Interpretador adaptativo especializado

Por que as versões novas do python estão mais rápidas?

Live de Python #272

Roteiro

1. Um contexto anterior

Uma explicação básica pra gente poder entender junto :)

2. Versão 3.11 e suas novidades

Interpretador adaptativo especializado

3. Versão 3.12 e o aperfeiçoamento

A nova DSL, Aceleração inicial e mais adaptações

4. Versão 3.13 o início de um sonho

Superblocos, Tier 2, Micro-ops e compilador JIT

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3

Ademar Peixoto, Adriana Cavalcanti, Adriano Casimiro, Alexandre Girardello, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Allan Kleitson, Alysson Oliveira, Andre, Andre Azevedo, Andre Makoski, Andre Paula, Apc 16, Aslay Clevisson, Aurelio Costa, Ayr-ton, Bernarducs, Brisa Nascimento, Bruno Barcellos, Bruno Batista, Bruno Freitas, Bruno Lira, Bruno Ramos, Bruno Santos, Caio Nascimento, Carlos Gonçalves, Carlos Ramos, Cecilia Oliveira, Christian Fischer, Daniel Aguiar, Daniel Bianchi, Daniel Brito, Daniel Real, Daniel Souza, Daniel Wojcickoski, Danillo Ferreira, Danilo Boas, Danilo Silva, David Couto, David Kwast, Denis Bernardo, Dgeison Peixoto, Diego Guimarães, Dino, Edgar, Eduardo Oliveira, Elton Guilherme, Emerson Rafael, Ennio Ferreira, Érico Andrei, Everton Silva, Fabio Barros, Fábio Barros, Fabio Faria, Fabiokleis, Fabricio Biazzotto, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Filipe Oliveira, Francisco Aclima, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Ramos, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Giovanna Teodoro, Giuliano Silva, Guilherme Felitti, Guilherme Ostrock, Guilherme Piccioni, Guilherme Silva, Gustavo Suto, Haelmo Almeida, Harold Gautschi, Heitor Fernandes, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Henrique Sebastião, Hiago Couto, Igor Cruz, Igor Taconi, Ivan Santiago, Jairo Lenfers, Janael Pinheiro, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jlx, Joao Rocha, Jonas Araujo, Jonatas Leon, Jônatas Silva, Jorge Silva, Jornada Filho, Jose Barroso, Jose Edmario, José Gomes, Joseíto Júnior, Jose Mazolini, Jose Terra, Josir Gomes, Jrborba, Juan Felipe, Juliana Machado, Julio Batista-silva, Julio Franco, Júlio Sanchez, Kael, Kaio Peixoto, Leandro O., Leandro Pina, Leandro Vieira, Leonan Ferreira, Leonardo Alves, Leonardo Mello, Leonardo Nazareth, Letícia Sampaio, Logan Merazzi, Lucas Carderelli, Lucas Castro, Lucas Lattari, Lucas Mello, Lucas Mendes, Lucas Moura, Lucas Nascimento, Lucas Oliveira, Lucas Schneider, Lucas Simon, Luciano Ratamero, Luciano Teixeira, Luiz Carlos, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Mackilem Laan, Marcelo Araujo, Marcelo Fonseca, Marcio Freitas, Marcio Junior, Marco Gandra, Marco Mello, Marcos Almeida, Marcos Gomes, Marcos Oliveira, Marcos Rodrigues, Marina Passos, Marlon Rocha, Mateusamorim96, Mateus Lisboa, Matheus Vian, Mírian Batista, Mlevi Lsantos, Mugiwara Luffy, Murilo Carvalho, Murilo Viana, Ocimar Zolin, Otávio Carneiro, Patrick Felipe, Pedro Henrique, Peterson Santos, Philipe Vasconcellos, Phmmdev, Prof Santana, Pydocs Pro, Pytonyc, Rafael Araújo, Rafael Costa, Rafael Faccio, Rafael Ferreira, Rafael Paranhos, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Renan, Renan Sebastião, Renã Pedroso, Renato José, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Costa, Rinaldo Magalhaes, Rodrigo Barretos, Rodrigo Oliveira, Rodrigo Vaccari, Rodrigo Vieira, Rogério Nogueira, Rui Jr, Samanta Cicilia, Santhiago Cristiano, Selmison Miranda, Shinodinho, Tales Ribeiro, Téo Calvo, Tharles Andrade, Thedarkwithin, Thiago Araujo, Thiago Borges, Thiago Lucca, Thiago Paiva, Tiago, Tiago Henrique, Tomás Tamantini, Tony Dias, Tyrone Damasceno, Valdir, Varlei Menconi, Victor Reis, Vinicios Lopes, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vitor Silva, Vladimir Lemos, Wagner Gabriel, Williamslews, Willian Lopes, Zeca Figueiredo

Obrigado você

- Live totalmente informativa

Avisos

- Live totalmente informativa
- Objetivo de explicar de forma simples

FAÇA PERGUNTAS!

Avisos

- Live totalmente informativa
- Objetivo de explicar de forma simples
- **Foco em pessoas não cientistas da computação**

Papers serão citados para quem quiser se aprofundar no assunto!

Avisos

- Live totalmente informativa
- Objetivo de explicar de forma simples
- **Foco em pessoas não cientistas da computação**
- **Não falaremos sobre o JIT, iremos até ele!**

Considere uma introdução ao JIT

Avisos

Uma breve revisão :)

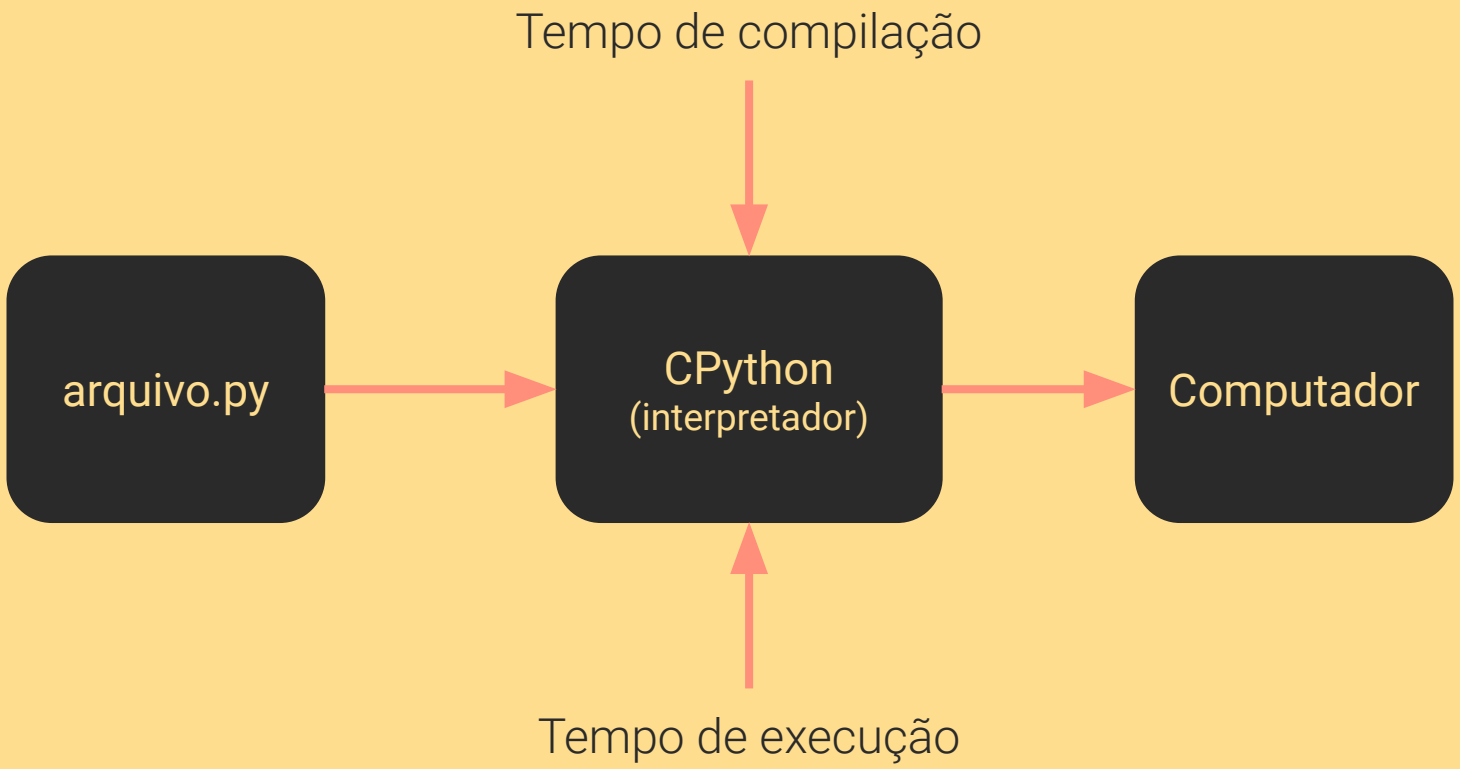
Um contexto antes

O que aconteceu?

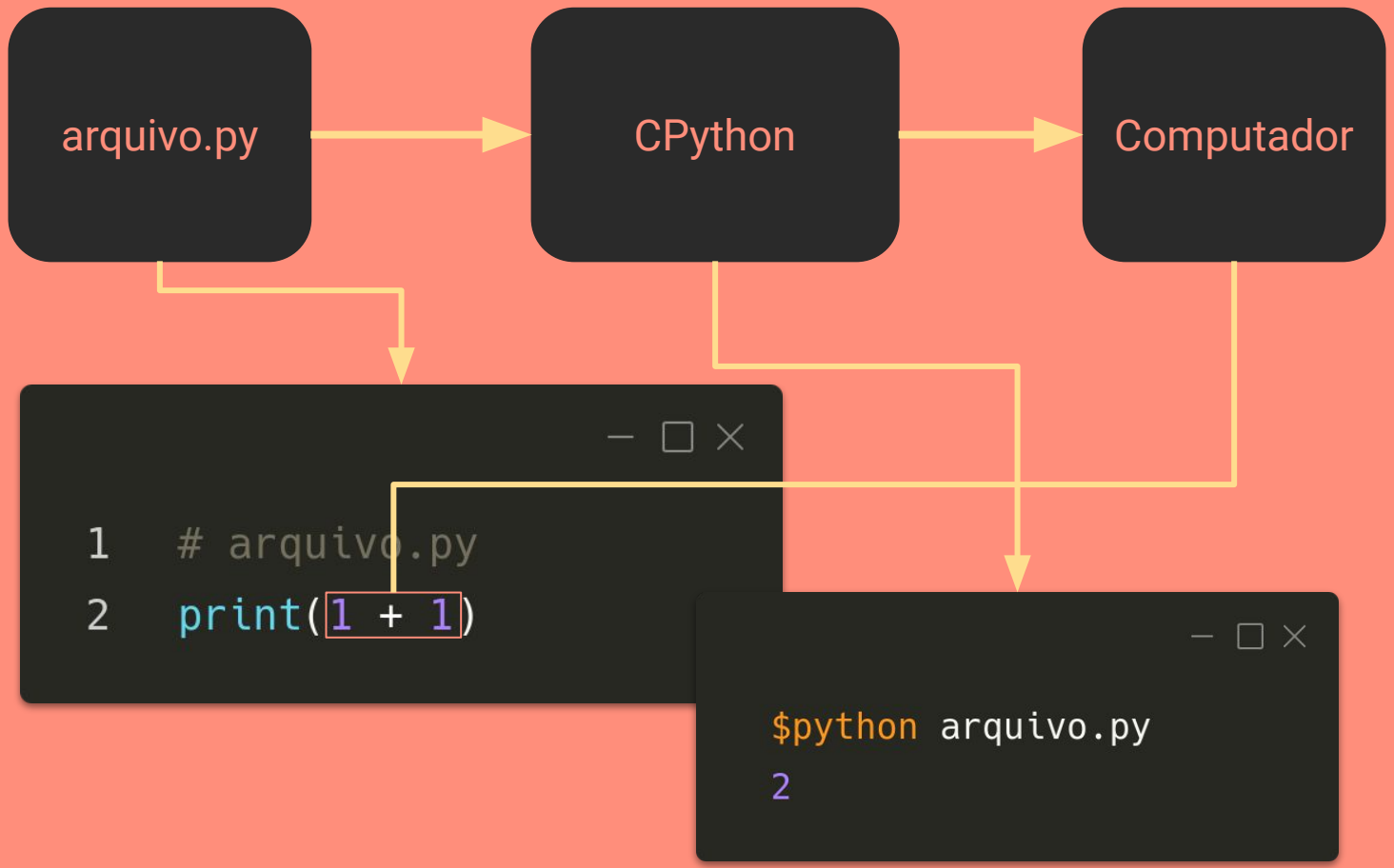
O interpretador na versão **3.11** do python faz melhorias significativas de desempenho.

