MORE ON CONDITIONAL EXCECUTION

LECTURE 02-2 PART 1

THOUGHTS ON HOMEWORK 1 AND 2
USING ELIF
NESTED LOOPS; BREAK; CONTINUE

JIM FIX, REED COLLEGE CSCI 121

THIS 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
- **Note:** We've posted our feedback about Homework 1 on Gradescope.
  - ➡ Gave an additional 0 to 10 points for good ***style*** and correct ***approach***.

VARIOUS

- Some things:
  - Comments, variable names, data conversions, other style issues, …
  - Using `elif`
  - Extracting digits:
    - ones digit
      - `ones = number - (number // 10) * 10`
      -
    - integer with same last two digits as number
    - tens digit, hundreds digit, etc.
  - Group exercise with digits

VARIOUS

- Some things:
  - Comments, variable names, data conversions, other style issues, …
  - Using `elif`
  - Extracting digits:
    - ones digit
      - `ones = number - (number // 10) * 10`
      - `ones = number % 10`
    - integer with same last two digits as number
    - tens digit, hundreds digit, etc.
  - Group exercise with digits

SYNTAX: CASCADING IF-ELIF-...-ELSE STATEMENT

Below is a template for conditional statements:

```
if condition-1:
    execute if condition 1 holds
    ...
elif condition-2:
    execute if condition 1 does not hold but condition 2 does
    ...
...
else:
    executed if no condition above holds
    ...
lines of code executed after, in all cases
```

CASCADING IF STATEMENT

- Here is some other autograder code using the cascading conditional:

```
attempts = number_previous_submissions + 1
mesg =  "Great work passing all the tests!\n"
mesg += "You submitted " + str(attempts) + " times.\n"
if attempts <= 2:
    mesg += "You earned the full points.\n"
    mesg += "Excellent!"
elif attempts <= 6:
    mesg += "You earned 80% of the points.\n"
    mesg += "Nicely done."
else:
    mesg += "This is a few more times than we'd prefer.\n"
    mesg += "We awarded half of the points."

print(mesg)
```

SYNTAX: CASCADING IF–ELIF–...–ELIF STATEMENT

Below is a template for conditional statements:

```
if condition-1:
    execute if condition1 holds
    ...
elif condition-2:
    execute if condition1 does not hold but condition2 does
    ...
...
elif condition-n:
    execute if conditions 1 through (n-1) do not hold but condition n does
    ...
lines of code executed after, in all cases
```

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(".")
```

BACK TO WHILE LOOPS...

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

- *It outputs a multiplication table, organized by tab characters:*

```
    1   2   3   4   5   6   7   8
    2   4   6   8   10  12  14  16
    3   6   9   12  15  18  21  24
    4   8   12  16  20  24  28  32
    5   10  15  20  25  30  35  40
    6   12  18  24  30  36  42  48
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

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

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

- ***It outputs a multiplication table, organized by tab characters:***

```
    1   2   3   4   5   6   7   8
    2   4   6   8   10  12  14  16
    3   6   9   12  15  18  21  24
    4   8   12  16  20  24  28  32
    5   10  15  20  25  30  35  40
    6   12  18  24  30  36  42  48
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 1
while a <= 6:
    b = 1
    while b <= 8:
        print("\t"str(a*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 multiplication table, organized by tab characters:***

```
    1   2   3   4   5   6   7   8
    2   4   6   8   10  12  14  16
    3   6   9   12  15  18  21  24
    4   8   12  16  20  24  28  32
    5   10  15  20  25  30  35  40
    6   12  18  24  30  36  42  48
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*b).rjust(4), 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 multiplication table, organized by tab characters:*

```
   0   0   0   0   0   0   0   0
   0   1   2   3   4   5   6   7
   0   2   4   6   8  10  12  14
   0   3   6   9  12  15  18  21
   0   4   8  12  16  20  24  28
   0   5  10  15  20  25  30  35
   0   6  12  18  24  30  36  42
Done.
```

RECALL: THE TEMPLATE (WAS: ANATOMY OF A COUNTING LOOP)

- Here is the standard structure of a "counting loop":

*initialize the **count** to the **start-value***

`while` ***count*** **`<`** ***one-too-far*** **`:`**

    *actions to perform with that particular **count** value*

    *increment the **count** by 1*

*at this point can now use the fact that **count** == **one-too-far***

- This is an extremely common coding pattern...

