FS12

Week 01
Python Basics I

Mikhail Masyagin
Daniil Devyatkin

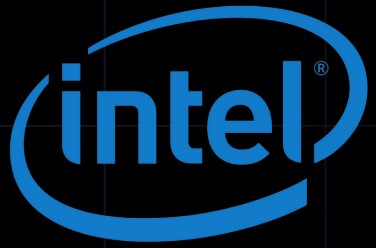
Linux runtime essentials I

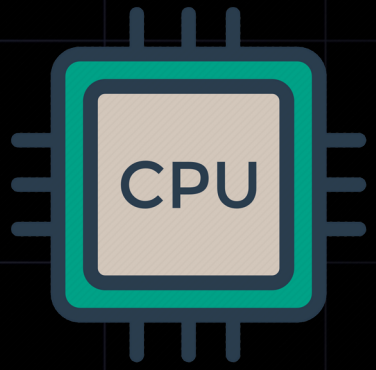
- It is impossible to write efficient Python code without a sufficient understanding of the hardware architecture and operating system nuances of target platform.
- Here we will discuss following topics:
  - memory model, in which program is stored and operates;
  - types of variables and their life-cycle;
  - program decomposition into functions and function call mechanism.

Linux runtime essentials II

- This course will mostly focus on platforms, equipped with 64-bit Intel processors and work-running under the Linux Operating System.

+





Virtual Address Space (VAS)

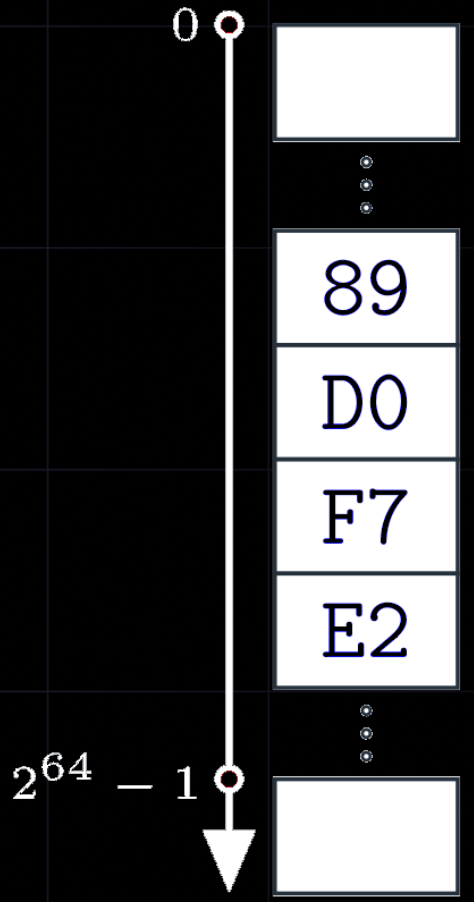
- From a programmers view, program code and its data of the are placed in an array of 2^64 bytes. Each byte's number in this array is called an address, and all 2^64 [0, 2^64) addresses is the program's virtual address space. **2^64 B = 163.84 PB = a lot**
- In a reality, only a small amount of addresses correspond to the cells of the physical memory. An attempt to read or write to an address, that is not mapped into physical memory, will result in error.
- On Intel CPU only [0, 2^48) addresses available to program.

Machine Instructions

- Machine code is a sequence of elementary processor instructions, each of occupies one or more consecutive bytes.
- When a program is launched, the Linux OS places the its machine code in a VAS, which makes it possible to access instructions via their addresses. In this case, instruction's address is the address of its first byte is hidden.
- We can assume that only one processor instruction is executed per time. Its address is stored in a processors memory region called the IP register.

Memory contents

- The meaning of values, stored in memory, is determined by the operations, performed on them.
- For example bytes 89 D0 F7 E2 can be:
  - -2.29 x 10^21;
  - -487075703;
  - 3807891593;
  - 137.208.247.226;
  - mov %edx, %eax;
    mul %edx.



Functions

- The function is a program fragment that performs certain calculations based on the set of parameters passed to it and returns the result of calculations.
- Example:
  - `` `print("Hello, World!")` - `print` `` is a function.
- Function address is the address of the machine instruction at which its execution begins.

