Functional Abstraction

Announcements

Lambda Function Environments

Environment Diagrams with Lambda

A lambda function's parent is the current frame in which the lambda expression is evaluated

https://pythontutor.com/composingprograms.html#code=a%20%3D%201%0Adef%20f%28g%29%3A%0A%20%20%20%20a%20%3D%202%0A%20%20%20%20return%20lambda%20y%3A%20a%20*%20g%28y%29%0Af%28lambda%20y%3A%20a%20%2B%20y%29%28a%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Environment Diagrams with Lambda

A lambda function's parent is the current frame in which the lambda expression is evaluated

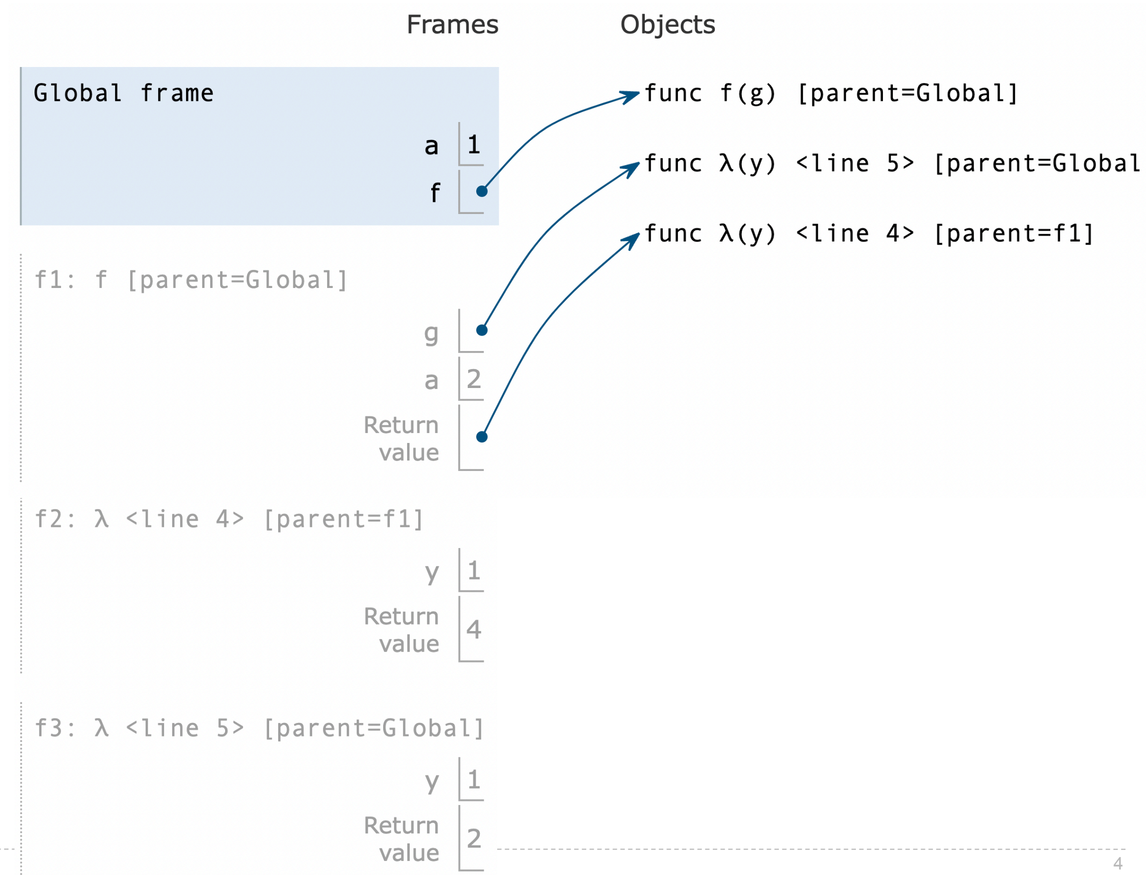
```
a = 1
def f(g):
    a = 2
    return lambda y: a * g(y)
f(lambda y: a + y)(a)
```


https://pythontutor.com/composingprograms.html#code=a%20%3D%201%0Adef%20f%28g%29%3A%0A%20%20%20%20a%20%3D%202%0A%20%20%20%20return%20lambda%20y%3A%20a%20*%20g%28y%29%0Af%28lambda%20y%3A%20a%20%2B%20y%29%28a%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Environment Diagrams with Lambda

A lambda function's parent is the current frame in which the lambda expression is evaluated

```
1  a = 1
2  def f(g):
3      a = 2
4      return lambda y: a * g(y)
5  f(lambda y: a + y)(a)
```

