

# Programming in Python

Lecture 1

**Introduction.**

# Table of contents

- ✦ Course plan and grading system
- ✦ An Algorithm
- ✦ Paradigms of programming
- ✦ Concept of data type
- ✦ Python language: let's start
- ✦ Yandex Contest verdicts

# Programming in Python

*“Whether you want to uncover the secrets of the universe, or you just want to pursue a career in the 21<sup>st</sup> century, basic computer programming is an essential skill to learn.”*

*Stephen Hawking*

## *Main goals*

- ♦ To learn Python
- ♦ To understand algorithms and to develop algorithmic thinking
- ♦ To understand, what is a good style of programming
- ♦ Get started with data
- ♦ To practice
- ♦ ...
- ♦ To practice

# Course plan

- ✦ Introduction.
- ✦ Numeric data types. Variables. Conditional Code.
- ✦ Number systems.
- ✦ Strings.
- ✦ Tuples. Lists.
- ✦ Functions and recursion.
- ✦ Sets. Dictionaries.
- ✦ Elements of functional programming.
- ✦ Sort and found.
- ✦ Basic data structures.
- ✦ Python for data analysis. First project.

# Grading system



$$\text{Final grade} = 0.4 * E + 0.6 * OA$$

Where: E – Exam mark, OA – Ongoing Assessment

$$OA = 10 * \frac{RP + BP}{RP_{max}}$$

RP – Regular Points: *Homework (Contests), Control Work (CW)*

$$RP_{max} = \sum_{activities} \text{max possible } RP$$

BP - Bonus Points:

*Extra tasks (with \*) in contests (for main groups)*

*Quizzes*

*Working during the workshop*

*Activity at almost all workshops*

*Other achievements at the teacher discretion*

# Grading system



## **Additional conditions:**

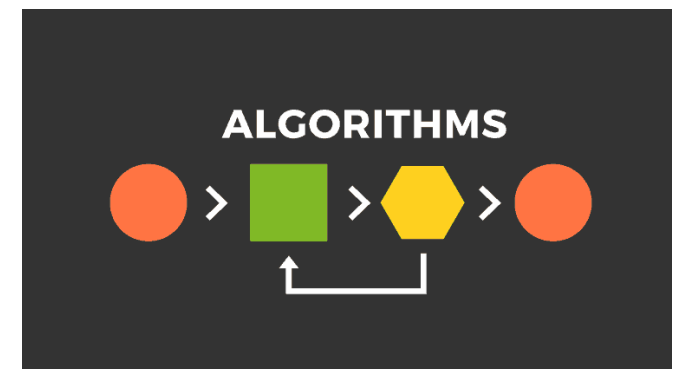
1. Final grades for the CW and Contests are given after the defenses (inviting students to defenses is at the discretion of the teacher). The student should be informed of the need to defend the work no later than 2 weeks after the deadline for its submission. In case of plagiarism, one task nullifies the entire work.
2. If (in % of full mark) Contests - CW  $\geq 50$  then RP for Contests (all with the deadlines earlier, then CW) divided by 2.

# An algorithm

- ♦ The sequence of steps carried out during a computation are defined by ***an algorithm***
  - ❖ an algorithm can be thought of as a “recipe” or “prescription”
  - ❖ “follow these steps and you will solve your problem”
- ♦ An algorithm includes a complete description of
  - ❖ the set of ***inputs***, or starting conditions
    - ▶ a full specification of the problem to be solved
  - ❖ the set of ***outputs***
    - ▶ descriptions of valid solutions to the problem
  - ❖ a sequence of ***operations*** that will eventually produce the output
    - ▶ steps must be simple and precise



USSR commemorative stamp



# Programming languages

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."  
Edsger W. Dijkstra

- ♦ If we want a machine to carry out a computation on, we need to write a ***program***
- ♦ A ***programming language***
  - ❖ is a notation for describing the steps of a computation
  - ❖ formal sign system for recording computer programs.
  - ❖ defines a set of **lexical, syntactic, and semantic rules** that determine the appearance of the program and the actions that the computer will perform under its control.
- ♦ currently encyclopedia of programming languages <http://progopedia.com/> contains:
  - ❖ 114 programming languages,
  - ❖ 30 dialects,
  - ❖ 182 implementations,
  - ❖ 258 versions.



# Computer program. Stages.

## Task:

print the string, which contains 10 symbols 'A'.

## Algorithms:

1. Constrain the string from 10 symbols "A" and print it;
2. Run 10 times the cycle, which print one symbol "A"
3. Use multiplication of a number and a string

## Programming:

1. *`print('AAAAAAAAAA')`*
2. *`for i in range (10): print('A', end = "")`*
3. *`print(10*'A')`*

# Programming languages

Low level

High level

Machine language  
Assembler

Imperative  
(Procedural)

