

程序设计基本元素



有了理论就够了吗？

① 我们有了编程的理论基石，所以我们就可以顺利地编出想要的程序了吗？

② 手搓图灵机？ 0000001100001001010010101010010100000000

③ 还是手搓 λ 公式？ $\lambda x . \lambda y . \lambda z . \lambda m . \lambda n . \lambda p . \lambda q . (\dots)(\dots)(\dots)(\dots)$

我们关心的 vs 实际的

$$x \div y$$

vs

$(q_5, 1, R, q_0)$

$(q_0, 1, R, q_0)$

$(q_0, 0, L, q_1)$

$(q_1, 1, b, q_2)$

$(q_2, 0, L, q_3)$

$(q_3, 1, R, q_4)$

$(q_4, 0, R, q_4)$

$(q_4, 1, b, q_5)$

$(q_5, 0, R, q_6)$

$(q_6, 1, L, q_1)$

$(q_1, 0, L, q_1)$

$(q_3, 0, R, q_7)$

$(q_7, 0, R, q_7)$

$(q_7, 1, b, q_8)$

$(q_8, 0, R, q_9)$

$(q_9, 1, b, q_8)$

$(q_9, 0, L, q_{11})$

$(q_6, 0, L, q_{11})$

我们关心的 vs 实际的

n

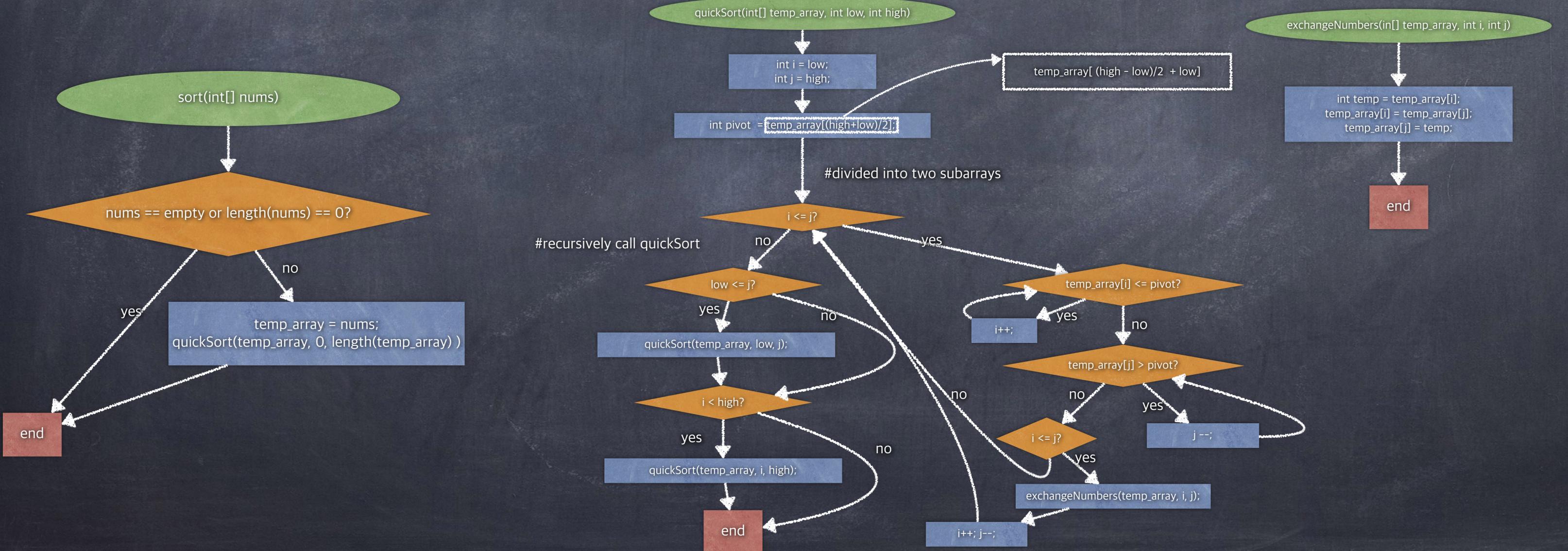
vs

$(\lambda n . \lambda f . \lambda x . n (\lambda g . \lambda h . h (g f)) (\lambda u . x) (\lambda u . u)) (\lambda f . \lambda x . f^n x)$

