

WRITING YOUR OWN FUNCTIONS AND PROCEDURES

LECTURE 03-1

JIM FIX, REED COLLEGE CSC1 121

INDENTATION

- ▶ Python reads the functions, looking for its indented lines of code

```
def square(x):  
    return x * x
```

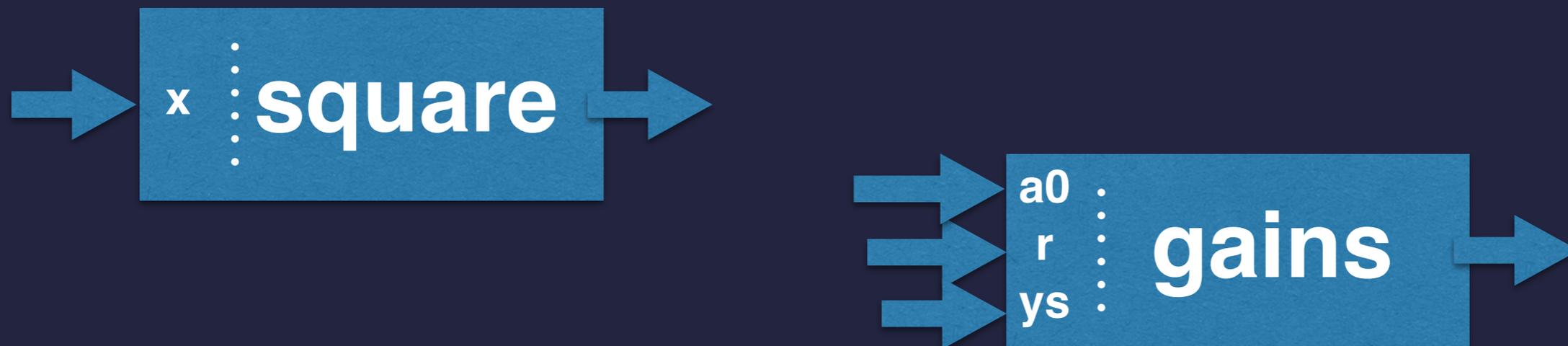
```
def gains(initial, yearly_rate, years):  
    multiplier = 1.0 + yearly_rate / 100.0  
    growth = multiplier ** years  
    amount = initial * growth  
    return amount - initial
```

```
def distanceFromTo(startX, startY, endX, endY):  
    changeX = endX - startX  
    changeY = endY - startY  
    distanceSquared = changeX**2 + changeY**2  
    return distanceSquared ** 0.5
```

each function's lines are indented by 4 spaces

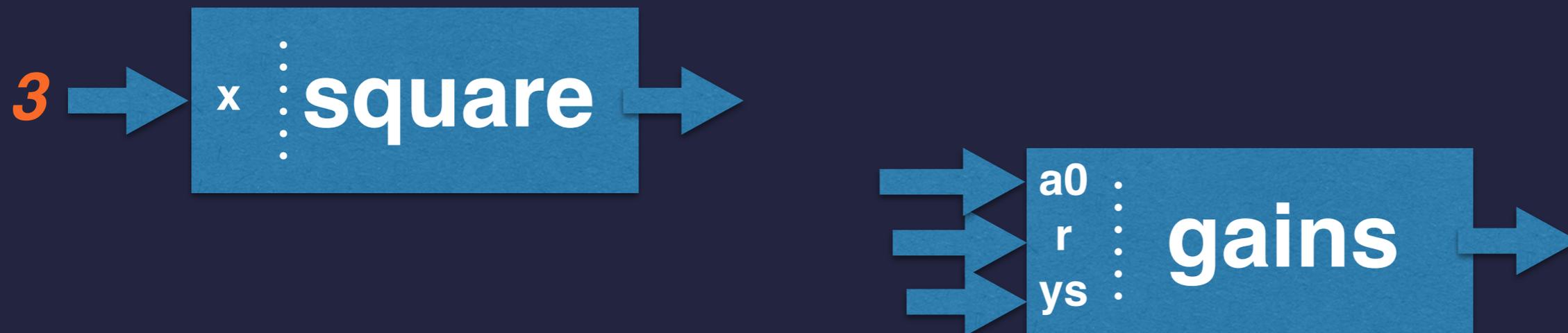
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