Declarative

Fortran  
Pascal  
C

SQL

Logic  
Prolog

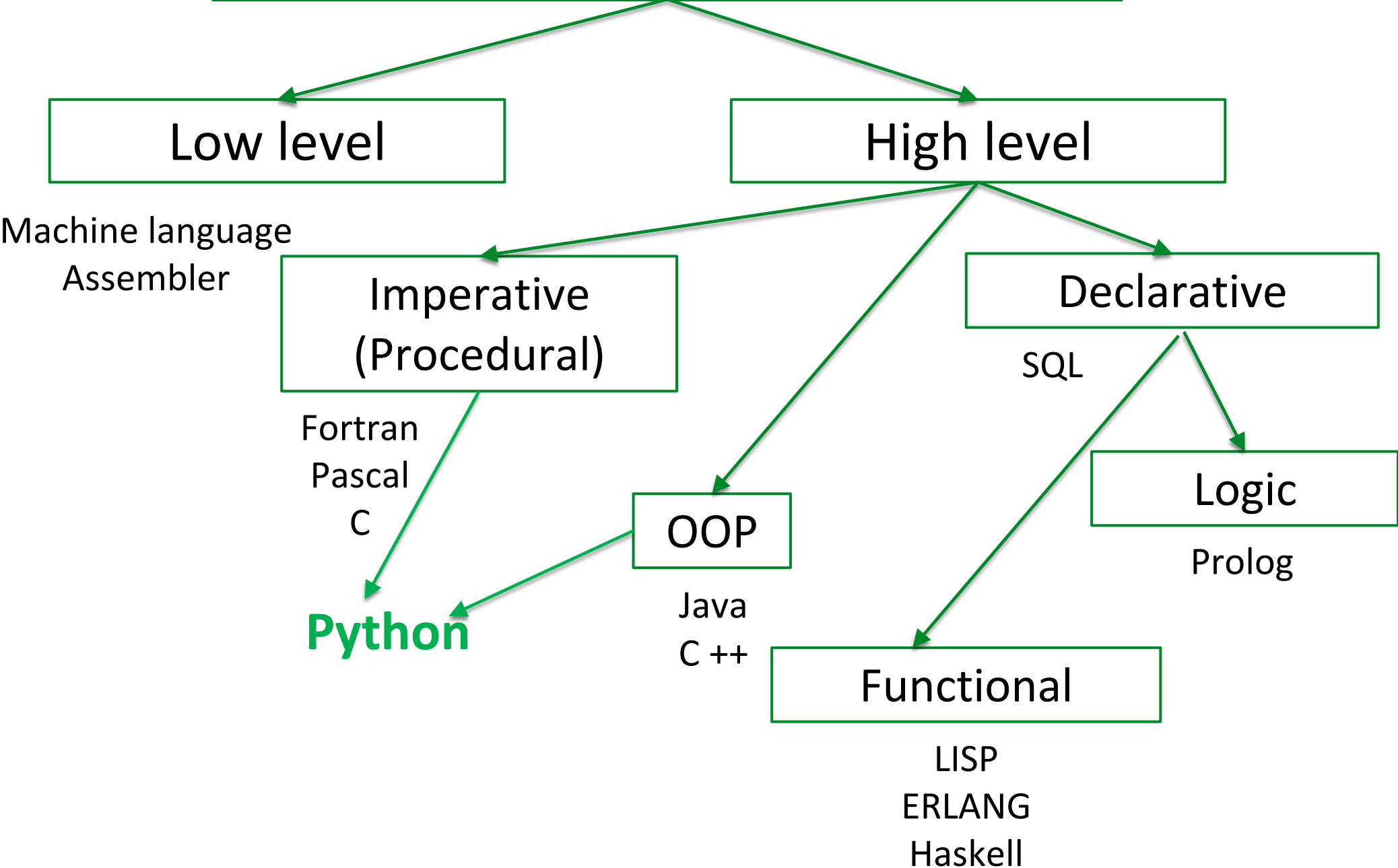
OOP

Java  
C ++

Functional

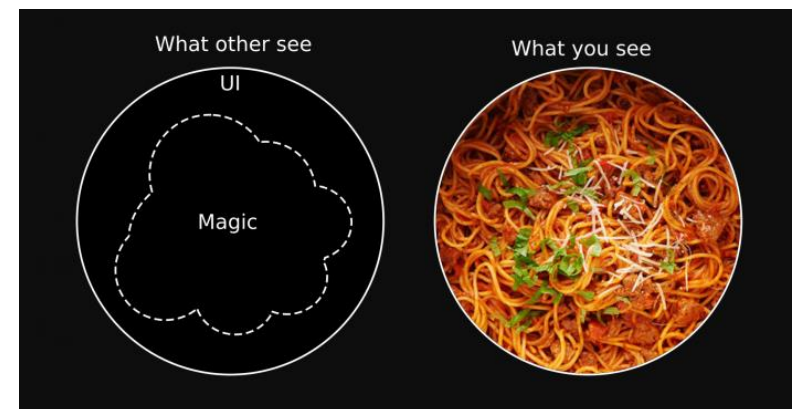
LISP  
ERLANG  
Haskell

**Python**



# Imperative

- ✦ Oldest approach.
- ✦ Closest to the actual mechanical behavior of a computer => original imperative
- ✦ languages were abstractions of assembly language.
- ✦ **A program is a list of instructions that change a memory state until desired “end state” is achieved.**
- ✦ Useful for quite simple programs.
- ✦ Difficult to scale.
- ✦ Soon (often/ may) it led to spaghetti code (with incomprehensible logic).

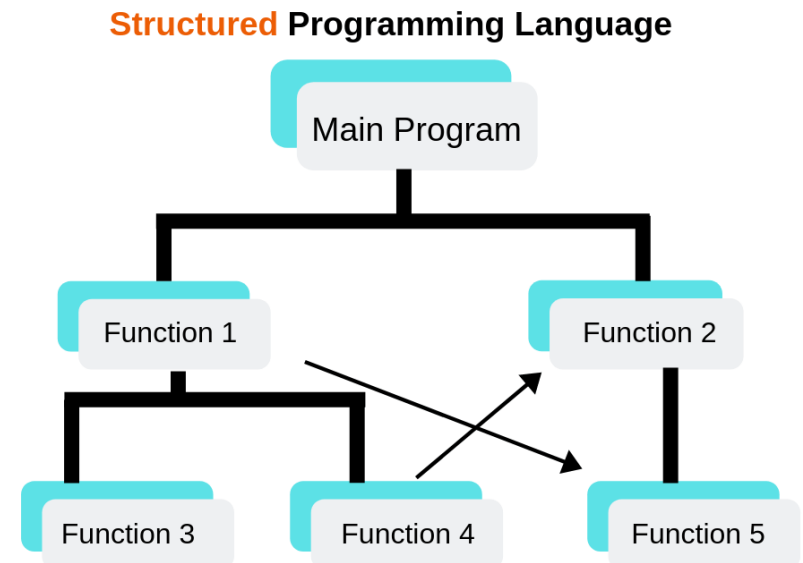


# Structured → Procedural

- ✦ **GoTo** Statement Considered Harmful, by Edsger Dijkstra in 1968.
- ✦ **Structured program theorem (Bohm-Jacopini):** sequencing, selection, and iteration are sufficient to express any computable function.
- ✦ Calculations are described in the form of **instructions, step by step changing the state of the program / data.**
- ✦ Divide the code in procedures: routines, subroutines, modules methods, or functions.

## Advantages:

1. Division of work.
2. Debugging and testing.
3. Maintenance.
4. Reusability.



# Declarative

- ✦ A programming paradigm that **expresses the logic of a computation without describing its control flow.**
- ✦ Declarative languages state **what should be done, not how.** You describe the desired result without delving into the instructions.
- ✦ Common declarative languages include those of database query languages (e.g., SQL, XQuery), regular expressions, logic programming, functional programming.

