Recap

Data Structures

Data Structures

Lists `[items]`

Data Structures

Lists `[items]`

Dictionaries `{key: value}`

Data Structures

Lists `[items]`

Dictionaries `{key: value}`

Tuples `(frozen, sequence)`

Data Structures

Lists `[items]`

Dictionaries `{key: value}`

Tuples `(frozen, sequence)`

Sets `{unique, hashable, values}`

Data Structures

Lists `[items]`

Dictionaries `{key: value}`

Tuples `(frozen, sequence)`

Sets `{unique, hashable, values}`

Comprehensions `[f(xs) for xs in iter]`

Familiar Functions

Recall

The def keyword is used to define a new function

```
def fn_name(param1, param2):
    value = do_something()
    return value
```

Basic Functions: Nuances

Return

Return

- All functions return *some* value

Return

- All functions return *some* value
  - Even if that value is `None`

Return

- All functions return *some* value
    - Even if that value is `None`
- No `return` statement or just `return` implicitly returns `None`

Return

- All functions return *some* value
  - Even if that value is `None`

The interpreter suppresses printing `None`

- No `return` statement or just `return` implicitly returns `None`

Return

- All functions return *some* value
  - Even if that value is **`None`**

The interpreter suppresses printing **`None`**

- No **`return`** statement or just **`return`** implicitly returns **`None`**
- Returning multiple values

Return

- All functions return *some* value
  - Even if that value is **`None`**
- No **`return`** statement or just **`return`** implicitly returns **`None`**
- Returning multiple values
  - Use a tuple!

The interpreter suppresses printing **`None`**

Return

- All functions return *some* value
    - Even if that value is `None`

The interpreter suppresses printing `None`

- No `return` statement or just `return` implicitly returns `None`
- Returning multiple values
    - Use a tuple!

```
return value1, value2, value3
```

Return

- All functions return *some* value
  - Even if that value is `None`
- No `return` statement or just `return` implicitly returns `None`
- Returning multiple values
  - Use a tuple!

```
return value1, value2, value3
```

The interpreter suppresses printing `None`

Be careful! Callers may not expect a tuple as a return value

Function Execution and Scopes

Function Execution and Scopes

- Function execution introduces a new local symbol table

Function Execution and Scopes

- Function execution introduces a new local symbol table
  - Think baggage tags and suitcases

Function Execution and Scopes

- Function execution introduces a new local symbol table
  - Think baggage tags and suitcases
- Variable assignments (L-values)

Function Execution and Scopes

- Function execution introduces a new local symbol table
  - Think baggage tags and suitcases
- Variable assignments (L-values)
  - Add entry to local symbol table

Function Execution and Scopes

- Function execution introduces a new local symbol table
  - Think baggage tags and suitcases
- Variable assignments (L-values)
  - Add entry to local symbol table
- Variable references (R-values)

Function Execution and Scopes

- Function execution introduces a new local symbol table
  - Think baggage tags and suitcases
- Variable assignments (L-values)
  - Add entry to local symbol table
- Variable references (R-values)
  - First, look in local symbol table

Function Execution and Scopes

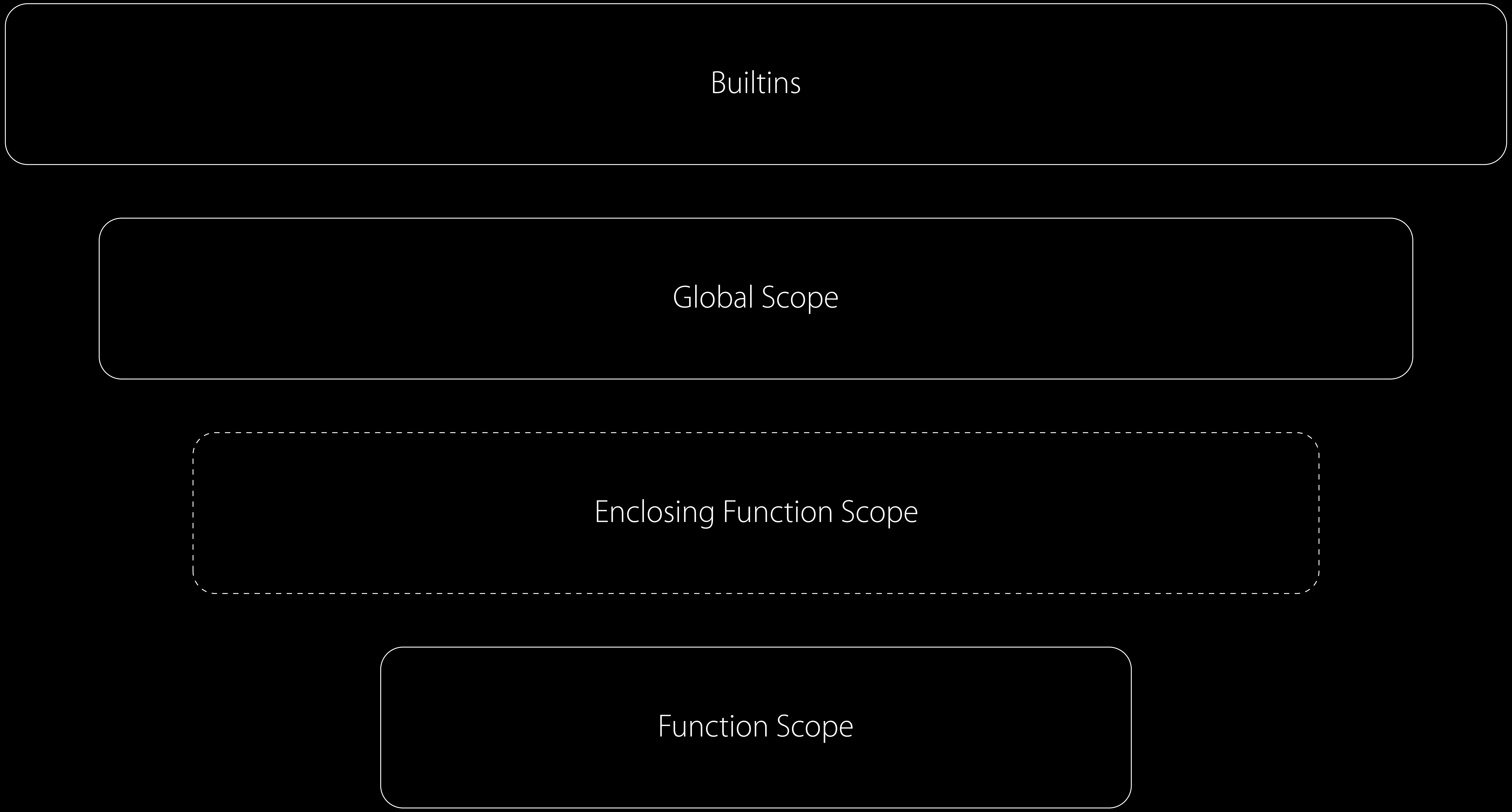
- Function execution introduces a new local symbol table
  - Think baggage tags and suitcases
- Variable assignments (L-values)
  - Add entry to local symbol table
- Variable references (R-values)
  - First, look in local symbol table
  - Next, check symbol tables of enclosing functions (unusual)