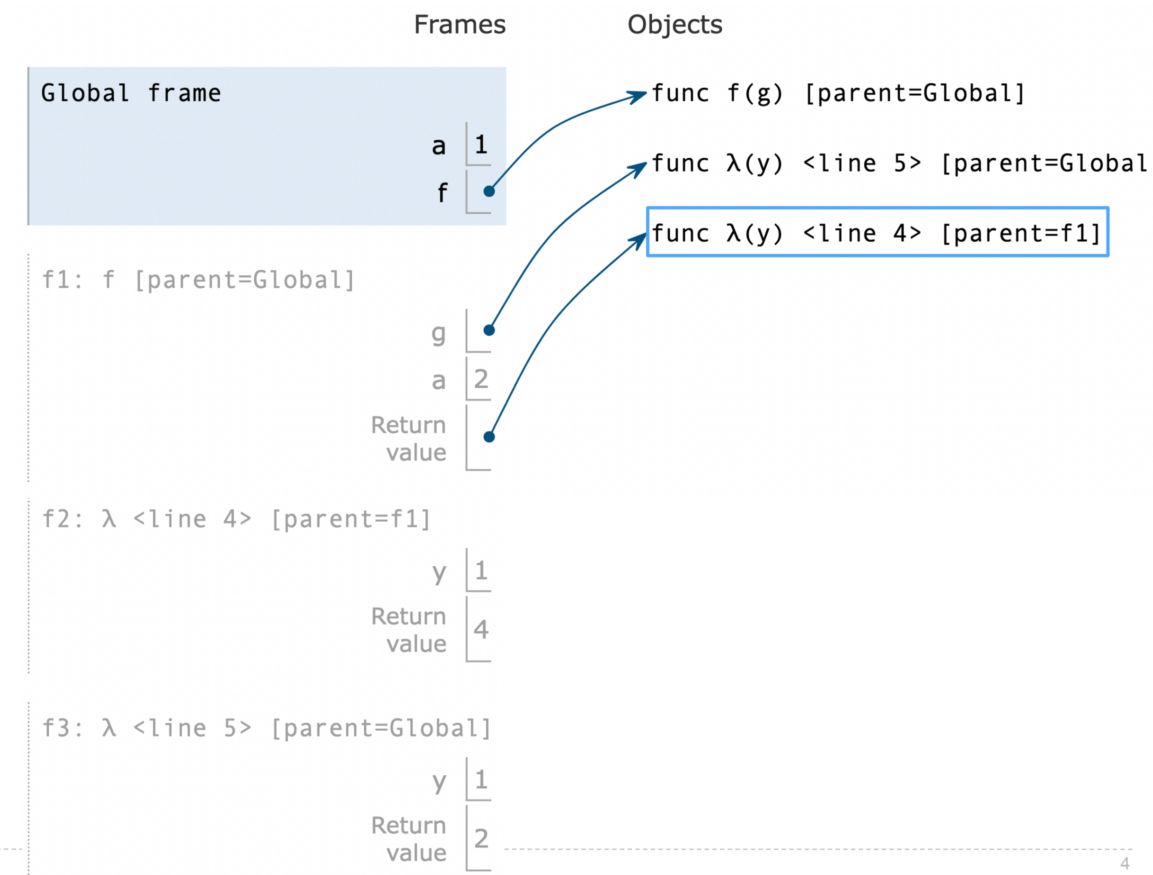


https://pythontutor.com/composingprograms.html#code=a%20%3D%201%0Adef%20f%28g%29%3A%0A%20%20%20%20a%20%3D%202%0A%20%20%20%20return%20lambda%20y%3A%20a%20*%20g%28y%29%0Af%28lambda%20y%3A%20a%20%2B%20y%29%28a%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Environment Diagrams with Lambda

A lambda function's parent is the current frame in which the lambda expression is evaluated

```
1  a = 1
2  def f(g):
3      a = 2
4      return lambda y: a * g(y)
5  f(lambda y: a + y)(a)
```

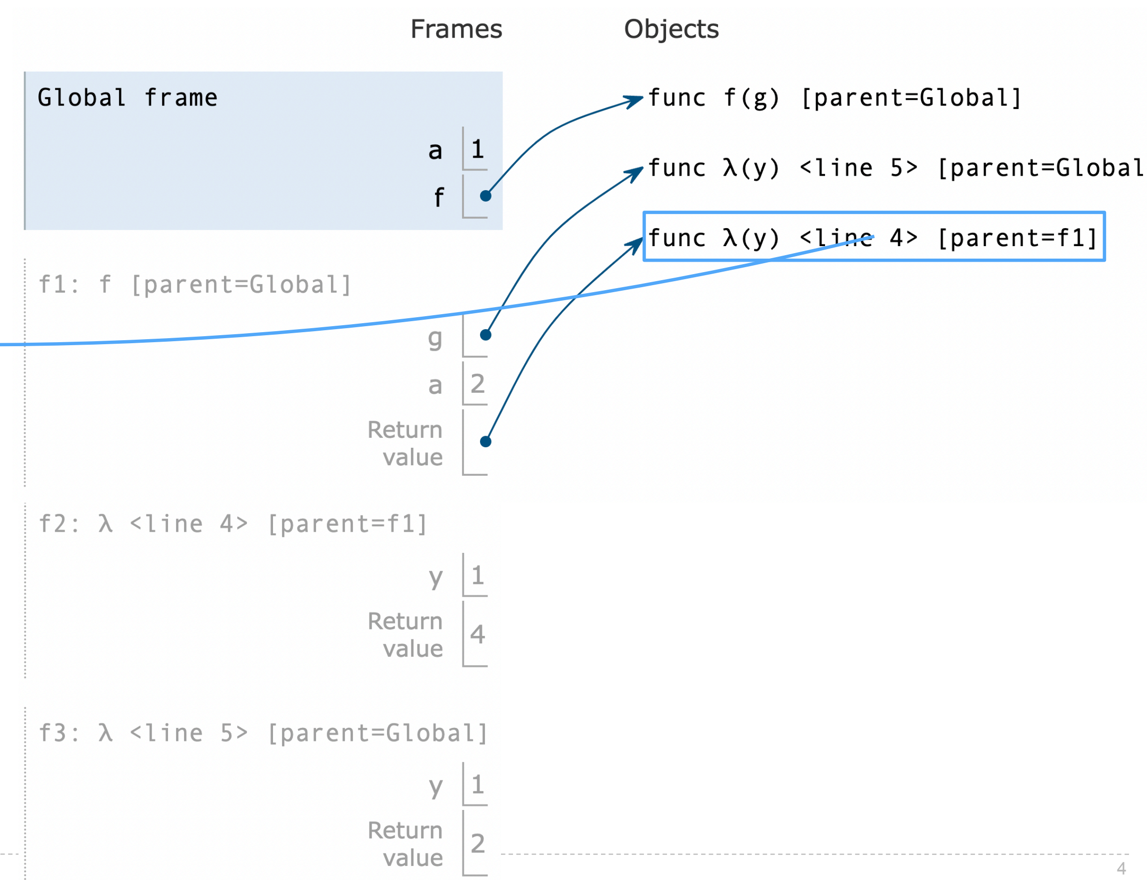


https://pythontutor.com/composingprograms.html#code=a%20%3D%201%0Adef%20f%28g%29%3A%0A%20%20%20%20a%20%3D%202%0A%20%20%20%20return%20lambda%20y%3A%20a%20*%20g%28y%29%0Af%28lambda%20y%3A%20a%20%2B%20y%29%28a%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Environment Diagrams with Lambda