# Functional programming

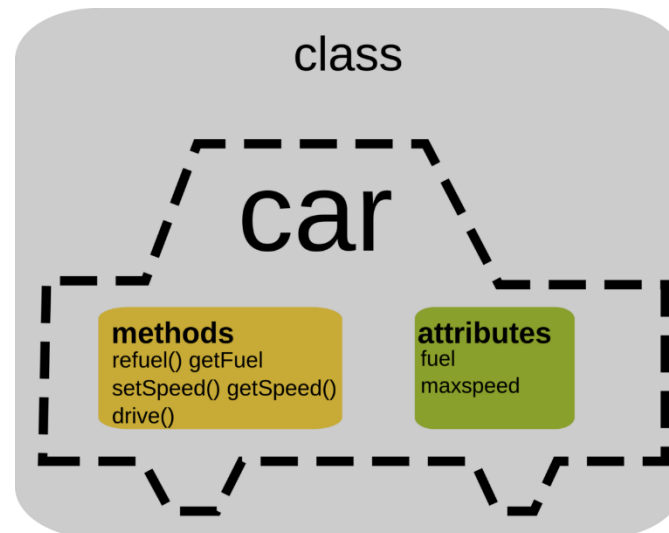
- ✦ Nearly as old as imperative programming.
- ✦ Created by John McCarthy with LISP (list processing) in the late 1950s.
- ✦ **The key concept is an expression.**
- ✦ This is a way to write programs with the aid of the only action – **function's calling.**
- ✦ Memory is not used as a place of data and programs storage.
- ✦ Intermediate variables, assignment operators, and loops are not used. **A program is a sequence of descriptions of functions and expressions.**
- ✦ The expression is calculated by going from the complex to the simple. All expressions are the lists.

Often functional programs are:

1. Easier to read.
2. Easier to debug and maintain.
3. Easier to parallelize.

# Object-oriented programming

- ✦ This is **programming from objects**.
- ✦ A program is a collection of related objects. **Each object is a set of some data and a set of actions** (methods) that it can do.
- ✦ 1970s: Smalltalk from the Xerox PARC.
- ✦ Large impact on software industry.
- ✦ Partial support in several languages: structures in C (and structs in older versions of Matlab). Now even Fortran has OO support.



# Data types in programming

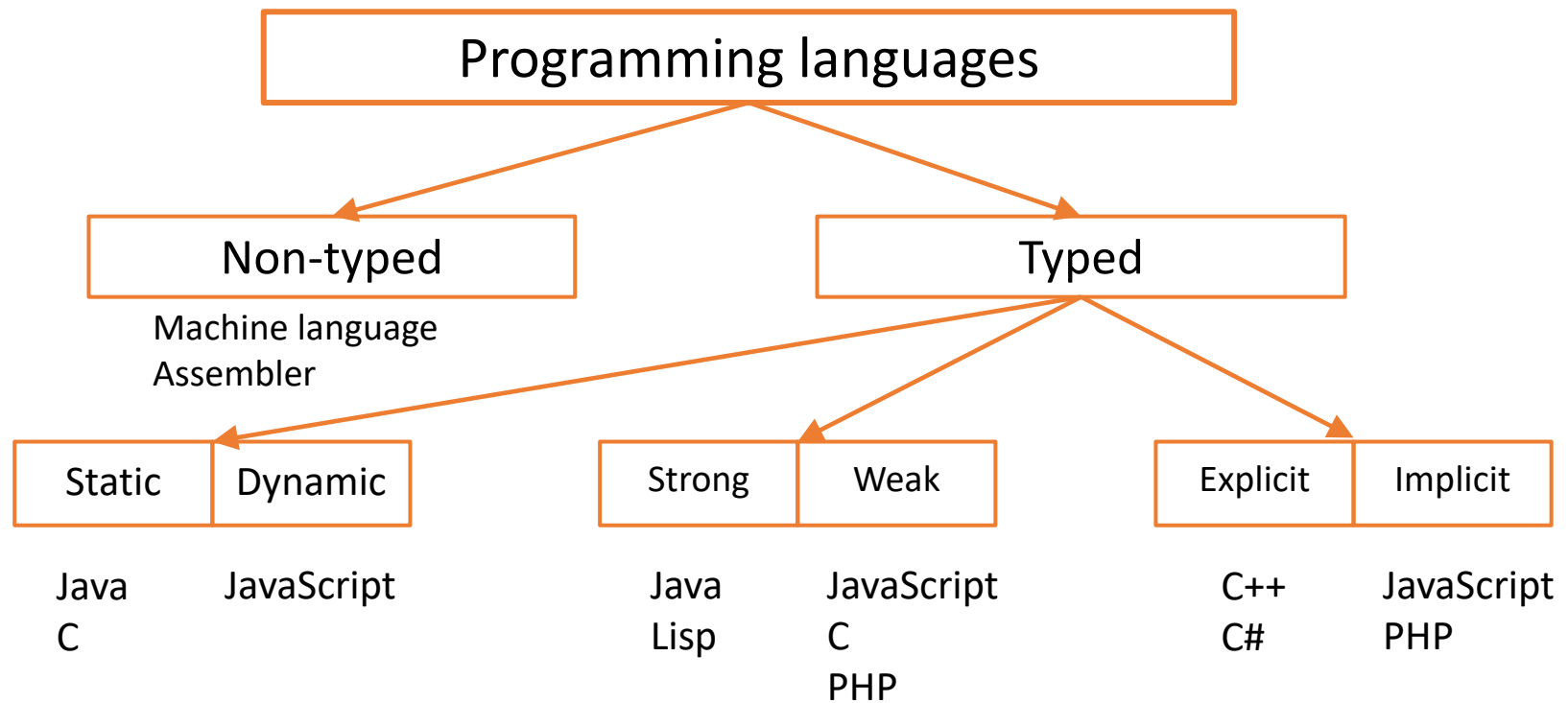
The **concept of data type** appeared in high-level programming languages as a reflection of the fact that data:

- ♦ Have **different sets of possible values**
- ♦ **Stored** in computer memory **in different ways**
- ♦ Require **different size of memory**
- ♦ Are processed using **different processor instructions**



