MORE ABOUT CONDITIONS AND LOOPS

LECTURE 02-2

NESTED LOOPS
BREAK; CONTINUE
SHORT-CIRCUITING
CHECKING CONDITION RESULTS

JIM FIX, REED COLLEGE CSCI 121

MONDAY

- We'll start lecture with ***a short quiz***
  - Will be a short programming puzzle whose code you will write on paper.
  - It will be something like Homework 1:
    - basic Python scripting
    - `input` with prompts
    - formatting output with `print`
    - integer division (using `//` and `%`)
    - string arithmetic

GROUP EXERCISE

- Write code that computes the smallest digit of a positive integer

```
number = int(input("Enter a positive integer: "))

????
????
????

print("Its minimum digit is ", end="")
print(minimum_digit, end="")
print(".")
```

ONE SOLUTION TO THE EXERCISE

- Here is code that computes the minimum digit of a number

```
number = int(input("Enter a positive integer: "))
minimum_digit = number % 10
to_check = number // 10
while to_check > 0:
    digit = to_check % 10
    if digit < minimum_digit:
        minimum_digit = digit
    to_check = to_check // 10
print("Its minimum digit is ", end="")
print(minimum_digit, end="")
print(".")
```

ANATOMY OF A WHILE LOOP

- The template below gives the syntax of a while loop statement:

lines of statements to execute first

while ***condition-expression*** **:**

    lines of statements to execute if the condition holds

    ...

lines of statements to executed when the condition no longer holds

EXECUTION OF A WHILE LOOP

- The template below gives the syntax of a while loop statement:

    lines of "set up" statements to execute first

    while ***condition-expression*****:**

    lines of "loop body" statements to execute if the condition holds

    ...

    lines of "follow up" to execute when the condition no longer holds

- Here is how Python executes this code:

1. Executes the **set up** code.
2. It evaluates the **condition**. If `False` it *skips* to **Step 5**.
3. Otherwise, if `True`, it evaluates the **loop body**'s code.
4. It goes back to **Step 2**.
5. It executes the **follow up**, and subsequent, code.

NOTES

- Loop bodies can contain other control flow statements:
  - For example, you can have `if` statements or other `while` statements.
  - If another loop statement is inside, then it is a ***nested loop***.
  - If a `break` statement, we can jump out of the loop mid-body.
  - If a `continue` statement, we can jump back to the condition mid-body.

NESTING CONTROL STATEMENTS WITHIN A LOOP

- Of course you can put a conditional statement within a loop's body.

```
count = 0
while count < 6:
    if count % 2 == 0:
        print(str(count) + ” is even.”)
    else:
        print(str(count) + ” is odd.”)
    count = count + 1
print(“Done.”)
```

- Output of the script above:

```
0 is even.
1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

- ***What does the code above do???***

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=" ")
        b = b + 1
    print()
    a = a + 1
print("Done.")
```

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=" ")
        b = b + 1
    print()
    a = a + 1
print("Done.")
```

Inner loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=“ ”)
        b = b + 1
    print()
    a = a + 1
print(“Done.”)
```

Inner loop, along with set-up/follow-up

Outer loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=" ")
        b = b + 1
    print()
    a = a + 1
print("Done.")
```

Executed once for each value of a.

Inner loop, along with set-up/follow-up

Outer loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

BREAKING OUT OF A LOOP

- Here is another way of writing the counting loop.

```
print("Counting from 0 to 5:")
count = 0
while True:
    if count >= 6:
        break
    print(count)
    count = count + 1
print("Done.")
```

- The code uses a **`break`** statement to jump down to the follow-up code.
- If within several loops, it jumps to just after the innermost one.
- This is an artificial example
- Using **`break`** statements can sometimes make code more readable than code that expresses all the "break out" or stopping conditions.

USING CONDITION VARIABLES TO GOVERN LOOPING

- Using **break** to express other break-out conditions:

```
while count < 6:
    if somethingElseMakesMeStop(...)
        break
    ...
    count = count + 1
print(“Done.”)
```

- I worry that **break** can sometimes be missed by other coders.
- I sometimes prefer using explicit break-out conditions instead, like so:

```
done = False
while !done and count < 6:
    if somethingElseMakesMeStop(...)
        done = True
    if not done:
        ...
        count = count + 1
print(“Done.”)
```

USING CONDITION VARIABLES TO GOVERN LOOPING

- Using **`break`** to express other break-out conditions:

```
while count < 6:
    if somethingElseMakesMeStop(...)
        break
    ...
    count = count + 1
print("Done.")
```

PLEASE use `break` sparingly, and with taste.

- I worry that **`break`** can sometimes be missed by other coders.
- I usually prefer using explicit break-out conditions instead, like so:

```
done = False
while !done and count < 6:
    if somethingElseMakesMeStop(...)
        done = True
    if not done:
        ...
        count = count + 1
print("Done.")
```

CONTINUING ON IN A LOOP WITHOUT COMPLETING THE BODY

- A complex example:

```
print("Enter a series of payments, ending with 'Done'")
sum = 0

while True:
    entry = input("Enter a payment: ")
    if entry == "Done":
        break
    amount = int(entry)
    if amount < 0:
        print("A negative payment? must be a typo.")
        continue
    print("Thank you.")
    sum += amount
    print("The total so far is $" + str(sum) + ".")

print("Okay. The total is $" + str(sum) + ".")
```

- The code uses a **`break`** statement to exit the loop.
- The code uses a **`continue`** statement to skip the rest of the body, loop again.

CONTINUING ON IN A LOOP WITHOUT COMPLETING THE BODY

- A complex example:

```
print("Enter a series of payments, ending with 'Done'")
sum = 0

while True:
    entry = input("Enter a payment: ")
    if entry == "Done":
        break
    amount = int(entry)
    if amount < 0:
        print("A negative payment? must be a typo.")
        continue
    print("Thank you.")
    sum += amount
    print("The total so far is $" + str(sum) + ".")

print("Okay. The total is $" + str(sum) + ".")
```

PLEASE use* `continue` *sparingly, and with taste.

- The code uses a **`break`** statement to exit the loop.
- The code uses a **`continue`** statement to skip the rest of the body, loop again.

SHORT-CIRCUITED LOGIC CONNECTIVES

- Evaluation of `and` and `or` is ***short-circuited***:

```
>>> x = 0
>>> 45 / x
ERROR!!!
>>> (x == 0) or ((45 / x) > 10)
True
>>> (x != 0) and ((45 / x) > 10)
False
```

- Python doesn't bother with the right of `or` if the left is `True`.
- Python doesn't bother with the right of `and` if the left is `False`.
- This means the result of the `and` is executed *kind of like this*:

```
if x != 0:
    result_of_and = (45 / x) > 10
else:
    result_of_and = False
```

CHECKING BOOLEAN VALUES

- Many beginning programmers are tempted to write this code:

```
all_correct = (passed == tested)
print("Your code passed " + str(passed))
print(" out of " + str(tested) + "tests.")
if all_correct == True:
    print("Your code passed all our tests!")
    if not on_time:
        print("But you submitted after the deadline.")
```

CHECKING BOOLEAN VALUES IS REDUNDANT

- Many beginning programmers are tempted to write this code:

```
all_correct = (passed == tested)
print("Your code passed " + str(passed))
print(" out of " + str(tested) + "tests.")
if all_correct == True:
    print("Your code passed all our tests!")
    if not on_time:
        print("But you submitted after the deadline.")
```

CHECKING BOOLEAN VALUES IS REDUNDANT

- Write this code instead:

```
all_correct = (passed == tested)
print("Your code passed " + str(passed))
print(" out of " + str(tested) + "tests.")
if all_correct == True:
    print("Your code passed all our tests!")
    if not on_time:
        print("But you submitted after the deadline.")
```

- By using `if`, you are *already checking* whether the condition `== True`.

CHECKING BOOLEAN VALUES IS REDUNDANT

- Write this code instead:

```
all_correct = (passed == tested)
print("Your code passed " + str(passed))
print(" out of " + str(tested) + "tests.")
if all_correct:
    print("Your code passed all our tests!")
    if not on_time:
        print("But you submitted after the deadline.")
```

- By using `if`, you are *already checking* whether the condition `== True`.

PROGRAMMER-DEFINED FUNCTIONS

LECTURE 02-2 (CONT'D)

JIM FIX, REED COLLEGE CSCI 121

READING

- Today's lecture material can be supplemented with:
  - **Reading**:
    - Ch. 3, 6 (functions)
    - CP 1.3-1.4 (user-defined functions)

PROGRAMMER-DEFINED FUNCTIONS

- You introduce new functions, and their code, with a `def` statement.
- The code below defines a squaring function:

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

- Here it is in use:

```
>>> square(4)
16
>>> y = 5
>>> square(y)
25
>>> square(y+2)
49
```

- It takes a single value as its parameter. It returns back the square of that value.

PROGRAMMER-DEFINED FUNCTIONS

- The code below computes the distance between two locations on a map:

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

- Here it is in use:

```
>>> distanceFromTo(1.5,2,4.5,6)
5.0
```

- It takes four values as parameters, and returns a value back.

PROGRAMMER-DEFINED FUNCTIONS

- This calculates the gains on an amount due to a yearly rate of interest:

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

- Here it is in use:

```
>>> gains(100,5,2)
10.25
>>> print(gains(100,5,1))
5.0
>>> a0 = 100
>>> a1 = a0 + gains(a0,5,1)
>>> a2 = a1 + gains(a1,5,1)
>>> a2
110.25
```

INDENTATION

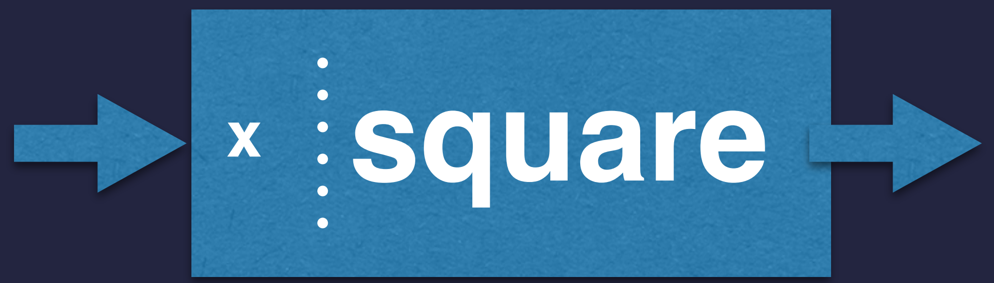
- Python reads the functions, looking for its indented lines of code

```
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

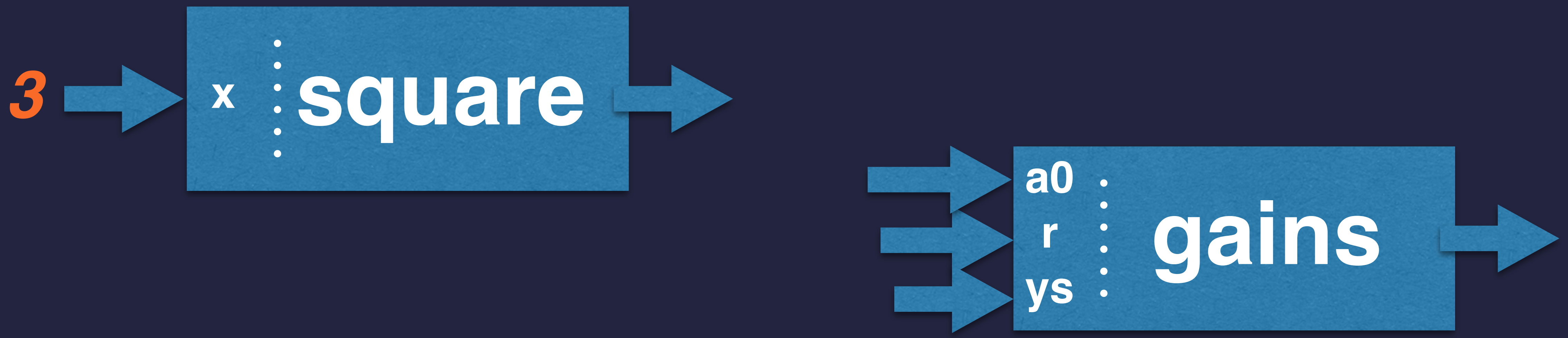
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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:



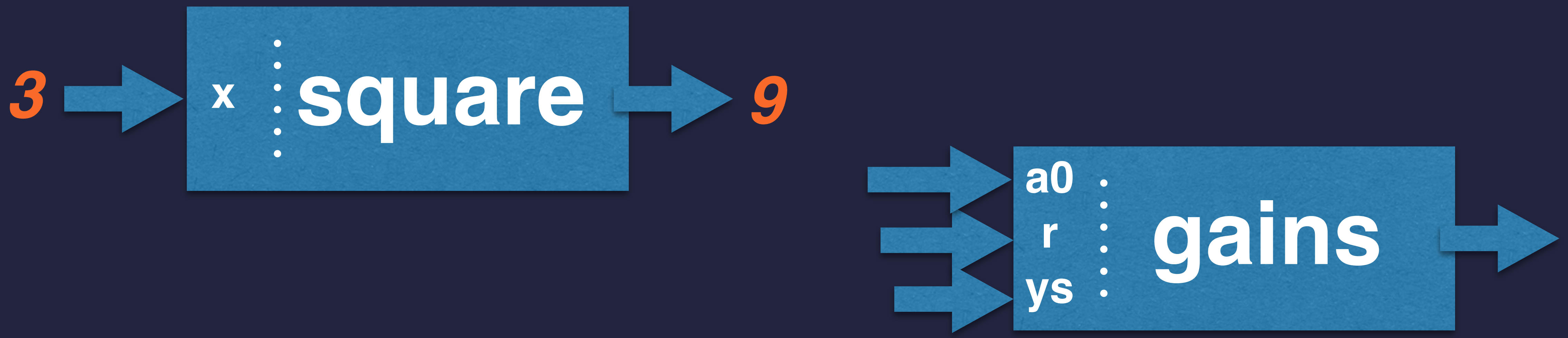
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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:



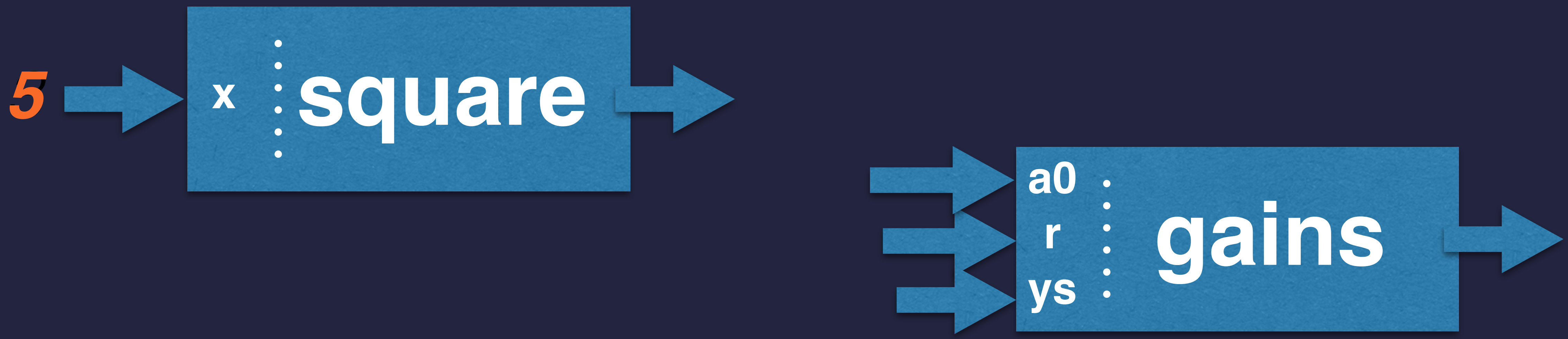
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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:



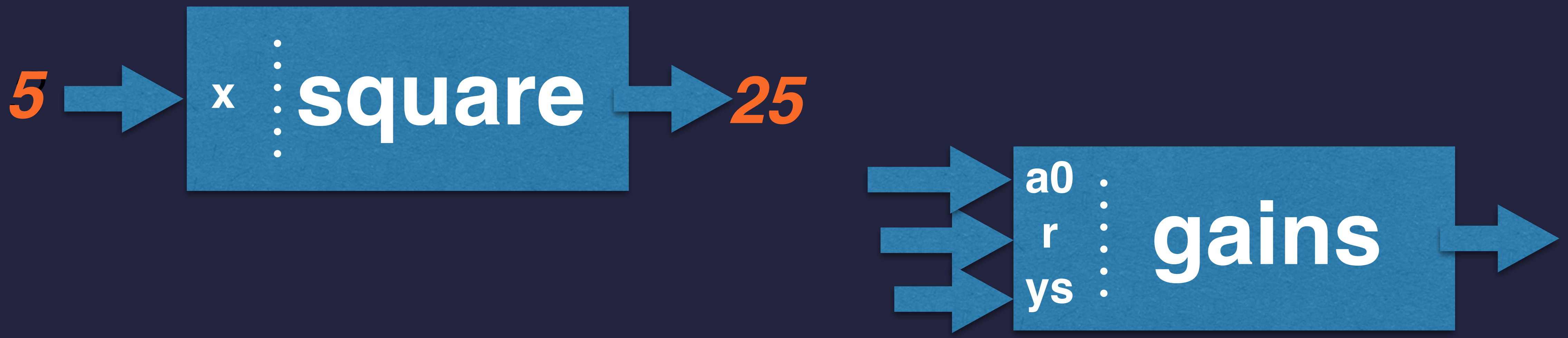
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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:



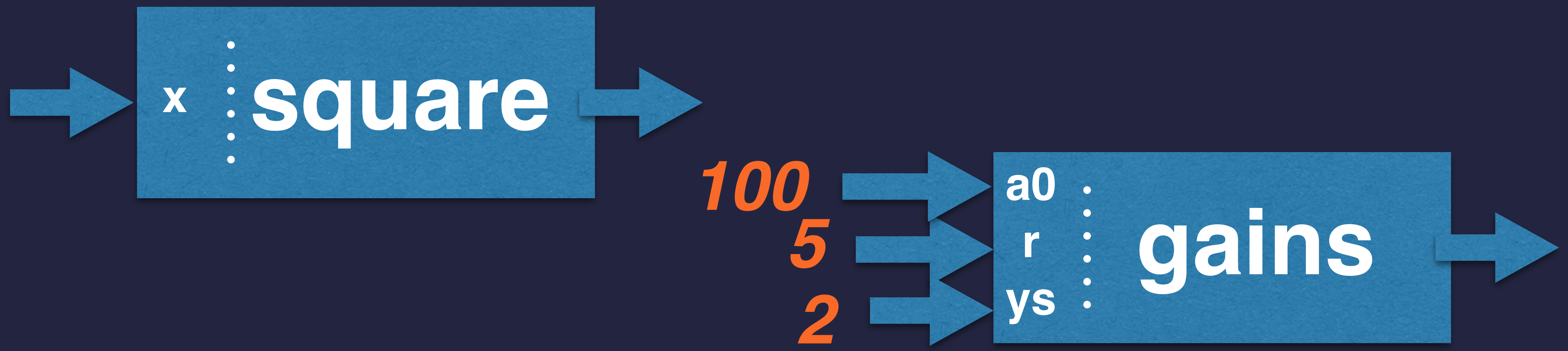
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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:



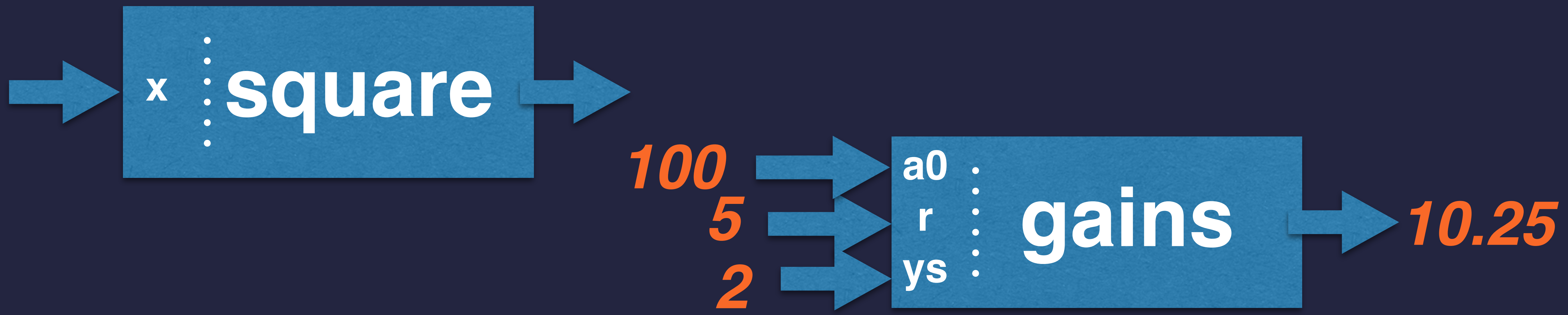
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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:



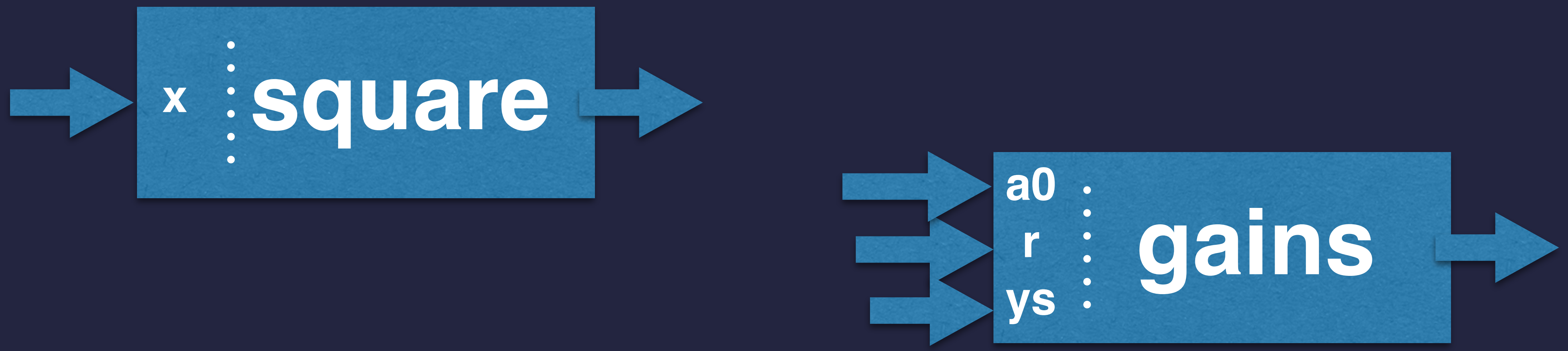
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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:



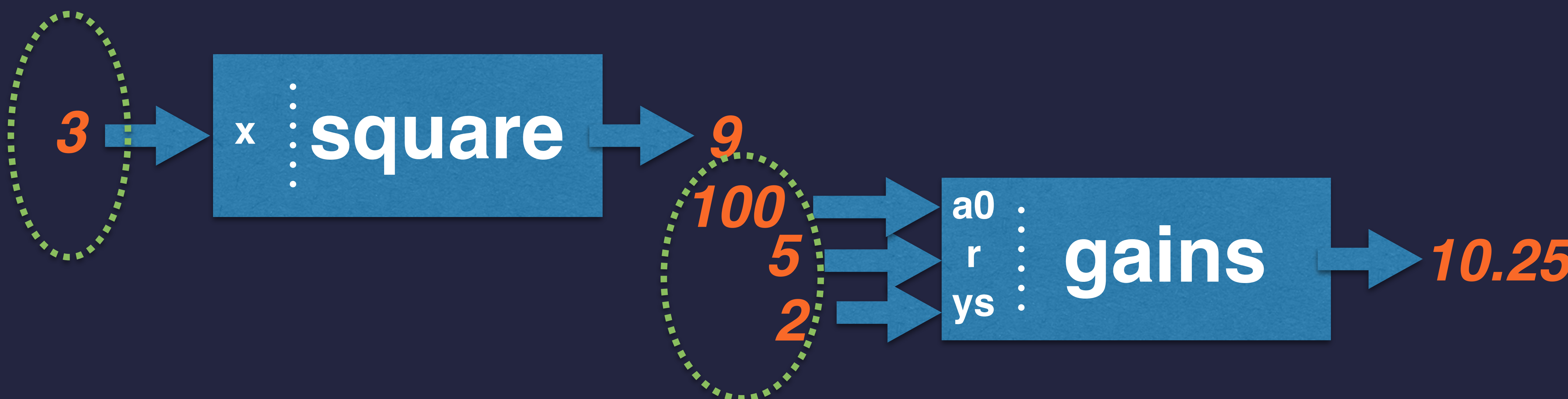
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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

