FS12

Week 01
Python Basics I

Mikhail Masyagin
Daniil Devyatkin

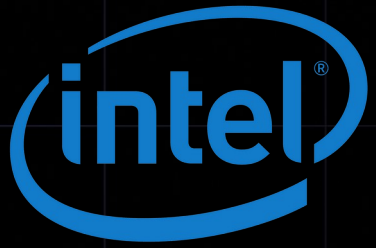
Linux runtime essentials I

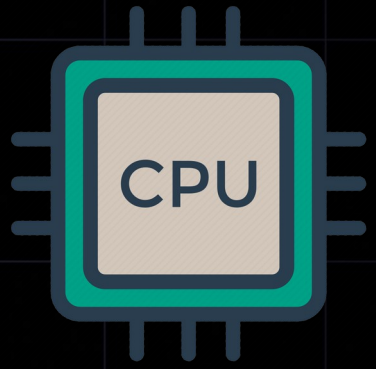
- It is impossible to write efficient Python code without a sufficient understanding of the hardware architecture and operating system nuances of target platform.
- Here we will discuss following topics:
  - memory model, in which program is stored and operates;
  - types of variables and their life-cycle;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

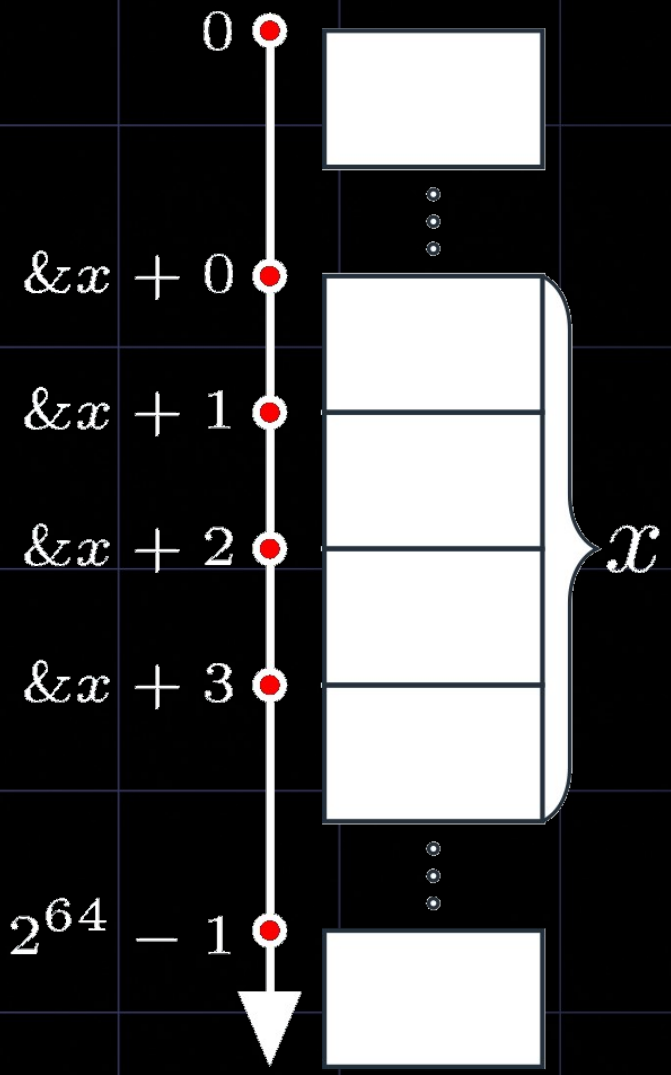
Variables I

- 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

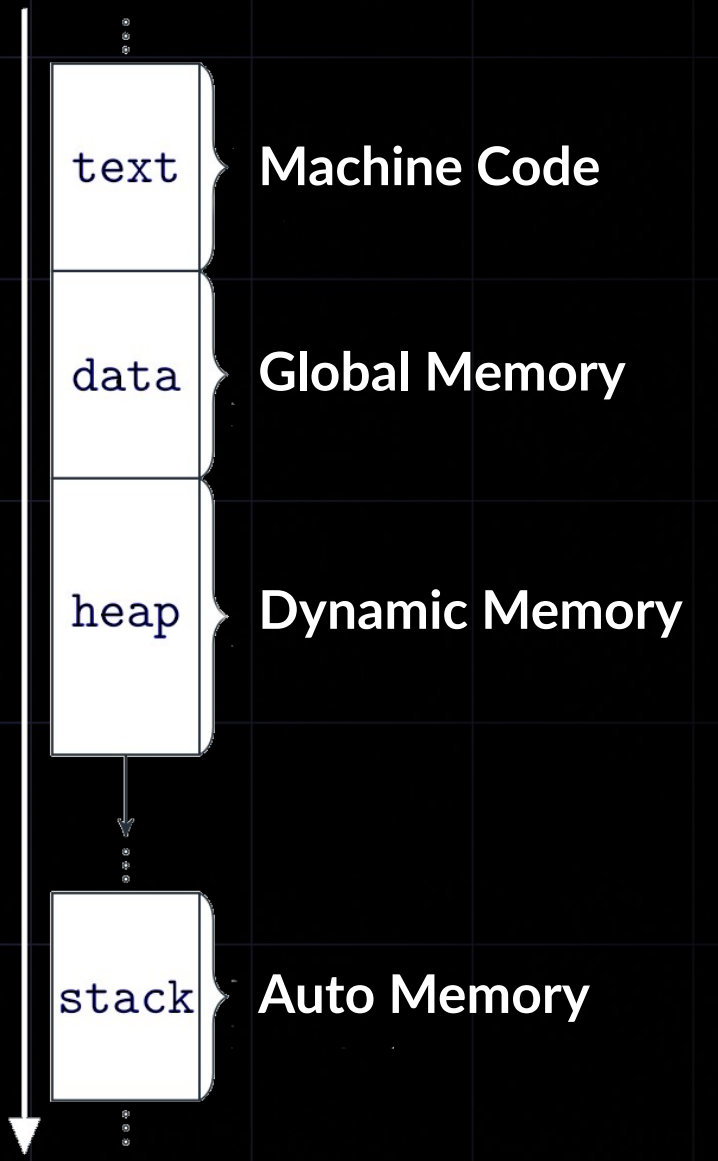
