2 Python 程序的解析和运行

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆ 了解 Python 代码的解析与编译过程；
- ⋆⋆ 能够阅读并理解抽象语法树与字节码；
- ⋆⋆ 理解 Python 虚拟机执行字节码的机制。

2.1 Python 语言

Python 是一种解释执行的编程语言，以简洁易读的语法著称，是人工智能、数据科学和快速原型开发的首选语言。它拥有丰富的标准库和第三方生态，如 NumPy、Pandas、Matplotlib、Scikit-learn 和 PyTorch，使得数据处理、可视化、机器学习和深度学习都能高效实现。作为当前 AI 编程中最主要的语言，同学们通过本章内容学习将了解 Python 程序的运行原理及其优缺点，从而能够对现有 AI 软件生态和发展趋势形成一定的理解和分析能力。

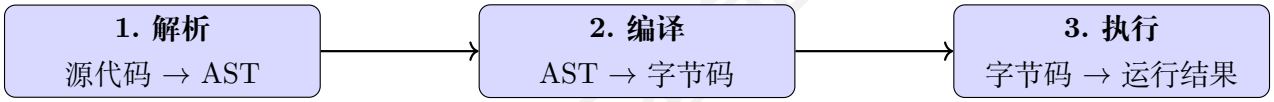

图 2.1: Python 程序运行过程

Python 程序的运行过程可以概括为三个主要阶段：

1) **语法分析 (Parsing)**：解释器首先读取源代码，将其转换为抽象语法树（Abstract Syntax Tree，AST）。AST 描述了程序的语法结构与层次关系，为后续编译阶段提供结构化表示。

2) **字节码生成（Compilation）**：抽象语法树被编译为字节码（Bytecode）。字节码是一种与具体硬件平台无关的中间表示形式，通常以 `.pyc` 文件形式缓存。

3) **虚拟机执行（Execution）**：Python 虚拟机（Python Virtual Machine, PVM）逐条解释执行字节码。执行过程中采用基于栈的执行模型，维护调用栈与运行时状态，从而完成表达式计算、变量管理以及函数调用等操作。

可以使用以下命令查看 Python 源代码在不同阶段的表示形式：

```
# 查看抽象语法树（AST）
python -m ast toy.py
# 查看词元（token）序列
python -m tokenize toy.py
# 查看字节码
python -m dis toy.py
# 将 Python 源代码编译为字节码（.pyc）
python -m compileall toy.py
```

代码 2.1: Python 代码分析与编译命令

2.2 语法解析

语法解析一般分为词法分析和句法分析两步。本节内容是传统编译原理课程的经典部分，如果同学们已经学习过编译，可以直接跳过。

2.2.1 词法分析

语法解析的第一步是词法解析（Lexical Analysis）。词法解析的目标是将源程序的字符流（Character Stream）划分为一系列具有语法意义的词元（Token），并过滤掉对语法结构无关的细节，例如空白符和注释。每个词元通常由其类别以及可选的属性（如标识符名称、常量值）组成。

由于 Python 是一种基于缩进的编程语言，代码块的结构由行首缩进直接决定，而非通过显式的分隔符（如花括号）表示。因此，Python 的词法分析器不仅负责将字符流转换为关键字、标识符、常量等基本记号，还需要根据行首缩进层级的变化显式地产生 `INDENT` 和 `DEDENT` 记号，以刻画代码块的开始与结束。

```
def relu(x):
    return max(0.0, x)

print(relu(100))
```

代码 2.2: python 代码示例

以代码 2.2 中的程序为例，其词法分析后的词元序列（抽象表示）如下所示：

```
ENCODING(utf-8)
NAME(def) NAME(relu) ( NAME(x) ) : NEWLINE
INDENT
NAME(return) NAME(max) ( NUMBER(0.0), NAME(x) ) NEWLINE
NL
DEDENT
NAME(print) ( NAME(relu) ( NUMBER(100) ) ) NEWLINE
ENDMARKER
```