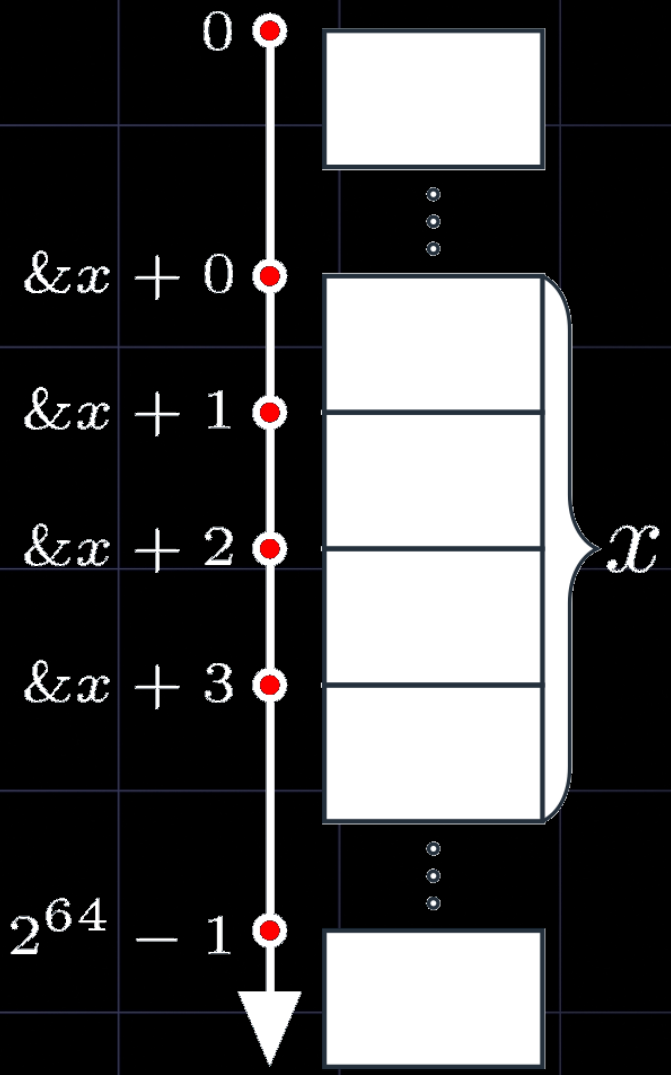
Variables I

- A variable is a bytes collection that occupies a contiguous part of the VAS and is used to store some value.
- The address of the first byte of the VAPs part, occupied by a variable, is called the address of the variable. All variables have different addresses.
- Accessing a variable is reading or writing its value by its address. Variables are created and destroyed while the program is running. The lifetime of a variable is the time interval from its creation to its destruction.

