You can order print and ebook versions of *Think Python 3e* from Bookshop.org and Amazon.

In [1]:
```
from os.path import basename, exists

def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve

        local, _ = urlretrieve(url, filename)
        print("Downloaded " + str(local))
    return filename

download('https://github.com/AllenDowney/ThinkPython/raw/v3/thinkpython.py')
download('https://github.com/AllenDowney/ThinkPython/raw/v3/diagram.py');

import thinkpython
```

```
Downloaded thinkpython.py
Downloaded diagram.py
```

Return Values

In previous chapters, we've used built-in functions -- like `abs` and `round` -- and functions in the math module -- like `sqrt` and `pow`. When you call one of these functions, it returns a value you can assign to a variable or use as part of an expression.

The functions we have written so far are different. Some use the `print` function to display values, and some use turtle functions to draw figures. But they don't return values we assign to variables or use in expressions.

In this chapter, we'll see how to write functions that return values.

Some functions have return values

When you call a function like `math.sqrt`, the result is called a **return value**. If the function call appears at the end of a cell, Jupyter displays the return value immediately.

In [2]:
```
import math

math.sqrt(42 / math.pi)
```

Out[2]:
```
3.656366395715726
```

If you assign the return value to a variable, it doesn't get displayed.

In [3]:
```
radius = math.sqrt(42 / math.pi)
```

But you can display it later.

In [4]:
```
radius
```

Out[4]:
```
3.656366395715726
```

Or you can use the return value as part of an expression.

In [5]:
```
radius + math.sqrt(42 / math.pi)
```

Out[5]:
```
7.312732791431452
```

Here's an example of a function that returns a value.

In [6]:
```
def circle_area(radius):
    area = math.pi * radius**2
    return area
```

`circle_area` takes `radius` as a parameter and computes the area of a circle with that radius.

The last line is a `return` statement that returns the value of `area`.

If we call the function like this, Jupyter displays the return value.

In [7]:
```
circle_area(radius)
```

Out[7]:
```
42.00000000000001
```

We can assign the return value to a variable.

In [8]:
```
a = circle_area(radius)
```

Or use it as part of an expression.

In [9]:
```
circle_area(radius) + 2 * circle_area(radius / 2)
```

Out[9]:
```
63.000000000000014
```

Later we can display the value of the variable we assigned the result to.

In [10]:
```
a
```

Out[10]:
```
42.00000000000001
```

But we can't access `area`.

In [11]:
```
%%expect NameError

area
```

```
NameError: name 'area' is not defined
```

`area` is a local variable in a function, so we can't access it from outside the function.

And some have None

If a function doesn't have a `return` statement, it returns `None`, which is a special value like `True` and `False`. For example, here's the `repeat` function from Chapter 3.

In [12]:
```
def repeat(word, n):
    print(word * n)
```

If we call it like this, it displays the first line of the Monty Python song "Finland".

In [13]:
```
repeat('Finland, ', 3)
```

```
Finland, Finland, Finland,
```

This function uses the `print` function to display a string, but it does not use a `return` statement to return a value. If we assign the result to a variable, it displays the string anyway.

In [14]:
```
result = repeat('Finland, ', 3)
```

```
Finland, Finland, Finland,
```

And if we display the value of the variable, we get nothing.

In [15]:
```
result
```

`result` actually has a value, but Jupyter doesn't show it. However, we can display it like this.

In [16]:
```
print(result)
```

```
None
```

The return value from `repeat` is `None`.

Now here's a function similar to `repeat` except that has a return value.

In [17]:
```
def repeat_string(word, n):
    return word * n
```

Notice that we can use an expression in a `return` statement, not just a variable.

With this version, we can assign the result to a variable. When the function runs, it doesn't display anything.

In [18]:
```
line = repeat_string('Spam, ', 4)
```

But later we can display the value assigned to `line`.