- Variables can be classified by 4 groups:
  - **global variables** – variables, which exist since the very start of program and until its complete end;
  - **local variables** – variables, which automatically created on a function call and deleted on exit from a function;
  - **formal function parameters** - 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

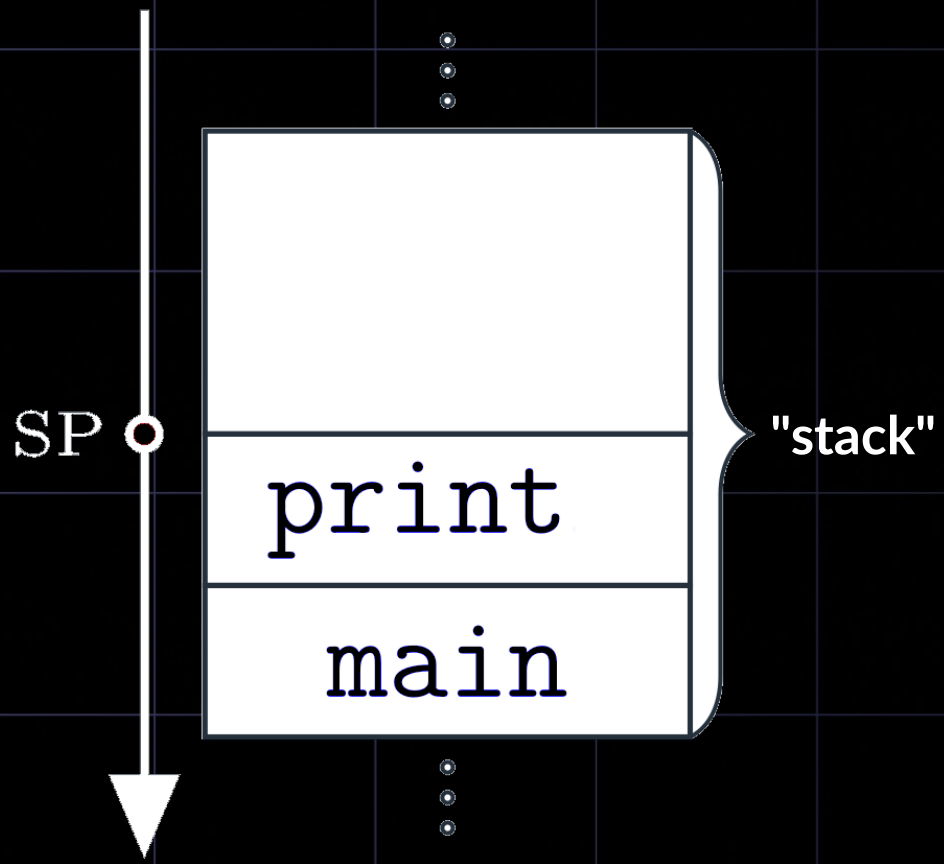
- VAP can be classified by 4 regions:
  - **"text"** – machine code is placed here;
  - **"data"** – global variables are placed here;
  - **"heap"** – dynamic variable are placed here;
  - **"stack"** – local variables are placed here;



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.