现实问题是复杂的

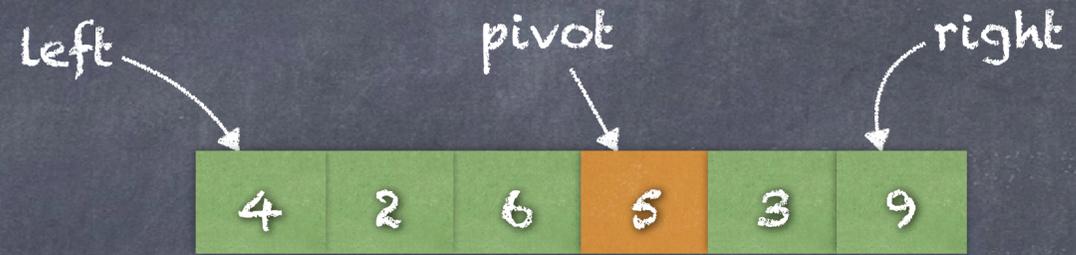
考虑如下场景

给定一个整数序列，我们让其按照大小进行排序

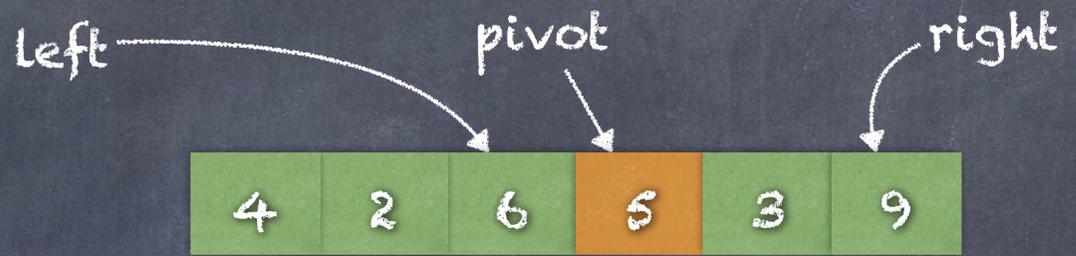


考虑如下场景

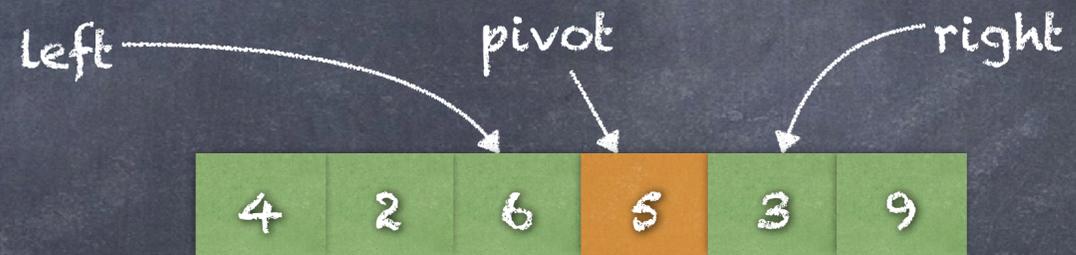
步骤1: 挑选一个 pivot



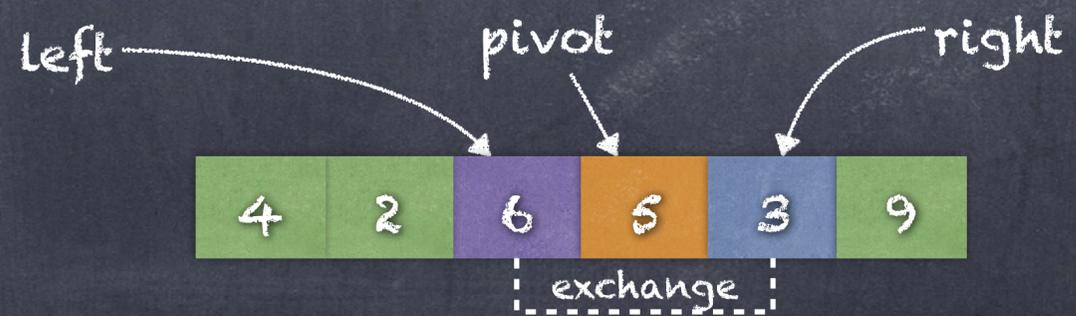
步骤2: 移动指针直到找到一个大于 pivot 的值



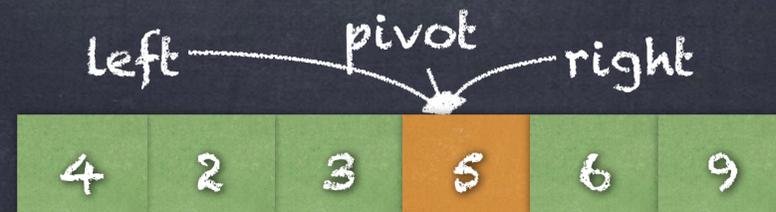
步骤3: 移动指针直到找到一个小于 pivot 的值



步骤4: 交换这两个指针的值



步骤5: 重复上述过程, 直至两个指针相遇或者交叉



注意力是有限的

- ① 如果把精力都用在处理图灵机这样繁琐的步骤，那么更高层面的复杂事物将无法构建。
- ② 此外，大量底层的繁琐步骤实际是重复的，我们应该避免这种重复。（Don't Repeat Yourself, 即DRY）

复杂带来的额外问题

- ① 易错性！

- ② 人是会犯错误的，代码写的越多，错误的可能性就越高。

- ③ 而如果我们有一些重复性的事情上犯了错误，那这个错误是“价值”不高的。

- ④ 因为如果我们可以避免重复，那我们完全可以避免此类错误！

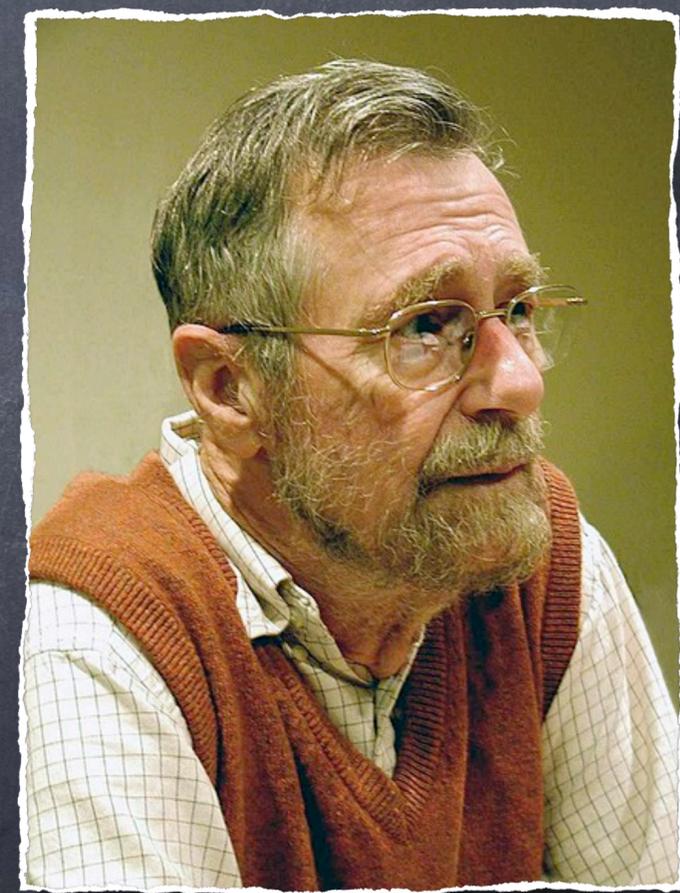
复杂带来的额外问题

注：在一些不良的编程范式下（比如乱用 GOTO 语句），正确率会更小

◎ 易错性！

◎ Dijkstra 在《Structured Programming》里说到，如果一个程序的一个子模块的正确性是 P ，那么整个程序的正确性是 P^n 。

◎ 因此，程序规模越大，越容易出错，而不停的重复只会加大错误率。



Edsger Wybe Dijkstra

可维护性问题

- ◎ 大量冗余的底层代码是难以维护的！
- ◎ 想象一下修改所有用到减法的地方，使得其支持负数。。。。。

如何构建复杂系统？

如何使得所构建的复杂的系统尽量正确？

如何优化已有代码？

我们缺乏了两方面

- ① 对数据的高效表示
- ② 对过程的合理抽象

让我们开始学习更好地编程

程序设计的基本元素

① 现代编程语言一般为我们提供下面三种机制：

① 基本表达式

① 组合的方式

① 抽象的方法

表达式 (Expression)

● 一个表达式就是为了表达计算并求值 (evaluate) 。

$$18 + 45$$

$$f(x)$$

$$\frac{6}{23}$$

$$\begin{pmatrix} 45 \\ 18 \end{pmatrix}$$

$$\sqrt{23234578}$$

$$2^{100}$$

$$\sin \pi$$

$$\lim_{x \rightarrow \infty} \frac{1}{x}$$

$$\sum_{i=1}^{100} i$$

$$7 \bmod 2$$

$$|-1253|$$

$$\log_2 1024$$

下面我们将以 Python 作为我们的首个教学语言

其使用者众多，具有成熟的应用生态，简洁，可读性强，并支持多种编程风格

Python 中的表达式

① 基本表达式

数字

2

字符串

“Hello!”

名字

add

② 运算表达式

1+2

15//3

③ 调用表达式

max(max(2,3), 5*min(-1, 4))

Python 中的表达式

- 所有的表达式都可以用函数调用的形式表达

一个表达式的解析



运算符和操作数本身也是表达式

值 (Values)

- 程序操纵着值

- 值代表着不同类型的数据

整数(Integers): 2 44 -3

字符串(Strings): "Hello!" "软件工程"

浮点数(Floats): 3.14 4.5 -2.0

布尔数(Booleans): True False

表达式求值

1. 求值 (Evaluate)

- 对运算符子表达式求值 (是什么运算)
- 对每个操作数子表达式求值 (具体值)

2. 作用 (Apply)

- 将运算符的值应用在操作数的值上 (即将函数作用在具体参数值上)



求值

```
add (add (6, mul(4, 6)), mul(3, 5))
```

人类求值

我们喜欢由里向外求值

```
add (add (6, mul (4, 6)), mul (3, 5))
```

```
add (add (6, 24), mul (3, 5))
```

```
add (add (6, 24), mul (3, 5))
```

```
add (30, mul (3, 5))
```

```
add (30, mul (3, 5))
```

```
add (30, 15)
```

```
add (30, 15)
```