Variables II

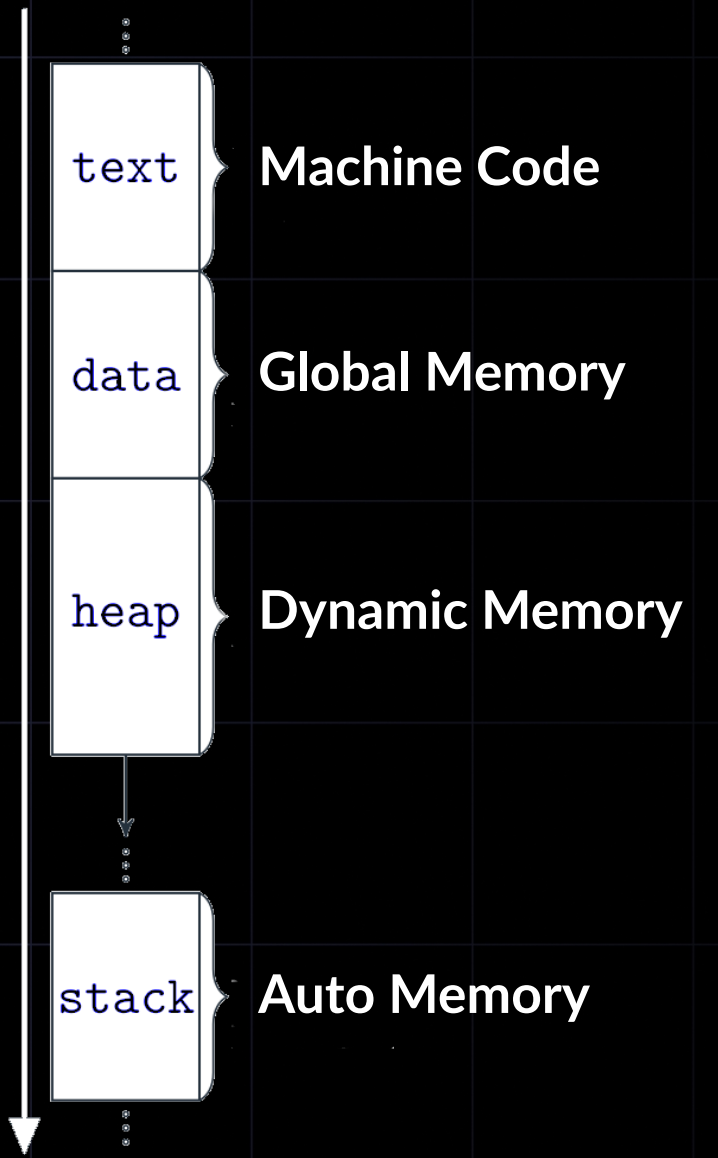
- Variables can be classified by 4 groups:
  - **global variables** – variables, which exist since the very start of program and until its complete end;
  - **local variables** – variables, which automatically created on a function call and deleted on exit from a function;
  - **formal function parameters** - local variables in which the values, passed during the function call, are written;
  - **dynamic variables** – are created and deleted via memory manager.

Variables III



VAPs Map

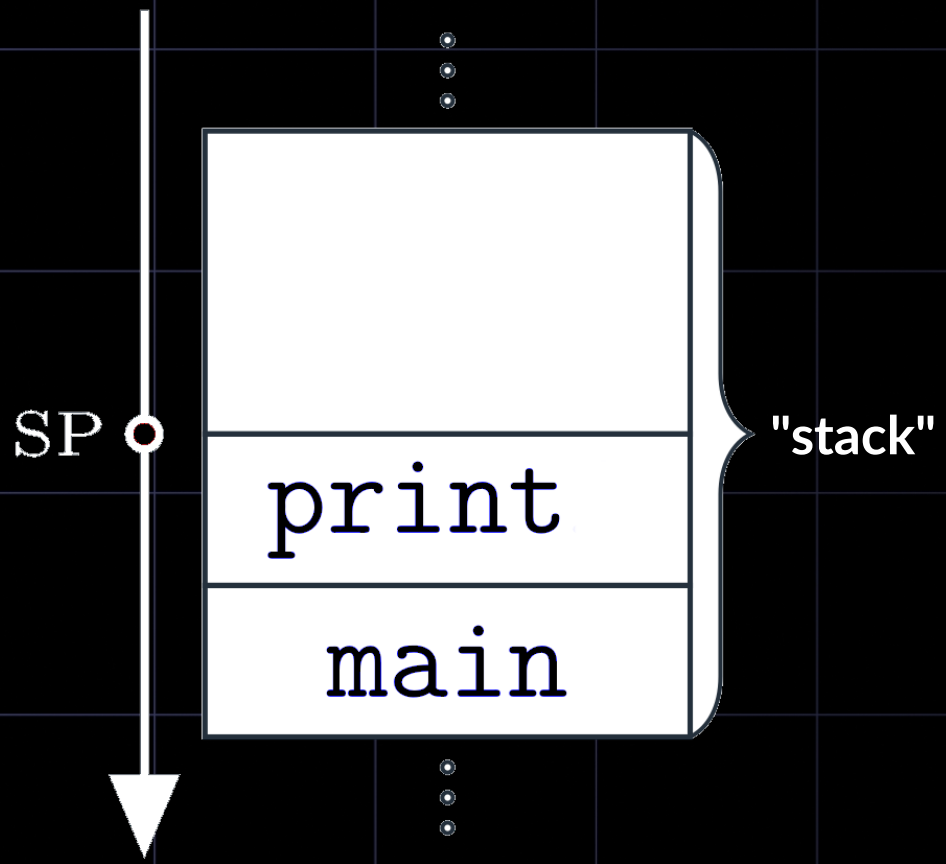
- VAP can be classified by 4 regions:
  - **"text"** – machine code is placed here;
  - **"data"** – global variables are placed here;
  - **"heap"** – dynamic variable are placed here;
  - **"stack"** – local variables are placed here;



Function Frames I

- A function frame is a section of automatic memory that stores local variables (formal parameters too) and the address to which control should be transferred after the function ends (return address).
- When the program starts, the frame of the "main" function is created in the higher "stack" addresses and control is transferred to its address.
- If `print` function is called from "main" function, then print’s frame is placed before the main frame and its return address is "main" address.