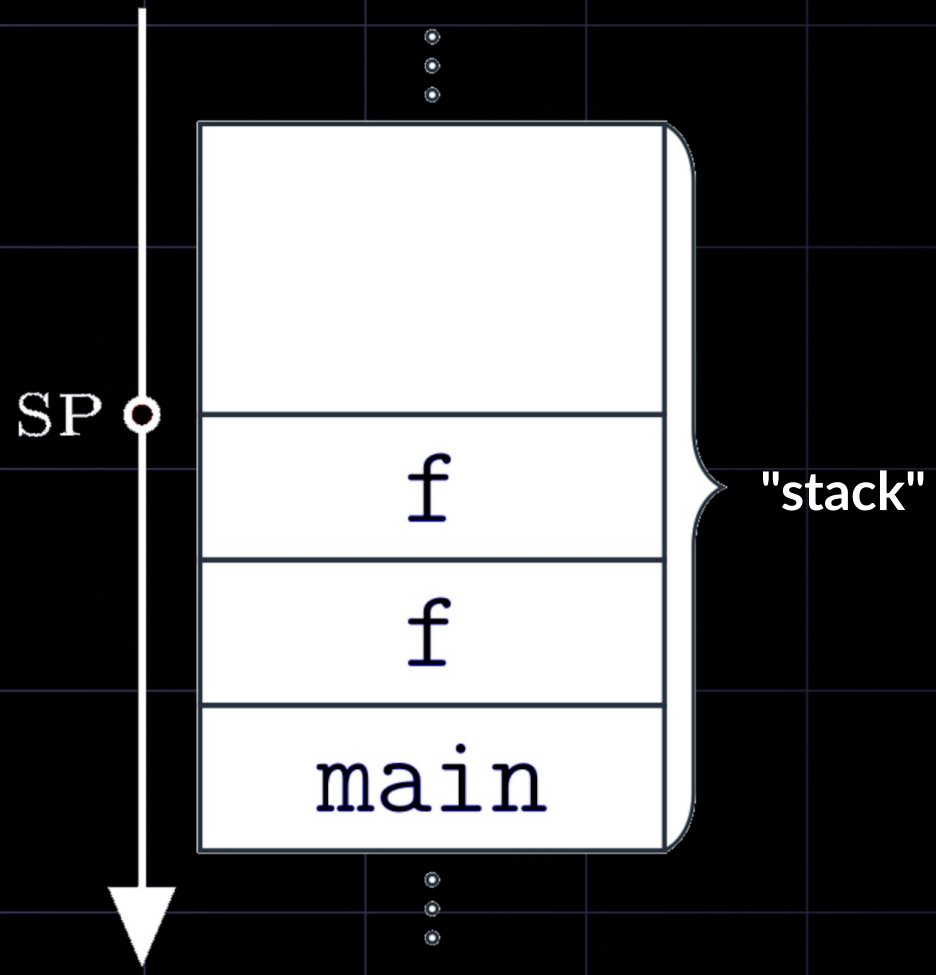
Function Frames II



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?

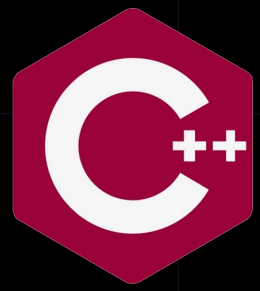
- Now you have a simple understanding on what is going on during the program's runtime and... you still do not know anything about Python's runtime.

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

Python’s Interpreter I

- We know that Python is an interpreted programming language, however, we do not know, what does it mean.
- Let’s try to know it by comparing compilers, interpreters and virtual machines.

Compiler

- Program Code → Machine Code.
- Processes whole program at once.
- Needed only once after the program is done.
- Allows to detect most errors during compilation.
- Need not to be present in VAP during program execution.
- Compiled programs usually run faster.
- **Compilation takes some time**.
- Compiler can better **optimize** program.




Interpreter

- Program Code → Byte Code|Direct Execution.
- Processes parts of program per time.

- Runs each time the program is executed.
- Most errors are caught during execution.

- Must be in VAP during program execution.
- Interpreted programs usually run **slower**.
- **Interpreter doesn't take time** before program run.
- Interpreter can't optimize program.

Virtual Machine

- Program Code → Byte Code.
- Runs each time the program is executed.
- Can detect most errors during compilation part.
- Must be in VAP during program execution.
- Between interpreters and compilers in speed, but sometimes faster both of them.
- Can optimize program dynamically depending on its input data and workloads.

From Source code to Runtime

- Now you know that you can user compiler, interpreter of virtual machine to make your program's text a running application. Let's dive deeper into their mechanisms.
- Each of them consists of two main parts:
  - **Frontend** – transforms program's source code to **intermediate representation** (IR);
  - **Backend** – runs IR (interpreter, vm), optimizes and transforms IR to machine code (compiler, vm).

Frontend

- Frontend often consists of two main parts: **lexer** and **parser**.
- **Lexer** splits program's text into sequence of tokens and then gives it to parser.
- **Parser** operates list of tokens and transforms them into binary tree.
- Let's see this process, using arithmetical expression as an example.

