Lecture 2

Environments

Most slides were borrowed from John DeNero

My Office Hours

- Not sure for this week yet.
- Will set them up next week for sure.
- Email me if urgent
  - We can skype if needed.

https://sites.google.com/a/eng.ucsd.edu/dsc20-winter-2019/staff-hours

Last time: pure functions

- Functions are pure if they just *return* a value. It doesn't change the state of the computer or external inputs. Must produce outputs that depend only on the inputs:
  - abs function

Last time: pure functions

- Functions are pure if they just *return* a value.
- It doesn't change the state of the computer or external inputs.
- Must produce outputs that depend only on the inputs:

```
def doubleStuff(a_list):
    """ Return a new list with doubled
elements from a_list. """
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)
    return new_list

things = [2, 5, 10]
things = doubleStuff(things)
print(things)

[4, 10, 20]
```

Last time: non-pure functions, have side effects

- In addition to returning a value (might be *None*), applying a non-pure function can generate side effects, which make some change to the state of the interpreter or program.
    - print function
    - time.time() because it returns a value based on the state of a clock, which was not an input to the function.

```
new_list = []

def doubleStuff(a_list):
    """Creates a new list with
doubled elements from a_list. """

    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)

things = [2, 5, 10]
doubleStuff(things)
print(new_list)
[4, 10, 20]
```

i-clicker question

```
def question(n):
    if n > 0:
        return print(n)
    else:
        return


t = question(10)
print(t)
```

What would Python display?

A: 10

B: None

C: Error of some sort

D: 10
None

E: None
10

i-clicker question

```
def question(n):
    if n > 0:
        return print(n)
    else:
        return


t = question(-10)
print(t)
```

What would Python display?

A:
```
-10
```

B:
```
None
```

C:
```
Error of some sort
```

D:
```
-10
None
```

E:
```
None
-10
```

I-clicker question

```
from operator import mul, add
x = 3

def mul_add(a):
    return add(mul(a,2), 4)

def one_more(a):
    a = a + 1
    print(mul_add(a))

one_more(x)
```

What is the output?

A: 24

B: 12

C: None

D: 10

E: Error

I-clicker question

```
from operator import mul, add
a = 3


def one_more(a):
    a = a + 1
    print(mul(2, add(a, a)))


print(one_more(a))
```

What is the output?

A: 12

B: 16

C: None

D: 16
   16

E: None of the above

Short-circuiting

Short-circuiting

```
>>> False  and 1/0
```

A: False

B: True

E: Error

Short-circuiting

```
>>> False  and 1/0
```

A: False

B: True

E: Error

```
>>> (5 and True) or (1/0 or True)
```

A: 5

B: True

C: False

D: Error

Assignments

Assignment operator

- Assignment is our simplest means of abstraction
  - (demo)

- ```
  a = 5 ** 5
  b = 10 ** 4
  ```

Or *in one line*:

```
a, b = 5 ** 5, 10 ** 4
```
```

Execution rule

Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.

2. Bind all names to the left of = to those resulting values in the current frame.

Assignment operator

- Assignment is our simplest means of abstraction

A: b is 10

B: b is 5

C: b is None

D: Error

-
```
a = 5 ** 5
b = 10 ** 4
```

Or in one line: `a, b = 5 ** 5, 10 ** 4`

-
```
a = 5
b = a * 2

#b is 10
```

How about now?

```
a, b = 5, a*2

#b is ??
```

Can you swap two values?

How to swap *a* and *b*, *without* using a third variable?

```
a, b = 1, 2
```

A: No, I need to use more than one variable
B: Yes, I can do it in one line
C: Yes, I can do it using multiple lines
D: I do not know how to do it all
E: More than one possible answer

Can you swap two values without a third one

Version 1:

```
a, b = 1, 2

a, b = b, a
```

Version 2:

```
a, b = 1, 2     # a = 1, b = 2

a = a + b       # a = 3

b = a - b       # b = 3 - 2= 1

a = a - b       # b = 3 - 1= 2
```

Existing functions can get new names

(demo)

- Another way to give names to values:
  - def statement (let's us create our own functions)

Question

```
def sum_some(limit):
    return limit + 5

sum = sum_some(10)
num1 = 11
num2 = 19
result = sum(num1, num2)
print(result)
```

What is printed?

A: 30

B: 15

C: 40

D: 45

E: Error

Question

```
def sum_some(limit):
    return limit + 5

sum = sum_some(10)
num1 = 11
num2 = 19
result = sum(num1, num2)
print(result)
```

What is printed?

A: 30

B: 15

C: 40

B: 45

C: Error

Discussion question