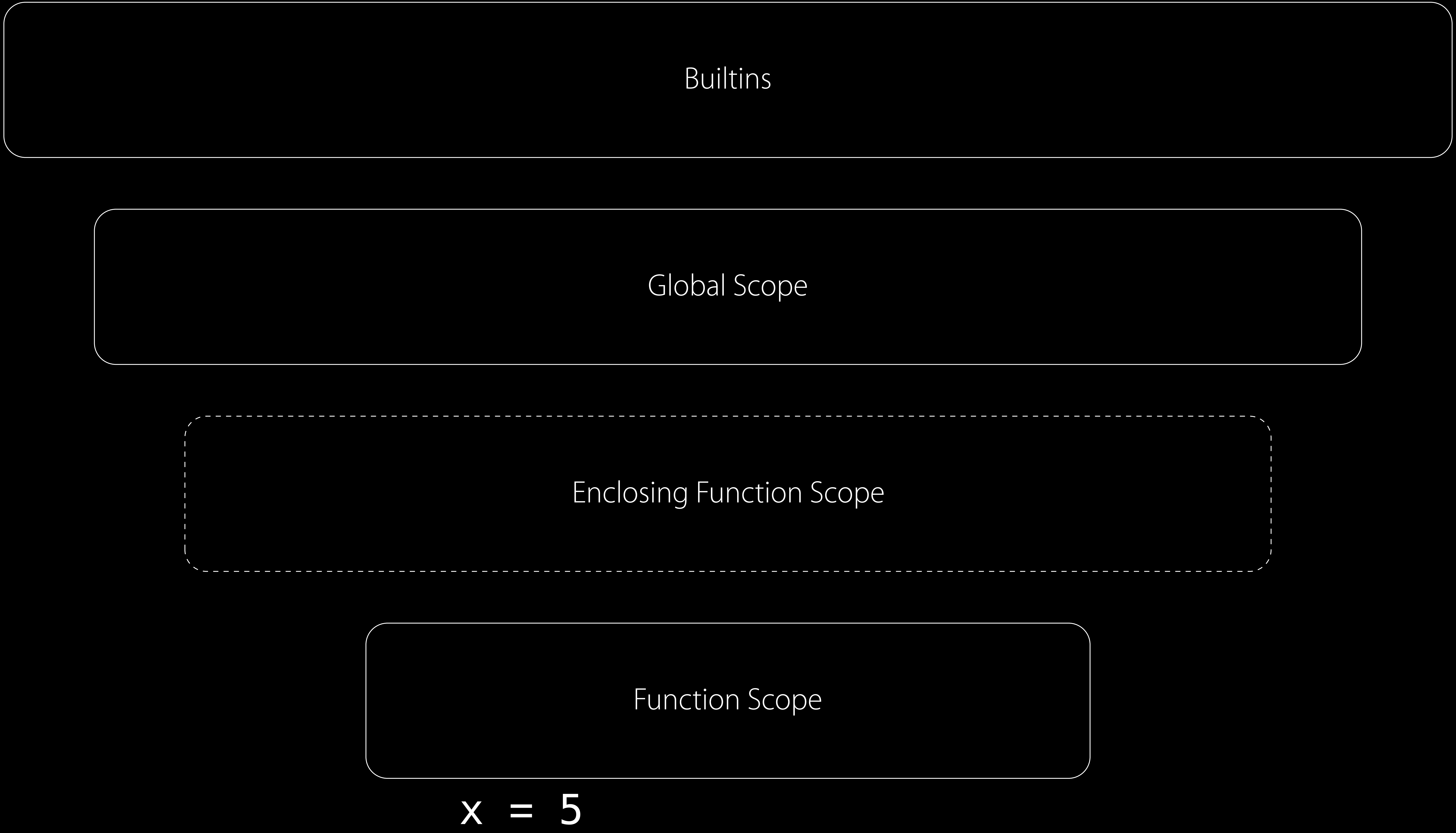
Function Execution and Scopes

- Function execution introduces a new local symbol table
  - Think baggage tags and suitcases
- Variable assignments (L-values)
  - Add entry to local symbol table
- Variable references (R-values)
  - First, look in local symbol table
  - Next, check symbol tables of enclosing functions (unusual)
  - Then, search global (top-level) symbol table

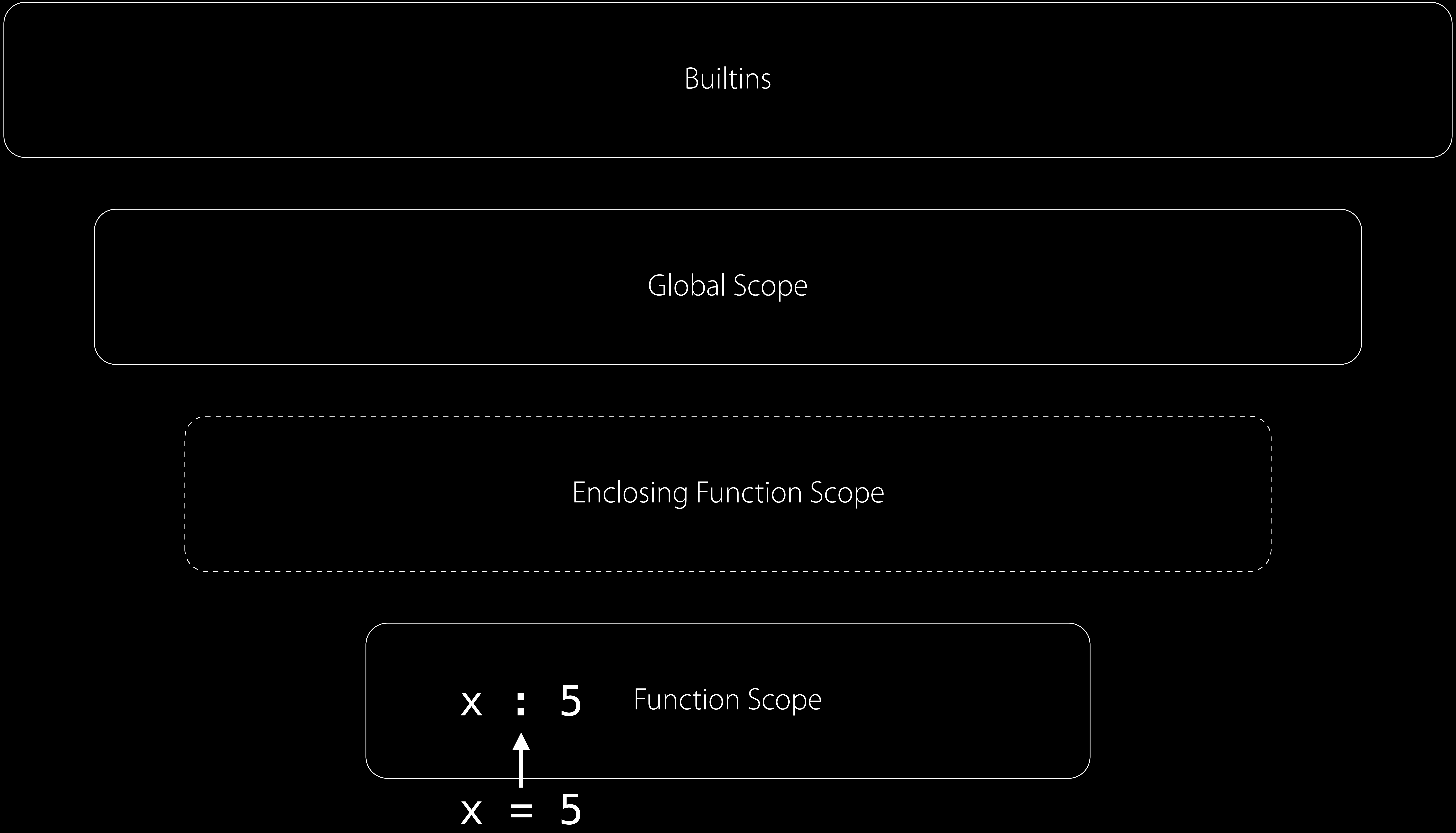
Function Execution and Scopes

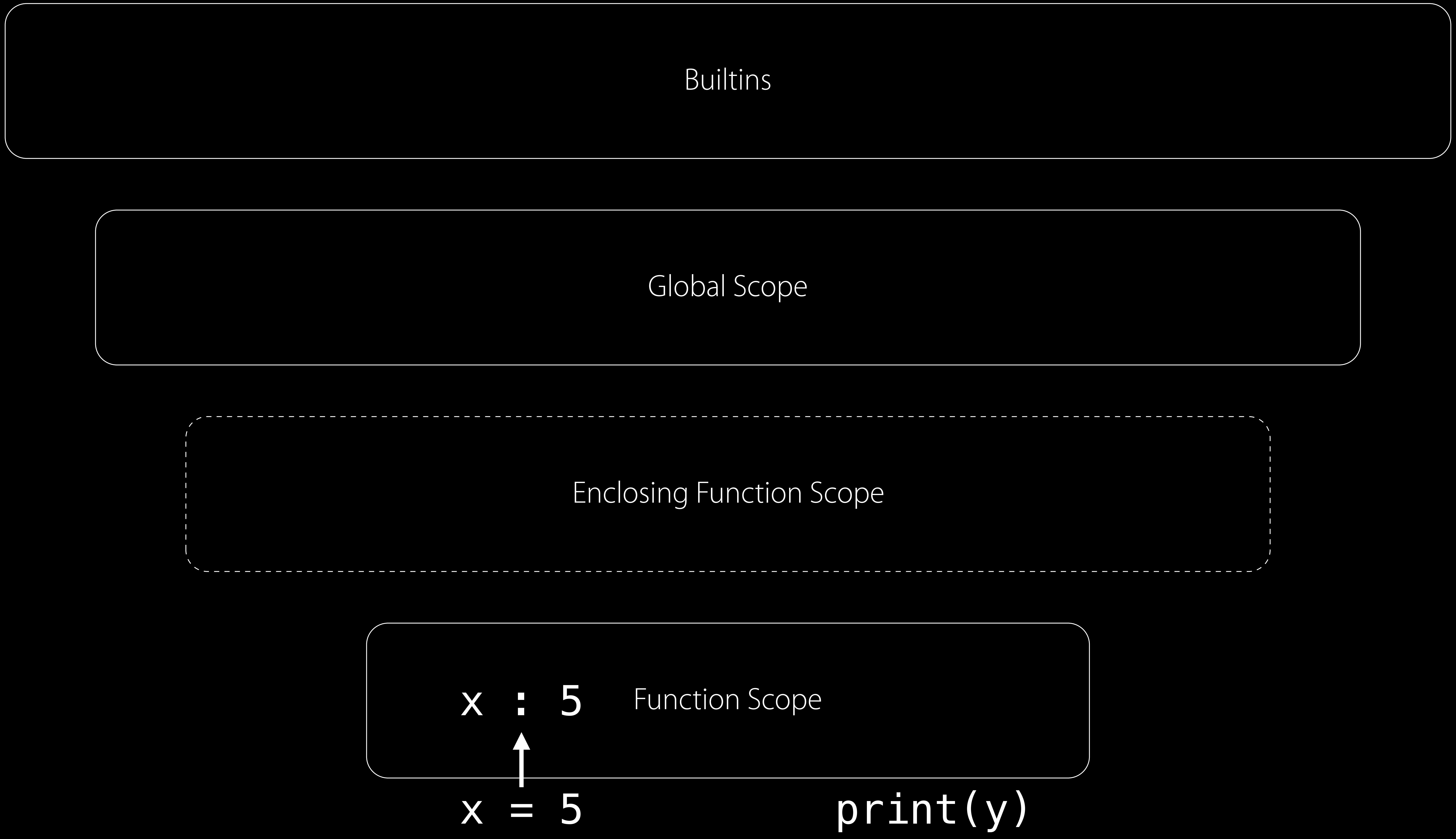
- Function execution introduces a new local symbol table
  - Think baggage tags and suitcases
- Variable assignments (L-values)
  - Add entry to local symbol table
- Variable references (R-values)
  - First, look in local symbol table
  - Next, check symbol tables of enclosing functions (unusual)
  - Then, search global (top-level) symbol table
  - Finally, check builtin symbols (`print`, `input`, etc)