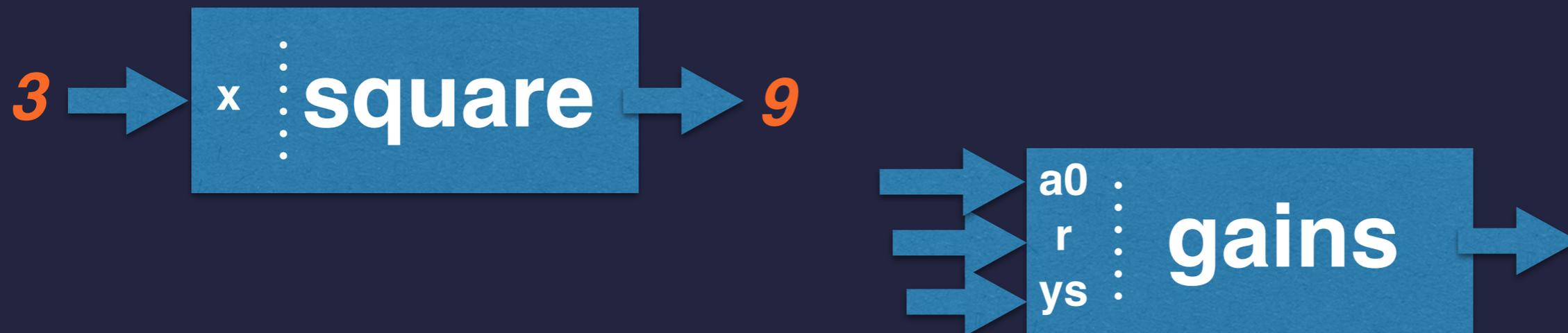
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



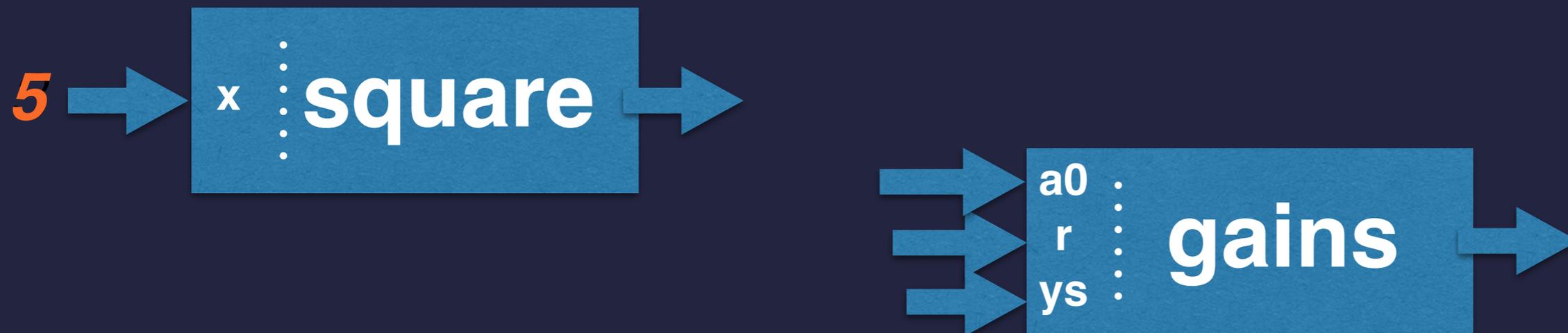
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



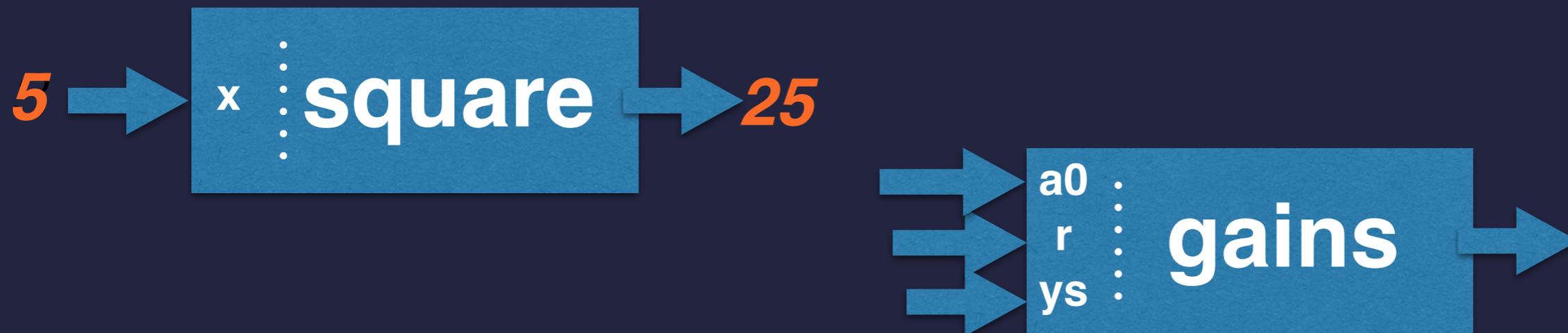
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



