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

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

```python
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
```
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  # The next Fibonacci number is the sum of the current one and its predecessor
        k = k + 1
    return curr

The Fibonacci Sequence

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

<table>
<thead>
<tr>
<th>fib</th>
<th>pred</th>
<th>curr</th>
<th>n</th>
<th>k</th>
</tr>
</thead>
</table>

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)  # The next Fibonacci number is the sum of the current one and its predecessor
        k = k + 1
    return curr

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

```
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

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

\[
\begin{array}{c|c|c|c}
\text{fib} & \text{pred} & \text{curr} \\
\hline
n & 5 & \\
k & 3 & \\
\end{array}
\]

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

```python
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

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.

```python
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.

```python
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.

```python
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

```python
>>> round(1.23)
1
```
A Guide to Designing Function

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

```python
>>> round(1.23)
1
>>> round(1.23, 1)
1.2
```
A Guide to Designing Function

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

```python
>>> round(1.23)
1
>>> round(1.23, 1)
1.2
>>> round(1.23, 0)
1
```
A Guide to Designing Function

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

```python
>>> round(1.23) 1
>>> round(1.23, 1) 1.2
>>> round(1.23, 0) 1
>>> round(1.23, 5) 1.23
```
A Guide to Designing Function

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

```python
>>> 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

```python
>>> 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)
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:

\[
\begin{align*}
\text{Square} & \quad \text{Circle} & \quad \text{Hexagon} \\
\end{align*}
\]
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.

<table>
<thead>
<tr>
<th>Shape:</th>
<th>Area:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td></td>
</tr>
<tr>
<td>Circle</td>
<td></td>
</tr>
<tr>
<td>Hexagon</td>
<td></td>
</tr>
</tbody>
</table>
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:**
- Square: \( r \)
- Circle: \( \pi \cdot r^2 \)
- Hexagon: \( \frac{3\sqrt{3}}{2} \cdot r^2 \)

**Area:**
- Square: \( r^2 \)
- Circle: \( \pi \cdot r^2 \)
- Hexagon: \( \frac{3\sqrt{3}}{2} \cdot r^2 \)
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:

\[ \text{Square}: \quad 1 \cdot r^2 \]

\[ \text{Circle}: \quad \pi \cdot r^2 \]

\[ \text{Hexagon}: \quad \frac{3\sqrt{3}}{2} \cdot r^2 \]
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:  
- Square: $r^2$
- Circle: $\pi \cdot r^2$
- Hexagon: $\frac{3\sqrt{3}}{2} \cdot r^2$

Area:
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:
- Square: $r^2$
- Circle: $\pi r^2$
- Hexagon: $\frac{3\sqrt{3}}{2} r^2$

Area:
- Square: $1 r^2$
- Circle: $\pi r^2$
- Hexagon: $\frac{3\sqrt{3}}{2} r^2$

Finding common structure allows for shared implementation
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

Area:

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.

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

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

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

(Demo)
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
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")
Summation Example

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

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

    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225
"""
```

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

A formal parameter that will be bound to a function

# Local function definitions; returning functions

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

    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    """Return a function that composes f and g."

    def h(x):
        return f(g(x))
    return h
```
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

    The function bound to term gets called here

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

A formal parameter that will be bound to a function
Summation Example

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

def summation(n, term):
    """Sum the first n terms of a sequence."
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

```python
>>> summation(5, cube)
225
"""
```

- 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
- The function bound to term gets called here
def cube(k):
    return pow(k, 3)

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

    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225

"""
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

```python
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

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

```bash
>>> add_three = make_adder(3)
>>> add_three(4)
7
"""
```
Locally Defined Functions

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

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

The name `add_three` is bound to a function

```python
>>> add_three = make_adder(3)
>>> add_three(4)
7
"""
```
Locally Defined Functions

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

A function that returns a function

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

The name add_three is bound to a function

```
>>> add_three = make_adder(3)
```

A def statement within another def statement

```
>>> add_three(4)
7
```
Locally Defined Functions