嵌套表达式

1

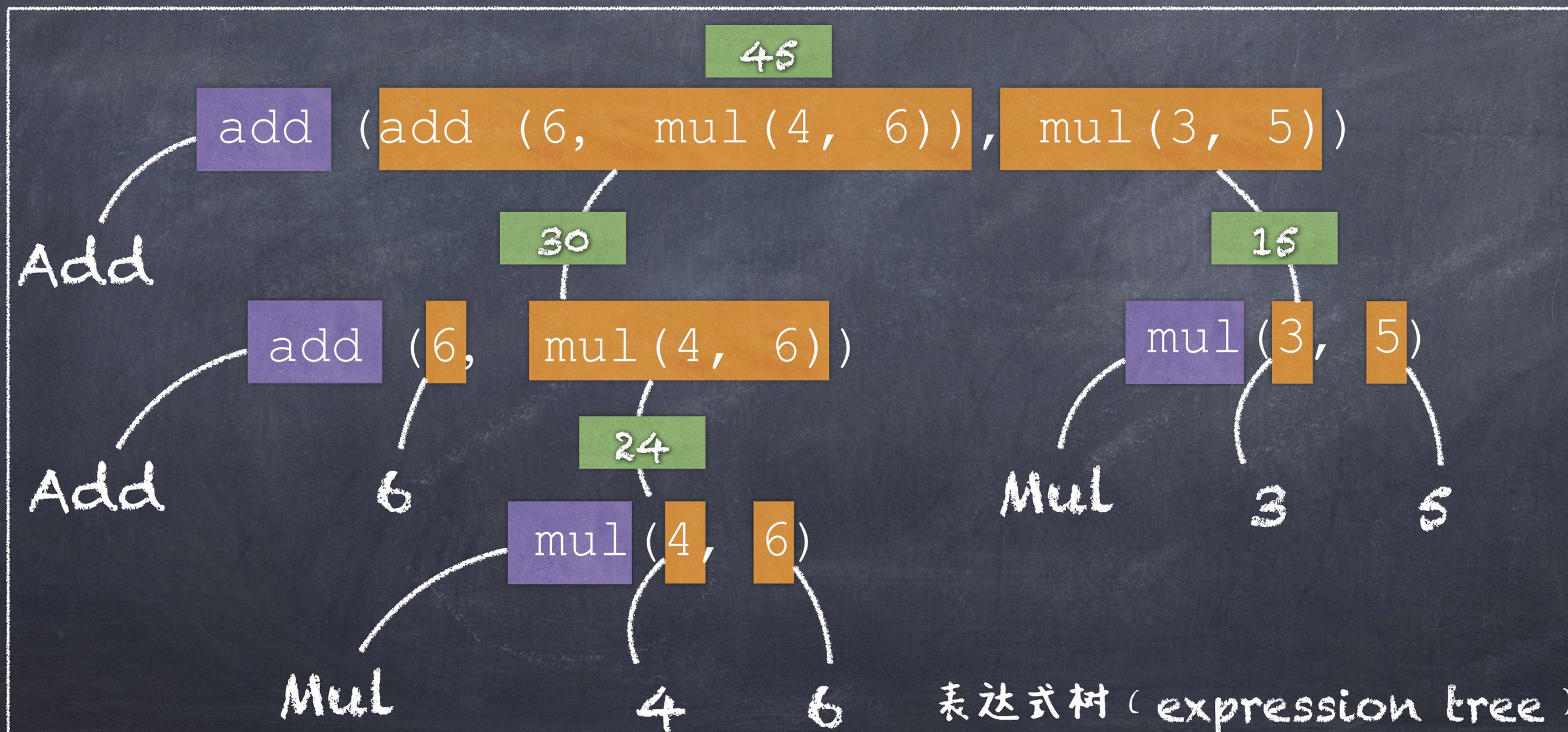
Evaluate operator

2

Evaluate operator

3

Apply



Functions, Values, Objects, Interpreters and Data

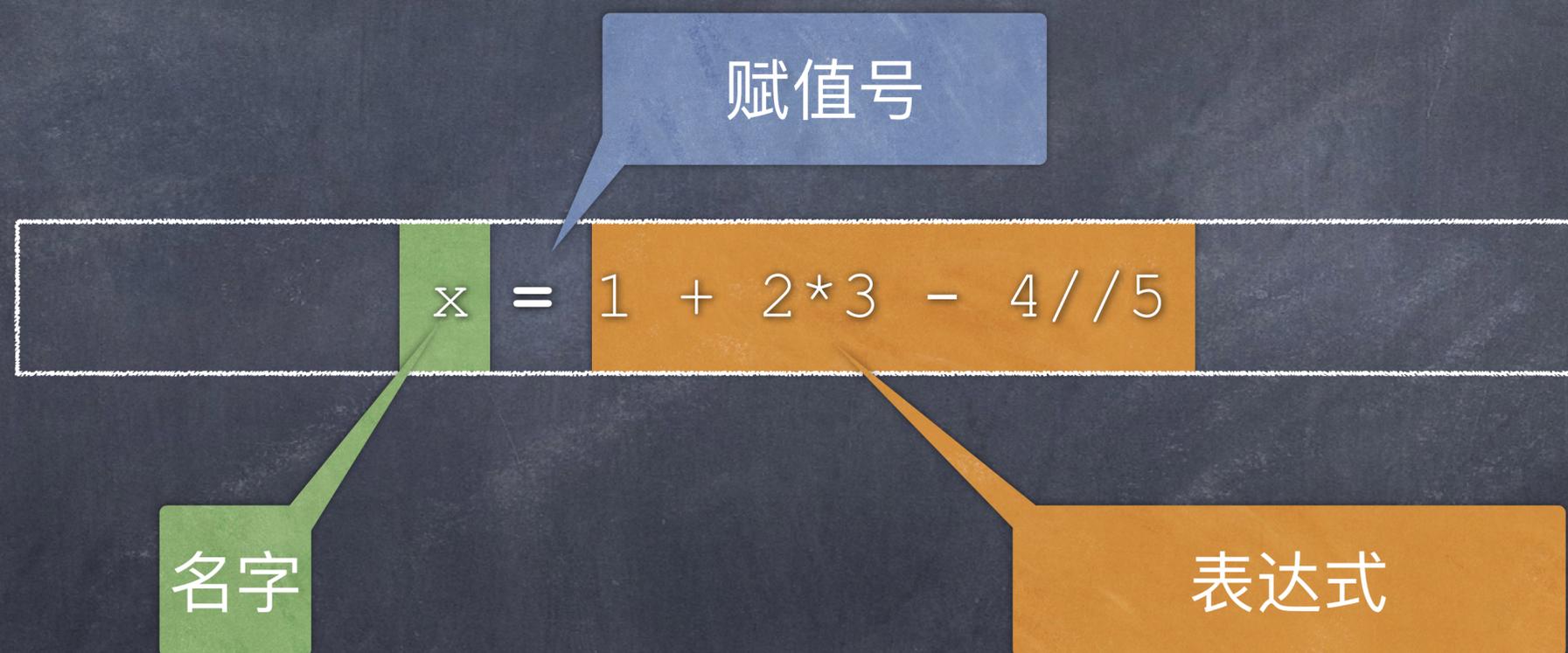
名字 (Names)

- 值可以赋予一个名字，使得我们可以更加容易的索引到他们
- 一个名字只能和一个值绑定！



赋值语句 (Assignment Statement)

在程序中，一种引入名字的方式就是利用赋值语句



注意：赋值并不求值，但是会影响程序

练习

```
>>> f = min
```

```
>>> f = max
```

```
>>> g, h = min, max
```

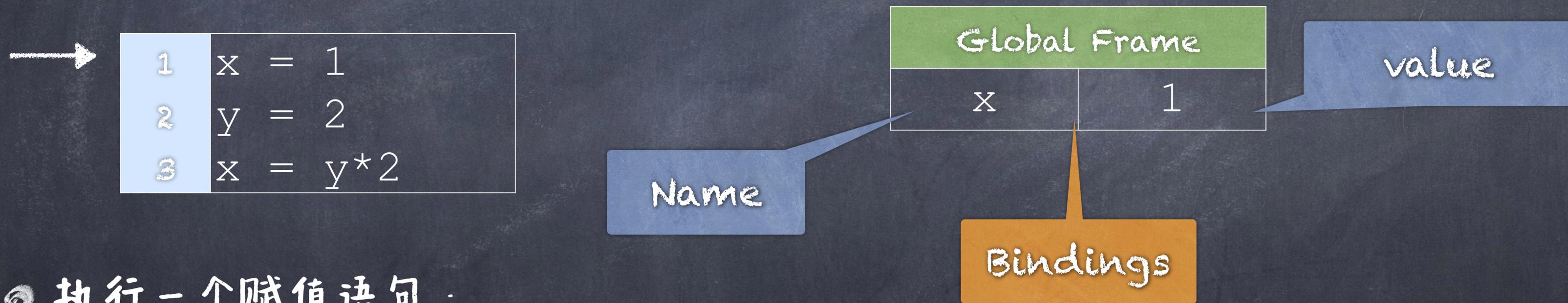
```
>>> max = g
```

```
>>> max(f(2, g(h(1, 5), 3)), 4)
```

```
???
```