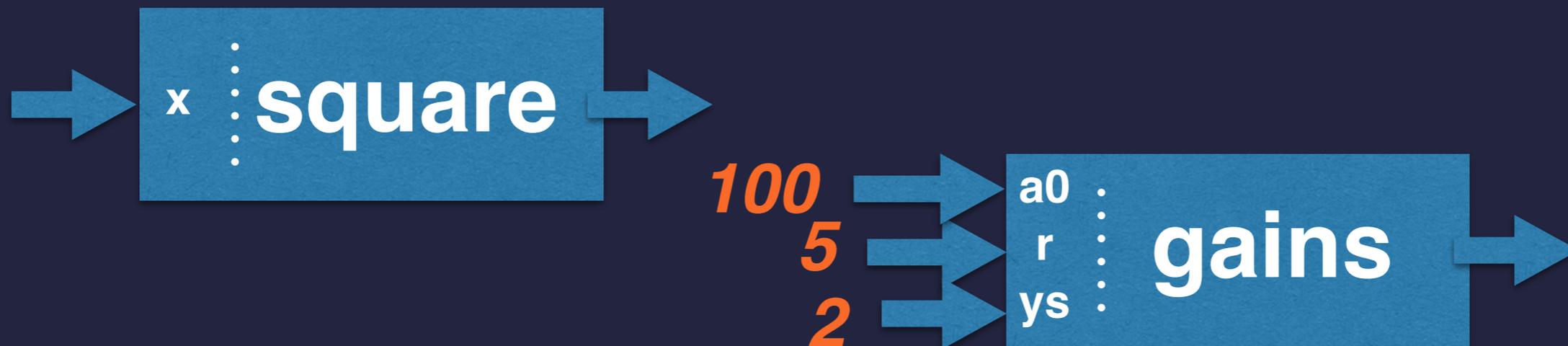
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



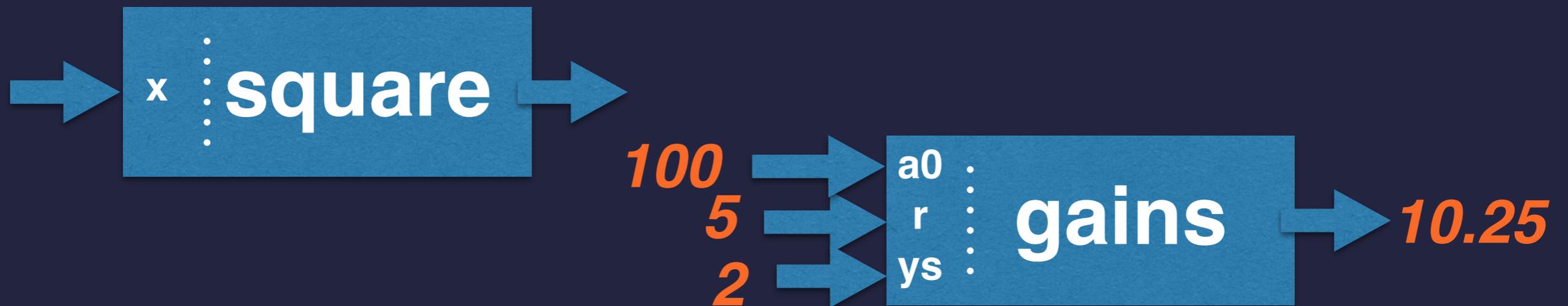
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



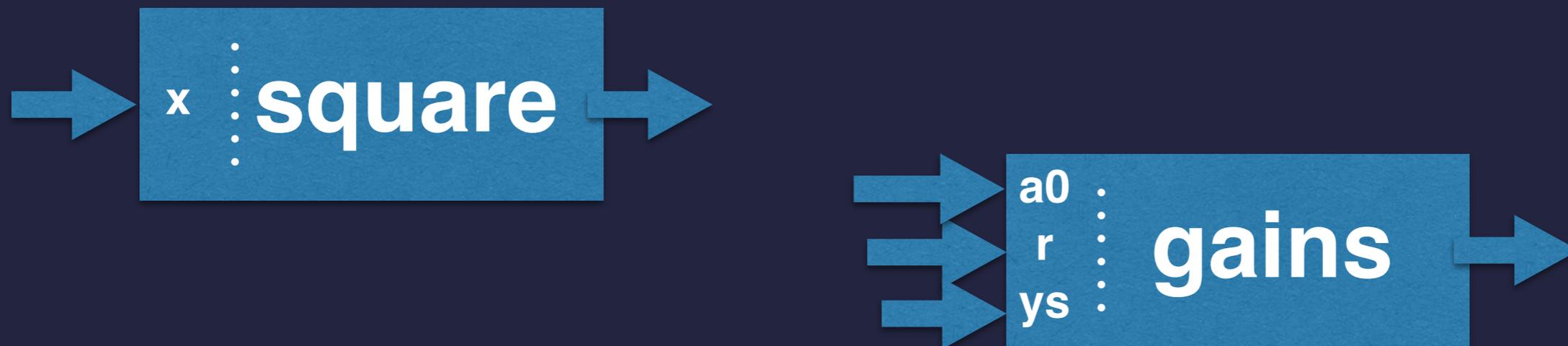
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



