Basic Algorithms & Programming

4TPU140U

2022 - rev. 0.5

Adrien Boussicault
Creative Commons License BY-NC-SA 4.0

Outline

1. Introduction
2. What is a computer? What is a program?
3. My first programs
4. How to run a program: Interpreter, Compiler and executable
5. Python syntax
6. Variables, assignments and memory
7. Types and Native operators
8. Control and loop structures
9. List and dictionary
10. Functions

Outline

1 Introduction
2 What is a computer? What is a program?
3 My first programs
4 How to run a program: Interpreter, Compiler and executable
5 Python syntax
6 Variables, assignments and memory
7 Types and Native operators
8 Control and loop structures
9 List and dictionary
10 Functions

Goals and some informations

- Goals:
  - Learn how to think with algorithms.
  - Translate into informatic programs.
- Tools:
  - Python programming language.
  - text editor **gedit**, **vim**, **Emacs** – programming environment.
  - Operating System **Unix – Linux**.
- WebSite:
  - `https://www.labri.fr/perso/boussica/enseignements_courants_fr.html`
  - `https://pythontutor.com/visualize.html`
- Book:
  - *Learning Python*, Mark Lutz, 5th Edition, O'REILLY

Organization and evaluation

The evaluation is composed of 4 marks. The final course mark is the average of that 4 marks.

I will publish a project with 4 parts. Each part of the project will give you a mark.

The rules of the project are the following:

- the project can be made in group,
- you can send me any number of revision you want,
- for each revision you send me, I will send you a review with a mark,
- the grades always go up. Reviews sould get you the maximum score of 20/20,
- you can get inspiration from any source, but you must cite your source,
- you can use existing source code if the code is free (see. Free Software Foundation for the definition of a free code/software).

Outline

1 Introduction

2 What is a computer? What is a program?

3 My first programs

4 How to run a program: Interpreter, Compiler and executable

5 Python syntax

6 Variables, assignments and memory

7 Types and Native operators

8 Control and loop structures

9 List and dictionary

10 Functions

A computer for an user point of view

A computer is an electric machine with the following devices:

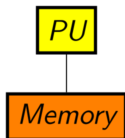
- screen to display informations,
- keyboard and mouse to interact with the machine,
- hard disk to store informations structured in files.

An User can ask to the computer to perform some tasks and obtain some informations.

A program is a special file containing the description of the task to perform. The computer performs that task if the user “double click” on that file with the mouse (it is called “launching the program”).

A computer for a computer scientist

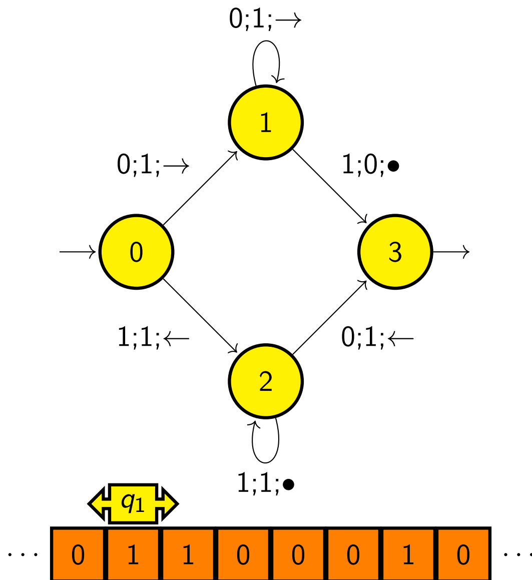
A computer is a processing unit (PU), that use a sequence of 0 and 1, called the memory, to read/edit that sequence.



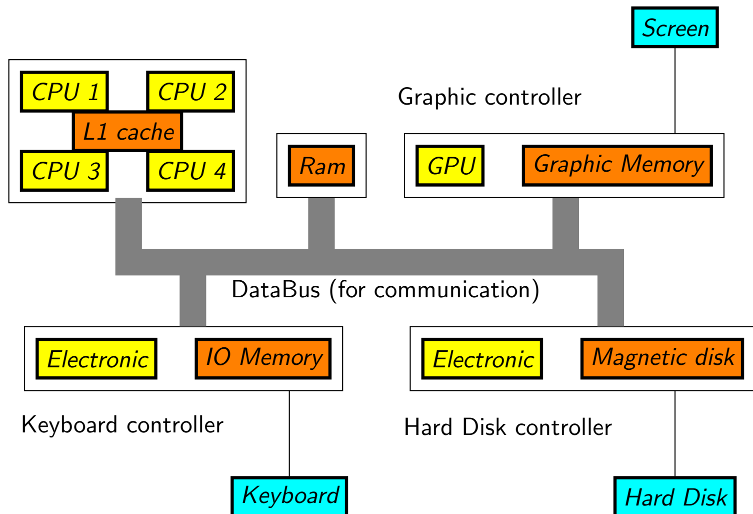
A program is a sequence of instructions, explaining to the Processing Unit how to read and edit the memory. Usually, the program is stored in the memory.

A program is like a cooking recipe. Recipient are replaced by memory, and foods by the sequences of 0 and 1 stored in the memory.

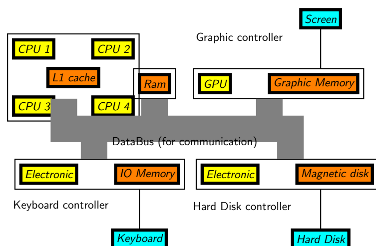
An example of theoretical computer: The Turing machine.



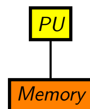
An example of computer architecture.

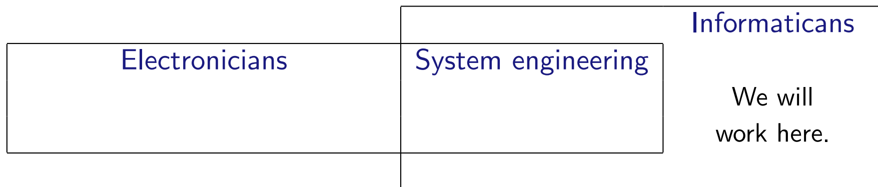


From electronicians to Informaticians. What is the subject of that course ?



Operating System
(Linux, Windaube)





Outline

1 Introduction
2 What is a computer? What is a program?
3 My first programs
4 How to run a program: Interpreter, Compiler and executable
5 Python syntax
6 Variables, assignments and memory
7 Types and Native operators
8 Control and loop structures
9 List and dictionary
10 Functions

Hello world printed on the terminal.

```
1 print('Hello World !')
```

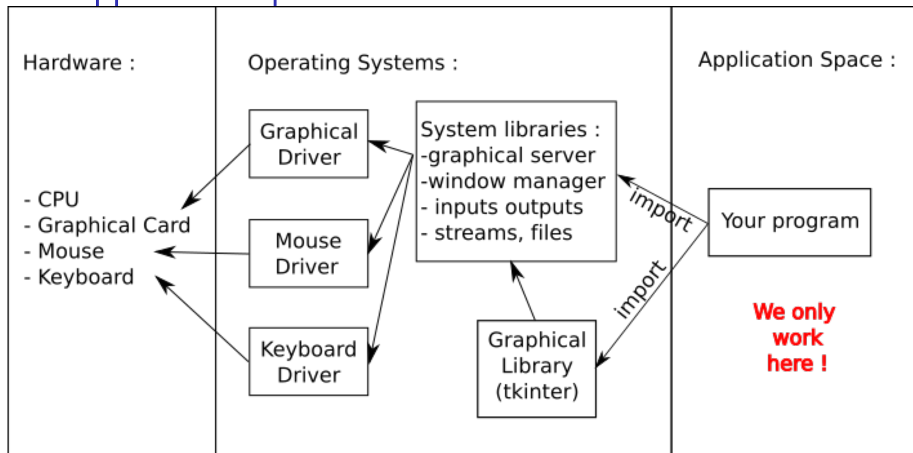
Computing some mathematical formulas.

```
1  from math import sqrt
2
3  value = 2348
4  result = sqrt(value)
5
6  print('The square root of' + str(value) + ' is ' + str(result))
7
8  result = 2 * result
9
10 print('The content of result in the memory has been edited.')
11 print('It was doubled.')
12 print(result)
```

Hello world displayed on a window with some drawings.

```
1  from tkinter import *
2  Fenetre = Tk()
3
4  zone_dessin = Canvas(
5    Fenetre,width=500,height=500,bg="white",bd=8
6  )
7  zone_dessin.pack()
8  zone_dessin.create_text(100,100, text="Hello World")
9  zone_dessin.create_line(0,0,500,500,fill="red",width=4)
10 zone_dessin.create_rectangle(150,150,350,350)
11 zone_dessin.create_oval(150,150,350,350,fill="orange",width=4)
12
13 Fenetre.mainloop()
```

The application space



The object of this course is to write programs in python in the application space.

The operating system allows to reduce the computer architecture to the simple PU–memory abstraction. This simplification simplifies the task of programming.