➡ **PLEASE TAKE THIS TEMPLATE TO HEART!!!!**

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
```

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

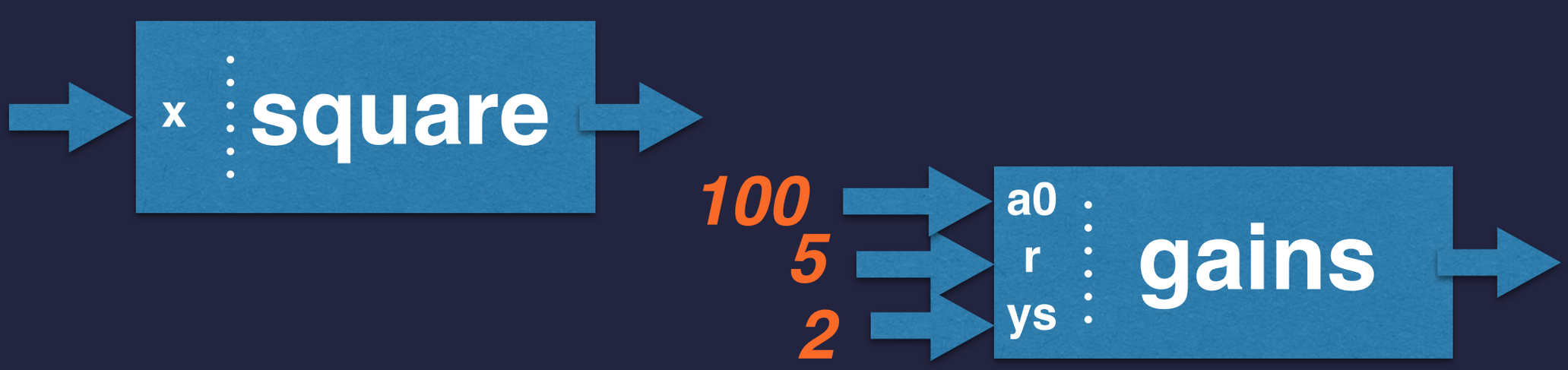
- A function takes one or more *parameter* values.
- It uses those values to compute its result.
- It then *returns* the result back to the calling expression.
- Functions can be thought of as "value factories" of a program:

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:

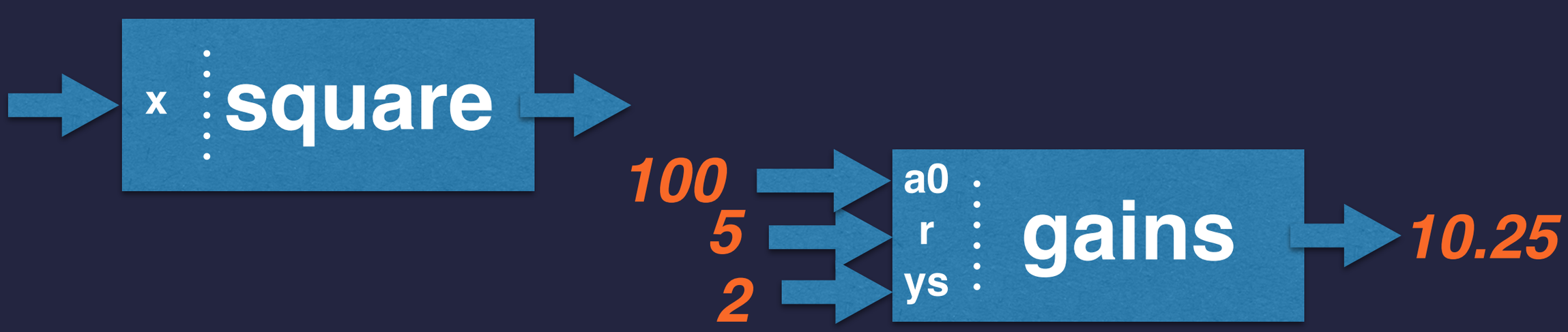
FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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:

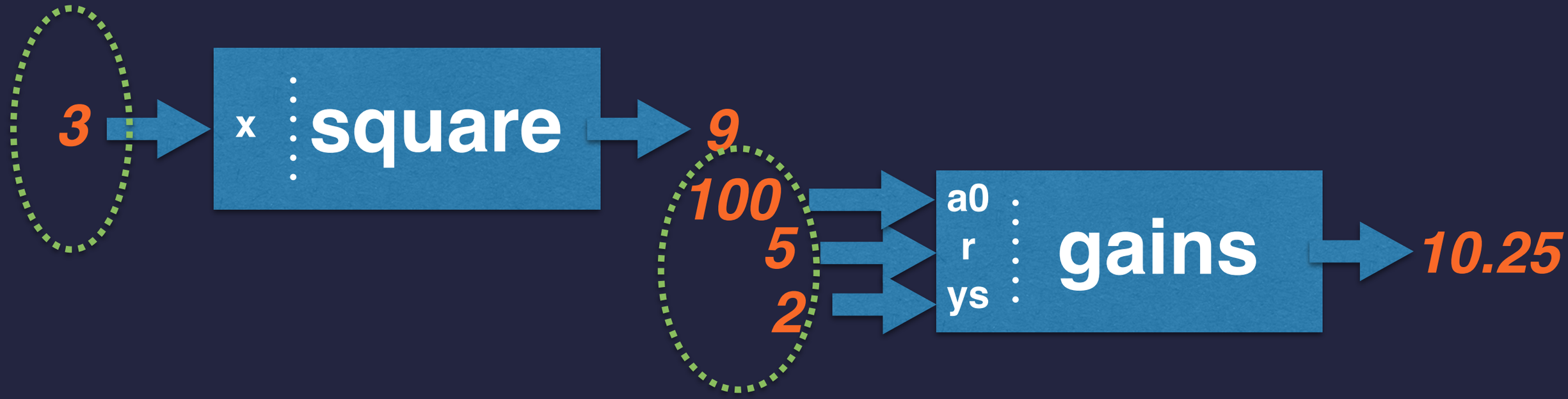


FUNCTIONS COMPUTE VALUES FROM THEIR PARAMETERS

- 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

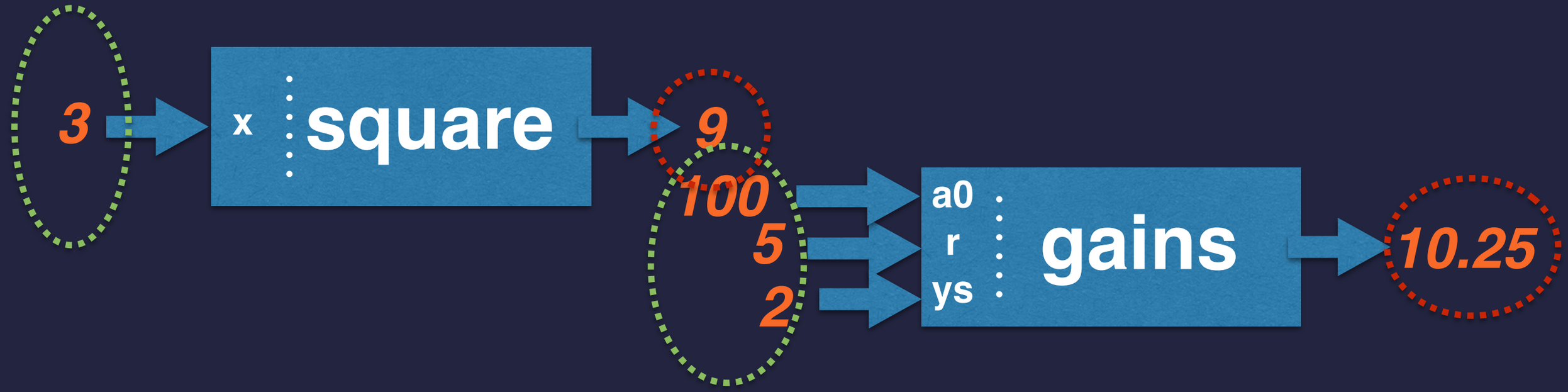
- A function takes one or more *parameter* values.
- It uses those values to compute its result.
- It then *returns* the result back to the calling expression.
- Functions can be thought of as "value factories" of a program:



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:

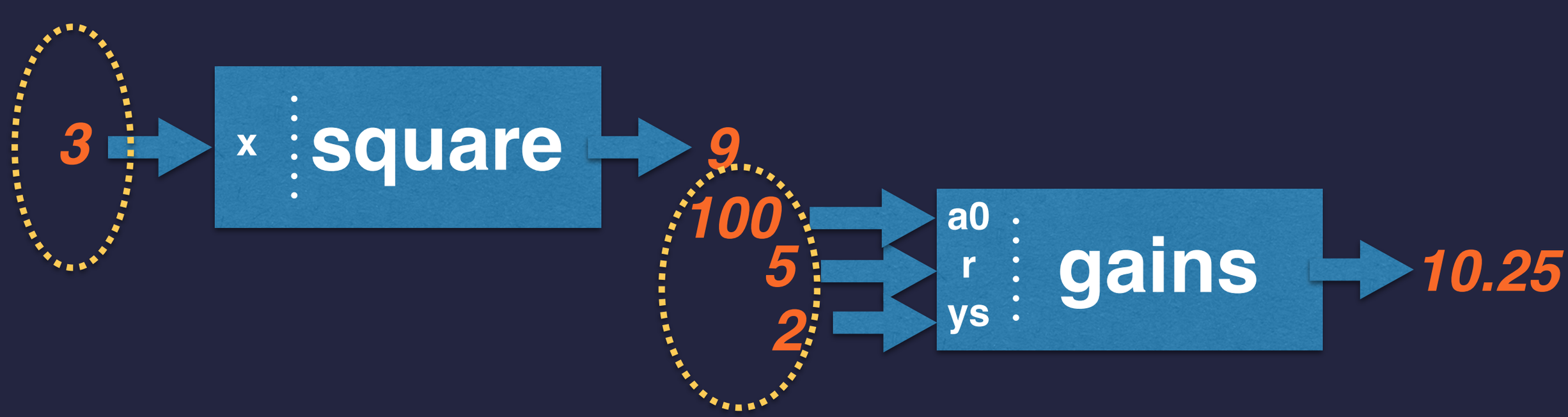


Parameters are fed in.

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*.

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.

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

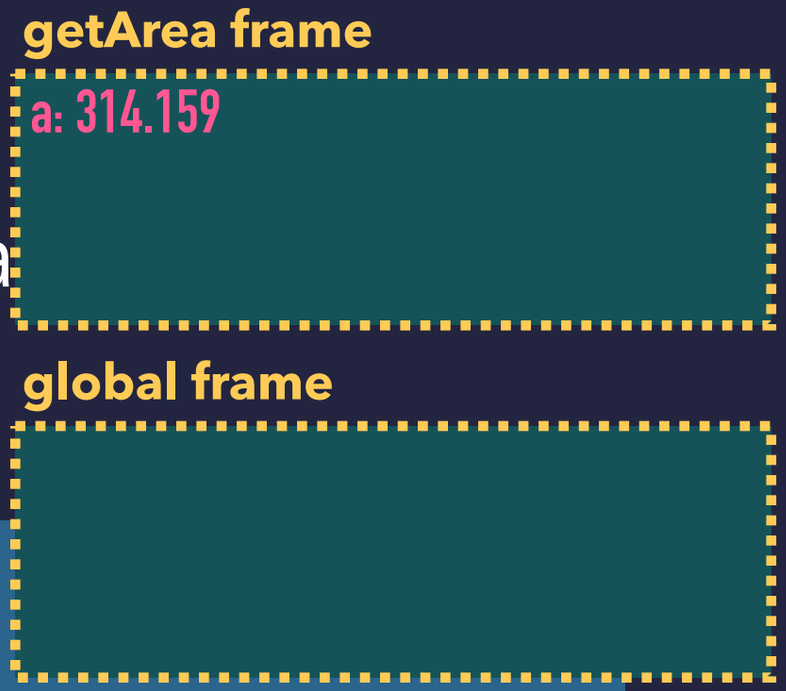
```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
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 crea
  local variables.




```
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)+”.”)
```