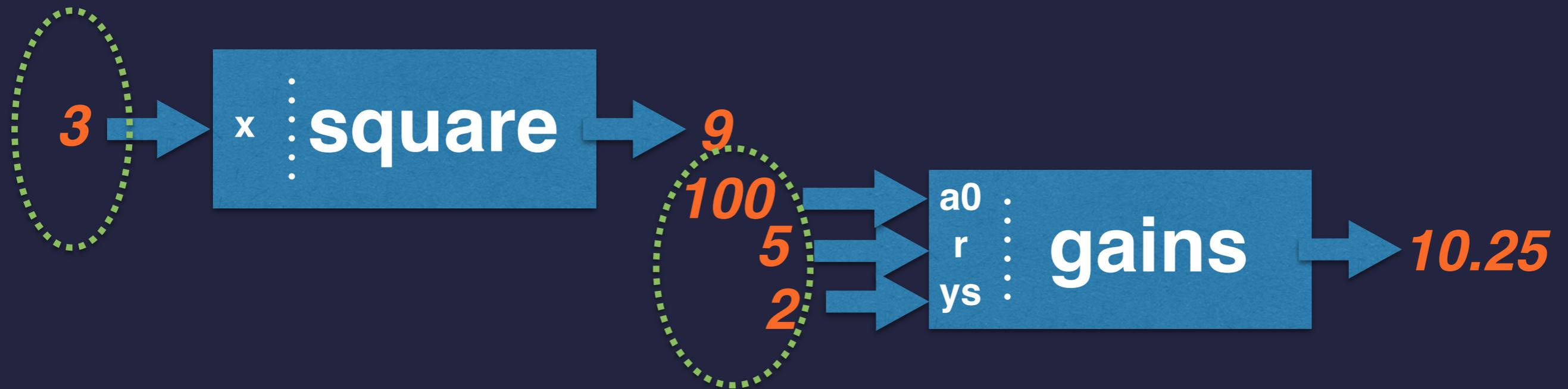
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

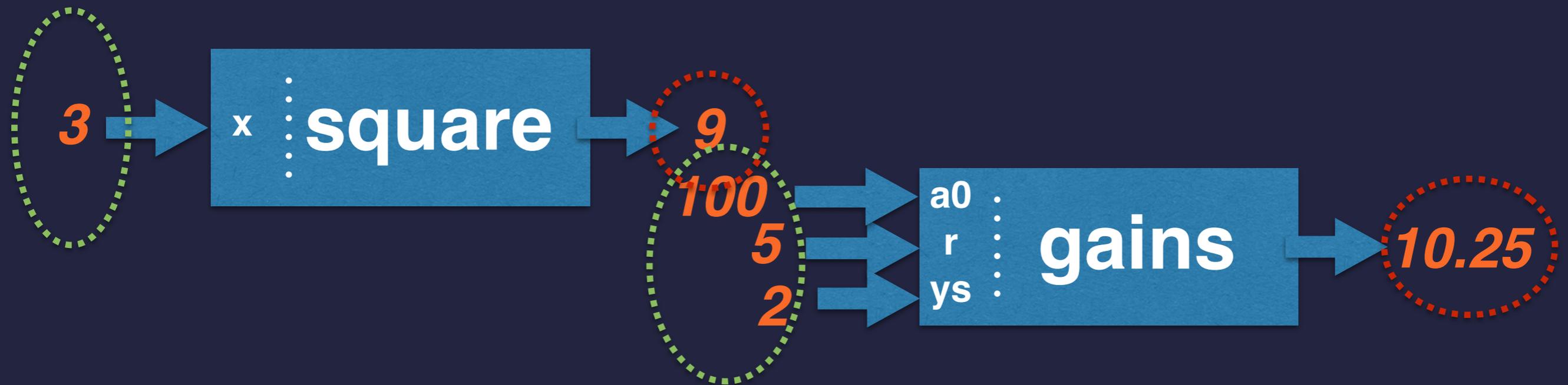
- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



Parameters are fed in.

FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:

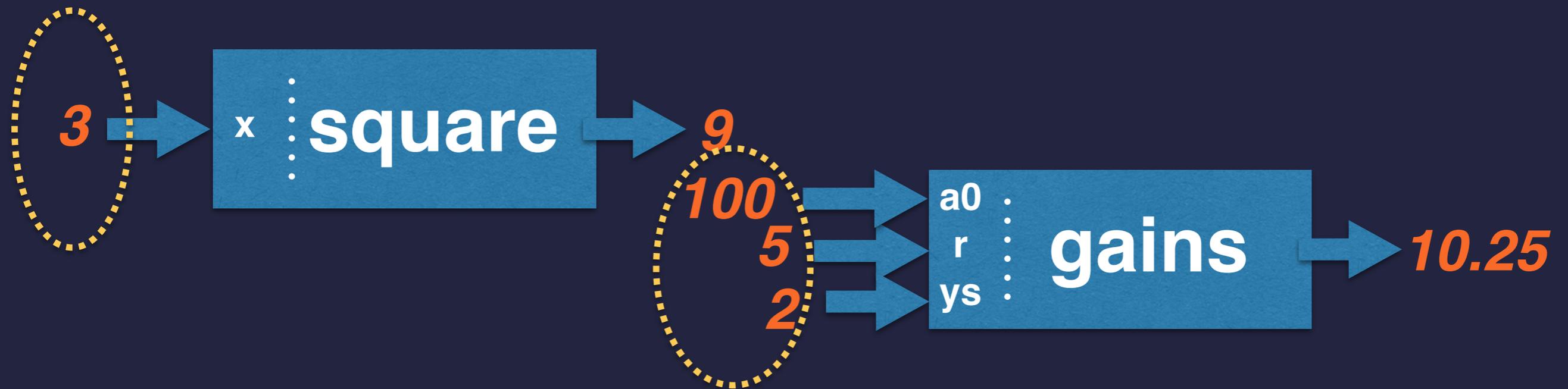


Parameters are fed in.