```
from operator import add, sub

a = add

b = sub

c, a = add, sub

add  = a

add (a (2, b (c (3, 1), 2)), 1)
```

What is the value of the final expression in this sequence?

A: -1

B: 0

C: 1

D: Aaah, my brain!!

E: It is not possible

Environment Diagrams
(what is going on within an interpreter)

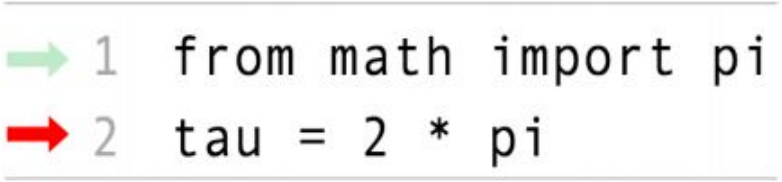
Demo

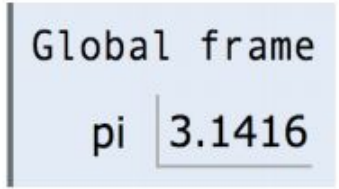
Environment Diagrams **visualize** the interpreter process:

http://pythontutor.com/composingprograms.html#mode=edit

Environment Diagrams

Environment diagrams visualize the interpreter's process.

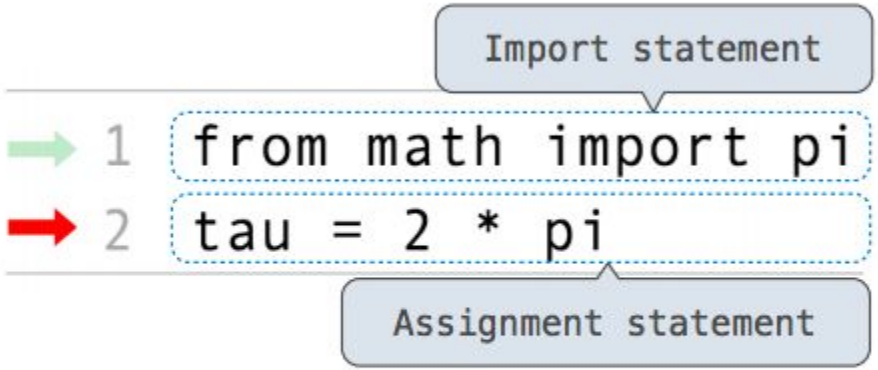


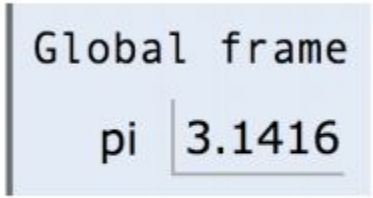


Code (left): **Frames (right):**

Environment Diagrams

Environment diagrams visualize the interpreter's process.





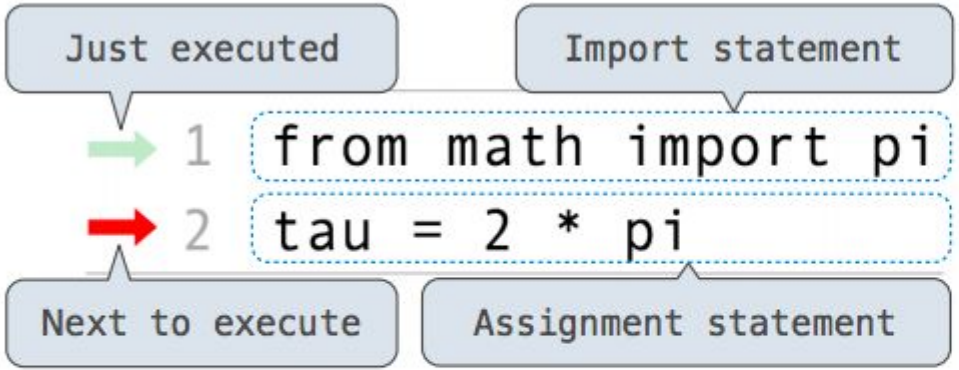
Code (left):

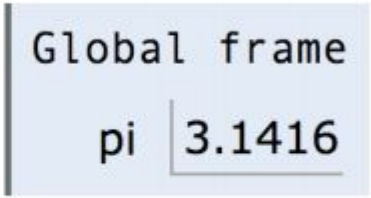
Frames (right):

Statements and expressions

Environment Diagrams

Environment diagrams visualize the interpreter's process.





Code (left):

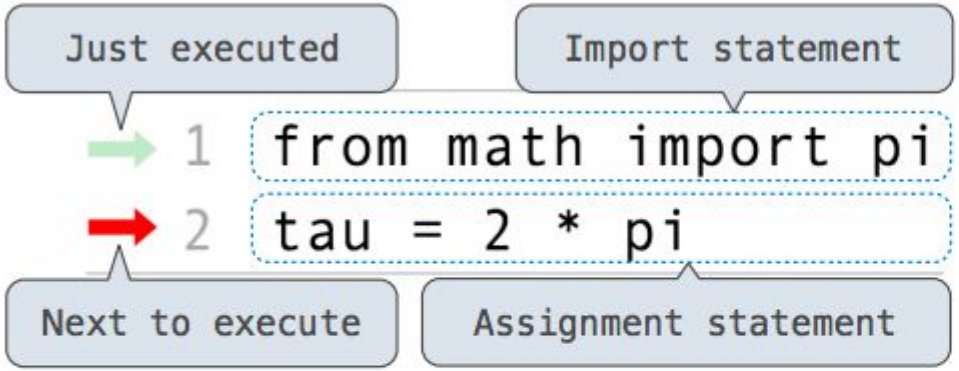
Statements and expressions

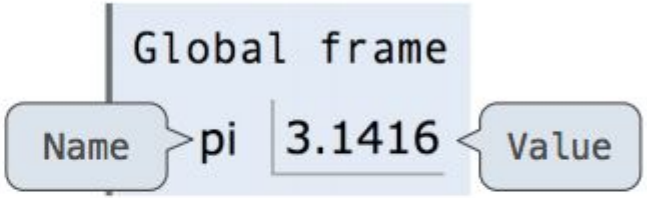
Arrows indicate evaluation order

Frames (right):

Environment Diagrams

Environment diagrams visualize the interpreter's process.





Code (left):

Statements and expressions

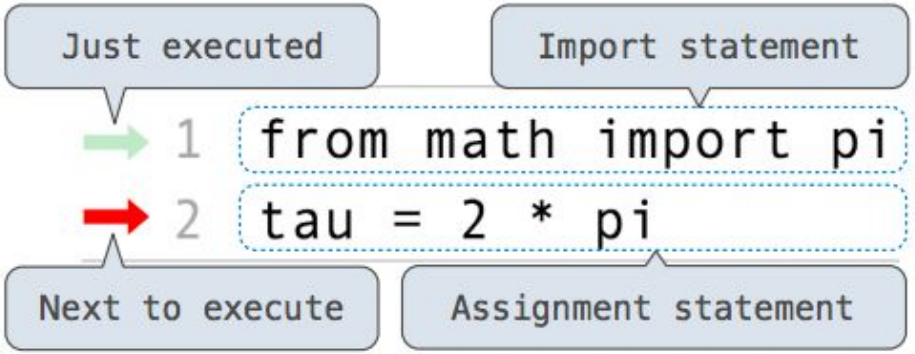
Arrows indicate evaluation order

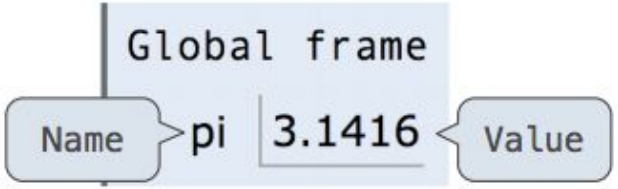
Frames (right):

Each name is bound to a value

Environment Diagrams

Environment diagrams visualize the interpreter's process.





Code (left):

Statements and expressions

Arrows indicate evaluation order

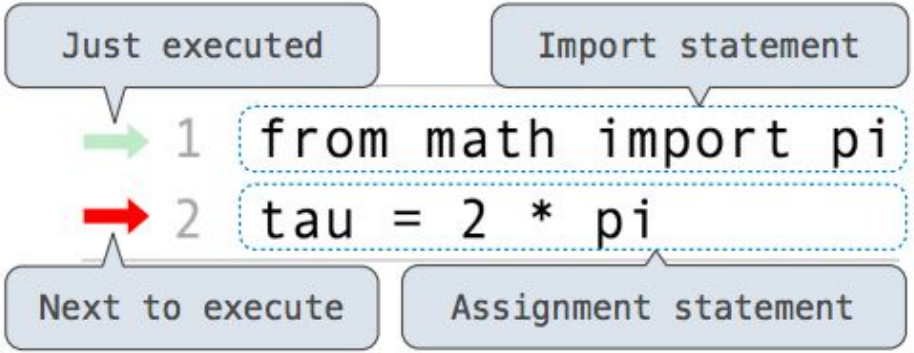
Frames (right):

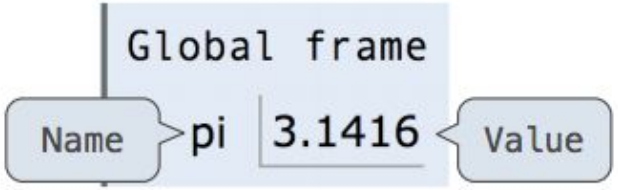
Each name is bound to a value

Within a frame, a name cannot be repeated

Environment Diagrams

Environment diagrams visualize the interpreter's process.





Code (left):

Statements and expressions

Arrows indicate evaluation order

Frames (right):

Each name is bound to a value

Within a frame, a name cannot be repeated

http://pythontutor.com/composingprograms.html#mode=edit

Discussion Question (demo)