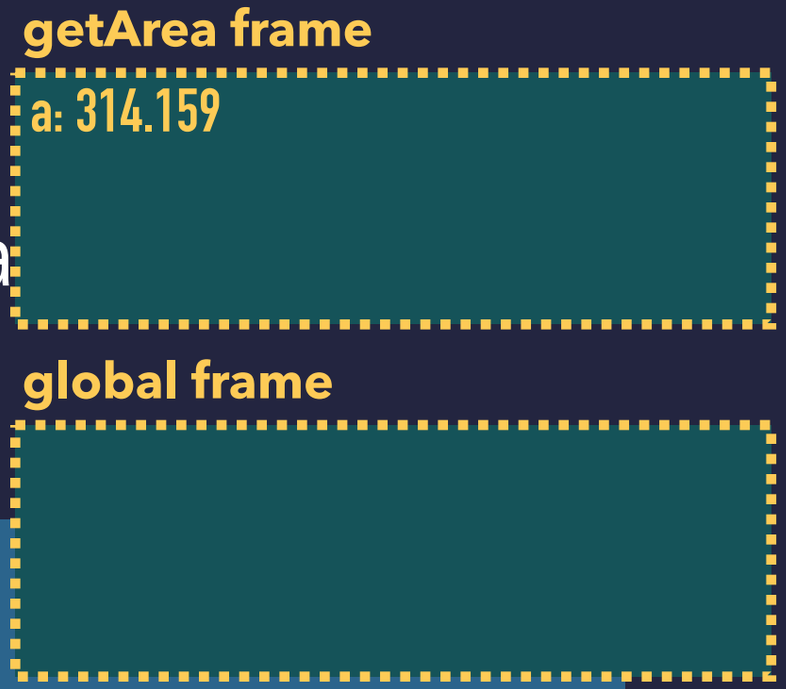
LOCAL VS. GLOBAL FRAMES

- When a function gets called, a *local frame* gets crea
  local variables.

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

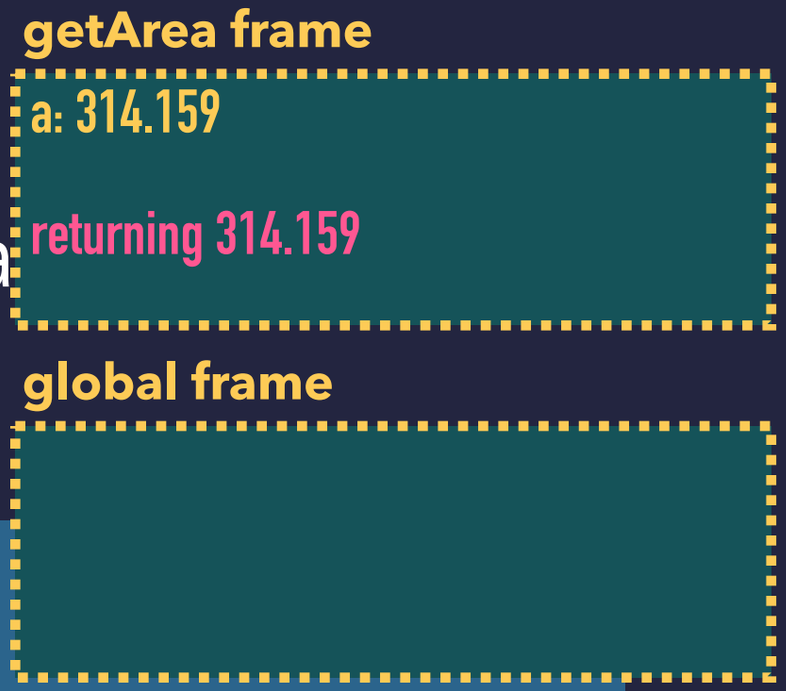
```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
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 crea
  local variables.



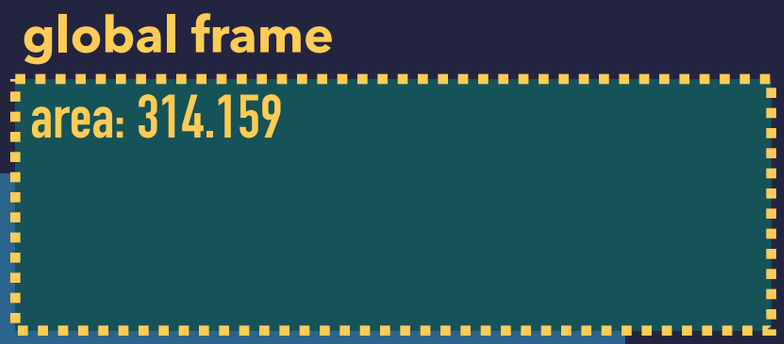

```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
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.




```
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 crea
  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 crea
  local variables.