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

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

```
>>> add_three = make_adder(3)
>>> add_three(4)
7
```

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

\[
\text{make_adder}(1) (2)
\]
Call Expressions as Operator Expressions

\[ \text{Operator} \]

\[ \text{make_adder}(1) \quad ( \quad 2 \quad ) \]
Call Expressions as Operator Expressions

Operator

make_adder(1) ( 2 )

Operand
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

make_adder(1)     (         2         )

Operand
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

make_adder(1) ( 2 )

An expression that evaluates to its argument

Operand
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

Operator

Operand

make_adder(1) ( 2 )
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

An expression that evaluates to its argument

Operand

make_adder(1) ( 2 )

make_adder(1)
Call Expressions as Operator Expressions

An expression that evaluates to a function

make_adder(1)     (         2         )

Operator

Operand

An expression that evaluates to its argument

func make_adder(n)
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

func make_adder(n) 1

make_adder(1) ( 2 )

make_adder(1)
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

make_adder(1)     (         2         )

Operator

Operand

func make_adder(n)

1     make_adder( n ):
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

**Operator**

**Operand**

```
func make_adder(n):
def adder(k):
    return k + n
return adder
```

```
make_adder(1)
```

```
( 2 )
```
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

make_adder(1) ( 2 )

func make_adder(n)

1

make_adder(n):
def adder(k):
    return k + n
    return adder

func adder(k)
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

Operator

Operand

make_adder(1)
(2)

func adder(k)

make_adder(1)

func make_adder(n)

1

```python
def adder(k):
    return k + n

func adder(k)
```

Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

make_adder(1)

Operand

An expression that evaluates to its argument

func adder(k)

make_adder(1)

func make_adder(n)

def adder(k):
    return k + n

func adder(k)
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

Operator

Operand

make_adder(1) ( 2 )

func adder(k)

make_adder(1)

func make_adder(n)

1

make_adder(n):
def adder(k):
    return k + n
return adder

func adder(k)
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

An expression that evaluates to its argument

Operand

func make_adder(n)

func adder(k)

make_adder(1) ( 2 )

make_adder(1)

func adder(k)

make_adder(n):

def adder(k):
    return k + n

return adder

func adder(k)
Lambda Expressions

(Demo)
Lambda Expressions
Lambda Expressions

```python
>>> x = 10
```
Lambda Expressions

```python
>>> x = 10

>>> square = x * x
```

20
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
```

20
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
```

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

```python
>>> square = lambda x: x * x
```
Lambda Expressions

>>> x = 10

An expression: this one evaluates to a number

>>> square = x * x

Also an expression: evaluates to a function

>>> square = lambda x: x * x
Lambda Expressions

>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
A function
Lambda Expressions

```python
>>> x = 10

>>> square = x * x
An expression: this one evaluates to a number

>>> square = lambda x: x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
A function
with formal parameter x
```
Lambda Expressions

>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
A function
    with formal parameter x
    that returns the value of "x * x"
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
Important: No "return" keyword!

A function

with formal parameter x
that returns the value of "x * x"
```
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
Important: No "return" keyword!
A function
    with formal parameter x
    that returns the value of "x * x"
Must be a single expression
```
Lambda Expressions

>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
Important: No "return" keyword!

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

>>> square(4)
16
Must be a single expression
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
Important: No "return" keyword!
A function with formal parameter x that returns the value of "x * x"

>>> square(4)
16
Must be a single expression

Lambda expressions are not common in Python, but important in general
```
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number
``` 

```python
>>> square = x * x
Also an expression: evaluates to a function
``` 

```python
>>> square = lambda x: x * x
Important: No "return" keyword!
A function
with formal parameter x
that returns the value of "x * x"
``` 

```python
>>> square(4)
16
Must be a single expression
``` 

Lambda expressions are not common in Python, but important in general.