```
a = 1
b = 2
b, a = a + b, b - a
```

What are the values for *a* and *b* after line 3?

A: a = 3, b = 2
B: a = 2, b = 3
C: a = 3, b = 1
D: a = 1, b = 3
E: None of the above

Discussion Question

```
a = 1
b = 2
b, a = a + b, b - a

b, a = 3, 1
b is 3
a is 1.
```

What are the values for *a* and *b* after line 3?

A: a = 3, b = 2
B: a = 2, b = 3
C: a = 3, b = 1
D: a = 1, b = 3
E: None of the above

Discussion Question (DEMO)

```
from operator import add, sub

a = add

b = sub

c, a = add, sub

add  = a

add (a (2, b (c (3, 1), 2)), 1)
```

What is the value of the final expression in this sequence?

A: -1

B: 0

C: 1

D: Aaah, my brain!!

E: It is not possible

Defining Functions

Defining new functions

Function definition is another way to do abstraction: binds names to *expressions*

```
def <function_name> (<formal_parameters>):
        <function_body>
        return <return expression>
```

Defining new functions

Function **signature:** how many arguments a function takes

```
def <function_name> (<formal_parameters>):
        <function_body>

        return <return expression>
```

Parameters vs. Arguments

The **arguments** are the data you pass into the function's **parameters**

```
from operator import sub, mul


def discriminant(a, b, c):
    return ...
```

```
# function call:

discriminant(1, 2, 3)
```

Formal Parameters

Arguments

Execution procedure for def statement

```
def <function_name> (<formal_parameters>):
    <function_body>
    return <return expression>
```

- Create a function with a given *signature*
- *Set* the body of that function to be everything indented after the first line
- Bind `<function_name>` to that function in the *current frame*
  - Each frame contains bindings, each of which associates a name with its corresponding value

Question

```
def my_func (x):

    return x + a + b
```

What do you think will happen after you created this function?

A: Interpreter checks for syntax errors. There are none. Nothing will happen.

B: Interpreter checks for logic errors. There are a few. Error message.

C: Interpreter does not check for any errors at this stage. Nothing will happen.

D: Something else.

Question (demo)

```
def my_func (x):

    return x + a + b
```

What do you think will happen after you created this function?

A: Interpreter checks for syntax errors. There are none. Nothing will happen.

B: Interpreter checks for logic errors. There are a few. Error message.

C: Interpreter does not check for any errors at this stage. Nothing will happen.

D: Something else.

Calling user-defined functions

Procedure for calling/applying user-defined functions:

- Add a *local* frame, forming a new environment
- Bind the function's *formal parameters* to its *arguments* in that frame
- Execute the body of the function in that new environment

Parameters vs. arguments

The **arguments** are the data you pass into the function's **parameters**

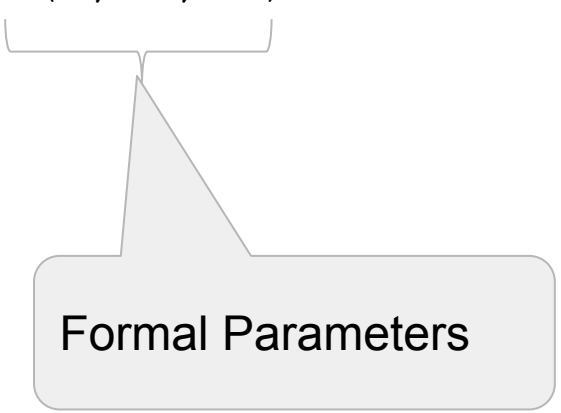
```
from operator import sub, mul