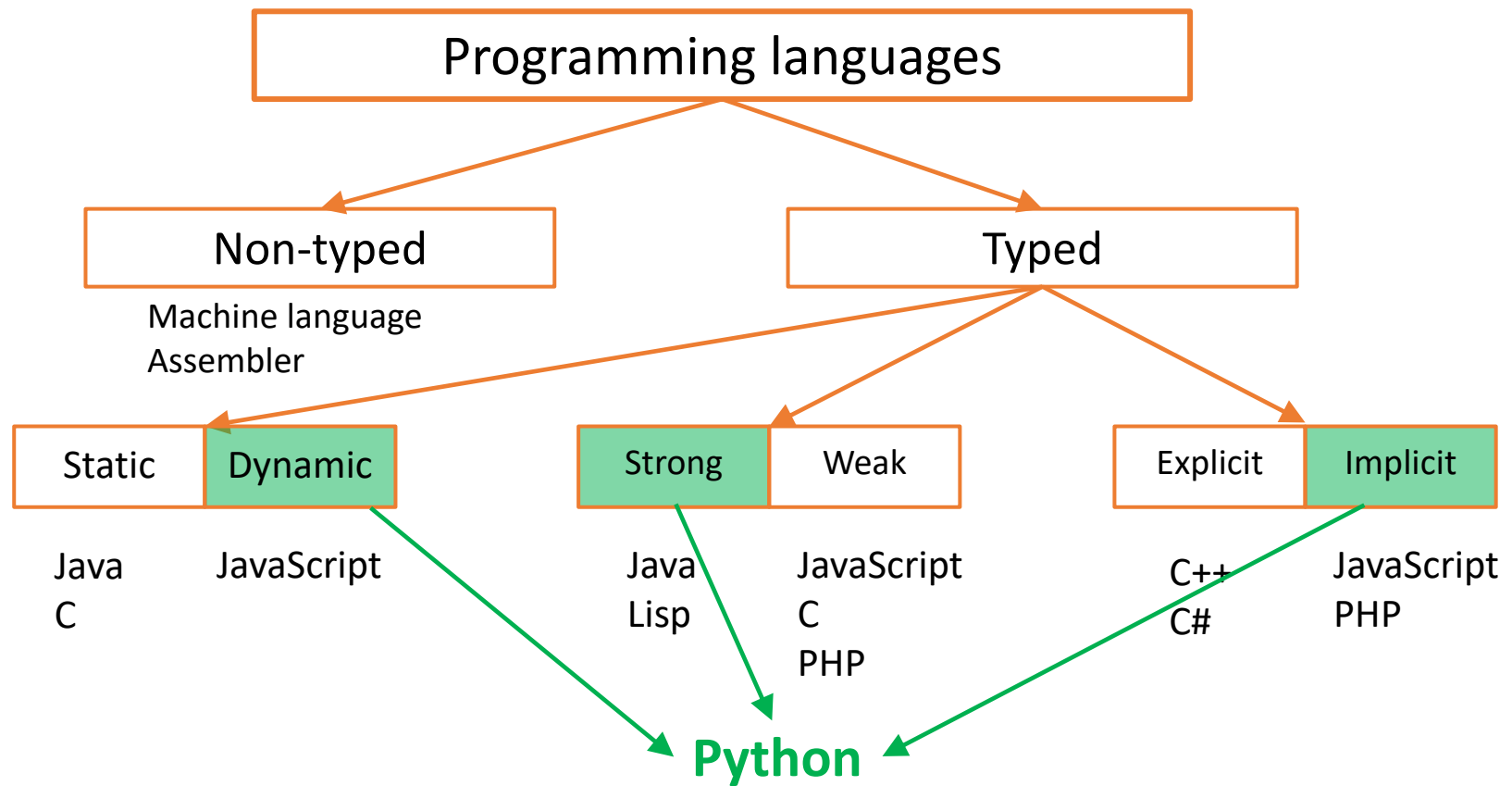
# Basic principles of data type concept

- ✦ **Any data type defines a range of values**
- ✦ **The type of any constant, variable or expression can be determined by its view or by its description.** There is no need to do any calculations for this.
- ✦ **Each operation or function requires arguments and gives a result of a certain type.** If the operation allows arguments of different types (for example, «+»), then the type of result is regulated by well-defined rules.
- ✦ **The most important basic operations are comparison and assignment,** that is, checking the equality (and order in the case of ordered types) and the action of «setting equality» (assignment).



# Typed programming languages

- ♦ **Static** – types of variables and functions **are defined at the compilation stage**. Examples: C, Java.
- ♦ **Dynamic** – all types **are found out during program execution**. Example: JavaScript.
- ♦ **Strong** – the language **does not allow mixing different types** in expressions and **does not perform automatic implicit conversions**, for example, you cannot subtract a set from a string. Examples: Java, Lisp.
- ♦ **Weak** – **many implicit conversions are performed automatically**, even if a loss of precision can occur or the conversion is ambiguous. Examples: C, JavaScript, PHP.
- ♦ **Explicit** – **types** of new variables, functions and their arguments **must be specified explicitly**. Examples: C++, C#.
- ♦ **Implicit** – languages with implicit typing shift this task to the compiler/interpreter. Examples: JavaScript, PHP.



# Advantages & disadvantages

## Non-typed languages. Advantages.

- ✦ You can write at an extremely low level, and the compiler / interpreter will not interfere with any type checks. **The developer can perform any operations on any kind of data.**
- ✦ Code is rather **more efficient**.
- ✦ Transparency of instructions. Usually (with knowledge of the language) there is no doubt what this or that code does.

## Non-typed languages. Disadvantages.

- ✦ **Complexity.** It is inconvenient to represent composite data, such as lists, strings, structures.
- ✦ No checks. **Any meaningless actions**, such as subtracting an array pointer from a character, **will be considered completely normal**. It's hard to spot errors.
- ✦ **Low level of abstraction.** Working with any complex data type is no different from working with numbers, which of course will create a lot of difficulties.

# Advantages & disadvantages

## Static. Advantages.

- ✦ Types are checked only once - at the compilation stage. **You don't need to check all the time** if we are trying to divide a number by a string, etc.
- ✦ **Execution speed.**
- ✦ Allows you **to detect some errors already at the compilation stage** (especially in combination with strong typing).
- ✦ **Speeding up development** with IDE (Integrated development environment) support (eliminating misfitting types).

## Dynamic. Advantages.

- ✦ **It is convenient to describe generalized algorithms** (for example, sorting an array that will work for integers, for real numbers, for lists, and for strings).
- ✦ **Easy to learn** - languages with dynamic typing are usually very good for starting programming.
- ✦ The simplicity of creating universal collections - meshes of everything.

# Advantages & disadvantages

## Strong. Advantages.

- ✦ **Reliability** — an exception or compilation error, instead of misbehaving.
- ✦ **Speed** - There are no **hidden conversions** that **can be** quite **expensive**. All conversions must be written explicitly, so the programmer always knows which part of the code can be slow.
- ✦ **Understanding** the program's work — instead of implicit type conversion, the programmer writes everything himself, which means he understands that comparing strings and numbers is not magic.
- ✦ **Definiteness** - it is **precisely known what type is converted to what type**.

## Weak. Advantages.

- ✦ It is convenient to use mixed expressions (for example, from integers and real numbers).
- ✦ The programmer can ignore typing and focus on the task.
- ✦ The code is shorter.

# Advantages & disadvantages

## Explicit. Advantages.

- ◆ Each function has a **signature** (for example, *int add (int, int)*), which **allows you to easily understand what the function does**.
- ◆ The programmer at the beginning defines what type of values can be stored in a variable and **doesn't need to remember this**.



"Now! That should clear up a few things around here!"

## Implicit. Advantages.

- The code is shorter - `def add (x, y)` versus `int add (int x, int y)`.
- **Resistant to change**. For example, if a temporary variable has the same type as a function argument, then in an explicitly typed language, changing the type of the input argument requires changing the type of the temporary variable.