`A lambda function's parent is the current frame in which the lambda expression is evaluated`

```
1 a = 1
2 def f(g):
3     a = 2
4     return lambda y: a * g(y)
5 f(lambda y: a + y)(a)
```

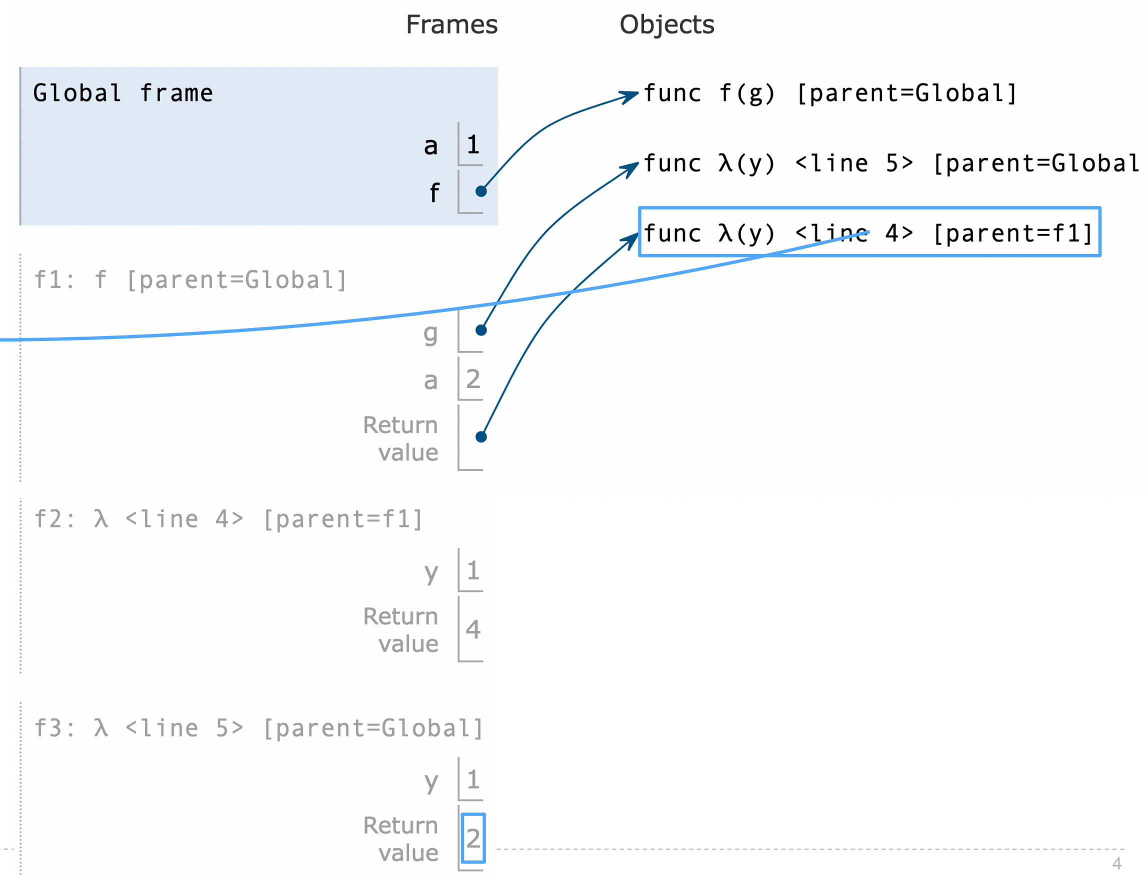


https://pythontutor.com/composingprograms.html#code=a%20%3D%201%0Adef%20f%28g%29%3A%0A%20%20%20%20a%20%3D%202%0A%20%20%20%20return%20lambda%20y%3A%20a%20*%20g%28y%29%0Af%28lambda%20y%3A%20a%20%2B%20y%29%28a%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Environment Diagrams with Lambda

A lambda function's parent is the current frame in which the lambda expression is evaluated

```
1  a = 1
2  def f(g):
3      a = 2
4      return lambda y: a * g(y)
5  f(lambda y: a + y)(a)
```