In [19]:
```
line
```

Out[19]:
```
'Spam, Spam, Spam, Spam, '
```

A function like this is called a **pure function** because it doesn't display anything or have any other effect -- other than returning a value.

Return values and conditionals

If Python did not provide `abs`, we could write it like this.

In [20]:
```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

If `x` is negative, the first `return` statement returns `-x` and the function ends immediately. Otherwise, the second `return` statement returns `x` and the function ends. So this function is correct.

However, if you put `return` statements in a conditional, you have to make sure that every possible path through the program hits a `return` statement. For example, here's an incorrect version of `absolute_value`.

In [21]:
```
def absolute_value_wrong(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

Here's what happens if we call this function with `0` as an argument.

In [22]:
```
absolute_value_wrong(0)
```

We get nothing! Here's the problem: when `x` is `0`, neither condition is true, and the function ends without hitting a `return` statement, which means that the return value is `None`, so Jupyter displays nothing.

As another example, here's a version of `absolute_value` with an extra `return` statement at the end.

In [23]:
```
def absolute_value_extra_return(x):
    if x < 0:
        return -x
    else:
        return x
```

```
    return 'This is dead code'
```

If `x` is negative, the first `return` statement runs and the function ends. Otherwise the second `return` statement runs and the function ends. Either way, we never get to the third `return` statement -- so it can never run.

Code that can never run is called **dead code**. In general, dead code doesn't do any harm, but it often indicates a misunderstanding, and it might be confusing to someone trying to understand the program.

Incremental development

As you write larger functions, you might find yourself spending more time debugging. To deal with increasingly complex programs, you might want to try **incremental development**, which is a way of adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points represented by the coordinates (x_1, y_1) and (x_2, y_2). By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a `distance` function should look like in Python -- that is, what are the inputs (parameters) and what is the output (return value)?

For this function, the inputs are the coordinates of the points. The return value is the distance. Immediately you can write an outline of the function:

In [24]:
```
def distance(x1, y1, x2, y2):
    return 0.0
```

This version doesn't compute distances yet -- it always returns zero. But it is a complete function with a return value, which means that you can test it before you make it more complicated.

To test the new function, we'll call it with sample arguments:

In [25]:
```
distance(1, 2, 4, 6)
```

Out[25]:
```
0.0
```

I chose these values so that the horizontal distance is `3` and the vertical distance is `4`. That way, the result is `5`, the hypotenuse of a `3-4-5` right triangle. When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function runs and returns a value, and we can start adding code to the body. A good next step is to find the differences `x2 - x1` and `y2 -`

`y1` . Here's a version that stores those values in temporary variables and displays them.

In [26]:
```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

If the function is working, it should display `dx is 3` and `dy is 4` . If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

In [ ]:
```
distance(1, 2, 4, 6)
```

Good so far. Next we compute the sum of squares of `dx` and `dy` :

In [27]:
```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

Again, we can run the function and check the output, which should be `25` .

In [28]:
```
distance(1, 2, 4, 6)
```

```
dsquared is:  25
```

Out[28]:
```
0.0
```

Finally, we can use `math.sqrt` to compute the distance:

In [29]:
```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    print("result is", result)
```

And test it.

In [30]:
```
distance(1, 2, 4, 6)
```

```
result is 5.0
```

The result is correct, but this version of the function displays the result rather than returning it, so the return value is `None` .

We can fix that by replacing the `print` function with a `return` statement.

In [31]:
```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

This version of `distance` is a pure function. If we call it like this, only the result is displayed.

In [32]:
```
distance(1, 2, 4, 6)
```

Out[32]: 5.0

And if we assign the result to a variable, nothing is displayed.

In [33]:
```
d = distance(1, 2, 4, 6)
```

The `print` statements we wrote are useful for debugging, but once the function is working, we can remove them. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product.

This example demonstrates incremental development. The key aspects of this process are:

1. Start with a working program, make small changes, and test after every change.
2. Use variables to hold intermediate values so you can display and check them.
3. Once the program is working, remove the scaffolding.

At any point, if there is an error, you should have a good idea where it is. Incremental development can save you a lot of debugging time.

Boolean functions

Functions can return the boolean values `True` and `False`, which is often convenient for encapsulating a complex test in a function. For example, `is_divisible` checks whether `x` is divisible by `y` with no remainder.

In [34]:
```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

Here's how we use it.