- A versão na totalidade tem melhorias entre 10%~60% de velocidade
- As melhorias são categorizadas em duas:
  - A inicialização do interpretador
    - Foi otimizada entre 10%~15%
  - Execução de código (runtime)
    - O assunto dessa live :)

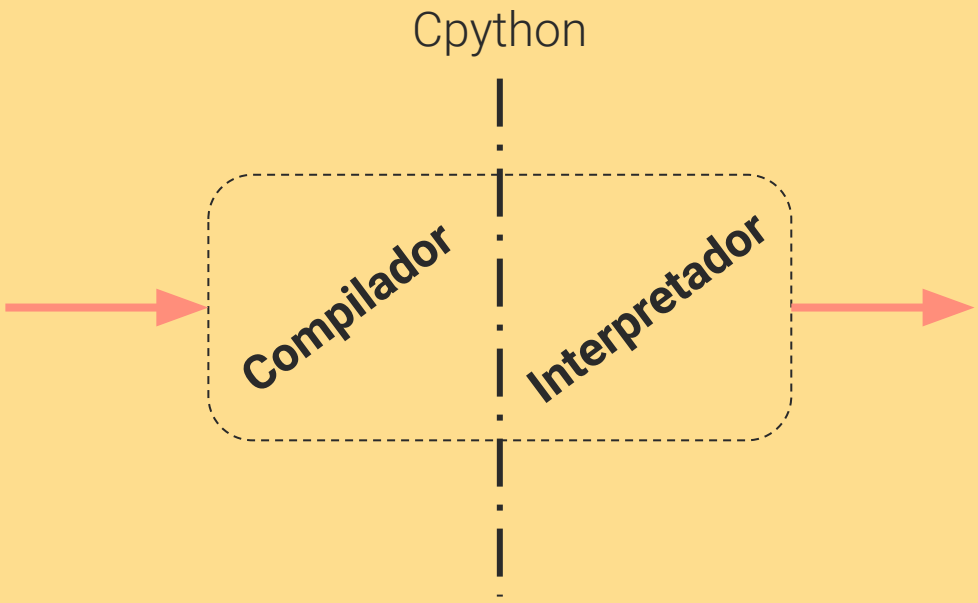
Uma visão do CPython [0]



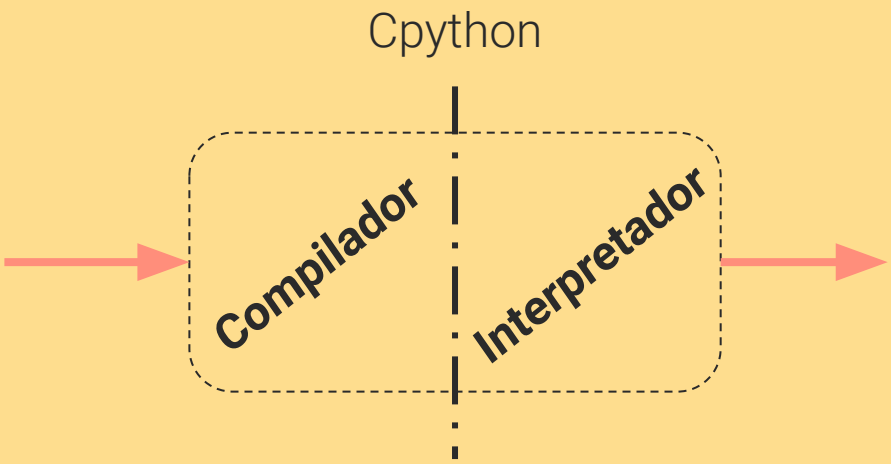
Uma visão do CPython [1]



Uma visão do CPython [2]

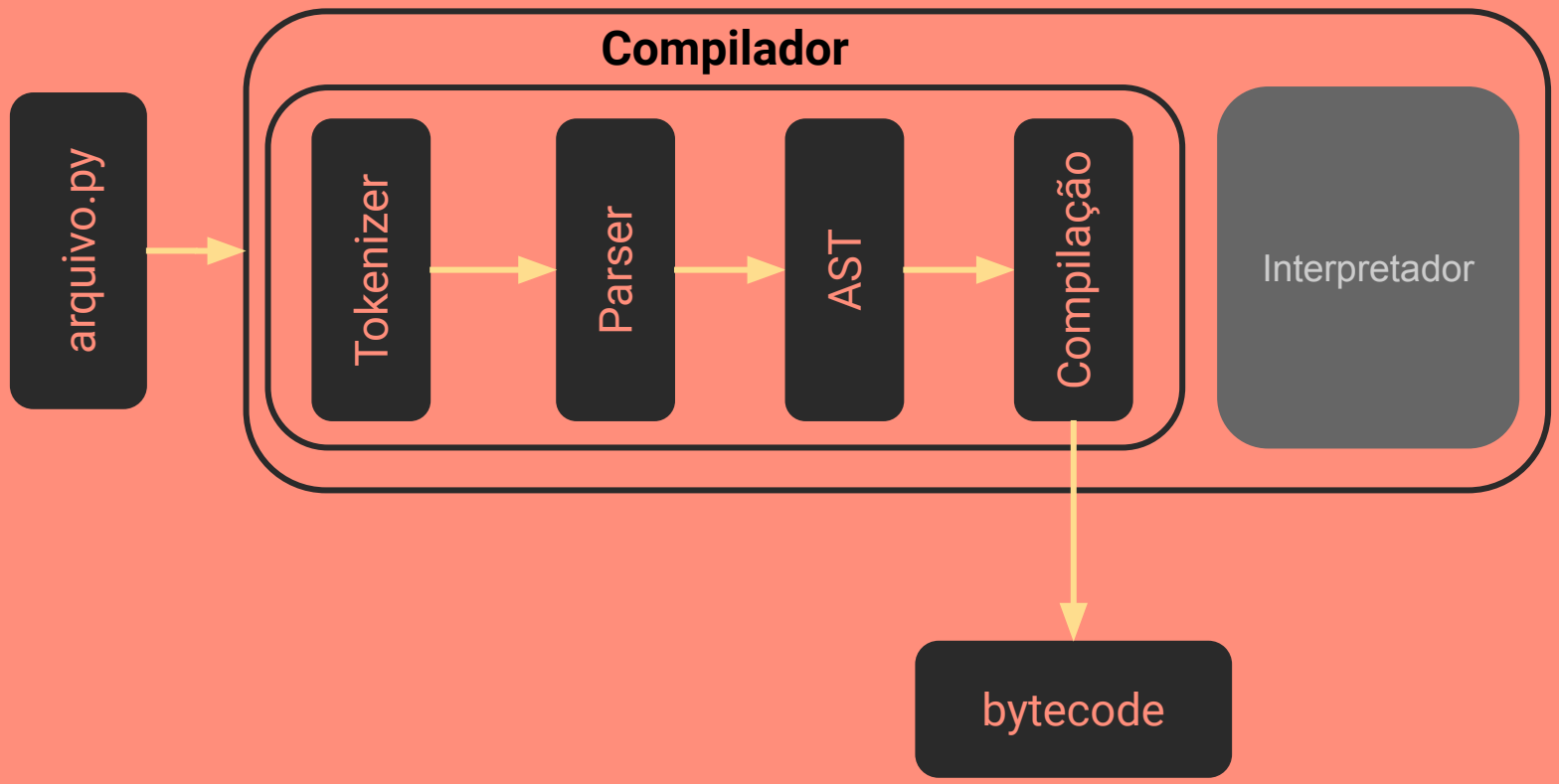


Uma visão do CPython [2]