Outline

1. Introduction
2. What is a computer? What is a program?
3. My first programs
4. How to run a program: Interpreter, Compiler and executable
5. Python syntax
6. Variables, assignments and memory
7. Types and Native operators
8. Control and loop structures
9. List and dictionary
10. Functions

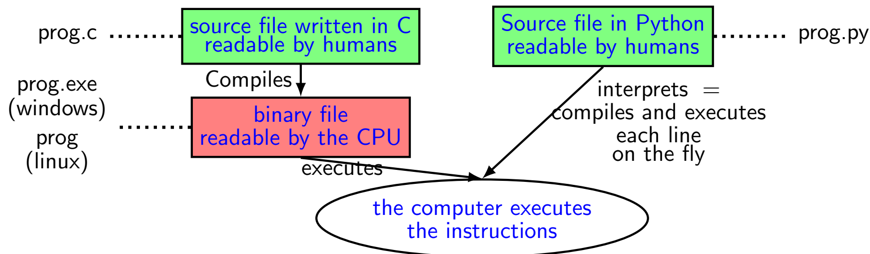
Executing a Python program in the terminal

- A program written in python, is ONLY readable for an human.
- To execute this program, we need to use another program, called an interpreter, to convert each python line to instructions that the Processing Unit can understand.
- Each line is converted on the fly in instructions readable by the PU.
- The interpreter for python programs is `python3`.
- To execute our program in a terminal, we need to open a terminal an execute the following command line:

```
1 python3 hello_world.py
```

The PU runs `python3` that opens `hello_wordl.py`. The PU converts all the lines in native PU instructions and runs them.

Compiled language vs Interpreted language



Executing a program written in C on the terminal:

```
1  # The PU compiles the
2  # program 'prog.c' to
3  # obtain the binary 'prog'
4  gcc prog.c -o prog
5
6  # The PU executes the binary
7  ./prog
```

Executing a program written in Python on the terminal:

```
1  # The PU interpret the
2  # program 'hello_world.py'
3  pyrhon3 hello_world.py
```

Outline

1 Introduction

2 What is a computer? What is a program?

3 My first programs

4 How to run a program: Interpreter, Compiler and executable

5 Python syntax

6 Variables, assignments and memory

7 Types and Native operators

8 Control and loop structures

9 List and dictionary

10 Functions

Python syntax

A python program is like a cookbook.

- a program must be read from bottom to top,
- each line contains an instruction and must be left aligned vertically.

Except in special cases (see control and loop structures), the computer executes the lines of a program from top to bottom, one line after another.

```
1 a = 5    # <- first instruction
2 a = a/2  # <- second instruction
3 print(a) # <- third instruction
```

```
1 a = 5
2    a = a/2  # <- Invalid syntax!!
3 print(a)
```

All text present after a # is ignored. We call that text a commentary.

Outline

1. Introduction
2. What is a computer? What is a program?
3. My first programs
4. How to run a program: Interpreter, Compiler and executable
5. Python syntax
6. Variables, assignments and memory
7. Types and Native operators
8. Control and loop structures
9. List and dictionary
10. Functions

Warnings about the memory model of this Lecture.

The memory model we present in this course doesn't correspond exactly to the Python memory management.

This model is compatible with the way Python use memory. By using that model you will not make mistakes.

The model we will present is a model that better fit the following languages: C/C++, Java, Javascript, pascal, and others.

It's a compromise to allow you to better learn other languages

To know more details about why the present model differs from Python memory management, consult the slides in annexes.

The memory

The memory is a sequence of bits (0 or 1). It looks like:

$$...1100010100100101001111011110...$$

The hexadecimal code is a way to code 4 bits into a letter:

$$\begin{array}{llll} 0000 \Longrightarrow 0 & 0100 \Longrightarrow 4 & 1000 \Longrightarrow 8 & 1100 \Longrightarrow c \\ 0001 \Longrightarrow 1 & 0101 \Longrightarrow 5 & 1001 \Longrightarrow 9 & 1101 \Longrightarrow d \\ 0010 \Longrightarrow 2 & 0110 \Longrightarrow 6 & 1010 \Longrightarrow a & 1110 \Longrightarrow e \\ 0011 \Longrightarrow 3 & 0111 \Longrightarrow 7 & 1011 \Longrightarrow b & 1111 \Longrightarrow f \end{array}.$$

For example, the following sequence of bits:

$$\ldots \underbrace{1100}_{c}\underbrace{0101}_{5}\underbrace{0010}_{2}\underbrace{0101}_{5}\underbrace{0011}_{3}\underbrace{1101}_{d}\underbrace{1110}_{e}\ldots$$

is coded by:

$$\ldots c5253de \ldots$$

Addresses

A sequence of 8 bits is called a Byte. In informatics it is the unit of measurement.

All the memory is splited in Bytes. The address of a Byte is its position in the memory.

For example, in the memory

$$aef0a1023e\cdots$$

- the address of the first byte ae is 0,
- the address of the second byte $f0$ is 1,
- the address of the third byte $a1$ is 2.

Addresses always starts at 0.

Small memory space

Addresses allow the computer to know the location where informations are stored in memory.

A *"small memory space"* is a sequence of Bytes that allows to code a memory address or a small real/integer. Usually addresses and native integers/reals are coded with 8 bytes.

Variables and assignments (1/3)

Memory manipulation is done with variables.

A variable is the name we give to a particualr small memory space (64 bits = 8 bytes) in the memory.

When launching a program, a memory space is assigned to the program.

To reserve a small memory space in the program's memory, we use a variable in the following way:

```
1 my_varaible = 5
```

With that instruction, the Processing Unit:

- reserves a small space (64 bits) in the memory,
- gives the name `my_variable` to that small space,
- stores in that small space the value of 5 in binary format.

Variables and assignments (1/3)

Python program:

```
1 my_variable = 5
```

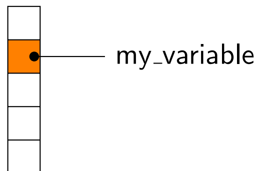
Memory:

Variables and assignments (1/3)

Python program:

```
1 my_variable = 5
2
```

Memory:

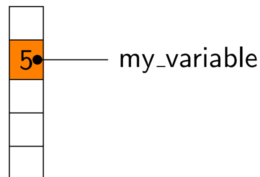


Variables and assignments (1/3)

Python program:

```
my_variable = 5

```

Memory:



Variables and assignments (2/3)

In the following program,

```
1 my_text = 'Hello World'
```

the Processing Unit:

- reserves a small space (64 bits) in memory,
- gives the name `my_text` to that small space,
- allocates a big space containing “Hello World”. That space containins at least 11 Bytes, (and it is not possible to store that quantity of data inside the memory of a variable)
- in the small space whose name is `my_text`, the Processing Unit stores the address in memory where “Hello World” is located.

Variables and assignments (2/3)

Python program:

```
1 my_text = 'Hello World'
```

Memory:

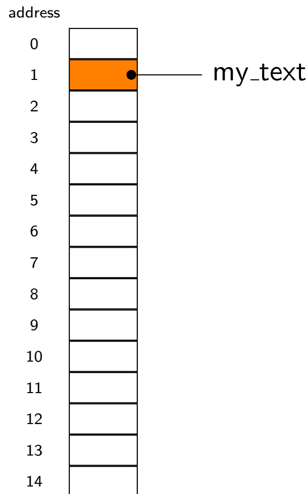
address	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

Variables and assignments (2/3)

Python program:

```
1 my_text = 'Hello World'
2
```

Memory:

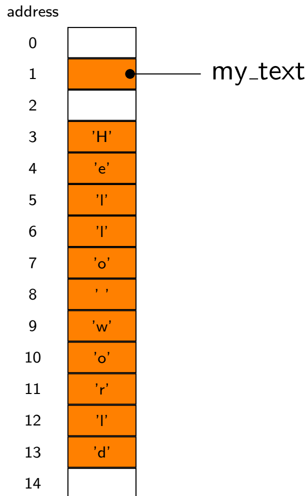


Variables and assignments (2/3)

Python program:

```
1 my_text = 'Hello World'
2
```

Memory:

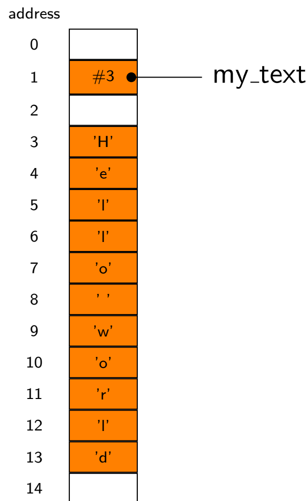


Variables and assignments (2/3)

Python program:

```
1 my_text = 'Hello World'
2
```

Memory:

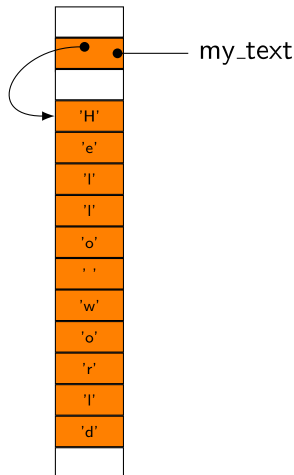


Variables and assignments (2/3)

Python program:

```
1 my_text = 'Hello World'
2
```

Memory:



Variables and assignments (3/3)

The = operators in informatics is different form the = operator in mathematics.