In [35]:
```
is_divisible(6, 4)
```

Out[35]: False

In [36]:
```
is_divisible(6, 3)
```

Out[36]: True

Inside the function, the result of the == operator is a boolean, so we can write the function more concisely by returning it directly.

In [37]:
```
def is_divisible(x, y):
    return x % y == 0
```

Boolean functions are often used in conditional statements.

In [38]:
```
if is_divisible(6, 2):
    print('divisible')
```

```
divisible
```

It might be tempting to write something like this:

In [39]:
```
if is_divisible(6, 2) == True:
    print('divisible')
```

```
divisible
```

But the comparison is unnecessary.

Recursion with return values

Now that we can write functions with return values, we can write recursive functions with return values, and with that capability, we have passed an important threshold -- the subset of Python we have is now **Turing complete**, which means that we can perform any computation that can be described by an algorithm.

To demonstrate recursion with return values, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition refers to the thing being defined. A truly circular definition is not very useful:

> vorpal: An adjective used to describe something that is vorpal.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol !, you might get something like this:

$$0! = 1$$
$$n! = n\ (n-1)!$$

This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n - 1$.

If you can write a recursive definition of something, you can write a Python program to evaluate it. Following an incremental development process, we'll start with a function that take `n` as a parameter and always returns `0`.

In [1]:
```
def factorial(n):
    return 0
```

Now let's add the first part of the definition -- if the argument happens to be `0`, all we have to do is return `1`:

In [2]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return 0
```

Now let's fill in the second part -- if `n` is not `0`, we have to make a recursive call to find the factorial of `n-1` and then multiply the result by `n`:

In [4]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        return n * recurse
```

The flow of execution for this program is similar to the flow of `countdown` in Chapter 5. If we call `factorial` with the value `3`:

Since `3` is not `0`, we take the second branch and calculate the factorial of `n-1` ...

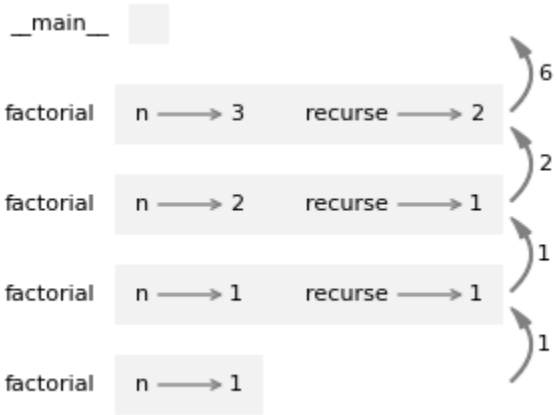
> Since `2` is not `0`, we take the second branch and calculate the factorial of `n-1` ...
>
> > Since `1` is not `0`, we take the second branch and calculate the factorial of `n-1` ...
> >
> > > Since `0` equals `0`, we take the first branch and return `1` without making any more recursive calls.
> >
> > The return value, `1`, is multiplied by `n`, which is `1`, and the result is returned.
>
> The return value, `1`, is multiplied by `n`, which is `2`, and the result is returned.

The return value `2` is multiplied by `n`, which is `3`, and the result, `6`, becomes the return value of the function call that started the whole process.

The following figure shows the stack diagram for this sequence of function calls.