Builtins
Global Scope
Enclosing Function Scope
Function Scope


Builtins
Global Scope
Enclosing Function Scope
Function Scope

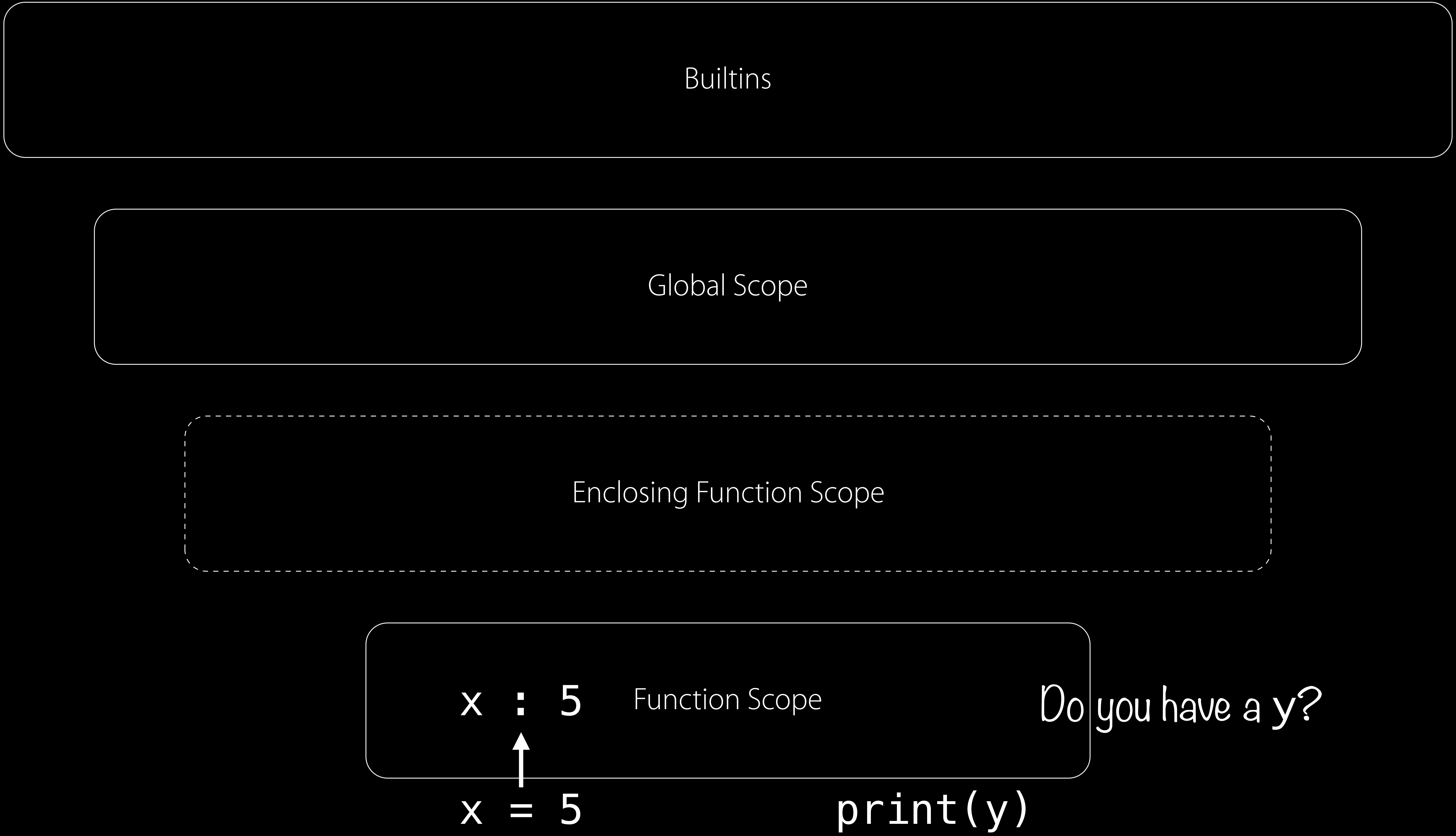
```
x = 5
```

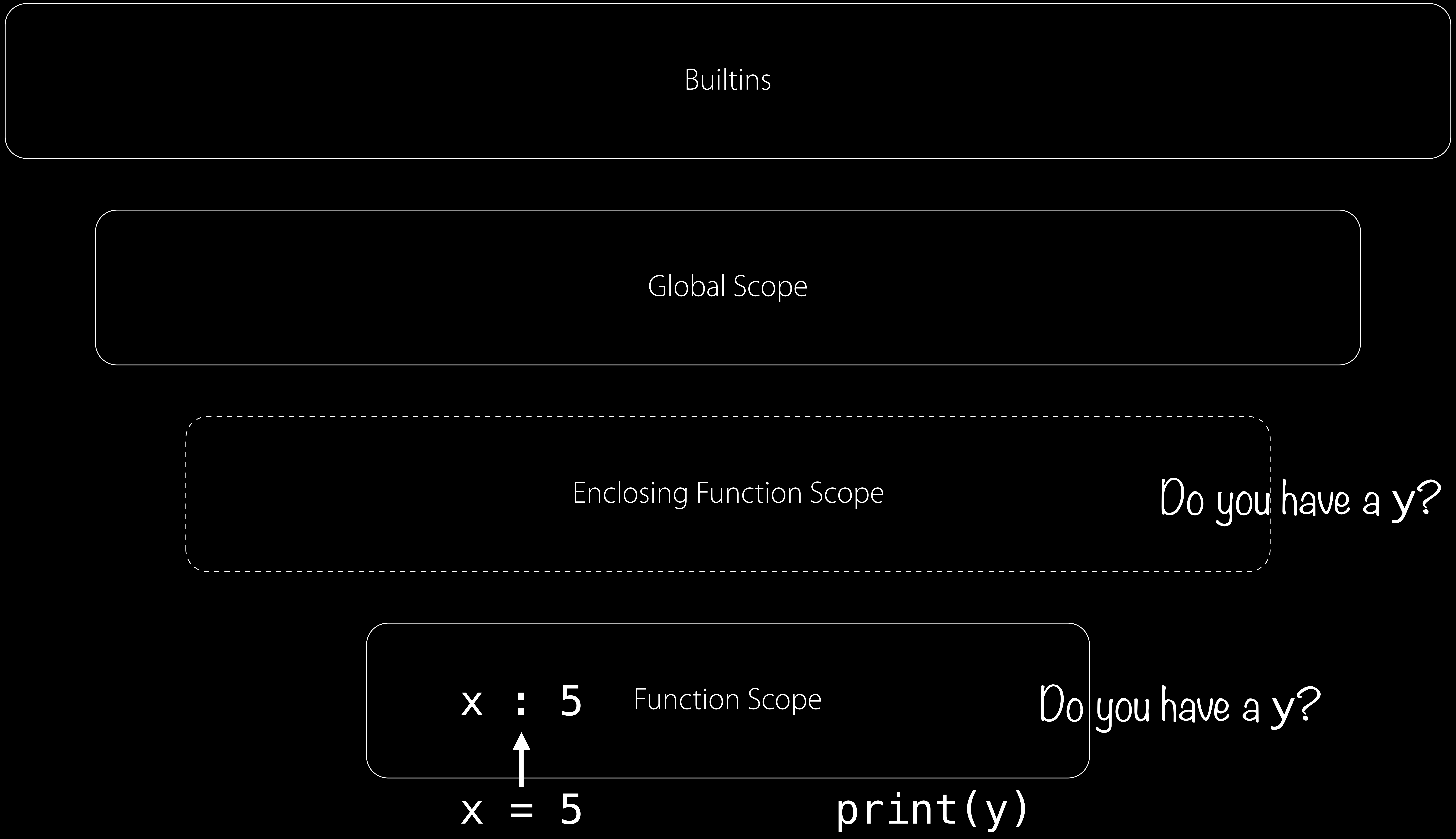

Builtins
Global Scope
Enclosing Function Scope
x : 5
Function Scope
x = 5


Builtins
Global Scope
Enclosing Function Scope
x : 5