A returned result comes out.

FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- ▶ A function takes one or more *parameter* values.
- ▶ It uses those values to compute its result.
- ▶ It then *returns* the result back to the calling expression.
- ▶ Functions can be thought of as “value factories” of a program:



The expected number, type, and ordering of parameters is the function's *interface*.

SYNTAX: FUNCTION DEFINITION

Below gives a template for function definitions:

```
def function-name (parameter-list) :  
    lines of statements that compute using the parameters  
    ...  
    return the-computed-value
```

- ▶ The parameter variables are called its **formal parameters**.
 - They don't have specific values when the function is defined.
- ▶ They represent the values that will get fed in with some call.
 - They vary, in a way, from call to call.

SYNTAX: FUNCTION DEFINITION

Below gives a template for function definitions:

```
def function-name (parameter-list) :  
    lines of statements that compute using the parameters  
    ...  
return the-computed-value
```

- ▶ Each line of the function's body is *indented with 4 spaces*.
 - This code is executed when the function is called.
- ▶ The last line is usually a **return** statement.

FUNCTION CALLS AS EXPRESSIONS

- ▶ Because functions compute and return a result, they are used within expressions.
- ▶ Can sometimes think of their definitions as being "cut and pasted" in.

For example, the expression

```
>>> square(3) + square(4)
```

- ▶ can be viewed as the same as this expression

```
>>> (3 * 3) + (4 * 4)
```

FUNCTION CALLS

Some more terminology:

- ▶ Below are two *calls*, or *uses*, of our `square` function:

```
sqrt(square(3) + square(4))
```

- Each use of a function occurs at a *call site* in the code.
- 3 is the *actual parameter* for its call site. As is 4 for *its* site.

FUNCTION CALLS

Some more terminology:

- ▶ Below are two *calls*, or *uses*, of our `square` function:

```
sqrt (square(3) + square(4))
```

- Each use of a function occurs at a *call site* in the code.
- 3 is the *actual parameter* for its call site. As is 4 for *its* site.

SCRIPTING WITH FUNCTIONS

- ▶ We can define functions in scripts.
- ▶ Lay out a series of useful function definitions at the top.
 - We call them in the main lines of the script...
 - ... but we might perhaps also call them in other functions.

EXAMPLE SCRIPT WITH FUNCTIONS

```
from math import pi, sqrt

def getFloat(prompt):
    return float(input(prompt))

def getArea():
    a = getFloat("Circle area? ")
    while a < 0.0:
        a = get_float("Not an area. Try again: ")
    return a

def radiusOfCircle(A):
    return sqrt(A / pi)

area = getArea()
radius = radiusOfCircle(area)
print("That circle's radius is "+str(radius)+".")
```

SCRIPTING WITH FUNCTIONS

Why should we define functions?

- Makes code readable.
- Creates reusable code components.
- Makes debugging and testing easier.
- Allows you to hide implementation.

With coding its good to take a "client/service" mentality:

- Write functions that serve other parts of the code well.
- The client code doesn't need to know the internals of a function, just the interface.

TESTING PYTHON FUNCTIONS

- ▶ You can also have Python files that only contain function definitions.
- ▶ You can load a Python file and interact with it using `-i`

```
C02MX1KLFH04:examples jimfix$ python3 -i example_functions.py
>>> square(3)
9
>>> gains(100,5,2)
10.25
>>> distanceFromTo(-1.0, 2.5, 2.0, 6.5)
5.0
>>>
```

THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ Function calls are another kind of "control flow". Python "jumps around."
- ▶ Below is an example with two: `getArea` and `radiusOfCircle`.

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ Function calls are another kind of "control flow". Python "jumps around."
- ▶ Below is an example with two: `getArea` and `radiusOfCircle`.

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

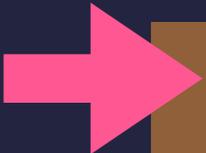
```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

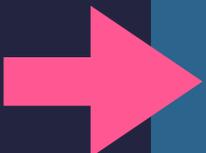
```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

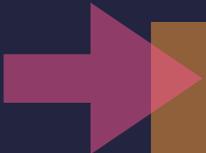
THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ The instruction pointer jumps from the main script code, up to the function's code, and then returns back.



```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

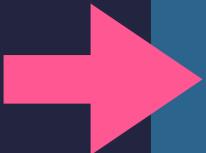
```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

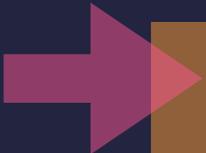
THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ The instruction pointer jumps from the main script code, up to the function's code, and then returns back.



```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

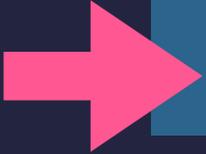
```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