代码 2.3: Code 2.2 的词法分析结果（Token 序列）

词法分析通常基于正则表达式实现，用于描述和识别各类词元的词法模式。每一类词元都可以通过一个正则表达式表示，词法分析器通过匹配这些模式，将连续的字符归约为一个个词元。表 2.1列举了正则表达式中单个字符的表示方法。

表 2.1: 单个字符元素正则表示方法

字符元素	表述形式	含义
特定字符	a	$x = a$
字符范围	$[ab]$	$x \in \{a, b\}$
字符范围	$[a-z]$	$x \in \{a, ..., z\}$
字符范围	$[a-zA-Z]$	$x \in \{a, ..., z, A, ..., Z\}$
通配符	.	$x \in \Sigma$
排除特定字符	$\hat{}a$	$x \in \Sigma \backslash \{a\}$
空字符	ϵ	$x \in \emptyset$
特定字符或空	$a?$	$x = a$ or $x = \epsilon$

在此基础上，单个字符可以通过选择、连接和闭包等方式组合形成复杂模式。表 2.2 采用递归构造的方法定义了正则表达式的组合规则。

表 2.2: 正则表达式构造方法，其中 S 和 T 为子正则表达式或单个字符

构造方式	符号	优先级	示例	含义
选择	|	1	$S\|T$	$x \in \{S \cup T\}$
连接		2	ST	$x \in \{st \mid s \in S, t \in T\}$
闭包	$*$	3	S^*	$x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, 0 \le n < \infty\}$
正闭包（扩展）	$+$	3	S^+	$x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, 1 \le n < \infty\}$
区间闭包（扩展）		3	$S^{\{min,max\}}$	$x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, min \le n \le max\}$

例如，在 Python 的词法规则中，标识符（NAME）和数值常量（NUMBER）可以分别用如下正则表达式近似描述：

$$\texttt{NAME} \rightarrow [A\text{-}Za\text{-}z\_][A\text{-}Za\text{-}z\_0\text{-}9]^*$$

$$\texttt{NUMBER} \rightarrow [0\text{-}9]^+(.[0\text{-}9]^+)?$$

其中，NAME 表示以字母或下划线开头、后续可以跟字母、数字或下划线的标识符，用于表示变量名和函数名；NUMBER 表示整数或浮点数形式的数值常量。在代码 2.2 的示例程序中，relu、x 和 max 均被识别为 NAME，而 0.0 和 100 被识别为 NUMBER。

需要注意的是，在 Python 的词法分析阶段，关键字（如 def 和 return）在词法分析输出中仍然被标记为 NAME 类型，其关键字属性是在后续语法分析阶段根据具体字符串值进行识别的。因此，在上述词元序列中，def 和 return 仍然以 NAME(def) 和 NAME(return) 的形式出现。在许多传统编程语言中，这些关键字通常会在词法分析阶段直接识别。

除上述两类词元外，词法分析器还会识别定界符和分隔符，例如括号、逗号和冒号等。这些符号通常通过对固定字符的直接匹配生成，其词法规则可以表示为：

$$\texttt{LPAREN} \rightarrow \texttt{(}$$

$$\texttt{RPAREN} \rightarrow \texttt{)}$$

$$\texttt{COMMA} \rightarrow \texttt{,}$$

$$\texttt{COLON} \rightarrow \texttt{:}$$

换行符 NEWLINE 用于表示逻辑行结束，其对应的字符通常为操作系统相关的行结束符，例如：

$$\texttt{NEWLINE} \rightarrow \texttt{\textbackslash n} \mid \texttt{\textbackslash r\textbackslash n}$$

其中，\n 是类 Unix 系统（如 Linux 和 macOS）常用的行结束符，而 \r\n 是 Windows 系统使用的行结束符。Python 在读取源文件时会统一处理这些不同形式的换行符。与之相关的还有 NL 词元，它同样表示换行符，但仅用于标记非逻辑行结束的换行，例如空行或括号内部的换行。在这些情况下，换行符不会结束当前语句，因此词法分析器生成 NL 而不是 NEWLINE。