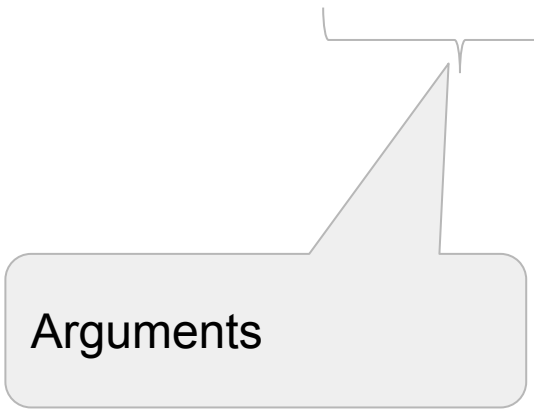

def discriminant(a, b, c):

    return ...
```

```
# function call:

discriminant(1, 2, 3)
```



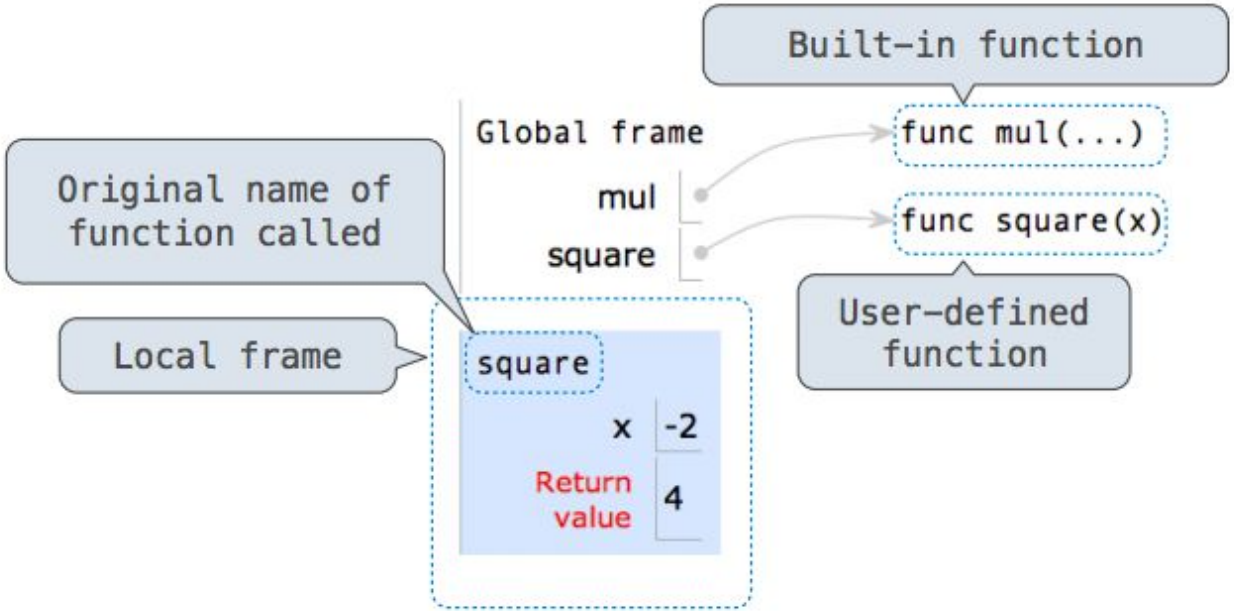


(Discriminant demo)

Calling user-defined functions

Procedure for calling/applying user-defined functions:

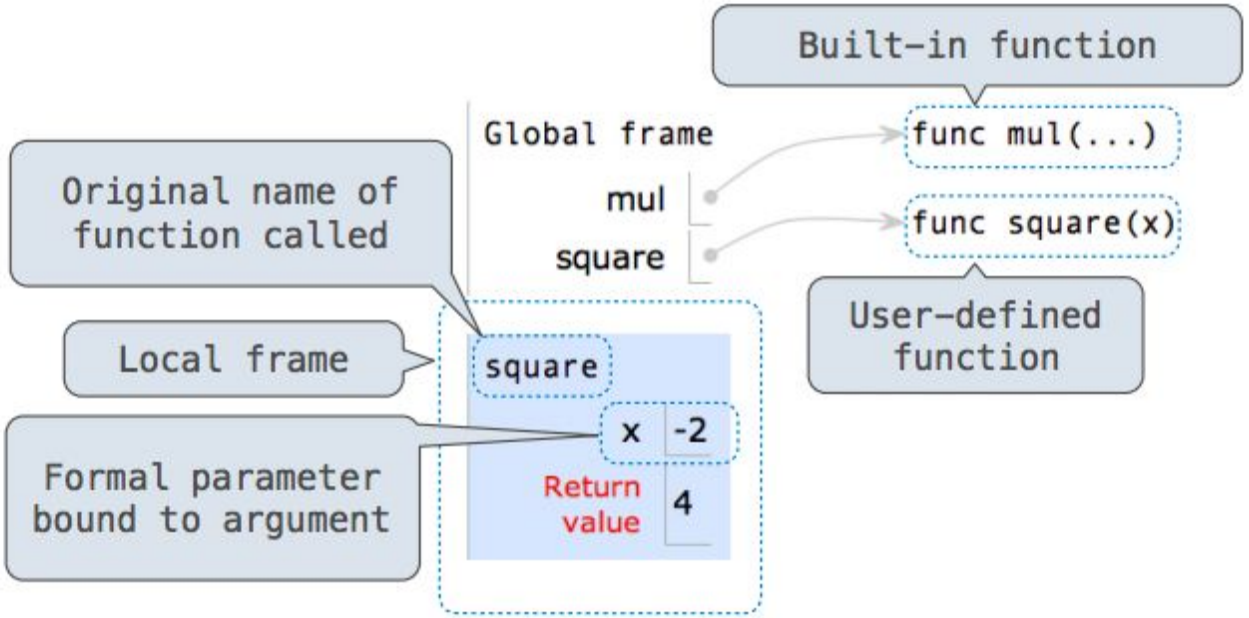
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```



Calling user-defined functions

Procedure for calling/applying user-defined functions (version 1):

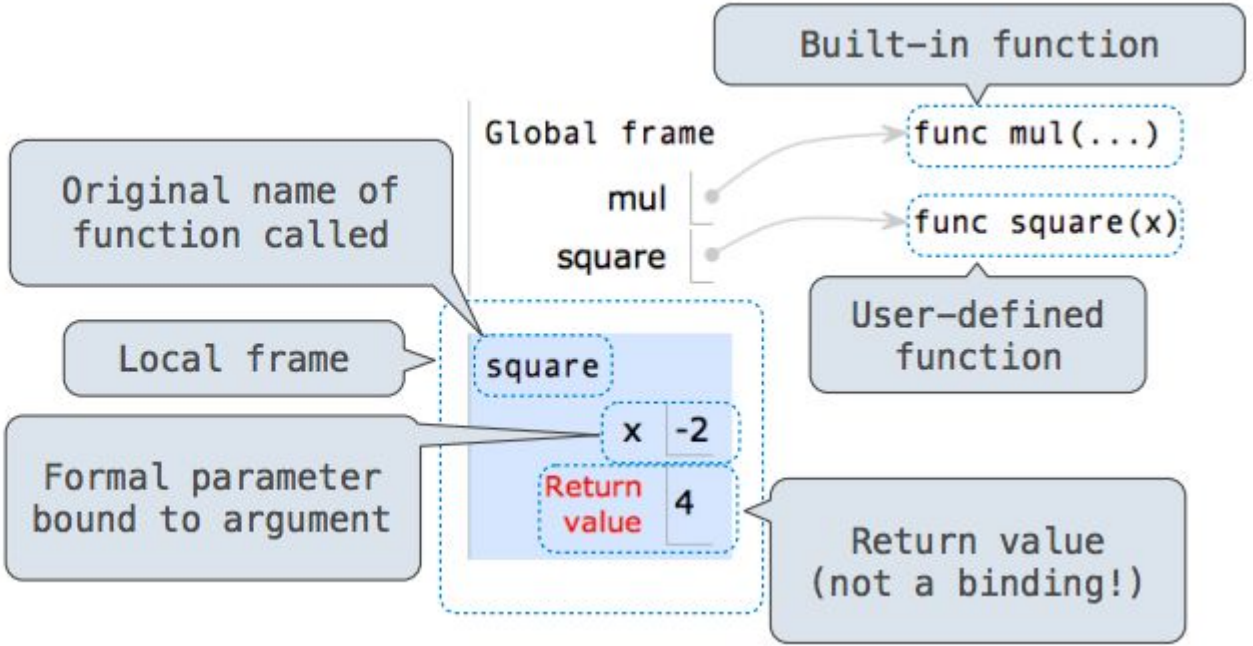
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```