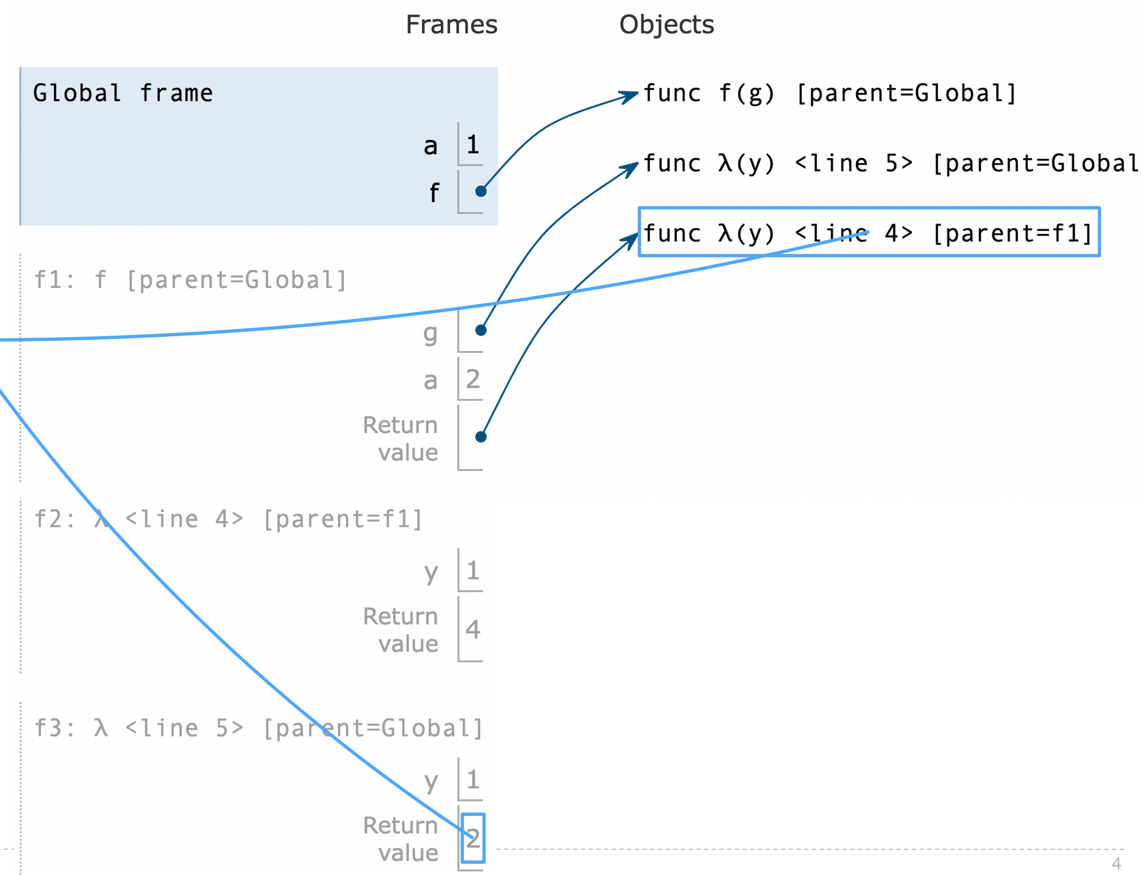


https://pythontutor.com/composingprograms.html#code=a%20%3D%201%0Adef%20f%28g%29%3A%0A%20%20%20%20a%20%3D%202%0A%20%20%20%20return%20lambda%20y%3A%20a%20*%20g%28y%29%0Af%28lambda%20y%3A%20a%20%2B%20y%29%28a%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Environment Diagrams with Lambda

```
A lambda function's parent is the current frame in which the lambda expression is evaluated
```

```
1  a = 1
2  def f(g):
3      a = 2
4      return lambda y: a * g(y)
5  f(lambda y: a + y)(a)
```



https://pythontutor.com/composingprograms.html#code=a%20%3D%201%0Adef%20f%28g%29%3A%0A%20%20%20%20a%20%3D%202%0A%20%20%20%20return%20lambda%20y%3A%20a%20*%20g%28y%29%0Af%28lambda%20y%3A%20a%20%2B%20y%29%28a%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

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

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

```
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

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

```
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

```
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

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

```
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)

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.

```
if __________:

    _________

else:

    _________
```

If Statements and Call Expressions

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

```
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.

```
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.

```
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.