值得注意的是，NL 的表示和识别超出了正则表达式的能力，因为它依赖于当前换行所在的上下文，例如是否位于括号、方括号或花括号内部，或者该行是否为空行。与之类似，INDENT 和 DEDENT 也无法直接由正则表达式描述，它们通过比较相邻逻辑行的缩进宽度生成，用于显式标记代码块的开始和结束。通过这些上下文敏感的词元，Python 的词法分析器能够正确处理缩进和空行，从而为后续的语法分析提供准确的输入。

此外，Python 源代码中还可以包含注释（COMMENT）。注释以字符 # 开始，一直延续到当前行结束，其词法规则可以近似表示为：

$$\texttt{COMMENT} \rightarrow \texttt{\#}[\hat{}\,\backslash n]^*$$

在词法分析的输出中，注释会被识别为独立的 COMMENT 词元，并且通常紧跟一个 NEWLINE 或 NL 词元，用于表示该注释所在行的结束。

与一些编程语言不同，Python 并没有专门的多行注释语法。虽然三引号字符串（'''...'''）有时被程序员用作“多行注释”，但在词法和语法层面上它们仍然被解析为字符串（STRING），而不是注释。字符串（STRING）用于表示文本常量，可以由单引号或双引号包围，也可以使用三引号表示多行字符串。有兴趣的同学可是尝试自己定义 Python 字符串的正则表达式。

通过这些正则表达式和规则，词法分析器可以将源代码中的字符序列归约为代码 2.3 所示的词元序列，为后续的句法分析提供输入。由于正则表达式只能刻画局部的、线性的结构，词法分析阶段并不负责处理嵌套关系或程序的层次结构，这些结构性信息将在句法分析阶段进一步处理。

2.2.2 句法分析

句法分析的目标是在词法分析产生的词元序列基础上，判断程序是否符合语言的语法规则，并据此构造程序的结构化表示，通常是一棵抽象语法树。语法解析阶段并不关心程序的“含义是否正确”（例如变量是否已定义、类型是否匹配），而只关注程序在结构上是否合法。大多数编程语言的语法可以形式化地描述为上下文无关文法（Context-Free Grammar, CFG）。一个上下文无关文法通常表示为一个四元组 $G = (V, \Sigma, P, S)$，其中：

- V 是非终结符集合；
- Σ 是终结符集合，对应词法分析阶段产生的词元；
- P 是产生式集合；
- $S \in V$ 是开始符号。

产生式通常写作 $S \to \beta$，其中左侧 S 是一个非终结符，右侧 β 是由终结符和非终结符组成的符号序列。通过一系列产生式的推导，语法解析器可以判断一个程序是否可以从开始符号 S 推导得到；若可以，则该程序在语法上是合法的。

以 Python 的一个简化子集为例，其核心语法规则可以形式化描述如下：

$$
\begin{aligned}
\mathit{Module} &\to \mathit{StmtList} \\
\mathit{StmtList} &\to \mathit{Stmt}\ \mathit{StmtList} \mid \epsilon \\
\mathit{Stmt} &\to \mathit{SimpleStmt} \mid \mathit{CompoundStmt} \\
\mathit{CompoundStmt} &\to \mathit{FunctionDef} \\
\mathit{FunctionDef} &\to \texttt{def}\ \mathit{Identifier}\ \texttt{(}\ \mathit{ParamList}\ \texttt{):}\ \texttt{NEWLINE}\ \mathit{Block} \\
\mathit{Block} &\to \texttt{INDENT}\ \mathit{StmtList}\ \texttt{DEDENT} \\
\mathit{SimpleStmt} &\to \mathit{ReturnStmt}\ \texttt{NEWLINE} \mid \mathit{ExprStmt}\ \texttt{NEWLINE} \\
\mathit{ReturnStmt} &\to \texttt{return}\ \mathit{Expr} \\
\mathit{ExprStmt} &\to \mathit{Expr}
\end{aligned}
$$

语法解析的输出通常是一棵抽象语法树。抽象语法树会刻意丢弃与语义无关的语法细节（如括号、分号、换行），只保留程序的层次结构和关键构造。以代码 2.2 为例，经语法解析后，该程序可被表示为代码 2.2 所示的抽象语法树。