Lexer I

- Let’s split following string into tokens sequence:

```
""""(   (48 -    7%2 ) / 24) * (((18-
5*2 + 12)))"""
```

- Tokens sequence:

```
"(", "(", "48", "-", "7", "%", "2", ")",
"/", "24", ")", "*", "(", "(", "(", "18",
"-", "5", "*", "2", "+", "12", ")", ")",
")"
```

Lexer II

- Let's tag each of tokens:

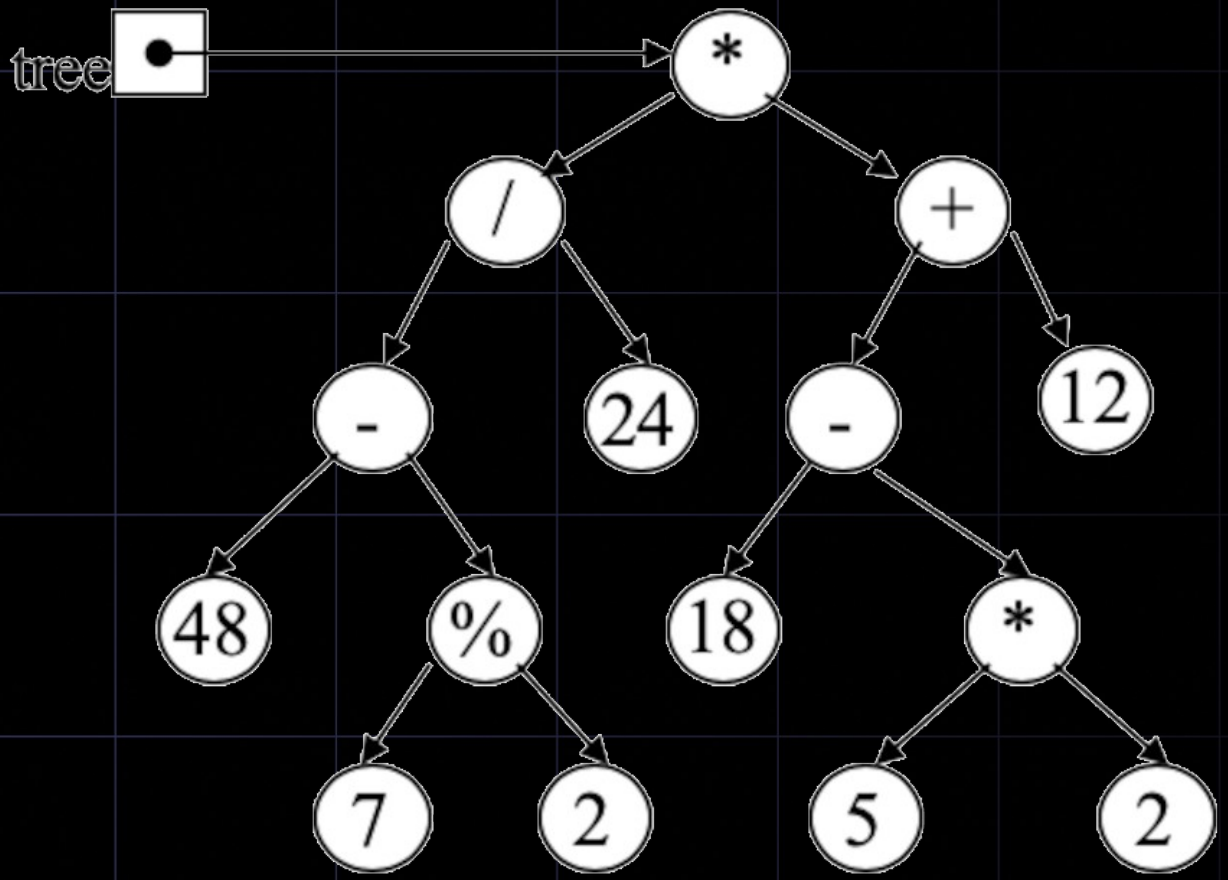
`lp`, **`lp`**, **`num`**, **`minus`**, **`num`**, **`mod`**, **`num`**, **`rp`**, **`div`**, **`num`**, **`rp`**, **`mul`**, **`lp`**, **`lp`**, **`lp`**, **`num`**, **`minus`**, **`num`**, **`mul`**, **`num`**, **`add`**, **`num`**, **`rp`**, **`rp`**, **`rp`**

Lexer III

- For human its obvious how to split given expression, into tokens however, for computer it is a big amount of work.
- Lexer has to:
  - Skip all unnecessary symbols (spaces, newlines);
  - Split given text into tokens;
  - Tag each token so following parts of translator can understand and distinguish them.

Parser I

- Let’s transform previous tokens sequence to tree according to arithmetical operations priority:



Parser II

- Parser consistently reads tokens sequence and transforms it into tree. In best case it can choose how to update tree on each token, however sometimes it needs to lookup next 2+ tokens (C++ grammar :)).
- Parser has to:
  - transform given tokens sequence into a program syntax tree.
  - find simple syntax errors and suggest how to fix them.