```
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:
     __________
```

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:

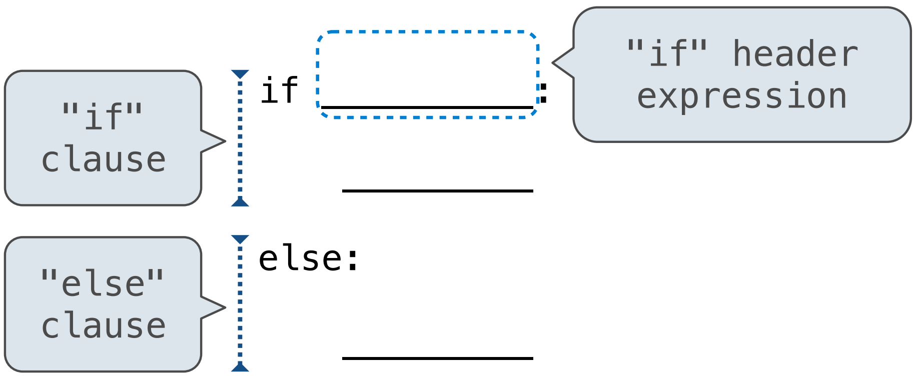
Each clause is considered in order.

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:

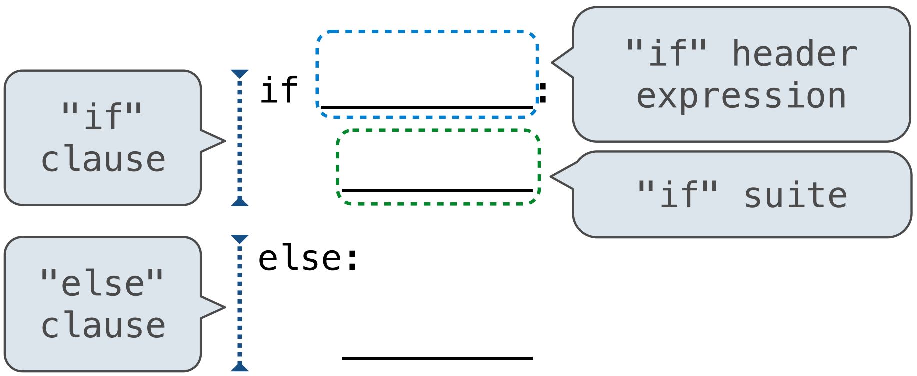
Each clause is considered in order.

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.

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:

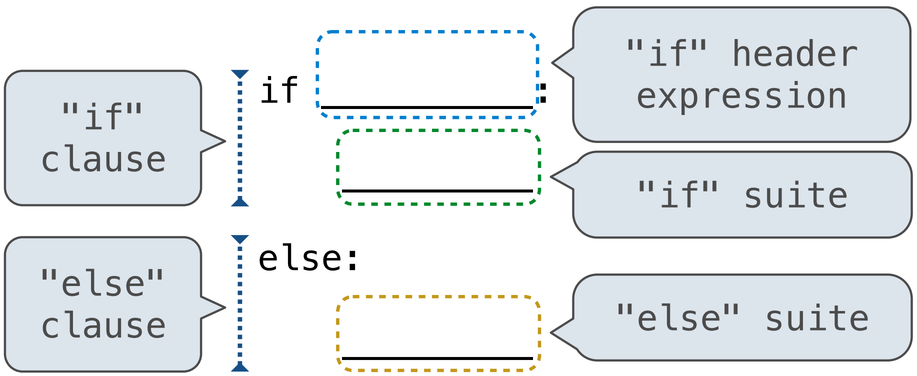
Each clause is considered in order.

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:

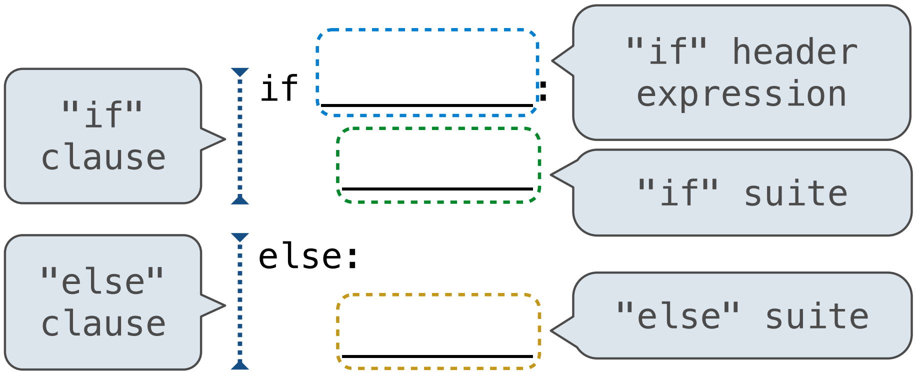
Each clause is considered in order.

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:

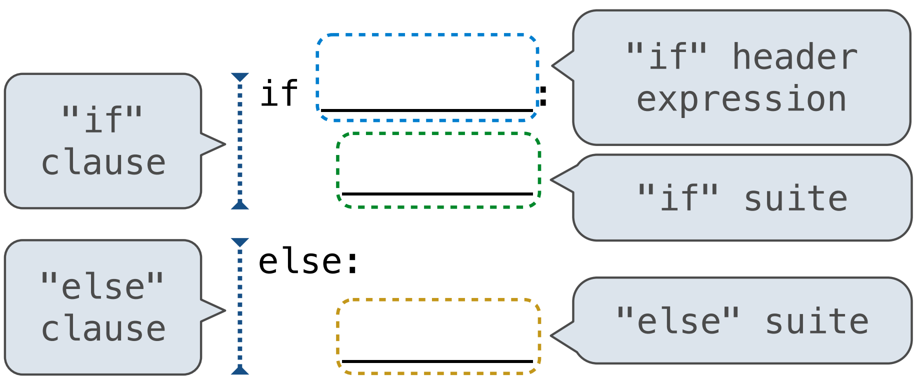
Each clause is considered in order.

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.



Execution Rule for Conditional Statements:

Each clause is considered in order.

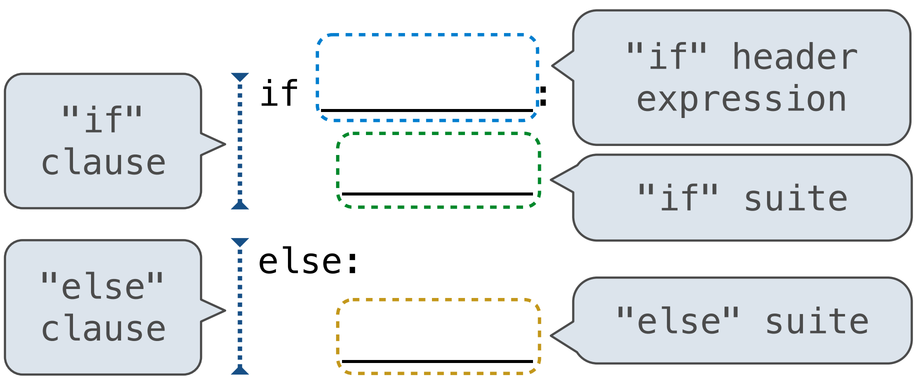
1. Evaluate the header's expression (if present).

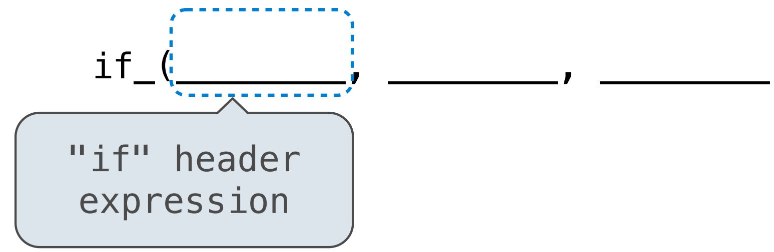
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.