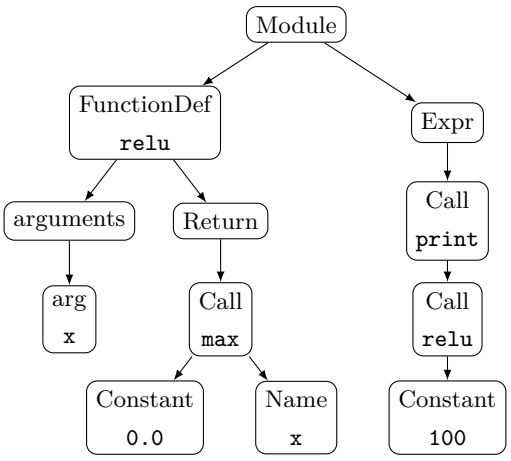

图 2.2: 代码 2.2 对应的抽象语法树

2.3 编译

在完成语法解析并构造抽象语法树之后，Python 并不会直接对抽象语法树进行解释执行，而是进入编译阶段。该阶段的主要任务是将抽象语法树转换为一种更适合虚拟机执行的中间表示。由于该中间表示由一组线性排列的指令组成，并以字节为基本单位对指令及其操作数进行编码，因此该表示通常被称为字节码。

从抽象层次上看，Python 的编译过程可以概括为：

$$\text{抽象语法树} \longrightarrow \text{代码对象} \longrightarrow \text{字节码}$$

代码对象（Code Object）是对一段可执行代码的封装，内部包含字节码指令序列以及与之配套的常量表、符号表、局部变量信息等元数据。以代码 2.2为例，在编译阶段共产生两个代码对象。第一个是模块级代码对象，用于描述源文件中所有顶层语句的执行过程，包括函数对象的创建以及后续的函数调用。第二个是函数 `relu` 对应的代码对象，该代码对象仅包含函数体内部的字节码指令，并作为常量存储在模块级代码对象中。

2.3.1 字节码指令

字节码指令是 Python 程序执行的最小操作单元,每条指令由操作码(Opcode)和可选操作数(Operand)组成，用于描述程序在虚拟机上的具体执行步骤。编译器会遍历 AST，根据不同类型的语法节点生成对应的字节码指令。这一过程通常是结构化的：例如，函数定义节点会生成函数对象相关的指令，而表达式节点则会生成计算与调用相关的指令。

以代码 2.2为例，其对应的字节码如代码 2.4 所示。

```
# -------------代码对象1: 模块----------------------------------------
# -------------源代码第1行: 函数定义 relu----------------------------
        0 LOAD_CONST               0 (<code object relu ...)
        2 LOAD_CONST               1 ('relu')
        4 MAKE_FUNCTION            0
        6 STORE_NAME               0 (relu)
# -------------源代码第4行: 调用 print(relu(100))--------------------
        8 LOAD_NAME                1 (print)
       10 LOAD_NAME                0 (relu)
       12 LOAD_CONST               2 (100)
       14 CALL_FUNCTION            1
       16 CALL_FUNCTION            1
       18 POP_TOP
       20 LOAD_CONST               3 (None)
       22 RETURN_VALUE
# -------------代码对象2: 函数 relu 函数体---------------------------
# -------------源代码第2行: 函数 relu 函数实现-----------------------
        0 LOAD_GLOBAL              0 (max)
        2 LOAD_CONST               1 (0.0)
        4 LOAD_FAST                0 (x)
        6 CALL_FUNCTION            2
        8 RETURN_VALUE
```

代码 2.4: 代码 2.2对应的字节码

上述字节码示例中的主要指令含义如下：

- `LOAD_CONST`：将指定的常量从常量表加载到操作数栈上。参数表示常量表中的索引，例如数值、字符串或函数对象。
- `LOAD_NAME` / `LOAD_GLOBAL`：将符号对应的对象从符号表加载到栈上。参数表示符号表中的索引。
- `STORE_NAME`：将栈顶对象绑定到指定名字，并存入符号表。参数表示符号表中的索引。
- `MAKE_FUNCTION`：根据栈顶的代码对象及默认参数信息创建函数对象，并压入栈。参数表示函数的标志或可选参数数量。
- `CALL_FUNCTION`：从栈中弹出函数和指定数量的参数执行调用，并将调用结果压入栈。参数表示函数调用时的参数数量。
- `POP_TOP`：弹出栈顶元素，通常用于丢弃表达式的返回值。此指令没有参数。
- `RETURN_VALUE`：从当前栈帧返回栈顶对象作为返回值。此指令没有参数。
- `LOAD_FAST`：将指定局部变量或参数从局部变量表加载到栈上。参数表示局部变量表中的索引。

