>>> locations = ["London", "Leeds", "Glasgow",  
"Sheffield"]  
>>> no_side_effects(locations)  
['London', 'Leeds', 'Glasgow', 'Sheffield']  
[['London', 'Leeds', 'Glasgow', 'Sheffield',  
'Birmingham', 'Bradford']]  
>>> print(locations)  
['London', 'Leeds', 'Glasgow', 'Sheffield']
```

```
>>> def side_effects(cities):  
...     print(cities)  
...     cities += ["Birmingham", "Bradford"]  
...     print(cities)  
...  
>>> locations = ["London", "Leeds", "Glasgow",  
"Sheffield"]  
>>> side_effects(locations)  
['London', 'Leeds', 'Glasgow', 'Sheffield']  
[['London', 'Leeds', 'Glasgow', 'Sheffield',  
'Birmingham', 'Bradford']]  
>>> print(locations)  
['London', 'Leeds', 'Glasgow', 'Sheffield',  
'Birmingham', 'Bradford']
```

```
>>> def side_effects(cities):  
...     print(cities)  
...     cities += ["Paris", "Marseille"]  
...     print(cities)  
...  
>>> locations = ["Lyon", "Toulouse", "Nice",  
"Nantes", "Strasbourg"]  
>>> side_effects(locations[:])  
['Lyon', 'Toulouse', 'Nice', 'Nantes',  
'Strasbourg']  
[['Lyon', 'Toulouse', 'Nice', 'Nantes',  
'Strasbourg', 'Paris', 'Marseille']]  
>>> print(locations)  
['Lyon', 'Toulouse', 'Nice', 'Nantes',  
'Strasbourg']
```

Variable length of Parameters

The asterisk "*" is used in Python to define a variable number of arguments. The asterisk character has to precede a variable identifier in the parameter list.

```
>>> def varpafu(*x) : print(x)
...
>>> varpafu()
()
>>> varpafu(34,"Do you like Python?", "Of course")
(34, 'Do you like Python?', 'Of course')
```

```
>>> def locations(city, *other_cities) :
print(city, other_cities)
...
>>> locations("Paris")
Paris ()
>>> locations("Paris", "Strasbourg", "Lyon",
"Dijon", "Bordeaux", "Marseille")
Paris ('Strasbourg', 'Lyon', 'Dijon',
'Bordeaux', 'Marseille')
```

A * can appear in function calls as well

```
>>> def f(x,y,z) :
...     print(x,y,z)
...
>>> p = (47,11,12)
>>> f(*p)
(47, 11, 12)
```

```
>>> f(p[0],p[1],p[2])
(47, 11, 12)
```

Topic 6d: Namespaces

Introduction

Namespace is a naming system for making names unique. In Python, they are implemented as dictionaries by a mapping from names, i.e. the keys of the dictionary, to objects, i.e. the values. Namespaces have different lifetimes, as per the points of time they are created.

Some namespaces in Python:

1. global names of a module

The global namespace of a module is generated when the module is read in. Module namespaces normally last until the script ends, i.e. the interpreter quits.

2. local names in a function or method invocation :

When a function is called, a local namespace is created for this function. This namespace is deleted either if the function ends, i.e. returns, or if the function raises an exception, which is not dealt with within the function.

3. built-in names: This namespace contains built-in functions, created when the Python interpreter starts up, and is never deleted.

Topic 6e: Global & Local Variables

Introduction

While in many or most other programming languages variables are treated as global if not otherwise declared, Python deals with variables the other way around. They are local, if not otherwise declared.

The driving reason behind this approach is that global variables are generally bad practice and should be avoided. In most cases where we are tempted to use a global variable, it's better to utilize a parameter for getting a value into function or return value to get it out.

So when you define variables inside a function definition, they are local to this function by default. This means that anything you will do to such a variable in the body of the function will have no effect on other variables outside of the function, even if they have the same name.

Examples:

```
def f():
    print(s)
s = "I love Paris in the summer!"
f()
```

```
def f():
    s = "I love London!"
    print(s)

s = "I love Paris!"
f()
print(s)
```

```
>>> def f():
...     print(s)
...     s = "I love London!"
...     print(s)
...
>>> s = "I love Paris!"
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in f
UnboundLocalError: local variable 's' referenced before assignment
```

```
def f():
    global s
    print(s)
    s = "Only in spring, but London is great
as well!"
    print(s)
```

```
s = "I am looking for a course in Paris!"
f()
print(s)
```

```
def f():
    s = "I am globally not known"
    print(s)

f()
print(s)
```

```
def f():
    x = 42
    def g():
        global x
        x = 43
        print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

f()
print("x in main: " + str(x))
```

Nonlocal Variables

Python3 introduced nonlocal variables as a new kind of variables. nonlocal variables have a lot in common with global variables. One difference to global variables lies in the fact that it is not possible to change variables from the module scope, i.e. variables which are not defined inside of a function, by using the nonlocal statement.

```
def f():
    global x
    print(x)

x = 3
f()
```

```
def f():
    nonlocal x
    print(x)

x = 3
f()
```

```
def f():
    x = 42
    def g():
        nonlocal x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))
```

```
x = 3
f()
print("x in main: " + str(x))
```

```
def f():
    #x = 42
    def g():
        nonlocal x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))
```

```
x = 3
f()
print("x in main: " + str(x))
```

```
def f():
    #x = 42
    def g():
        global x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))
```

```
x = 3
f()
print("x in main: " + str(x))
```

Topic 6f: Decorators

Introduction

A decorator in Python is any callable Python object that is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class. The modified functions or classes usually contain calls to the original function "func" or class "C".

First Steps to Decorators

```
>>> def succ(x):  
...     return x + 1  
...  
>>> successor = succ  
>>> successor(10)  
11  
>>> succ(10)  
11
```

```
>>> del succ  
>>> successor(10)  
11
```

Functions inside Functions

```
def f():  
  
    def g():  
        print("Hi, it's me 'g'")  
        print("Thanks for calling me")  
  
        print("This is the function 'f'")  
    print("I am calling 'g' now:")  
    g()  
  
f()
```

Functions as Parameters

```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()

f(g)
```

```
import math

def foo(func):
    print("The function " + func.__name__ + " was passed to foo")
    res = 0
    for x in [1, 2, 2.5]:
        res += func(x)
    return res

print(foo(math.sin))
print(foo(math.cos))
```

Functions returning Functions

```
def f(x):
    def g(y):
        return y + x + 3
    return g

nf1 = f(1)
nf2 = f(3)

print(nf1(1))
print(nf2(1))
```

```
def polynomial_creator(a, b, c):
    def polynomial(x):
        return a * x**2 + b * x + c
    return polynomial

p1 = polynomial_creator(2, 3, -1)
p2 = polynomial_creator(-1, 2, 1)

for x in range(-2, 2, 1):
    print(x, p1(x), p2(x))
```

First Decorator

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

def foo(x):
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo(42)
```

Correct decoration occurs in the line before function header. “@” is followed by the decorator function name.

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def succ(n):
    return n + 1

succ(10)
```

```
from math import sin, cos

def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

sin = our_decorator(sin)
cos = our_decorator(cos)

for f in [sin, cos]:
    f(3.1415)
```

The following example uses a decorator to count the number of times a function has been called.

```
def call_counter(func):
    def helper(x):
        helper.calls += 1
        return func(x)
    helper.calls = 0

    return helper

@call_counter
def succ(x):
    return x + 1

print(succ.calls)
for i in range(10):
    succ(i)

print(succ.calls)
```