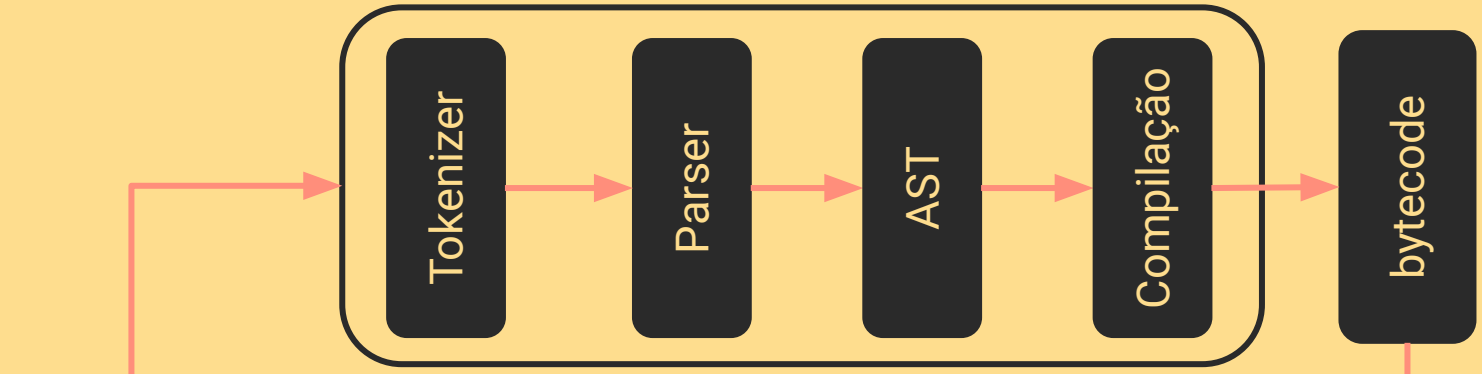


https://youtu.be/pxfZTAJDipY

Uma visão geral sobre o processo de compilação



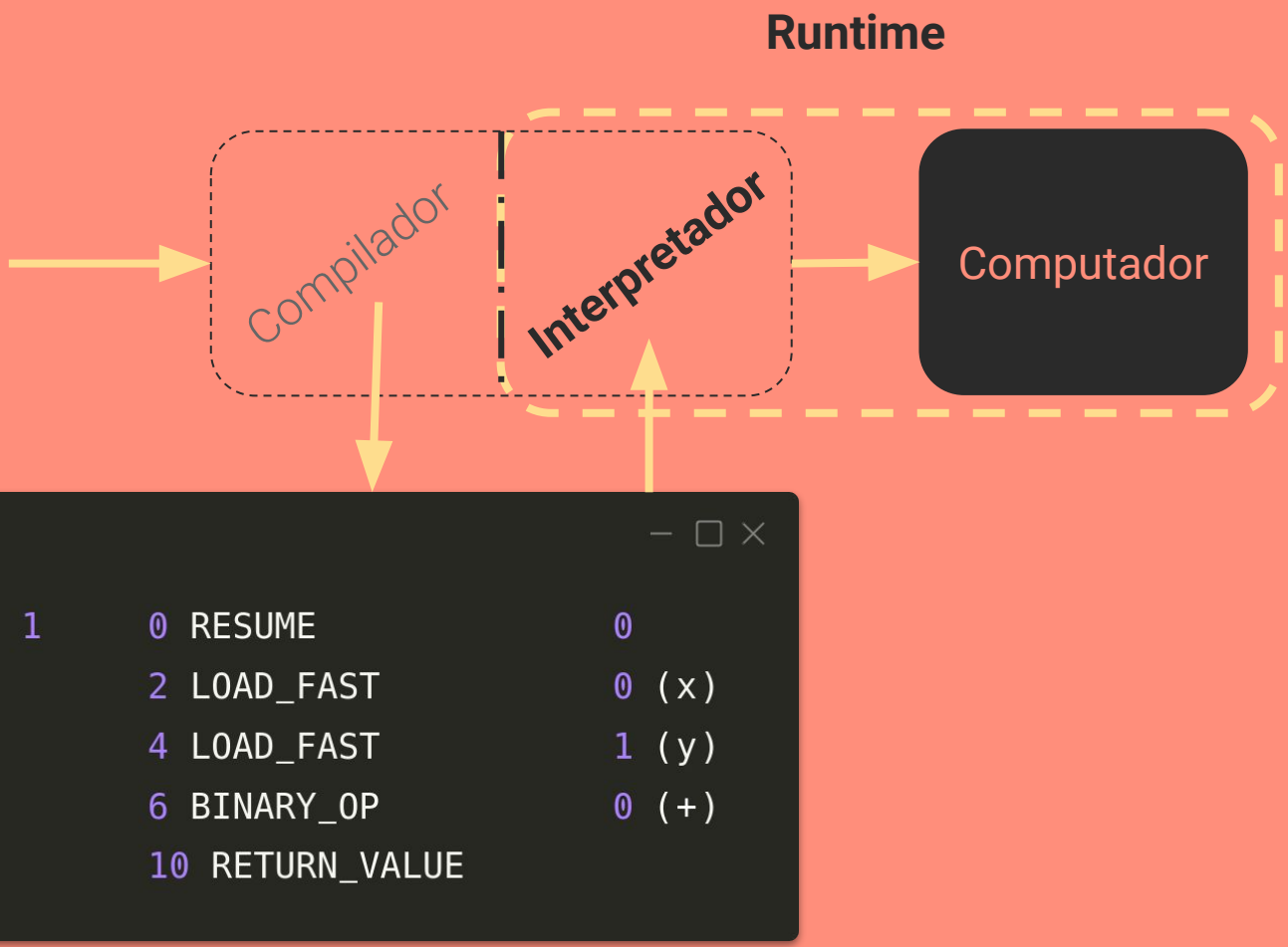
O resultado da compilação




```
1   def add(x, y):
2       return x + y
```

```
1       0 RESUME                   0

        2 LOAD_FAST                0 (x)
        4 LOAD_FAST                1 (y)
        6 BINARY_OP                0 (+)
       10 RETURN_VALUE
```

O runtime


Runtime
Compilador
Interpretador
Computador
1 0 RESUME 0
2 LOAD_FAST 0 (x)
4 LOAD_FAST 1 (y)
6 BINARY_OP 0 (+)
10 RETURN_VALUE

O processo de execução

Função **add** em bytecode

```
1       0 RESUME                   0
        2 LOAD_FAST                0 (x)
        4 LOAD_FAST                1 (y)
        6 BINARY_OP                0 (+)
       10 RETURN_VALUE
```

https://docs.python.org/pt-br/3/library/dis.html#python-bytecode-instructions

Memória

Stack

O processo de execução

Função **add** em bytecode

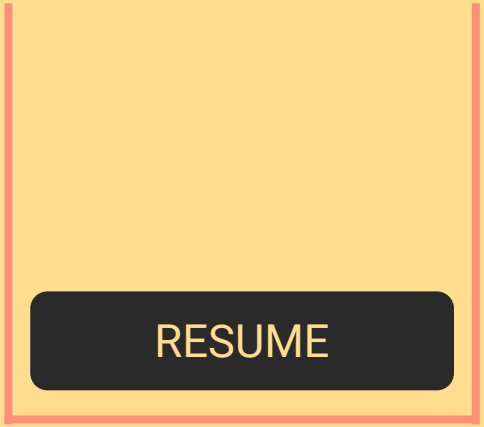
```
1           0 RESUME                   0
            2 LOAD_FAST                0 (x)
              LOAD_FAST                1 (y)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

Uma instrução nova, introduzida no 3.11.

Nosso objeto de estudo!

Memória

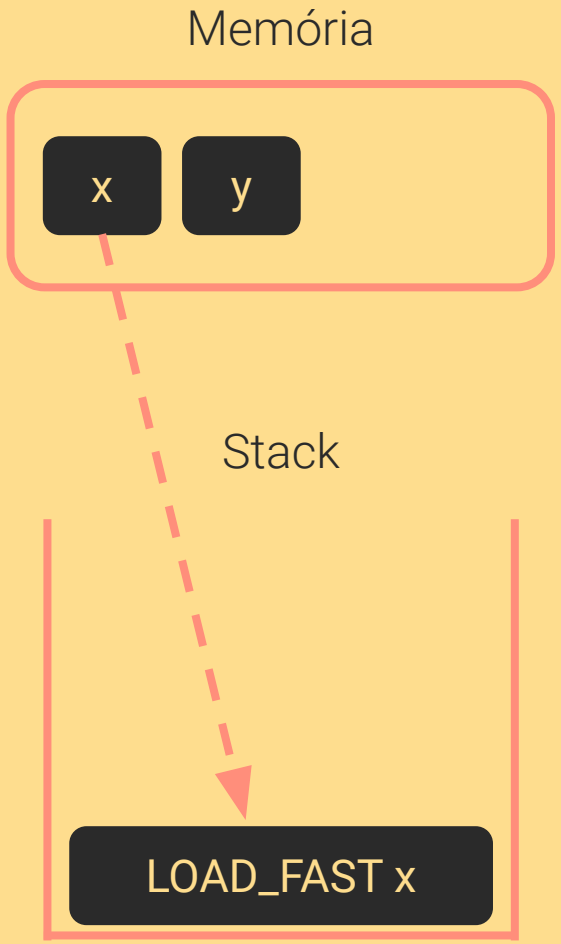
Stack



O processo de execução

Função **add** em bytecode

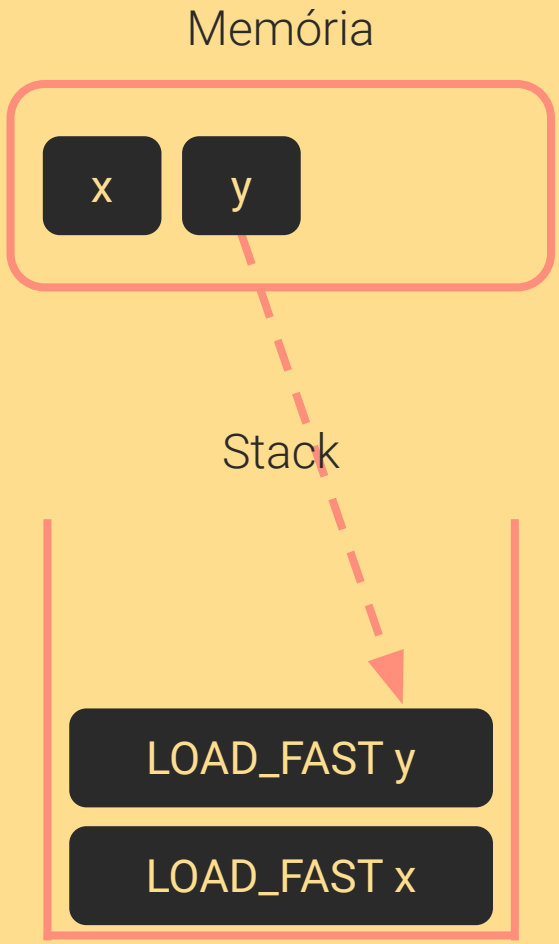
```
1           0 RESUME                   0

            2 LOAD_FAST                0 (x)
            4 LOAD_FAST                1 (y)
            6 BINARY_OP                0 (+)
           10 RETURN_VALUE
```



O processo de execução

Função **add** em bytecode

```
1       0 RESUME                   0
        2 LOAD_FAST                0 (x)
        4 LOAD_FAST                1 (y)
        6 BINARY_OP                0 (+)
       10 RETURN_VALUE
```



O processo de execução

Função **add** em bytecode

```
1       0 RESUME                   0
        2 LOAD_FAST                0 (x)
        4 LOAD_FAST                1 (y)
        6 BINARY_OP                0 (+)
       10 RETURN_VALUE
```

```
rhs = STACK.pop()
lhs = STACK.pop()
STACK.append(lhs op rhs)
```

Memória

Stack

O processo de execução

Função **add** em bytecode

```
1       0 RESUME                   0
        2 LOAD_FAST                0 (x)
        4 LOAD_FAST                1 (y)
        6 BINARY_OP                0 (+)
       10 RETURN_VALUE
```

```
rhs = STACK.pop()
lhs = STACK.pop()
STACK.append(lhs op rhs)
```

Memória

Stack

O processo de execução

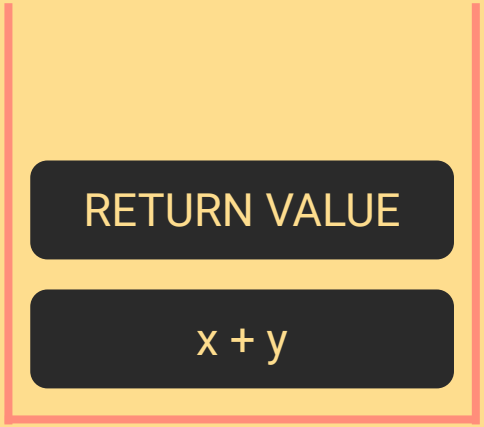
Função **add** em bytecode