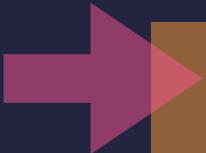
THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ The instruction pointer jumps from the main script code, up to the function's code, and then returns back.



```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



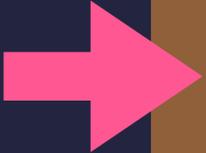
```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

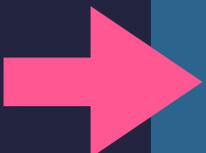


```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```



```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

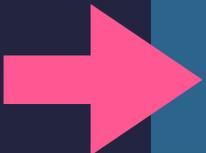


```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```



```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

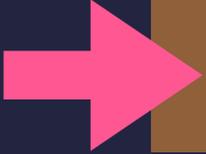
THE FLOW OF CONTROL WITH FUNCTIONS

- ▶ The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```



LOCAL VS. GLOBAL FRAMES

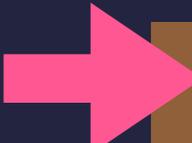
- ▶ When a function gets called, a *local frame* gets created for the function's local variables.

global frame



```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

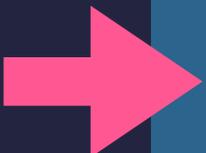
LOCAL VS. GLOBAL FRAMES

- ▶ When a function gets called, a *local frame* gets created with its own local variables.

getArea frame

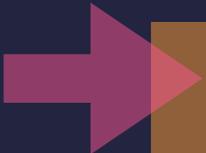
a: 314.159

global frame



```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

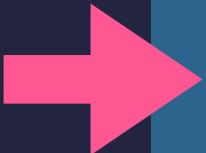
LOCAL VS. GLOBAL FRAMES

- ▶ When a function gets called, a *local frame* gets created with local variables.

getArea frame

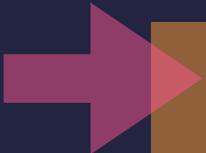
a: 314.159

global frame



```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

LOCAL VS. GLOBAL FRAMES

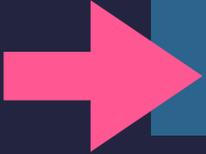
- ▶ When a function gets called, a *local frame* gets created with local variables.

getArea frame

a: 314.159

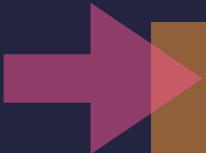
returning 314.159

global frame



```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

LOCAL VS. GLOBAL FRAMES

- ▶ When a function gets called, a *local frame* gets created for the function's local variables.

global frame

area: 314.159

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

LOCAL VS. GLOBAL FRAMES

- ▶ When a function gets called, a *local frame* gets created with its own local variables.

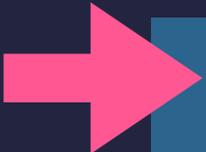
radiusOfCircle frame

someArea: 314.159

global frame

area: 314.159

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```



```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

LOCAL VS. GLOBAL FRAMES

- ▶ When a function gets called, a *local frame* gets created with local variables.

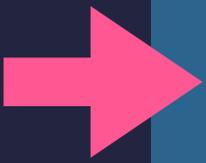
radiusOfCircle frame

```
someArea: 314.159  
pi: 3.141592653589793  
sqrt: <function that computes sqrt>
```

global frame

```
area: 314.159
```

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```



```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

LOCAL VS. GLOBAL FRAMES

- ▶ When a function gets called, a *local frame* gets created with local variables.

radiusOfCircle frame

```
someArea: 314.159  
pi: 3.141592653589793  
sqrt: <function that computes sqrt>  
returning 0.9999995776679783
```

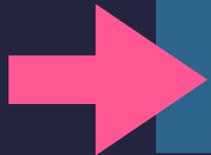
global frame

```
area: 314.159
```

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```



LOCAL VS. GLOBAL FRAMES

- ▶ When a function gets called, a *local frame* gets created for the function's local variables.

global frame

area: 314.159
radius: 0.9999995776679783

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

LOCAL VS. GLOBAL FRAMES

- ▶ When a function gets called, a *local frame* gets created for the function's local variables.

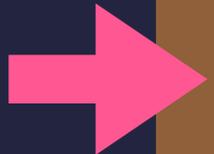
global frame

area: 314.159
radius: 0.9999995776679783

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```



IMPORT AND DEF CREATE FRAME ENTRIES

- ▶ Both **def** and **import** introduce names.
- ▶ These get placed in the frame of the block being executed.