Calling user-defined functions

Procedure for calling/applying user-defined functions (version 1):

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```

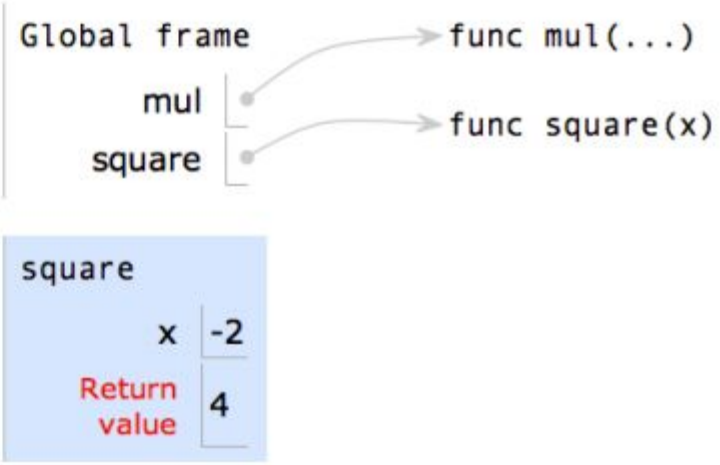


Calling user-defined functions

Procedure for calling/applying user-defined functions (version 1):

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```

A function's signature has all the information needed to create a local frame

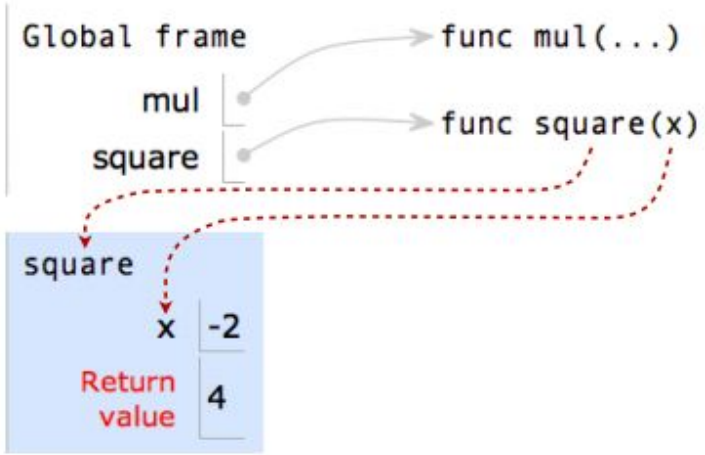


Calling user-defined functions

Procedure for calling/applying user-defined functions (version 1):