```
if_(________, ________, ________)
```

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:

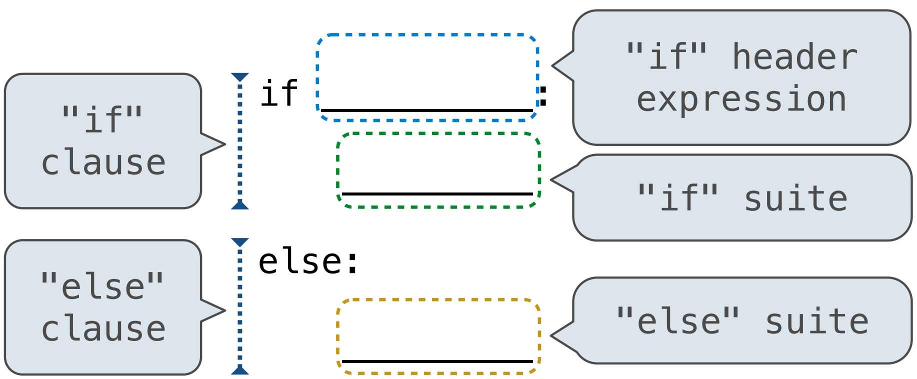
Each clause is considered in order.

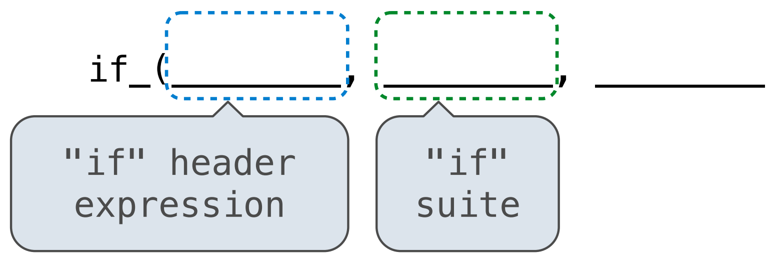
1. Evaluate the header's expression (if present).

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:

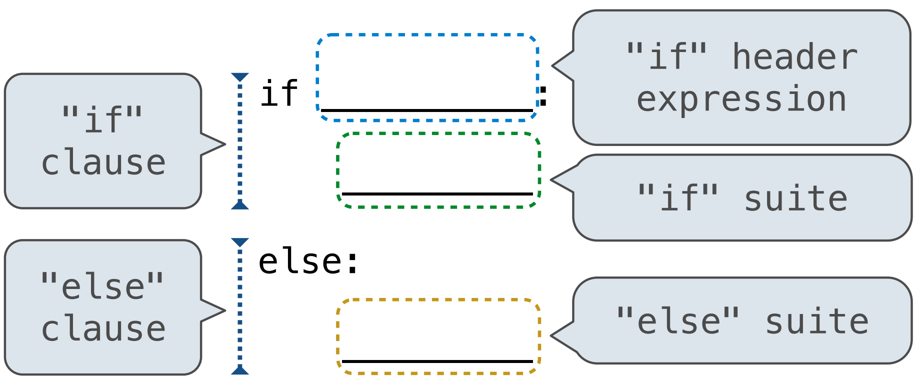
Each clause is considered in order.

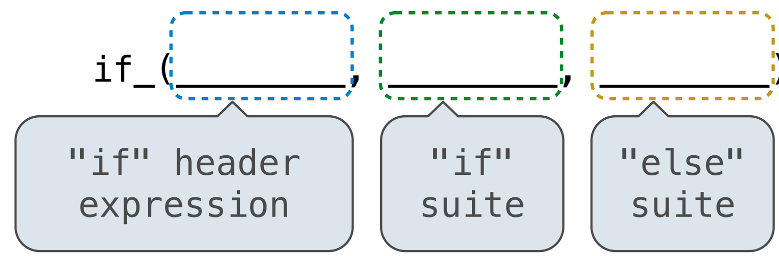
1. Evaluate the header's expression (if present).

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:

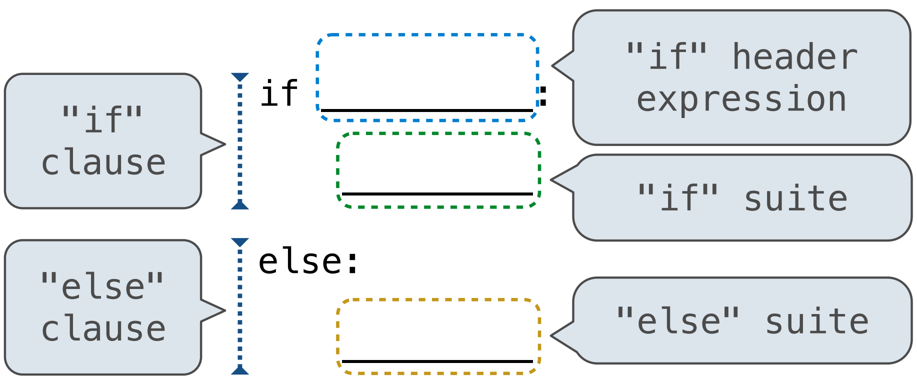
Each clause is considered in order.

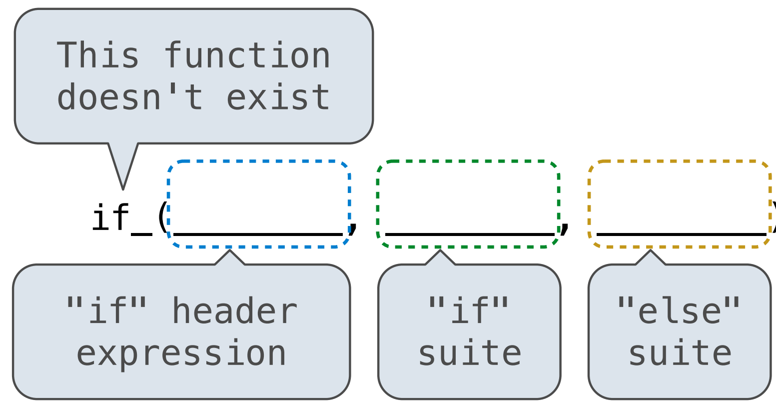
1. Evaluate the header's expression (if present).

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.





Execution Rule for Conditional Statements:

Each clause is considered in order.

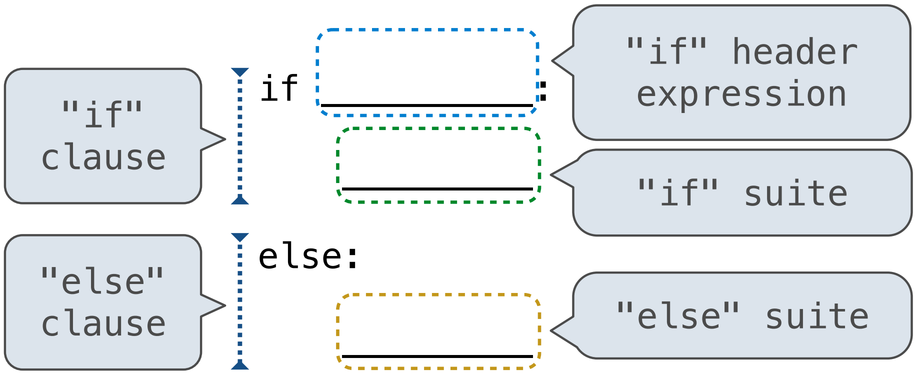
1. Evaluate the header's expression (if present).

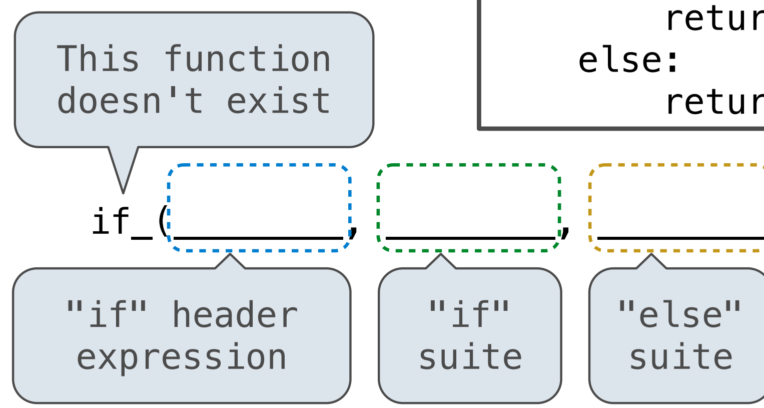
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.

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.

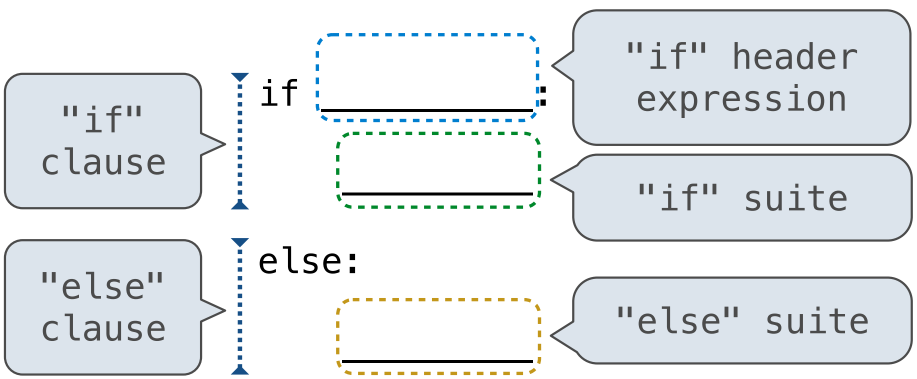
1. Evaluate the header's expression (if present).

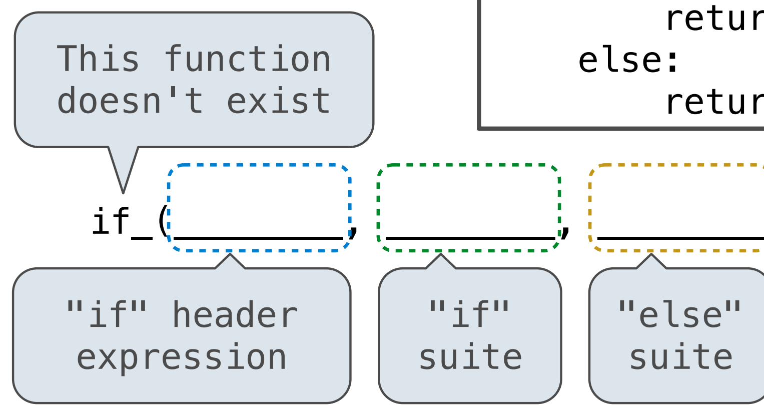
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.