```
1      0 RESUME                0
       2 LOAD_FAST             0 (x)
       4 LOAD_FAST             1 (y)
       6 BINARY_OP             0 (+)
      10 RETURN_VALUE
```

Retorna STACK[-1]

Memória

Stack



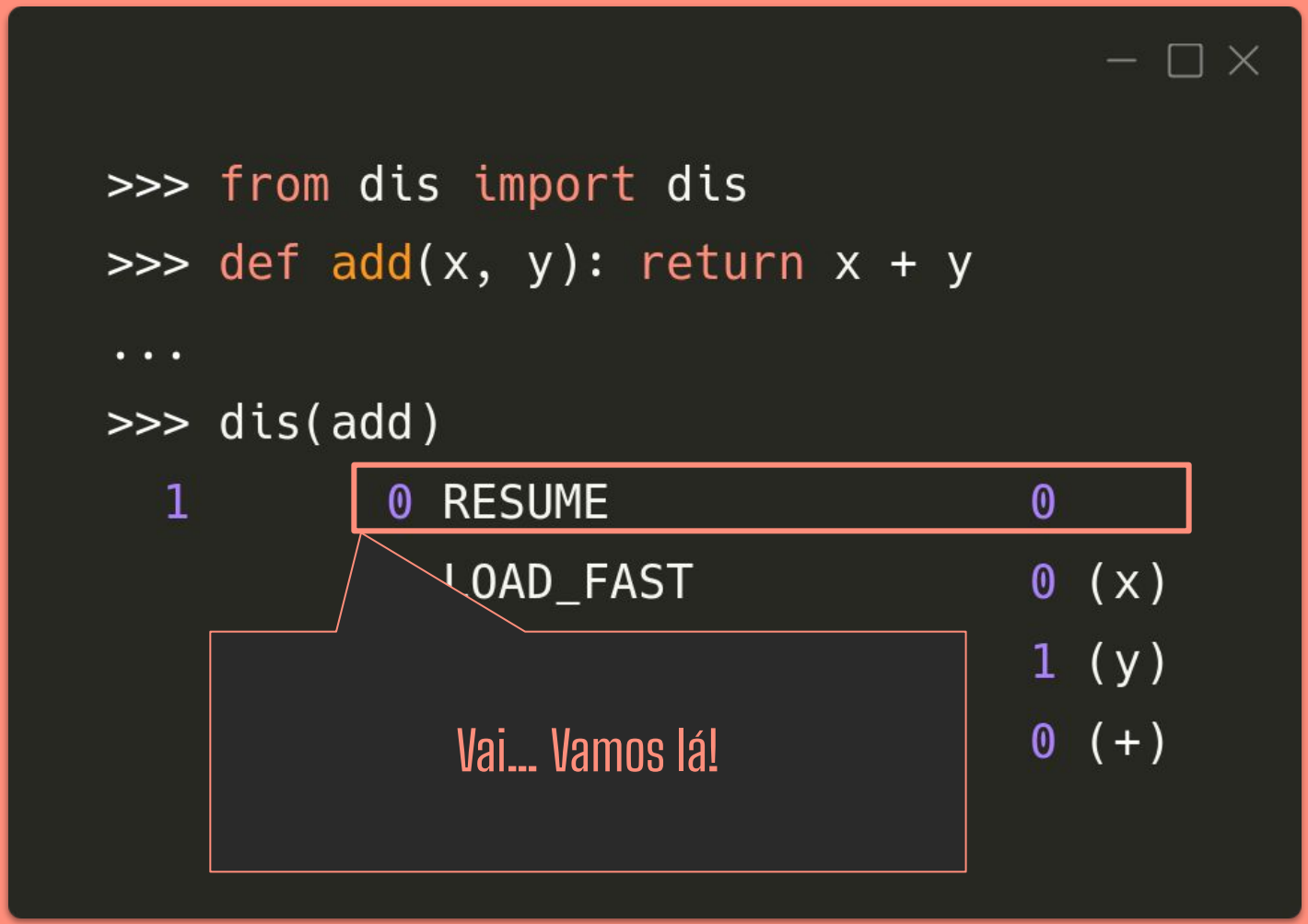
Como ver o bytecode do seu código? [shell]

```
python -m dis exemplo.py

  0        0 RESUME                0
  2        2 LOAD_CONST            0 (<code object add at 0x7b3, file "exemplo.py", line 2>)
           4 MAKE_FUNCTION         0
           6 STORE_NAME            0 (add)
 13        8 LOAD_CONST            1 (None)
          10 RETURN_VALUE

Disassembly of <code object add at 0x7b3, file "exemplo.py", line 2>:
  2        0 RESUME                0
  9        2 LOAD_FAST             0 (x)
           4 LOAD_FAST             1 (y)
           6 BINARY_OP             0 (+)
          10 RETURN_VALUE
```

Como ver o bytecode do seu código? [repl]

```
>>> from dis import dis
>>> def add(x, y): return x + y
...
>>> dis(add)
  1           0 RESUME                   0

              2 LOAD_FAST                0 (x)
              4 LOAD_FAST                1 (y)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
```

Como ver o bytecode do seu código? [repl]



Interpretador adaptativo especializado

3.11

RESUME

A operação **RESUME** foi introduzida no python 3.11 com algumas responsabilidades específicas.

A documentação define **RESUME** como uma **não operação**. Ela exerce algumas funções internas do interpretador. Como:

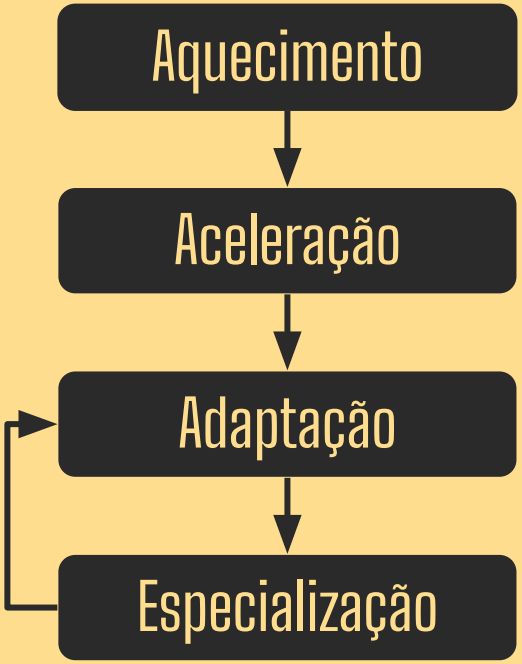
- Debug
- Rastreamento (*tracing*) e
- **Checagem de otimizações**

PEP 659: Interpretador adaptativo especializado

A ideia central dessa PEP é nomear e padronizar uma forma de "alterar" o código gerado pelo processo de compilação durante o tempo de execução ("recompilação").

Isso é divido em 4 partes:

- Aquecimento (*warmup*)
- Aceleração (*quickening*)
- Adaptação (*Adaptive*)
- Especialização (*Specializing*)

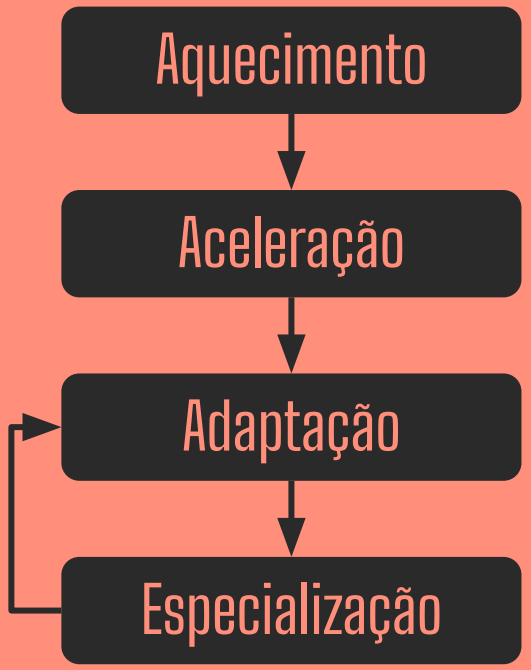


Aquecimento (*warm up*)

O **objeto do bytecode**, quando iniciado pelo interpretador, adiciona um contador chamado `**warmup**` (cujo o valor é -8).

Todas as vezes em que a *não* instrução **RESUME** é lida pelo interpretador, **+1** é adicionado ao valor **warmup**.

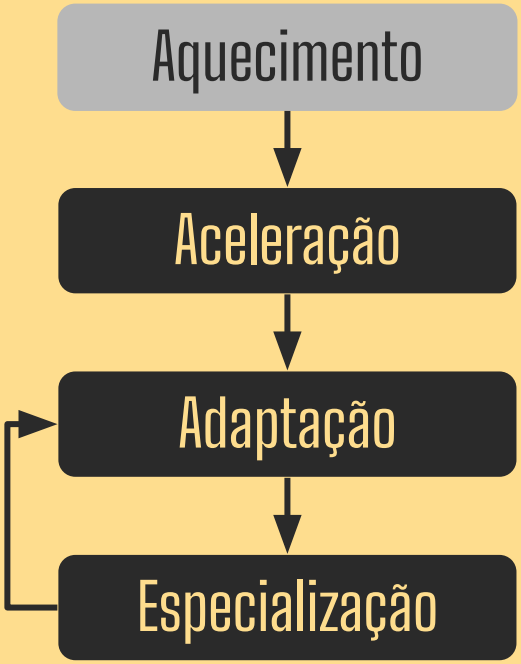
Quando **warmup** for igual a zero começa o processo de Aceleração (*quickening*).



Aceleração (*quickening*)

No processo de aceleração o interpretador **vai analisar as instruções** do bytecode e **procurar por chamadas que ele pode otimizar**.

Para instruções de **alocação** (STORE_FAST) e **recuperação** (LOAD_FAST), são criadas **superinstruções**.



Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters

KEVIN CASEY
Trinity College Dublin
M. ANTON ERTL
TU Wien
and
DAVID GREGG
Trinity College Dublin

2003 - 10.1145/1286821.1286828

O resultado do aquecimento

```
>>> from dis import dis
>>> def add(x, y): return x + y
...
>>> [add(1, 1) for _ in range(8)]
[2, 2, 2, 2, 2, 2, 2, 2]
```

Aquecendo!

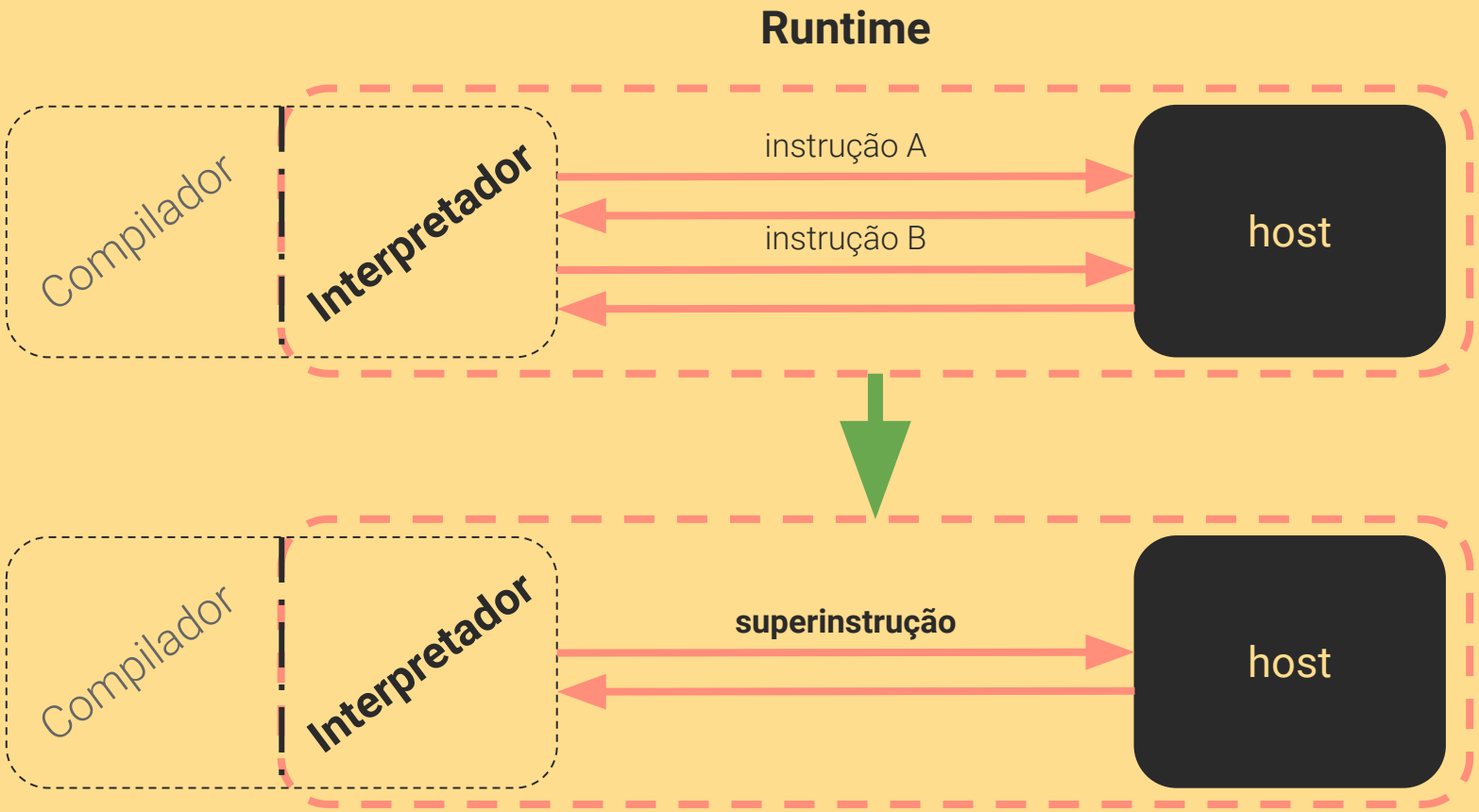
Superinstrução

Adaptação

```
>>> dis(add, adaptive=True)
  1           0 RESUME                   0

              2 LOAD_FAST__LOAD_FAST     0 (x)
              4 LOAD_FAST                1 (y)
              6 BINARY_OP_ADD_ADAPTIVE   0 (+)
             10 RETURN_VALUE
```

Superinstruções

A ideia das superinstruções é **combinar N instruções** (nessa versão somente duas) **em uma única instrução**.