In informatics, the = operator is an assignment.

In mathematics it is an equality assertion.

Outline

1. Introduction
2. What is a computer? What is a program?
3. My first programs
4. How to run a program: Interpreter, Compiler and executable
5. Python syntax
6. Variables, assignments and memory
7. Types and Native operators
8. Control and loop structures
9. List and dictionary
10. Functions

Python types (1/2)

The type of a variable is the type of the information we can access with that variable.

The different native types are the following:

Type	Example
Numbers	124 3.02 9999L 4.0e+2 3+4j
String	'spam' “d'guido”
List	[1,[2,'trois'],4]
Tuples	(1,'spam',4,'U',0)
Dictionary	{'food' : 'jam', 'taste': 'miam'}

Python types (2/2)

The type of a variable can be obtained with the function 'type':

```
1  a = 124
2  print(type(a))  # integer type       => <class 'int'>
3  b = 124.0
4  print(type(b))  # real type (float) => <class 'float'>
5  c = 'coucou'
6  print(type(c))  # string type       => <class 'str'>
7  d = [1,2,3]
8  print(type(d))  # list type         => <class 'list'>
9  e = (1,2,3)
10 print(type(e))  # tuple type        => <class 'tuple'>
11 f = {1:'soleil', 5:'pluie'}
12 print(type(e)) # dictionary type    => <class 'dict'>
```

Native Operators (1/3)

- Arithmetic:

```
1 print(3 + 5)               # Addition       =>          8
2 print('Hi ' + 'World')     # Concatenation  => 'Hi World'
3 print(3 - 5)               # Substraction   =>         -2
4 print(3 * 5)               # Multiplication =>         15
5 print(3 / 5)               # Division       =>        1.4
6 print(9 // 2) # Quotian of Euclidean division    =>    4
7 print(9 % 2)  # Remainder of Euclidean division =>    1
8 print(2 ** 3) # Power                            =>    8
```

- Comparison:

```
1 print(3 == 5)   # is equal             => False
2 print(3 < 5)    # is strictly lesser   => True
3 print(3 > 5)    # is strictly biger    => False
4 print(3 <= 5)   # is lesser or equal   => True
5 print(9 >= 2)   # is biger or equal    => False
6 print(9 != 2)   # is different         => True
```

Native Operators (2/3)

- Logic:

or	*True*	*False*
True	*True*	*True*
False	*True*	*False*

and	*True*	*False*
True	*True*	*False*
False	*False*	*False*

	True	*False*
not	*False*	*True*

```
1 print(True or False)  # Or operator  => True
2 print( not(True and False) ) #  => True
```

- Bitwise operation: On memory, a positive integer is encoded in binary code in the following way:

$$(01011)_{binary} = 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$$

```
1 print(5 << 2) # Bit shift of 5 by 2 to the left  => 20
2 print(5 >> 2) # Bit shift of 5 by 2 to the right => 1
3 # logical OR on each pair of the corresponding bits
4 print(5 | 3)  # => 7
5 # logical AND on each pair of the corresponding bits
6 print(5 & 3)  # => 1
```

Native Operators (3/3)

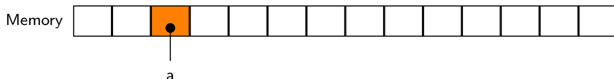
- Address comparator:

```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 print(a == b) # Compare data pointed by a and b => True
5 print(a is b) # Compare a and b addresses       => False
6 print(a == c) # Compare data pointed by a and c => True
7 print(a is c) # Compare a and c addresses       => True
```

Native Operators (3/3)

- Address comparator:

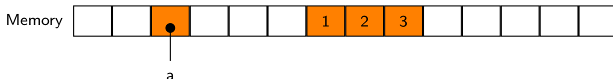
```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 print(a == b) # Compare data pointed by a and b => True
5 print(a is b) # Compare a and b addresses       => False
6 print(a == c) # Compare data pointed by a and c => True
7 print(a is c) # Compare a and b addresses       => True
```



Native Operators (3/3)

- Address comparator:

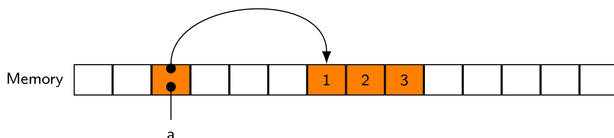
```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 print(a == b) # Compare data pointed by a and b => True
5 print(a is b) # Compare a and b addresses       => False
6 print(a == c) # Compare data pointed by a and c => True
7 print(a is c) # Compare a and b addresses       => True
```



Native Operators (3/3)

- Address comparator:

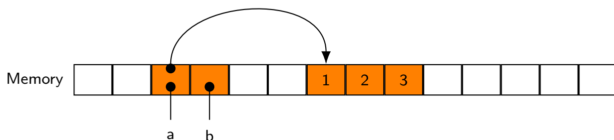
```
1 a ␣= [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 print(a == b) # Compare data pointed by a and b => True
5 print(a is b) # Compare a and b addresses        => False
6 print(a == c) # Compare data pointed by a and c => True
7 print(a is c) # Compare a and b addresses        => True
```



Native Operators (3/3)

- Address comparator:

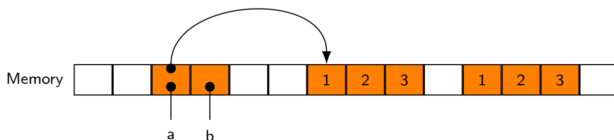
```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 print(a == b) # Compare data pointed by a and b => True
5 print(a is b) # Compare a and b addresses       => False
6 print(a == c) # Compare data pointed by a and c => True
7 print(a is c) # Compare a and b addresses       => True
```



Native Operators (3/3)

- Address comparator:

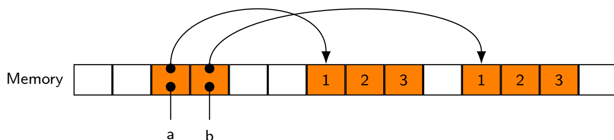
```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 print(a == b) # Compare data pointed by a and b => True
5 print(a is b) # Compare a and b addresses       => False
6 print(a == c) # Compare data pointed by a and c => True
7 print(a is c) # Compare a and b addresses       => True
```



Native Operators (3/3)

- Address comparator:

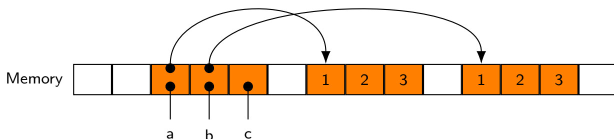
```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 print(a == b) # Compare data pointed by a and b => True
5 print(a is b) # Compare a and b addresses       => False
6 print(a == c) # Compare data pointed by a and c => True
7 print(a is c) # Compare a and b addresses       => True
```



Native Operators (3/3)

- Address comparator:

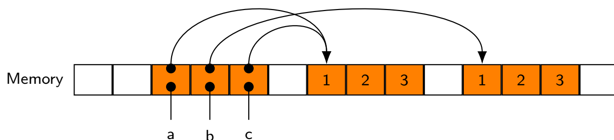
```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 print(a == b) # Compare data pointed by a and b => True
5 print(a is b) # Compare a and b addresses       => False
6 print(a == c) # Compare data pointed by a and c => True
7 print(a is c) # Compare a and b addresses       => True
```



Native Operators (3/3)

- Address comparator:

```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 print(a == b) # Compare data pointed by a and b => True
5 print(a is b) # Compare a and b addresses       => False
6 print(a == c) # Compare data pointed by a and c => True
7 print(a is c) # Compare a and b addresses       => True
```



Outline

1 Introduction

2 What is a computer? What is a program?

3 My first programs

4 How to run a program: Interpreter, Compiler and executable

5 Python syntax

6 Variables, assignments and memory

7 Types and Native operators

8 Control and loop structures

9 List and dictionary

10 Functions

Control structures or conditional jump (1/3)

Conditional jumps allow to choose between two sets of instructions to run according a preliminary condition.

The syntax is the following:

```
1 if CONDITION :
2   INSTRUCTIONS_TO_RUN_IF_CONDITION_IS_TRUE
3 else:
4   INSTRUCTIONS_TO_RUN_IF_CONDITION_IS_FALSE
```

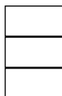
Indentation is important. All instructions with the same indentation are in the same block.

The blocks of instuctions `INSTRUCTIONS_TO_RUN_IF_CONDITION_IS_TRUE` is executed if CONDITION is True. Otherwise, the other block is executed.

