Higher-Order Functions

Announcements

Evaluating Boolean Expressions

False values in Python: False, 0, '', None *(more to come)*

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>**.

To evaluate the expression **not <exp>**:
1. Evaluate the subexpression **<exp>**.
2. The value is True if the result is a false value, and False otherwise

Python stops evaluating it when it knows the answer ("short circuiting"), and returns the value of the most recent subexpression it evaluated

(Demo)

Designing Functions

Describing Functions

A function's *domain* is the set of all inputs it might possibly take as arguments.

A function's *range* is the set of output values it might possibly return.

A pure function's *behavior* is the relationship it creates between input and output.

```
def square(x):
    """Return x * x."""
```

x is a number

square returns a non-negative real number

square returns the square of x

A Guide to Designing Functions

Give each function exactly one job, but make it apply to many related situations

```
>>> round(1.23)
1
```

```
>>> round(1.23, 1)
1.2
```

```
>>> round(1.23, 0)
1
```

```
>>> round(1.23, 5)
1.23
```

Don’t repeat yourself (DRY): Implement a process just once, but execute it many times

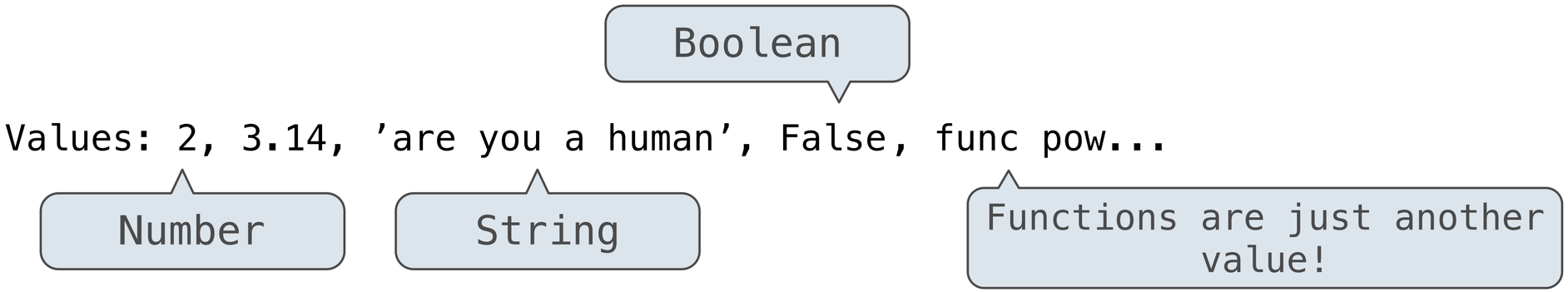
(Demo)

Higher-Order Functions

Values and Expressions

Expressions describe a computation, evaluate to a value

Call Expression: f()



Summation Example

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument (*not called "term"*)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

0 + 1 + 8 + 27 + 64 + 125

The function bound to term gets called here

Evaluation Procedure Cheat Sheet

Call expressions: **Parens**, e.g., f(x)

1. Evaluate the operator (function)
2. Evaluate the operands, from left to right (arguments)
3. Apply the function to the operands

Calling user defined functions:

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

Assignment: **= sign**, e.g., x = 1 + 2

The expression (right) is evaluated, and its value is assigned to the name (left)

Boolean expressions: **and**, **or**, **not** (see earlier slide for rule)

Summation Example, slowly!

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

```
summation(5, cube)
```

Summation Example, slowly!

```
summation(5, cube)
```

Call expressions: **Parens**, e.g., f(x)

1. Evaluate the operator (function)
2. Evaluate the operands, from left to right (arguments)
3. Apply the function to the operands

Calling user defined functions:

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

Assignment: **= sign**, e.g., x = 1 + 2

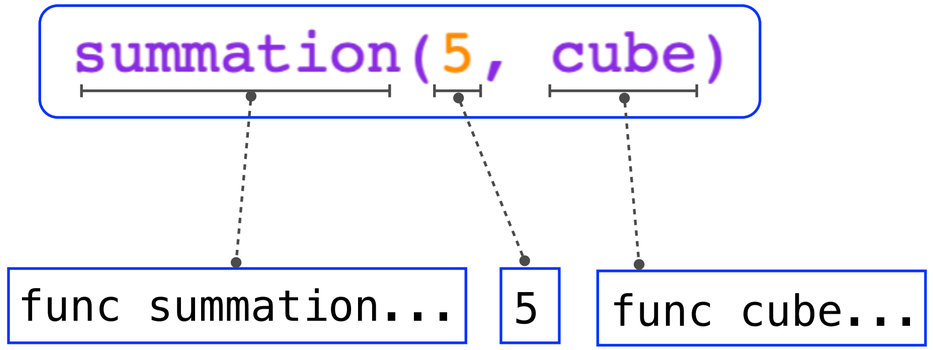
The expression (right) is evaluated, and its value is assigned to the name (left)

Boolean expressions: **and**, **or**, **not** (see earlier slide for rule)

Summation Example, slowly!

Call expressions: **Parens**, e.g., f(x)

1. Evaluate the operator (function)
2. Evaluate the operands, from left to right (arguments)
3. Apply the function to the operands

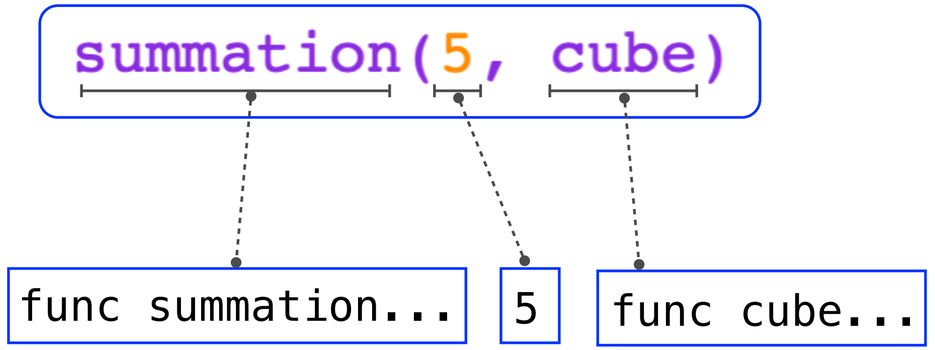



```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

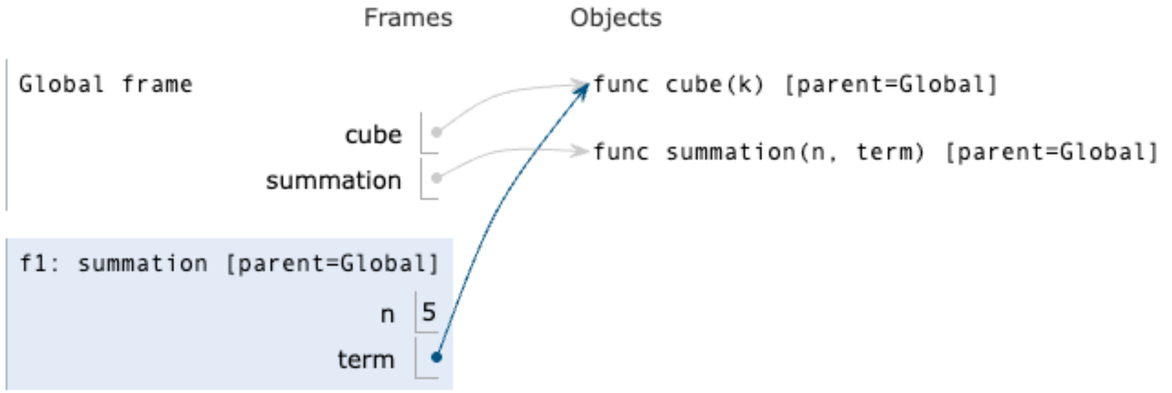
Summation Example, slowly!

Calling user defined functions:

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment



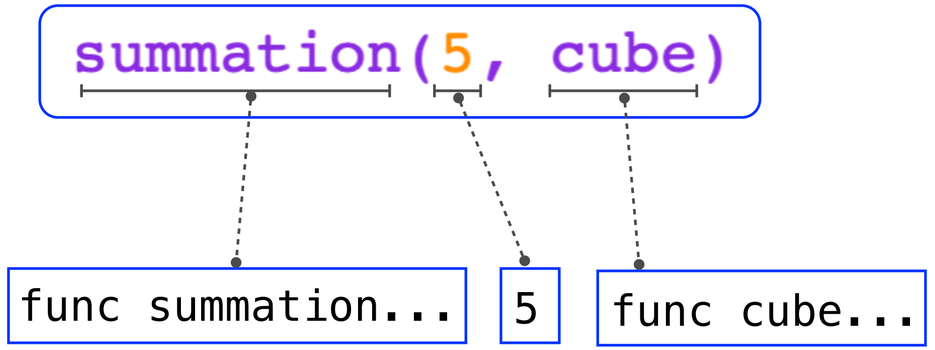

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```



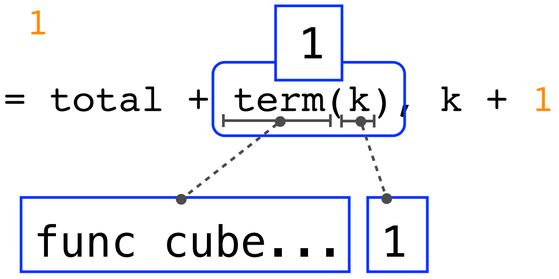
Summation Example, slowly!

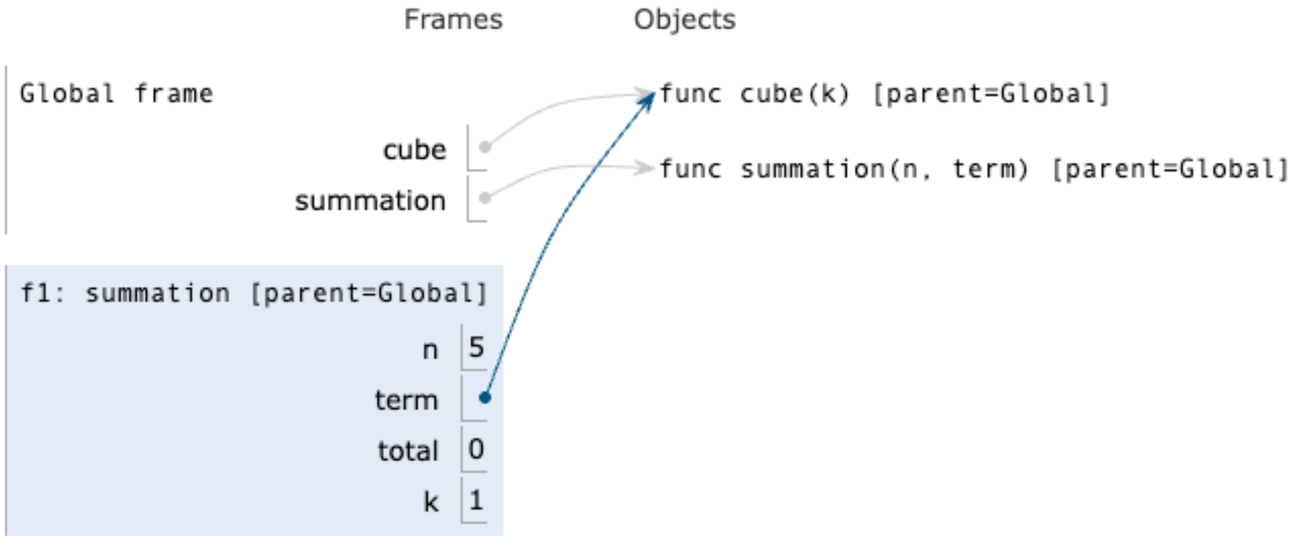
Call expressions: Parens, e.g., f(x)

1. Evaluate the operator (function)
2. Evaluate the operands, from left to right (arguments)
3. Apply the function to the operands




```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```



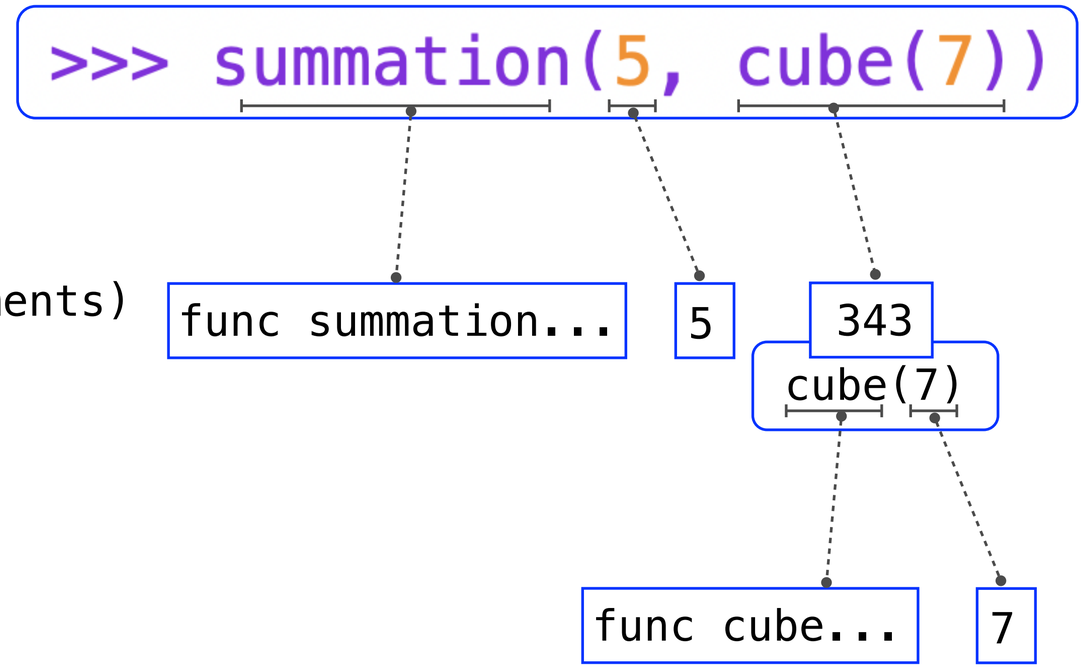


Summation Example, slowly!

summation(5, cube)

Call expressions: Parens, e.g., f(x)

1. Evaluate the operator (function)
2. Evaluate the operands, from left to right (arguments)
3. Apply the function to the operands



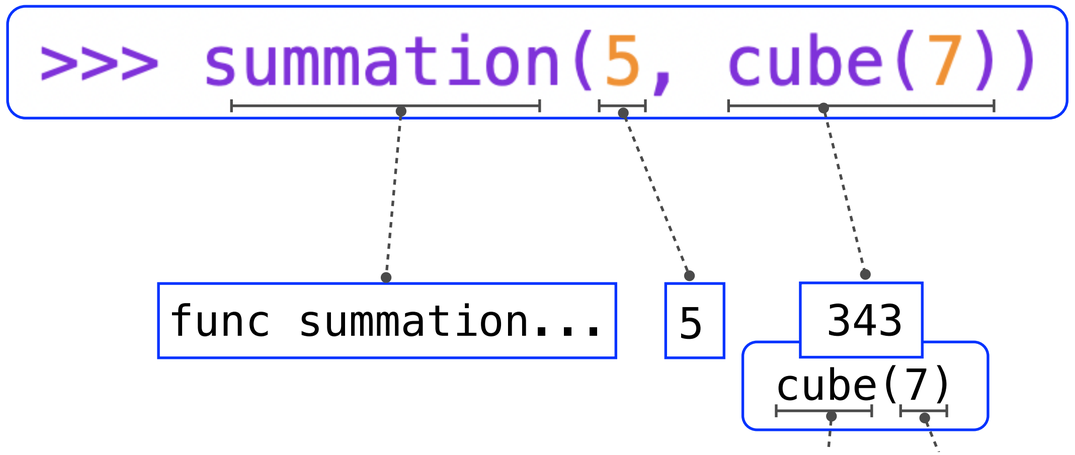

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Summation Example, slowly!

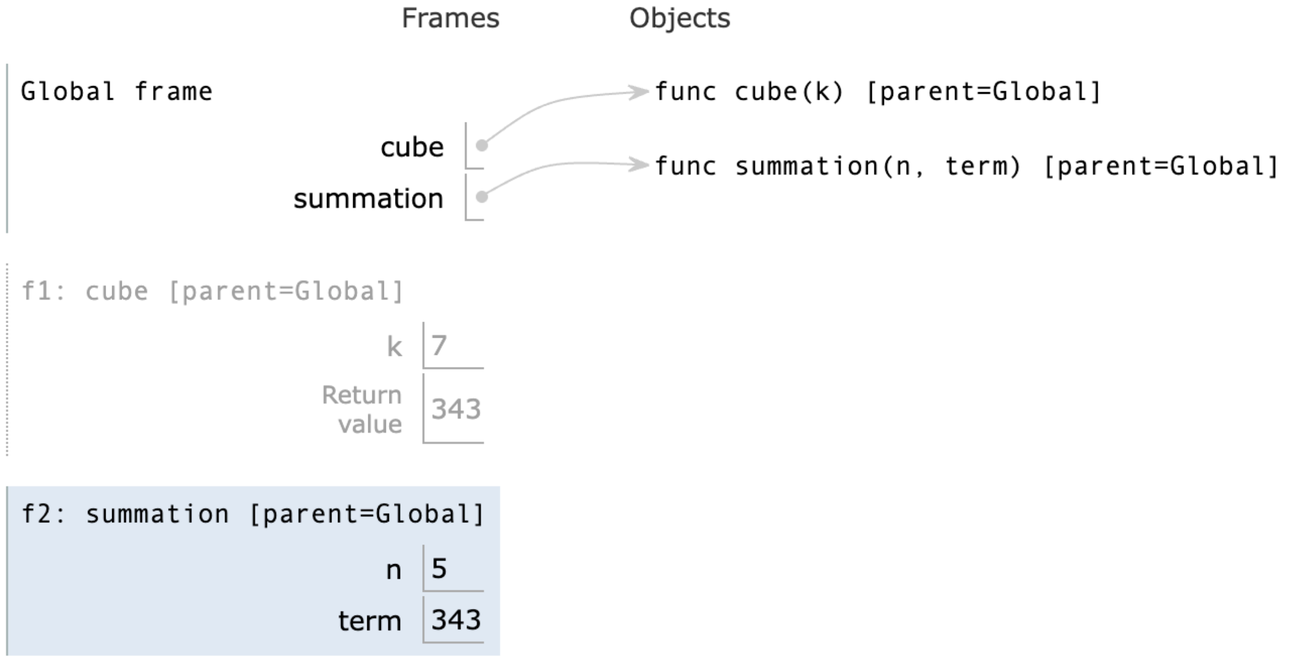
~~summation(5, cube)~~

Calling user defined functions:

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment




```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```



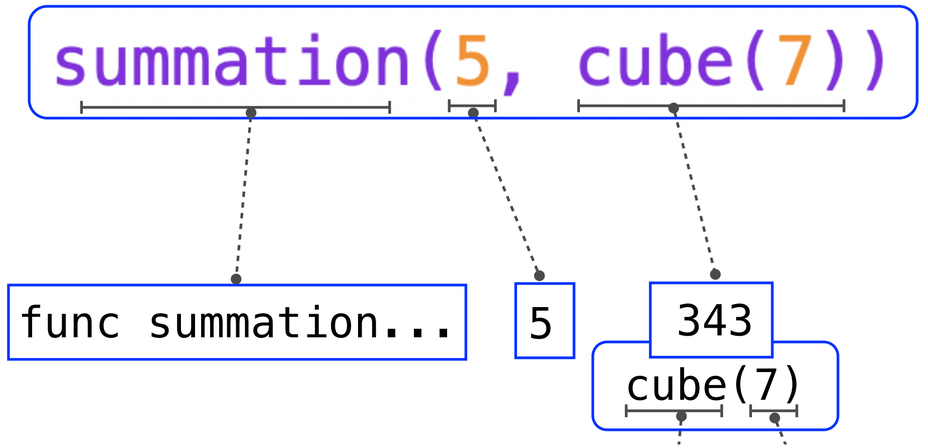
Summation Example, slowly!

~~summation(5, cube)~~

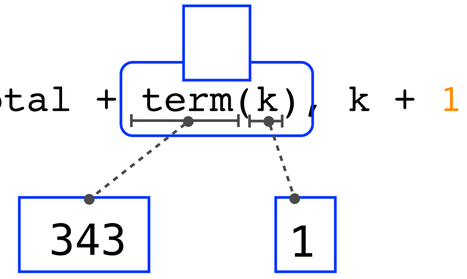
summation(5, cube(7))

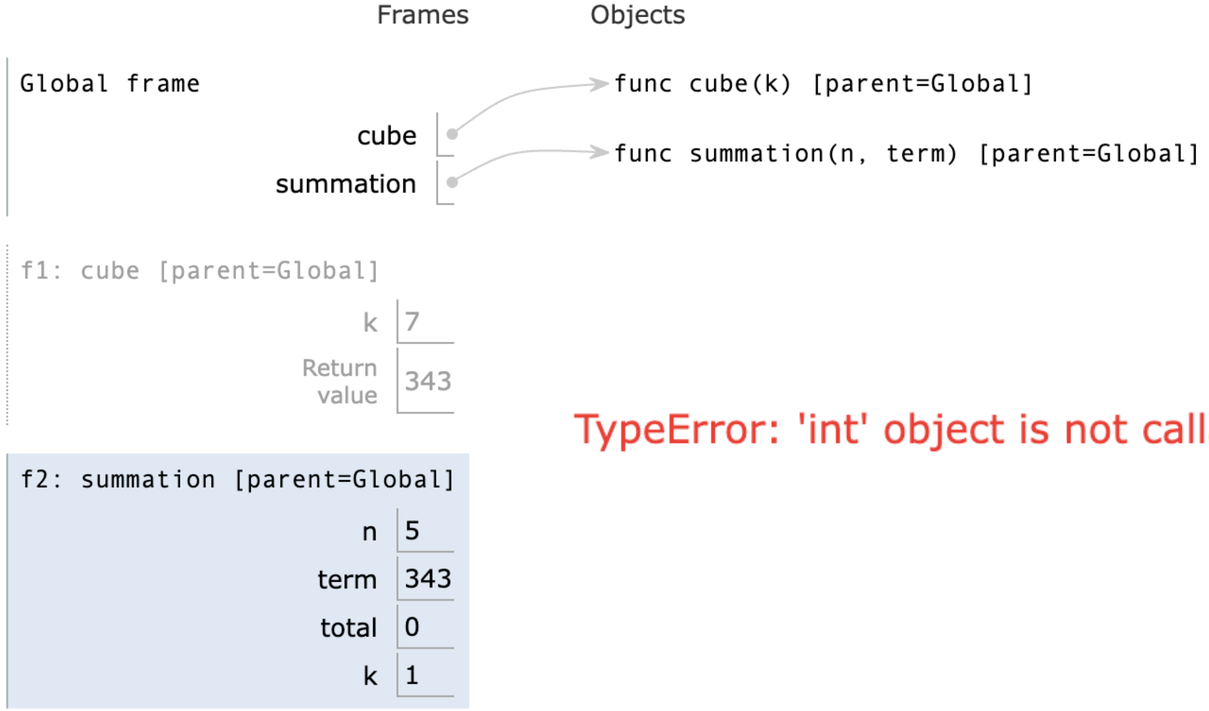
Call expressions: Parens, e.g., f(x)

1. Evaluate the operator (function)
2. Evaluate the operands, from left to right (arguments)
3. Apply the function to the operands




```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```





TypeError: 'int' object is not callable

Program Design

Modularity

Abstraction

Separation of Concerns

Modularity, Abstraction, and Separation of Concerns

I put it in the mailbox

Postal worker collects it, brings it to a distribution center

Truck carries letter from distribution center to airport

Letter goes on an airplane from SFO to Columbia Metro Airport

Truck carriers letter from airport to Columbia post office

Postal worker drives letter to Grandma's house

Abstraction: I only provide the address

Modularity: Pilot pilots the airplane

Separation of Concerns: Postal worker does not know how to pilot airplane

Modularity, Abstraction, and Separation of Concerns

```
curl https://cs61a.org
```

Abstraction: Go get the data at this address

Modularity: How do we break this into different pieces?

Where is the computer located that has this data?

How do we get there?

How do we send this text over a wire?

How do we share that wire with other users?

Twenty-One Rules

Two players alternate turns, on which they can add 1, 2, or 3 to the current total

The total starts at 0

The game end whenever the total is 21 or more

The last player to add to the total loses

(Demo)

Functions within other functions

(Demo)