```
//specialize.c
void  // 259
_PyCode_Quicken(PyCodeObject *code)
{
  // Lista e itera sobre os opcodes
  case RESUME:
      _Py_SET_OPCODE(instructions[i], RESUME_QUICK);
      break;
```

```
case LOAD_FAST:
    switch(previous_opcode) {
        case LOAD_FAST:
            _Py_SET_OPCODE(instructions[i - 1],
                           LOAD_FAST__LOAD_FAST);
            break;
        case STORE_FAST:
            _Py_SET_OPCODE(instructions[i - 1],
                           STORE_FAST__LOAD_FAST);
            break;
        case LOAD_CONST:
            _Py_SET_OPCODE(instructions[i - 1],
                           LOAD_CONST__LOAD_FAST);
            break;
    }
```

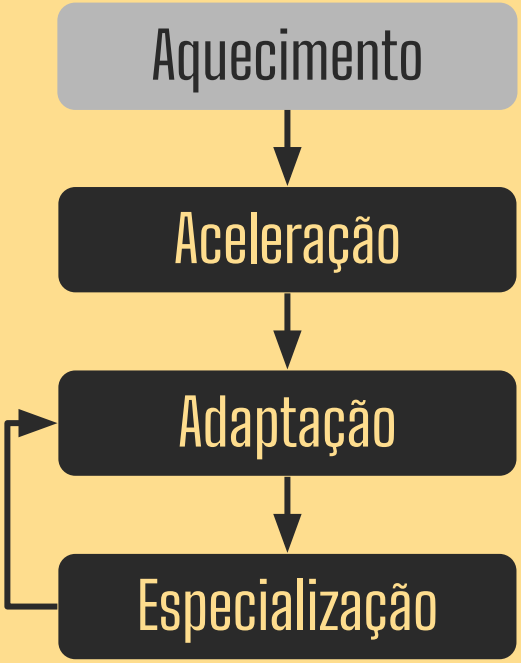
Todas as combinações entre **LOAD_FAST** e **STORE_FAST*** Se tornarão superinstruções

*: no restante do arquivo você pode ver os outros cases para cada instrução

Aceleração (*quickening*)

Outro ponto da aceleração é a troca de **operações genéricas por instruções adaptativas**.

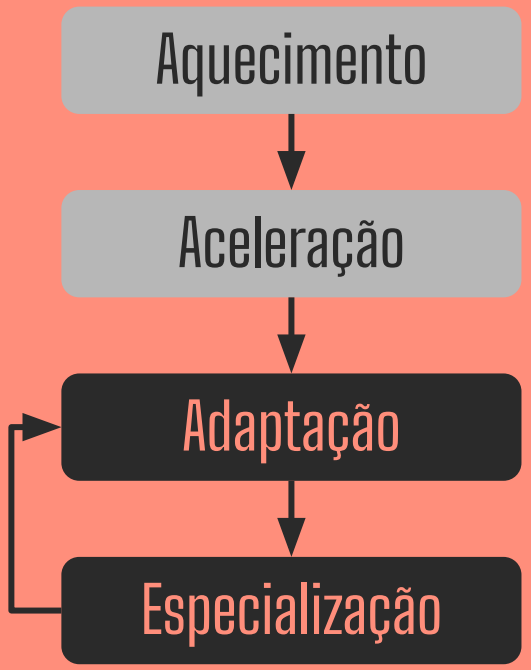
```c
// specialize.c
uint8_t _PyOpcode_Adaptive[256] = { // 20
    [LOAD_ATTR] = LOAD_ATTR_ADAPTIVE,
    [LOAD_GLOBAL] = LOAD_GLOBAL_ADAPTIVE,
    [LOAD_METHOD] = LOAD_METHOD_ADAPTIVE,
    [BINARY_SUBSCR] = BINARY_SUBSCR_ADAPTIVE,
    [STORE_SUBSCR] = STORE_SUBSCR_ADAPTIVE,
    [CALL] = CALL_ADAPTIVE,
    [PRECALL] = PRECALL_ADAPTIVE,
    [STORE_ATTR] = STORE_ATTR_ADAPTIVE,
    [BINARY_OP] = BINARY_OP_ADAPTIVE,
    [COMPARE_OP] = COMPARE_OP_ADAPTIVE,
    [UNPACK_SEQUENCE] = UNPACK_SEQUENCE_ADAPTIVE,
};
```



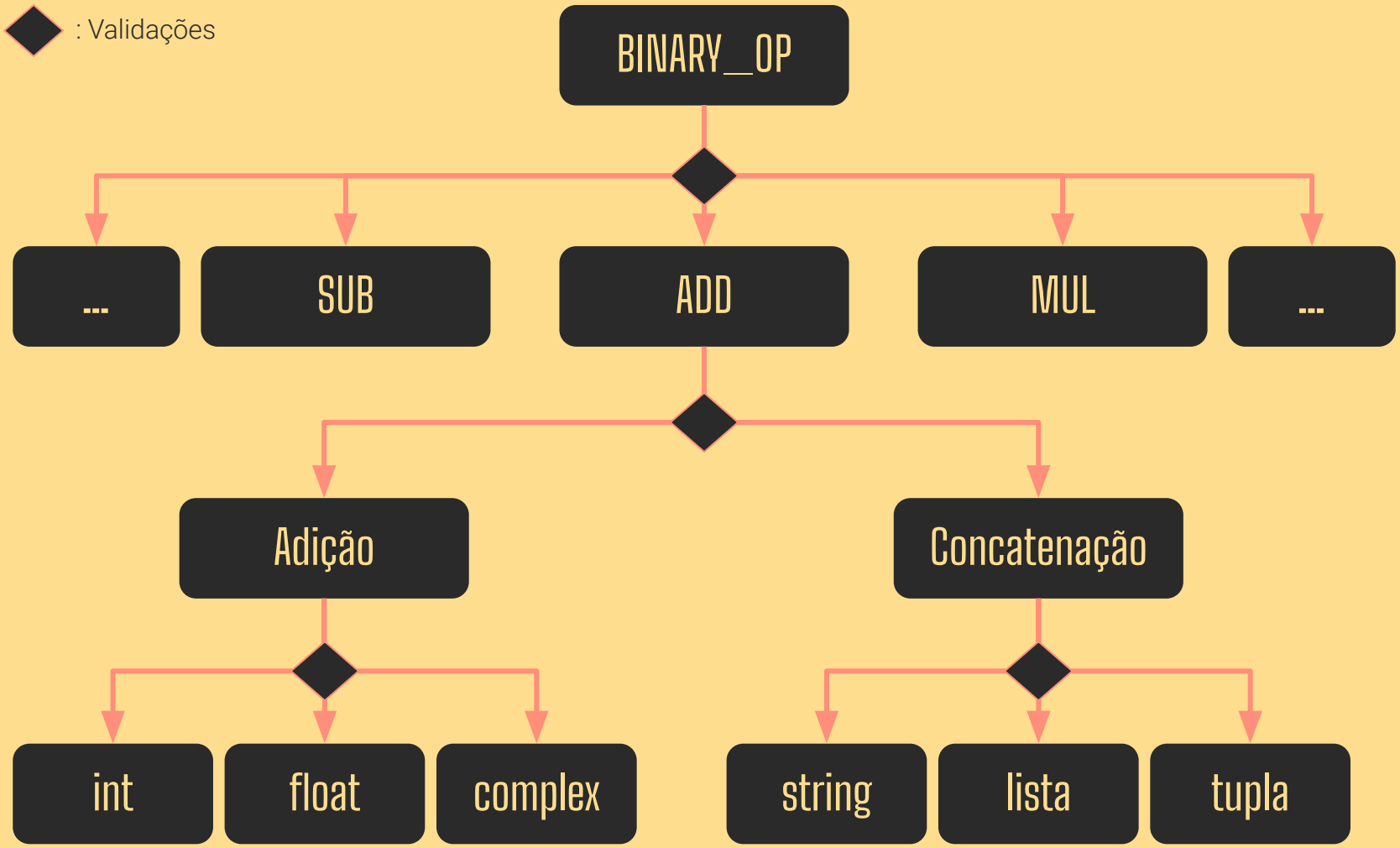
Adaptação

O processo de adaptação é um **estado intermediário de instruções** de bytecode.

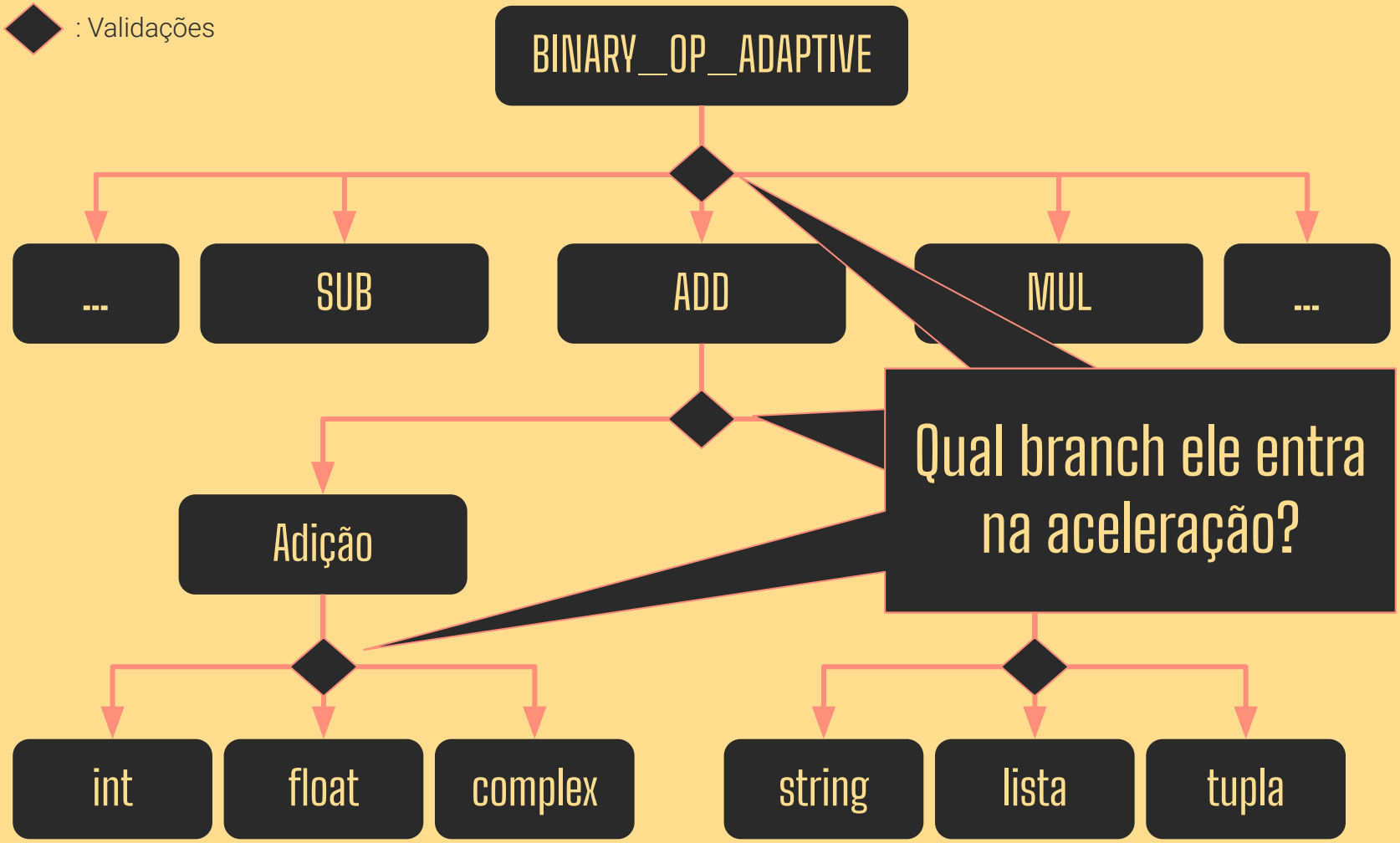
É o momento onde o interpretador troca uma **instrução genérica** por uma instrução "preditivo-adataptada".



O problema das instruções genéricas (BINARY_OP)



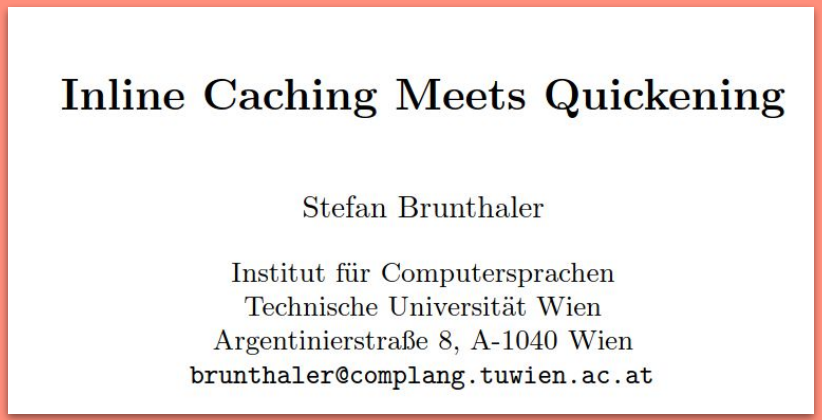
Predição em instruções (BINARY_OP_ADAPTIVE)



Adaptação - inline cache [Cache Embutido]

As instruções adaptativas contam com uma nova instrução de bytecode que as segue: **CACHE**.

A instrução cache tem um contador para dizer se a operação será especializada ou não!

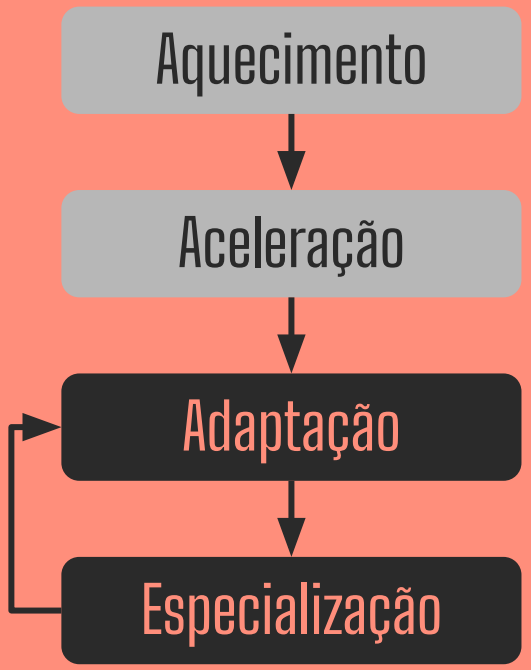


Inline Caching Meets Quickening

Stefan Brunthaler

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
brunthaler@complang.tuwien.ac.at

2010 - 10.5555/1883978.1884008



Cache embutido