JavaScript	- Dynamic	Weak	Implicit
Ruby	- Dynamic	Strong	Implicit
Python	- Dynamic	Strong	Implicit
Java	- Static	Strong	Explicit
PHP	- Dynamic	Weak	Implicit
C	- Static	Weak	Explicit
C++	- Static	Weak	Explicit
Perl	- Dynamic	Weak	Implicit
Objective-C	- Static	Weak	Explicit
C#	- Static	Strong	Explicit
Haskell	- Static	Strong	Implicit
Common Lisp	- Dynamic	Strong	Implicit
D	- Static	Strong	Explicit
Delphi	- Static	Strong	Explicit

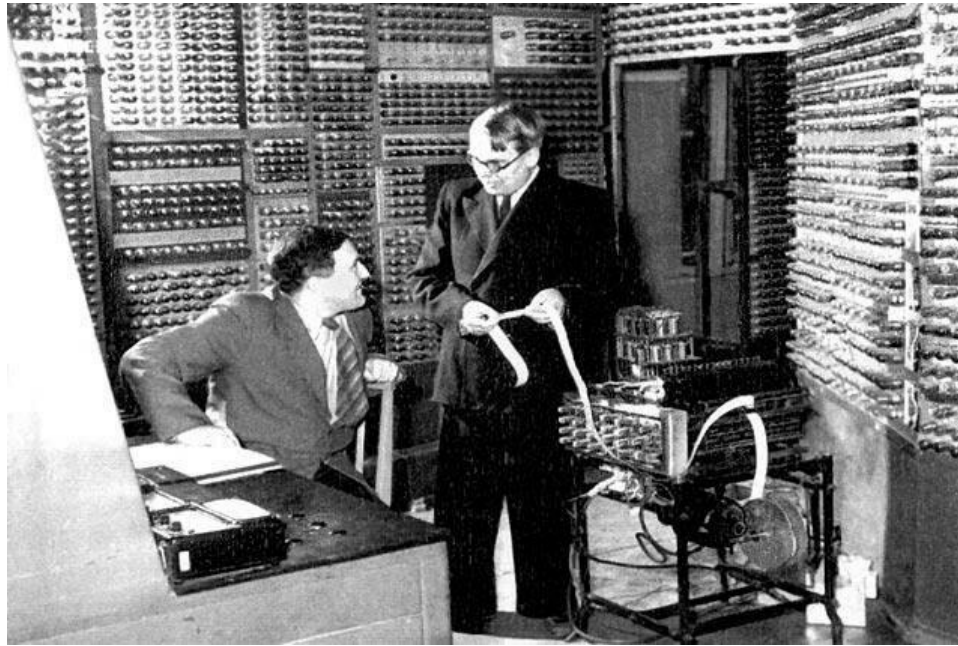
## A bit of history of programming and computers

[https://youtube.com/watch?v=aual\\_Ek0DQ&si=Oa4USvnyHPSDtVcw](https://youtube.com/watch?v=aual_Ek0DQ&si=Oa4USvnyHPSDtVcw)





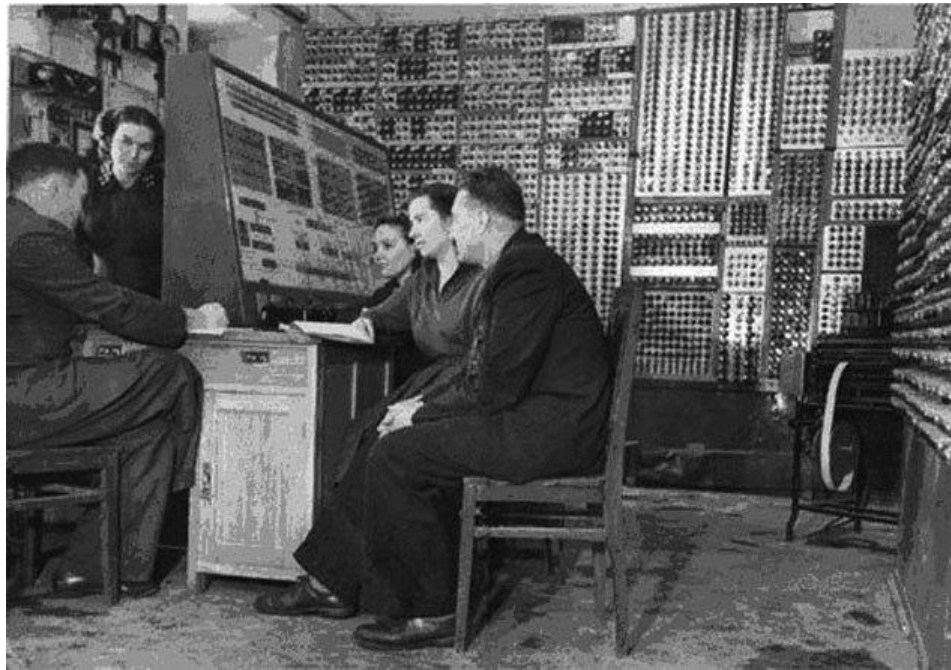
## History of computers in the USSR



This is what the first Soviet computer MESM (small electronic calculating machine) looked like "from the inside".

All the circuits of this machine were not in racks or cabinets, as was customary later, but were hung on the walls of the room.

## History of computers in the USSR



Large Electronic Calculating Machine (BESM)

- [Cartoon "Kitty"](https://www.youtube.com/watch?v=huyOoYRuqLQ) (<https://www.youtube.com/watch?v=huyOoYRuqLQ>)

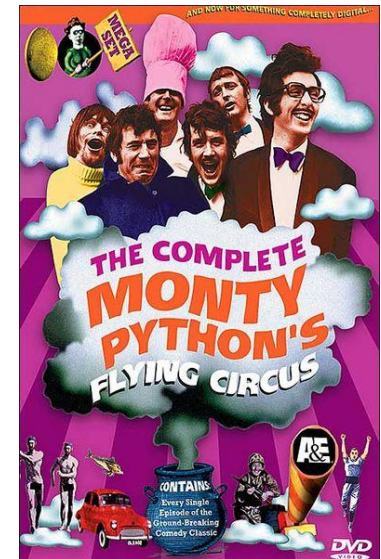




# Python

- 1990r. – dutch mathematician Guido van Rossum developed the Python
- 1991r. – first support of OOP
- 1994r. - Python 1.0
- 2000r. – Python 2.0
- 2003r. –Python 3.0
- 2014r. – Python 3.4
- ...
- 2023r. –Python 3.12

The language is named after the popular British comedy series of the 1970s, Monty Python's Flying Circus.

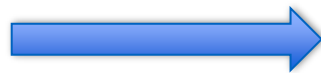


# Python is an interpreted language.

## Interpreted vs Compiled

**Compiler** – a program that translates source code written in a high-level programming language to the machine instructions.

