

HOF

Nested Functions

```
def maxpower(n,exp):  
    def count(m,cnt):  
        if m >= exp:  
            return cnt  
        return count(m*n, 1+cnt)  
    return count(1,0)
```

```
>>> maxpower(2,1000)
```

```
10
```

```
>>> maxpower(3,1000)
```

```
7
```

Nested Functions

```
def maxpower(n,exp):  
    def count(m,cnt):  
        if m >= exp:  
            return cnt  
        return count(m*n, 1+cnt)  
    return count(1,0)
```

Which variables live in the scope of **count** but not a local variable of **count**?

n **exp**

Nested Functions

```
def partial(op):  
    def action(a,b):  
        print(op(a,b))  
    return action  
f1 = partial(lambda x,y : x*2 + y*2)
```

```
>>> f1(5,10)
```

```
30
```

```
>>> f2 = partial(f1)
```

```
>>> f2(5,10)
```

```
30
```

```
None
```

Nested Functions

```
def partial(op):  
    def action(a,b):  
        print(op(a,b))  
    return action  
f1 = partial(lambda x,y : x*2 + y*2)
```

Which variables live in the scope of **action** but not local in **action**?

op

Lambda Expressions

A function definition: `lambda a,b,c : a + b`

Takes in 3 arguments Evaluates to a number
:

A function application:

`(lambda a,b,c : a + b)(1,2,3)`

➤ $(a+b)$ $\{\{a \rightarrow 1, b \rightarrow 2, c \rightarrow 3\}\}$

➤ $(1+2)$

➤ 3

A closure containing
binding environment,
An abstract concept

Lambda Expressions

A function definition:

lambda a, b, c : $a + b$

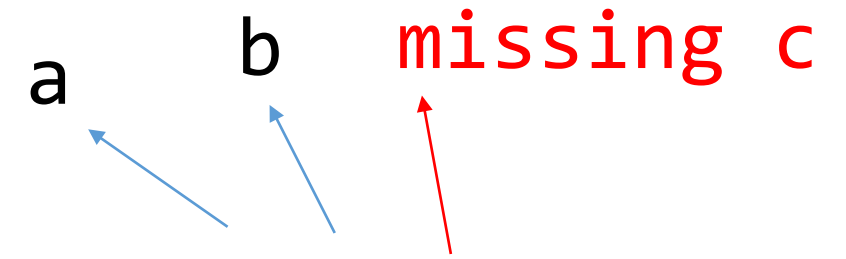
keyword parameters body

Similar to:

def $f(a, b, c)$:
 $\text{return } a + b$

keyword parameters body

Lambda Expressions


>>> (lambda a,b,c : a + b + c)(1,2)

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

(lambda a,b,c : a + b + c)(1,2)

TypeError: <lambda>() missing 1 required positional
argument: 'c'

Lambda Expressions

(**lambda** a : **lambda** b : a + b)

Takes one argument, a

Returns a function (which takes one argument, b; and returns the result a + b)

(**lambda** a : **lambda** b : **lambda** c : a + b + c)

Takes one argument, a

Returns a function (which takes one argument, b; and returns another function which takes one argument, c, and returns the result a + b + c).

Lambda Expressions

`(lambda a : lambda b : a + b)(1)`

➤ `(lambda b : a + b) {{ a→1 }}`

```
>>> x = (lambda a : lambda b : a + b)(1)
```

```
>>> x
```

```
<function <lambda>.<locals>.<lambda> at 0x016A8C90>
```

```
>>> x(3)
```

```
4
```

```
>>>
```

`(lambda b : a + b {{ a→1 }})(3)`

X is assigned to the return value of calling this function, and this return value is itself a function:

`(lambda b : a + b) {{ a→1 }}`

Lambda Expressions

$(\text{lambda } a : \text{lambda } b : a + b)(1)$

➤ $(\text{lambda } b : a + b) \{\{ a \rightarrow 1 \}\}$

$(\text{lambda } a : \text{lambda } b : a + b)(1)(2)$

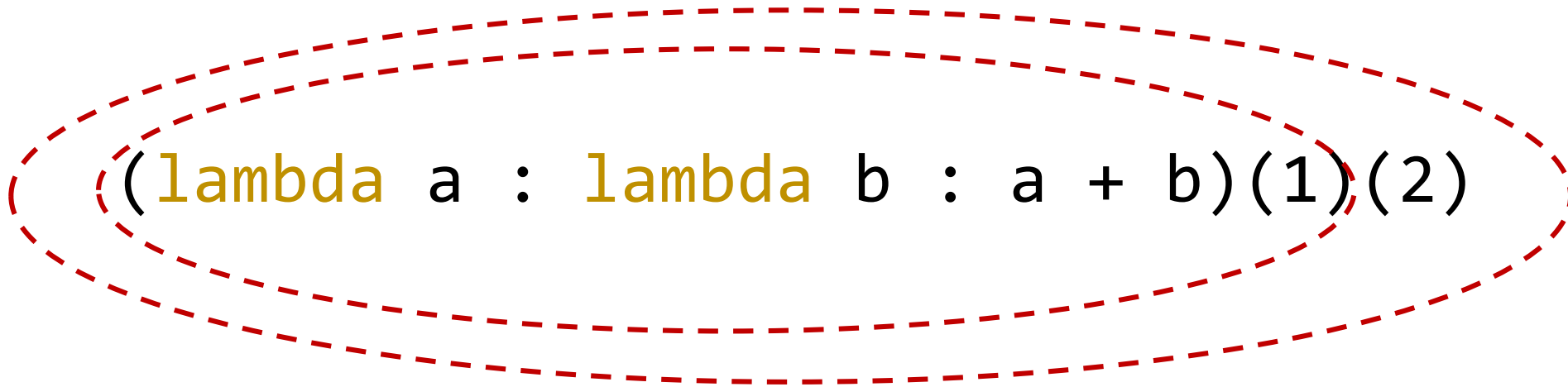
➤ $(\text{lambda } b : a + b \{\{ a \rightarrow 1 \}\})(2)$

➤ $a + b \{\{ b \rightarrow 2, a \rightarrow 1 \}\}$

➤ $1 + 2$

➤ 3

Lambda Expressions



The diagram illustrates the left-associative nature of function application in lambda calculus. It features the expression `(lambda a : lambda b : a + b)(1)(2)` in a monospace font, with the words `lambda` highlighted in yellow. Two concentric dashed red ellipses are drawn around the expression. The inner ellipse encloses the function definition `(lambda a : lambda b : a + b)` and the first argument `(1)`. The outer ellipse encloses the entire expression, including the second argument `(2)`. This visualizes the evaluation order where the function is first applied to `(1)`, and the result of that application is then applied to `(2)`.

```
(lambda a : lambda b : a + b)(1)(2)
```

Consecutive applications of function is treated as left associative.

Lambda Expressions

$(\text{lambda } x: x \ (\text{lambda } y: y))(\text{lambda } z : z)(1)$

- $(x \ (\text{lambda } y: y) \ \{\{ x \rightarrow \text{lambda } z : z \}\}) \ (1)$
- $((\text{lambda } z : z) \ (\text{lambda } y: y)) \ (1)$
- $(z \ \{\{ z \rightarrow \text{lambda } y : y \}\}) \ (1)$
- $(\text{lambda } y : y) \ (1)$
- $y \ \{\{ y \rightarrow 1 \}\}$
- 1