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

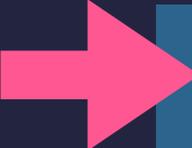
```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

REDO: DEF EXECUTION

- ▶ When a block has a **def**, a function object gets created.
- ▶ The new name's association is added to the frame.

global frame

getArea: <function that requests>



```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

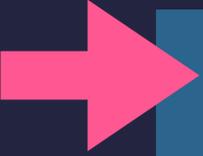
REDO: DEF EXECUTION

- ▶ When a block has a **def**, a function object gets created.
- ▶ The new name's association is added to the frame.

global frame

getArea: <function that requests>
radiusOfCircle: <function that sqrts>

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```



```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```

```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

REDO: DEF EXECUTION

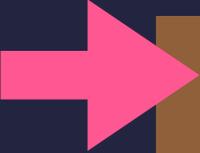
- ▶ When a block has a **def**, a function object gets created.
- ▶ The new name's association is added to the frame.

global frame

```
getArea: <function that requests>  
radiusOfCircle: <function that sqrts>  
area: 314.159
```

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

REDO: DEF EXECUTION

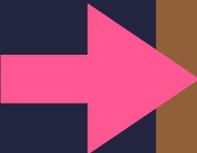
- ▶ When a block has a **def**, a function object gets created.
- ▶ The new name's association is added to the frame.

global frame

```
getArea: <function that requests>  
radiusOfCircle: <function that sqrts>  
area: 314.159  
radius: 0.9999995776679783
```

```
def getArea():  
    a = float(input("Circle area? "))  
    while a < 0.0:  
        a = float(input("Not an area. Try again:"))  
    return a
```

```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

FUNCTION CALLING MECHANISM

- Functions are passed the values of their arguments.
- Functions have their own variables, managed by their **local frame**.
 - The frame is initialized with a call:
 - ✦ The **formal** parameters are set to the **actual** argument values.
 - ✦ Assignment statements can introduce new local variables in the frame.
 - ✦ (So do nested **def** and **import** statements.)
- Functions **return** a value back to the calling statement.
 - Upon **return**, the function's local frame goes away.

A local frame's **lifetime** is the time between its function's call and return.

FUNCTION CALLING MECHANISM (CONT'D)

- Each function call leads to creation of a new frame.
- Frames due to calls **stack up**.
 - ➔ This happens when the script calls a function...
 - ➔ ...and that function calls a function. Etc.

We'll examine this more later after you've had some practice writing them.

MORE EXAMPLES: A PARITY FUNCTION

- ▶ Here is a function that returns the *parity* of a number as a string:

```
def getTheParityOf(n):  
    if n % 2 == 0:  
        return "even"  
    else:  
        return "odd"
```

MORE EXAMPLES: ABSOLUTE VALUE COMPUTATION AS A FUNCTION

- ▶ A **return** can appear within the function, not just as its last statement.
- ▶ For example, consider this function:

```
def absoluteValueOf(x):  
    if x < 0:  
        return -x  
    else:  
        return x
```

MORE EXAMPLES: ABSOLUTE VALUE COMPUTATION AS A FUNCTION

- ▶ A **return** can appear within the function, not just as its last statement.
- ▶ This version works too:

```
def absoluteValueOf(x):  
    if x < 0:  
        return -x  
    return x
```

- ▶ If **x** is negative, the function returns immediately with **-x**.
 - Otherwise it continues to the next line where it then instead returns **x**.

MORE EXAMPLES: PRIMENESS COMPUTATION

- ▶ A *prime number* is a positive integer with exactly two positive divisors.
- ▶ Write a function that determines whether or not an integer is prime:

```
def isPrime(number):  
    ????
```

MORE EXAMPLES: PRIMENESS COMPUTATION

- ▶ A *prime number* is a positive integer with exactly two positive divisors.
- ▶ Here is a function that determines whether or not a number is prime:

```
def isPrime(number):  
    if number <= 1:  
        return False  
  
    # Look for a divisor..  
    check = 2  
    while check < number:  
        if number % check == 0:  
            return False  
        check = check + 1  
  
    # We've tried all the possible factors.  
    return True
```

- ▶ Note that we might **return** before the loop.
- ▶ Or we might even **return** in the middle of the loop.
 - That return behaves like a **break** statement.

SUMMARY

- ▶ A function's code consists of an indented **body** of statements.
 - These statements are ones like the top-level ones used in scripts.
- ▶ The function's lines of code compute using the **parameter** variables.
- ▶ The last line executed is a **return** statement.
 - It computes a value that gets "handed" back or *returned*.
- ▶ A function might **return** within the middle of its code, not the end.
 - In that case, it exits immediately with that value
- ▶ A function can be **called** several times within a program's code.
 - With each call, different values are passed to the function.

BREAK

MORE EXAMPLES: MIXING TYPES WITH WHAT'S RETURNED

- ▶ The function below determines whether an integer `rating` is from 1 to 10.
- ▶ It returns either the integer or a string:

```
def assessRating(rating):  
    if (rating > 0) and (rating <= 10):  
        return rating  
    else:  
        return "not a rating"
```

- ▶ Below is it in use:

```
>>> assessRating(3)  
3  
>>> assessRating(11)  
"not a rating"
```

MISSING CASES?

- ▶ What happens if you (accidentally) forget a case?

```
def example(value):  
    if value > 0:  
        return "positive"  
    elif value < 0:  
        return "negative"
```

- ▶ What happens in the missing case?

```
>>> example(3)  
'positive'  
>>> example(-4)  
'negative'  
>>> example(0)  
????
```

MISSING CASES?

- ▶ What happens if you (accidentally) forget a case?

```
def example(value):  
    if value > 0:  
        return "positive"  
    elif value < 0:  
        return "negative"
```

- ▶ What happens in the missing case?