2.3.2 代码对象中的表

每个代码对象除了存储字节码指令外，还维护几张重要的表，字节码指令在执行过程中需要访问这些表来获取操作数或目标对象：

- **常量表**：存储字面量（如数值、字符串）和内部代码对象。指令 `LOAD_CONST n` 使用常量表索引 n 获取对应的常量。
- **符号表**：存储模块或函数中使用的名字（函数名、全局变量名等）。指令 `LOAD_NAME n` 或 `STORE_NAME n` 使用符号表索引 n。函数内部也可以通过 `LOAD_GLOBAL n` 访问模块级符号表中的对象。
- **局部变量表**：存储函数的局部变量名，包括参数和局部变量。指令 `LOAD_FAST n` / `STORE_FAST n` 使用局部变量表索引 n。

以代码对象 1 为例，其维护的表内容如下：

常量表

Index	Value
0	<code object relu>
1	'relu'
2	100
3	None

符号表

Index	Name
0	relu
1	print

局部变量表 (co_varnames)

Index	Variable

图 2.3: 代码对象 1（模块）中各表的内容

代码对象 2（函数 `relu`）维护的表内容如下：

常量表

Index	Value
0	0.0

符号表

Index	Name
0	max

局部变量表

Index	Variable
0	x

图 2.4: 代码对象 2（函数 `relu`）中各表的内容

2.4 程序执行

Python 字节码并不是由计算机或操作系统直接运行，而是通过一个称为 Python 虚拟机的软件层来完成。因此 Python 又被称为解释执行语言。

2.4.1 Python 虚拟机

虚拟机的核心作用是提供一个统一的执行环境，将高级 Python 字节码翻译为可以被计算机理解的指令。Python 虚拟机（PVM）采用栈式架构，程序的运行状态通过一个操作数栈保存。程序在执行时，会先将需要使用的操作数压入栈顶，字节码指令从栈顶弹出数据进行计算，结果再压回栈顶。

当 PVM 开始执行某个代码对象时，会创建一个执行帧（frame）。如果一个函数被多次调用，则该函数代码的对象可以对应多个帧。一个执行帧包含以下内容：指向代码对象的引用、指令计数器（program counter）、操作数栈等内容。