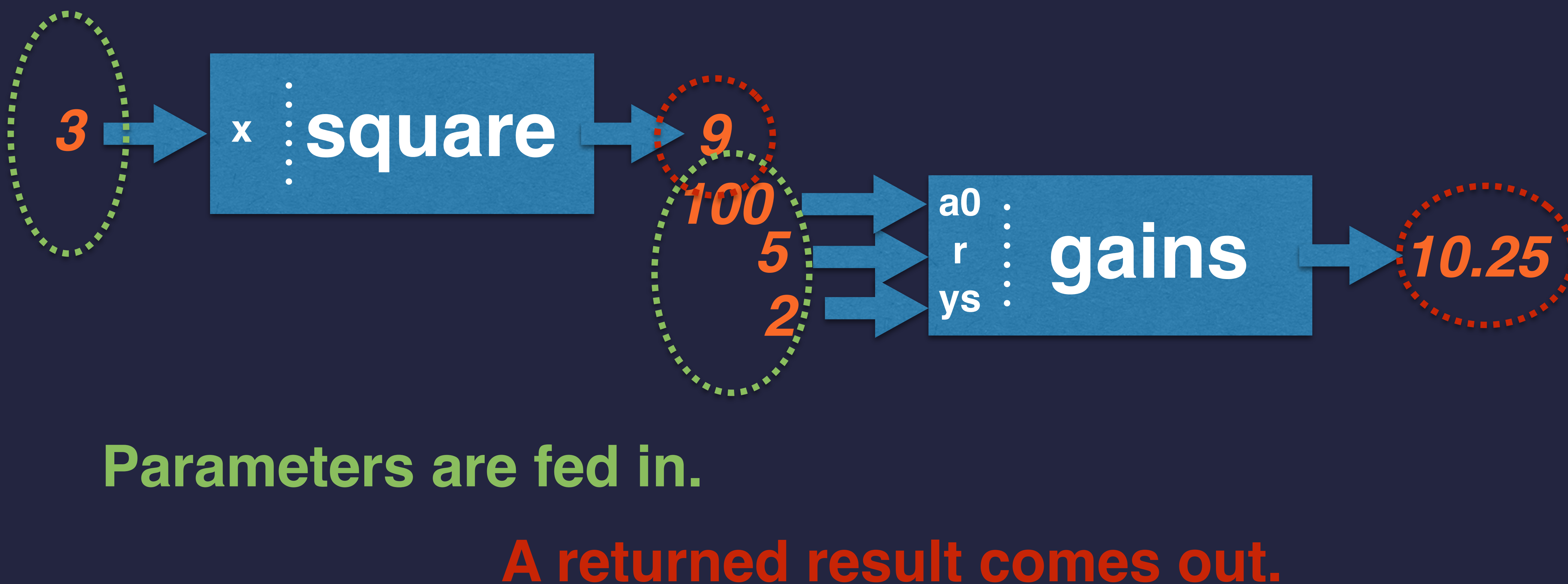
- A function takes one or more *parameter* values.
- It uses those values to compute its result.
- It then *returns* the result back to the calling expression.
- Functions can be thought of as “value factories” of a program:



Parameters are fed in.

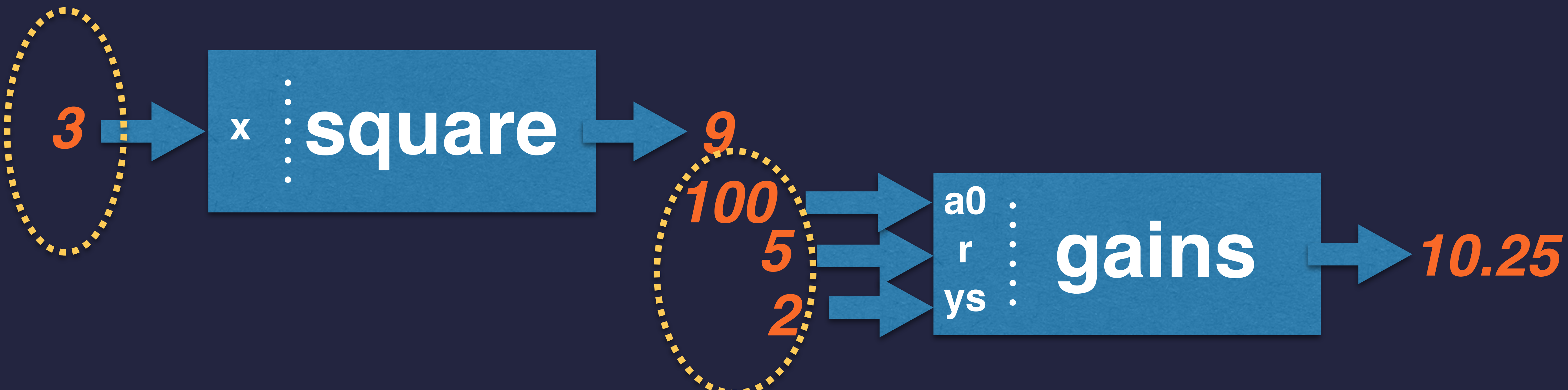
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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:



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*.

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)
```

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 often a `return` statement.

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.