Backend I

- Backend often consists of two main parts:
  - for compilers:
    - **Optimizer** transforms program syntax tree to make it faster and smaller.
    - **Machine code generator** generates code for target platform (for example Linux).
  - for interpreters/VMs:
    - **Byte-Code generator** generates byte-code.
    - **Interpreter** runs this byte-code (VMs also can optimize it on-the-fly).

Backend II (Optimizer)

- Optimizer traverses syntax tree and tries to minimize and optimize it. For example it can:
  - precalculate some constant expressions;
  - remove dead code;
  - Inline functions (copy function code into its parent function so there will not be direct function call);
  - change loops with constant iterations number to linear instructions flow;
  - etc.

Backend III (Machine Code Generator)

- Machine Code Generator generates code for target platform. The more it knows about target platform the faster machine code it can produce. Example:
  - if processor is equipped with 128-bit registers some consecutive additions can be rewritten with less instructions;
  - if processor has multiple cores some compilers can parallel code execution;
  - usage of special Intel's multimedia instructions set;
  - replace `memcpy` calls with hardware DMA calls.

Backend IV (Machine Code Generator)

- Machine Code Generator can additionally optimize code but its main goal is to produce machine code for target platform:
  - in simplest (mostly old) compilers it can be done in one-pass traverse of syntax tree;
  - In modern compilers its a really difficult procedure and its often written by big teams.

Backend V (Byte Code Generator)

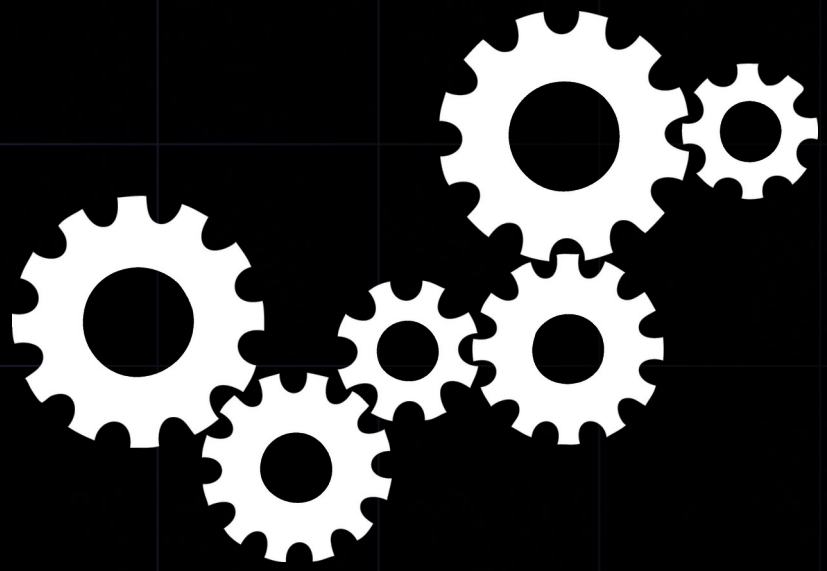
- Byte Code Generator is pretty similar to Machine Code Generator: it simply (ha-ha) generates code for target platform, but its target platform is not a real computer’s hardware but a virtual hardware: VM or byte code interpreter.

Backend VI (Interpreter)

- Interpreter operates just like a hardware processor: it runs byte code commands one by one or iterates through syntax tree and performs its operations. First is much faster, however, second is much easier to write (no byte code is required).

Python's Interpreter II

- Python is an interpreted programming language and its interpreter consists of following parts:
  - Frontend:
    - Lexer;
    - Parser;
  - Backend:
    - Byte Code Generator;
    - Byte Code Interpreter (stack-based virtual machine) (to be continued...);
    - Garbage Collector (to be continued...).



The hell with theory!

- Now you know how does Python interpreter run on Linux platform and how does Python code is being run via Python interpreter → you know how does Python code is being run on Linux platform.
- We can dive deep into Python programming right now!
- Let the game begin...