```
>>> example(3)  
'positive'  
>>> example(-4)  
'negative'  
>>> example(0)  
>>>
```

- ▶ Python's interactive session doesn't show the **None** value!

MISSING CASES

- ▶ What happens if you (accidentally) forget a case:

```
def example(value):  
    if value > 0:  
        return "positive"  
    elif value < 0:  
        return "negative"
```

- ▶ What happens in the missing case?

```
>>> print(example(3))  
positive  
>>> print(example(4))  
negative  
>>> print(example(0))  
None
```

MISSING CASES

- ▶ What happens if you (accidentally) forget a case:

```
def example(value):  
    if value > 0:  
        return "positive"  
    elif value < 0:  
        return "negative"
```

- ▶ What happens in the missing case?

```
>>> print(repr(example(3)))  
'positive'  
>>> print(repr(example(4)))  
'negative'  
>>> print(repr(example(0)))  
'None'
```

MISSING CASES

- ▶ What happens if you (accidentally) forget a case:

```
def example(value):  
    if value > 0:  
        return "positive"  
    elif value < 0:  
        return "negative"
```

- ▶ What happens in the missing case?

```
>>> print(example(3))  
positive  
>>> print(example(4))  
negative  
>>> print(example(0))  
None
```

- ▶ There is a special Python value **None** that is implicitly returned.
- ▶ Confusingly, the interpreter does not display the **None** value.

MISSING CASES

- ▶ What happens if you (accidentally) forget a case:

```
def example(value):  
    if value > 0:  
        return "positive"  
    elif value < 0:  
        return "negative"
```

- ▶ What happens in the missing case?

```
>>> print(example(3))  
positive  
>>> print(example(4))  
negative  
>>> print(example(0))  
None
```

- ▶ There is a special Python value **None** that is implicitly returned.
- ▶ *Make sure in your functions you've an explicit **return** for every case!*

PROGRAMMER-DEFINED PROCEDURES

- ▶ Python has the same `def` syntax for defining *procedures*
 - This is my term for a "function that does not return a value."
 - Instead, it does some stuff, performs some actions.
- ▶ For example

```
def printBoxTop(size):  
    dashes = "-" * size  
    print("+ " + dashes + "+")
```

```
def printBox(width):  
    printBoxTop(width)  
    print("|" + (" " * width) + "|")  
    printBoxTop(width)
```

- ▶ Below is its use. It's as if we've invented a `printBox` statement.

```
>>> printBox(4)  
+----+  
|    |  
+----+  
>>>
```

SYNTAX: PROCEDURE DEFINITION

Below gives a template for procedure definitions:

```
def procedure-name (parameter-list) :  
    lines of statements that compute using the parameters  
    ...  
    return
```

- ▶ The last line is often a **return** statement, but it isn't needed.
- ▶ There can also be **return** statements within the code.
 - ➔ These lead Python to exit the procedure as soon as they are reached.
 - ➔ Control returns back to where the procedure was called, continues there.

EXAMPLE SCRIPT WITH PROCEDURES

```
def printBoxTop(size):
    dashes = "-" * size
    print("+ " + dashes + "+")

def greetTheUser(name):
    print("Hi, " + name + ". Nice to meet ya!")

def printBox(w):
    printBoxTop(w)
    print("|" + (" " * w) + "|")
    printBoxTop(w)

user = input("What's your name? ")
greetTheUser(user)
print("I'd like to make you a box.")
width = int(input("How wide of a box would you like? "))
printBox(width)
print("Here is one that is twice as wide:")
printBox(width * 2)
```

PROCEDURES RETURN THE **NONE** VALUE

- ▶ All three of these procedures do the exact same thing:

```
def greetThenReturn_version1(name):  
    print("Hi, " + name + ".")
```

```
def greetThenReturn_version2(name):  
    print("Hi, " + name + ".")  
    return
```

```
def greetThenReturn_version3(name):  
    print("Hi, " + name + ".")  
    return None
```

- ▶ The first implicitly returns **None**. The first explicitly returns but implicitly returns **None**. The third explicitly returns the **None** value.

NONE IS WEIRDLY HANDLED BY THE PYTHON INTERPRETER

- ▶ Here is some fun with **None**, and with procedures (that return **None**):

```
>>> print("hello")
hello
>>> print(None)
None
>>> "hello"
'hello'
>>> None
>>> 3+4
7
>>> print(print("hello"))
hello
None
>>> greetThenReturn("Jim")
Hello, Jim.
>>> print(greetThenReturn("Jim"))
Hello, Jim.
None
```

FUNCTIONS VS. PROCEDURES

- ▶ In Python, procedures are really just functions.
 - Python doesn't distinguish procedures from functions.
 - This is just my personal dichotomy, from older languages (Pascal, C).
- ▶ **"Function"**:
 - A function gets passed some parameters, executes, and then returns a result.
 - A function is used within an expression.
- ▶ **"Procedure"**:
 - A procedure is something that (typically) performs some action/work but does not return a value.
 - A procedure is used as a statement of a line of code.
 - When a procedure's work is done, Python continues executing after the line where it was called. (Control "jumps" then returns.)