```
>>> from dis import dis
>>> def add(x, y): return x + y
...
>>> [add(1, 1) for _ in range(8)]
[2, 2, 2, 2, 2, 2, 2, 2]
```

Aquecendo!

Cache

```
>>> dis(add, adaptive=True, show_caches=True)
  1           0 RESUME                   0

              2 LOAD_FAST__LOAD_FAST     0 (x)
              4 LOAD_FAST                1 (y)
              6 BINARY_OP_ADD_ADAPTIVE   0 (+)
              8 CACHE                    0 (counter: 1)
             10 RETURN_VALUE
```

O cache embutido

A ideia é do cache é armazenar estados relacionados a instruções adaptativas e especializadas em uma nova instrução de bytecode.

Os caches podem armazenar diversos valores. Mas por padrão todos ofertam a variável `counter`, que é o controle de adaptação.

```
>>> dis(add, adaptive=True, show_caches=True)
  1           0 RESUME                   0

              2 LOAD_FAST__LOAD_FAST     0 (x)
              4 LOAD_FAST                1 (y)
              6 BINARY_OP_ADD_ADAPTIVE   0 (+)
              8 CACHE                    0 (counter: 1)
             10 RETURN_VALUE
```

Como o contador é usado?

```
  //c.eval
TARGET(BINARY_OP_ADAPTIVE) {
    assert(cframe.use_tracing == 0);
    _PyBinaryOpCache *cache = (_PyBinaryOpCache *)next_instr;
    if (ADAPTIVE_COUNTER_IS_ZERO(cache)) {
        PyObject *lhs = SECOND();
        PyObject *rhs = TOP();
        next_instr--;
        _Py_Specialize_BinaryOp(lhs, rhs, next_instr, oparg, &GETLOCAL(0));
        DISPATCH_SAME_OPARG();
    }
    else {
        STAT_INC(BINARY_OP, deferred);
        DECREMENT_ADAPTIVE_COUNTER(cache);
        JUMP_TO_INSTRUCTION(BINARY_OP);
    }
}
```

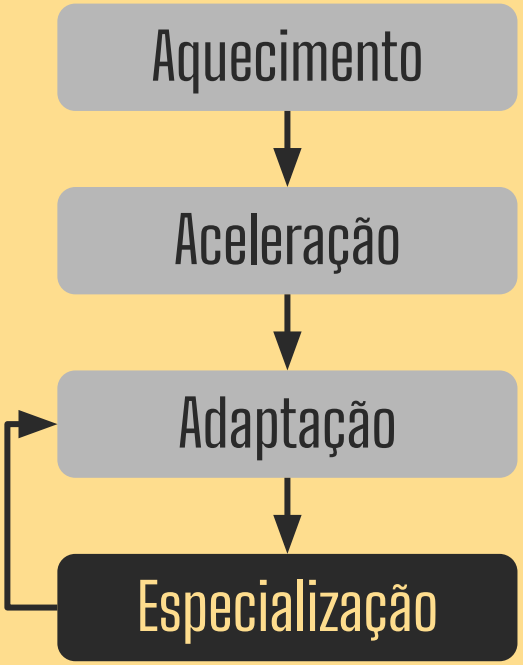
Se counter == 0; especializa

Senão; decrementa

Especialização

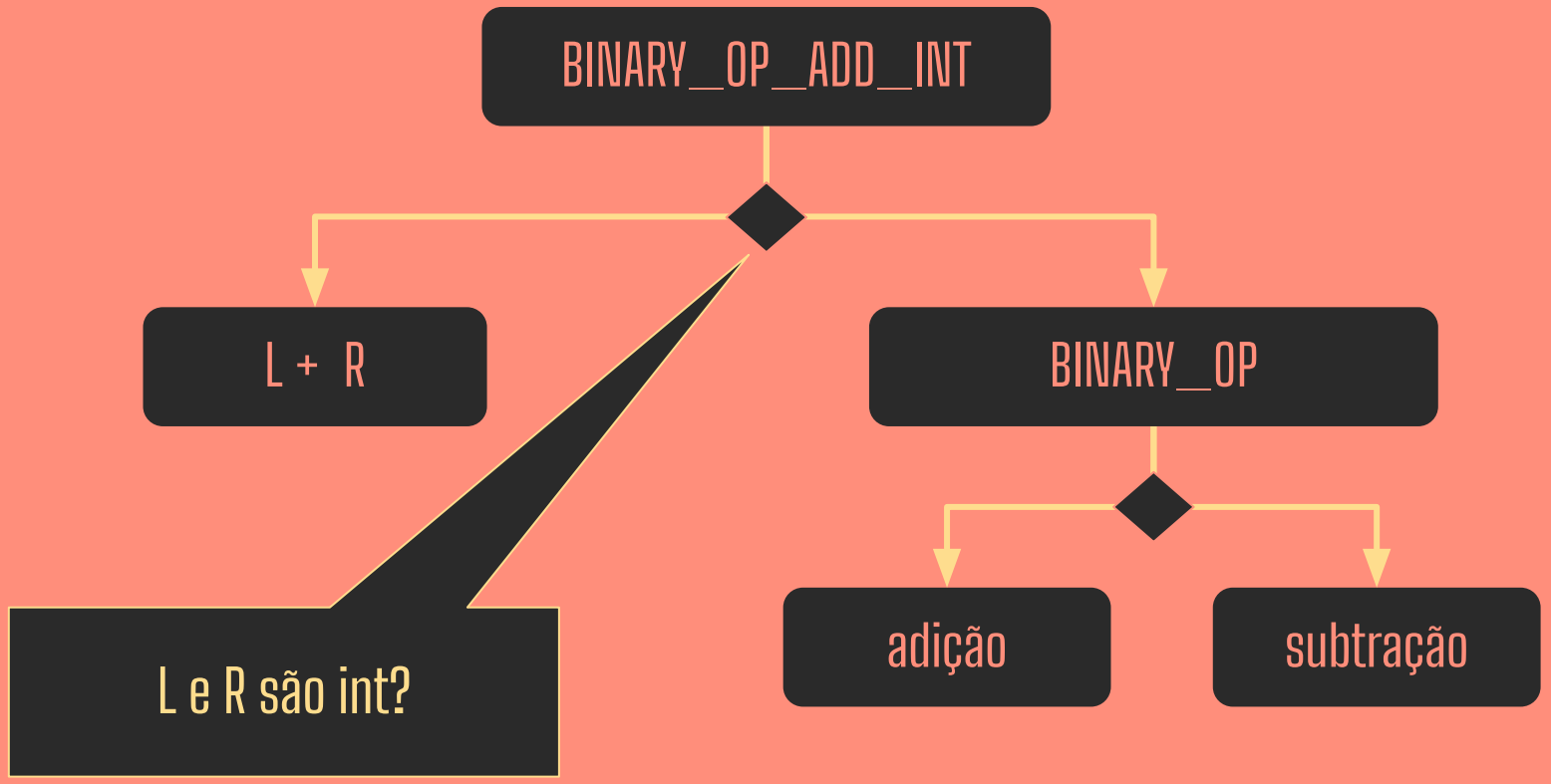
A ideia da especialização é pular mais etapas de checagem do interpretador e trocar a instrução do bytecode para fazer a **operação exata a requerida** pelo código.

Se a soma é entre dois números inteiros, a operação será adaptativa será trocada por um bytecode que faz exatamente isso.



BINARY_OP_ADD_INT e fallback

: Validações

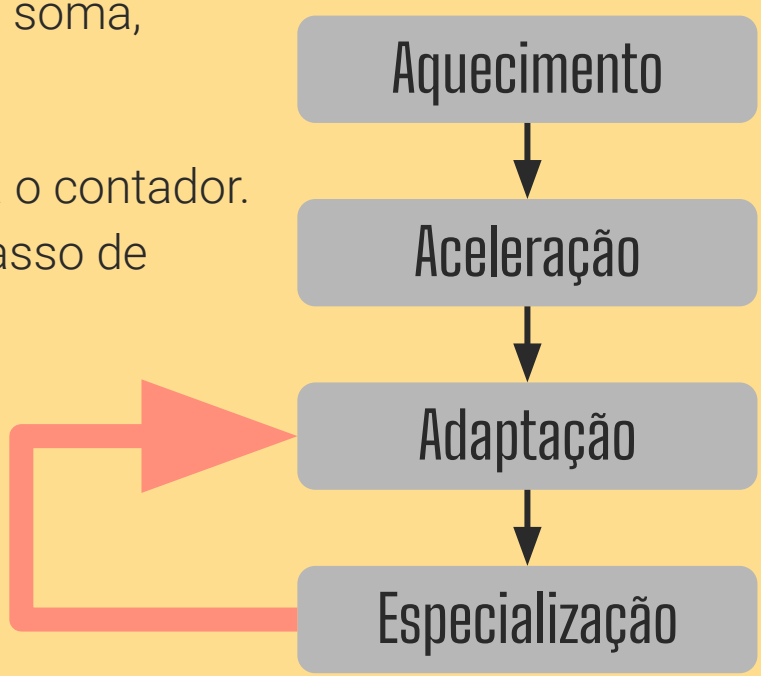


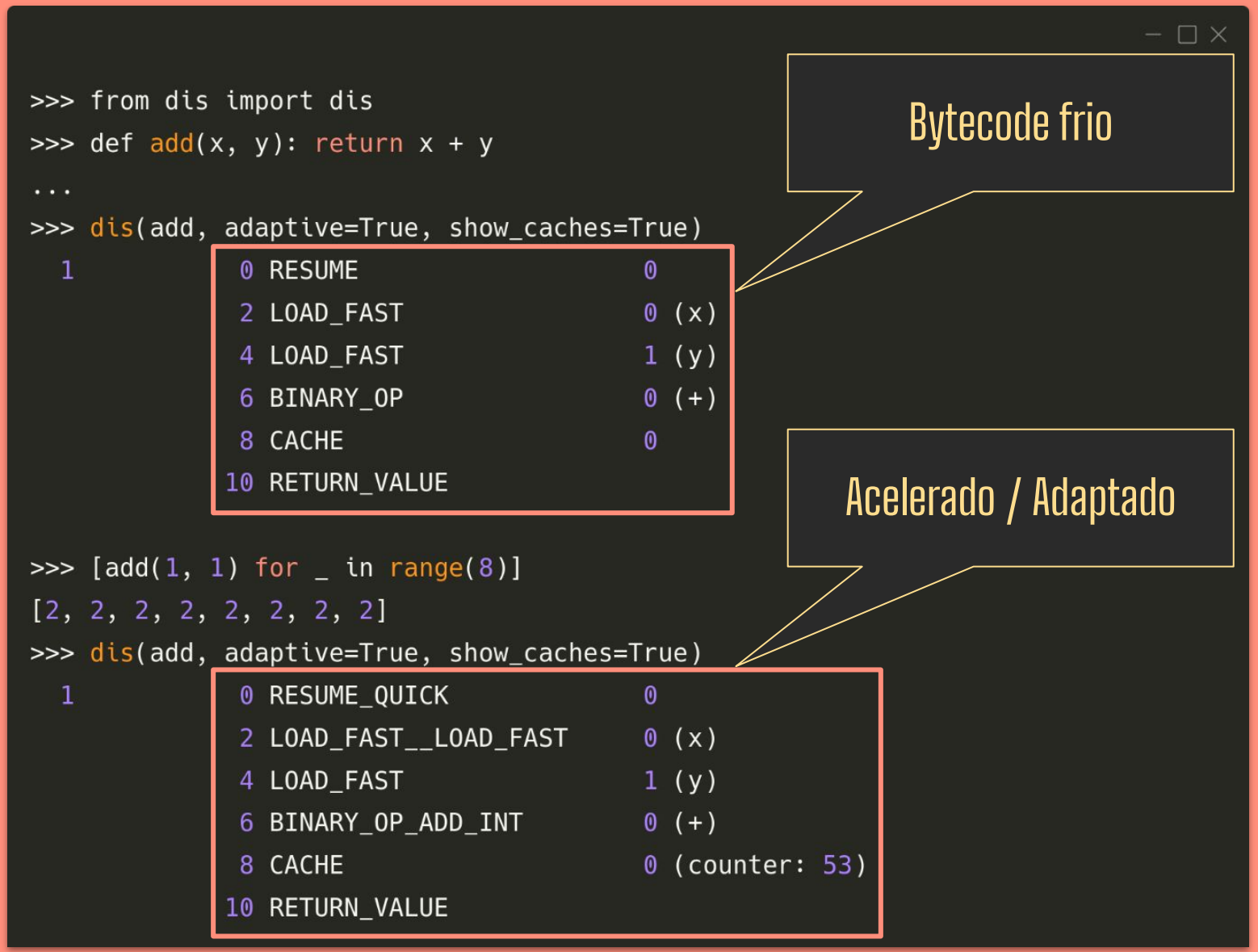
DesEspecialização

Pelo python de ser uma linguagem dinâmica, no meio do caminho as operações especializadas por não fazer mais sentido.

Ex: Depois de um tempo chamando int na soma, passamos a chamar só com string.

Cada "erro" de especialização decrementa o contador. Quando ele for igual a zero, ele volta ao passo de adaptação.




>>> from dis import dis
>>> def add(x, y): return x + y
...
>>> dis(add, adaptive=True, show_caches=True)
1 0 RESUME 0
2 LOAD_FAST 0 (x)
4 LOAD_FAST 1 (y)
6 BINARY_OP 0 (+)
8 CACHE 0
10 RETURN_VALUE
Bytecode frio
>>> [add(1, 1) for _ in range(8)]
[2, 2, 2, 2, 2, 2, 2, 2]
>>> dis(add, adaptive=True, show_caches=True)
1 0 RESUME_QUICK 0
2 LOAD_FAST__LOAD_FAST 0 (x)
4 LOAD_FAST 1 (y)
6 BINARY_OP_ADD_INT 0 (+)
8 CACHE 0 (counter: 53)
10 RETURN_VALUE
Acelerado / Adaptado