*print(2\*‘Hello world! ’)*



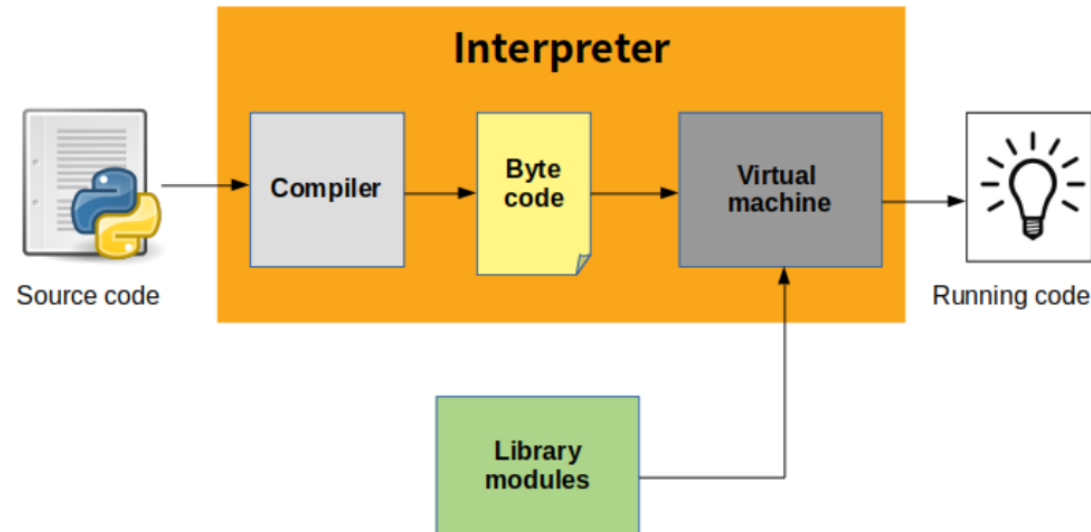
048660:	02	00	00	C2	04	00	8D	84	
048668:	24	1C	01	00	00	68	00	01	
048670:	00	00	8D	4C	24	1C	50	8D	
048678:	54	24	1C	51	8D	44	24	1C	
048680:	52	50	8D	4C	24	30	68	00	
048688:	01	00	00	51	56	FF	15	D8	

The simplest interpreter reads and executes program text **command by command**.

### Most **interpreters**

- first translate the entire program into an intermediate representation (bytecode),
- then execute (interpret) command by command using a virtual machine.

# Python is an interpreted language.



**Bytecode** is a compact representation of a program that has already passed syntactic and semantic analysis. It explicitly encodes types, scopes, etc. It **is a low-level machine-independent code** and provides portability of programs.

However, since Python does not create binary machine code, **programs written in it more often run slower** than their counterparts written in compiling languages such as C.

The virtual machine (non the microprocessor) executes instructions.



# Advantages and disadvantages

```
graph TD; Libraries --- CI[Component integration]; Libraries --- OS[Open source]; Libraries --- P[Portability]; Libraries --- HSD[High speed of development]; Libraries --- SQ[Software Quality]; Libraries --- LRS[Low running speed];
```

Component integration

Software  
Quality

Open source

High speed of development

Libraries

Portability

Low running  
speed

# Basic data types

[illegible]

## Integers by default could be any size

```

isinstance(True, int) # True
isinstance(False, int) # True
issubclass(bool, int) # True

```

# Operations with integer & float

operator/ function	description
$X + Y$	addition
$X - Y$	subtraction
$X * Y$	multiplication
$X / Y$	division
$X // Y$	Division rounded down (to the nearest smaller)
$X \% Y$	Remainder of division
<code>divmod(X,Y)</code>	pair ( $X // Y$ , $X \% Y$ )
$-X$	Change the sign of a number
<code>abs(X)</code>	absolute value
$X ** Y$	exponentiation
<code>pow(X, Y[, Z])</code>	X to the power of Y, [ X to the power of Y modulo Z: calculation of the remainder of dividing the number X in degree Y by the number Z (module)]
<code>X.as_integer_ratio()</code>	Returns a pair of integers whose ratio is equal to real X

# Types conversion

operator/ function	description
<code>X.is_integer()</code>	Check whether the value of float X is an integer
<code>int([string-number], [number system base])</code>	Converts a string-number in the system from 2 to 36 (by default in decimal) to an integer in decimal
<code>bin(X)</code>	Converts an integer to a binary string
<code>hex(X)</code>	Converts integer to hexadecimal string
<code>oct(X)</code>	Converts integer to octal string

## Boolean

operator	description
X and Y	Logic “and”. The condition will be true if both operands are true.
X or Y	Logic “or”. If at least one of the operands is true, then the whole expression will be true.
not X	Logic “not”. Change the value to the opposite.

## Containment

a in B	Returns true if the element a is present in the sequence B, otherwise returns false.
a not in B	Returns true if element a is not in the sequence B.

## Identicalness

X is Y	Returns true if X and Y point to the same object.
X is not Y	Returns true if X and Y point to the different objects.

# Comparison

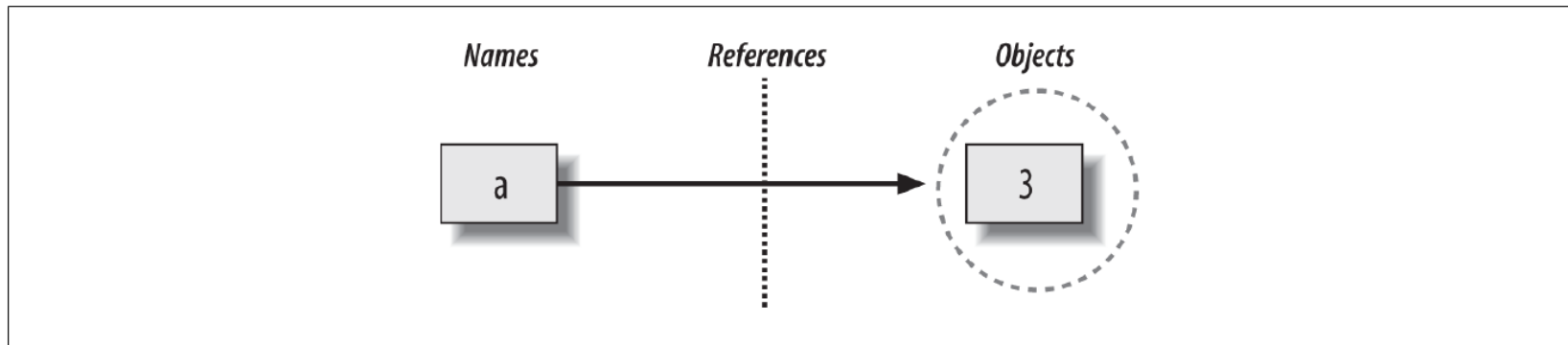
operator	description
<code>X == Y</code>	Checks if X and Y are equal. If yes, then the condition becomes true.
<code>X != Y</code>	Checks if X and Y are equal. If not, then the condition becomes true.
<code>X &gt; Y</code>	Checks if the value of X is greater than the value of Y. If yes, then the condition becomes true.
<code>X &lt; Y</code>	Checks if the value of X is less than the value of Y. If yes, then the condition becomes true.
<code>X &gt;= Y</code>	Checks the value of X more than or equal to the value of Y. If yes, then the condition becomes true.
<code>X &lt;= Y</code>	Checks the value of X less than or equal to the value of Y. If yes, then the condition becomes true.