Function Scope

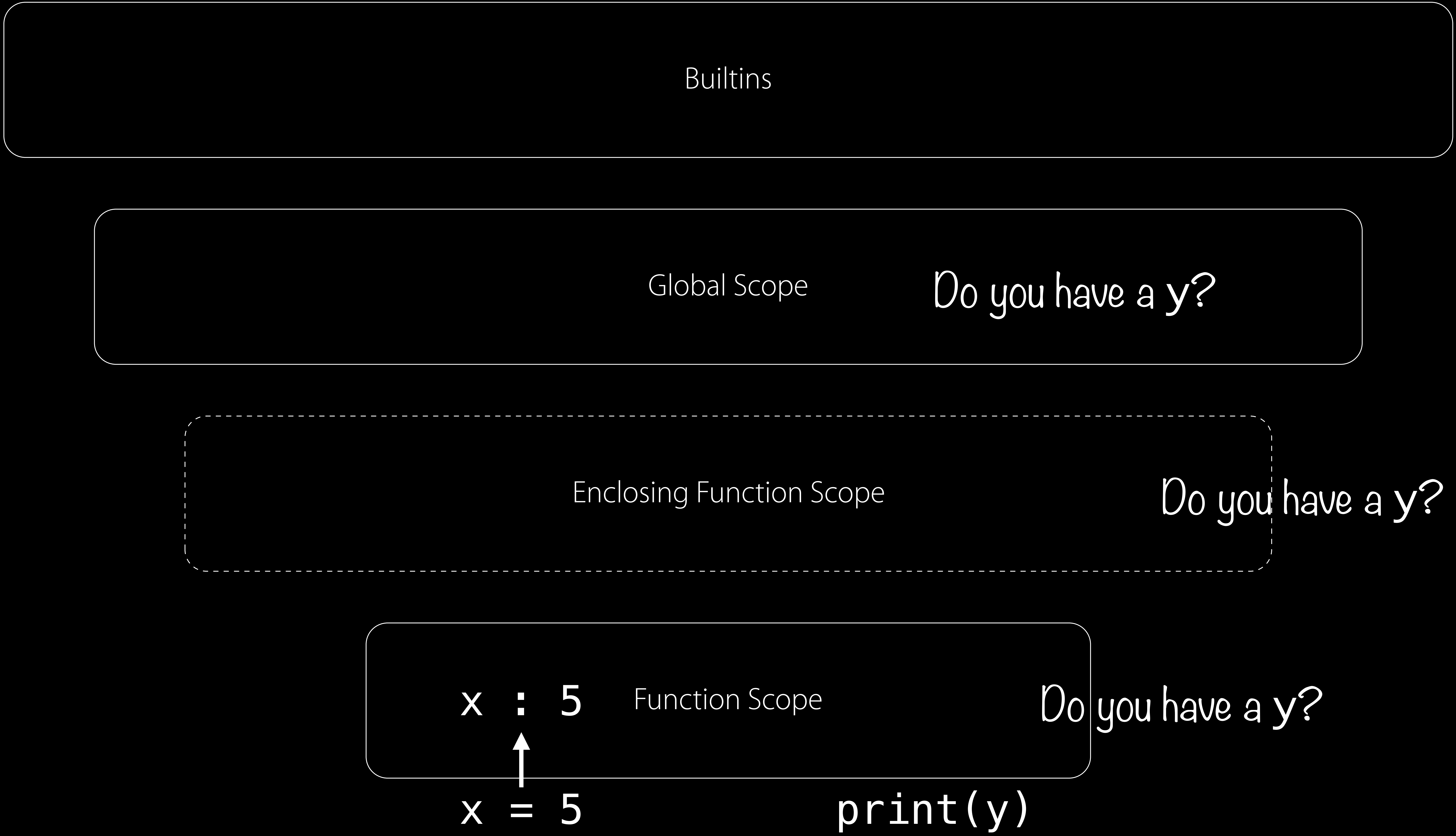
```
x = 5
```

```
print(y)
```

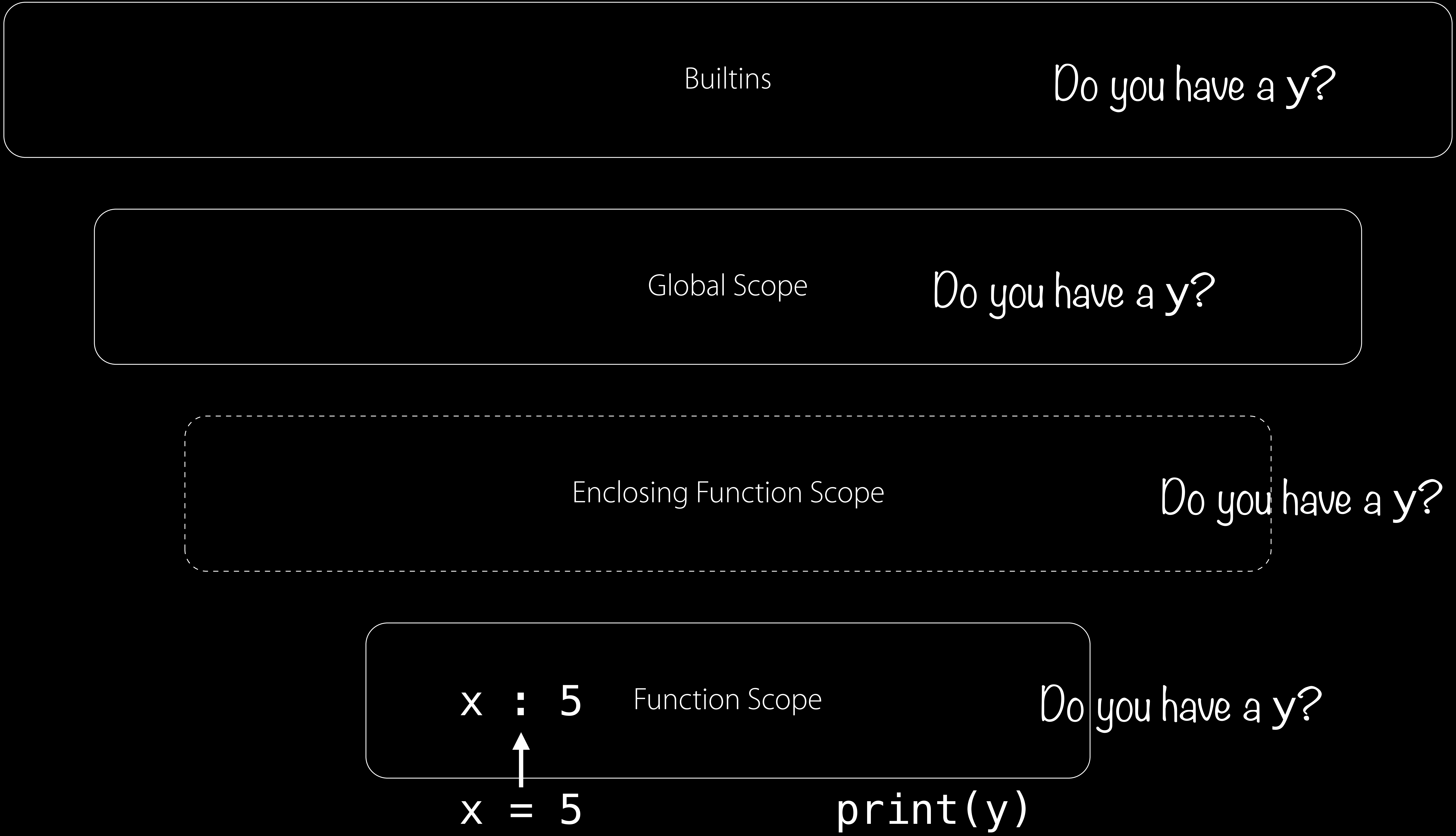

Builtins
Global Scope
Enclosing Function Scope
x : 5
Function Scope
Do you have a y?
x = 5
print(y)


Builtins
Global Scope
Enclosing Function Scope
Do you have a y?
x : 5
Function Scope
Do you have a y?

```
x = 5
print(y)
```