```
>>> from dis import dis
>>> def add(x, y): return x + y
...
>>> dis(add, adaptive=True, show_caches=True)
  1           0 RESUME                   0
              2
              4
              6
              8
             10

>>> [add(1, 1) f
[2, 2, 2, 2, 2,
>>> dis(add, ada
  1           0
              2
              4
              6
              8
             10
```

```
>>> [add('1', '1') for _ in range(53)]
['11', ..., '11']
>>> dis(add, adaptive=True, show_caches=True)
  1           0 RESUME_QUICK             0
              2 LOAD_FAST__LOAD_FAST     0 (x)
              4 LOAD_FAST                1 (y)
              6 BINARY_OP_ADAPTIVE       0 (+)
              8 CACHE                    0 (counter: 501)
             10 RETURN_VALUE
```

Voltando a adaptação

```
>>> [add('1', '1') for _ in range(50)]
['11', ..., '11']
>>> dis(add, adaptive=True, show_caches=True)
  1           0 RESUME_QUICK             0
              2 LOAD_FAST__LOAD_FAST     0 (x)
              4 LOAD_FAST                1 (y)
              6 BINARY_OP_ADD_UNICODE     0 (+)
              8 CACHE                    0 (counter: 53)
             10 RETURN_VALUE
```

Readaptado

Tá, olhando a soma foi simples...

Vamos olhar esse processo para outra instrução. Uma mais complexa:

```
def get_builtin():
    return str(42)
```

Tá, olhando a soma foi simples...

Vamos olhar esse processo para outra instrução. Uma mais complexa:

```
>>> dis(get_builtin)
1           0 RESUME                   0
            2 LOAD_GLOBAL              1 (NULL + str)
           14 LOAD_CONST               1 (42)
           16 PRECALL                  1
           20 CALL                     1
           30 RETURN_VALUE
```

Tá, olhando a soma foi simples...

Vamos olhar esse processo para outra instrução. Uma mais complexa:

```
>>> dis(get_builtin)
1           0 RESUME                   0
            2 LOAD_GLOBAL              1 (NULL + str)
           14 LOAD_CONST               1 (42)
           16 PRECALL                  1
           20 CALL
           30 RETURN_VALUE
```

Procura no global, se não achar no builtins

Otimização antes da chamada

(foi removida em versões posteriores- bpo-44525)

Tá, olhando a soma foi simples...

ma mais complexa:

```
>>> dis(get_builtin)
1           0 RESUME                   0

            2 LOAD_GLOBAL              1 (NULL + str)
           14 LOAD_CONST               1 (42)
           16 PRECALL                  1
           20 CALL
           30 RETURN_VALUE
```

Procura no global, se não achar no builtins

Otimização antes da chamada

(foi removida em versões posteriores- bpo-44525)

Aquecida e adaptada

```
>>> dis(get_builtin, adaptive=True, show_caches=True)
  1           0 RESUME_QUICK              0
              2 LOAD_GLOBAL_BUILTIN       1 (NULL + str)
              4 CACHE                     0 (counter: 53)
              6 CACHE                     0 (index: 75)
              8 CACHE                     0 (module_keys_version: 71)
             10 CACHE                     0
             12 CACHE                     0 (builtin_keys_version: 70)
             14 LOAD_CONST                1 (42)
             16 PRECALL_NO_KW_STR_1       1
             18 CACHE                     0 (counter: 53)
             20 CALL_ADAPTIVE             1
             22 CACHE                     0 (counter: 0)
             24 CACHE                     0 (func_version: 0)
             26 CACHE                     0
             28 CACHE                     0 (min_args: 0)
             30 RETURN_VALUE
```

Especializações e adaptações

```
const uint8_t _PyOpcode_Deopt[256] = {
    [ASYNC_GEN_WRAP] = ASYNC_GEN_WRAP,
    [BEFORE_ASYNC_WITH] = BEFORE_ASYNC_WITH,
    [BEFORE_WITH] = BEFORE_WITH,
    [BINARY_OP] = BINARY_OP,
    [BINARY_OP_ADAPTIVE] = BINARY_OP,
    [BINARY_OP_ADD_FLOAT] = BINARY_OP,
    [BINARY_OP_ADD_INT] = BINARY_OP,
    [BINARY_OP_ADD_UNICODE] = BINARY_OP,
    [BINARY_OP_INPLACE_ADD_UNICODE] = BINARY_OP,
    [BINARY_OP_MULTIPLY_FLOAT] = BINARY_OP,
    [BINARY_OP_MULTIPLY_INT] = BINARY_OP,
    [BINARY_OP_SUBTRACT_FLOAT] = BINARY_OP,
    [BINARY_OP_SUBTRACT_INT] = BINARY_OP,
    [BINARY_SUBSCR] = BINARY_SUBSCR,
    ...
```

https://github.com/python/cpython/blob/3.11/Include/internal/pycore_opcode.h#L55

Código dos bytecodes

```
TARGET(BINARY_OP) {
    PREDICTED(BINARY_OP);
    PyObject *rhs = POP();
    PyObject *lhs = TOP();
    assert(0 <= oparg);
    assert((unsigned)oparg < Py_ARRAY_LENGTH(binary_ops));
    assert(binary_ops[oparg]);
    PyObject *res = binary_ops[oparg](lhs, rhs);
    Py_DECREF(lhs);
    Py_DECREF(rhs);
    SET_TOP(res);
    if (res == NULL) {
        goto error;
    }
    JUMPBY(INLINE_CACHE_ENTRIES_BINARY_OP);
    DISPATCH();
}
```

https://github.com/python/cpython/blob/3.11/Python/ceval.c#L1758

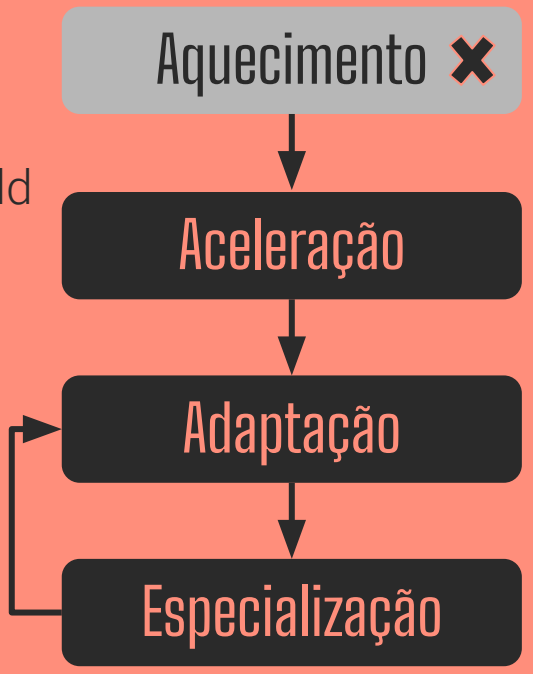
Sem aquecimento

3.12

Novidades da versão 3.12

Na versão 3.12 algumas mudanças foram feitas nesse sistema:

1. O **aquecimento é feito na inicialização**
   a. **O bytecode é sempre acelerado**
   b. **Superinstruções + inline cache**
2. Alterações estruturais no fonte do cpython
   a. Os cases agora estão em Python/bytecodes.c
   b. Agora as opcodes podem ser mudadas no build
3. Novos casos de adaptação/especialização



O bytecode compilado