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.

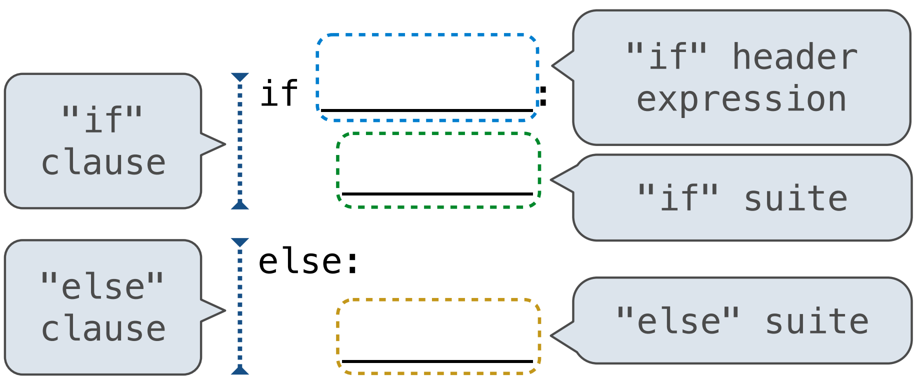
1. Evaluate the header's expression (if present).
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.

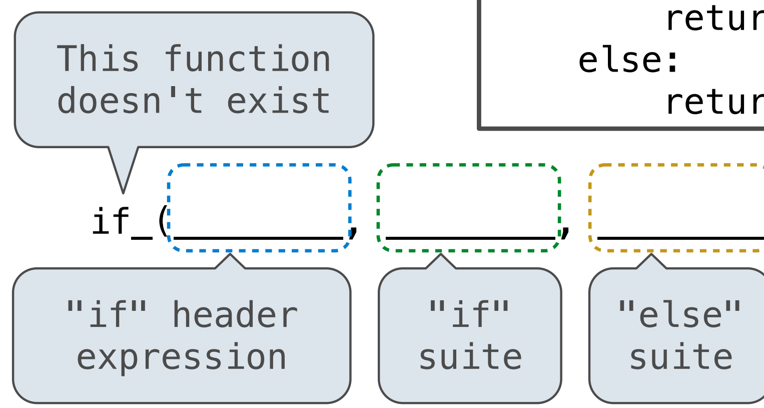
Evaluation Rule for Call Expressions:

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.

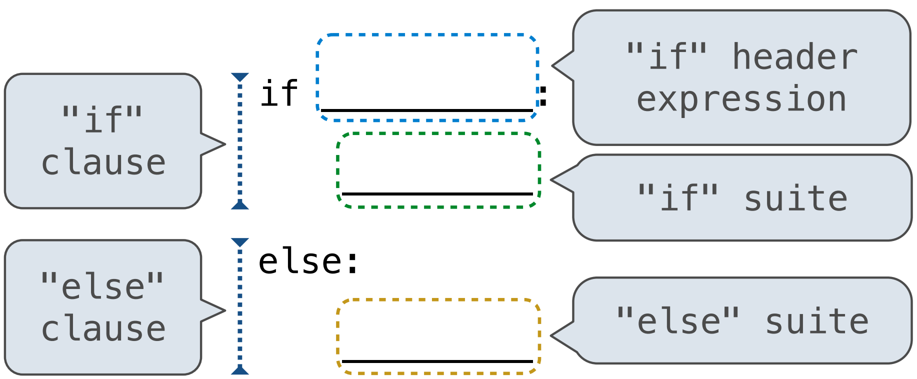
Evaluation Rule for Call Expressions:

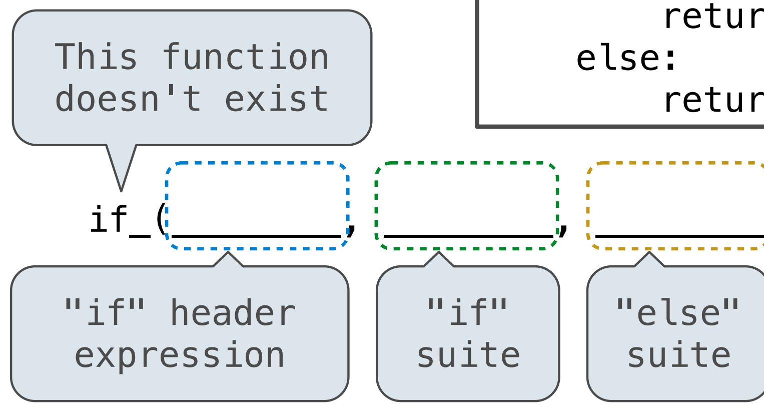
1. Evaluate the operator and then the operand subexpressions

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:

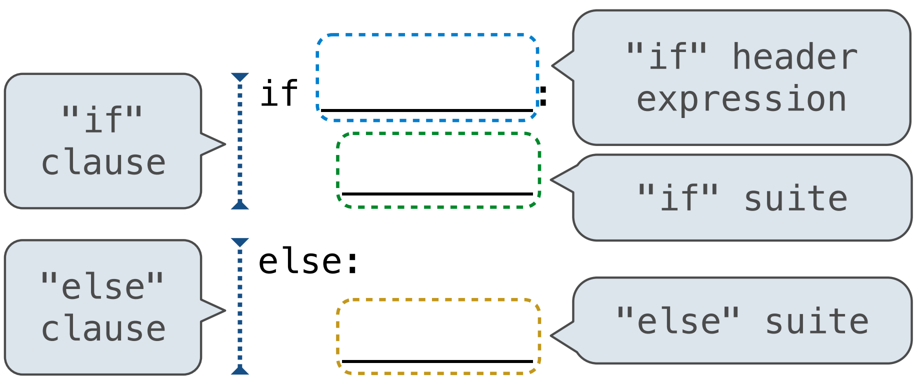
1. Evaluate the operator and then the operand subexpressions

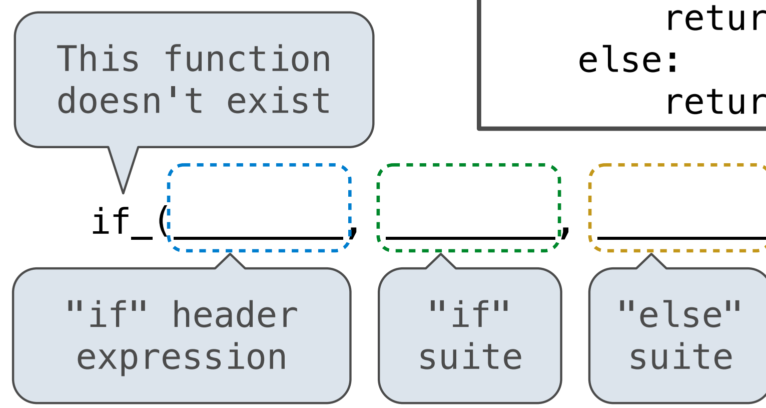
2. Apply the function that is the value of the operator to the arguments that are the values of the operands

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.

(Demo)

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 **<left>** **and** **<right>**:

1. Evaluate the subexpression **<left>**.

2. If the result is a false 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>**.

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>**:

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)

Abstraction

Functional Abstractions

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

Functional Abstractions