Lambda expressions in Python cannot contain statements at all!
Lambda Expressions Versus Def Statements
Lambda Expressions Versus Def Statements
Lambda Expressions Versus Def Statements

\[
square = \text{lambda } x: x * x
\]

VS
Lambda Expressions Versus Def Statements

\[
square = \text{lambda } x: x \times x
\]

\[
\text{def square(x): return } x \times x
\]
Lambda Expressions Versus Def Statements

\[
\text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def square}(x):
\]
\[
\text{\quad return } x \times x
\]

- Both create a function with the same domain, range, and behavior.
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 bind that function to the name square.
Lambda Expressions Versus Def Statements

\[ \text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def square}(x): \text{return } x \times x \]

• Both create a function with the same domain, range, and behavior.
• Both bind that function to the name square.
• Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Lambda Expressions Versus Def Statements

\[
\text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def square}(x): \quad \text{return} \ x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \times x
\]

VS

\[
def square(x):
    return x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Lambda Expressions Versus Def Statements

\[
square = \text{lambda } x: x \times x
\]

\[
def \text{square}(x):
    \text{return } x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Lambda Expressions Versus Def Statements

\[
\text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def square}(x):
\]
\[
\text{return } x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Lambda Expressions Versus Def Statements

\[
square = \text{lambda } x: x \times x
\]

\[
def \text{square}(x):
    \text{return } x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name \( \text{square} \).
- Only the \text{def} statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Return
Return Statements
Return Statements

A return statement completes the evaluation of a call expression and provides its value:
Return Statements

A return statement completes the evaluation of a call expression and provides its value:

\[ f(x) \text{ for user-defined function } f: \text{ switch to a new environment; execute } f's \text{ body} \]
Return Statements

A return statement completes the evaluation of a call expression and provides its value:

- \( f(x) \) for user-defined function \( f \): switch to a new environment; execute \( f \)'s body
- `return` statement within \( f \): switch back to the previous environment; \( f(x) \) now has a value
Return Statements

A return statement completes the evaluation of a call expression and provides its value:

- \( f(x) \) for user-defined function \( f \): switch to a new environment; execute \( f \)'s body

- `return` statement within \( f \): switch back to the previous environment; \( f(x) \) now has a value

Only one return statement is ever executed while executing the body of a function
Return Statements

A return statement completes the evaluation of a call expression and provides its value:

\[ f(x) \] for user-defined function \( f \): switch to a new environment; execute \( f \)'s body

\[ \text{return} \] statement within \( f \): switch back to the previous environment; \( f(x) \) now has a value

Only one return statement is ever executed while executing the body of a function

```python
def end(n, d):
    """Print the final digits of N in reverse order until D is found."

>>> end(34567, 5)
7
6
5
"""
```
Return Statements

A return statement completes the evaluation of a call expression and provides its value:

- \( f(x) \) for user-defined function \( f \): switch to a new environment; execute \( f \)'s body
- \textbf{return} statement within \( f \): switch back to the previous environment; \( f(x) \) now has a value

Only one return statement is ever executed while executing the body of a function

```python
def end(n, d):
    """Print the final digits of \( N \) in reverse order until \( D \) is found."

    >>> end(34567, 5)
    7
    6
    5
    """
    while n > 0:
        last, n = n % 10, n // 10
        print(last)
```
Return Statements

A return statement completes the evaluation of a call expression and provides its value:

$f(x)$ for user-defined function $f$: switch to a new environment; execute $f$'s body

return statement within $f$: switch back to the previous environment; $f(x)$ now has a value

Only one return statement is ever executed while executing the body of a function

```python
def end(n, d):
    """Print the final digits of N in reverse order until D is found."

    >>> end(34567, 5)
    7
    6
    5
    """

    while n > 0:
        last, n = n % 10, n // 10
        print(last)
        if d == last:
            return None
```
Return Statements

A return statement completes the evaluation of a call expression and provides its value:

- \( f(x) \) for user-defined function \( f \): switch to a new environment; execute \( f \)'s body
- \texttt{return} statement within \( f \): switch back to the previous environment; \( f(x) \) now has a value

Only one return statement is ever executed while executing the body of a function

```python
def end(n, d):
    """Print the final digits of N in reverse order until D is found."

>>> end(34567, 5)
7
6
5
"""
while n > 0:
    last, n = n % 10, n // 10
print(last)
if d == last:
    return None       # Demo
```

(Demo)
Control
If Statements and Call Expressions

Let's try to write a function that does the same thing as an if statement.
If Statements and Call Expressions

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

```python
if __________:
    __________
else:
    __________
```
If Statements and Call Expressions

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

```python
if ________:
    _______
else:
    _______
```

*Execution Rule for Conditional Statements:*
If Statements and Call Expressions

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

```python
if ________:
    ________
else:
    ________
```

Execution Rule for Conditional Statements:
Each clause is considered in order.
If Statements and Call Expressions

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

```python
if ________:
    _______
else:
    _______
```

**Execution Rule for Conditional Statements:**
Each clause is considered in order.

1. Evaluate the header's expression (if present).
If Statements and Call Expressions

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

```python
if _________:
    _________
else:
    _________
```

**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.
If Statements and Call Expressions

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

```python
if ________:
    _______
else:
    _______
```

**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.
If Statements and Call Expressions

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

"if" clause
if _________:
    ________

"else" clause
else:
    ________

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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
If Statements and Call Expressions

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

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

**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.
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.

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

### Evaluation Rule for Call Expressions:
Let's try to write a function that does the same thing as an if statement.

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

**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
If Statements and Call Expressions

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

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

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
If Statements and Call Expressions

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

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

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.
Control Expressions
Logical Operators
Logical Operators

To evaluate the expression `<left> and <right>`:
Logical Operators

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

1. Evaluate the subexpression `<left>`.,
Logical Operators

To evaluate the expression \(<\text{left}> \text{ and } \text{right}>\):

1. Evaluate the subexpression \(<\text{left}>\).

2. If the result is a false value \(v\), then the expression evaluates to \(v\).
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> 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>`:
Logical Operators

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>`. 
Logical Operators

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.
Logical Operators

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>`.
Logical Operators

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>`.

(Demo)
Conditional Expressions
Conditional Expressions

A conditional expression has the form

\[ \text{<consequent>} \text{ if } \text{<predicate>} \text{ else } \text{<alternative>} \]
Conditional Expressions

A conditional expression has the form

<consequent> if <predicate> else <alternative>

Evaluation rule:
Conditional Expressions

A conditional expression has the form

\[ \text{<consequent>} \text{ if } \text{<predicate>} \text{ else } \text{<alternative>} \]

**Evaluation rule:**

1. Evaluate the \text{<predicate>} expression.
Conditional Expressions

A conditional expression has the form

<consequent> if <predicate> else <alternative>

Evaluation rule:

1. Evaluate the <predicate> expression.

2. If it's a true value, the value of the whole expression is the value of the <consequent>. 
Conditional Expressions

A conditional expression has the form

```plaintext
<consequent> if <predicate> else <alternative>
```

**Evaluation rule:**

1. Evaluate the `<predicate>` expression.

2. If it's a true value, the value of the whole expression is the value of the `<consequent>`.

3. Otherwise, the value of the whole expression is the value of the `<alternative>`.
Conditional Expressions

A conditional expression has the form

\[ \text{<consequent>} \text{ if } \text{<predicate>} \text{ else } \text{<alternative>} \]

Evaluation rule:

1. Evaluate the \text{<predicate>} expression.

2. If it's a true value, the value of the whole expression is the value of the \text{<consequent>}.

3. Otherwise, the value of the whole expression is the value of the \text{<alternative>}.

```python
>>> x = 0
>>> abs(1/x if x != 0 else 0)
0
```