In [4]:
```
from diagram import Frame, Stack, make_binding

main = Frame([], name='__main__', loc='left')
frames = [main]

ns = 3, 2, 1
recurses = 2, 1, 1
results = 6, 2, 1

for n, recurse, result in zip(ns, recurses, results):
    binding1 = make_binding('n', n)
    binding2 = make_binding('recurse', recurse)
    frame = Frame([binding1, binding2],
                  name='factorial', value=result,
                  loc='left', dx=1.2)
    frames.append(frame)

binding1 = make_binding('n', n)
frame = Frame([binding1], name='factorial', value=1,
              shim=1.2, loc='left', dx=1.4)
frames.append(frame)

stack = Stack(frames, dy=-0.45)
```

In [5]:
```
from diagram import diagram, adjust

width, height, x, y = [2.74, 2.26, 0.73, 2.05]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```



The return values are shown being passed back up the stack. In each frame, the return value is the product of `n` and `recurse`.

In the last frame, the local variable `recurse` does not exist because the branch that creates it does not run.

Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become overwhelming. An alternative is what I call the "leap of faith". When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the right result.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `abs` or `math.sqrt`, you don't examine the bodies of those functions -- you just assume that they work.

The same is true when you call one of your own functions. For example, earlier we wrote a function called `is_divisible` that determines whether one number is divisible by another. Once we convince ourselves that this function is correct, we can use it without looking at the body again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works and then ask yourself, "Assuming that I can compute the factorial of $n-1$, can I compute the factorial of n?" The recursive definition of factorial implies that you can, by multiplying by n.

Of course, it's a bit strange to assume that the function works correctly when you haven't finished writing it, but that's why it's called a leap of faith!

Fibonacci

After `factorial`, the most common example of a recursive function is `fibonacci`, which has the following definition:

$$\begin{aligned}\text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2)\end{aligned}$$

Translated into Python, it looks like this:

In [2]:
```
def fibonacci(n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

If you try to follow the flow of execution here, even for small values of n, your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, you can be confident that the last `return` statement is correct.

As an aside, this way of computing Fibonacci numbers is very inefficient. In Chapter 10 I'll explain why and suggest a way to improve it.

Checking types

What happens if we call `factorial` and give it `1.5` as an argument?

In [5]:
```
#%%expect RecursionError

factorial(1.5)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[5], line 3
      1 #%%expect RecursionError
----> 3 factorial(1.5)

Cell In[4], line 5, in factorial(n)
      3     return 1
      4 else:
----> 5     recurse = factorial(n-1)
      6     return n * recurse

Cell In[4], line 5, in factorial(n)
      3     return 1
      4 else:
----> 5     recurse = factorial(n-1)
      6     return n * recurse

    [... skipping similar frames: factorial at line 5 (2974 times)]

Cell In[4], line 5, in factorial(n)
      3     return 1
      4 else:
----> 5     recurse = factorial(n-1)
      6     return n * recurse

RecursionError: maximum recursion depth exceeded
```

It looks like an infinite recursion. How can that be? The function has a base case -- when `n`

`== 0`. But if `n` is not an integer, we can *miss* the base case and recurse forever.

In this example, the initial value of `n` is `1.5`. In the first recursive call, the value of `n` is `0.5`. In the next, it is `-0.5`. From there, it gets smaller (more negative), but it will never be `0`.

To avoid infinite recursion we can use the built-in function `isinstance` to check the type of the argument. Here's how we check whether a value is an integer.

In [6]:
```
isinstance(3, int)
```

Out[6]:
```
True
```

In [7]:
```
isinstance(1.5, int)
```

Out[7]:
```
False
```

Now here's a version of `factorial` with error-checking.