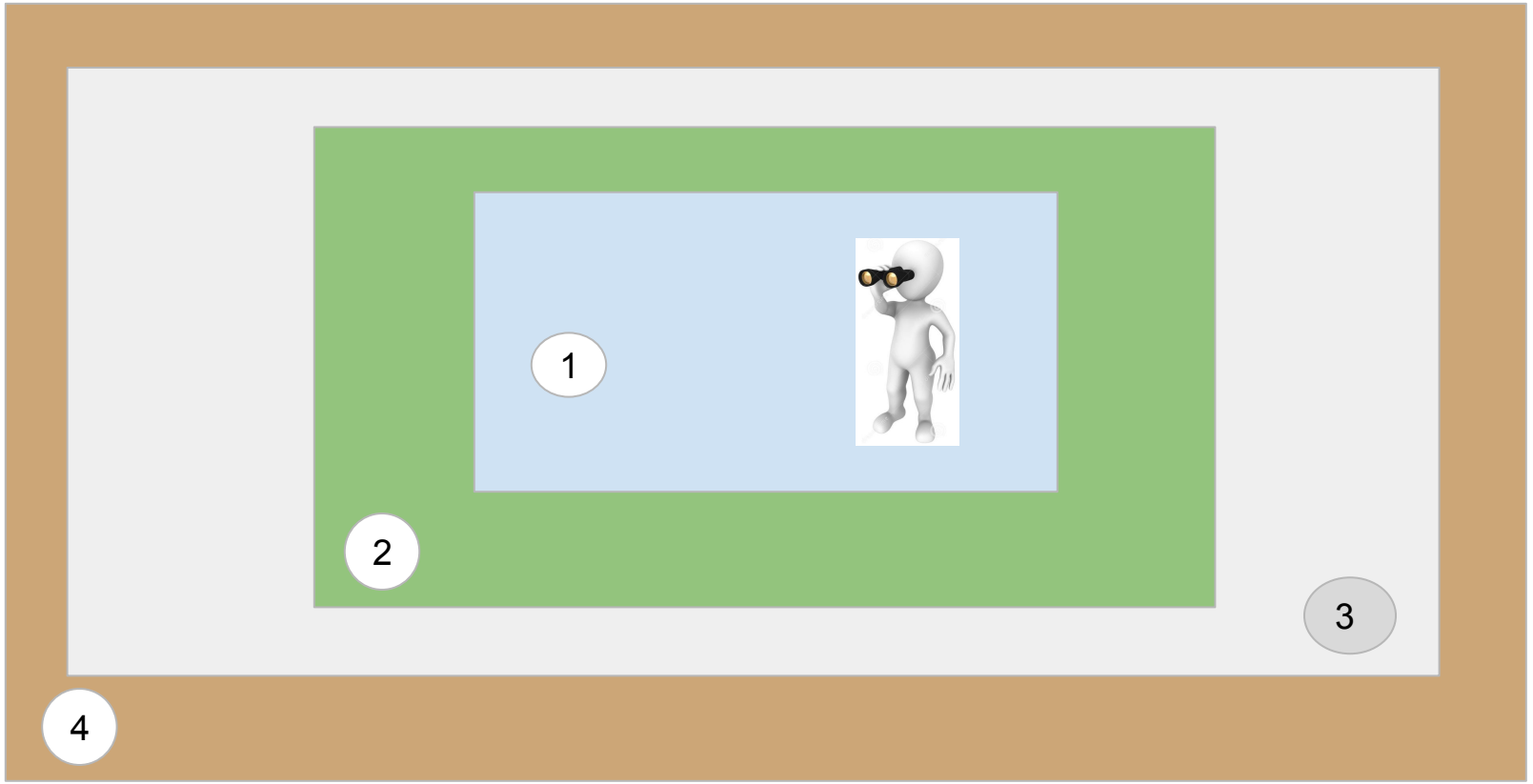
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```

A function's signature has all the information needed to create a local frame

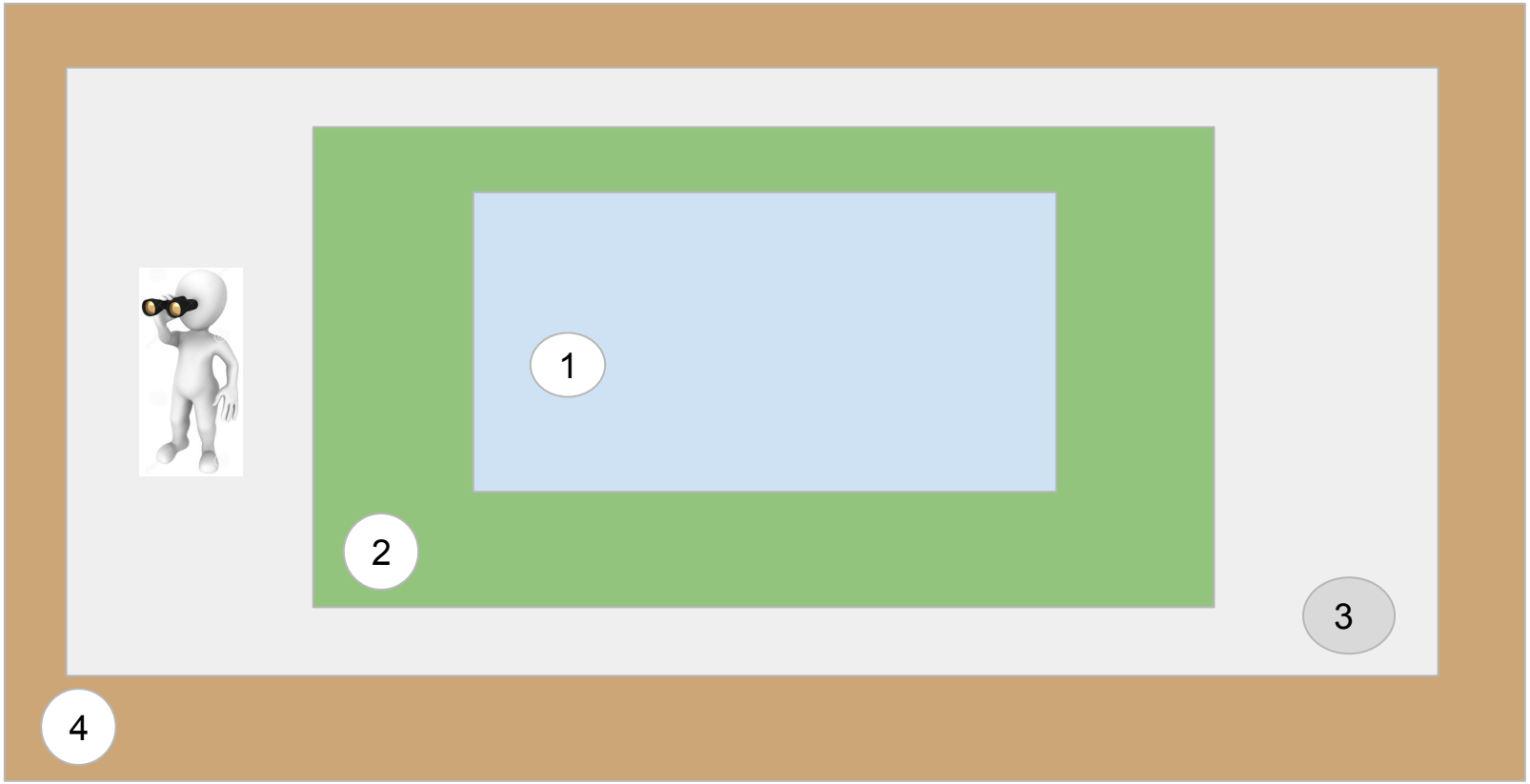


Looking Up Names In Environments

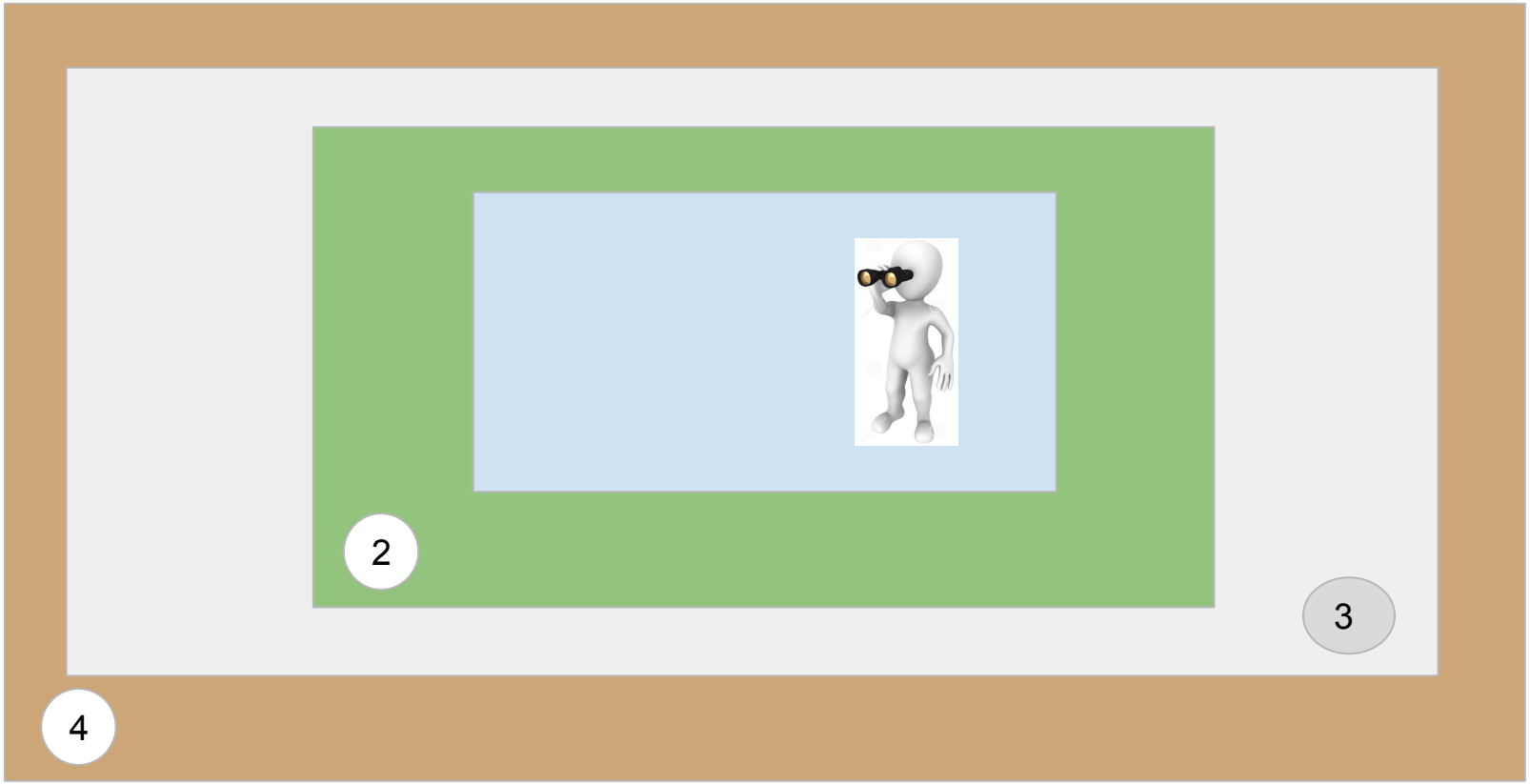
Find the circle



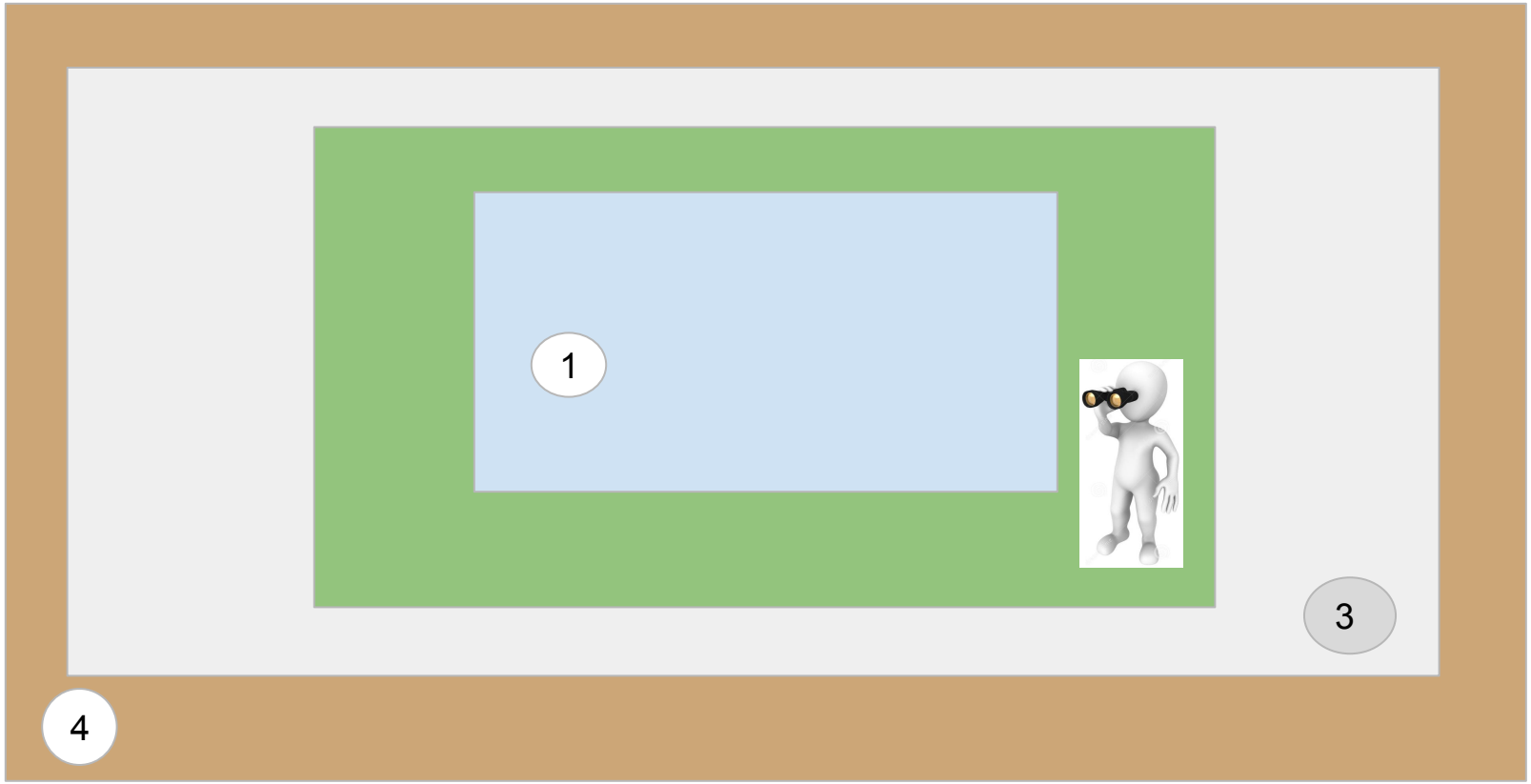
Find the circle



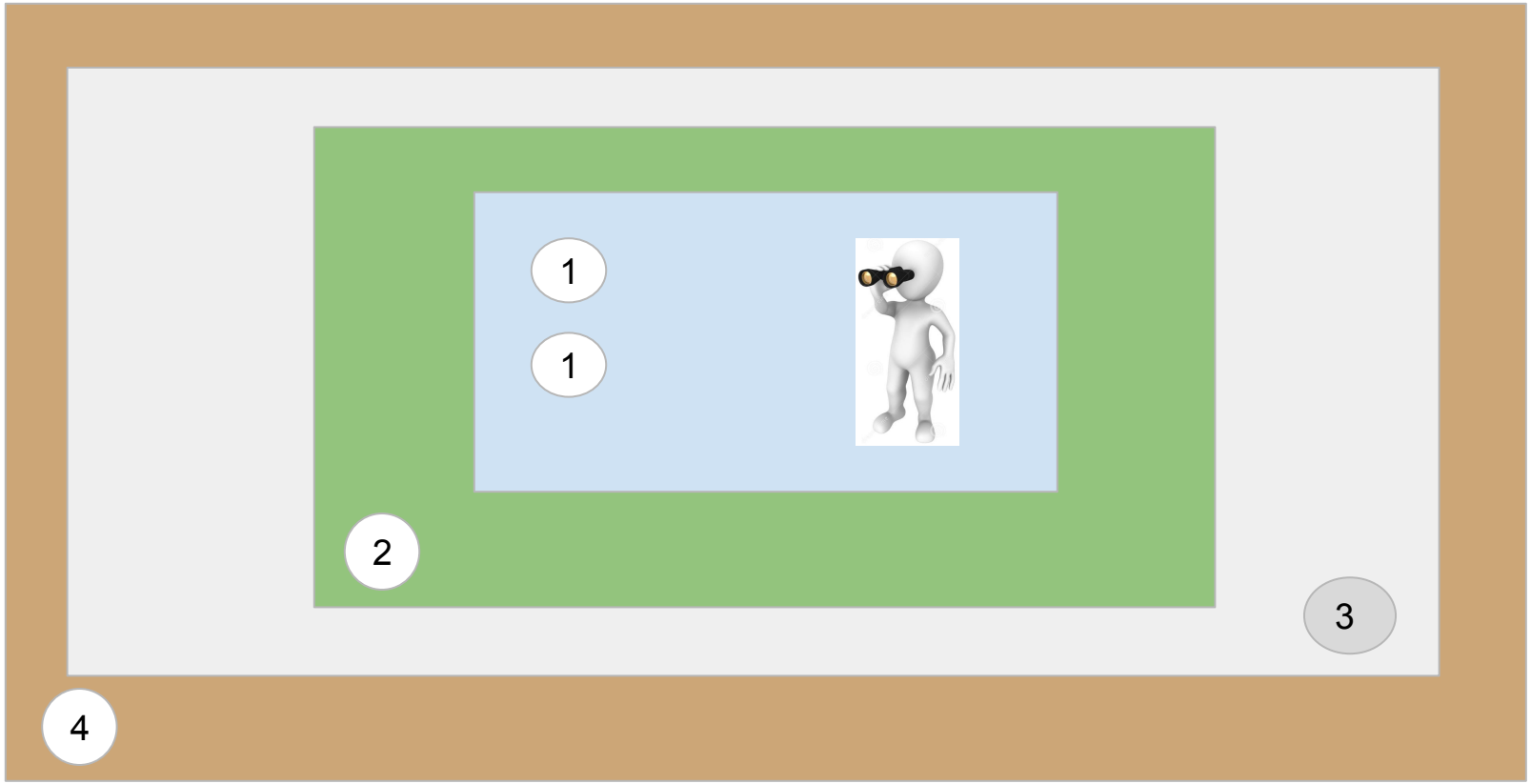
Find the circle



Find the circle



Find the circle



Looking Up Names In Environments

- Every expression is evaluated in the context of an environment.
- *Environments* are the memory that keeps track of the correspondence between names and values.
- So far, the *current* environment is either:
  - The ***global frame*** alone, or
  - A ***local frame***, followed by the global frame.
- *Frame is a binding between names and values.*

Looking Up Names In Environments

- So far, the current environment is either:
  - The ***global frame*** alone, or
  - A ***local frame***, followed by the global frame.

Important:

1. An environment is a **sequence** of frames.

2. A name evaluates to the value bound to that name in the **earliest** frame of the current environment in which that name is found.

Looking Up Names In Environments

Important:

1. An environment is a **sequence** of frames.

2. A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Example: look up the name in the `discriminant` function:

- Look for that name in the local frame
- Look for it in the global frame

Question

```
def number (number):
    return number ** number + number



number(3)
```

What will be a result of the function call?

A: 12

B: 30

C: Unpredicted value

D: Error

E: Nothing will be printed (None)

(Demo)