# Assignment operator

operator	description
<code>spam = 'Spam'</code>	Canonical. Associates a variable (or data structure element) with a single object.
<code>spam, ham = 'yum', 'YUM'</code>	Tuple assignment (positional)
<code>[spam, ham] = ['yum', 'YUM']</code>	List assignment (positional)
<code>a, b, c, d = 'spam'</code>	Sequence assignment, generalized. Associates a tuple of names with a string of characters: the name <b>a</b> is assigned the symbol <b>'s'</b> , the name <b>b</b> is assigned the symbol <b>'p'</b> , etc.
<code>string = 'SPAM'</code> <code>a, b, c, d = string</code>	You need the same number of elements on both sides. <code>a, b, c, d == ('S', 'P', 'A', 'M')</code> .
<code>a, *b = 'spam'</code>	Extended sequence unpacking operation. Variable <b>a</b> will be assigned the first character from the right string right, and variable <b>b</b> - the remainder of the string, that is, variable <b>a</b> will be set to <b>'s'</b> , and variable <b>b</b> to <b>'pam'</b> . As a result, we get a simple alternative to the operation of extracting slices.
<code>a, b, c = string[0], string[1], string[2:]</code>	<code>a, b, c = ('S', 'P', 'AM')</code>
<code>spam = ham = 'lunch'</code>	Group assignment of one value. Equivalent - <code>ham = 'lunch'</code> и <code>spam = ham</code> .
<code>spams += 42</code>	Combined Assignment Statement (equivalent <code>spams = spams + 42</code> )
<code>x, y, z = range(3)</code>	<code>X = 0, y = 1, z = 2</code>

# Variables. Dynamic typing.

Python data types are automatically determined at run time, and not as a result of declarations in program code. **Variables are created during the assignment operation**; they can refer to objects of any types and must be assigned some values before they can be called.

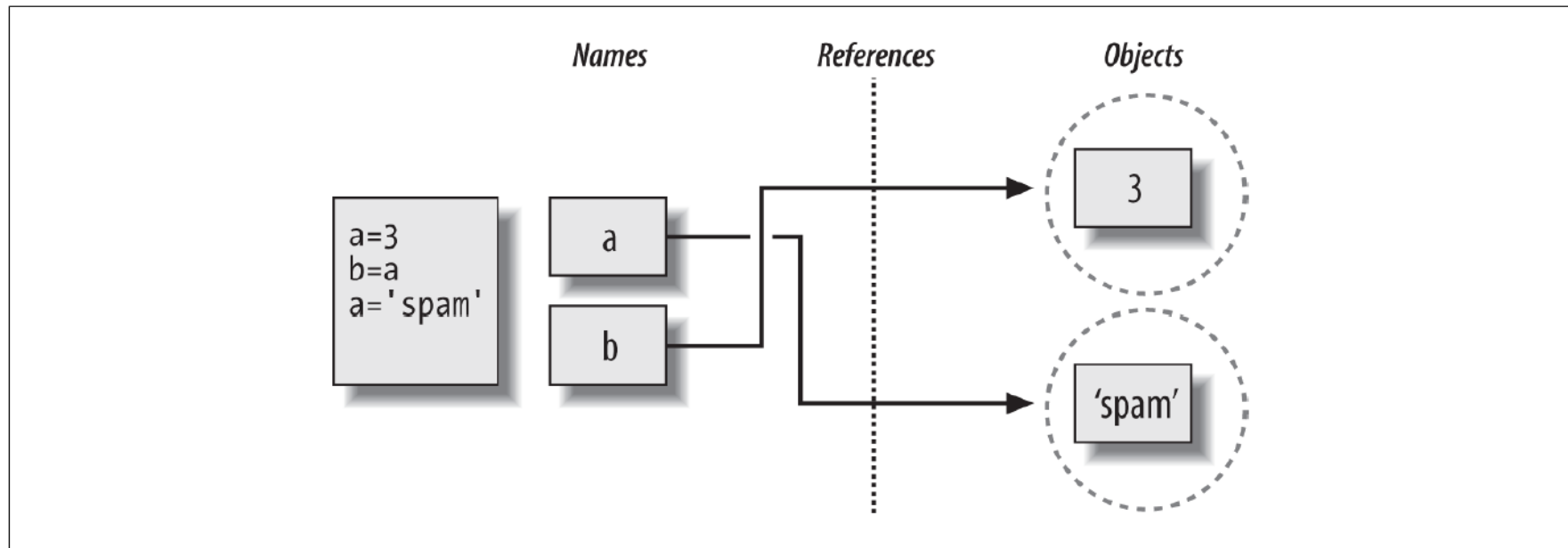
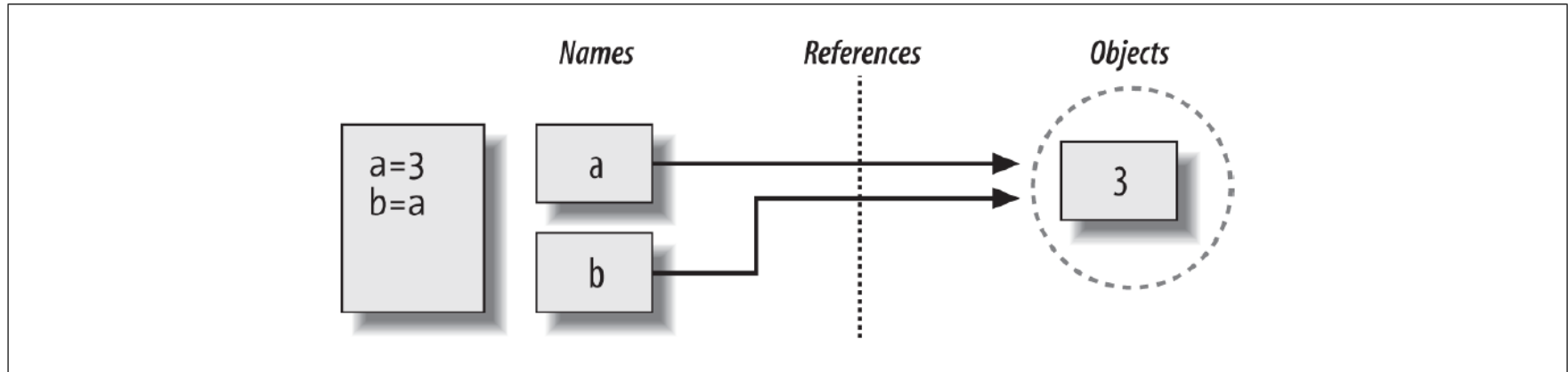