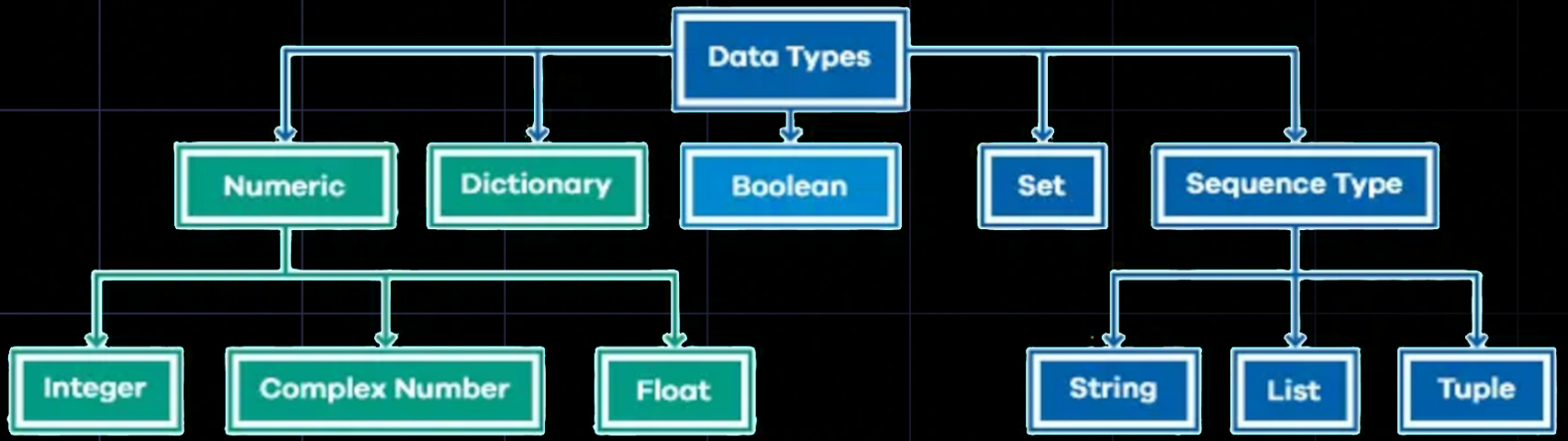
Python Variables I

- Python is a **dynamically typed** **type-safe** programming language:
- **Dynamically typed** means that each variable can change its type during the programs runtime (for example, it was number and then become string).
- **Type-safe** means that you can not for example sum number and string: it will result in interpreters error in runtime (some languages like C, C++ or JavaScript without `use strict` allows this nightmare).

Python Variables II (Types)

- Python supports following basic data types:
  - numeric: integers, float, complex;
  - logical (yes/no, true/false): boolean;
  - sequences: string, list, tuple.
  - maps: dict.
  - sets: set.
- All python data types are objects but we will talk about it a little bit later.
- Today we will talk about numeric, boolean and string types.

Python Variables III (Types)



Python Variables IV (Assignments)

- Python uses really simple variable assignment expression:

```
sm_integer = 1
sm_float_1, sm_float_2 = 1.0, 78.1e
sm_complex = 78.1e-1j
sm_bool_t, sm_bool_f = True, False
sm_str_q, sm_str_dq = 'kek', "lol"
```

- You can assign multiple variables at once:

```
a, b, c, d = 1, True, 1.0, "lol"
e = f = g = 3
```

Python Variables V (Operations)

- Python supports following numeric operations:
  - sum: `a + b` (5 + 2 = 7)
  - sub: `a - b` (5 - 2 = 3)
  - mul: `a * b` (5 * 2 = 10)
  - div: `a / b` (5 / 2 = 2.5)
  - int div: `a // b` (5 // 2 = 2)
  - mod: `a % b` (5 % 2 = 1)
- Complex > float > integers. Type of result is type of "bigger" type.

Python Variables VI (Operations)

- All numbers in Python (and all programming languages) are just sequences of bytes, so you can use logical operations (bit operations) over integers:
  - or: `` `a | b` `` (0x01 | 0x10 = 0x11)
  - and: `` `a & b` `` (0x01 & 0x10 = 0x00)
  - xor: `` `a ^ b` `` (0x01 ^ 0x10 = 0x11)
  - not: `` `~ a` `` (~ 0x01 = - 0x10)
  - lshift: `` `a << b` `` (0x01 << 1 = 0x10)
  - rshitf: `` `a >> b` `` (0x10 >> 1 = 0x08)

Python Variables VII (Operations)

- Python supports following logical operations between boolean values:
  - or: `a or b` (True or False = True)
  - and: `a and b` (True and False = False)
  - not: `not a` (not True = False; not False = True)
- Python strings support addition and multiplication:
  - sum: `a + b` ('kek' + 'lol' = 'keklol')
  - mul: `a * b` ('kek' * 3 = 'kekkekkek')
  - len: `not len(a)` (len('kek') = 3)

Python Variables VIII (Operations)

- You can convert numeric and string values to boolean values using following operators:
  - eq: `a == b` (5 == 2 = False)
  - neq: `a != b` (5 != 2 = True)
  - more `a > b` (5 > 2 = True)
  - more-eq `a >= b` (5 >= 2 = True)
  - less: `a < b` (5 < 2 = False)
  - less-eq: `a <= b` (5 <= 2 = False)

Python Conditional Operators

- Conditional operators allow to execute code depending on some condition. Let’s imagine that we want to implement code, which creates absolute value of number. We can do it in two ways.

if-elif-else

```
num = 10
abs_num = None
if num > 0:
    abs_num = num
elif num < 0:
    abs_num = -num
else:
    abs_num = 0
print(abs_num)
```

- or simpler:

```
num = 10
abs_num = None
if num > 0:
    abs_num = num
else:
    abs_num = -num
print(abs_num)
```

… if … else (ternary operator)

```
num = 10
abs_num = num if num > 0 else (-num if
num < 0 else 0)
print(abs_num)
```