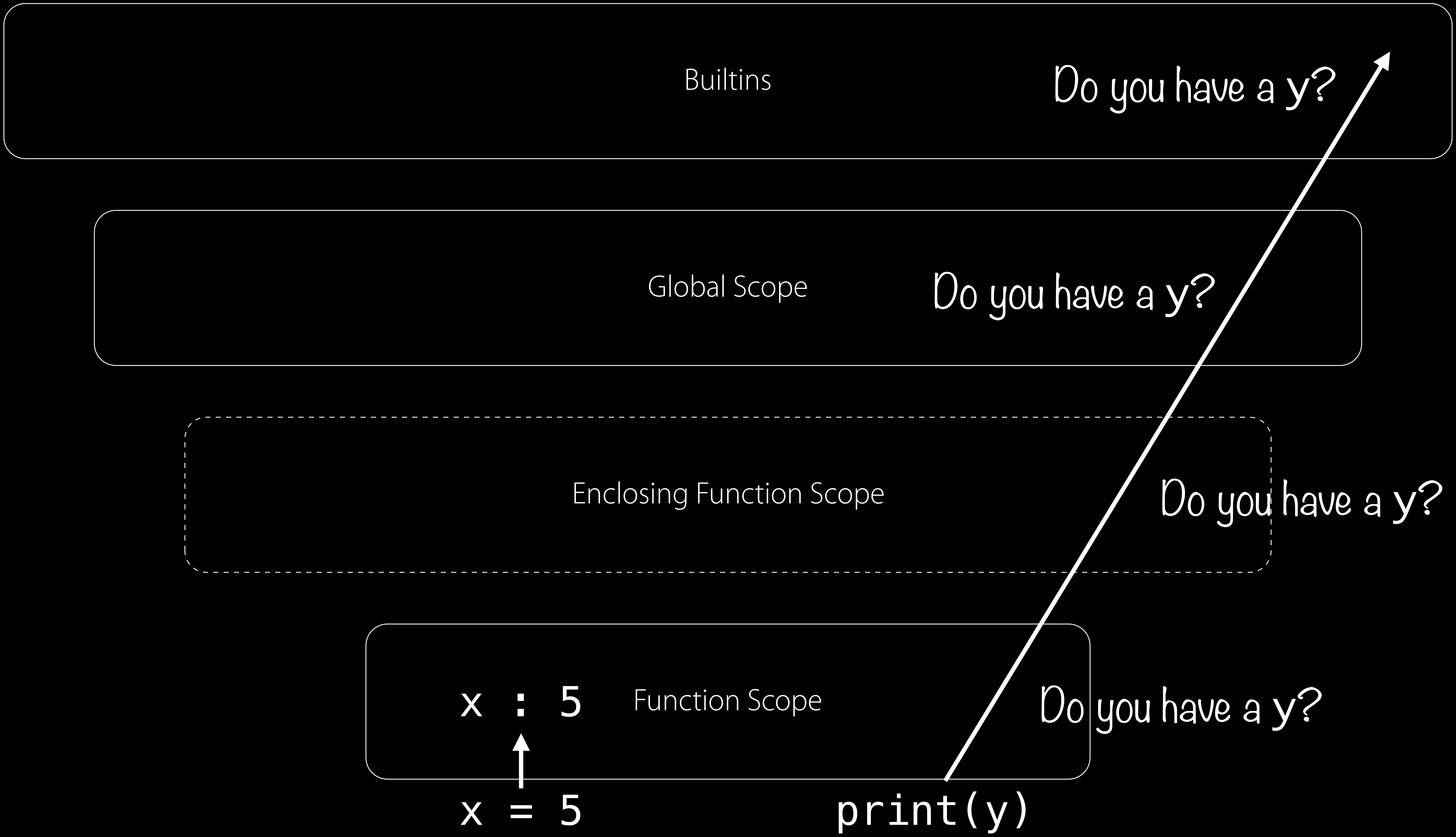

Builtins
Global Scope
Do you have a y?
Enclosing Function Scope
Do you have a y?
x : 5
Function Scope
Do you have a y?

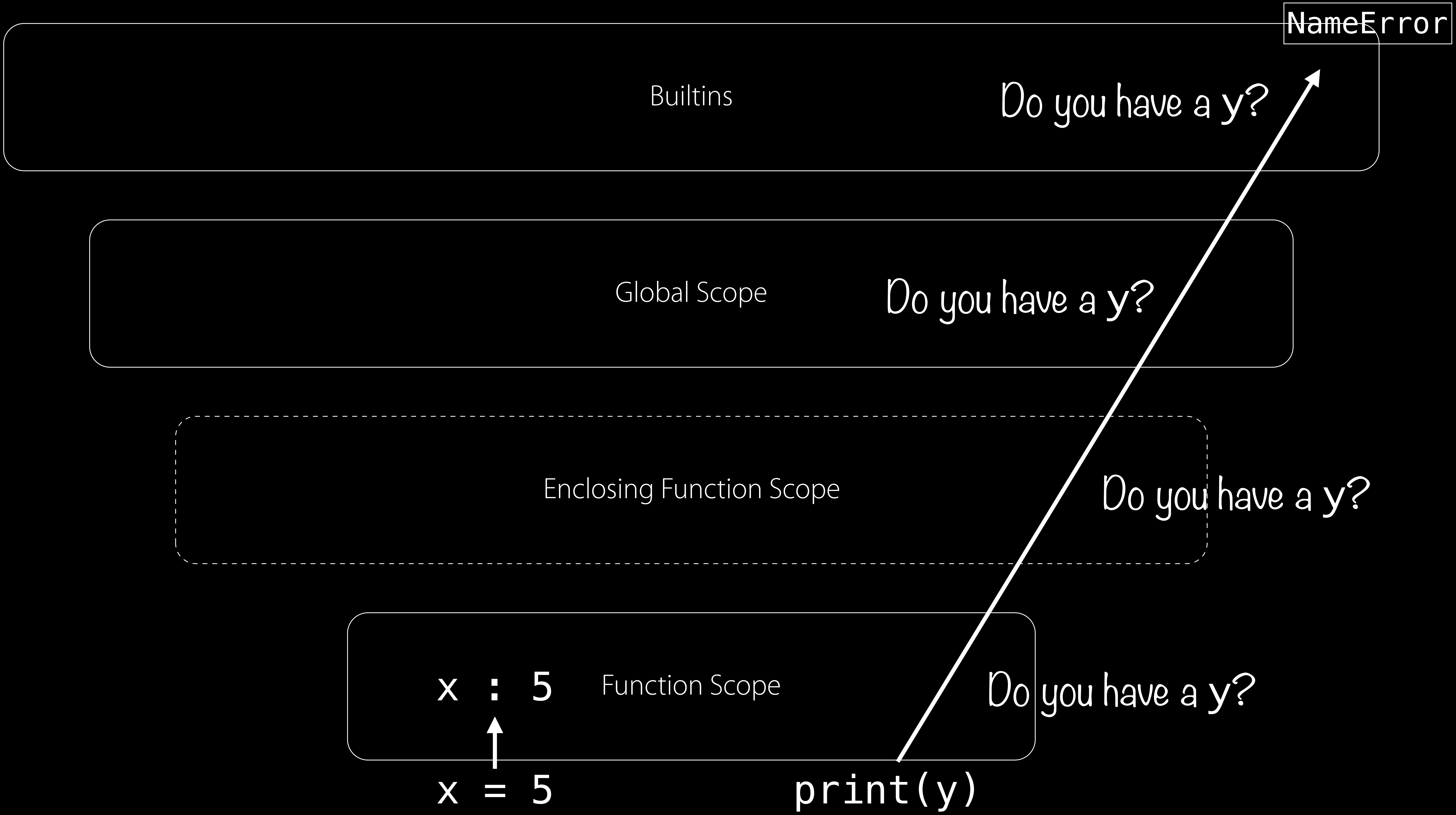
```
x = 5                print(y)
```


Builtins
Do you have a y?
Global Scope
Do you have a y?
Enclosing Function Scope
Do you have a y?
x : 5
Function Scope
Do you have a y?

```
x = 5
```

```
print(y)
```


Builtins
Do you have a y?
Global Scope
Do you have a y?
Enclosing Function Scope
Do you have a y?
x : 5
Function Scope
Do you have a y?
x = 5
print(y)


NameError
Builtins
Do you have a y?
Global Scope
Do you have a y?
Enclosing Function Scope
Do you have a y?
x : 5
Function Scope
Do you have a y?
x = 5
print(y)

If / For Scope

If / For Scope

- Notably, only* function definitions define new scopes

*Also classes... kinda (Wk 5)

If / For Scope

- Notably, only* function definitions define new scopes
- If statements, for loops, while loops, etc do *not* introduce a new scope.

*Also classes… kinda (Wk 5)

If / For Scope

- Notably, only* function definitions define new scopes
- If statements, for loops, while loops, etc do *not* introduce a new scope.

```python
if success:
    desc = 'Winner!'
else:
    desc = 'Loser :('
print(desc)
```

*Also classes... kinda (Wk 5)

Pass-By-Value or Pass-By-Reference?

Pass-By-Value or Pass-By-Reference?

- Variables *are* copied into function's local symbol table

Pass-By-Value or Pass-By-Reference?

- Variables *are* copied into function's local symbol table
    - But variables are just references to objects!

Pass-By-Value or Pass-By-Reference?

- Variables *are* copied into function's local symbol table
  - But variables are just references to objects!
- Best to think of it as *pass-by-object-reference*

Pass-By-Value or Pass-By-Reference?

- Variables *are* copied into function's local symbol table
  - But variables are just references to objects!
- Best to think of it as *pass-by-object-reference*
  - If a mutable object is passed, caller will see changes

Default Parameters

Default / Named Parameters

Default / Named Parameters

- Specify a default value for one or more parameters

Default / Named Parameters

- Specify a default value for one or more parameters
    - Called with fewer arguments than it is defined to allow

Default / Named Parameters

- Specify a default value for one or more parameters
  - Called with fewer arguments than it is defined to allow
- Usually used to provide "settings" for the function.

Default / Named Parameters

- Specify a default value for one or more parameters
    - Called with fewer arguments than it is defined to allow
- Usually used to provide "settings" for the function.
- Why?