Control structures or conditional jump (2/3)

Example:

```
1 a = 2
2 if(a % 2 == 0):
3     print('a is even')
4 else:
5     print('a is odd')
6 print('end')
```



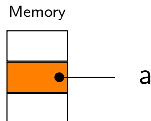
If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Control structures or conditional jump (2/3)

Example:

```
1 a = 2
2 if(a % 2 == 0):
3     print('a is even')
4 else:
5     print('a is odd')
6 print('end')
```



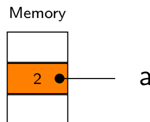
If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Control structures or conditional jump (2/3)

Example:

```
1 a = 2
2 if(a % 2 == 0):
3     print('a is even')
4 else:
5     print('a is odd')
6 print('end')
```



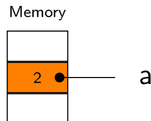
If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Control structures or conditional jump (2/3)

Example:

```
1 a = 2
2 if(a % 2 == 0):      True
3     print('a is even')
4 else:
5     print('a is odd')
6 print('end')
```



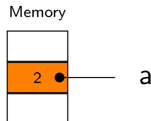
If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Control structures or conditional jump (2/3)

Example:

```
1 a = 2
2 if(a % 2 == 0):
3     print('a is even')
4 else:
5     print('a is odd')
6 print('end')
```



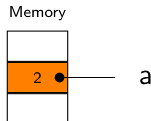
If you run that program, you obtain, on the terminal:

```
1 a is even
2 .
```

Control structures or conditional jump (2/3)

Example:

```
1 a = 2
2 if(a % 2 == 0):
3     print('a is even')
4 else:
5     print('a is odd')
6 print('end')
```



If you run that program, you obtain, on the terminal:

```
1 a is even
2 end
```

Control structures or conditional jump (3/3)

If the else block is not needed, you can omit them:

```
1 say_something = False
2 if(say_something):
3     print('Hello')
```

The last program is equivalent to:

```
1 say_something = False
2 if(say_something):
3     print('Hello')
4 else:
5     pass
```

Loop and iterations

Loop and iteration allow to repeat some block of instructions.

For example, we want to ask 10 numbers to the user to compute the sum of that numbers.

We want to repeat:

```
1 print('give a number')
2 number = int(input())
```

And with those 10 numbers compute and print the sum.

In Python, we can use

- While structures,
- For structures.

While structures (1/2)

A while structure repeats a block of instructions until a special condition becames false.

The syntax is the following:

```
1 while CONDITION :
2   INSTRUCTIONS_TO_REPEAT
```

`INSTRUCTIONS_TO_REPEAT` is repeated indefinittely as long as CONDITION is True.

While structure (2/2)

The program that asks for 10 numbers and returns the sum is:

```
1 counter = 1
2 sum = 0
3 while counter <= 10:
4   print('give a number')
5   number = int(input())
6   sum = sum + number
7   counter = counter + 1
8 # End of the loop
9 print('the sum of the 10 numbers is' +str(sum))
```

For structures (1/5)

A for structure repeats a block of instructions a fixed number of times, by enumerating some element from a given set.

For each element of the given set, a variable containing that element is created and the block of instruction is executed.

The syntax is the following:
containing that element is stored in the m Each time the block of instructions is executed, a variable conta of a given set.

```
1 for variable in SET_OF_ELEMENTS :
2   INSTRUCTIONS_TO_REPEAT
```

For structure (2/5)

Here an example, that prints the name of a day present in a tuple.

```
1 days = ('monday', 'tuesday', 'wednesday')
2 for item in days:
3   print(item)
```

The execution gives:

```
1 monday
2 tuesday
3 wednesday
```

For structure (3/5)

Here an example, that prints all integers from 0 to 6 excluded:

```
1 for i in range(6):
2   print(i)
```

The execution gives:

```
0
1
2
3
4
5
```

The function `range` creates a list from 0 to 6 excluded: $[0 - 6[$.

The range fonction accepts also 3 parameters:

`range(BeginningValue,EndValue,step)`.

For example, the program:

```
1 for i in range(1, 6, 2):
2   print(i)
```

gives:

```
1
3
5
```

For structure - Exercise 1 (4/5)

With a for structure, write the program that asks, to the user, 10 numbers and returns the sum of that numbers.

For structure - Exercise 1 (4/5)

With a for structure, write the program that asks, to the user, 10 numbers and returns the sum of that numbers.

```
1 sum = 0
2 for counter in range(10)
3   print('give a number')
4   number = int(input())
5   sum = sum + number
6 print('the sum of the 10 numbers is' +str(sum))
```

For structure - Exercise 1 (4/5)

With a for structure, write the program that asks, to the user, 10 numbers and returns the sum of that numbers.

```
1 sum = 0
2 for counter in range(10)
3   print('give a number')
4   number = int(input())
5   sum = sum + number
6 print('the sum of the 10 numbers is' +str(sum))
```

Loops written with a for structure is simpler, and more robust.

Indeed, if we forget to update the counter in a while structure an infinite loops occurs

For structure - Exercise 2 (5/5)

What does this program ?

```
1  animals = ('rat', 'mouse', 'cobaye')
2  N = 10
3  for animal in animals:
4      result = 0
5      for i in range(N):
6          print('Give the weight for ' + animal)
7          weight = int(input())
8          result = result + weight
9      print('Sum of weights for ' + animal + ' is ' + str(result))
10 print('end')
```

Outline

1. Introduction
2. What is a computer? What is a program?
3. My first programs
4. How to run a program: Interpreter, Compiler and executable
5. Python syntax
6. Variables, assignments and memory
7. Types and Native operators
8. Control and loop structures
9. List and dictionary
10. Functions

List (1/4)

A list is a data structure that codes an ordered sequence of elements. We can access to any element of a list with the position of the element in the list.

- To declare a list:

```
1 list_1=[]
2 list_2=[1,3.5,'la maison']
```

- To get the size of a list: `len(LIST)`

```
1 print(len(list_1)) # => 0
2 print(len(list_2)) # => 3
```

List (2/3)

- To add and get an element of a list:

```
1  list_1=[]
2  list_2=[1,3.5,'la maison']
3
4  list_1.append(5)  # We append 5 in list_1
5  list_1.append(2)  # We append 2 in list_1
6  print(list_1)     # => [5, 2]
7
8  print(list_2[0])  # => 1
9  print(list_2[1])  # => 3.5
10 print(list_2[2])  # => la maison
```

For a list LIST, the cells are indexed from 0 to $len(LIST) - 1$.

List: memory managment (2/4)

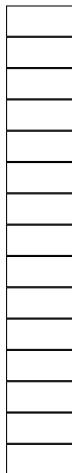
Program:

```
1 list_1 = [1, 2]
2 list_1.append(5)
3 print(list_1)
4
5 list_2 = list_1
6 list_2[1] = 42
7
8 print(list_1)
9 print(list_2)
```

The execution:

```
1
2
3 .
```

Memory:

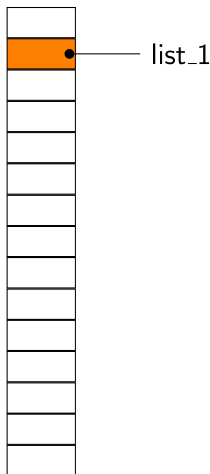


List: memory managment (2/4)

Program:

```
1  list_1 = [1, 2]
2  list_1.append(5)
3  print(list_1)
4
5  list_2 = list_1
6  list_2[1] = 42
7
8  print(list_1)
9  print(list_2)
10
```

Memory:



The execution:

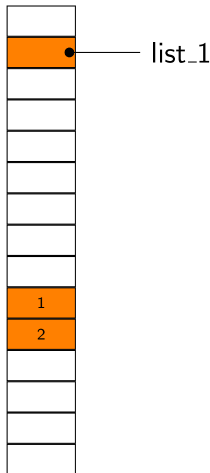
```
1
2
3 .
```

List: memory managment (2/4)

Program:

```
1  list_1 = [1, 2]
2  list_1.append(5)
3  print(list_1)
4
5  list_2 = list_1
6  list_2[1] = 42
7
8  print(list_1)
9  print(list_2)
10
```

Memory:



The execution:

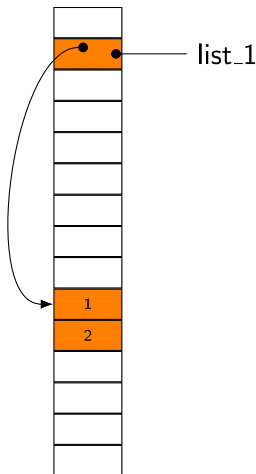
```
1
2
3  .
```

List: memory managment (2/4)

Program:

```
1  list_1 = [1, 2]
2  list_1.append(5)
3  print(list_1)
4
5  list_2 = list_1
6  list_2[1] = 42
7
8  print(list_1)
9  print(list_2)
10
```

Memory:



The execution:

```
1
2
3 .
```

List: memory managment (2/4)