```python
def square(x):
    return mul(x, x)
```

```python
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument.

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square.

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number.

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul.

Functional Abstractions

```python
def square(x):
    return mul(x, x)
```

```python
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

```
def square(x):
    return pow(x, 2)
```

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

```
def square(x):
    return pow(x, 2)
```

```
def square(x):
    return mul(x, x-1) + x
```

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

```
def square(x):
    return pow(x, 2)
```

```
def square(x):
    return mul(x, x-1) + x
```

If the name "square" were bound to a built-in function, sum_squares would still work identically.

Choosing Names

Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

Names should convey the meaning or purpose
of the values to which they are bound.

Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).

Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

From:	**To:**

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).

Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

From:	**To:**
true_false	rolled_a_one

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).

Choosing Names

Names typically don't matter for correctness

but

they matter a lot for composition

From:	**To:**
true_false	rolled_a_one
d	dice

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).

Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

From:	**To:**
true_false	rolled_a_one
d	dice
helper	take_turn

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).

Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

From:	**To:**
true_false	rolled_a_one
d	dice
helper	take_turn
my_int	num_rolls

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).

Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

From:	**To:**
true_false	rolled_a_one
d	dice
helper	take_turn
my_int	num_rolls
l, I, O	k, i, m

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).

Which Values Deserve a Name

Reasons to add a new name

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

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

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

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

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

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

```
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

```
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

```
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

More Naming Tips

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

```
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

More Naming Tips

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

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

  is preferable to

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

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

```
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

More Naming Tips

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

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

  is preferable to

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

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

  ```
  n, k, i - Usually integers
  x, y, z - Usually real numbers
  f, g, h - Usually functions
  ```

Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

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

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

PRACTICAL GUIDELINES

Meaningful parts of complex expressions:

```
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

More Naming Tips

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

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

  is preferable to

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

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

  ```
  n, k, i - Usually integers
  x, y, z - Usually real numbers
  f, g, h - Usually functions
  ```

Errors & Tracebacks

Taxonomy of Errors

Syntax Errors	Detected by the Python interpreter (or editor) before the program executes
Runtime Errors	Detected by the Python interpreter while the program executes
Logic & Behavior Errors	Not detected by the Python interpreter; what tests are for

Taxonomy of Errors

Syntax Errors	Detected by the Python interpreter (or editor) before the program executes
Runtime Errors	Detected by the Python interpreter while the program executes
Logic & Behavior Errors	Not detected by the Python interpreter; what tests are for

(Demo)