```
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)+”.”)
```

radiusOfCircle frame

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

global frame

area: 314.159

LOCAL VS. GLOBAL FRAMES

- When a function gets called, a *local frame* gets crea
  local variables.

```
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)+”.”)
```

radiusOfCircle frame

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

returning 0.9999995776679783

global frame

area: 314.159

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.

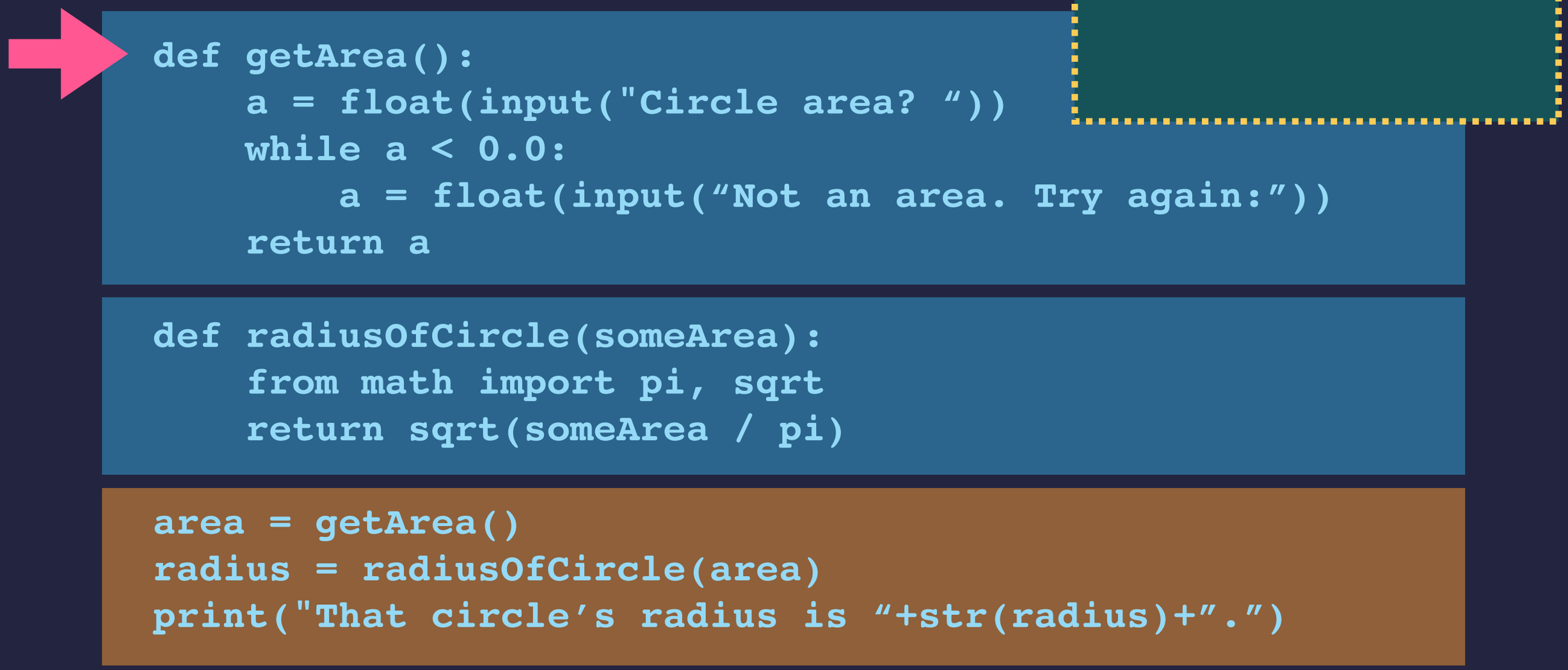
```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
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.



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.
- Function 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 exacrtly 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 exacrtly 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.