```
python -m dis exemplo.py
# ...
  1           0 RESUME                   0

  2           2 LOAD_FAST                0 (x)
              4 LOAD_FAST                1 (y)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
```

O bytecode aquecido por padrão

```
>>> from dis import dis
>>> def add(x, y): return x + y
...
>>> dis(add, adaptive=True, show_caches=True)
  1           0 RESUME                   0

              2 LOAD_FAST__LOAD_FAST     0 (x)
              4 LOAD_FAST                1 (y)
              6 BINARY_OP                0 (+)
              8 CACHE                    0 (counter: 17)
             10 RETURN_VALUE
```

O início de um sonho

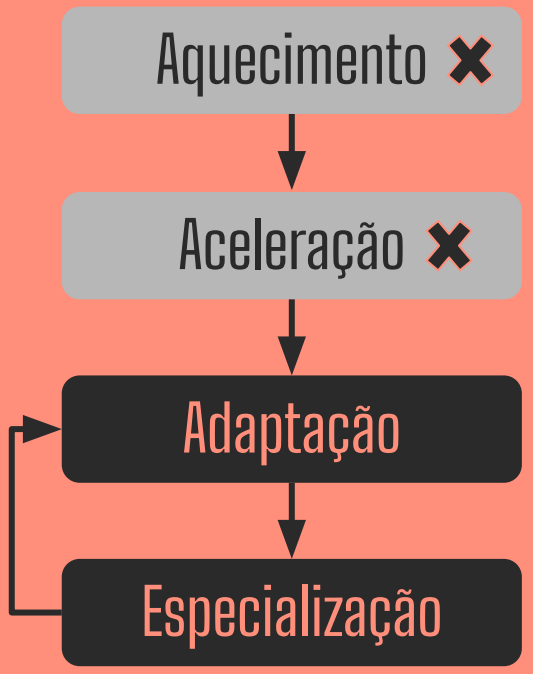
3.13

Novidades da 3.13

Na versão 3.13 do python **não existe mais o processo de adaptação**, o **bytecode é compilado em sua versão acelerada**. Também existe extensão das possibilidades de instruções.

Tópicos quentes dessa versão:

- **Tiers** de otimizações (ao nível de compilação)
  - **LOOPS**
  - Rastreamento
  - Micro-ops
  - Superblocks (super-micro-op)
- **Executors**
  - Subinterpreters
- **JIT**



Compilação + superinstruções

```
python -m dis exemplo.py
# ...
  1           RESUME                   0
  2           LOAD_FAST_LOAD_FAST      1 (x, y)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

O sistema de tiers

Uma nomenclatura adotara nas otimizações na versão 3.13 são os tiers. Eles querem dizer **níveis de otimização**:

- Tier 1: Interpretador adaptativo especializado
- Tier 2: Micro operações otimizadas
  - o JIT é opcional nessa etapa!

Tier 2

O tier dois atualmente só funciona em loops. Qual a ideia?

- Após a 17a (JUMP_BACKWARD) a execução do loop ele coloca um trace!
  - Sim, **atualmente** somente loops!
  - Quando o trace é adicionado, as operações serão analisadas
  - Substituídas por micro operações

```
def fib(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a +b
    return a
```

```
L1:     FOR_ITER_RANGE          8 (to L2)
        STORE_FAST              3 (_)

        LOAD_FAST_LOAD_FAST    33 (b, a)
        LOAD_FAST               2 (b)
        BINARY_OP_ADD_INT       0 (+)
        STORE_FAST_STORE_FAST  33 (b, a)
        JUMP_BACKWARD          10 (to L1)
```

PYTHON_LLTRACE=2 ./python fib.py

Mostrar o código, pq não cabe no slide :)

Minha compilação do python 3.13:

./configure --enable-experimental-jit=interpreter --with-pydebug

Como ver as microops em debug?

PYTHON_LLTRACE=2 ./python <arquivo>.py

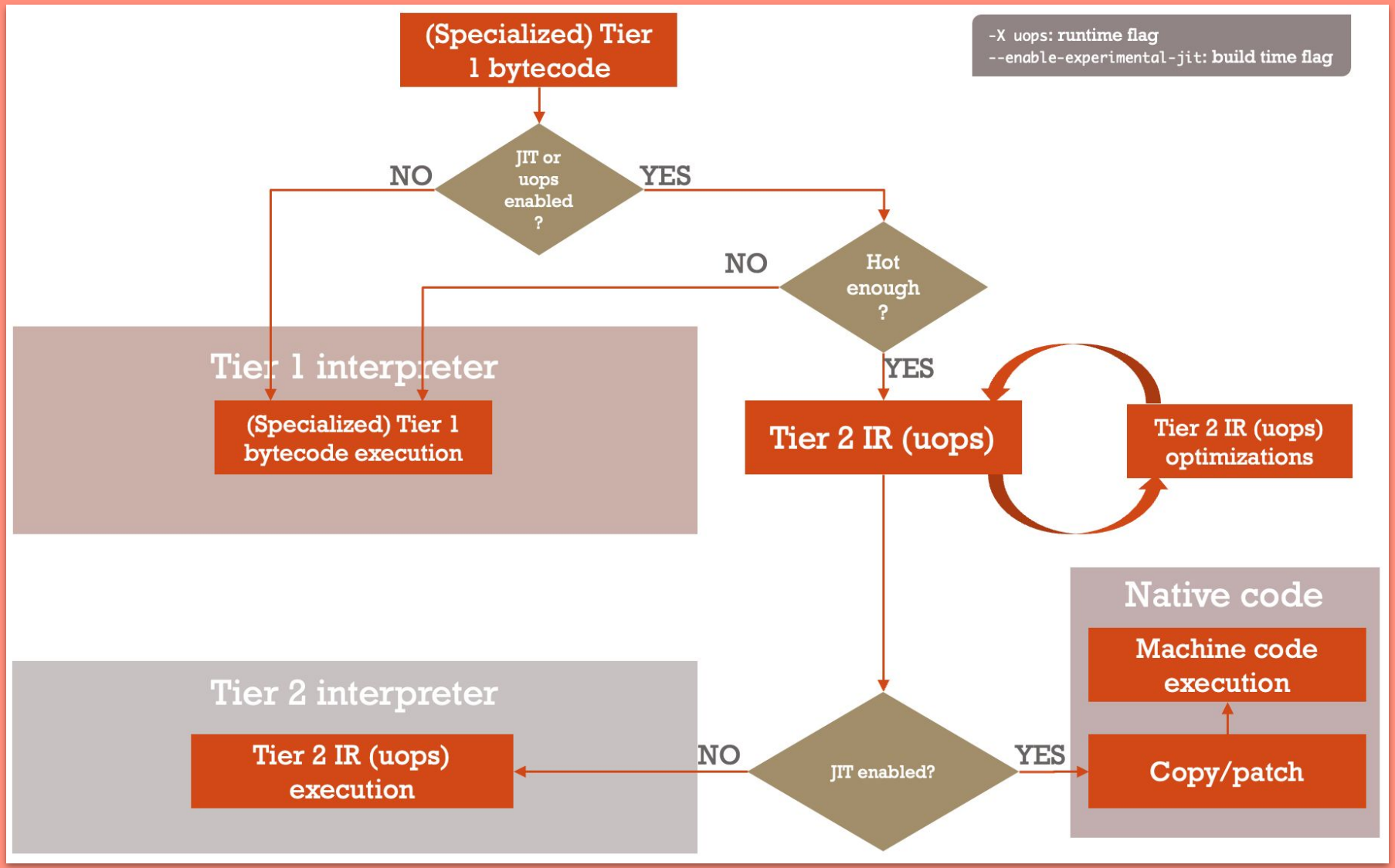
PEP 744: Compilação JIT

PPE

Então, conversei sobre isso com o restante da equipe do Faster CPython e achamos que o esquema a seguir faria sentido para o 3.13 (nomes sujeitos a bikeshedding):

- `--enable-experimental-jit=no` (o padrão): Não crie o JIT ou o interpretador micro-op. O novo `PYTHON_JIT` variável de ambiente não tem efeito.
- `--enable-experimental-jit=interpreter`: não crie o JIT, mas *crie* e habilite o interpretador micro-op. Isso é útil para aqueles de nós que estão desenvolvendo ou depurando micro-ops (mas não querem lidar com o JIT) e é equivalente a usar o `-X uops` ou `PYTHON_UOPS=1` opções hoje. `PYTHON_JIT=0` pode ser usado para desabilitar o interpretador micro-op em tempo de execução.
- `--enable-experimental-jit=yes-off`: crie o JIT, mas não o habilite por padrão. `PYTHON_JIT=1` pode ser usado para habilitá-lo em tempo de execução.
- `--enable-experimental-jit=yes` (ou apenas `--enable-experimental-jit`): Crie o JIT e ative-o por padrão. `PYTHON_JIT=0` pode ser usado para desativá-lo em tempo de execução.

https://discuss.python.org/t/pep-744-jit-compilation/50756/42



https://discuss.python.org/t/pep-744-jit-compilation/50756/28

Links

Documentação:

- **OPCODES**: https://docs.python.org/pt-br/3/library/dis.html#python-bytecode-instructions
- **What's new 3.11**: https://docs.python.org/3/whatsnew/3.11.html#pep-659-specializing-adaptive-interpreter
- **What's new 3.13**: https://docs.python.org/3.13/whatsnew/3.13.html#an-experimental-just-in-time-jit-compiler

PEPs:

- **PEP-659**: https://peps.python.org/pep-0659/
- **PEP-744** https://peps.python.org/pep-0744/

Papers:

- **Inline caching meets quickening**: https://doi.org/10.5555/1883978.1884008
- **Optimizations for a Java Interpreter Using Instruction Set Enhancement**: https://publications.scss.tcd.ie/tech-reports/reports.05/TCD-CS-2005-61.pdf
- **Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters**: https://doi.org/10.1145/1286821.1286828

Código-fonte do Cpython:

- **3.11**: https://github.com/python/cpython/tree/3.11
- **3.12**: https://github.com/python/cpython/tree/3.12
- **3.13**: https://github.com/python/cpython/tree/3.13

Lives citadas:

- **218**: https://youtu.be/pxfZTAJDipY
- **238**: https://youtu.be/nWmPEgTwGMM

Vídeos:

- **Brandt Bucher: Inside CPython 3.11's new specializing, adaptive interpreter**: https://youtu.be/shQtrn1v7sQ
- **Brandt Bucher – A JIT Compiler for CPython**: https://youtu.be/HxSHIpEQRjs
- **Ken Jin, Jules Poon: How two undergrads from the other side of the planet are speeding up…**: https://youtu.be/p57zI4qPVZY