- **Variables** - are records in the system table.
- **Objects** are areas of memory with enough capacity to represent the values of these objects. **Each object has two standard fields: a type descriptor** used to store information about the type of object, and a **reference counter** used to determine when the memory occupied by the object can be freed.
- **Links** are pointers to objects.



# Shared links

- multiple names refer to the same object



# If statements (selection)

```
if <test1>:           # If statement with conditional expression test1
    <statements1>     # statements
elif <test2>:         # non-obligatory elif
    <statements2>
else:                 # non-obligatory else
    <statements3>
```

The *if* statement is used to check conditions:

if the *test1* condition is true, statement1 block is executed,

1) otherwise, the *test2* condition is checked and, if true, *statements1* are executed (there may be several *elif* blocks), otherwise statements3 are executed.

The «*elif*» (else if) and «*else*» are optional.

Example. Guess the number.

```
number = 23
```

```
guess = int(input(Input integer: '))
```

```
if guess == number:
```

```
    print('Congratulations, you are right,') # block begin
```

```
    print(although you won no prize!) # ,block ends
```

```
elif guess < number:
```

```
    print(No, the number guessed is bigger than this. ')
```

```
    # Inside the block, you can do whatever you want....
```

```
else:
```

```
    print(No, the number guessed is smaller than this.)
```

```
    # guess must be greater than number to get here
```

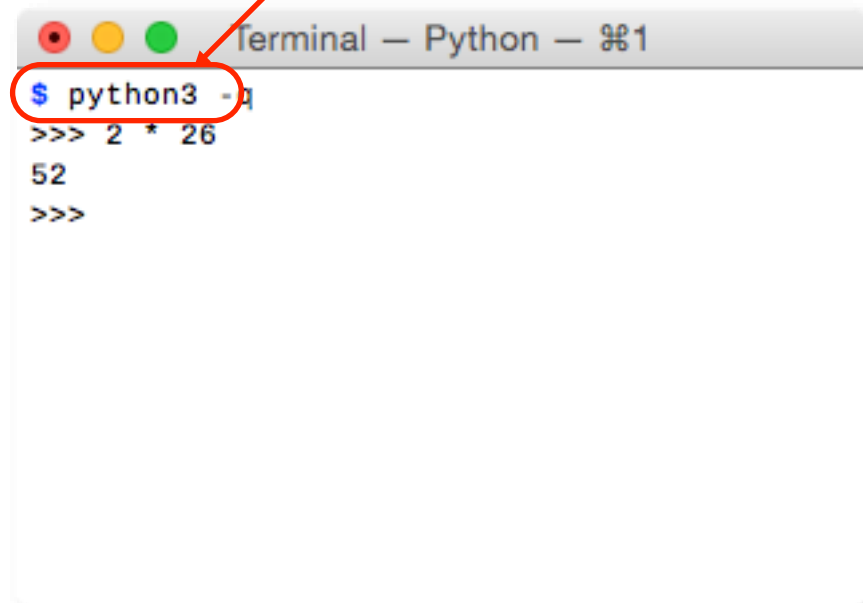
```
print('The end')# This last expression is always executed after the statement if is  
executed.
```

# Running Python Interactively

♦ An interactive language is basically a fancy calculator

1. start Python
2. Python prints a prompt to let you know it is ready
3. type an expression
4. Python evaluates the expression and prints the result
5. go back to step 2

*OS command to start Python*

A screenshot of a macOS Terminal window titled "Terminal — Python — %1". The window shows the command `$ python3 -i` entered at the prompt, which is circled in red. Below this, the Python prompt `>>>` is shown, followed by the expression `2 * 26`, the result `52`, and another `>>>` prompt. A red arrow points from the text "OS command to start Python" to the `python3` command in the terminal.

```
$ python3 -i
>>> 2 * 26
52
>>>
```

# Input & output

*print('Hello, world!')*

*print('2 + 3 =', 2 + 3)*

*print(1, 2, 3, 4, sep=' + ', end="")*

*print(' = ', 1 + 2 + 3 + 4, sep="")*

*# sep & end could be any string.*

*Input()* – reads a line from the console, to finish entering the line, press Enter

*name = input()*

*n = int(input())*

# The Zen of Python

```
Python Console
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Illustrations for each point of the Python philosophy can be found here:  
<http://www.russianlutheran.org/python/zen/zen.html>

## Yandex Contest

Message	Short	Meaning	Possible reason
OK	OK	The solution accepted	The program works correctly on the built set of tests
Compilation error	CE	Compilation error	1. The program made a syntax or semantic error 2. The language is incorrect
Wrong answer	WA	Wrong answer	1.error in the program 2.incorrect algorithm
Presentation error	PE	The testing system cannot verify the output, as its format does not match the one described in the task conditions	1.Incorrect output format 2.The program does not print the result 3.Extra output
Time-limit exceeded	TL	Программа превысила установленный лимит времени	1. ошибка в программе 2. неэффективное решение
Memory limit exceeded	ML	The program has exceeded the memory limit	1.an error in the program (for example, infinite recursion) 2.inefficient solution
Run-time error	RT	The program terminated with a non-zero return code	1. runtime error 2. a program in C or C ++ does not end with the operator return 0 3. a nonzero return code is specified explicitly

## Useful links:

- <https://www.python.org/>
- <https://docs.python.org/3/>
- PyCharm (Education Edition) - <https://www.jetbrains.com/pycharm-edu/>
- Tutorial по Pycharm - <http://pythonworld.ru/osnovy/pycharm-python-tutorial.html>
- Mark Lutz. “Programming Python”
- <https://pythontutor.ru/> (in Russian)
- <http://wiki.cs.hse.ru>



# Useful links:

- “Introduction to Computer Science and Object-Oriented Programming Python.” (<https://stepik.org/course/56730/promo> )
- “Python. Functional Programming.” (<https://stepik.org/course/2057/promo>)
- “Introduction to Python” (<https://app.datacamp.com/learn/courses/intro-to-python-for-data-science> )