Default / Named Parameters

- Specify a default value for one or more parameters
    - Called with fewer arguments than it is defined to allow
- Usually used to provide "settings" for the function.
- Why?
    - Present a simplified interface for a function

Default / Named Parameters

- Specify a default value for one or more parameters
    - Called with fewer arguments than it is defined to allow
- Usually used to provide "settings" for the function.
- Why?
    - Present a simplified interface for a function
    - Provide reasonable defaults for parameters

Default / Named Parameters

- Specify a default value for one or more parameters
    - Called with fewer arguments than it is defined to allow
- Usually used to provide "settings" for the function.
- Why?
    - Present a simplified interface for a function
    - Provide reasonable defaults for parameters
    - Declare intent to caller that parameters are "extra"

```python
def ask_yn(prompt,
           retries=4,
           complaint='...'):
```

Required argument `prompt`

```
def ask_yn(prompt,
           retries=4,
           complaint='...'):
```

Required argument `prompt`

```
def ask_yn(prompt,
           retries=4,
           complaint='...'):
```

Optional argument `retries` defaults to 4

Required argument `prompt`

```
def ask_yn(prompt,
           retries=4,
           complaint='...'):
```

Optional argument `retries` defaults to 4

Optional argument `complaint` defaults to 'Enter Y/N'

Keyword Arguments

Keyword Arguments

```
def ask_yn(prompt, retries=4, complaint='Enter Y/N!'):
    for i in range(retries)
        ok = input(prompt)
        if ok == 'Y':
            return True
        if ok == 'N':
            return False
        print(complaint)
    return False
```

Examples

```
print(…, sep=' ', end='\n', file=sys.stdout, flush=False)
range(start, stop, step=1)
enumerate(iter, start=0)
int(x, base=10)
pow(x, y, z=None)
seq.sort(*, key=None, reverse=None)
```

Examples

```
print(…, sep=' ', end='\n', file=sys.stdout, flush=False)
range(start, stop, step=1)
enumerate(iter, start=0)
int(x, base=10)
pow(x, y, z=None)
seq.sort(*, key=None, reverse=None)

subprocess.Popen(args, bufsize=-1, executable=None,
stdin=None, stdout=None, stderr=None, preexec_fn=None,
close_fds=True, shell=False, cwd=None, env=None,
universal_newlines=False, startupinfo=None,
creationflags=0, restore_signals=True,
start_new_session=False, pass_fds=())
```

Examples

```
print(…, sep=' ', end='\n', file=sys.stdout, flush=False)
range(start, stop, step=1)
enumerate(iter, start=0)
int(x, base=10)
pow(x, y, z=None)
seq.sort(*, key=None, reverse=None)

subprocess.Popen(args, bufsize=-1, executable=None,
stdin=None, stdout=None, stderr=None, preexec_fn=None,
close_fds=True, shell=False, cwd=None, env=None,
universal_newlines=False, startupinfo=None,
creationflags=0, restore_signals=True,
start_new_session=False, pass_fds=())
```

Wow...

```
ask_yn(prompt, retries=4, complaint='...')
```

ask_yn(prompt, retries=4, complaint='...')

```
# Call with only the mandatory argument
ask_yn('Really quit?')
ask_yn(prompt, retries=4, complaint='...')
```

```
# Call with only the mandatory argument
ask_yn('Really quit?')

# Call with one keyword argument
ask_yn('OK to overwrite the file?', retries=2)
```

ask_yn(prompt, retries=4, complaint='...')

```
# Call with only the mandatory argument
ask_yn('Really quit?')

# Call with one keyword argument
ask_yn('OK to overwrite the file?', retries=2)

# Call with one keyword argument - in any order!
ask_yn('Update status?', complaint='Just Y/N')
```

ask_yn(prompt, retries=4, complaint='...')

```
# Call with only the mandatory argument
ask_yn('Really quit?')

# Call with one keyword argument
ask_yn('OK to overwrite the file?', retries=2)

# Call with one keyword argument - in any order!
ask_yn('Update status?', complaint='Just Y/N')

# Call with all of the keyword arguments
ask_yn('Send text?', retries=2, complaint='Y/N please!')
```

Rules about Function Calls

Rules about Function Calls

- Keyword arguments must follow positional arguments

Rules about Function Calls

- Keyword arguments must follow positional arguments
- All keyword arguments must identify some parameter

Rules about Function Calls

- Keyword arguments must follow positional arguments
- All keyword arguments must identify some parameter
  - Even positional ones

Rules about Function Calls

- Keyword arguments must follow positional arguments
- All keyword arguments must identify some parameter
  - Even positional ones
- No parameter may receive a value more than once

Rules about Function Calls

- Keyword arguments must follow positional arguments
- All keyword arguments must identify some parameter
    - Even positional ones
- No parameter may receive a value more than once

```python
def fn(a): pass
fn(0, a=0)
# Not allowed! Multiple values for a
```

Variadic Positional Arguments

Variadic Positional Arguments

Variadic Positional Arguments

- A parameter of form `*args` captures excess positional args

Variadic Positional Arguments

- A parameter of form `*args` captures excess positional args
    - These excess arguments are bundled into an `args` tuple

Variadic Positional Arguments

- A parameter of form `*args` captures excess positional args
    - These excess arguments are bundled into an `args` tuple
- Why?

Variadic Positional Arguments

- A parameter of form `*args` captures excess positional args
  - These excess arguments are bundled into an `args` tuple
- Why?
  - Call functions with any number of positional arguments

Variadic Positional Arguments

- A parameter of form `*args` captures excess positional args
    - These excess arguments are bundled into an `args` tuple
- Why?
    - Call functions with any number of positional arguments
    - Capture all arguments to forward to another handler

Variadic Positional Arguments

- A parameter of form `*args` captures excess positional args
    - These excess arguments are bundled into an `args` tuple
- Why?
    - Call functions with any number of positional arguments
    - Capture all arguments to forward to another handler
        - Used in subclasses, proxies, and decorators

Variadic Positional Arguments

- A parameter of form **`*args`** captures excess positional args
  - These excess arguments are bundled into an **`args`** tuple
- Why?
  - Call functions with any number of positional arguments
  - Capture all arguments to forward to another handler
    - Used in subclasses, proxies, and decorators

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Variadic Positional Arguments

Variadic Positional Arguments

```
# Suppose we want a product function that works as so:
product(3, 5)  # => 15
product(3, 4, 2)  # => 24
product(3, 5, scale=10)  # => 150
```

Variadic Positional Arguments

```
# Suppose we want a product function that works as so:
product(3, 5)  # => 15
product(3, 4, 2)  # => 24
product(3, 5, scale=10)  # => 150


# product accepts any number of arguments
def product(*nums, scale=1):
```

Variadic Positional Arguments

```python
# Suppose we want a product function that works as so:
product(3, 5)  # => 15
product(3, 4, 2)  # => 24
product(3, 5, scale=10)  # => 150


# product accepts any number of arguments
def product(*nums, scale=1):
    p = scale
    for n in nums:
        p *= n
    return p
```

Variadic Positional Arguments

```python
# Suppose we want a product function that works as so:
product(3, 5)  # => 15
product(3, 4, 2)  # => 24
product(3, 5, scale=10)  # => 150


# product accepts any number of arguments
def product(*nums, scale=1):
    p = scale
    for n in nums:
        p *= n
    return p
```

Named parameters after `*args` are 'keyword-only' arguments

Unpacking Variadic Positional Arguments

Unpacking Variadic Positional Arguments

```
# Suppose we want to find 2 * 3 * 5 * 7 * … up to 100
def is_prime(n):  pass  # Some implementation
```

Unpacking Variadic Positional Arguments

```
# Suppose we want to find 2 * 3 * 5 * 7 * … up to 100
def is_prime(n):  pass  # Some implementation

# Extract all the primes
primes = [number for number in range(2, 100)
          if is_prime(number)]
```

Unpacking Variadic Positional Arguments

```python
# Suppose we want to find 2 * 3 * 5 * 7 * … up to 100
def is_prime(n):  pass  # Some implementation

# Extract all the primes
primes = [number for number in range(2, 100)
          if is_prime(number)]

print(product(*primes))  # equiv. to product(2, 3, 5, …)
```

Unpacking Variadic Positional Arguments

```python
# Suppose we want to find 2 * 3 * 5 * 7 * … up to 100
def is_prime(n):  pass  # Some implementation

# Extract all the primes
primes = [number for number in range(2, 100)
          if is_prime(number)]

print(product(*primes))  # equiv. to product(2, 3, 5, …)
```

The syntax `*seq` unpacks a sequence into its constituent components

Variadic Keyword Arguments

Variadic Keyword Arguments

Variadic Keyword Arguments

- A parameter of the form `**kwargs` captures all excess keyword arguments

Variadic Keyword Arguments

- A parameter of the form `**kwargs` captures all excess keyword arguments
    - These excess arguments are bundled into a `kwargs` dict

Variadic Keyword Arguments

- A parameter of the form `**kwargs` captures all excess keyword arguments
    - These excess arguments are bundled into a `kwargs` dict
- Why?