赋值

名字是在一个环境 (Environment) 中被一个值所绑定

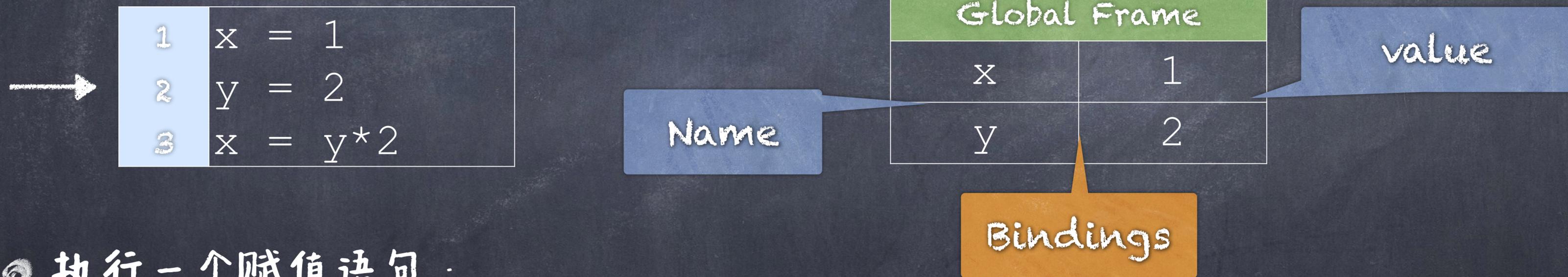


执行一个赋值语句:

1. 对“=”号后面的表达式求值
2. 在当前环境中将该值绑定“=”号左边的名字

赋值

名字是在一个环境 (Environment) 中被一个值所绑定



执行一个赋值语句：

1. 对“=”号后面的表达式求值
2. 在当前环境中将该值绑定“=”号左边的名字

赋值

名字是在一个环境 (Environment) 中被一个值所绑定

→

1	x = 1
2	y = 2
3	x = y * 2

Global Frame	
x	4
y	2

Name

Final value

Bindings

执行一个赋值语句：

1. 对“=”号后面的表达式求值
2. 在当前环境中将该值绑定“=”号左边的名字

回顾之前的例子

→

```
1 f = min
2 f = max;
3 g, h = min, max
4 max = g
5 max(f(2, g(h(1, 5), 3)), 4)
```

frames

Global Frame	
f	•

objects

func min(...) [parent=Global]



回顾之前的例子

```
1 f = min
2 f = max;
3 g, h = min, max
4 max = g
5 max(f(2, g(h(1, 5), 3)), 4)
```

frames



objects

func max(...) [parent=Global]



回顾之前的例子

→

```
1 f = min
2 f = max;
3 g, h = min, max
4 max = g
5 max(f(2, g(h(1, 5), 3)), 4)
```

frames

Global Frame	
f	●
g	●
h	●

objects

func max(...) [parent=Global]

func min(...) [parent=Global]



回顾之前的例子

```
1 f = min
2 f = max;
3 g, h = min, max
4 max = g
5 max(f(2, g(h(1, 5), 3)), 4)
```

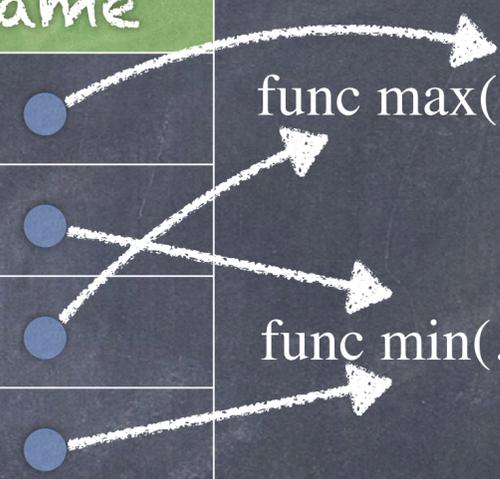
frames

Global Frame	
f	●
g	●
h	●
max	●

objects

func max(...) [parent=Global]

func min(...) [parent=Global]



回顾之前的例子

```
1 f = min
2 f = max;
3 g, h = min, max
4 max = g
5 max(f(2, g(h(1, 5), 3)), 4)
```

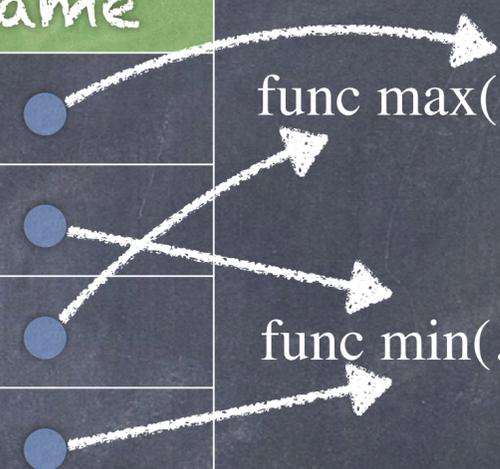
frames

Global Frame	
f	●
g	●
h	●
max	●

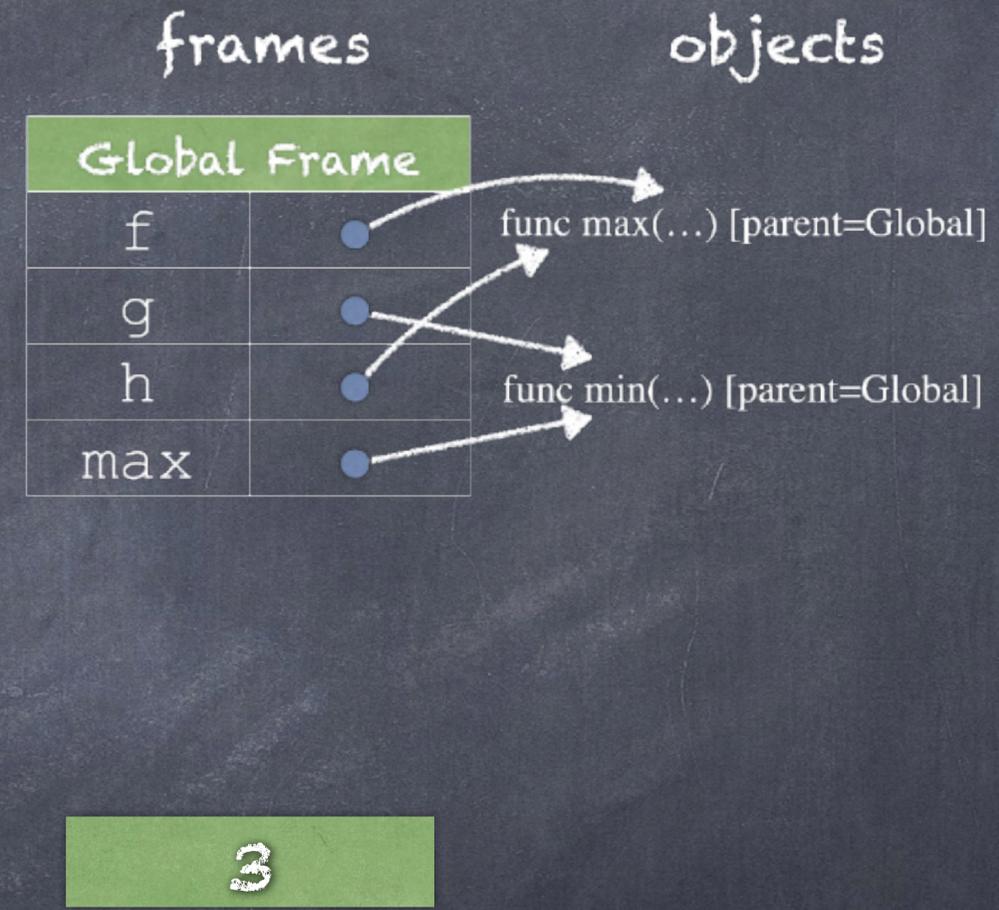
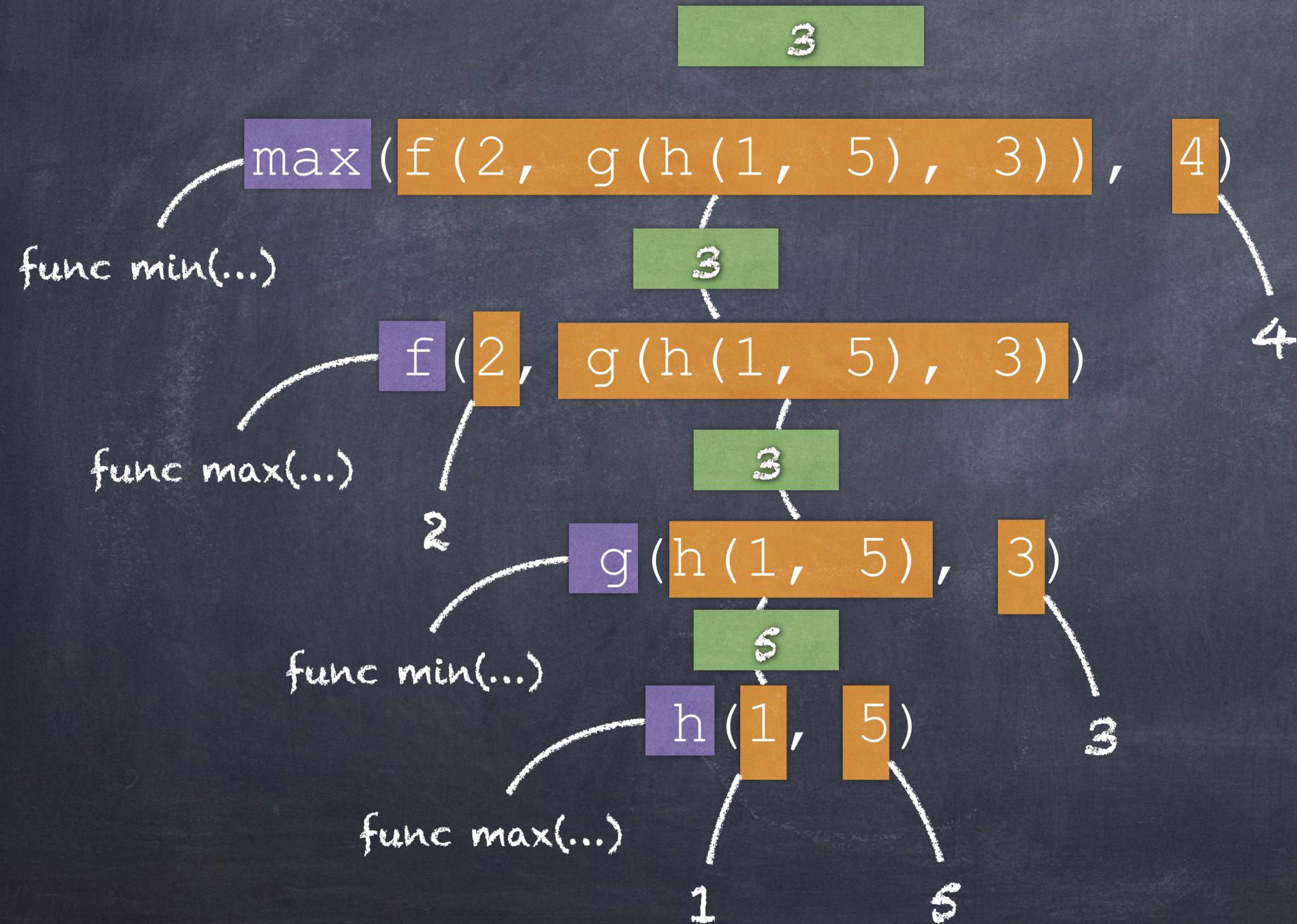
objects

func max(...) [parent=Global]

func min(...) [parent=Global]



回顾之前的例子



函数 (Functions)

函数

① 函数让我们可以将表达式、计算的序列都抽象 (abstract) 出来

② 函数接受一些输入 (即实际参数, arguments), 然后将其转化为输出 (即返回值, return value)。



函数

- 我们通过使用 `Def` 语句来创建函数。

定义函数

函数签名 (signature), 给定名字和参数

```
def <name> (<parameters>):  
    return <return expression>
```

函数体 (body), 定义了当函数作用时产生的的计算

```
def square(x):  
    return x * x  
y = square(-2)
```

● 执行一个 def 语句:

1. 创建一个以 `<name> (<parameters>)` 为签名的函数。
2. 将函数体设为冒号行后面的“缩进”的所有语句。
3. 将 `<name>` 绑定到当前环境的这个函数。

函数在环境图中的形式

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 y = square(-2)
```

Global Frame	
mul	• →
square	• ↘

内建 (内置) 函数
(built-in)

func mul(...) [parent=Global]

func square(x) [parent=Global]

自定义函数
(user-defined)

• Def语句就是一种赋值语句，其将名字和函数值
(function value) 绑定。

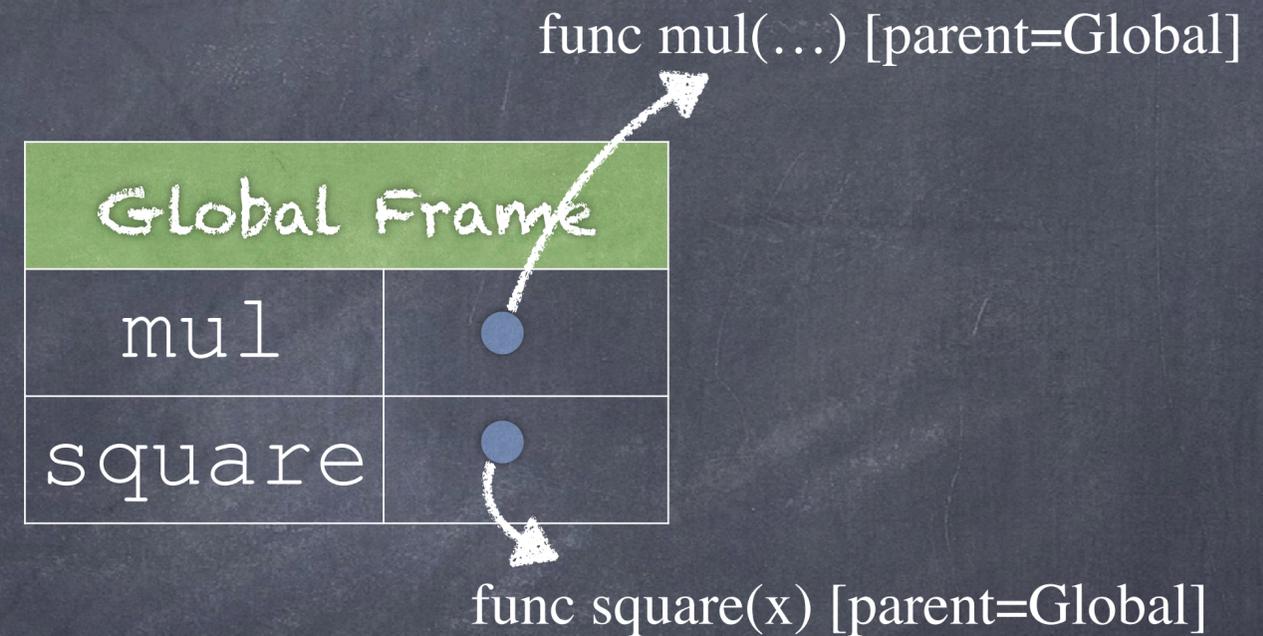
函数调用

调用（作用）一个自定义函数：

1. 创建一个新的环境帧（`Environment frame`）。
2. 将函数的形式参数（`parameters`）在这个新环境中绑定到实参（`arguments`）上。
3. 在该新环境中执行函数体。

函数调用

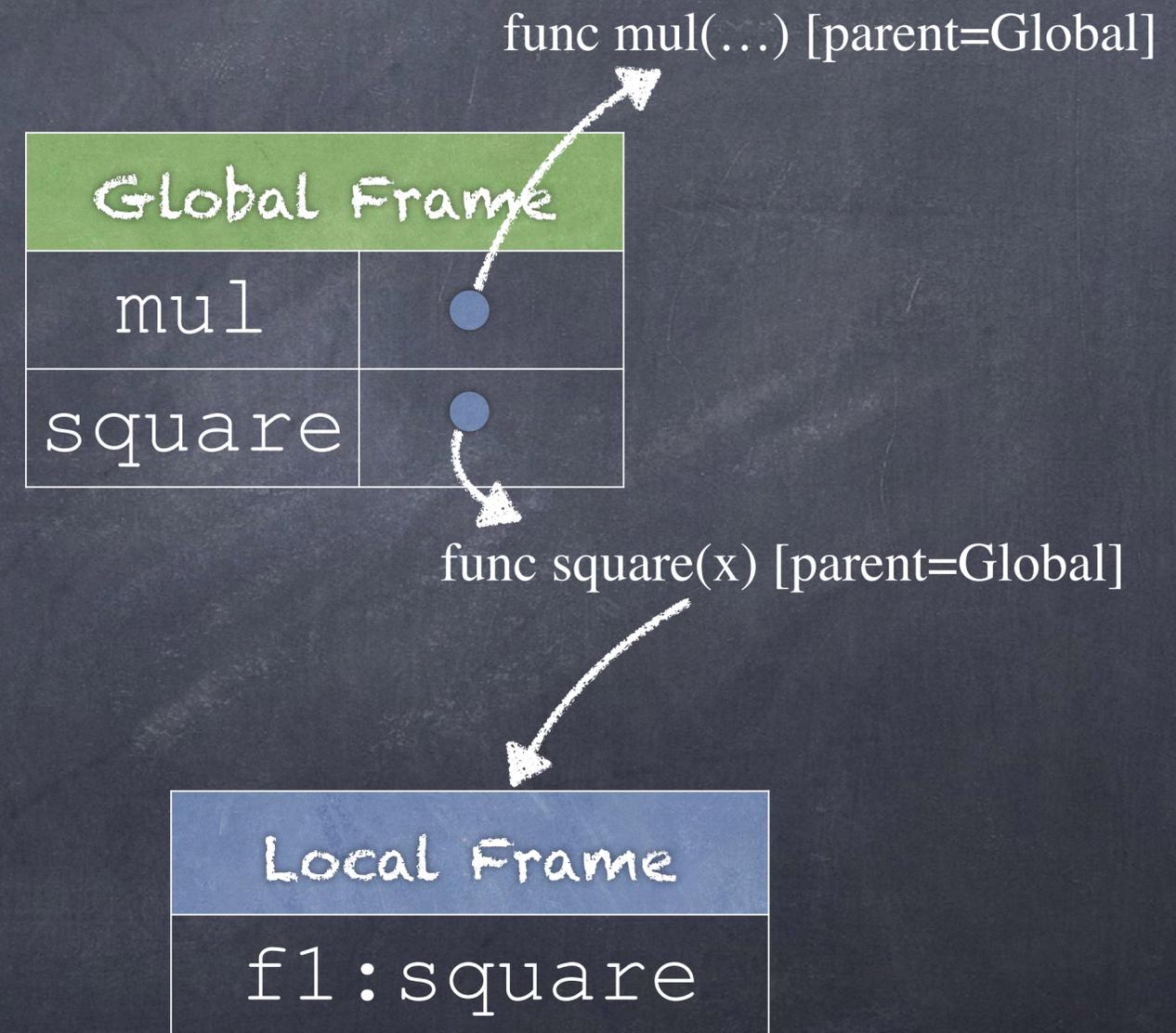
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 y = square(-2)
```



函数调用

1. 创建一个新的环境帧 (Environment frame)

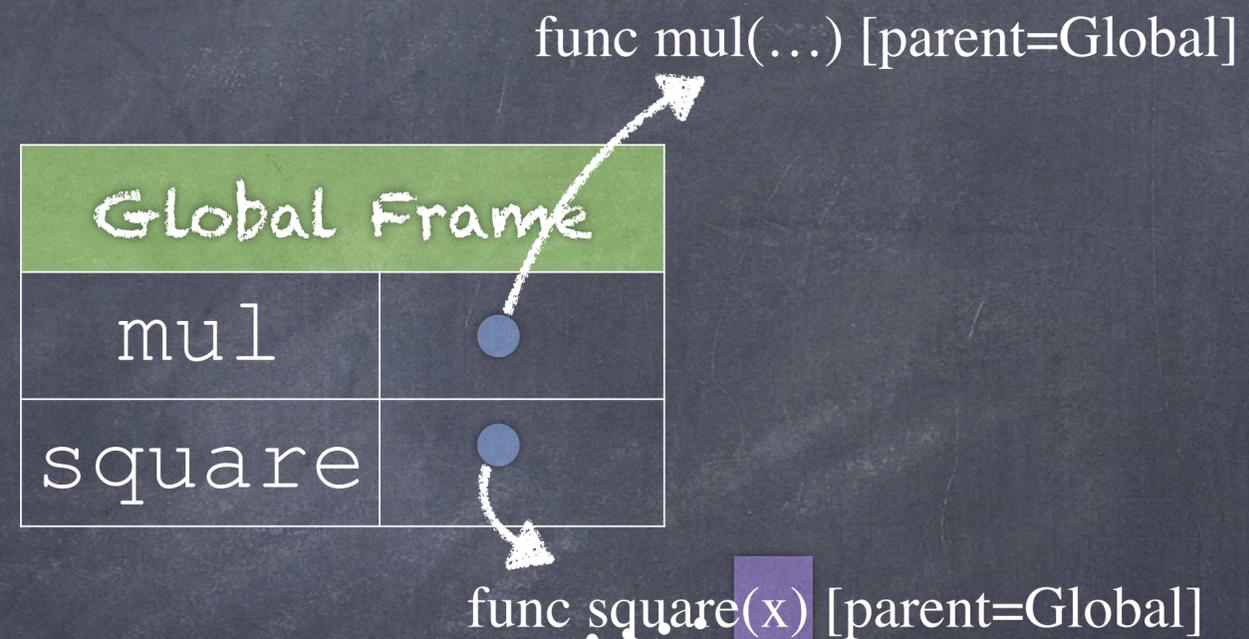
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 y = square(-2)
```



函数调用

2. 将函数的形式参数在这个新环境中绑定到实参上

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 y = square(-2)
```



形参

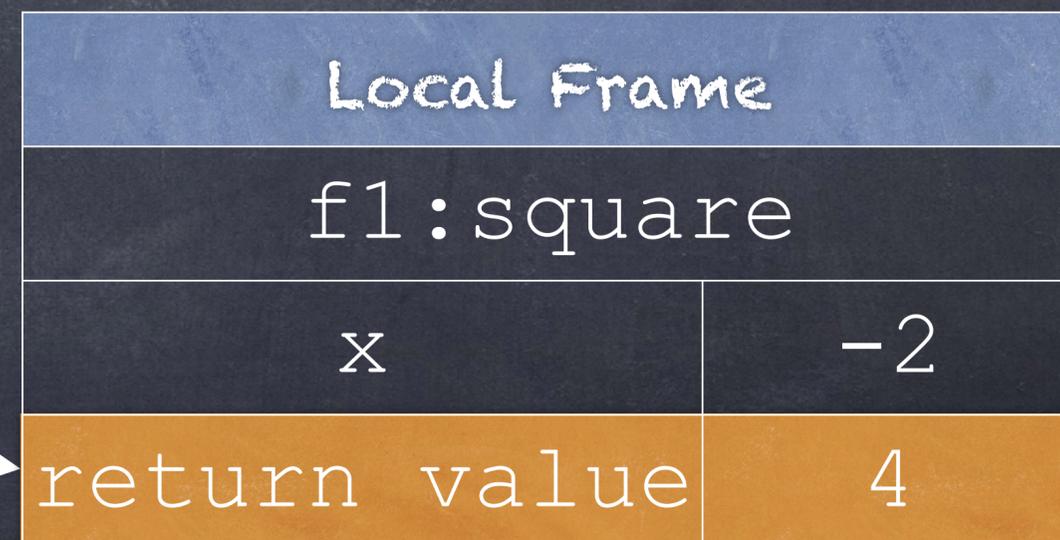
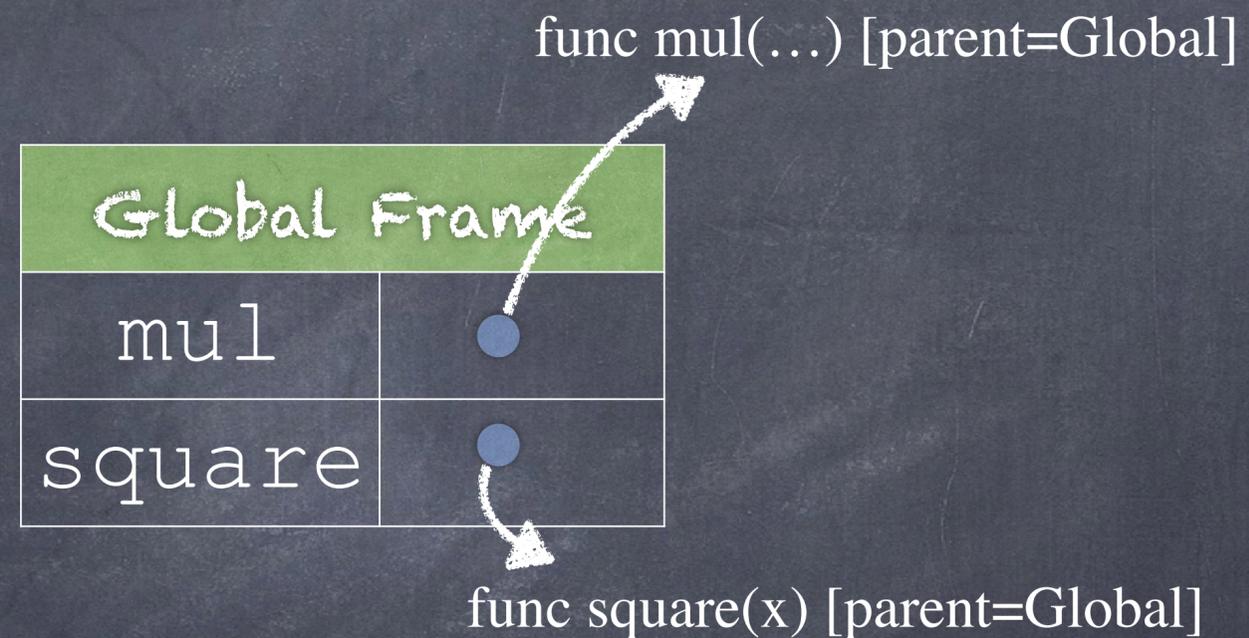


实参

函数调用

3. 在该新环境中执行函数体。

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 y = square(-2)
```



回顾

求值 (Evaluate)

对运算符子表达式求值

对每个操作数子表达式求值

作用 (Apply)

将运算符的值应用在操作数的值上

在新环境中执行函数体

```
def square(x):  
    return x * x
```

运算符: square
函数: func square(x)

```
square(1 - 3)
```

intrinsic name

操作数: 1 - 3
实参: -2

Local Frame	
f1: square	
x	-2
return	4

形参绑定到实参

回顾

绑定：

- = 号会将名字和值绑定到当前环境中
- def 会将函数的名字和具体的函数绑定到当前环境中
- import 语句将名字和值（可以是具体函数）绑定到当前环境中

回顾

比如：

模块

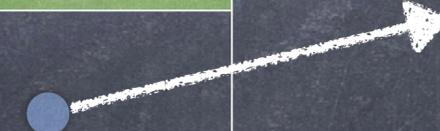
```
from operator import mul
```

Global Frame

mul

内建（内置）函数
(built-in)

func mul(...) [parent=Global]



回顾

比如：

```
from src.lab00 import *
```

Global Frame	
twenty_twenty_two	●
sum	●
diff_square	●

自定义函数

func twenty_twenty_two()

func sum(a, b)

func diff_square(a, b)

回顾

👁 比如：

模块

```
from math import pi
```

Global Frame

pi

3.141592654

回顾

比如：

```
from test import radius
```

Global Frame

radius

10

sample

test.py

test.py

```
radius = 10
```

```
[>>> from test import radius
```

```
[>>> radius
```

```
10
```

```
>>>
```

print 和 none

print 和 none

- 特殊的关键词 `None` 在 Python 里代表 `Nothing`。
- 一个函数如果没有返回一个值的话，就是返回 `None`。
- 注意: `None` 不会作为一个表达式的值被解释器显示

None意味着没有东西返回

```
>>> def does_not_return_square(x):  
...     x * x  
... 
```

没有返回值

None值不会被显示

```
>>> does_not_return_square(4)  
>>> sixteen = does_not_return_square(4)  
>>> sixteen + 4
```

sixteen绑定到None

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

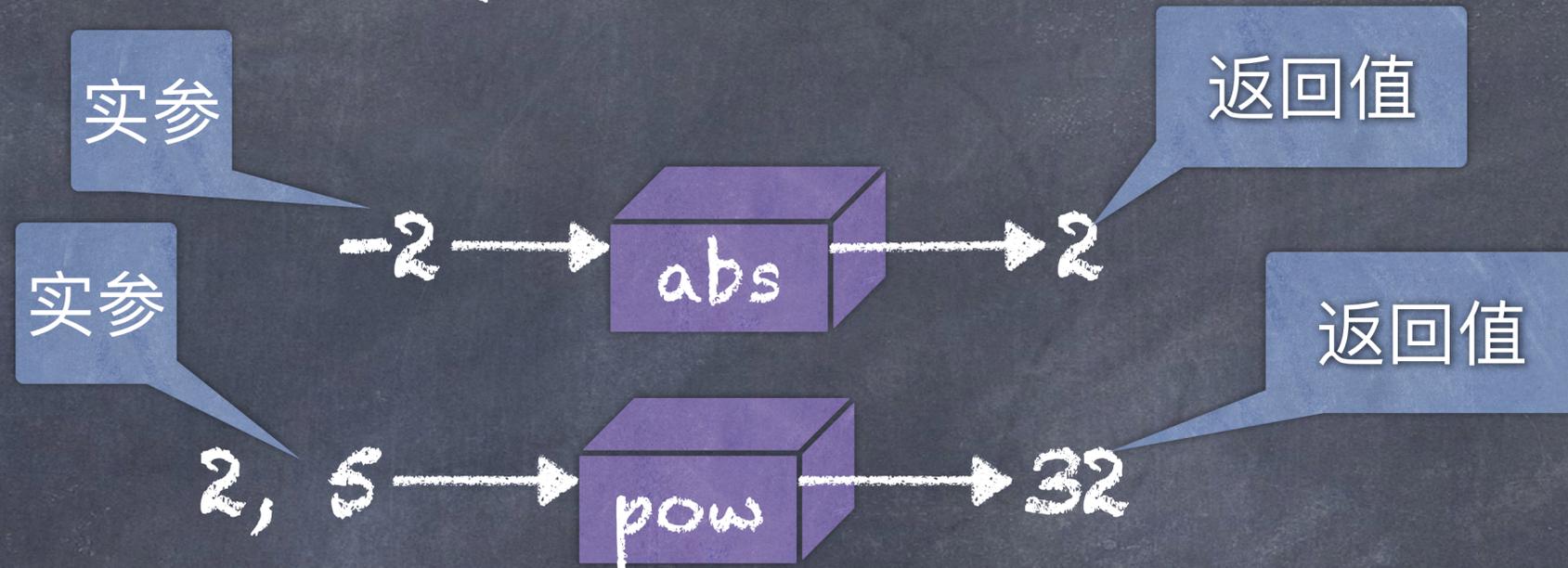
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

纯函数 Pure Functions VS

非纯函数 Non-pure Functions

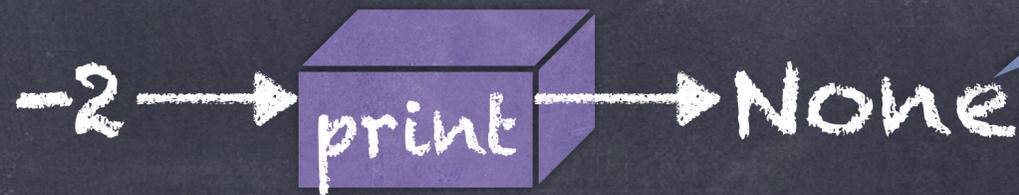
纯函数

- 只返回值



非纯函数

- 会有一些额外的作用



python显示"-2"

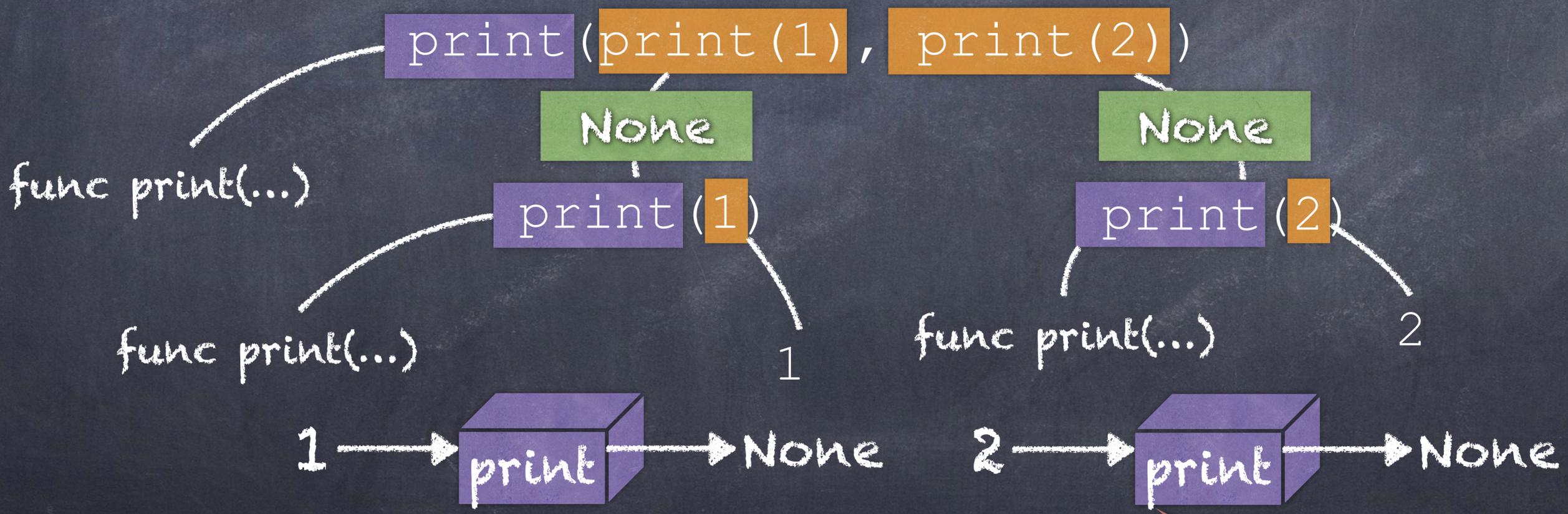
额外作用不是值，是调用函数所发生的“事”

含有print的嵌套表达式



```
>>> print(print(1), print(2))  
1  
2  
None None
```

python显示"None None"



python显示"1"

python显示"2"

print vs return

函数设计

除了正确的理解和写出函数之外，还应追求良好的风格

函数设计

一些约定俗称的规范：

- 函数名称应该小写，以下划线分隔。提倡描述性的名称。

- 比如对参数应用的操作（如 `add`、`print`、`set_score`），或者结果（如 `max`、`abs`）

- 参数名称应小写，以下划线分隔。提倡单个词的名称。

- 参数名称应该反映参数在函数中的作用（非常明确时，也可用单个字母，但要避免类似 `l`、`I` 等和数字容易混淆的字母）。

函数的设计

👁️ 什么是一个好的函数设计？

🎯 每个函数应该只做一个任务！

👤 不要重复你自己！

🌱 应该更具有—般性！

函数的设计

- ◎ 每个函数应该只做一个任务！
- ◎ 这个任务可以使用短小的名称来定义。
- ◎ 顺序执行多个任务的函数应该拆分在多个函数中。

```

public static String testableHtml( PageData pageData,
boolean includeSuiteSetup

) throws Exception {
WikiPage wikiPage = pageData.getWikiPage(); StringBuffer buffer = new StringBuffer();

if (pageData.hasAttribute("Test")) {
if (includeSuiteSetup) { WikiPage suiteSetup =
PageCrawlerImpl.getInheritedPage( SuiteResponder.SUITE_SETUP_NAME, wikiPage
);
if (suiteSetup != null) {
WikiPagePath pagePath = suiteSetup.getPageCrawler().getFullPath(suiteSetup);
String pagePathName = PathParser.render(pagePath); buffer.append("!include -setup .")
.append(pagePathName) .append("\n");
} }
WikiPage setup = PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
if (setup != null) { WikiPagePath setupPath =
wikiPage.getPageCrawler().getFullPath(setup); String setupPathName = PathParser.render(setupPath); buffer.append("!include -setup .")
.append(setupPathName) .append("\n");
} }
buffer.append(pageData.getContent()); if (pageData.hasAttribute("Test")) {
WikiPage teardown = PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
if (teardown != null) { WikiPagePath teardownPath =
wikiPage.getPageCrawler().getFullPath(teardown);
String teardownPathName = PathParser.render(teardownPath); buffer.append("\n")
.append("!include -teardown .") .append(teardownPathName) .append("\n");
}
if (includeSuiteSetup) { WikiPage suiteTeardown =
PageCrawlerImpl.getInheritedPage( SuiteResponder.SUITE_TEARDOWN_NAME, wikiPage
);
if (suiteTeardown != null) {
WikiPagePath pagePath = suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
String pagePathName = PathParser.render(pagePath); buffer.append("!include -teardown .")
.append(pagePathName) .append("\n");
} }
} pageData.setContent(buffer.toString());
return pageData.getHtml();
}

```

```

public static String renderPageWithSetupsAndTeardowns(
PageData pageData, boolean isSuite
) throws Exception {

boolean isTestPage = pageData.hasAttribute("Test");
if (isTestPage) {
WikiPage testPage = pageData.getWikiPage();
StringBuffer newPageContent = new StringBuffer();
includeSetupPages(testPage, newPageContent, isSuite);
newPageContent.append(pageData.getContent());
includeTeardownPages(testPage, newPageContent, isSuite);
pageData.setContent(newPageContent.toString());
}
return pageData.getHtml();
}

```



《clean code》中的例子

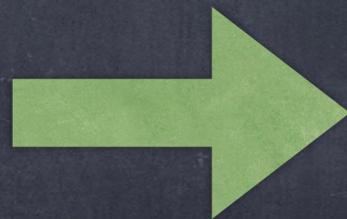
函数的设计

● 不要重复你自己！

● 多个代码段不应该描述重复的逻辑。逻辑应该只实现一次，指定一个名称，并且多次使用

● 如果发现自己复制粘贴一段代码，可能发现了一个使用函数抽象的机会。

```
print("ctrl c");  
print("ctrl c");  
print("ctrl c");  
print("ctrl c");
```



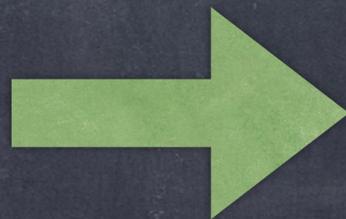
```
def printc():  
    print("ctrl c")  
  
def callMultiple(f, n):  
    for i in range(0, n):  
        f()  
  
callMultiple(printc, 4)
```

函数的设计

● 应该更具有有一般性！

● 很多特定功能其实是某个更加泛化功能的特例

```
def square(x):  
    return x * x
```



```
from math import pow  
square = lambda x: pow(x, 2)
```

函数的设计

● 应该附带函数的文档说明

- 文档字符串通常使用三个引号。第一行描述函数的任务。随后的一些行描述参数，并且澄清函数的行为。

```
def pressure(v, t, n):  
    """Compute the pressure in pascals of an ideal gas.  
  
    Applies the ideal gas law: http://en.wikipedia.org/wiki/Ideal\_gas\_law  
  
    v -- volume of gas, in cubic meters  
    t -- absolute temperature in degrees kelvin  
    n -- particles of gas  
    """  
  
    k = 1.38e-23 # Boltzmann's constant  
    return n * k * t / v
```

Any questions ?