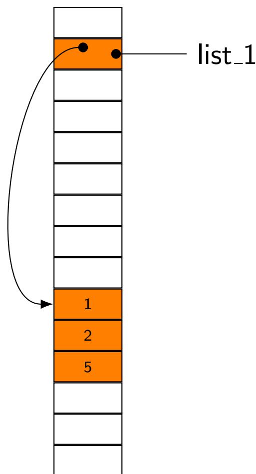
Program:

```
list_1 = [1, 2]
list_1.append(5)
print(list_1)

list_2 = list_1
list_2[1] = 42

print(list_1)
print(list_2)

```

The execution:

```
1
2
3 .
```

Memory:

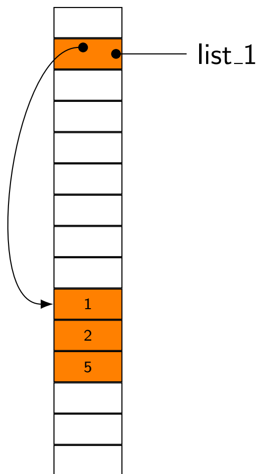


List: memory managment (2/4)

Program:

```
1  list_1 = [1, 2]
2  list_1.append(5)
3  print(list_1)
4
5  list_2 = list_1
6  list_2[1] = 42
7
8  print(list_1)
9  print(list_2)
10
```

Memory:



The execution:

```
1  [1, 2, 5]
2
3  .
```

List: memory managment (2/4)

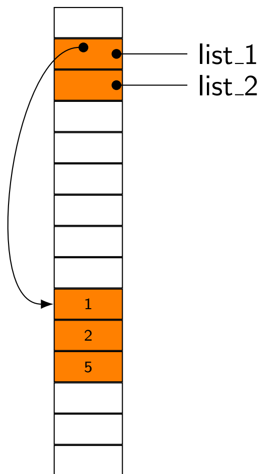
Program:

```
1  list_1 = [1, 2]
2  list_1.append(5)
3  print(list_1)
4
5  list_2 = list_1
6  list_2[1] = 42
7
8  print(list_1)
9  print(list_2)
10
```

The execution:

```
1  [1, 2, 5]
2
3  .
```

Memory:



List: memory managment (2/4)

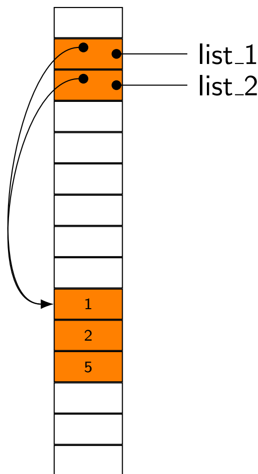
Program:

```
1 list_1 = [1, 2]
2 list_1.append(5)
3 print(list_1)
4
5 list_2 = list_1
6 list_2[1] = 42
7
8 print(list_1)
9 print(list_2)
10
```

The execution:

```
1 [1, 2, 5]
2
3 .
```

Memory:

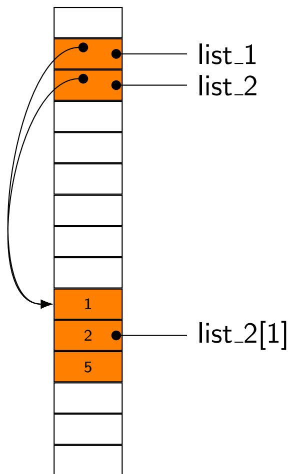


List: memory managment (2/4)

Program:

```
1  list_1 = [1, 2]
2  list_1.append(5)
3  print(list_1)
4
5  list_2 = list_1
6  list_2[1] = 42
7
8  print(list_1)
9  print(list_2)
10
```

Memory:



The execution:

```
1  [1, 2, 5]
2
3  .
```

List: memory managment (2/4)

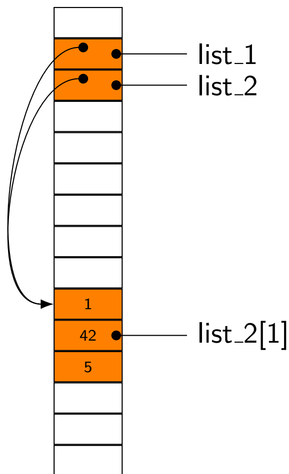
Program:

```
1  list_1 = [1, 2]
2  list_1.append(5)
3  print(list_1)
4
5  list_2 = list_1
6  list_2[1] = 42
7
8  print(list_1)
9  print(list_2)
10
```

The execution:

```
1  [1, 2, 5]
2
3  .
```

Memory:



List: memory managment (2/4)

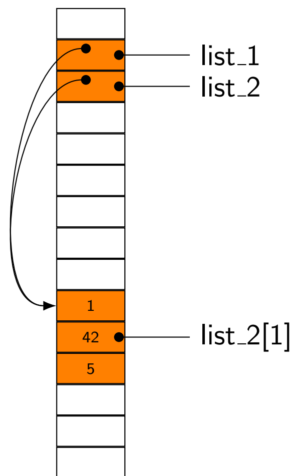
Program:

```
1  list_1 = [1, 2]
2  list_1.append(5)
3  print(list_1)
4
5  list_2 = list_1
6  list_2[1] = 42
7
8  print(list_1)
9  print(list_2)
10
```

The execution:

```
1  [1, 2, 5]
2  [1, 42, 5]
3  .
```

Memory:



List: memory managment (2/4)

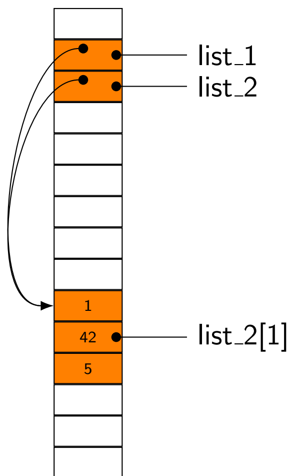
Program:

```
1  list_1 = [1, 2]
2  list_1.append(5)
3  print(list_1)
4
5  list_2 = list_1
6  list_2[1] = 42
7
8  print(list_1)
9  print(list_2)
10
```

The execution:

```
1  [1, 2, 5]
2  [1, 42, 5]
3  [1, 42, 5]
```

Memory:

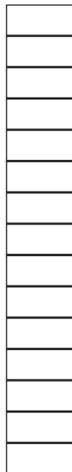


List: How to copy a list ? (3/4)

Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
```

Memory:



The execution:

```
1
2
3 .
```

List: How to copy a list ? (3/4)

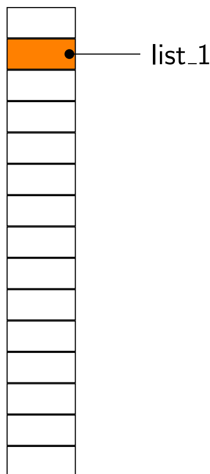
Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

The execution:

```
1
2
3 .
```

Memory:



List: How to copy a list ? (3/4)

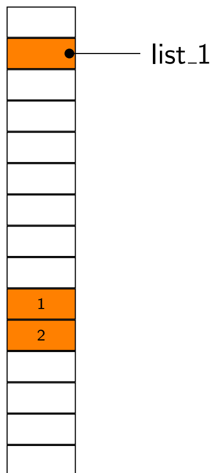
Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

The execution:

```
1
2
3  .
```

Memory:



List: How to copy a list ? (3/4)

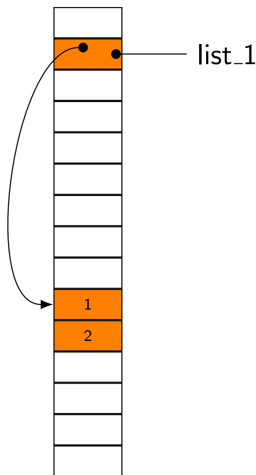
Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

The execution:

```
1
2
3 .
```

Memory:

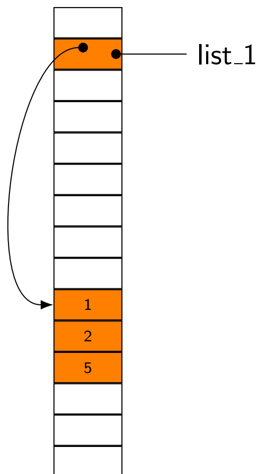


List: How to copy a list ? (3/4)

Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

Memory:



The execution:

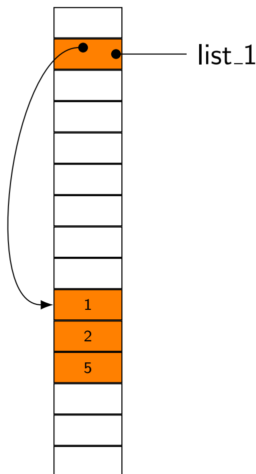
```
1
2
3 .
```

List: How to copy a list ? (3/4)

Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

Memory:



The execution:

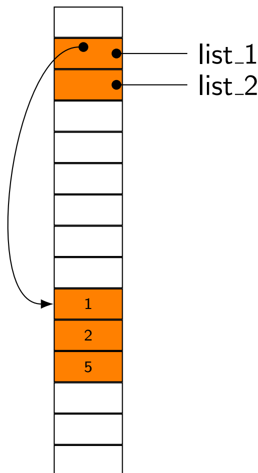
```
1  [1, 2, 5]
2
3  .
```

List: How to copy a list ? (3/4)

Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

Memory:



The execution:

```
1  [1, 2, 5]
2
3  .
```

List: How to copy a list ? (3/4)

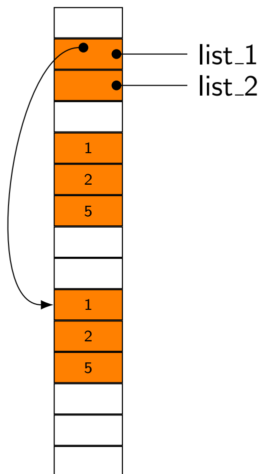
Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

The execution:

```
1  [1, 2, 5]
2
3  .
```

Memory:

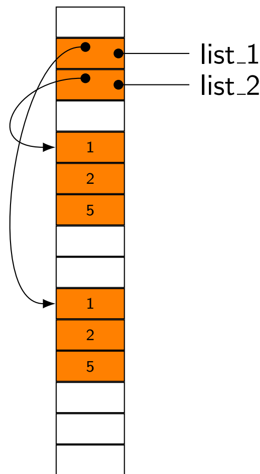


List: How to copy a list ? (3/4)

Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

Memory:



The execution:

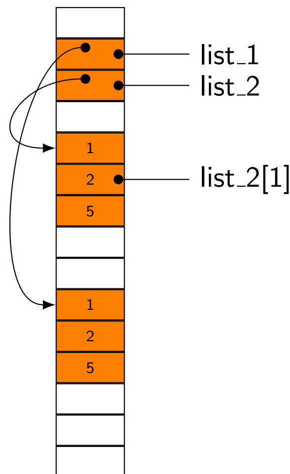
```
1  [1, 2, 5]
2
3  .
```

List: How to copy a list ? (3/4)

Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

Memory:



The execution:

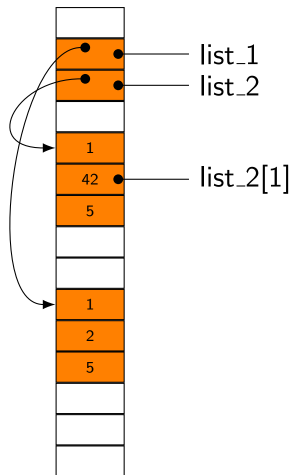
```
1  [1, 2, 5]
2
3  .
```

List: How to copy a list ? (3/4)

Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

Memory:



The execution:

```
1  [1, 2, 5]
2
3  .
```

List: How to copy a list ? (3/4)

Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

Memory:

list_1
list_2
list_2[1]

Memory
•
•
1
42
5
1
2
5

The execution:

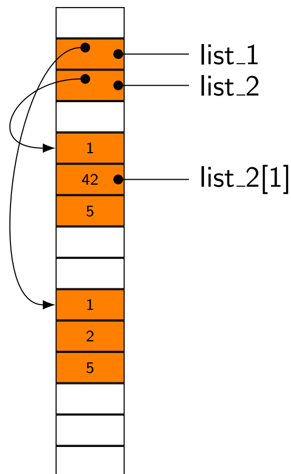
```
1  [1, 2, 5]
2  [1, 2, 5]
3  .
```

List: How to copy a list ? (3/4)

Program:

```
1  from copy import deepcopy
2
3  list_1 = [1, 2]
4  list_1.append(5)
5  print(list_1)
6
7  list_2 = deepcopy(list_1)
8  list_2[1] = 42
9
10 print(list_1)
11 print(list_2)
12
```

Memory:



The execution:

```
1 [1, 2, 5]
2 [1, 2, 5]
3 [1, 42, 5]
```

List: Exercise (4/4)

Analyse the following program:

```
1  l1 = [1, 2, 3]
2  l2 = [1, 2, 3]
3
4  l1[1] = [1, 2, 3]
5  l2[1] = l1
6
7  l2[1][0] = 42
8  l2[1][1][0] = 5
9
10 print(l1)
11 print(l2)
```

Is the progam correct ? If not, correct the program. Explain what it makes and draw the memory.

To check your solution, use the website:
https://pythontutor.com/visualize.html

Dictionary

A dictionary is a data structure which allows to access to the element called a value by the name of the box containing the data. This name is called the key of the element (or the key of the value).

- To declare a dictionary:

```
1 dict_1={}
2 dict_2={'bread':3, 'milk': 5, 'butter': 'vide'}
```

- To use a dictionary:

```
1 dict_1['animal']='chat' # set a value to the key 'animal'
2 print(dict_1['animal']) # get the value of 'animal' => chat
3 print( 'milk' in dict_2 ) # ask if a key exists => True
4
5 print( len(dict_2) ) # the size of the dictionay => 3
6 print( dict_2.keys() ) # get all the keys
7                        # => ['bread', 'milk', 'butter']
8 print( list(dict_2.items()) ) # get all the associations
9       # => [('bread', 3), ('milk', 5), ('butter', 'vide')]
```

Outline

1. Introduction
2. What is a computer? What is a program?
3. My first programs
4. How to run a program: Interpreter, Compiler and executable
5. Python syntax
6. Variables, assignments and memory
7. Types and Native operators
8. Control and loop structures
9. List and dictionary
10. Functions

Function (1/4)

A function is a block of instructions associated with a name. In the program, that block is executed each time the name is called.

The memory space of a function is independent from the memory space of the main program or other functions.

To share data, function have parameters, parameters are variable stored in the memory space of the function. When a function is executed, for each parameters, a value is given. The parameter memory are intialized with these values.

To stop the execution of a function, we can use the following instruction in the block of instructions:

```
1 return VALUE
```

When that instruction is executed, the function stops, releases all it's memory (all data is lost) and returns the value VALUE as result.

Function (2/4)

- Definition:

```
1 def function_name(LIST_OF_PARAMETER_NAMES):
2      INSTRUCTIONS_TO_EXECUTE
```

- Stop a function and return some value as result:

```
1 return VALUE
```

- Call a function:

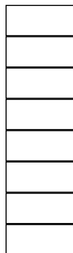
```
function_name(List_OF_PARAMETER_VALUES)
```

- Call and collect a function result:

```
result = function_name(List_OF_PARAMETER_VALUES)
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

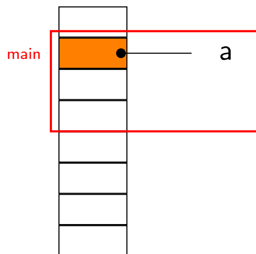


If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5 
6 a = 5
7 b = square(a)
8 square(b)
```

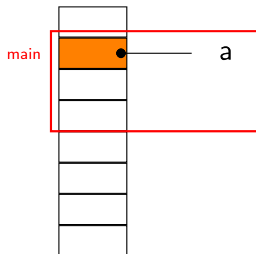


If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

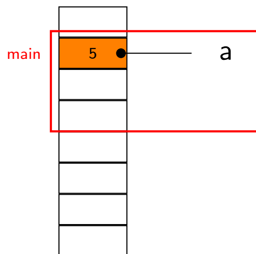


If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

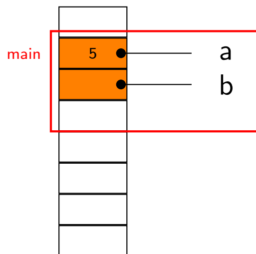


If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

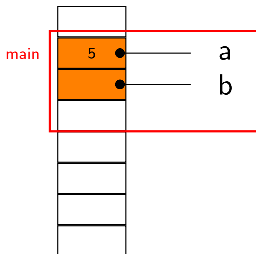


If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

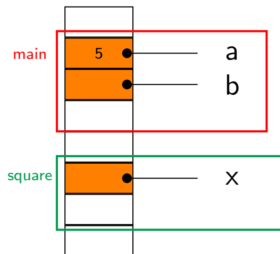


If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

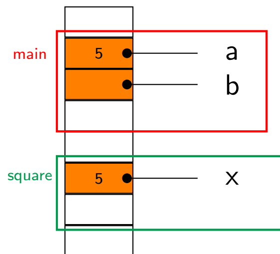


If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

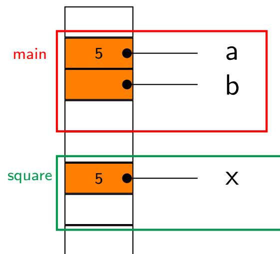


If you run that program, you obtain, on the terminal:

```
1 .
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

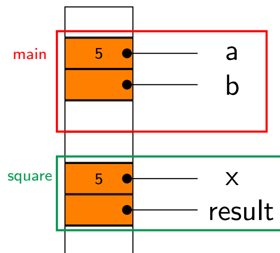


If you run that program, you obtain, on the terminal:

```
1 Square of 5
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```



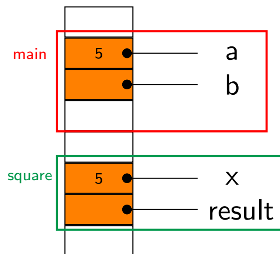
If you run that program, you obtain, on the terminal:

```
1 Square of 5
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

25

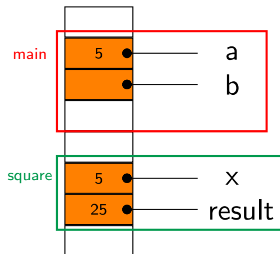


If you run that program, you obtain, on the terminal:

```
1 Square of 5
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

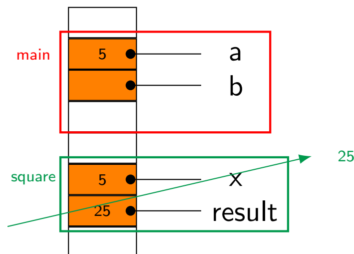


If you run that program, you obtain, on the terminal:

```
1 Square of 5
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

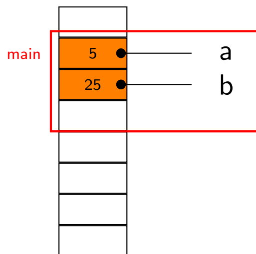


If you run that program, you obtain, on the terminal:

```
1 Square of 5
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

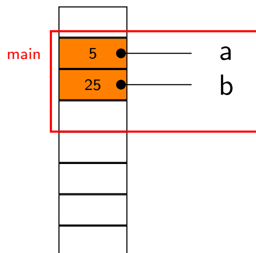


If you run that program, you obtain, on the terminal:

```
1 Square of 5
2 .
```

Function: a simple example (2/4)

```
1 def square(x):
2   print( 'Square of ' + str(x))
3   result = x * x
4   return result
5
6 a = 5
7 b = square(a)
8 square(b)
```

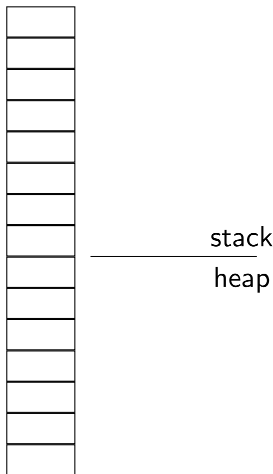


If you run that program, you obtain, on the terminal:

```
1 Square of 5
2 Square of 25
```

Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

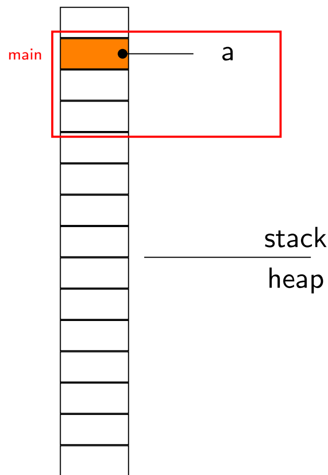


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] = [3, 4]
5     return a
6 
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```



Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

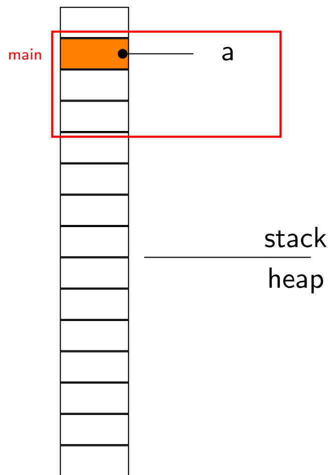
All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

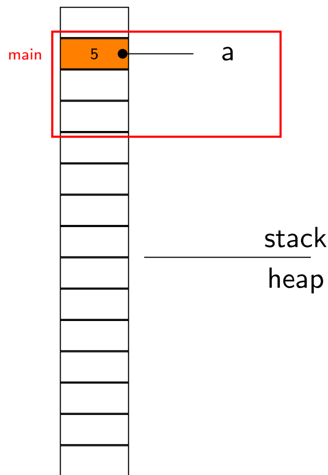


Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

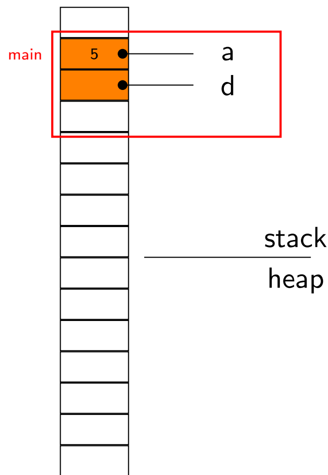


Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

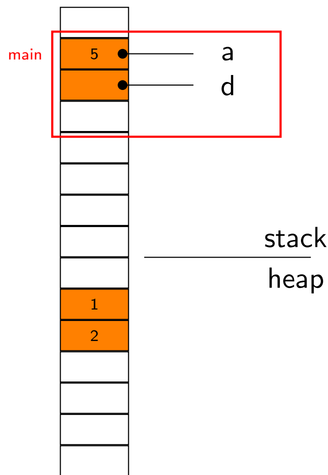


Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

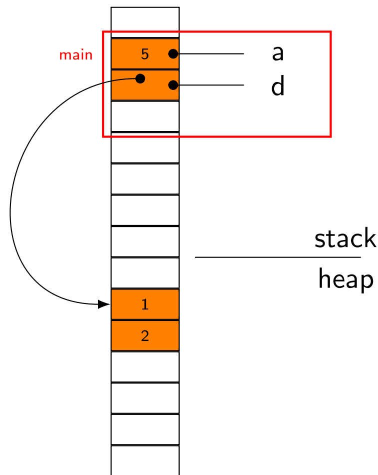
Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.



Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

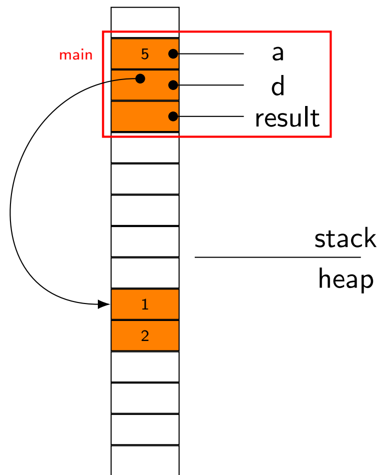


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

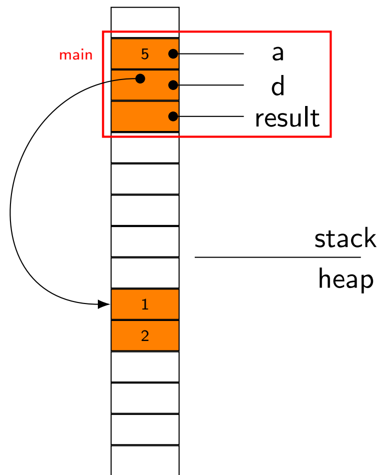


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```



Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

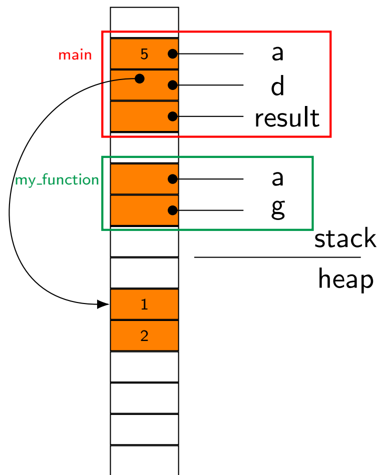
All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] = [3, 4]
5     return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

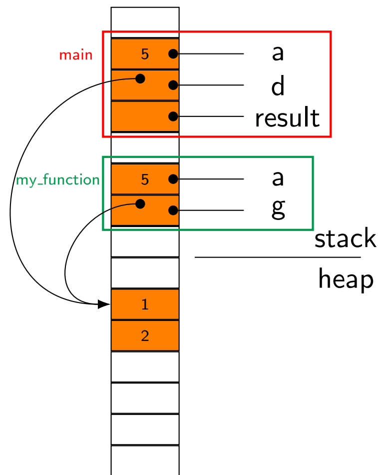


Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] = [3, 4]
5     return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

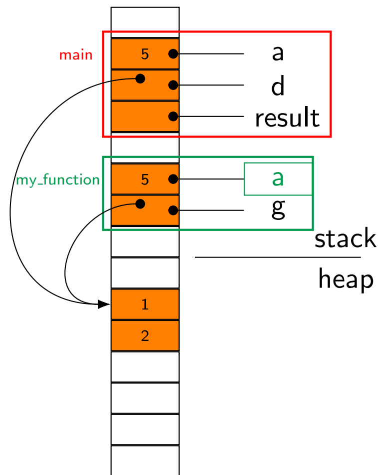
Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.



Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

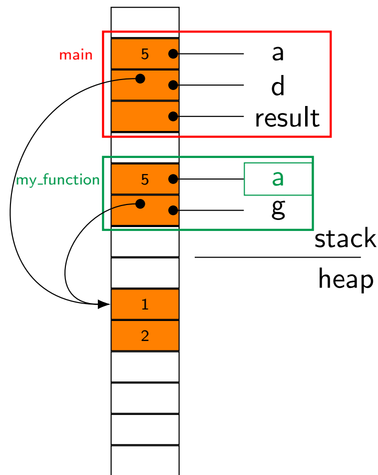


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

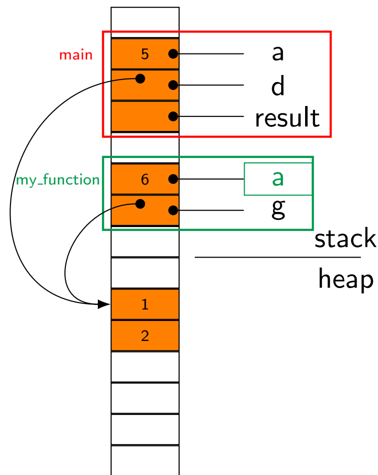


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] = [3, 4]
5     return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

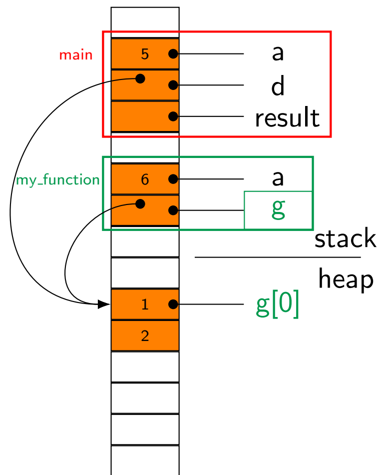


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

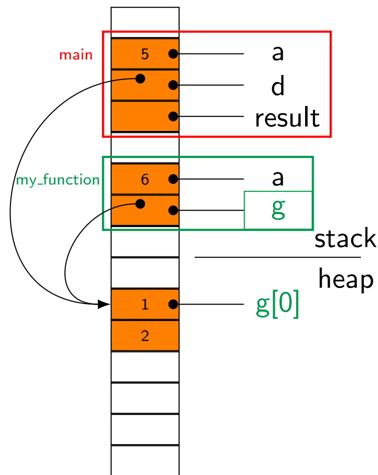


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] = [3, 4]
5     return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

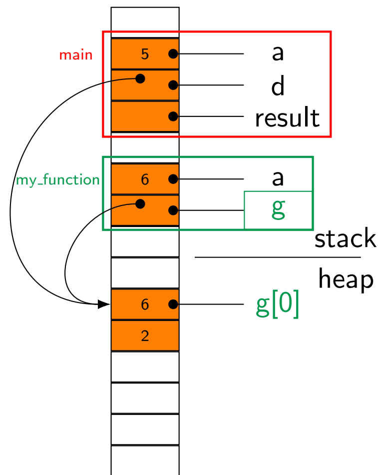


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] ␣ [3, 4]
5     return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

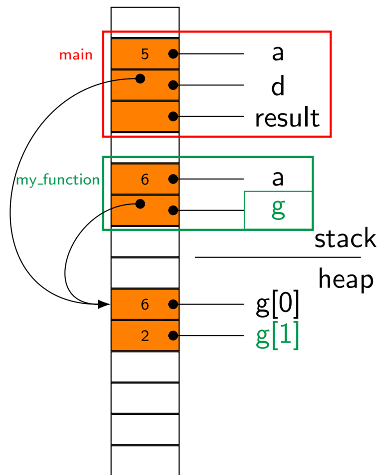


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1  def my_function(a, g):
2      a = a + 1
3      g[0] = a
4      g[1] = [3, 4]
5      return a
6
7  a = 5
8  d = [1, 2]
9  result = my_function(a, d) + 1
```