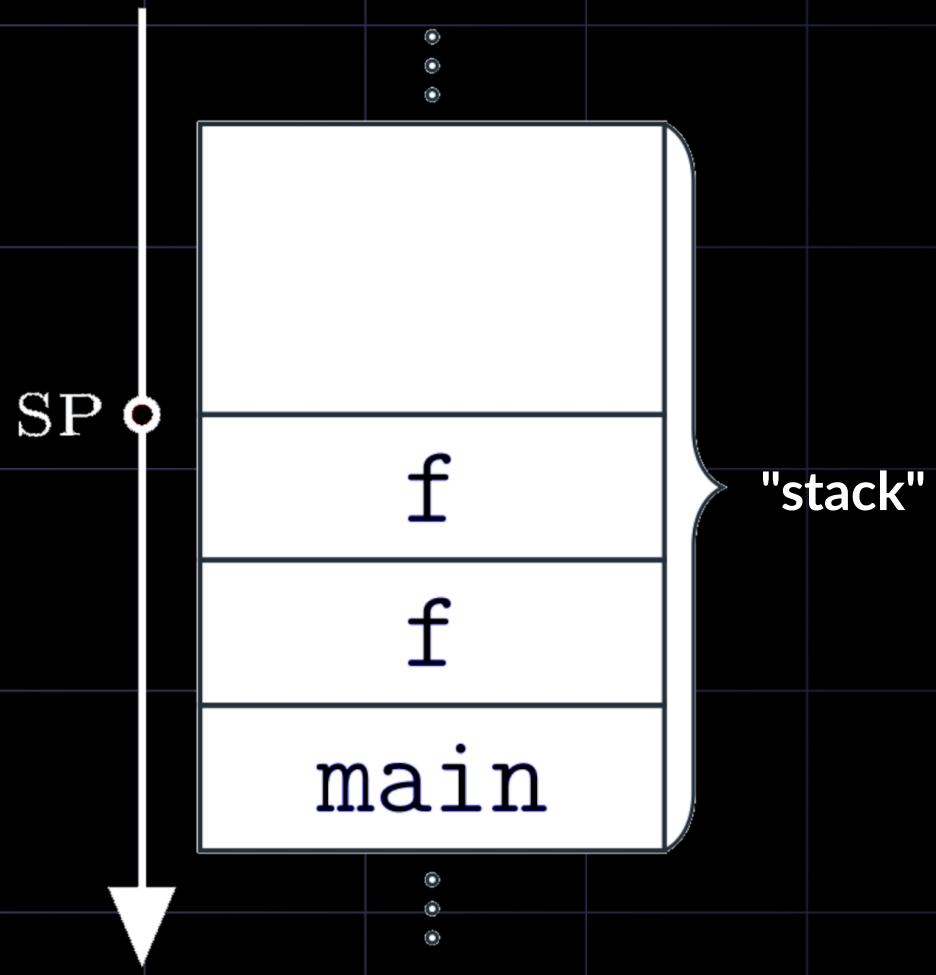
Function Frames II



Recursion Support I

- Function’s local variables storage in a frame, that is created at the time the function is called, allows recursion, i.e. a function can call to itself.
- For example, function "f" is called from the "main" function, and the function "f" is called from it again. As a result, 2 frames of the function "f" appear in the "stack" area, each with its own set of local variables (they are not intersecting!!!).
- It is worth to note that the use of recursion can lead to exhaustion of free space in the "stack" area.

Recursion Support II



What's next?

- Now you have a simple understanding on what is going on during the program's runtime and... you still do not know anything about Python's runtime.

- The main reason of it is that Python is an interpreted programming language.