

词汇表

2025年1月19日 9:51

★ A

annotate 注释
argument 实参

★ B

bracket 方括号
brace 括号 curly braces

★ C

call function 函数调用
cube 三次幂
colon 冒号

★ D

discriminant 判别式
decorator 装饰器
denominator 分母
divide 除法
divisor 约数

★ E

expression 表达式

★ F

factorial 阶乘
Compute the falling factorial of n to depth k .
 $(n)_k = n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)$

★ G

generic 通用的

★ H

★ I

interpreter 解释器
indices 指标
iteration 可迭代

★ J

★ K

★ L

★ M

modulo 取余%

★ N

numerator 分子

★ O

over除以 n over d

operator运算符

operand操作数

★ P

prime 质数

parenthese括号

parameter形参

- 在调用函数时，传递的值被称为 实参 (argument)，而在定义函数时，接收的变量被称为 形参 (parameter/formal parameter)

```
def add(a, b): # a 和 b 是形参 (parameters)
    return a + b
```

```
result = add(3, 5) # 3 和 5 是实参 (arguments)
print(result) # 输出 8
```

product乘积

★ Q

quadratic平方的

quotient商

★ R

remainder余数

★ S

square平方，是numpy的，python没有直接的

syntax语法

statement语句

- 表达式 (Expression) 产生一个值，而 语句 (Statement) 是执行一个操作，不返回值。

示例：

表达式

a + b # 这是一个表达式，结果为a和b的和

语句

x = a + b # 这是一个语句，其中 "a + b" 是表达式，而整个 "x = a + b" 是赋值语句

signature函数签名

signature 指的是函数的参数列表及返回值的定义，即函数的输入和输出。主要用于描述函数的参数类型、默认值、返回值类型等信息。

```
from inspect import signature
```

```
def greet(name: str, age: int = 18) -> str:
```

```
return f"Hello, {name}. You are {age} years old."
```

```
sig = signature(greet)
```

```
print(sig) # 输出: (name: str, age: int = 18) -> str
```

★ T

time 乘

★ U

★ V

★ W

★ X

★ Y

★ Z

pow

2025年1月24日 19:10

`pow(x, y)`

计算 x 的 y 次幂

python console

2025年1月24日 10:17

如果你在 Python Console 里想切换到某个目录，不能用 `cd`，而应该用：

```
import os
os.chdir("E:/CS61a/disc01") # 替代 cd 命令
print(os.getcwd()) # 确保已经切换
```

```
import sys
sys.path.append("E:/CS61a/1") # 让 Python 也能找到这个目录
```

举例：

```
import sys
import os
```

```
os.chdir("E:/CS61a/1") # 切换目录
sys.path.append("E:/CS61a/1") # 手动添加 Python 搜索路径
```

```
from race import race # 现在可以成功导入
print(race(5, 7))
```

DEBUG

2025年1月22日 9:57

<https://cs61a.org/articles/debugging/>

一、回溯

二、doctest

- doctest 是 Python 标准库中的一个模块，它可以在 函数字符串（docstring） 里写测试代码，并在运行时自动验证这些测试。
- `python -m doctest ex.py` 会自动运行 `ex.py` 里的所有 doctest 测试。只在 docstring 里编写的测试会被 doctest 识别并运行。
- 运行 doctest 时 如果所有测试通过，不会有任何输出，但如果有任何错误，会显示对比的结果。

`python -m doctest file.py -v`
可以看到详细的过程

testmod

1.直接在.py文件里加

```
from doctest import testmod
testmod()
```

直接显示测试结果

run_docstring_examples

`run_docstring_examples()` 函数：`run_docstring_examples()` 函数是 doctest 模块提供的一种工具，它允许你指定一个特定的函数以及全局环境来验证该函数的 docstring 中的测试案例。

```
from doctest import run_docstring_examples
run_docstring_examples(divide_exact, globals(), True)
```

第一个参数：需要测试的函数，例如 `divide_exact`。

第二个参数：`globals()`，它是 Python 的内置函数，用于返回当前全局符号表。这个参数让 `run_docstring_examples()` 可以在全局作用域中找到该函数，并执行 docstring 中的代码。

第三个参数：`True` 表示以 详细模式 输出测试的结果。你可以选择设置为 `False`，只输出简单的通过或失败结果。

三、error types

OK检测的方法

2025年1月18日 21:21

```
pip install ok  
python ok -q pokemon --local
```

做lab的办法

2025年1月21日 20:22

terminal切到文件夹

python ok -q 对应的名称 -u --local

python ok -q double_eights --local

0 None False

2025年1月22日 19:26

- 0:
 - 0 是一个 **整数**，它的值是零。
 - 在布尔上下文中，0 被视为 **False**。例如，if 0: 的条件将会被认为是 False，即该语句块不会执行。
- None:
 - None 是一个 **特殊类型**，用来表示“没有值”或者“空值”。
 - 在布尔上下文中，None 被视为 **False**，例如，if None: 也会被认为是 False。
 - None 不是一个数值，它是 Python 中的一个特定类型 `NoneType` 的唯一值。

If elif else

2024年2月7日 9:48

判断是互斥且有顺序的。

优先级if>elif>else

每个 if 语句都会被单独检查，即**所有 if 语句**都可能执行。

elif 只会在前面的 if 语句不成立时才执行，**一旦有一个条件满足，后续的 elif 和 else 语句**就不会再被检查。不能有多余个else

```
... if score >= 90:
...     print("A")
... if score >= 80:
...     print("B")
... if score >= 70:
...     print("C")
... else:
...     print("D")
...
B
C
>>> score = 85
...
... if score >= 90:
...     print("A")
... elif score >= 80:
...     print("B")
... elif score >= 70:
...     print("C")
... else:
...     print("D")
...
B
>>> score = 60
...
... if score >= 90:
...     print("A")
... if score >= 80:
...     print("B")
... if score >= 70:
...     print("C")
... else:
...     print("D")
... else:
...     print("E")
Traceback (most recent call last):
  File "C:\Users\zzzibo\.conda\envs\pythonProject\Lib\code.py", line 63, in runsource
    code = self.compile(source, filename, symbol)
    ~~~~~^
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\zzzibo\.conda\envs\pythonProject\Lib\codeop.py", line 160, in __call__
    return _maybe_compile(self.compiler, source, filename, symbol)
    ~~~~~^
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\zzzibo\.conda\envs\pythonProject\Lib\codeop.py", line 73, in _maybe_compile
    ~~~~~^
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
>>>
```

return print

2025年1月22日 22:15

在函数内部，`return` 会立即终止函数的执行并返回一个值。如果函数在某个条件下执行了 `return`，后面的代码就不会再执行了。（见 `disc01 prime.py`）

`return` 在 Python（以及大多数编程语言）中的作用 远远超越 `while`、`if`、`for` 或其他控制结构，它是整个函数的终结点，一旦执行，整个函数就会立即终止，不管 `return` 语句嵌套在多少层控制结构中。

`print` 会将内容输出到屏幕上，但不会终止函数的执行，函数会继续执行后续的代码。
`print` 会在屏幕上显示字符串的内容，而 `return` 返回的是值本身，不会直接显示，除非你用 `print` 来打印它。（见 `lab01.py`）

- `print()` 只是把内容显示在屏幕上，不会存入变量。
- `return` 才是函数真正的返回值，会存到变量里。

见 `lab02-Q2`

print

2025年1月21日 20:00

What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

This expression	Evaluates to	Interactive Output
5	5	5
print(5)	None	5
print(print(5))	None	5 None
delay(delay())(6)()	6	delayed delayed 6
print(delay(print())(4))	None	delayed 4 None

- 在 Python 中，当我们在嵌套作用域（闭包）中定义变量时，如果嵌套函数内部重新定义了一个与外部变量同名的变量，那么在嵌套函数内部，外部变量的作用域就会被遮蔽（shadowed），也就是说，嵌套函数无法直接引用外层变量。

1. pirate 函数：接受一个参数 arggg，在内部定义了一个嵌套函数 plunder，然后返回这个嵌套函数。
2. plunder 函数：也接受一个参数 arggg。
3. 由于 plunder 中定义了一个参数 arggg，这会遮蔽外层 pirate 中的参数 arggg。

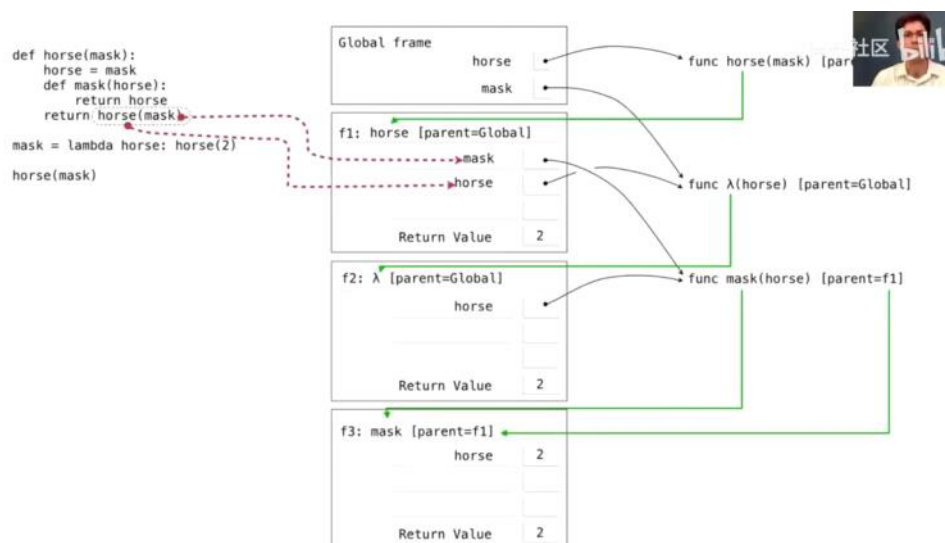
见lab01

```
>>> def how_big(x):
...     if x > 10:
...         print('huge')
...     elif x > 5:
...         return 'big'
...     if x > 0:
...         print('positive')
...     else:
...         print(0)
>>> how_big(7)      # A returned string is displayed with single quotes
_____ 'big'
>>> print(how_big(7)) # A printed string has no quotes
_____ big
>>> how_big(12)
_____ huge
positive
>>> print(how_big(12))
_____ huge
```

```
positive
None
>>> print(how_big(1), how_big(0))
_____positive
0
None None
```

house(mask)

2025年2月1日 10:18



- `mask = lambda horse: horse(2)`

这里 `mask` 是一个 lambda 函数，它接收一个参数 `horse`。

它的功能是：调用 `horse(2)`，将 2 作为参数传递给 `horse`。

`mask` 本身只是一个函数，不会直接等于 2 或其他值。

只有调用 `mask` 时，并且传递一个合适的参数（如另一个函数），才能得到结果。

如果传递的参数是可以调用的函数，例如

```
def my_function(x):  
    return x * 2
```

```
a = mask(my_function)
```

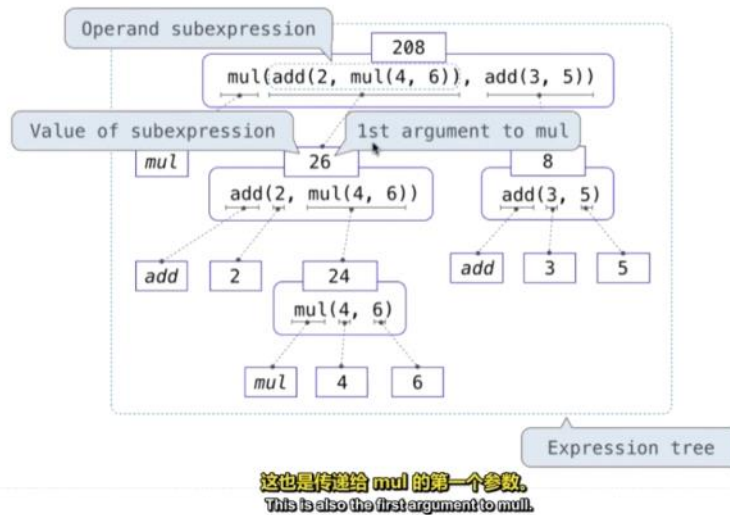
结果是 `a=4`

functions

2025年1月17日 17:21

View → Tool Windows → Python Console

Evaluating Nested Expressions



定义函数

Defining Functions

Assignment is a simple means of abstraction: binds names to values

Function definition is a more powerful means of abstraction: binds names to *expressions*

Function *signature* indicates how many arguments a function takes

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

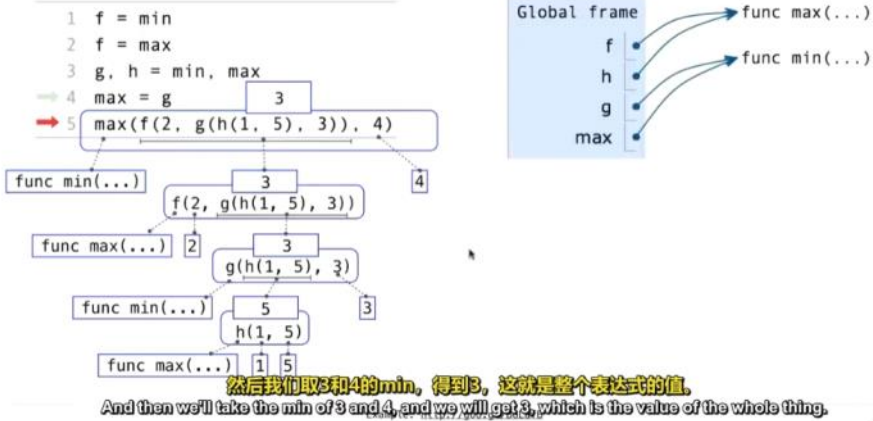
Function *body* defines the computational process expressed by a function

Execution procedure for `def` statements:

1. Create a function with signature `<name>(<formal parameters>)`
2. Set the body of that function to be everything indented after the first line
3. Bind `<name>` to that function in the current frame

Discussion Question 1 Solution

(Demo)



None Indicates that Nothing is Returned

The special value `None` represents nothing in Python

A function that does not explicitly return a value will return `None`

Careful: `None` is not displayed by the interpreter as the value of an expression

```
>>> def does_not_square(x):
...     x * x
...
>>> does_not_square(4)
None value is not displayed
>>> sixteen = does_not_square(4)
>>> sixteen + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

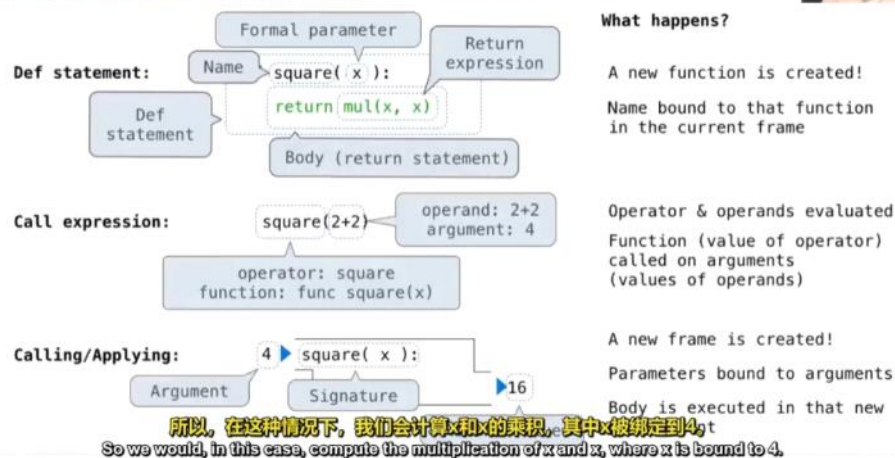
The name `sixteen` is now bound to the value `None`

我将得到一个类型错误, 说不支持操作数类型的加法, `None`类型和整数。
I will get a type error that says unsupported operand types for plus, none type and int.

control

2025年1月17日 19:58

Life Cycle of a User-Defined Function



作用域 (Scope)

优先级LEGB

Local (局部) > Enclosing (外部) > Global (全局) > Built-in (内置)

control-statement

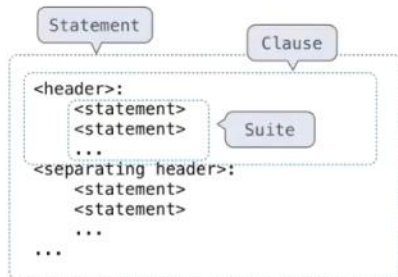
2025年1月18日 11:03

Statements



A **statement** is executed by the interpreter to perform an action

Compound statements:

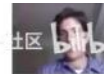


The first header determines a statement's type

The header of a clause "controls" the suite that follows

def statements are compound statements

Conditional Statements



(Demo)

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x < 0:  
        return -x  
    elif x == 0:  
        return 0  
    else:  
        return x
```

1 statement,
3 clauses,
3 headers,
3 suites

Execution rule for conditional statements:

Each clause is considered in order.

1. Evaluate the header's expression.
2. If it is a true value, execute the suite & skip the remaining clauses.

所以最简单的版本只有一个if子句。

So the simplest version just has an if clause.

Syntax Tips

1. Always starts with "if" clause.
2. Zero or more "elif" clauses.
3. Zero or one "else" clause, always at the end.

higher control

2025年1月19日 9:38

Discussion Question

Is this alternative definition of `fib` the same or different from the original `fib`?

```
def fib(n):
    """Compute the nth Fibonacci number?"""
    pred, curr = 0, 1
    k = 1
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

但它更好，因为它可以正确计算第 0 个 Fibonacci 数。
But it's even better because it can compute the 0th Fibonacci number correctly.

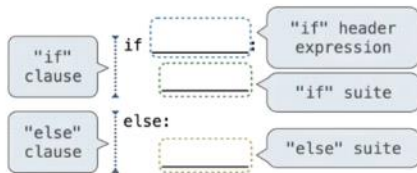
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987



fibonacci左右对比，右更好

If Statements and Call Expressions

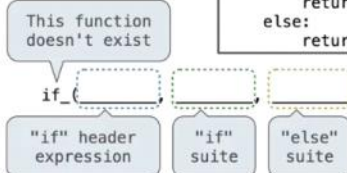
Let's try to write a function that does the same thing as an if statement.



Execution Rule for Conditional Statements:

Each clause is considered in order.

1. Evaluate the header's expression (if present).
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.



Evaluation Rule for Call Expressions:

1. Evaluate the operator and then the operand subexpressions
2. Apply the function that is the value of the operator to the arguments that are the values of the operands

```
def if_(c, t, f):
    if c:
        return t
    else:
        return f
```

This function doesn't exist

can if statement convert to if function?

```
~/lec $ python3 -i ex.py
>>> sqrt(16)
4.0
>>> sqrt(-16)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> real_sqrt(16)
4.0
>>> real_sqrt(-16)
0
>>>
```

```
def if_(c, t, f):
    if c:
        t
    else:
        f

from math import sqrt

def real_sqrt(x):
    """Return the real part of the square root of x."""
    if x >= 0:
        return sqrt(x)
    else:
        return 0
```

error's reason:

```
~/lec $ python3 -i ex.py
>>> sqrt(16)
4.0
>>> sqrt(-16)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> real_sqrt(16)
4.0
>>> real_sqrt(-16)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ex.py", line 11, in real_sqrt
    return if_(x >= 0, sqrt(x), 0)
ValueError: math domain error
>>>
```

```
def if_(c, t, f):
    if c:
        return t
    else:
        return f

from math import sqrt

def real_sqrt(x):
    """Return the real part of the square root of x."""
    return if_(x >= 0, sqrt(x), 0)
```



when we replace the conditional statement with a call expression, we had a different evaluation rule

在评估这个过程中，我们先确定我们要调用的函数以及我们在执行主体之前调用它的所有三个参数。For evaluating this, we figure out what function we're going to call and all three arguments that we're calling it on before we ever execute the body. 在我们讨论0或者其他内容之前，由于对负数进行了平方根运算，我们就遇到了一个错误。And before we ever got to the 0 or got to this, we reached an error because the square root of a negative number was taken

call expression don't allow you to skip evaluating parts of the call expression, all the parts are always evaluated before the function is called

这与控制语句不同，控制语句实际上会选择执行语句的哪些部分以及跳过哪些部分。And that's different from control statements, which actually pick which parts of the statement are executed and which parts are skipped.

举例：

调用表达式 (Call Expression) 中的所有部分（例如参数）都会在函数被调用之前被计算。

这意味着所有的参数都会被计算出来，无论这些参数是否会在函数内部使用。

示例：包含副作用 (Side Effects)

```
def print_value(x):
    print(f"Value: {x}")
    return x
```

```
result = print_value(3 + 2) # 这里 3 + 2 会先被计算，输出 "Value: 5"
```

在这个例子中,虽然函数 print_value 只是打印出参数 x, 但 3 + 2 仍然会在函数调用之前被计算，导致副作用发生（即打印出 "Value: 5"）。

hc-logical operator

2025年1月19日 10:50

logical operator

short-circuit evaluation一旦可以确定结果，Python 就不会再计算后续的表达式。

and

To evaluate the expression <left> and <right>:

1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

To evaluate the expression <left> or <right>:

1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

and 会返回第一个假值，而 or 会返回第一个真值。

逻辑与运算符（and）求值规则：

首先计算左侧表达式（<left>）：

- 计算左边的子表达式（<left>）。如果它已经是一个假值（如 False、None、0、空字符串等），那么整个 and 表达式的结果就是左侧表达式的值。
- 如果左侧的结果是一个假值：
如果左侧的表达式结果是一个假值，那么整个 and 表达式的结果就是左侧表达式的结果。Python 在这种情况下 不会再计算右侧的表达式（<right>），因为无论右侧的表达式是什么，and 运算的结果已经确定了。
- 否则，返回右侧表达式的值：
如果左侧的表达式不是假值（即它的值被认为是“真”的），那么 Python 会继续计算右侧的表达式，并且返回右侧表达式的值。

举个例子：

```
python
a = 0
b = 5
```

```
result = a and b
```

步骤 1：首先计算 a，它的值是 0（假值）。

步骤 2：由于 a 是假值，整个 and 表达式的结果是 0，不需要再计算右边的 b。

再看一个例子：

```
python
a = 10
b = 5
```

```
result = a and b
```

步骤 1：首先计算 a，它的值是 10（真值）。

步骤 2：由于 a 不是假值，继续计算右侧的 b，它的值是 5。

步骤 3: 整个 and 表达式的结果是 b 的值, 即 5。

重点:

短路求值 (Short-circuit evaluation) : 当左侧表达式的结果为假值时, Python 会立即返回左侧的结果, 而不会去计算右侧的表达式。这样可以提高效率, 避免不必要的计算。



```
~/lec$ python3 -i ex.py
>>> has_big_sqrt(1)
False
>>> has_big_sqrt(1000)
True
>>> has_big_sqrt(-1000)
False
>>>

from math import sqrt
def has_big_sqrt(x):
    return x > 0 and sqrt(x) > 10
```

逻辑或运算符 (or) 的求值规则:

1. 首先计算左侧表达式 (<left>) :
计算左边的子表达式 (<left>)。如果它已经是一个“真”值 (例如 True、非零数字、非空字符串等), 那么整个 or 表达式的结果就是左侧表达式的值。
2. 如果左侧的结果是一个真值:
如果左侧表达式的结果是“真值”, 那么 or 表达式的结果就是左侧表达式的值。Python 不会再计算右侧的表达式 (<right>), 因为无论右侧的表达式是什么, or 运算的结果已经确定了。
3. 否则, 返回右侧表达式的值:
如果左侧的表达式是“假值” (如 False、None、0、空字符串等), 那么 Python 会继续计算右侧的表达式, 并返回右侧表达式的值。

例子:

1. 当左侧是 “真值” 时:

```
a = 10
b = 5
```

```
result = a or b # 这里a是10, 是真值
```

步骤 1: 计算 a, 它的值是 10 (真值)。

步骤 2: 由于 a 是真值, 整个 or 表达式的结果是 10, Python 不会继续计算右侧的 b。

2. 当左侧是 “假值” 时:

```
a = 0
b = 5
```

```
result = a or b # 这里a是0, 假值
```

步骤 1: 计算 a, 它的值是 0 (假值)。

步骤 2: 由于 a 是假值, Python 继续计算右侧的表达式 b, 并返回 b 的值, 即 5。

```
from math import sqrt

def has_big_sqrt(x):
    return x > 0 and sqrt(x) > 10

def reasonable(n):
    return n == 0 or 1/n != 0
```

```
3.print(3) or ""
```

- **执行** `print(3)`
 - `print(3)` 会输出 3, 但它的 **返回值**是 `None` (`print()` 函数没有返回值)。
- **计算** `None or ""`
 - `None` 是 `falsy`, 所以 `or` 继续执行 `""` (空字符串)。
 - `""` 也是 `falsy`, 但它作为 `or` 的最终结果返回。

not




- 在 Python 中，**非零的数值（包括正数和负数）都是 True。**
- 只有 0 被认为是 False。
- not 运算符会 **对布尔值取反**：
 - not True \rightarrow False
 - not False \rightarrow True

hc-generalize

2025年1月19日 11:47

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:   

Area: $1 \cdot r^2$ $\pi \cdot r^2$ $\frac{3\sqrt{3}}{2} \cdot r^2$

那么前三个的公式有什么共同之处，以及为什么这些公式在数学上是正确的。
So what we're going to do is look at these three formulas and find out what they have in common and what makes them specific to a particular shape.

```
~/lec3 python3 -i ex.py
>>> area_circle(10)
314.1592653589793
>>> 10
~/lec3 python3 -i ex.py
>>> area_hexagon(10)
259.8076211353316
>>> area_hexagon(-10)
259.8076211353316
>>>
```

```
"""Generalization."""
from math import pi, sqrt

def area_square(r):
    return r * r

def area_circle(r):
    return r * r * pi

def area_hexagon(r):
    return r * r * 3 * sqrt(3) / 2
```

是一样的，这有点不对。
is the same thing, which is not quite right.

```
~/lec3 python3
Python 3.3.1 (v3.3.1:d9893d13c628, Apr 6 2013, 11:07:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> assert 3 > 2, 'Math is broken'
>>> assert 2 > 3, 'That is false'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: That is false
>>>
```

generalize后，见gene_area.py:

```
"""Generalization."""
from math import pi, sqrt

def area(r, shape_constant):
    assert r > 0, 'A length must be positive'
    return r * r * shape_constant

def area_square(r):
    return area(r, 1)

def area_circle(r):
    return area(r, pi)

def area_hexagon(r):
    return area(r, 3 * sqrt(3) / 2)
```

assert 是 Python 中的一个内置语句，用于断言某个条件是否为真。如果条件为真（即表达式的值为 True），程序会继续执行；如果条件为假（即表达式的值为 False），程序会抛出一个 AssertionError 异常，并终止执行。

第二个例子 三次幂：

```
~/lec3 python3 -m doctest ex.py
~/lec3 python3 -m doctest ex.py
~/lec3
```

```
"""Generalization."""
def sum_naturals(n):
    """Sum the first N natural numbers.
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + k, k + 1
    return total

def sum_cubes(n):
    """Sum the first N cubes of natural numbers.
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + pow(k, 3), k + 1
    return total
```

它和第一个完全相同，除了这个 pow k, 3，所以这个函数可以做得更通用。
They're exactly identical the same, except for this pow k, 3, and this is a pow k, 3, adding the number 3, so maybe we can do better.

environments

2025年1月21日 10:47

```
>>> def f(x, y):
...     return g(x)
...
>>> def g(a):
...     return a + y
...
>>> f(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
  File "<stdin>", line 2, in g
NameError: global name 'y' is not defined
>>>
```

The screenshot shows the Online Python Tutor interface. On the left, the code editor displays the same code as the terminal above. A red arrow points to line 5, which is the first line of the function `g`. Below the code editor, a progress bar indicates 'Step 6 of 9'. The error message 'NameError: global name 'y' is not defined' is shown in red. Below the error message, a legend indicates that a grey arrow points to the line that has just executed, and a red arrow points to the next line to execute. On the right, the 'Frames' and 'Objects' panels are visible. The 'Global frame' contains objects for `f` (pointing to `func f(x, y)`) and `g` (pointing to `func g(a)`). The `f` frame contains `x` (1) and `y` (2). The `g` frame contains `a` (1). At the bottom, there is a 'Generate URL' button and a disclaimer about the tool's use for education research.

Generate URL

By using this tool, you agree to let the 61A staff and their associates record your input, collect it as part of an anonymized dataset of student examples, and release it publicly for the purposes of education research.

To report a bug, click the 'Generate URL' button, paste the URL along with a brief error description in an email, and send the email to philip@pgbovine.net

This version of Online Python Tutor supports Python 3.2 with limited module imports and no file I/O. It is designed for teaching programming, not for running or debugging production code. The code is open source on [GitHub](#).

To share this visualization, click the 'Generate URL' button, paste the URL along with a brief error description in an email, and send the email to philip@pgbovine.net.

Have a question? Maybe the [FAQ](#) or other [documentation](#) can help. Or check out its code at [GitHub](#).

与之前的闭包对比

```
Python
>>> def f(x, y):
...     return g(x)
...
>>> def g(a):
...     return a + y
...
>>> f(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
  File "<stdin>", line 2, in g
NameError: global name 'y' is not defined
>>>
```

```
1 def make_adder(n):
2     def adder(k):
3         return k + n
4     return adder
```



现在与我们在 `make_adder` 中看到的进行对比，其中 `adder` 的主体实际上可以引用 `n`，因为它是嵌套的。

Now contrast that with what we saw with `make adder`, where the body of `adder` could in fact refer to `n`, because it was nested.

higher-order function闭包

2025年1月20日 16:17

Implement the function `make_repeater` that takes a one-argument function `f` and a positive integer `n`. It returns a one-argument function, where `make_repeater(f, n)(x)` returns the value of `f(f(...f(x)...`) in which `f` is applied `n` times to `x`. For example, `make_repeater(square, 3)(5)` squares 5 three times and returns 390625, just like `square(square(square(5)))`.

来自 <<https://www.learncs.site/docs/curriculum-resource/cs61a/homework/hw02>>

```
def make_repeater(f, n):
    """Returns the function that computes the nth application of f.
```

```
>>> add_three = make_repeater(increment, 3)
>>> add_three(5)
8
>>> make_repeater(triple, 5)(1) # 3 * 3 * 3 * 3 * 3 * 1
243
>>> make_repeater(square, 2)(5) # square(square(5))
625
>>> make_repeater(square, 3)(5) # square(square(square(5)))
390625
"""
*** YOUR CODE HERE ***
```

从上面的需求来看，我们需要设计一个函数 `make_repeater(f, n)`，它能够在返回的函数被调用时，正确地将 `f` 函数应用 `n` 次。要实现这个功能，闭包是非常合适的，因为它可以“记住”函数 `f` 和 `n` 的值，并且能够在返回的函数中使用这些值。

how to use the inner function

详见lab02

example:

```
def multiply_by(m):
    def multiply(n):
        return n * m
    return multiply
```

In this particular case, we defined the function `multiply` within the body of `multiply_by` and then returned it. Let's see it in action:

```
>>> multiply_by(3)
<function multiply_by.<locals>.multiply at ...>
>>> multiply(4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'multiply' is not defined
```

A call to `multiply_by` returns a function, as expected. However,

calling `multiply` errors, even though that's the name we gave the inner function. This is because the name `multiply` only exists within the frame where we evaluate the body of `multiply_by`.

So how do we actually use the inner function? Here are two ways:

```
>>> times_three = multiply_by(3) # Assign the result of the call expression to a name
>>> times_three(5) # Call the inner function with its new name
15
>>> multiply_by(3)(10) # Chain together two call expressions
30
```

有一道很好的练习题目见lab02

lambda expression

2025年1月21日 11:10

Lambda Expressions

```
>>> x = 10
>>> square = x * x
>>> square = lambda x: x * x
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

A function
with formal parameter x
that returns the value of "x * x"

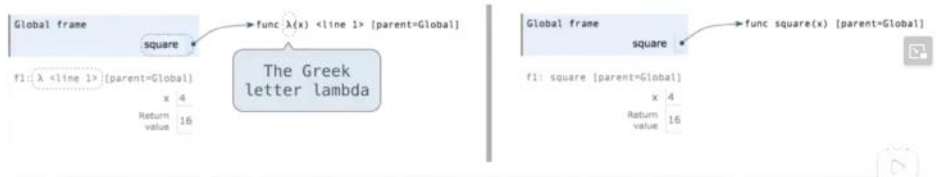
阅读 lambda 表达式的方式是将其视为一个带有形式参数 x 的函数，返回 x 乘以 x 的值。
所以最终，人们会在你周围开始在非正式的谈话中使用 lambda 这个词。
And the way to read a lambda expression is as a function with formal parameter x that returns the value of x times x . So eventually, people will start to use the word lambda in casual conversation around you.

lambda无需return，直接写表达式，所以总是创建简单的函数，评估单个表达式

Lambda Expressions Versus Def Statements

 `square = lambda x: x * x` VS `def square(x):`
 `return x * x`

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the frame in which they were defined.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name.



lambda匿名函数

2024年2月14日 16:32

函数的定义中

- def关键字，可以定义带有名称的函数
- lambda关键字，可以定义匿名函数（无名称）
- 有名称的函数，可以基于名称重复使用。
- 无名称的匿名函数，只可临时使用一次。

lambda 传入参数：函数体(一行代码)

带有名称的函数：

- 通过def关键字，定义一个函数，并传入，如下图：

```
def test_func(compute):  
    result = compute(1, 2)  
    print(result)
```

```
def compute(x, y):  
    return x + y
```

```
test_func(compute)      # 结果：3
```

- 也可以通过lambda关键字，传入一个一次性使用的lambda匿名函数

```
def test_func(compute):  
    result = compute(1, 2)  
    print(result)
```

```
test_func(lambda x, y: x + y)      # 结果：3
```

使用def和使用lambda，定义的函数功能完全一致，只是lambda关键字定义的函数是匿名的，无法二次使用

- Lambda expressions can be used as an operator or operand

```
negate = lambda f, x: -f(x)  
negate(lambda x: x * x, 3)
```

来自 <<https://www.learncs.site/docs/curriculum-resource/cs61a/lab/lab02#higher-order-functions>>

- 匿名函数有个限制，就是只能有一个表达式，不用写return，返回值就是该表达式的结果。
- 用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

```
f = lambda x:x*x  
print(f(5))
```

同样，也可以把匿名函数作为返回值返回，比如：

```
def build(x, y):  
    return lambda: x * x + y * y
```

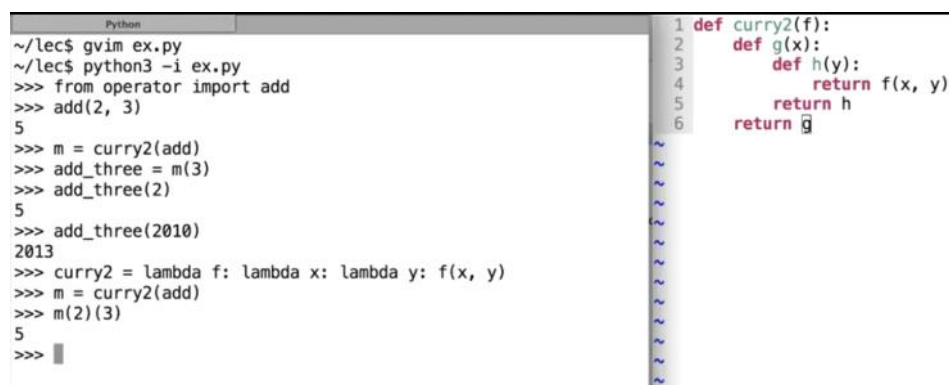

function curring

2025年1月21日 10:47

Function Currying

```
def make_adder(n):  
    return lambda k: n + k
```

```
>>> make_adder(2)(3)  
5  
5  
>>> add(2, 3)  
5
```



结构分析

1. **最外层的** `lambda f::`
 - 这是定义了一个接受参数 `f` 的匿名函数。这个函数的作用是接收一个函数 `f` 作为输入，返回一个新的函数，这个新函数会继续接收参数。
2. **中间的** `lambda x::`
 - 在 `lambda f:` 返回的函数中，`lambda x:` 定义了一个接受单一参数 `x` 的匿名函数。并返回一个函数，这个新函数依赖传入的 `x` 值。
3. **最内层的** `lambda y::`
 - 最内层的 `lambda y:` 定义了一个接受单一参数 `y` 的匿名函数。这个函数的目的是在外层函数已经提供了 `x` 的值后，再接收 `y` 作为参数。
4. 定义的顺序非常重要！必须是先传入 `f`, 再传入 `x`, 再传入 `y` 详见 lab02-Q3
`curry2(f)(x)(y)` # 正确!
`curry2(x)(f)(y)` # ✗ 不符合结构
`curry2(2)` # ✗ 2 不是函数

每一层的 lambda 是返回了一个新的函数，且每一层都依赖于外层传入的值。

lambda 嵌套求值时，先计算内部，再计算外部！

思考：

```
result = (lambda x: 2 * (lambda x: 3)(4) * x)(5)
```

Functional Abstraction

2025年1月31日 15:54

函数抽象就是给某个计算过程起个名字，然后整个过程都用这个名字来引用，而不用担心具体的实现细节。

So functional abstraction is giving a name to some computational process and then referring to that process as a whole without worrying about its implementation details.

The screenshot shows a video player with the title "Functional Abstractions". It displays two Python functions: `def square(x): return mul(x, x)` and `def sum_squares(x, y): return square(x) + square(y)`. Below the code, a quiz asks "What does sum_squares need to know about square?". The options and answers are: "Square takes one argument." (Yes), "Square has the intrinsic name square." (No), "Square computes the square of a number." (Yes), and "Square computes the square by calling mul." (No). At the bottom, there is a subtitle in Chinese: "或者我可以发明一些奇怪的计算平方的方式，比如这样。" and its English translation: "Or I could invent some strange way of computing squares, such as this." followed by another code example: `def square(x): return mul(x, x-1) + x`.

Choosing Names

The screenshot shows a video player with the title "Choosing Names". It contains text stating: "Names typically *don't* matter for correctness **but** they matter a lot for composition". Below this, there is a table with two columns: "From:" and "To:". The "From:" column lists: `true_false`, `d`, `play_helper`, `my_int`, and `l, I, 0`. The "To:" column lists: `rolled_a_one`, `dice`, `take_turn`, `num_rolls`, and `k, i, m`. To the right of the table, there are three bullet points: "Names should convey the *meaning* or *purpose* of the values to which they are bound.", "The type of value bound to the name is best documented in a function's docstring.", and "Function names typically convey their effect (print), their behavior (triple), or the value returned (abs).".

Which values deserve a name何时需要取名

1.例如 “我计算了a平方加b平方的平方根，既在条件语句中又在赋值语句中，最好给这个东西取个名字，然后在这里和那里都用一下。”

The screenshot shows a video player with the title "Which Values Deserve a Name". It displays the text: "Repeated compound expressions:".

Which Values Deserve a Name

Repeated compound expressions:

```
if sqrt(square(a) + square(b)) > 1:  
    x = x + sqrt(square(a) + square(b))
```



```
hypotenuse = sqrt(square(a) + square(b))  
if hypotenuse > 1:  
    x = x + hypotenuse
```

2.避免表达式过于复杂，最好将有意义的部分提取出分，比如这个二次方程的判别式，这样更易阅读，而不是试图理解整个嵌套

Meaningful parts of complex expressions:

```
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```



```
discriminant = sqrt(square(b) - 4 * a * c)  
x = (-b + discriminant) / (2 * a)
```

3.但有时取长的名字更好


More Naming Tips

- Names can be long if they help document your code:

```
average_age = average(age, students)
```


is preferable to

```
# Compute average age of students  
aa = avg(a, st)
```

- Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc. 

`n, k, i` – Usually integers

`x, y, z` – Usually real numbers

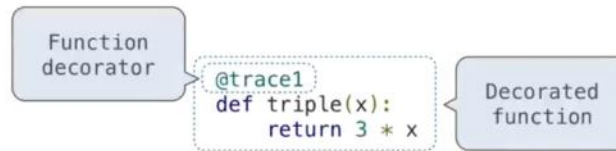
`f, g, h` – Usually functions 

decorator

2025年2月1日 19:57

Function Decorators

(demo)



is identical to

```
def triple(x):
    return 3 * x
triple = tracel(triple)
```

```
>>> sum_squares_up_to(5)
Calling <function sum_squares_up_to at 0x1006ee290> on argument
5
Calling <function square at 0x1006ee170> on argument 1
Calling <function square at 0x1006ee170> on argument 2
Calling <function square at 0x1006ee170> on argument 3
Calling <function square at 0x1006ee170> on argument 4
Calling <function square at 0x1006ee170> on argument 5
55
>>> ^D
~/lec5 python3 -i ex.py
>>> sum_squares_up_to(5)
Calling <function sum_squares_up_to at 0x1006ee290> on argument
5
Calling <function square at 0x1006ee170> on argument 1
Calling <function square at 0x1006ee170> on argument 2
Calling <function square at 0x1006ee170> on argument 3
Calling <function square at 0x1006ee170> on argument 4
Calling <function square at 0x1006ee170> on argument 5
55
>>> ^
```

```
1 def tracel(fn):
2     """Returns a version of fn that first prints
3     it is called.
4
5     fn - a function of 1 argument
6     """
7     def traced(x):
8         print('Calling', fn, 'on argument', x)
9         return fn(x)
10    return traced
11
12 @tracel
13 def square(x):
14     return x * x
15
16 @tracel
17 def sum_squares_up_to(n):
18     k = 1
19     total = 0
20     while k <= n:
21         total, k = total + square(k), k + 1
22     return total
23
```

所以如果你想要一个快捷方式，要将一个函数转换为另一个函数，你可以简单地放一个注解，这称为装饰器。
For transforming a function into another function, you can just put an annotation, which is called a decorator.

*arg

2025年2月3日 10:03

*arg

```
def make_test_dice(*outcomes):
```

加*可以传递任意数量的参数，甚至可以不传递任何参数，并将它们打包成一个元组。

比如：

```
def printed(f):
```

```
    def print_and_return(*args): # 接受任意数量的参数
```

```
        result = f(*args) # 将这些参数传递给目标函数 f
```

```
        print('Result:', result) # 打印结果
```

```
        return result # 返回结果
```

```
    return print_and_return # 返回包装函数
```

使用示例

```
printed_pow = printed(pow) # 将 pow 函数传递给 printed 函数
```

```
printed_pow(2, 8) # 调用 pow(2, 8)
```

再如：

```
def make_test_dice(*outcomes):
```

```
    """Return a dice function that cycles through the outcomes."""
```

```
    def dice():
```

```
        outcome = outcomes[0] # 获取当前结果的第一个骰子值
```

```
        outcomes = outcomes[1:] + [outcome] # 将第一个结果移到末尾，形成循环
```

```
        return outcome # 返回当前的骰子结果
```

```
    return dice # 返回骰子函数
```

参考题目见hog project problem8

nonlocal index

2025年2月4日 15:48

内部嵌套的函数加nonlocal index表明是修改外部作用域（非全局）的Index变量，如果不使用 nonlocal，会遇到以下问题：

`index = (index + 1) % len(outcomes)` 会创建一个新的局部变量 `index`，这会导致外部的 `index` 不会被更新，循环过程中的 `index` 会在每次调用时被重置，结果就是 `dice()` 每次都会返回相同的值，而不会按顺序循环。

recursive functions

2025年2月6日 11:17

所以递归函数就是一个函数，其主体要么直接调用自身，要么间接调用自身
So a recursive function is a function whose body calls itself either directly or indirectly.

一个简单的递归案例：

Sum Digits Without a While Statement

```
def split(n):  
    """Split positive n into all but its last digit and its last digit."""  
    return n // 10, n % 10  
  
def sum_digits(n):  
    """Return the sum of the digits of positive integer n."""  
    if n < 10:  
        return n  
    else:  
        all_but_last, last = split(n)  
        return sum_digits(all_but_last) + last
```

- The def statement header is similar to other functions
- Conditional statements check for base cases
- Base cases are evaluated without recursive calls
- Recursive cases are evaluated with recursive calls

Mutual recursion

相互递归是指两个不同的函数互相调用

Mutual recursion occurs when two different functions call each other.

Luhn Algorithm

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., $7 * 2 = 14$), then sum the digits of the products (e.g., 10: $1 + 0 = 1$, 14: $1 + 4 = 5$).
2. Take the sum of all the digits.

1	3	8	7	4	3
2	3	1+6=7	7	8	3

= 30

The Luhn sum of a valid credit card number is a multiple of 10.


```
Python
~/lec$ python3 -i ex.py
>>> luhn_sum(2)
2
>>> luhn_sum(32)
8
>>> luhn_sum(5105105105100)
20
>>> █
```

I

```
1 def split(n):
2     return n // 10, n % 10
3
4 def sum_digits(n):
5     if n < 10:
6         return n
7     else:
8         all_but_last, last = split(n)
9         return sum_digits(all_but_last) + last
10
11 def luhn_sum(n):
12     if n < 10:
13         return n
14     else:
15         all_but_last, last = split(n)
16         return luhn_sum_double(all_but_last) + last
17
18 def luhn_sum_double(n):
19     all_but_last, last = split(n)
20     luhn_digit = sum_digits(2 * last)
21     if n < 10:
22         return luhn_digit
23     else:
24         return luhn_sum(all_but_last) + luhn_digit
```



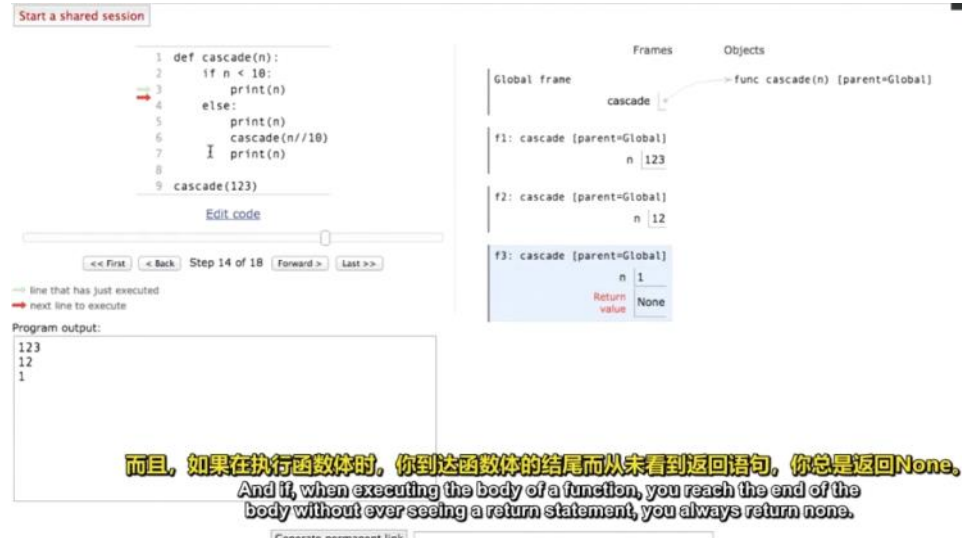
所以，如果你想在你的信用卡上尝试这个，你可以检查确保它有效，方法是确保它的luhn_sum是10的倍数。

So, if you want to try this out on your own credit card, you can check to make sure that it's valid by making sure that its Luhn sum is a multiple of 10.

打印类题目

2025年2月6日 16:30

cascade recursion



而且, 如果在执行函数体时, 你到达函数体的结尾而从未看到返回语句, 你总是返回None.
And if, when executing the body of a function, you reach the end of the body without ever seeing a return statement, you always return none.

调用栈中的结构

每一次调用 `cascade` 函数时, Python 都会为该调用分配一个新的 **栈帧 (frame)**。栈帧会保存:

1. 当前调用的参数 (比如 `n` 的值)。
2. 函数内的局部变量。
3. 当前执行到的代码位置。

当递归调用完成时, 当前栈帧被销毁, 并将控制权交还给上一个栈帧, 继续执行未完成的代码。

当前状态

1. 调用栈显示了 3 个栈帧:
 - `f1` 是最外层调用 `cascade(123)`。
 - `f2` 是第二层调用 `cascade(12)`。
 - `f3` 是最内层调用 `cascade(1)`。
2. 在 `f3` 中:
 - `n = 1`, 符合基本情况 (`n < 10`)。
 - 打印 1。
 - 然后返回 `None`, 这意味着 `f3` 完成了任务, 被销毁。

回溯到上一层 (`f2`)

1. 回到 `f2` 的调用栈 (`n = 12`), 此时 `cascade(n // 10)` 的调用已经完成 (刚刚处理完 `n = 1` 的情况)。
2. 挂起的代码继续执行:

`print(n)` # 这是 `cascade(12)` 递归调用后的部分

`n = 12`, 所以此时打印 12。

为什么会回到 `print(12)`?

这是因为在递归调用时, `cascade(12)` 调用了 `cascade(1)`, 并将控制权暂时交给了 `cascade(1)`。当 `cascade(1)` 执行完毕返回后, `cascade(12)` 会继续执行它的剩余代码:

1. `cascade(1)` 返回后, `cascade(12)` 的递归调用部分完成。
2. 剩下的 `print(n)` 还没有被执行, 所以回溯到这里, 打印 12。

一道很好的打印类题目

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f,g,n):
    if n:
        f(n)
        g(n)

grow = lambda n:f_then_g(grow,print,n//10)
shrink = lambda n:f_then_g(print,shrink,n//10)

inverse_cascade(123)
```

如何训练递归思维

2025年2月8日 21:15

以hw03里的q3为例

递归思维的关键点：

递归的思考方式是：**先相信它能解决子问题，然后自己只负责当前这一步的选择。**

1. **假设递归调用已经正确**（即 `sum_from(k+2)` 能解决 `k+2` 之后的所有问题）。
2. **解决当前这一步**（比如 `odd_func(k) + even_func(k+1)`）。
3. **让递归推进到终止条件**（if `k > n`: 终止）。
4. 如果内部有辅助函数，**先return一个极限最小或最大值**让函数启动起来

counting partition分割问题

2025年2月6日 20:39

Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

```
count_partitions(6, 4)
```

```
2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6
```

这是一个详尽的列表，列出了使用大小不超过4的零件构成6的所有分区的情况。
And this is an exhaustive list of everything that counts as a partition of 6 using parts up to size 4.

Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

```
count_partitions(6, 4)
```

```
2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6
```

count_partitions函数的目的只是告诉我们有多少不同的分区。
The purpose of the count_partitions function is just to tell us how many different partitions there are.

Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

```
count_partitions(6, 4)
```

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
 - Use at least one 4
 - Don't use any 4
- Solve two simpler problems:
 - `count_partitions(2, 4)`
 - `count_partitions(6, 3)`
- Tree recursion often involves exploring different choices.

万岁。
Hooray.

在整数分割问题中，`count_partitions(n, m)` 的含义是：

将整数 n 分割成若干部分，其中每个部分的大小不超过 m 。

在分解过程中，我们将问题拆分为两种情况：

- 选择当前最大值 m ：从 n 中减去 m ，然后继续分割剩余部分。

- 2. 不选择当前最大值 m : 直接把最大值限制减少到 $m - 1$, 继续分割整个 n 。

$\text{count_partitions}(n, m) = \text{count_partitions}(n - m, m) + \text{count_partitions}(n, m - 1)$

- $\text{count_partitions}(n - m, m)$: 表示选择了 m 后分割剩下的部分。
- $\text{count_partitions}(n, m - 1)$: 表示不选择 m , 只使用更小的数分割。

为什么是 (2, 4) 和 (6, 3)?

第一种情况: 选择当前最大值

- 选择一个 $m = 4$:
 - 如果我们至少使用一个 4, 那么剩下的部分是 $6 - 4 = 2$ 。
 - 这意味着, 我们要继续分割 2, 并且分割的最大值仍然可以是 4 (因为 4 可以继续被使用, 只要 n 足够大)。
 - 因此, 递归子问题为 $\text{count_partitions}(2, 4)$ 。

第二种情况: 不选择当前最大值

- 完全不使用 $m = 4$:
 - 如果不使用 4, 那么我们只能使用小于 4 的值, 即最大值限制变为 3。
 - 此时, 我们需要分割完整的 6, 但分割的每一部分大小不能超过 3。
 - 因此, 递归子问题为 $\text{count_partitions}(6, 3)$ 。

为什么不是 (3, 3)?

如果你将第二种情况误解为 (3, 3), 那么你可能错误地认为:

- 在第一种情况下, 从 $n = 6$ 减去 $m = 4$, 得到剩余的 2。
- 在第二种情况下, 你可能以为要直接分割剩下的 $6 - m = 3$, 并且最大值限制变为 3。

Counting Partitions



The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

• Recursive decomposition: finding simpler instances of the problem.

• Explore two possibilities:

• Use at least one 4

• Don't use any 4

• Solve two simpler problems:

• $\text{count_partitions}(2, 4)$ ----->

• $\text{count_partitions}(6, 3)$ ----->

• Tree recursion often involves exploring different choices.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

counting partition从小到大递归

2025年2月9日 12:43

```
count_partitions(n, smallest) =  
count_partitions(n - smallest, smallest) + count_partitions(n, smallest + 1)
```

- **第一项** `count_partitions(n - smallest, smallest)`:
 - 选择 `smallest`, 继续使用它, 减少 `n`。
- **第二项** `count_partitions(n, smallest + 1)`:
 - 不使用 `smallest`, 改用更大的数。

递归终止条件

- `n == 0`: 找到了一种有效组合, 返回 1
- `n < 0`: 无法组成 `n`, 返回 0
- `smallest > n`: 没有更多数可以尝试, 返回 0

disc04-sum_fun

2025年2月10日 22:20

- $n < 0$ 的时候，没有办法用 1 到 m 的数字加出 n ，所以返回 `[]`（表示没有解）。
- $n == 0$ 的时候，唯一可能的方式是用 **空列表** `[]`，因为没有任何数字能加出 0，但空列表的和就是 0。所以返回 `[[]]`（表示一种方式，即空列表）。

1. 递归的最终目标

我们的函数 `sums(n, m)` 返回的是一个“列表的列表”，即 **所有可能的和为 n 的列表**。

`sums(5, 3)` # 返回 `[[1, 3, 1], [2, 1, 2], [2, 3], [3, 2]]`

2. 为什么 `return [[]]`?

当 $n == 0$ 时，唯一的方式是 **不使用任何数字**，即 **空列表** `[]`。但是 **我们的函数返回的是“列表的列表”**，所以我们需要把空列表 `[]` 作为一个可能的解，包裹在外层列表中：

`return [[]]` # 表示“唯一一种方式就是空列表”

举个例子

假设 `sums(0, m)` 直接返回 `[]`，那么在递归过程中，我们的 `rest` 变量就会遍历一个空列表 `[]`，导致没有任何结果被构造出来。

但如果 `return [[]]`，递归会继续正常进行：

```
sums(1, 3) = [[1] + rest for rest in sums(0, 3)]  
            = [[1] + rest for rest in [[]]]  
            = [[1] + []]  
            = [[1]]
```

这样 `sums(1, 3)` 就能正确返回 `[[1]]`。

如果 `sums(0, 3)` 直接返回 `[]`，那么：

```
sums(1, 3) = [[1] + rest for rest in []]  
            = [] # 什么都不会执行！
```

这样就会错误地丢失 `[[1]]` 这个解！

```
result = result + [[k] + rest for rest in sums(n - k, m) if rest == [] or rest[0] != k]  
if rest == [] or rest[0] != k 的作用是：
```

1. **允许** `rest == []` **通过**（因为 `sums(0, m) == [[]]`）。
2. **确保** `rest[0] != k`（防止相邻数字相同）。

换句话说：

- 如果 `rest` 是 `[]`（空列表），那么 `rest[0]` **根本不存在**，所以我们 **不能检查** `rest[0] != k`，否则会出错。
- 如果 `rest` **不是** `[]`，**那就要检查** `rest[0] != k`，确保相邻数字不同。

所以 `or` 相当于做了一个“兜底”：

- 如果 `rest[0] == k` (本来要被剔除)
 - 但 `rest == []`, 那么 这部分仍然会被保留。

这确保了 `sums(0, m) == [[]]` 的结果 能够继续参与递归。

好题

2025年2月14日 19:18

cats minimum_mewtations

lists

2025年2月9日 18:44

<https://www.learncs.site/docs/curriculum-resource/cs61a/lab/lab03#required-questions>

Working with Lists

```
>>> digits = [1, 8, 2, 8]
The number of elements
>>> len(digits)
4
An element selected by its index
>>> digits[3]
8
>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]
>>> getitem(digits, 3)
8
```

如果我想通过索引找到一个元素，我可以使用元素选择语法，
或者我可以使用 operator 模块中的 getitem 函数，它有相同的效果。
If I want to find an element selected by its index, I can use element selection syntax,
or I can use the getitem function in the operator module, which has the same effect.

查找list中一个element

1.使用index

2.使用operator中的getitem

合并复制list

digits = [1,8,2,8]

Concatenation and repetition

```
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
>>> add([2, 7], mul(digits, 2))
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

nested list

Nested lists

```
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```

Built-in operators for testing whether an element appears in a compound value

```
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 8 in digits
True
>>> 5 not in digits
True
>>> not(5 in digits)
True
```

```
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 5 in digits
False
>>> '1' == 1
False
>>> '1' in digits
False
>>> █
```

in 判断不了子序列，只能判断单独的元素

```
>>> [1, 8] in digits
False
>>> [1, 2] in [3, [1, 2], 4]
True
>>> [1, 2] in [3, [[1, 2]], 4]
False
>>> █
```

for

2025年2月9日 19:01

```
def count(s, value):
    """Count the number of times that val
    in sequence s.

    >>> count([1, 2, 1, 2, 1], 1)
    3
    """
    total, index = 0, 0
    while index < len(s):
        element = s[index]
        if element == value:
            total += 1
        index += 1
    return total
```

```
def count(s, value):
    """Count the number of times that val
    in sequence s.

    >>> count([1, 2, 1, 2, 1], 1)
    3
    """
    total = 0
    for element in s:
        if element == value:
            total += 1
    return total
```

For Statement Execution Procedure

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value (a sequence)
2. For each element in that sequence, in order:
 - A. Bind <name> to that element in the current frame

sequence unpack

Sequence Unpacking in For Statements

A sequence of fixed-length sequences

```
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0
```

A name for each element in a fixed-length sequence

Each name is bound to a value, as in multiple assignment

```
>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1
>>> same_count
2
```

这里使用的是 **序列解包** (Sequence Unpacking) `for x, y in pairs:` 语句中的 `x, y` 并不是接收整个子列表 `[1, 2]`、`[2, 2]` 等，而是把子列表中的每个元素逐个拆开并赋给 `x` 和 `y`。

例如：

- 对于子列表 `[1, 2]`，`x` 被赋值为 `1`，`y` 被赋值为 `2`。
- 对于子列表 `[2, 2]`，`x` 被赋值为 `2`，`y` 被赋值为 `2`。
- 对于子列表 `[3, 2]`，`x` 被赋值为 `3`，`y` 被赋值为 `2`。
- 对于子列表 `[4, 4]`，`x` 被赋值为 `4`，`y` 被赋值为 `4`。

你再品

```
pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]
same_count = 0
```

```
for pair1, pair2 in zip(pairs, pairs[1:]):
    x = pair1
    y = pair2
    if x == y:
        same_count = same_count + 1

print(same_count)
```

range

2025年2月9日 19:52

a range is a sequence of consecutive integers

length: ending value - starting value

element selection: starting value + index

range 如果只有一个数字，就是结束值，起始值为0

The Range Type

A range is a sequence of consecutive integers.*

..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

range(-2, 2)

Length: ending value - starting value

Element selection: starting value + index

```
>>> list(range(-2, 2))  
[-2, -1, 0, 1]
```

List constructor

```
>>> list(range(4))  
[0, 1, 2, 3]
```

Range with a 0 starting value

* Ranges can actually represent more general integer sequences.

```
>>> range(5, 8)  
range(5, 8)  
>>> list(range(5, 8))  
[5, 6, 7]  
>>> range(4)  
range(0, 4)  
>>> list(range(4))  
[0, 1, 2, 3]  
>>>
```

```
>>> ^D  
~/lec$ python3 -i ex.py  
>>> sum_below(5)  
10  
>>> ^D  
~/lec$ python3 -i ex.py  
>>> cheer()  
Go Bears!  
Go Bears!  
Go Bears!  
>>>
```

```
def sum_below(n):  
    total = 0  
    for i in range(n):  
        total += i  
    return total  
  
def cheer():  
    for i in range(3):  
        print('Go Bears!')
```

list comprehensions

2025年2月9日 20:01

[<expression> for <element> in <sequence>]

[<expression> for <element> in <sequence> if <conditional>]

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o', 'p']
>>> [letters[i] for i in [3, 4, 6, 8]]

['d', 'e', 'm', 'o']
```

```
>>> odds = [1, 3, 5, 7, 9]
>>> [x+1 for x in odds]
[2, 4, 6, 8, 10]
>>> [x for x in odds if 25 % x == 0]
[1, 5]
>>> [x+1 for x in odds if 25 % x == 0]
[2, 6]
>>> []
```

if 25 % x == 0 表示要筛选出能整除 25 的那些 x 值。

对于筛选出的每个 x, 计算 x+1

如何得到因数列表

```
def divisors(n):
    return [1] + [x for x in range(2, n) if n%x==0]
```

见cats-fastest_words

box and pointer notation

2025年2月10日 15:24

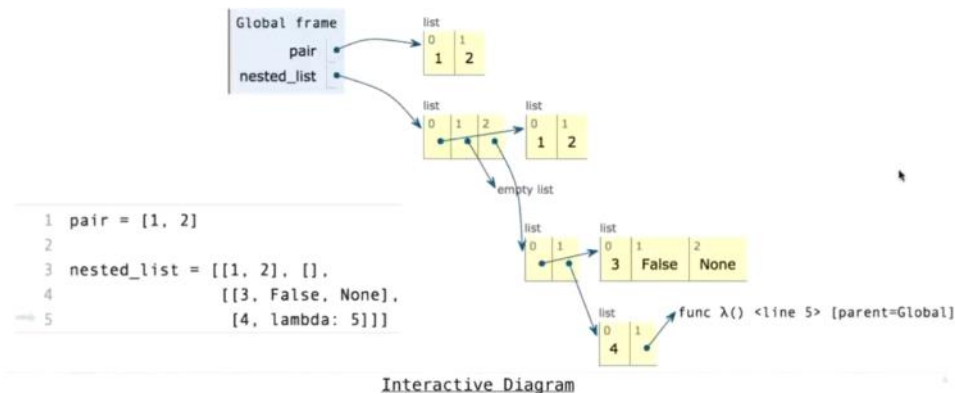
The Closure Property of Data Types

- A method for combining data values satisfies the closure property if: The result of combination can itself be combined using the same method
- Closure is powerful because it permits us to create hierarchical structures
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

Lists can contain lists as elements (in addition to anything else)

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element.
Each box either contains a primitive value or points to a compound value



slicing

2025年2月10日 15:32

切片是一种你可以对列表和范围等列执行的操作。

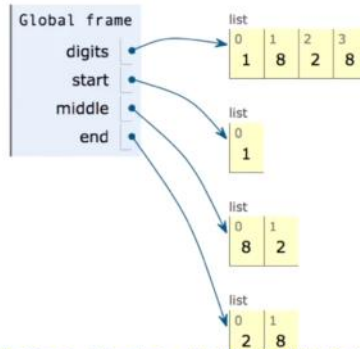
Slicing is an operation that you can perform on sequences such as lists and ranges.

```
>>> odds = [3, 5, 7, 9, 11]
>>> list(range(1, 3))
[1, 2]
>>> [odds[i] for i in range(1, 3)]
[5, 7]
>>> odds[1:3]
[5, 7]
>>> odds[:3]
[3, 5, 7]
>>> odds[1:]
[5, 7, 9, 11]
>>> odds[:]
[3, 5, 7, 9, 11]
>>>
```

slicing always creates new values

Slicing Creates New Values

```
1 digits = [1, 8, 2, 8]
2 start = digits[:1]
3 middle = digits[1:3]
4 end = digits[2:]
```



但是当我切出1时，说我想从开头一直到但不包括索引1，我得到了一个新的列表，其中包含值1。

But when I sliced the 1 out, saying that I want to go from the beginning all the way up but not including index 1, I got a new list with the value 1 in it.

Interactive Diagram

sum

2025年2月10日 15:50

• `sum(iterable[, start]) -> value`

Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.

- **sum(iterable, start=0)**

- 第一个参数 `iterable` 必须是一个可迭代对象（如列表、元组等）。
- 第二个参数 `start` 是可选的，表示累加的起始值，默认为 0。

- **sum(0, [2, 3]) 报错，0 不是一个可迭代对象**

- **sum([2, 3], 0) 正确，得到 5**

- **sum([2, 3, 4])**

- **结果：9**

- 原因：sum 函数可以计算列表中的数值之和，因此 [2, 3, 4] 的元素相加后为 9。

- **sum(['2', '3', '4'])**

- **错误：TypeError: unsupported operand type(s) for +: 'int' and 'str'**

- 原因：sum 函数的默认行为是累加数值。这里列表中的元素是字符串，而 sum 的起始值默认为整数 0，尝试将 0 和 '2'（字符串）相加时会报错。

- **sum([2, 3, 4], 5)**

- **结果：14**

- 原因：5 和 2, 3, 4 的类型相同，都是整数，所以加法可以正常执行。

- **[2, 3] + [4]**

- **结果：[2, 3, 4]**

- 原因：列表之间的 + 运算会将它们拼接在一起，而不是求和。因此，这里直接得到一个新列表 [2, 3, 4]。

- **sum([[2, 3], [4]], [])**

- **结果：[2, 3, 4]**

- 原因：当 sum 的第二个参数设置为 [] 时，sum 会将 [[2, 3], [4]] 中的列表依次拼接，结果为 [2, 3, 4]。

- **0 + [2, 3]**

- **错误：TypeError: unsupported operand type(s) for +: 'int' and 'list'**

- 原因：数字和列表无法直接用 + 相加，类型不匹配，因此报错。

- **sum([[2, 3], [4]])**

- **错误：TypeError: unsupported operand type(s) for +: 'int' and 'list'**

- 原因：

这里的 iterable 是 [[2, 3], [4]]，其中的元素是 列表类型，默认起始值是 0（整数），因此会尝试执行 0 + [2, 3]，但整数和列表无法直接相加。如果想让 sum([[2, 3], [4]]) 起作用，可以显式指定起始值为一个空列表 []

max

2025年2月10日 16:02

- `max(iterable[, key=func]) -> value`
`max(a, b, c, ...[, key=func]) -> value`

With a single iterable argument, return its largest item.
With two or more arguments, return the largest argument.

```
>>> max(range(5))
4
>>> max(0, 1, 2, 3, 4)
4
>>> max(range(10), key=lambda x: 7-(x-4)*(x-2))
3
>>> (lambda x: 7-(x-4)*(x-2)) ( 3 )
8
>>> (lambda x: 7-(x-4)*(x-2)) ( 2 )
7
>>> (lambda x: 7-(x-4)*(x-2)) ( 4 )
7
>>> (lambda x: 7-(x-4)*(x-2)) ( 5 )
4
>>> (lambda x: 7-(x-4)*(x-2)) ( 5 )
```

见cat -fastest_words

all

2025年2月10日 16:49

• `all(iterable) -> bool`

Return True if `bool(x)` is True for all values `x` in the iterable.
If the iterable is empty, return True.

```
>>> bool(5)
True
>>> bool(True)
True
>>> bool(-1)
True
>>> bool(0)
False
>>> █
```

```
>>> bool('hello')
True
>>> bool('')
False
>>> █
```

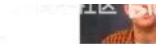
```
>>> range(5)
range(0, 5)
>>> [x < 5 for x in range(5)]
[True, True, True, True, True]
>>> all([x < 5 for x in range(5)])
True
>>> all(range(5))
False
>>> █
```

`[x < 5 for x in range(5)]` 中, `<expression>` 是 `x < 5`, 它返回布尔值 `True` 或 `False`, 而不是直接返回 `x` 的值。

strings are sequences

2025年2月10日 17:31

Strings are Sequences



Length and element selection are similar to all sequences

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

Careful: An element of a string is itself a string, but with only one element!

However, the "in" and "not in" operators match substrings

```
>>> 'here' in "Where's Waldo?"
True
>>> 234 in [1, 2, 3, 4, 5]
False
>>> [2, 3, 4] in [1, 2, 3, 4, 5]
False
```

When working with strings, we usually care about whole words more than letters

dictionary

2025年2月10日 17:38

```
>>> numerals = {'I': 1, 'V': 5, 'X': 10}
>>> numerals
{'I': 1, 'V': 5, 'X': 10}
>>> numerals['X']
10
>>> numerals[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
>>> numerals['V']
5
>>> numerals['X']
10
>>> numerals['X-ray']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'X-ray'
>>> numerals[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 5
>>> █
```

dictionary单向查找，只能通过key找value

```
>>> list(numerals)
['I', 'V', 'X']
>>> numerals.values()
dict_values([1, 5, 10])
>>> sum(numerals.values())
16
>>> list(numerals.values())
[1, 5, 10]      T      Much
>>> █
```

```
>>> {1: 'item'}
{1: 'item'}
>>> {1: ['first', 'second'], 3: 'third'}
{1: ['first', 'second'], 3: 'third'}
dictionary value could be a list
>>> d = {1: ['first', 'second'], 3: 'third'}
>>> d[1]
['first', 'second']
>>> d[3]
'third'
>>> len(d)
2
>>> len(d[1])
2
>>> len(d[3])
5      T
>>> █
```

two restrictions to the key

1. can't repeat a key

```
>>> {1: 'first', 1: 'second'}
{1: 'second'}
>>> █
```

2.key itself cannot be a list or a dictionary

```
>>> {[1]: 'first'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> {{}: 'zero'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
>>> █
```

value 没有限制, dictionary也可以作为value

dictionary comprehensions

2025年2月10日 19:24

Dictionary Comprehensions



```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

Short version: {<key exp>: <value exp> for <name> in <iter exp>}

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
 2. Create an empty *result dictionary* that is the value of the expression
 3. For each element in the iterable value of `<iter exp>`:
 - A. Bind `<name>` to that element in the new frame from step 1
 - B. If `<filter exp>` evaluates to a true value, then add to the result dictionary an entry that pairs the value of `<key exp>` to the value of `<value exp>`
- `{x * x: x for x in [1, 2, 3, 4, 5] if x > 2}` evaluates to `{9: 3, 16: 4, 25: 5}`

练习见lab04 -divide

仔细品:

```
def divide(numbers, range_list):
    # 返回的字典应该是:
    # {
    #     2: [10, 20, 30],
    #     3: [15, 30],
    #     5: [10, 15, 20, 25, 30]
    # }
    return {x:[y for y in range_list if y % x == 0] for x in
numbers if x > 2}

print(divide([2, 3, 5], [10, 15, 20, 25, 30]))
```

所以公式

`{<key exp>: <value exp> for <key_name> in <key_iter exp> if <key_filter exp>}`

除了<value exp>外都是针对key的

data abstraction-rational numbers

2025年2月11日 11:39

Rational Numbers

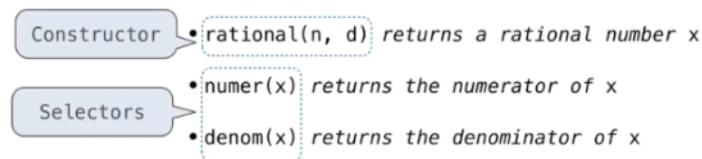
$$\frac{\text{numerator}}{\text{denominator}}$$

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation may be lost!

Assume we can compose and decompose rational numbers:



Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

Example

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$
$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

General Form

Rational Number Arithmetic Implementation

```
def mul_rational(x, y):  
    return rational(numer(x) * numer(y),  
                    denom(x) * denom(y))  
  
def add_rational(x, y):  
    nx, dx = numer(x), denom(x)  
    ny, dy = numer(y), denom(y)  
    return rational(nx * dy + ny * dx, dx * dy)  
  
def equal_rational(x, y):  
    return numer(x) * denom(y) == numer(y) * denom(x)
```

Constructor

Selectors

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

- `rational(n, d)` returns a rational number `x`
- `numer(x)` returns the numerator of `x`
- `denom(x)` returns the denominator of `x`

These functions implement an abstract data type for rational numbers

```
~/lec$ python3 -i ex.py
>>> x, y = rational(1, 2), rational(3, 8)
>>> print_rational(mul_rational(x, y))
3 / 16
>>> x
<function rational.<locals>.select at 0x10293e6a8>
>>>
```

```
return rational(nx * dy + ny * dx, dCS自学网 哔哩哔哩

def mul_rational(x, y):
    """Multiply rational numbers x and y"""
    return rational(number(x) * number(y),
                    rational_are_equal(x, y):
    """Return whether rational numbers x and y are equal."""
    return number(x) * denom(y) == number(y) * denom(x)

def print_rational(x):
    """Print rational x."""
    print(number(x), "/", denom(x))

# Constructor and selectors

def rational(n, d):
    """Construct a rational number x that represents n/d."""
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select

def number(x):
    """Return the numerator of rational number x."""
    return x('n')

def denom(x):
    """Return the denominator of rational number x."""
    return x('d')
```

Rational Data Abstraction Implemented as Functions

```
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select
```

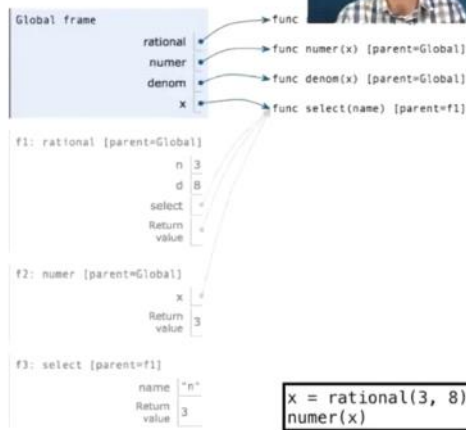
This function represents a rational number

Constructor is a higher-order function

```
def number(x):
    return x('n')
```

```
def denom(x):
    return x('d')
```

Selector calls the object itself



Interactive Diagram

abstraction barriers

2025年2月11日 15:31

这些分隔非常重要，因为它们允许您对程序的一部分进行更改，而其他部分可以利用这些更改，而无需以任何方式中断或创建不一致。

These separations are important because they allow you to make changes to one part of your program and have other parts take advantage of those changes without breaking in any way or creating inconsistencies

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational

Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
---	-----------------------------	------------------------

这个屏障表示使用有理数进行计算的任何东西应该只能使用这些函数，并且不应该使用不同层次的函数。
That barrier says anything that's using rational numbers to perform computation should only do it in terms of these functions and should not be using functions of a different layer.

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational

Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
---	-----------------------------	------------------------

Implement selectors and constructor for rationals	two-element lists	list literals and element selection
---	-------------------	-------------------------------------

有一些东西的细节甚至比我们现在看到的还要低。
There is stuff that's even lower detail than what we've seen so far.

Violating Abstraction Barriers

```
add_rational( [1, 2], [1, 4] )
```

Does not use constructors

Twice!

```
def divide_rational(x, y):  
    return [ x[0] * y[1], x[1] * y[0] ]
```

No selectors!

And no constructor!

如果我看到别人电脑上有这段代码，我想我真的会点火，看着它燃烧。
If I saw somebody's computer with this code on it, I think I would honestly just light it on fire and just watch it burn.

reducing to lowest terms

2025年2月11日 17:10

一对由两个值组成，以一种方式捆绑在一起，使你可以将它们视为一个单元，一个整体，尽管有两个部分。

A pair consists of two values that are joined together, bundled together in such away that you can treat them as a unit, as a whole, even though there are two parts

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2

>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

A list literal:
Comma-separated expressions in brackets

"Unpacking" a list

Element selection using the selection operator

Element selection function

Representing Rational Numbers

```
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]

def numer(x):
    """Return the numerator of rational number X."""
    return x[0]

def denom(x):
    """Return the denominator of rational number X."""
    return x[1]
```

Construct a list

Select item from a list

为了实现我们想要的，也就是选择有理数的分母，我们将在这里从列表选择一个项目。
We're going to select an item from a list here in order to implement what we want, which is selecting the denominator of a rational number.

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \frac{5}{2}$$
$$\frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$
$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2}$$
$$\frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from fractions import gcd

def rational(n, d):
    """Construct a rational number x that represents n/d."""
    g = gcd(n, d)
    return [n//g, d//g]
```

Greatest common divisor