- or simpler:

```
num = 10
abs_num = num if num > 0 else 0
print(abs_num)
```

`match-case` I

- Sometimes you just want to compare your variable value with some fixed values. You can do it via `` `if-elif-else` ``, however in Python 3.10 (be careful, only Python >= 3.10 (!!!)) supports better operator: `` `match-case` ``.
- `` `match-case` `` allows you to use powerful pattern-matching technique in Python. Originally it was used in functional programming languages.

match-case II

```
num = 10
abs_num = None
match (1 if num > 0 else (-1 if num <
0 else 0)):
case 1:
    abs_num = num
case -1:
    abs_num = -num
case _:
    abs_num = 0
print(abs_num)
```

Inner conditional operators I

- You can place one conditional operator inside another:

```
num = 10
if num > 0:
    if num > 5:
        new_num = num * 5
    else:
        new_num = num / 5
else:
    new_num = 0
```

Inner conditional operators II

- You can embed `match-case` into `if-else` too:

```
num = 10
if num > 0:
    match num:
        case 10:
            new_num = 100
        case _:
            new_num = num
else:
    new_num = 0
```

Inner conditional operators III

- You can embed `match-case` into every `if-elif-else` and every `if-elif-else` into every `match-case`.
- You can NOT embed anything into ternary operator (!!!)

Python Cycle Operators

- Cycle operators allow to execute code multiple times. You can think about cycles iterations like about code duplicates (complete loop unrolling optimization).
- Sometimes number of cycles is known during compile time (byte code generation), sometimes not.
- You have two main options for writing cycles in Python.
- You have to learn `itertools` library to write optimal loops in Python.

for-in-loop I

- basic loop:

```
for i in range(0, 10, 2):
    print(i)
> 0\n 2\n 4\n 6\n 8\n
```

- loop with continue:

```
for i in range(0, 10, 2):
    if (i % 3 == 0) and (i != 0):
        continue
    print(i)
> 0\n 2\n 4\n 8\n
```

`for-in`-loop II

- loop with break:

```
for i in range(0, 10, 2):
    if (i % 4 == 0) and (i != 0):
        break
    print(i)
> 0\n 2\n
```

- loop with else (how to determine if loop was exited by end of iterations or by break):

`for-in`-loop III

- loop with else:

```
for i in range(0, 10, 2):
    if (i % 4 == 0) and (i != 0):
        break
    print(i)
else:
    print('Exit on normal execution')
> 0\n 2\n
```

`for-in`-loop IV

- loop with else:

```
for i in range(0, 10, 2):
    if (i % 100 == 0) and (i != 0):
        break
    print(i)
else:
    print('Exit on normal execution')
> 0\n 2\n 4\n 6\n 8\n
> Exit on normal execution
```

`range` (sadly not rover)

- `` `range(0, n, step)` `` allows you to iterate through values from `0` to `n-1` with given step.
- `` `range(0, n)` `` allows you to iterate through values from `0` to `n-1` with step `1`.
- `` `range(n)` `` allows you to iterate through values from `0` to `n-1` with step `1`.

`while`-loop I

- `for-in`-loop is essential for iterating over ranges and sequences. You will know about sequences iteration in the next lesson. However, it does not provide full control over the loop. To get it you can use another kind of cycle operator: `while`.

```
while condition:
    do something
```

```
while days before session > 1:
    drink(beer), date(girls), have(fun)
```

`while`-loop II

- Example:

```
i = 0
while i < 10:
    print(i)
    i += 2
```

- `while` loops support `break`, `continue` and `else` operators:

...

`while`-loop III

```
i = 0
while i < 20:
    if (i % 3 == 0) and (i != 0):
        continue
    if (i % 10 == 0) and (i != 0):
        break
    print(i)
    i += 2
else:
    print('Exit on normal execution')
> 0\n 2\n 4\n ???
```

`while`-loop IV

`while`-loop V

```
i = 0
while i < 20:
    if (i % 3 == 0) and (i != 0):
        i += 2
        continue
    if (i % 10 == 0) and (i != 0):
        break
    print(i)
    i += 2
> 0\n 2\n 4\n 8\n
```

Inner loops I

- You can place one loop inside another:

```
for i in range(5):
    if i == 3:
        continue
    elif i == 4:
        break
    j = 0
    while j < i:
        print(j)
        j += 1
    ...
```

```
...
if j == 4:
    break
```

```
> 0\n 0\n 1\n
```

- Be careful (!!!): `break` and `continue` work only for one covering loop.