Variadic Keyword Arguments

- A parameter of the form `**kwargs` captures all excess keyword arguments
  - These excess arguments are bundled into a `kwargs` dict
- Why?
  - Allow arbitrary named parameters, usually for configuration

Variadic Keyword Arguments

- A parameter of the form `**kwargs` captures all excess keyword arguments
  - These excess arguments are bundled into a `kwargs` dict
- Why?
  - Allow arbitrary named parameters, usually for configuration
  - Similar: capture all arguments to forward to another handler

Variadic Keyword Arguments

- A parameter of the form `**kwargs` captures all excess keyword arguments
  - These excess arguments are bundled into a `kwargs` dict
- Why?
  - Allow arbitrary named parameters, usually for configuration
  - Similar: capture all arguments to forward to another handler
    - Used in subclasses, proxies, and decorators

Variadic Keyword Arguments

Variadic Keyword Arguments

```python
def authorize(quote, **speaker_info):
    print(">", quote)
    print("-" * (len(quote) + 2))
    for k, v in speaker_info.items():
        print(k, v, sep=': ')
```

Calling Variadic Keyword Arguments

Calling Variadic Keyword Arguments

```
authorize(
    "If music be the food of love, play on.",
    playwright="Shakespeare",
    act=1,
    scene=1,
    speaker="Duke Orsino"
)
```

Calling Variadic Keyword Arguments

```
authorize(
    "If music be the food of love, play on.",
    playwright="Shakespeare",
    act=1,
    scene=1,
    speaker="Duke Orsino"
)
```

```
speaker_info = {
  'act': 1,
  'scene': 1,
  'speaker': "Duke Orsino",
  'playwright': "Shakespeare"
}
```

Calling Variadic Keyword Arguments

```
authorize(
    "If music be the food of love, play on.",
    playwright="Shakespeare",
    act=1,
    scene=1,
    speaker="Duke Orsino"
)
```

```
speaker_info = {
  'act': 1,
  'scene': 1,
  'speaker': "Duke Orsino",
  'playwright': "Shakespeare"
}
```

```
# > If music be the food of love, play on.
# ----------------------------------------
# act: 1
# scene: 1
# speaker: Duke Orsino
# playwright: Shakespeare
```

Example: Formatting Strings

```
fstr.format(*args, **kwargs)
```

All positional arguments go into `args`

```
fstr.format(*args, **kwargs)
```

All positional arguments go into `args`

```
fstr.format(*args, **kwargs)
```

All keyword arguments go into `kwargs`

fstr.format(*args, **kwargs)

fstr.format(*args, **kwargs)

```
# {n} refers to the nth positional argument in `args`
"First, thou shalt count to {0}".format(3)
```

fstr.format(*args, **kwargs)

```
# {n} refers to the nth positional argument in `args`
"First, thou shalt count to {0}".format(3)          args = (3, )
```

fstr.format(*args, **kwargs)

```
# {n} refers to the nth positional argument in `args`
"First, thou shalt count to {0}".format(3)          args = (3, )
"{0} shalt thou not count, neither count thou {1},
excepting that thou then proceed to {2}".format(4, 2, 3)
```

fstr.format(*args, **kwargs)

```
# {n} refers to the nth positional argument in `args`
"First, thou shalt count to {0}".format(3)        args = (3, )
"{0} shalt thou not count, neither count thou {1},
excepting that thou then proceed to {2}".format(4, 2, 3)
                                                args = (4, 2, 3)
```

fstr.format(*args, **kwargs)

```
# {n} refers to the nth positional argument in `args`
"First, thou shalt count to {0}".format(3)          args = (3, )
"{0} shalt thou not count, neither count thou {1},
excepting that thou then proceed to {2}".format(4, 2, 3)
                                                  args = (4, 2, 3)

# {key} refers to the optional argument bound by key
"lobbest thou thy {weapon} towards thy foe".format(
    weapon="Holy Hand Grenade of Antioch"
)
```

fstr.format(*args, **kwargs)

```python
# {n} refers to the nth positional argument in `args`
"First, thou shalt count to {0}".format(3)        args = (3, )
"{0} shalt thou not count, neither count thou {1},
excepting that thou then proceed to {2}".format(4, 2, 3)
                                               args = (4, 2, 3)

# {key} refers to the optional argument bound by key
"lobbest thou thy {weapon} towards thy foe".format(
    weapon="Holy Hand Grenade of Antioch"
)
        kwargs = {"weapon": "Holy Hand Grenade of Antioch"}
```

Complicated Example

Complicated Example

```
"{0}{b}{1}{a}{0}{2}".format(
    5, 8, 9, a='z', b='x'
)
```

Complicated Example

```
"{0}{b}{1}{a}{0}{2}".format(
    5, 8, 9, a='z', b='x'
)
```

```
args = (5, 8, 9)
kwargs = {'a':'z', 'b':'x'}
```

Complicated Example

```
"{0}{b}{1}{a}{0}{2}".format(
    5, 8, 9, a='z', b='x'
)
# => 5x8z59
```

```
args = (5, 8, 9)
kwargs = {'a':'z', 'b':'x'}
```

Cute Trick: Unpacking Variadic Keyword Arguments

```
x = 3
foo = 'fighter'
y = 4
bar = 'bell'
z = 5
```

Cute Trick: Unpacking Variadic Keyword Arguments

```
x = 3

foo = 'fighter'

y = 4

bar = 'bell'

z = 5
```

local symbol table

```
{
    'x': 3,
    'foo': 'fighter',
    'y': 4,
    'bar': 'bell',
    'z': 5, …
}
```

Cute Trick: Unpacking Variadic Keyword Arguments

```
x = 3
foo = 'fighter'
y = 4
bar = 'bell'
z = 5
```

local symbol table

```
{
    'x': 3,
    'foo': 'fighter',
    'y': 4,
    'bar': 'bell',
    'z': 5, …
}
```

```
print("{z}^2 = {x}^2 + {y}^2".format(x=x, y=y, z=z))
```

Cute Trick: Unpacking Variadic Keyword Arguments

```
x = 3
foo = 'fighter'
y = 4
bar = 'bell'
z = 5
```

local symbol table

```
{
    'x': 3,
    'foo': 'fighter',
    'y': 4,
    'bar': 'bell',
    'z': 5, …
}
```

```
print("{z}^2 = {x}^2 + {y}^2".format(x=x, y=y, z=z))
print("{z}^2 = {x}^2 + {y}^2".format(**locals()))
```

Cute Trick: Unpacking Variadic Keyword Arguments

```
x = 3
foo = 'fighter'
y = 4
bar = 'bell'
z = 5
```

local symbol table

```
{
    'x': 3,
    'foo': 'fighter',
    'y': 4,
    'bar': 'bell',
    'z': 5, …
}
```

```
print("{z}^2 = {x}^2 + {y}^2".format(x=x, y=y, z=z))
print("{z}^2 = {x}^2 + {y}^2".format(**locals()))
# Equivalent to .format(x=3, foo='fighter', y=4, ...)
```

Cute Trick: Unpacking Variadic Keyword Arguments

```python
x = 3
foo = 'fighter'
y = 4
bar = 'bell'
z = 5
```

local symbol table

```
{
    'x': 3,
    'foo': 'fighter',
    'y': 4,
    'bar': 'bell',
    'z': 5, …
}
```

```python
print("{z}^2 = {x}^2 + {y}^2".format(x=x, y=y, z=z))
print("{z}^2 = {x}^2 + {y}^2".format(**locals()))
# Equivalent to .format(x=3, foo='fighter', y=4, ...)
```

Usually slow... and bad style, but can be useful for debugging!

Putting it All Together

A Valid Python Function Definition

```python
def foo(a, b, c=1, *d, e=1, **f)
```

A Valid Python Function Definition

Mandatory positional arguments

```
def foo(a, b, c=1, *d, e=1, **f)
```

A Valid Python Function Definition

Mandatory positional arguments

```
def foo(a, b, c=1, *d, e=1, **f)
```

Optional keyword argument

A Valid Python Function Definition

Variadic positional argument list – scoops up excess positional args into a tuple

Mandatory positional arguments

```python
def foo(a, b, c=1, *d, e=1, **f)
```

Optional keyword argument

A Valid Python Function Definition

Mandatory positional arguments

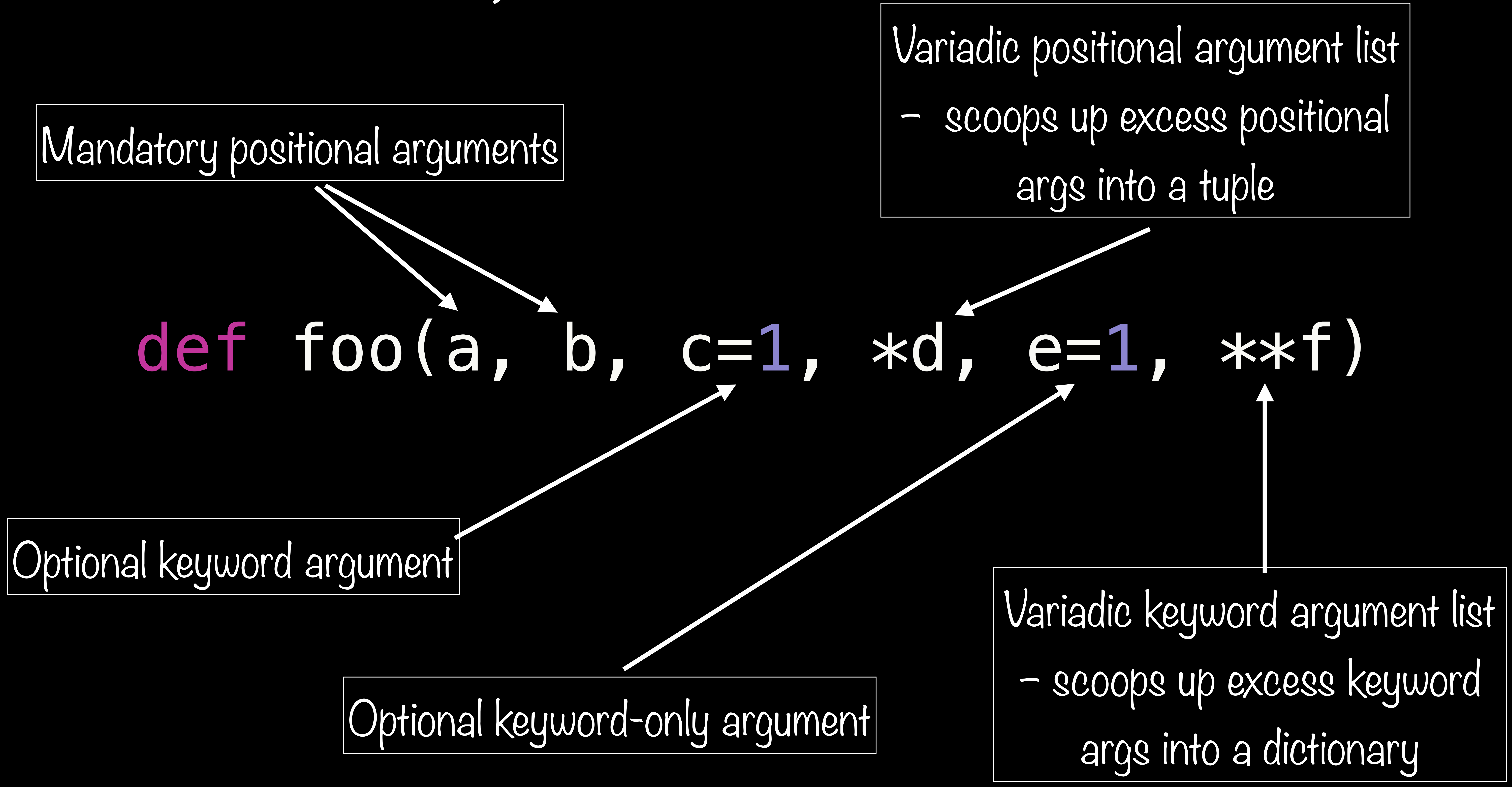
Variadic positional argument list – scoops up excess positional args into a tuple

```python
def foo(a, b, c=1, *d, e=1, **f)
```