以代码对象 1 为例，其执行过程如下。图 2.5展示了执行帧操作数栈的变化情况。

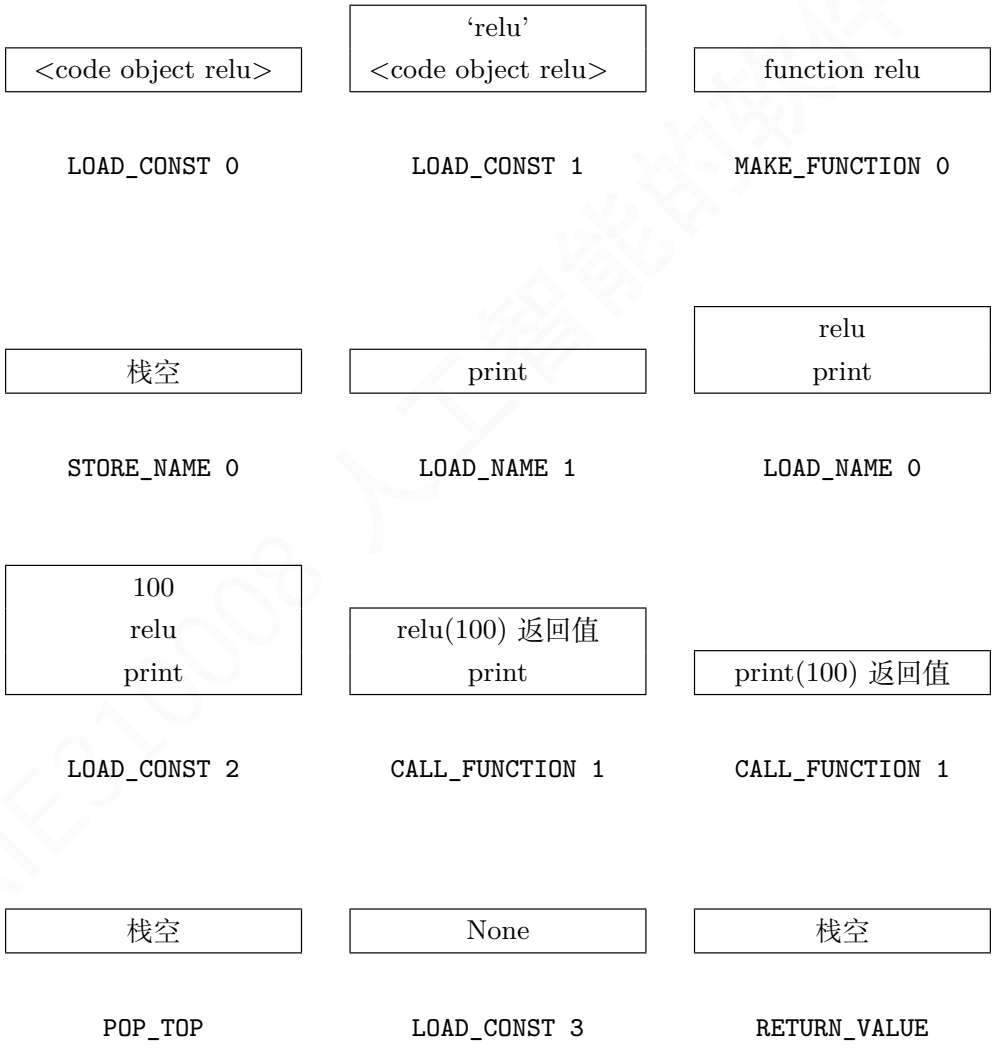

图 2.5: 代码对象 1（模块）执行过程中其操作数栈的变化示意图

1) `LOAD_CONST 0`：取常量表索引 0 → relu 的函数体，入栈。

2) `LOAD_CONST 1`：取常量表索引 1 → 函数名 'relu'，入栈。

3) MAKE_FUNCTION 0：生成函数对象（0 表示无默认参数），弹出栈顶代码对象和名字，将函数对象入栈。

4) STORE_NAME 0：符号表索引 0 → relu，将栈顶函数对象保存在全局符号表。

5) LOAD_NAME 1：符号表索引 1 → print，入栈。

6) LOAD_NAME 0：符号表索引 0 → relu，入栈。

7) LOAD_CONST 2：常量表索引 2 → 100，入栈作为函数参数。

8) CALL_FUNCTION 1：调用 relu(100)，参数数量 1，返回值压栈。

9) CALL_FUNCTION 1：调用 print(返回值)，参数数量 1。

10) POP_TOP：弹出 print 返回值 None。

11) LOAD_CONST 3：常量表索引 3 → None，作为模块返回值。

12) RETURN_VALUE：返回栈顶值，模块执行结束。

下面对代码对象 2（函数 relu）的执行流程进行分析。图 2.6 展示了执行帧操作数栈的变化情况。

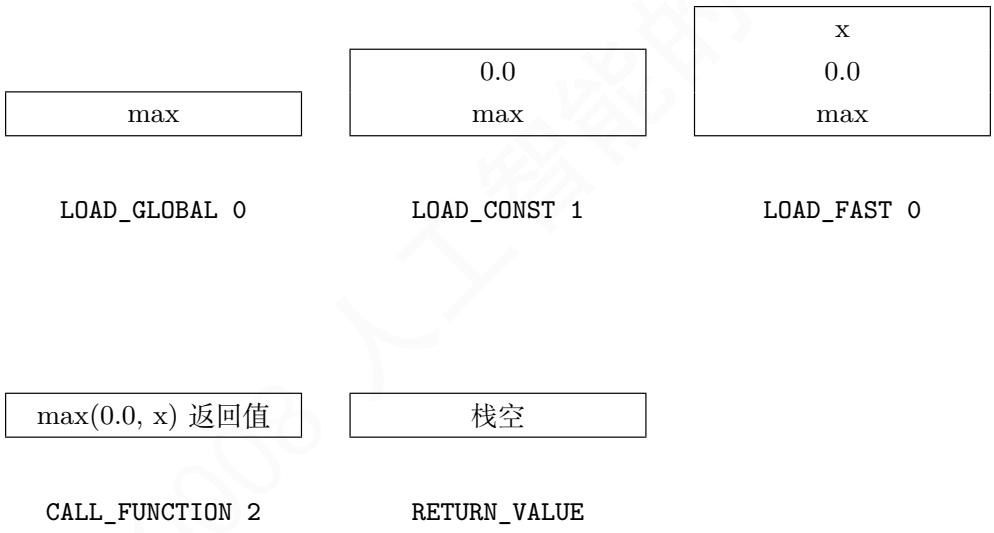

图 2.6: 代码对象 2（函数 relu）执行过程中其操作数栈的变化示意图

1) LOAD_GLOBAL 0：从函数符号表加载 max 函数到栈顶。

2) LOAD_CONST 1：将常量表索引 1 → 0.0 入栈。

3) LOAD_FAST 0：将局部变量表索引 0 → 参数 x（例如 100）入栈。

4) CALL_FUNCTION 2：调用 max(0.0, x)，参数数量 2，返回值压入栈顶。

5) RETURN_VALUE：返回栈顶值作为函数返回结果，函数执行结束。

练习

1) 定义一个正则表达式，用于匹配 Python 的单引号、双引号及三引号字符串字面量，并使用 Python 的 `re` 模块对若干示例字符串进行测试，观察匹配结果是否符合预期。

```python
import re

# 规则示例：pattern_name = r'[A-Za-z_][A-Za-z0-9_]*'
# TODO: 定义正则表达式匹配单/双/三引号字符串
pattern_string = r'填写你的正则'

# 示例字符串列表
examples = [
    's1 = "Hello, world!"',
    "s2 = 'single quotes'",
    '''
    s3 = multi
        line
        string
    ''',
]

for text in examples:
    matches = re.findall(pattern, text, flags=re.DOTALL)
    print(f"Text: {text!r}")
    print(f"Matches: {matches}")
    print("-" * 40)
```

代码 2.5: Python 字符串正则 PoC 框架

2) 以上节课的神经网络程序为例，

   a) 分析代码与其词元序列及抽象语法树的对应关系；

   b) 生成字节码，并解释其执行过程。

3) `python -m` 命令的功能本质上是通过调用相应模块实现的，如下代码所示。测试下列代码的效果。

```python
import sys
import ast
import dis
import tokenize
from io import BytesIO

def usage():
    print("Usage: python pycview.py <source.py> -ast|-ir|-tok")
    sys.exit(1)

def load_source(filename):
    with open(filename, "r", encoding="utf-8") as f:
        return f.read()

def dump_ast(source, filename):
    tree = ast.parse(source, filename=filename)
```

```
    print(ast.dump(tree, indent=2))
def dump_ir(source, filename):    code = compile(source, filename, "exec")
    dis.dis(code)

def dump_tokens(source):
    reader = BytesIO(source.encode("utf-8")).readline
    for tok in tokenize.tokenize(reader):
        print(tok)

def main():
    if len(sys.argv) != 3:
        usage()
    filename = sys.argv[1]
    option = sys.argv[2]
    if option not in ("-ast", "-ir", "-tok"):
        usage()
    source = load_source(filename)
    if option == "-ast":
        dump_ast(source, filename)
    elif option == "-ir":
        dump_ir(source, filename)
    elif option == "-tok":
        dump_tokens(source)

if __name__ == "__main__":
    main()
```

代码 2.6: python 代码解析模块示例