Inner loops II

- Python is an interpreted programming language and most of them (including our hero) are badly optimized for loops & multiple inner loops, so you have to always try to avoid them.
- We will talk about it later but most curious of you can read about Python `list`s, `tuple`s and `itertools` library at home :)

Inner loops II

- Python is an interpreted programming language and most of them (including our hero) are badly optimized for loops & multiple inner loops, so you have to always try to avoid them.
- We will talk about it later but most curious of you can read about Python `list`s, `tuple`s and `itertools` library at home :)

Variable Scopes I

- If you define variable in Python source code outside conditional operator's or loop's scopes, it will be available until program stops.
- If you define variable inside conditional operator's scope, it will be available in outer scope only if this part of conditional operator was executed:

```
if 5 > 1:
    if 4 > 2:
        a = 4
print(a)
> 4
```

```
if 1 > 5:
    a = 4
print(a)
> NameError: name 'a'
is not defined
```

Variable Scopes II

- If you define variable inside loop scope and loop will iterate through it, it will be available outside its scope:

```
for i in range(3):
    print(i)
print(i)
> 0\n 1\n 2\n 2\n
i = 0
while i < 1:
    i, kek = 1, -5
print(kek) > -5\n
```

```
i = 0
while i < 0:
    i, kek = 1, -5
print(kek)
> NameError: name
'kek' is not defined
```

Functions I

- We have already discussed functions call mechanism in Linux: it is based on stack. In Python's interpreter function call mechanism is mostly the same.
- Each function has from 0 to infinite number of parameters and from 0 to infinite number of output variables.
- Functions can call each other.
- Sometimes function can call itself (recursion).

Functions II

- Function can be define in Python via keyword `` `def` ``.
- Function input parameters are provided in parentheses: `` `(a, b, c, ...)` ``.
- Function return values are return via keyword `` `return` ``.
- Let's write simple function, which returns absolute value on number:

```
def abs(x):
    return x if x > 0 else -x
```

Functions III

- To call a function you just need to type its name with its parameters in parentheses. You have already experienced it with `print` function:

```
print('Hello, World!')
> Hello, World!
abs(10)
> 10
kek = abs(-5)
print(kek)
> 5
```

Functions IV

- Now let's implement function `sign`, which returns sign of a number. Then let's rewrite `abs` function using it:

```
def sign(x):
    return 1 if x > 0 else (-1 if x < 0 else 0)

def abs(x):
    return x if sign(x) > 0 else -x
```

- We did it to show how one function calls another.

Functions V

- Now let's implement function `` `all_ops` ``, which returns sum, sub, mul and div of two numbers:

```
def all_ops(x, y):
    return x + y, x - y, x * y, x / y
a, b, c, d = all_ops(10, 5)
print(a, b, c, d)
> 15 5 50 2.0
```

- This function shows how to pass/return multiple values to/from function.

Recursion I

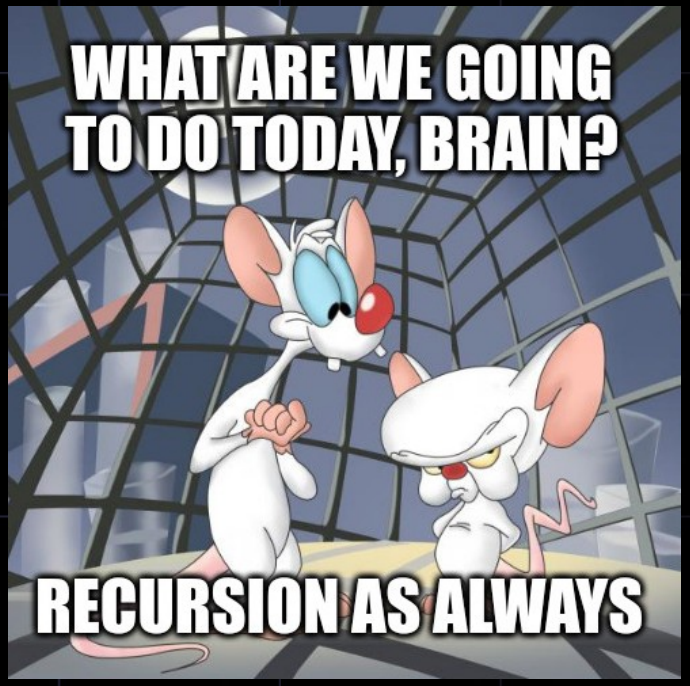
- In one of previous examples we called one function from another. Now let's call function from itself.
- Let's write a function for Fibonacci numbers calculation. Fibonacci sequence: 1 1 2 3 5 8 13 21...

```
def fibonacci(n):
    return 1 if n <= 2 else\
      fibonacci(n-1) + fibonacci(n-2)

for i in range(1, 9):
    print(fibonacci(i))
> 1\n 1\n 2\n 3\n 5\n 8\n 13\n 21\n
```

Recursion II

- You can add `print` calls to `fibonacci` function code so you can see how does it calculates given number.
- Now we are going to make recursion with two functions.



Recursion III

```
def kek(n):
    print('kek', n)
    if n == 0:
        return 0
    return lel(n-1)
```

```
def lel(n):
    print('lel', n)
    if n == 0:
        return 0
    return kek(n)
```

```
kek(5)
> kek 5\n lel 4\n kek 4\n lel 3\n kek
3\n lel 2\n kek 2\n lel 1\n kek 1\n
lel 0\n 0
```


Game over.