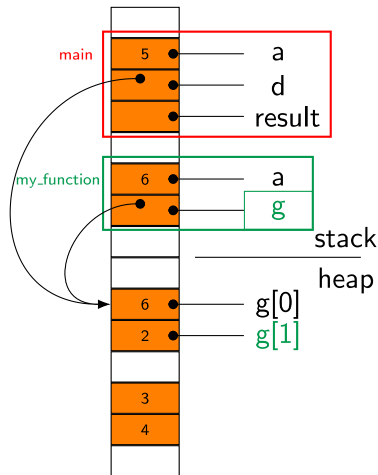


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] = [3, 4]
5     return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```



Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

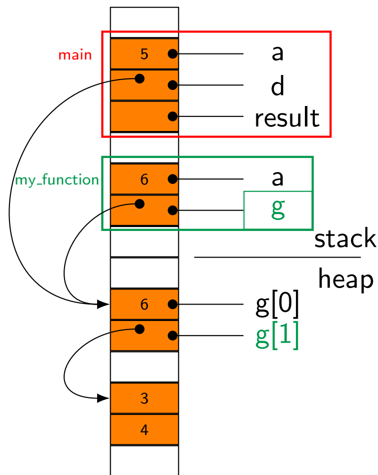
All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] = [3, 4]
5     return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

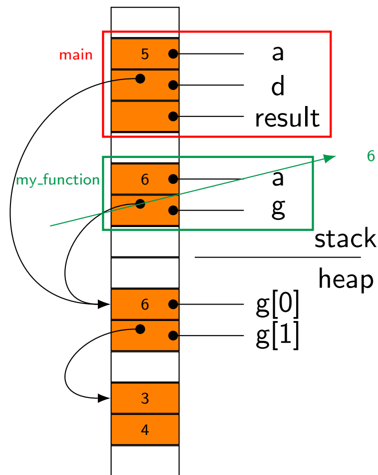


Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] = [3, 4]
5     return a
6                      6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.



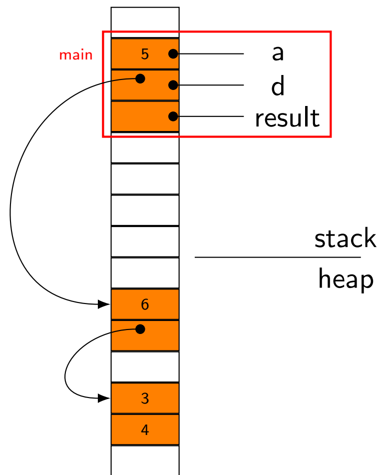
Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

7

Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

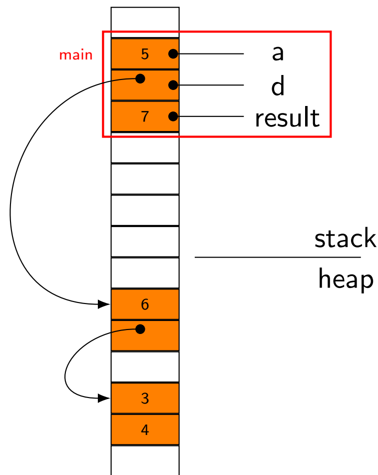


Function: a complex example (4/4)

```
1 def my_function(a, g):
2     a = a + 1
3     g[0] = a
4     g[1] = [3, 4]
5     return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

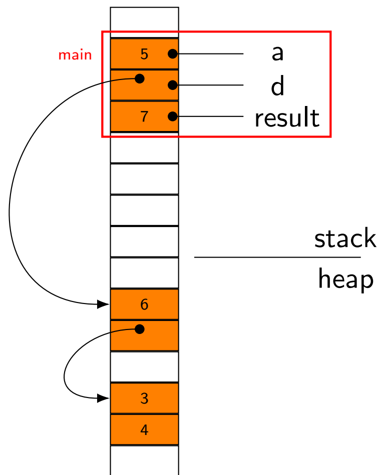
Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.



Function: a complex example (4/4)

```
1 def my_function(a, g):
2   a = a + 1
3   g[0] = a
4   g[1] = [3, 4]
5   return a
6
7 a = 5
8 d = [1, 2]
9 result = my_function(a, d) + 1
```

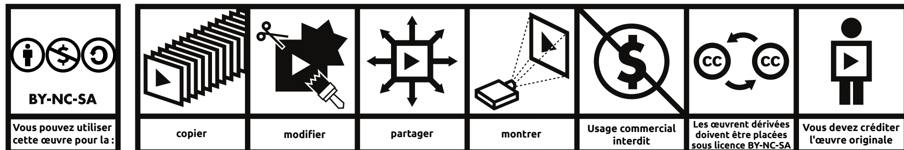


Memories of Variables and functions are allocated in a special memory space called the stack. Only memories of the stack are released when a function stops.

All other data are stored in a special memory space called the heap. This memory is automatically released only when it is not used anymore.

Licence

This course is under Creative Commons License BY-NC-SA 4.0:



Attribution - Pas d'utilisation commerciale - Partage dans les mêmes conditions 4.0 – Cette licence autorise autrui à copier, modifier, partager votre œuvre, sauf pour des utilisations commerciales. Les personnes utilisant votre œuvre s'engagent à la créditer, à intégrer un lien vers la licence et à indiquer si des modifications ont été effectuées à l'œuvre. Si elles réalisent des modifications, l'œuvre dérivée devra être diffusée sous licence CC-BY-NC-SA également. **https://creativecommons.org/licenses/by-nc-sa/4.0/deed.fr**

Licence

This course is under Creative Commons License BY-NC-SA 4.0: