

Higher-Order Functions

Announcements

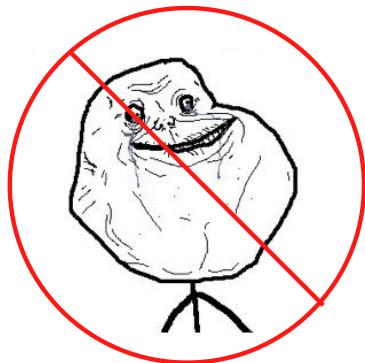
Office Hours: You Should Go!

Office Hours: You Should Go!

You are not alone!

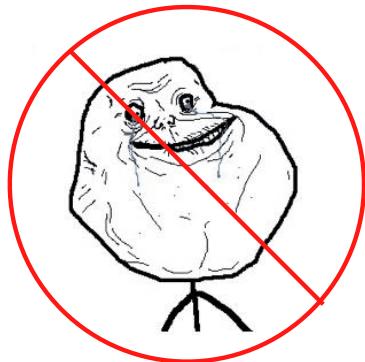
Office Hours: You Should Go!

You are not alone!



Office Hours: You Should Go!

You are not alone!



<http://cs61a.org/office-hours.html>

Iteration Example

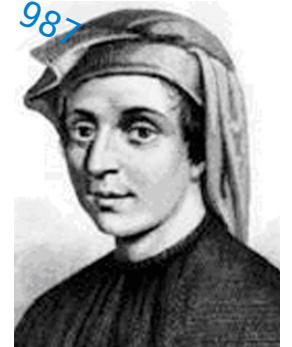
The Fibonacci Sequence

The Fibonacci Sequence



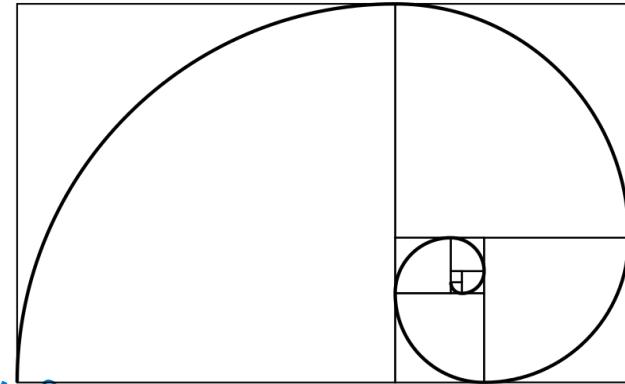
The Fibonacci Sequence

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



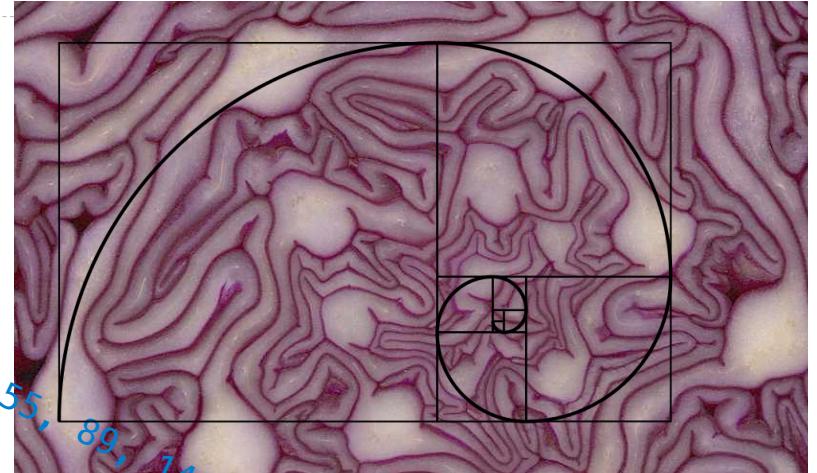
The Fibonacci Sequence

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



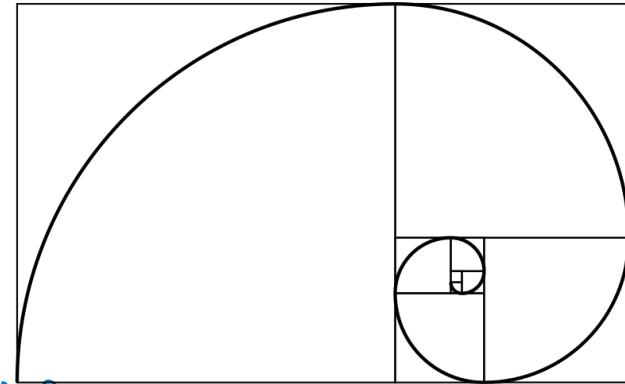
The Fibonacci Sequence

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



The Fibonacci Sequence

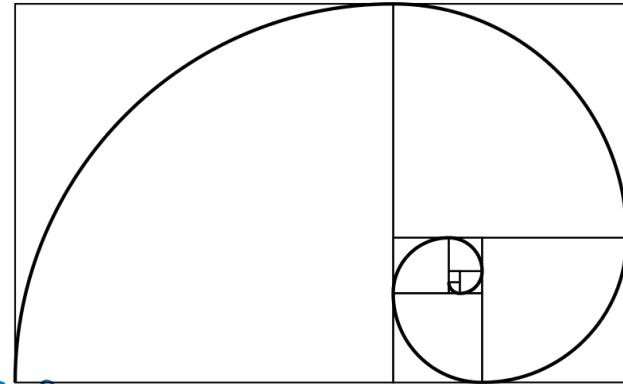
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987



The Fibonacci Sequence

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

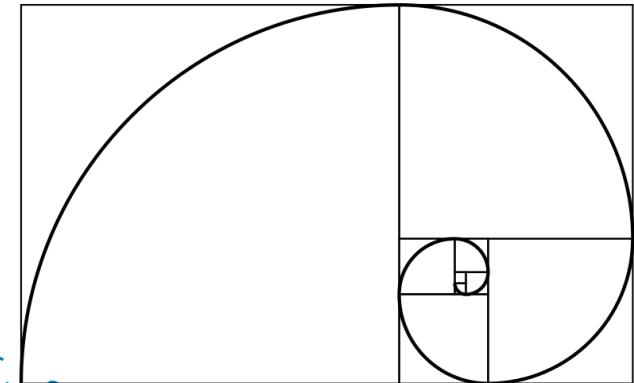
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987



The Fibonacci Sequence

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

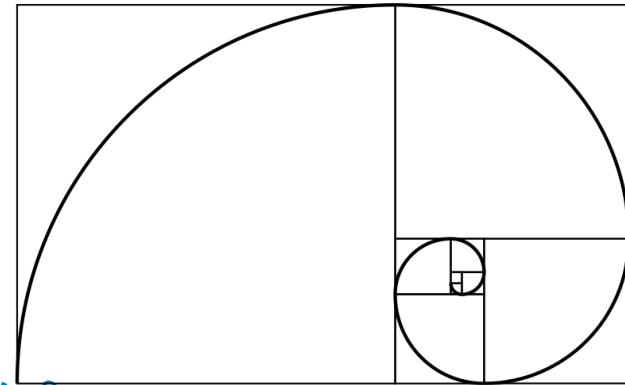
The next Fibonacci number is the sum of
the current one and its predecessor



The Fibonacci Sequence

fib	pred	[]
	curr	[]
	n	[5]
	k	[]

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



```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

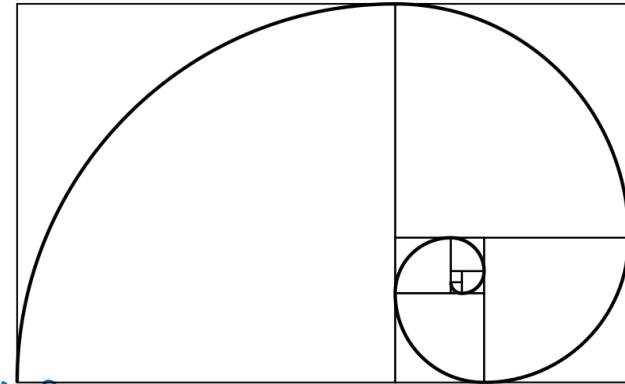
The next Fibonacci number is the sum of
the current one and its predecessor



The Fibonacci Sequence

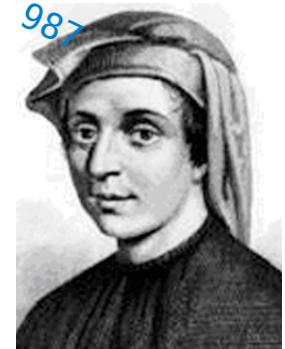
fib	pred	[]
	curr	[]
	n	[5]
	k	[1]

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



```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

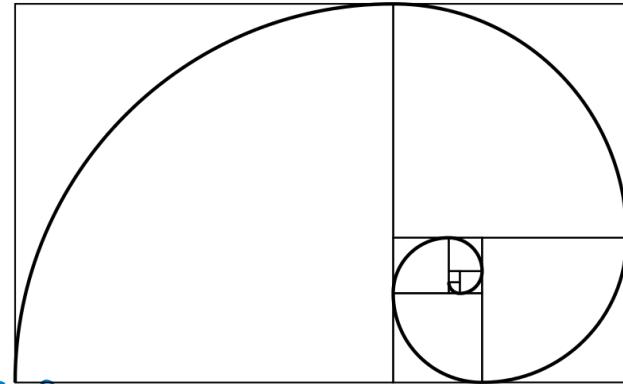
The next Fibonacci number is the sum of
the current one and its predecessor



The Fibonacci Sequence

fib	pred	[]
	curr	[]
	n	[5]
	k	[1]

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



```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of
the current one and its predecessor



The Fibonacci Sequence

fib	pred	[]
	curr	[]
	n	[5]
	k	[2]

0,

1,

1,

2,

3,

5,

8,

13,

21,

34,

55,

89,

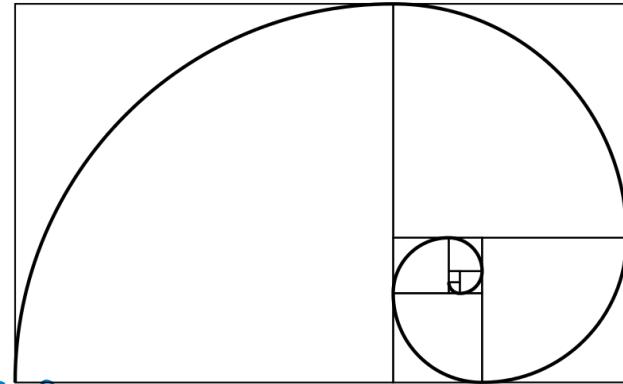
144,

233,

377,

610,

987,



```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of
the current one and its predecessor



The Fibonacci Sequence

fib	pred	[]
	curr	[]
	n	[5]
	k	[3]

0,

1,

1,

2,

3,

5,

8,

13,

21,

34,

55,

89,

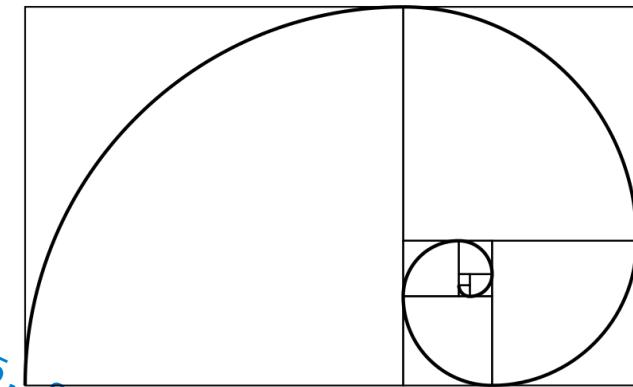
144,

233,

377,

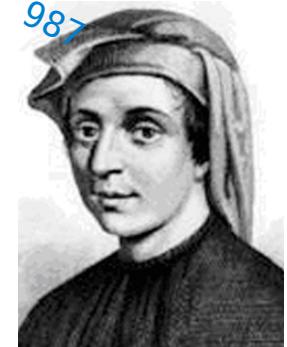
610,

987,



```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

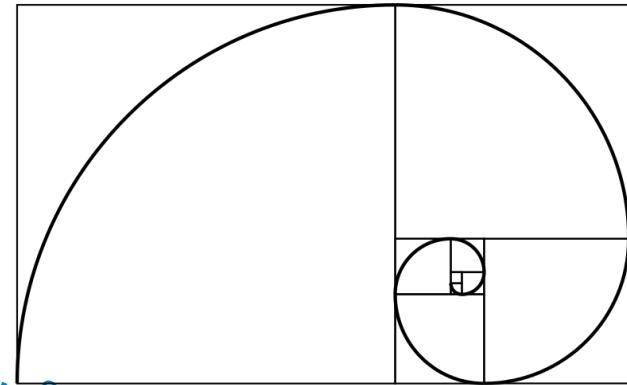
The next Fibonacci number is the sum of
the current one and its predecessor



The Fibonacci Sequence

fib	pred	[]
	curr	[]
	n	[5]
	k	[4]

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



```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

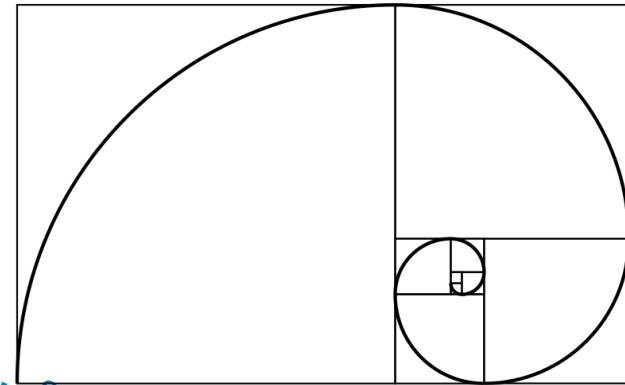
The next Fibonacci number is the sum of
the current one and its predecessor



The Fibonacci Sequence

fib	pred	[]
	curr	[]
	n	[5]
	k	[5]

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

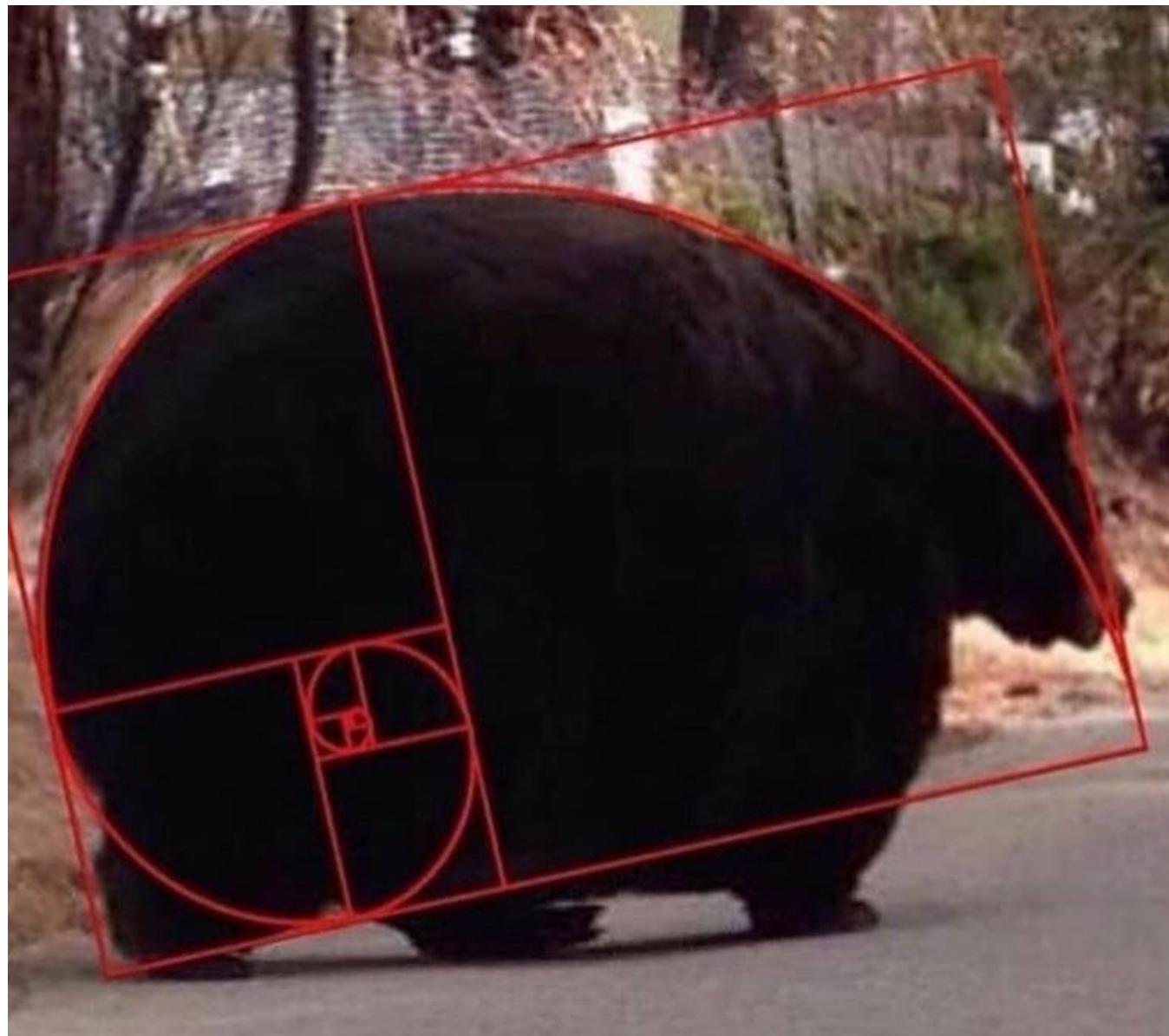


```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
    k = 1             # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of
the current one and its predecessor



Go Bears!



Designing Functions

Describing Functions

Describing Functions

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

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.

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.

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."""
```

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

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

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 Function

A Guide to Designing Function

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

A Guide to Designing Function

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

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

A Guide to Designing Function

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

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

A Guide to Designing Function

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

```
>>> round(1.23)      >>> round(1.23, 1)      >>> round(1.23, 0)  
1                      1.2                      1
```

A Guide to Designing Function

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

```
>>> round(1.23)      >>> round(1.23, 1)      >>> round(1.23, 0)      >>> round(1.23, 5)  
1                      1.2                      1                      1.23
```

A Guide to Designing Function

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

```
>>> round(1.23)      >>> round(1.23, 1)      >>> round(1.23, 0)      >>> round(1.23, 5)  
1                      1.2                      1                      1.23
```

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

A Guide to Designing Function

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

```
>>> round(1.23)      >>> round(1.23, 1)      >>> round(1.23, 0)      >>> round(1.23, 5)  
1                      1.2                      1                      1.23
```

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

(Demo)

Generalization

Generalizing Patterns with Arguments

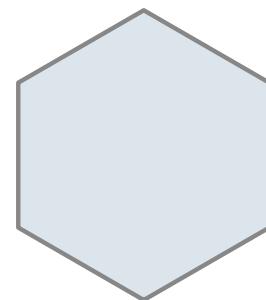
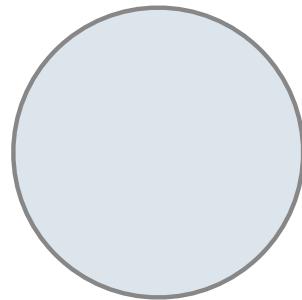
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

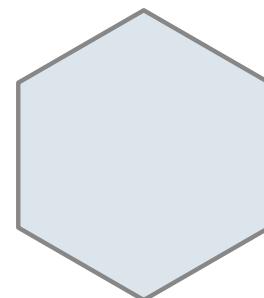
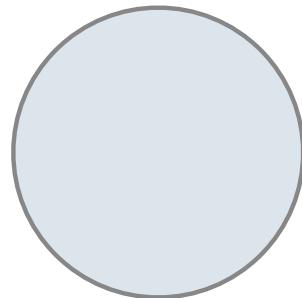
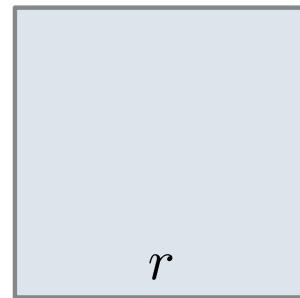
Shape:



Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

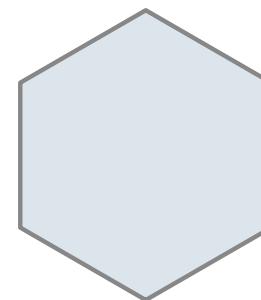
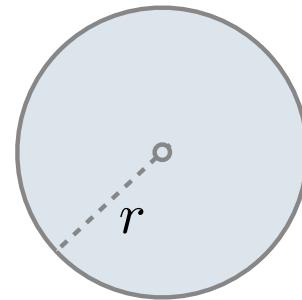
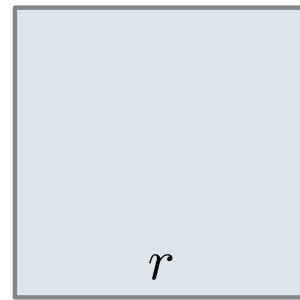
Shape:



Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

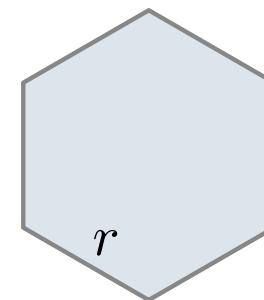
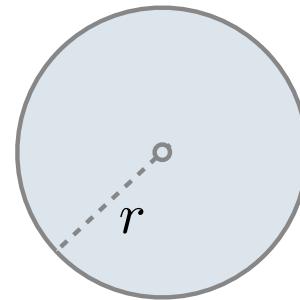
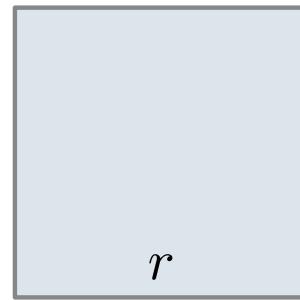
Shape:



Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

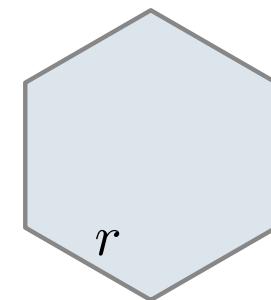
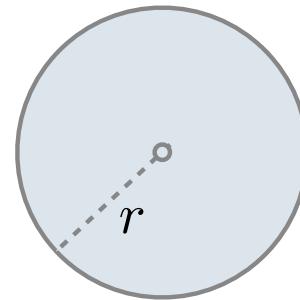
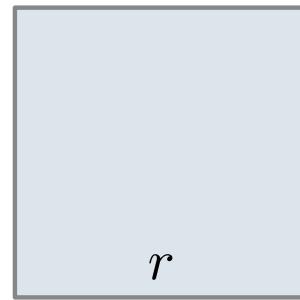
Shape:



Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

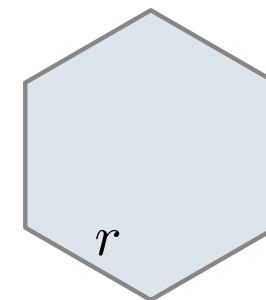
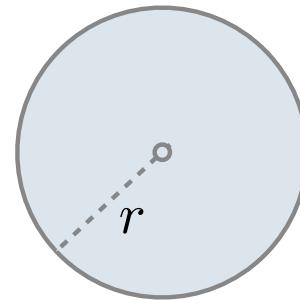
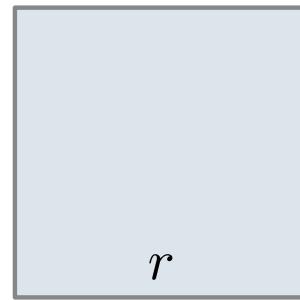


Area:

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:



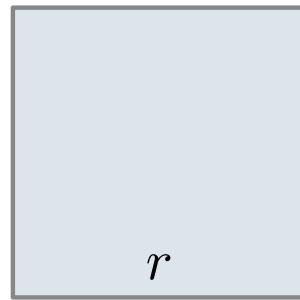
Area:

$$r^2$$

Generalizing Patterns with Arguments

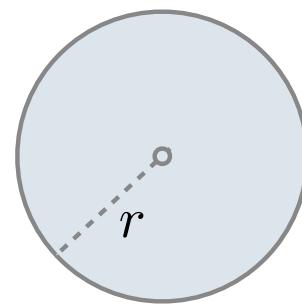
Regular geometric shapes relate length and area.

Shape:

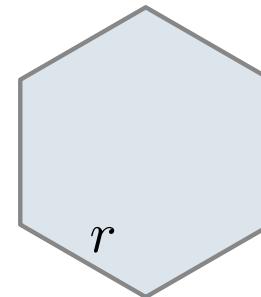


Area:

$$r^2$$



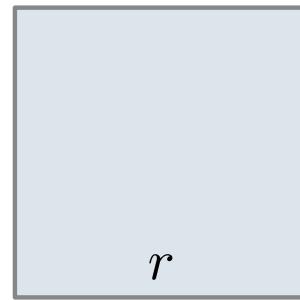
$$\pi \cdot r^2$$



Generalizing Patterns with Arguments

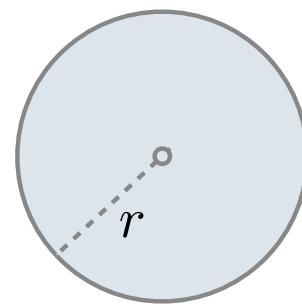
Regular geometric shapes relate length and area.

Shape:

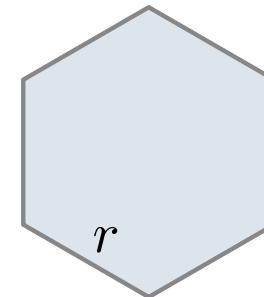


Area:

$$r^2$$



$$\pi \cdot r^2$$

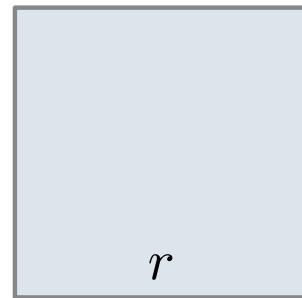


$$\frac{3\sqrt{3}}{2} \cdot r^2$$

Generalizing Patterns with Arguments

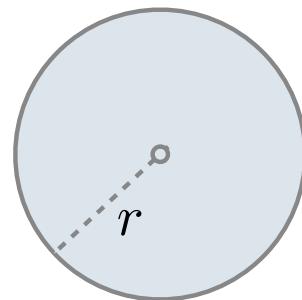
Regular geometric shapes relate length and area.

Shape:

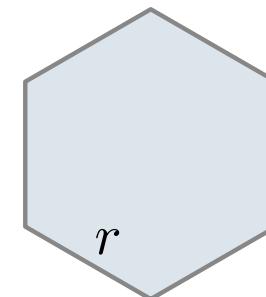


Area:

$$1 \cdot r^2$$



$$\pi \cdot r^2$$

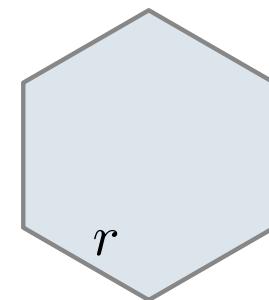
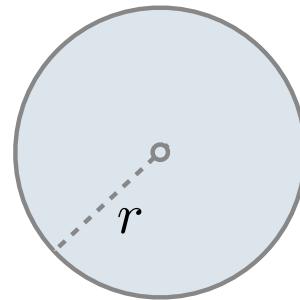
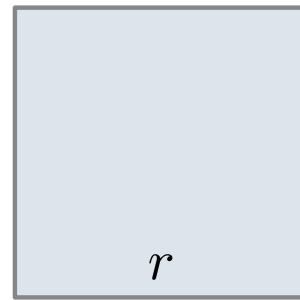


$$\frac{3\sqrt{3}}{2} \cdot r^2$$

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:



Area:

$$\boxed{1} \cdot r^2$$

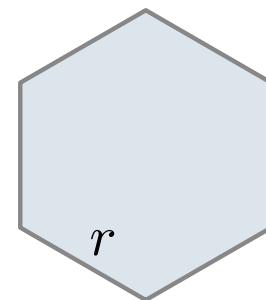
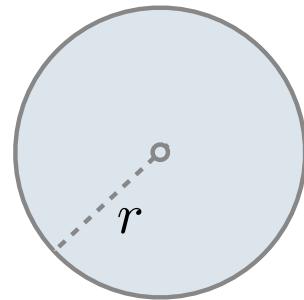
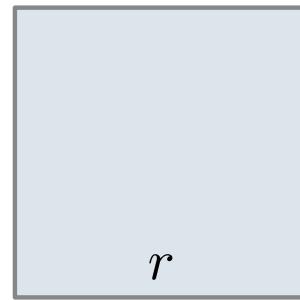
$$\pi \cdot r^2$$

$$\frac{3\sqrt{3}}{2} \cdot r^2$$

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:



Area:

$$\boxed{1} \cdot r^2$$

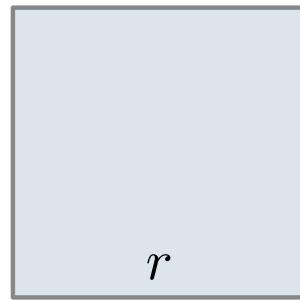
$$\boxed{\pi} \cdot r^2$$

$$\frac{3\sqrt{3}}{2} \cdot r^2$$

Generalizing Patterns with Arguments

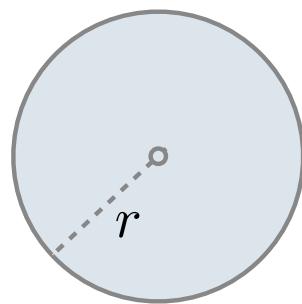
Regular geometric shapes relate length and area.

Shape:

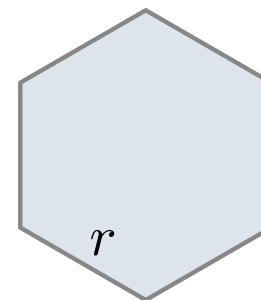


Area:

$$\boxed{1} \cdot r^2$$



$$\boxed{\pi} \cdot r^2$$

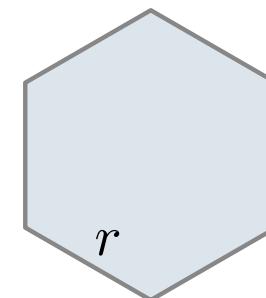
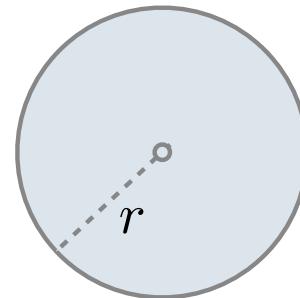
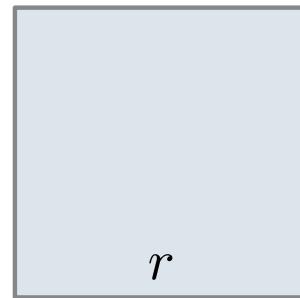


$$\boxed{\frac{3\sqrt{3}}{2}} \cdot r^2$$

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:



Area:

$$\boxed{1} \cdot r^2$$

$$\boxed{\pi} \cdot r^2$$

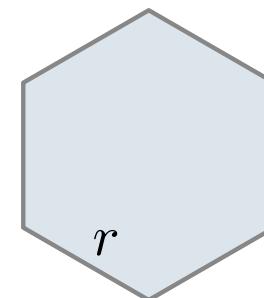
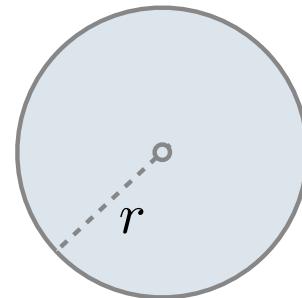
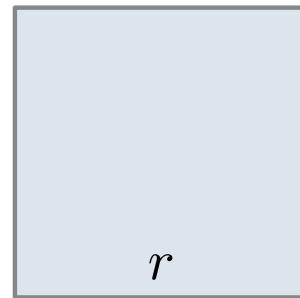
$$\boxed{\frac{3\sqrt{3}}{2}} \cdot r^2$$

Finding common structure allows for shared implementation

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:



Area:

$$\boxed{1} \cdot r^2$$

$$\boxed{\pi} \cdot r^2$$

$$\boxed{\frac{3\sqrt{3}}{2}} \cdot r^2$$

Finding common structure allows for shared implementation

(Demo)

Higher-Order Functions

Generalizing Over Computational Processes

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^5 k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^5 k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^5 \frac{8}{(4k-3) \cdot (4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^5 k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^5 k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^5 \frac{8}{(4k-3) \cdot (4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^5 k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^5 k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^5 \frac{8}{(4k-3) \cdot (4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^5 k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^5 k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^5 \frac{8}{(4k-3) \cdot (4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^5 k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^5 k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^5 \frac{8}{(4k-3) \cdot (4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

(Demo)

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
```

Summation Example

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

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

```
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

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

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

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

A formal parameter that will
be bound to a function

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

Summation Example

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

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

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

A formal parameter that will
be bound to a function

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

The function bound to term
gets called here

Summation Example

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

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

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

A formal parameter that will
be bound to a function

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

The cube function is passed
as an argument value

The function bound to term
gets called here

Summation Example

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

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

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

A formal parameter that will
be bound to a function

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

$0 + 1 + 8 + 27 + 64 + 125$

The cube function is passed
as an argument value

The function bound to term
gets called here

Functions as Return Values

(Demo)

Locally Defined Functions

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.

>>> add_three = make_adder(3)
>>> add_three(4)
7
"""
def adder(k):
    return k + n
return adder
```

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

A function that returns a function

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.

>>> add_three = make_adder(3)
>>> add_three(4)
7
"""
def adder(k):
    return k + n
return adder
```

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

```
A function that  
returns a function  
  
def make_adder(n):  
    """Return a function that takes one argument k and returns k + n.  
  
    >>> add_three = make_adder(3)  
    >>> add_three(4)  
    7  
    """  
def adder(k):  
    return k + n  
return adder
```

The name add_three is bound to a function

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

```
A function that  
returns a function  
  
def make_adder(n):  
    """Return a function that takes one argument k and returns k + n.  
  
    >>> add_three = make_adder(3)  
    >>> add_three(4)  
    7  
    """  
def adder(k):  
    return k + n  
return adder
```

The name add_three is bound to a function

A def statement within another def statement

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

```
A function that  
returns a function  
  
def make_adder(n):  
    """Return a function that takes one argument k and returns k + n.  
  
    >>> add_three = make_adder(3)  
    >>> add_three(4)  
    7  
    """  
def adder(k):  
    return k + n  
return adder
```

The name `add_three` is bound to a function

A def statement within another def statement

Can refer to names in the enclosing function

Call Expressions as Operator Expressions

Call Expressions as Operator Expressions

```
make_adder(1)      (      2      )
```

Call Expressions as Operator Expressions

Operator



```
make_adder(1) ( 2 )
```

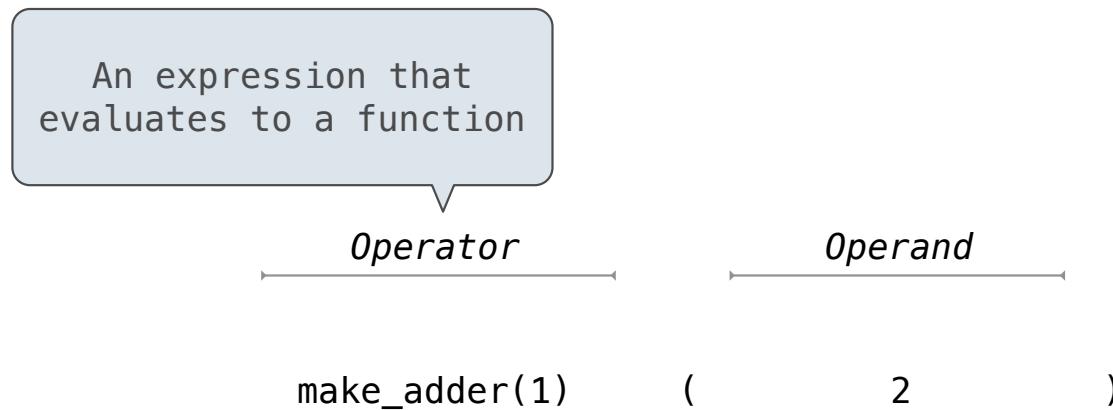
Call Expressions as Operator Expressions

Operator *Operand*

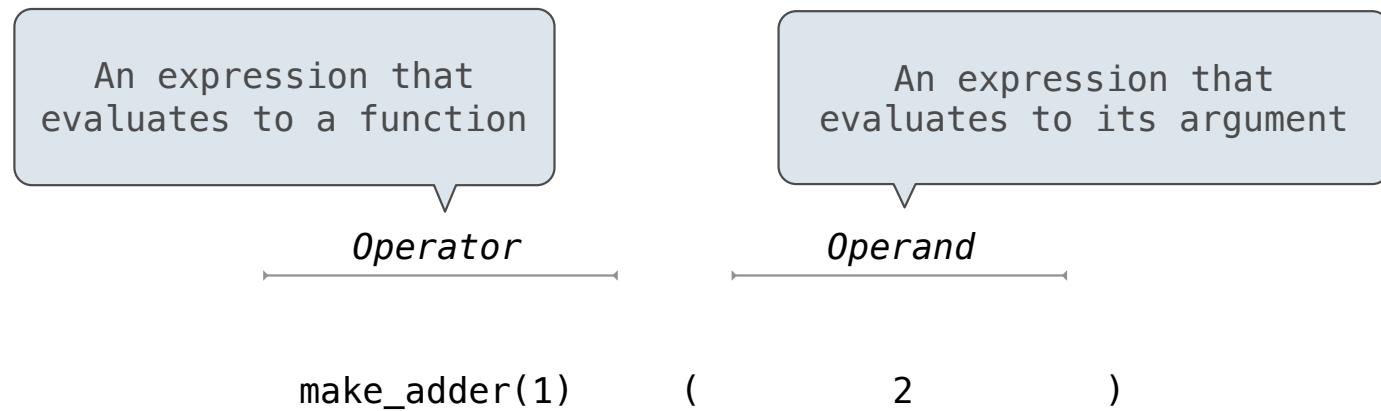
The diagram illustrates the structure of a call expression. It consists of three horizontal arrows pointing to the right. The first arrow spans from the start of the string "make_adder(1)" to the opening parenthesis. The second arrow spans from the opening parenthesis to the digit "2". The third arrow spans from the digit "2" to the closing parenthesis. Above the first arrow, the word "Operator" is written in italicized font. Above the second arrow, the word "Operand" is written in italicized font.

```
make_adder(1)      (      2      )
```

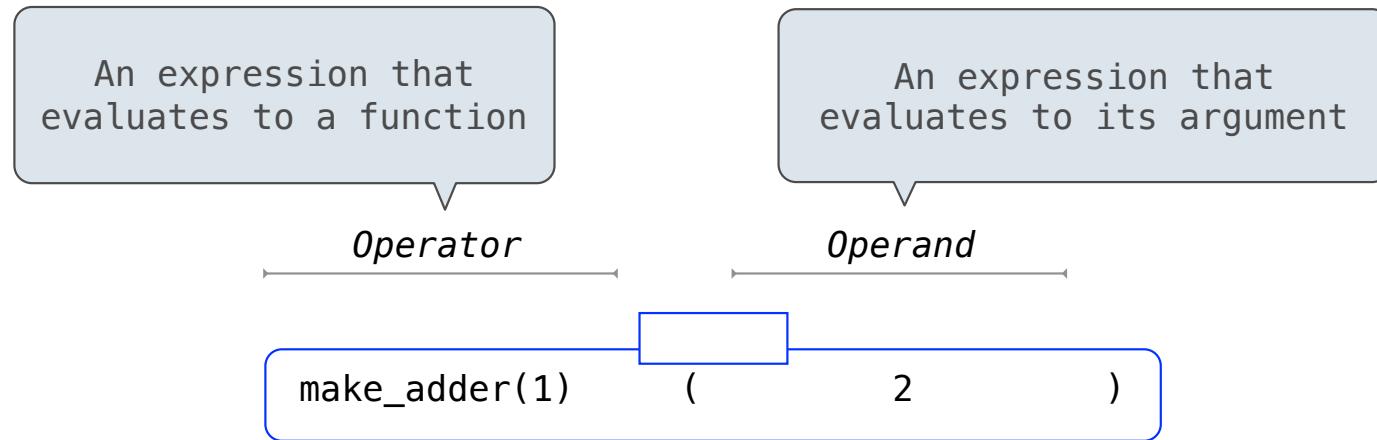
Call Expressions as Operator Expressions



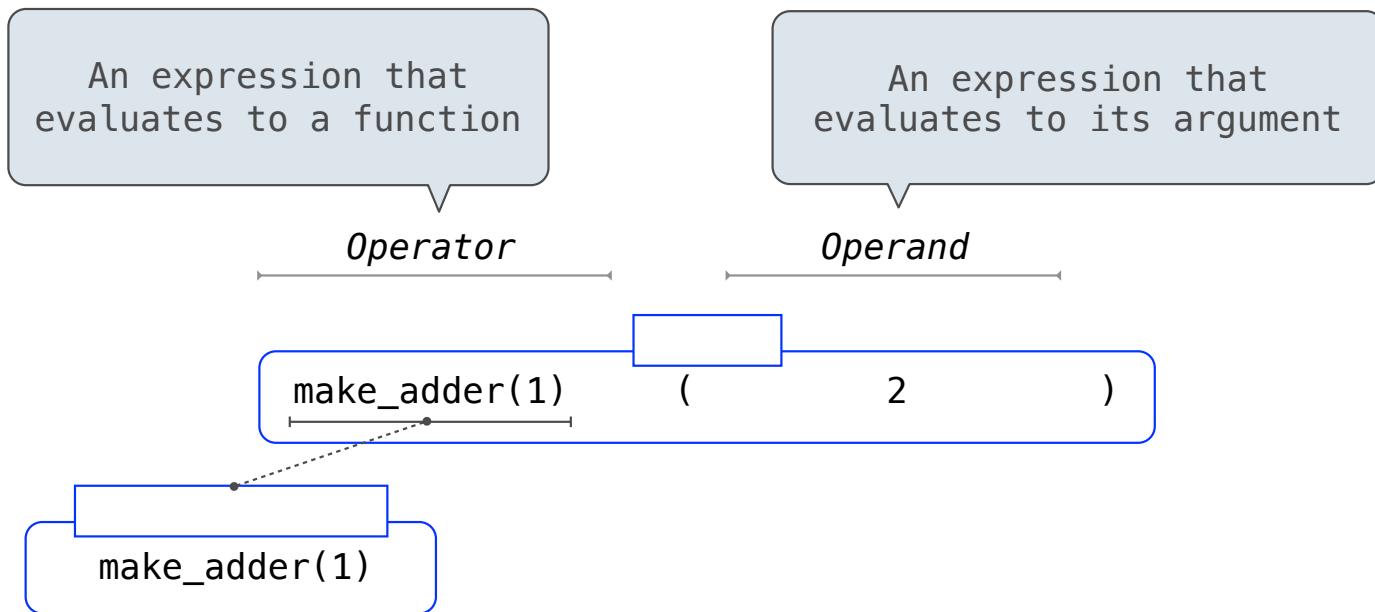
Call Expressions as Operator Expressions



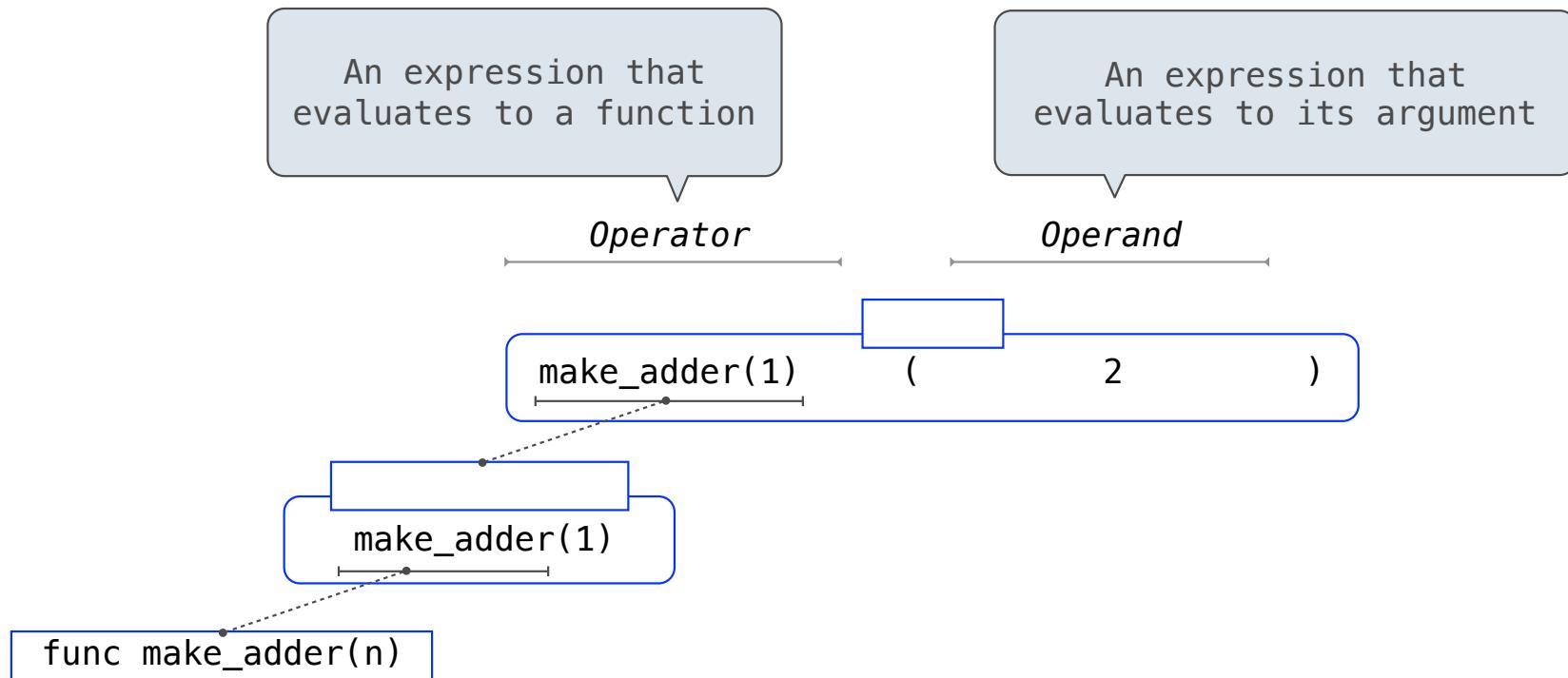
Call Expressions as Operator Expressions



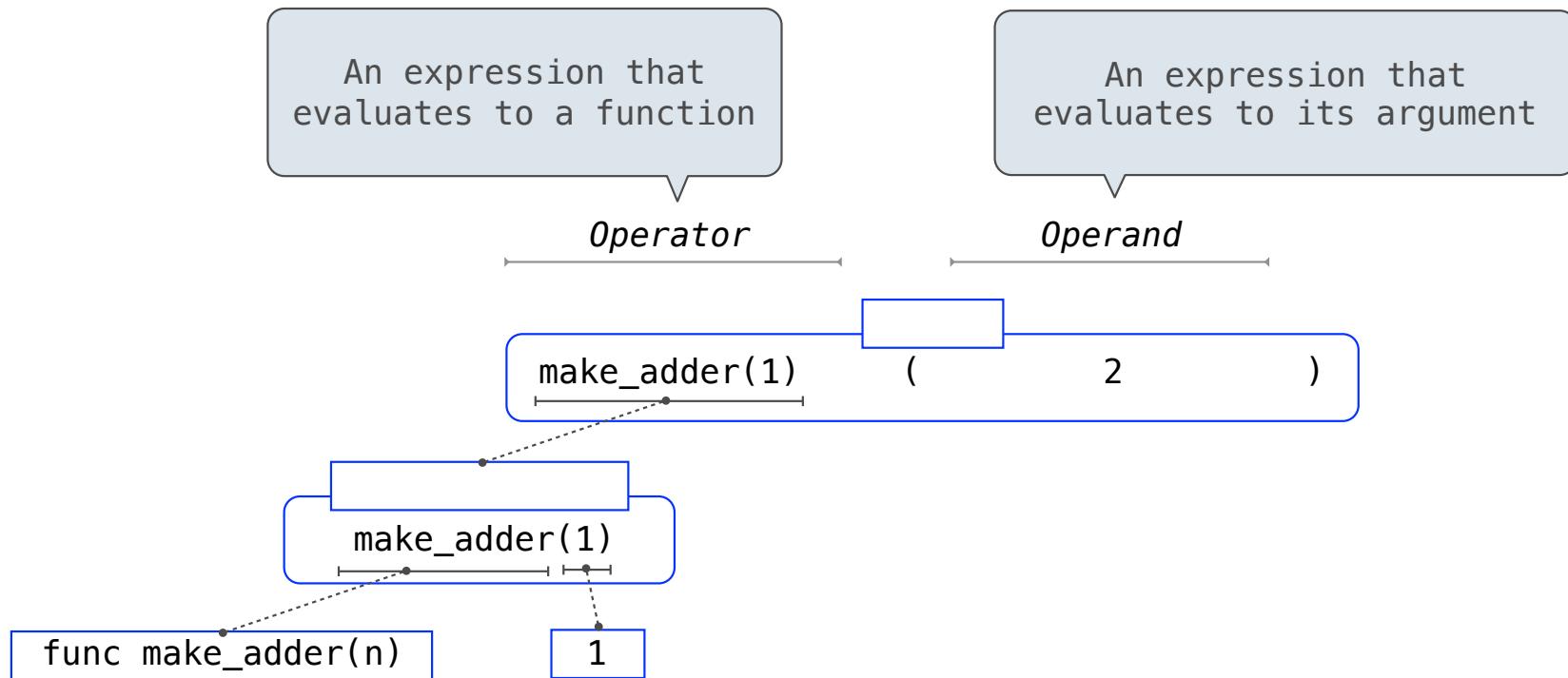
Call Expressions as Operator Expressions



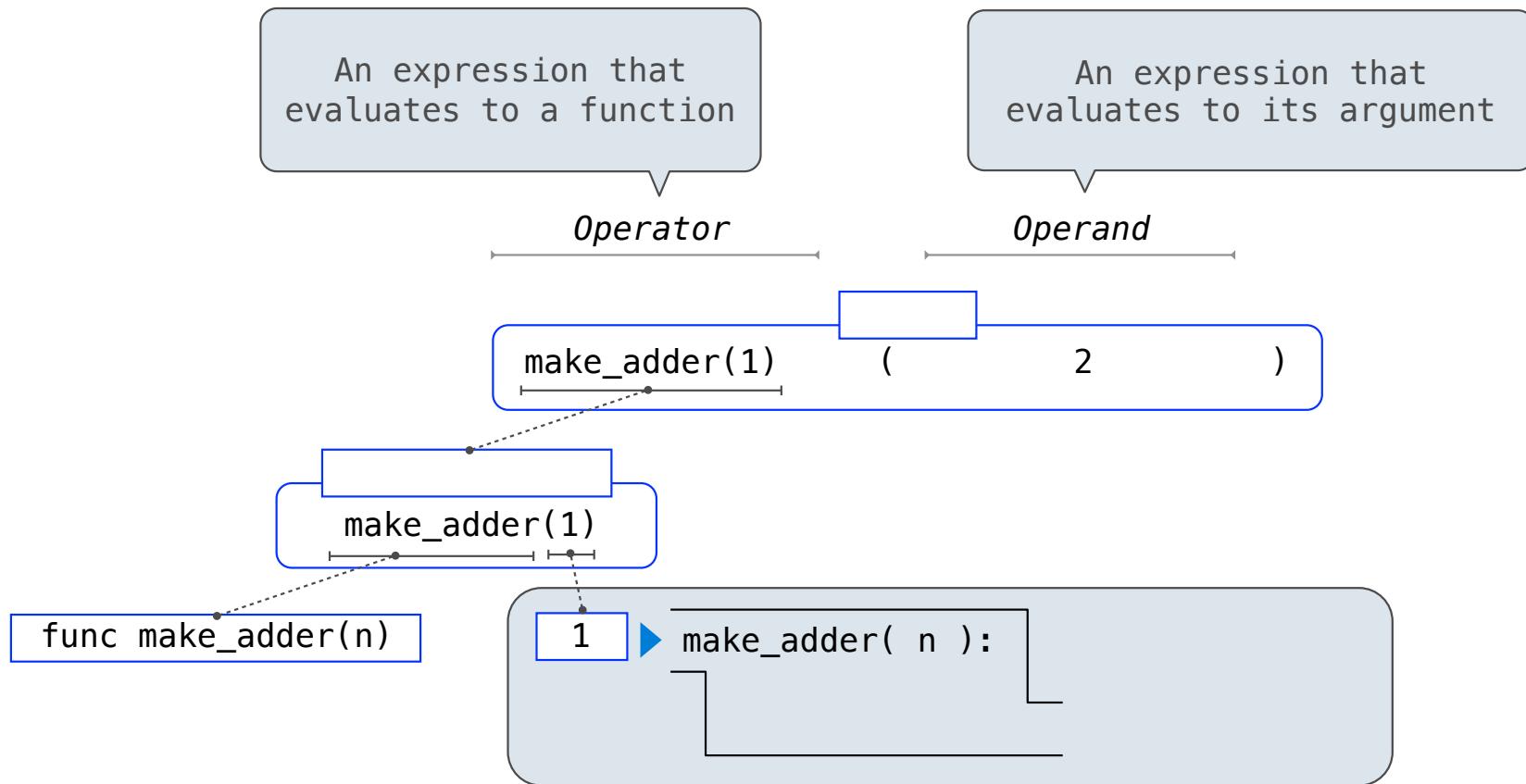
Call Expressions as Operator Expressions



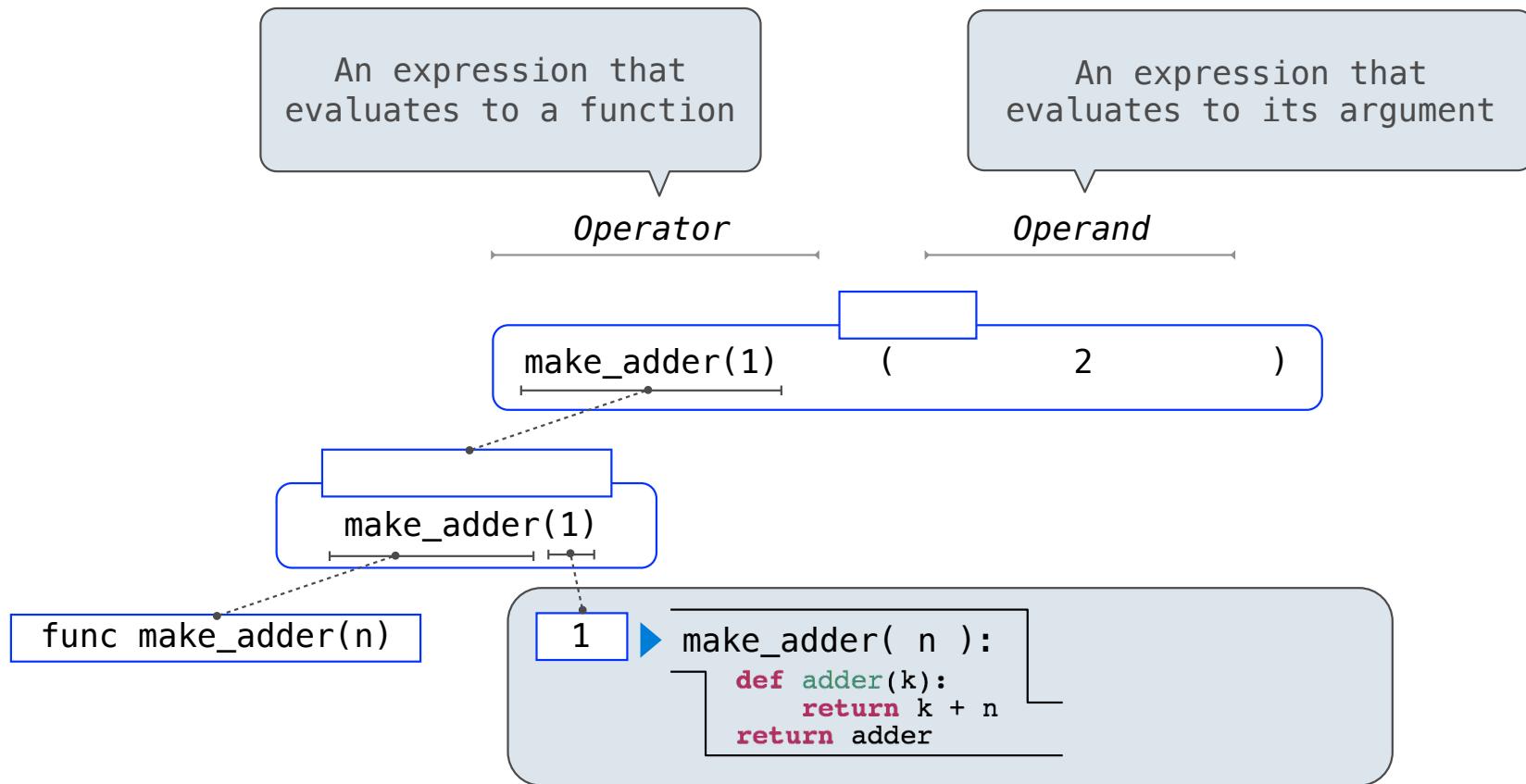
Call Expressions as Operator Expressions



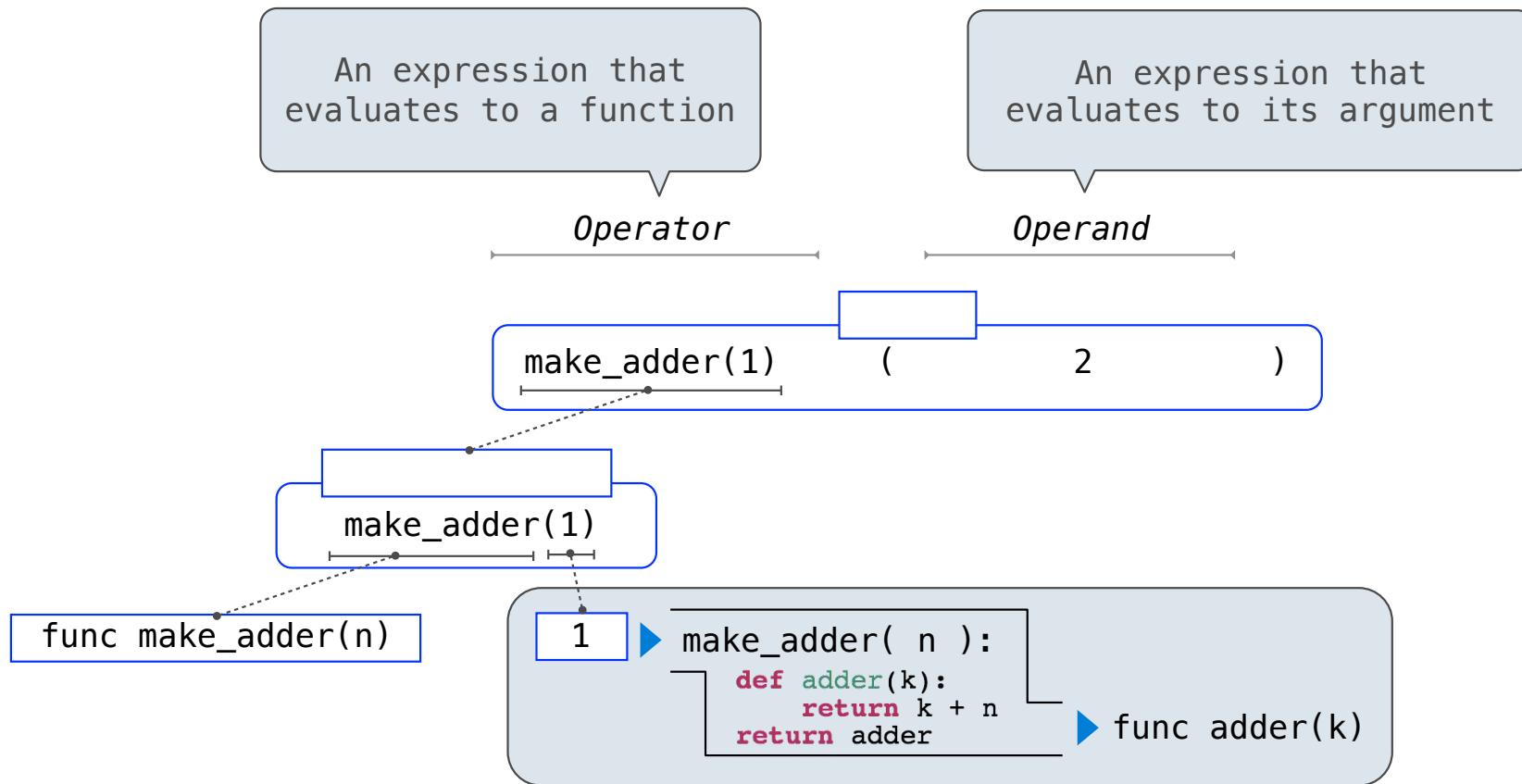
Call Expressions as Operator Expressions



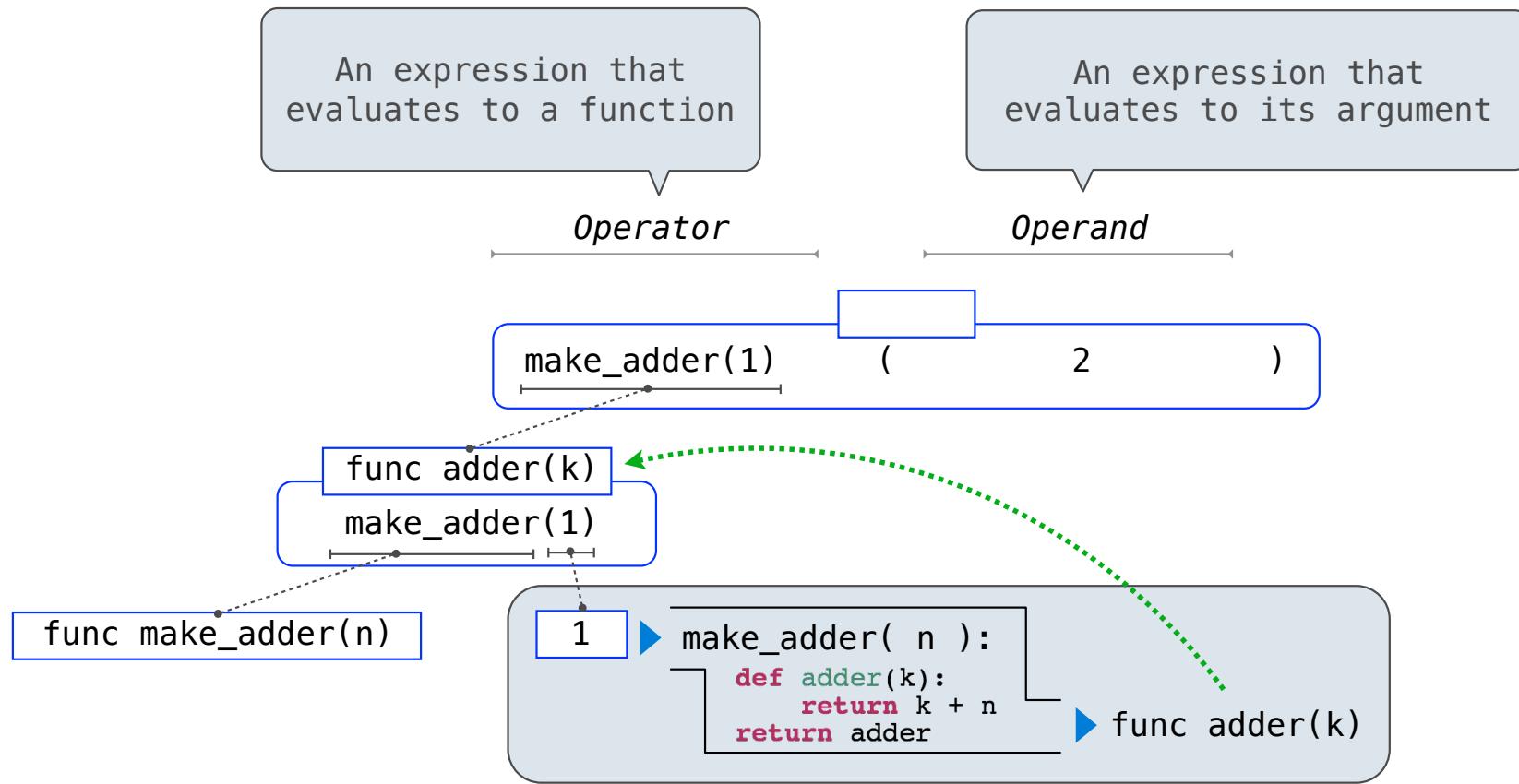
Call Expressions as Operator Expressions



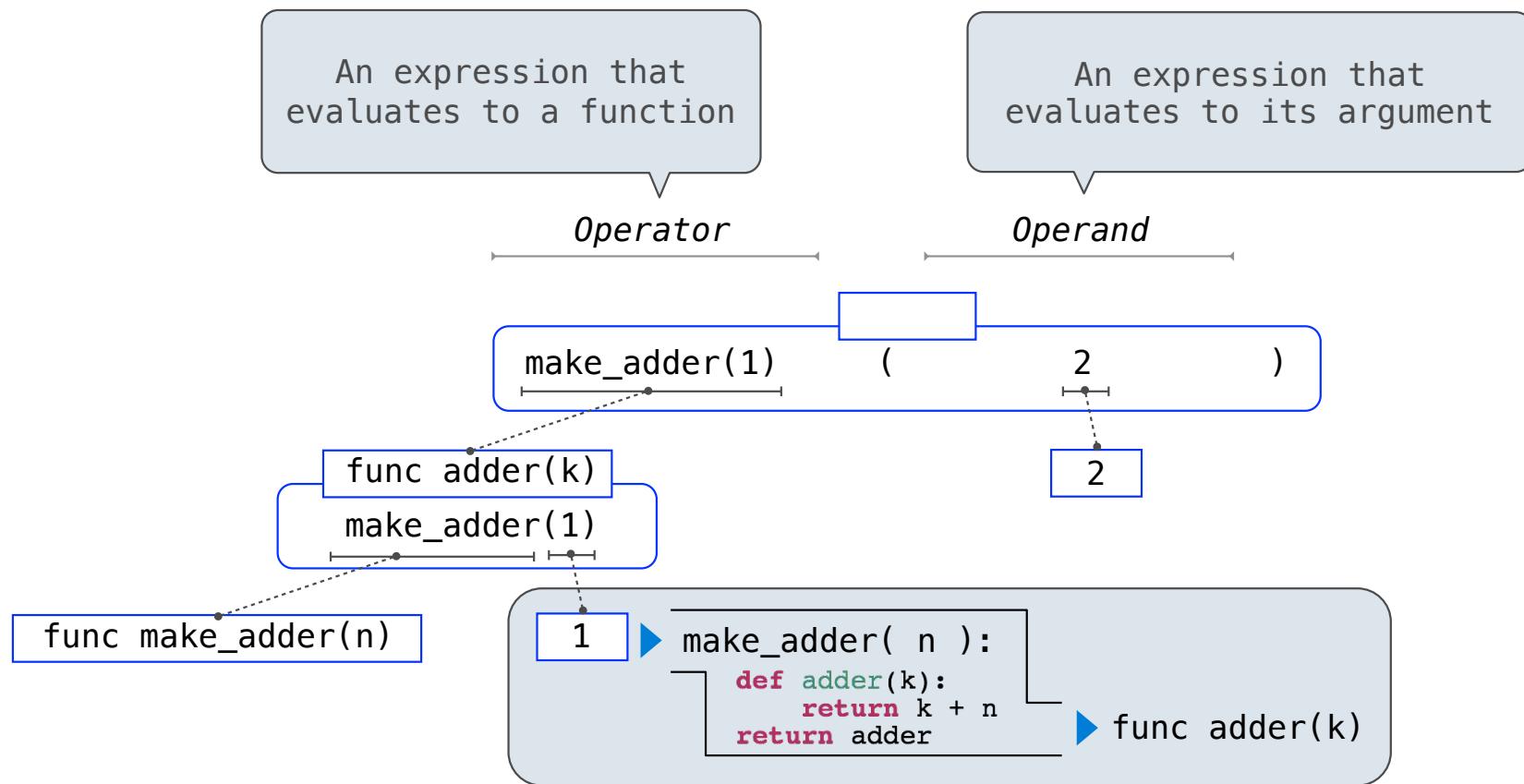
Call Expressions as Operator Expressions



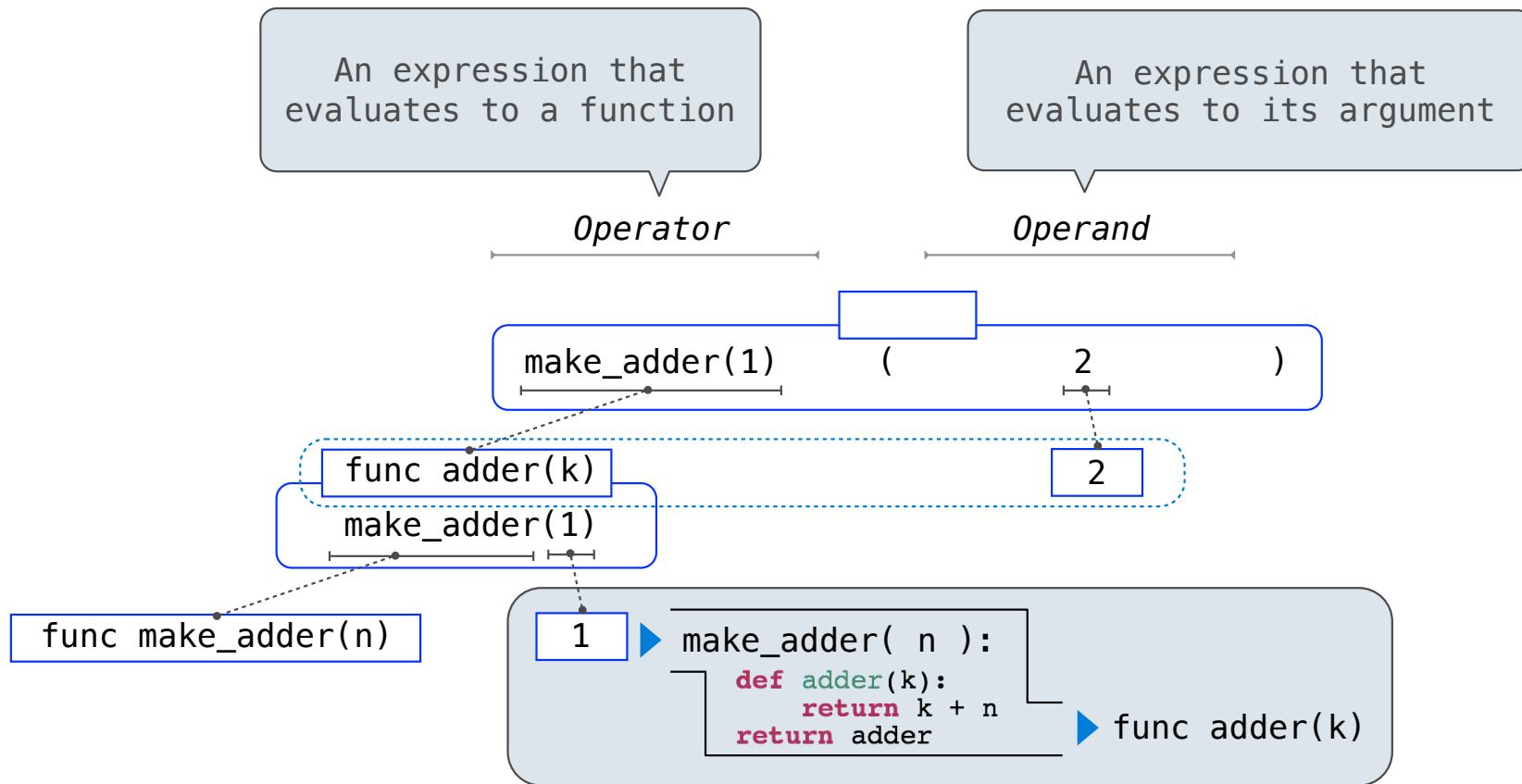
Call Expressions as Operator Expressions



Call Expressions as Operator Expressions



Call Expressions as Operator Expressions



Call Expressions as Operator Expressions