Optional keyword argument

Optional keyword-only argument

A Valid Python Function Definition



```python
def foo(a, b, c=1, *d, e=1, **f)
```

Mandatory positional arguments
Variadic positional argument list – scoops up excess positional args into a tuple
Optional keyword argument
Optional keyword-only argument
Variadic keyword argument list – scoops up excess keyword args into a dictionary

Aside: Code Style

Function Comments

Function Comments

- The first string literal *inside* a function body is a docstring

Function Comments

- The first string literal *inside* a function body is a docstring
    - First line: one-line summary of the function

Function Comments

- The first string literal *inside* a function body is a docstring
    - First line: one-line summary of the function
    - Subsequent lines: extended description of function

Function Comments

- The first string literal *inside* a function body is a docstring
    - First line: one-line summary of the function
    - Subsequent lines: extended description of function
- Describe parameters (value / expected type) and return

Function Comments

- The first string literal *inside* a function body is a docstring
  - First line: one-line summary of the function
  - Subsequent lines: extended description of function
- Describe parameters (value / expected type) and return
  - Many standards have emerged (javadoc, reST, Google)

Function Comments

- The first string literal *inside* a function body is a docstring
  - First line: one-line summary of the function
  - Subsequent lines: extended description of function
- Describe parameters (value / expected type) and return
  - Many standards have emerged (javadoc, reST, Google)
  - Just be consistent!

Function Comments

- The first string literal *inside* a function body is a docstring
    - First line: one-line summary of the function
    - Subsequent lines: extended description of function
- Describe parameters (value / expected type) and return
    - Many standards have emerged (javadoc, reST, Google)
    - Just be consistent!
- The usual rules apply too! List pre-/post-conditions, if any.

Example: Function Docstrings

Example: Function Docstrings

```python
def my_function():
```

Example: Function Docstrings

```python
def my_function():
    """Summary line: do nothing, but document it.

    Description: No, really, it doesn't do anything.
    """
    pass
```

Example: Function Docstrings

```python
def my_function():
    """Summary line: do nothing, but document it.

    Description: No, really, it doesn't do anything.
    """
    pass

print(my_function.__doc__)
```

Example: Function Docstrings

```
def my_function():
    """Summary line: do nothing, but document it.

    Description: No, really, it doesn't do anything.
    """
    pass

print(my_function.__doc__)

# Summary line: Do nothing, but document it.
#
#     Description: No, really, it doesn't do anything.
```

General Good Practices

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

Use blank lines to separate functions and logical sections inside functions.

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

Use blank lines to separate functions and logical sections inside functions.

Use spaces around operators and after commas, but not directly inside delimiters

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

Use blank lines to separate functions and logical sections inside functions.

Use spaces around operators and after commas, but not directly inside delimiters

```
a = f(1, 2) + g(3, 4)
```

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

Use blank lines to separate functions and logical sections inside functions.

Use spaces around operators and after commas, but not directly inside delimiters

```
a = f(1, 2) + g(3, 4)
```

Commenting Comment all nontrivial functions.

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

Use blank lines to separate functions and logical sections inside functions.

Use spaces around operators and after commas, but not directly inside delimiters

```
a = f(1, 2) + g(3, 4)
```

Commenting Comment all nontrivial functions.

Add header comments at the top of files before any imports.

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

Use blank lines to separate functions and logical sections inside functions.

Use spaces around operators and after commas, but not directly inside delimiters

```
a = f(1, 2) + g(3, 4)
```

Commenting Comment all nontrivial functions.

Add header comments at the top of files before any imports.

If possible, put comments on a line of their own.

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

Use blank lines to separate functions and logical sections inside functions.

Use spaces around operators and after commas, but not directly inside delimiters

```
a = f(1, 2) + g(3, 4)
```

Commenting Comment all nontrivial functions.

Add header comments at the top of files before any imports.

If possible, put comments on a line of their own.

Naming Use snake_case for variables and functions (CamelCase for classes)

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

Use blank lines to separate functions and logical sections inside functions.

Use spaces around operators and after commas, but not directly inside delimiters

```
a = f(1, 2) + g(3, 4)
```

Commenting Comment all nontrivial functions.

Add header comments at the top of files before any imports.

If possible, put comments on a line of their own.

Naming Use snake_case for variables and functions (CamelCase for classes)

Decomposition and Logic Same as in 106s

First-Class Functions

First-Class Functions

First-Class Functions

```
def echo(arg): return arg
```

First-Class Functions

```python
def echo(arg): return arg

type(echo)  # <class 'function'>
```

First-Class Functions

```
def echo(arg): return arg

type(echo)  # <class 'function'>
hex(id(echo))  # 0x1003c2bf8
```

First-Class Functions

```
def echo(arg): return arg

type(echo)  # <class 'function'>
hex(id(echo))  # 0x1003c2bf8
print(echo)  # <function echo at 0x1003c2bf8>
```

First-Class Functions

```
def echo(arg): return arg

type(echo)  # <class 'function'>
hex(id(echo))  # 0x1003c2bf8
print(echo)  # <function echo at 0x1003c2bf8>

foo = echo
```

First-Class Functions

```python
def echo(arg): return arg

type(echo)  # <class 'function'>
hex(id(echo))  # 0x1003c2bf8
print(echo)  # <function echo at 0x1003c2bf8>

foo = echo
hex(id(foo))  # '0x1003c2bf8'
```

First-Class Functions

```python
def echo(arg): return arg

type(echo)  # <class 'function'>
hex(id(echo))  # 0x1003c2bf8
print(echo)  # <function echo at 0x1003c2bf8>

foo = echo
hex(id(foo))  # '0x1003c2bf8'
print(foo)  # <function echo at 0x1003c2bf8>
```

First-Class Functions

```
def echo(arg): return arg

type(echo)  # <class 'function'>
hex(id(echo))  # 0x1003c2bf8
print(echo)  # <function echo at 0x1003c2bf8>

foo = echo
hex(id(foo))  # '0x1003c2bf8'
print(foo)  # <function echo at 0x1003c2bf8>

isinstance(echo, object)  # => True
```

Functions are Objects

Questions

Questions

What can you do with function objects?

Questions

What can you do with function objects?

What attributes does a function object possess?

Questions

What can you do with function objects?

What attributes does a function object possess?

Can I pass a function as a parameters to other functions?

Questions

What can you do with function objects?

What attributes does a function object possess?

Can I pass a function as a parameters to other functions?

Can a function return another function?

Questions

What can you do with function objects?

What attributes does a function object possess?

Can I pass a function as a parameters to other functions?

Can a function return another function?

How can I modify a function object?

Questions

What can you do with function objects?

What attributes does a function object possess?

Can I pass a function as a parameters to other functions?

Can a function return another function?

How can I modify a function object?

WE MUST GO DEEPER

Summary

Reference

Reference

All functions return *some* value (possibly None)

Reference

All functions return *some* value (possibly None)

Functions define scopes via symbol tables

Reference

All functions return *some* value (possibly None)

Functions define scopes via symbol tables

Parameters are passed by object reference

Reference

All functions return *some* value (possibly None)

Functions define scopes via symbol tables

Parameters are passed by object reference

Functions can have optional keyword arguments

Reference

All functions return *some* value (possibly None)

Functions define scopes via symbol tables

Parameters are passed by object reference

Functions can have optional keyword arguments

Functions can take a variable number of args and kwargs

Reference

All functions return *some* value (possibly None)

Functions define scopes via symbol tables

Parameters are passed by object reference

Functions can have optional keyword arguments

Functions can take a variable number of args and kwargs

Use docstrings and good style

Reference

All functions return *some* value (possibly None)

Functions define scopes via symbol tables

Parameters are passed by object reference

Functions can have optional keyword arguments

Functions can take a variable number of args and kwargs

Use docstrings and good style

Functions are objects too (?!)