In [8]:
```
def factorial(n):
    if not isinstance(n, int):
        print('factorial is only defined for integers.')
        return None
    elif n < 0:
        print('factorial is not defined for negative numbers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

First it checks whether `n` is an integer. If not, it displays an error message and returns `None`.

In [9]:
```
factorial('crunchy frog')
```

```
factorial is only defined for integers.
```

Then it checks whether `n` is negative. If so, it displays an error message and returns `None.`

In [10]:
```
factorial(-2)
```

```
factorial is not defined for negative numbers.
```

If we get past both checks, we know that `n` is a non-negative integer, so we can be confident the recursion will terminate. Checking the parameters of a function to make sure they have the correct types and values is called **input validation**.

Debugging

Breaking a large program into smaller functions creates natural checkpoints for debugging. If a function is not working, there are three possibilities to consider:

- There is something wrong with the arguments the function is getting -- that is, a precondition is violated.
- There is something wrong with the function -- that is, a postcondition is violated.
- The caller is doing something wrong with the return value.

To rule out the first possibility, you can add a `print` statement at the beginning of the function that displays the values of the parameters (and maybe their types). Or you can write code that checks the preconditions explicitly.

If the parameters look good, you can add a `print` statement before each `return` statement and display the return value. If possible, call the function with arguments that make it easy check the result.

If the function seems to be working, look at the function call to make sure the return value is being used correctly -- or used at all!

Adding `print` statements at the beginning and end of a function can help make the flow of execution more visible. For example, here is a version of `factorial` with print statements:

In [11]:
```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

`space` is a string of space characters that controls the indentation of the output. Here is the result of `factorial(3)` :

In [12]:
```
factorial(3)
```

```
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
```

Out[12]: 6

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but a little bit of scaffolding can save a lot of debugging.

Glossary

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

pure function: A function that does not display anything or have any other effect, other than returning a return value.

dead code: Part of a program that can never run, often because it appears after a `return` statement.

incremental development: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

scaffolding: Code that is used during program development but is not part of the final version.

Turing complete: A language, or subset of a language, is Turing complete if it can perform any computation that can be described by an algorithm.

input validation: Checking the parameters of a function to make sure they have the correct types and values

6.12 Exercises

In [13]:
```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

```
Exception reporting mode: Verbose
```

6.12.1 Ask a virtual assistant

In this chapter, we saw an incorrect function that can end without returning a value.

In [14]:
```
def absolute_value_wrong(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

And a version of the same function that has dead code at the end.

In [15]:
```
def absolute_value_extra_return(x):
    if x < 0:
        return -x
    else:
        return x

    return 'This is dead code.'
```

And we saw the following example, which is correct but not idiomatic.

In [16]:
```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

Ask a virtual assistant what's wrong with each of these functions and see if it can spot the errors or improve the style.

In [17]:
```
def absolute_value_wrong(x):
    if x < 0:
        return -x
    elif x > 0:
        return x
    else:
        return 0
```

In [ ]:
```
def absolute_value_extra_return(x):
    if x < 0:
        return -x
    else:
        return x
```

In [ ]:
```
def is_divisible(x, y):
    return x % y == 0
```

Then ask "Write a function that takes coordinates of two points and computes the distance between them." See if the result resembles the version of `distance` we wrote in this chapter.

In [1]:
```
## Per Gemini, Sept 16, 2024

import math

def distance_between_points(point1, point2):
  """Computes the Euclidean distance between two points.

  Args:
    point1: A tuple or list representing the coordinates of the first point.
    point2: A tuple or list representing the coordinates of the second point
```

```
  Returns:
    The Euclidean distance between the two points.
  """

  x1, y1 = point1
  x2, y2 = point2

  distance = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
  return distance

# Example usage:
point_a = (3, 4)
point_b = (0, 0)
distance = distance_between_points(point_a, point_b)
print(distance)  # Output: 5.0
```

```
5.0
```

6.12.2 Exercise

Use incremental development to write a function called `hypot` that returns the length of the hypotenuse of a right triangle given the lengths of the other two legs as arguments.

Note: There's a function in the math module called `hypot` that does the same thing, but you should not use it for this exercise!

Even if you can write the function correctly on the first try, start with a function that always returns `0` and practice making small changes, testing as you go. When you are done, the function should only return a value -- it should not display anything.

In [37]:
```
# Solution
import math

def hypot(x, y):
    return 0
```

In [38]:
```
print(hypot(3, 4))
```

```
0
```

In [40]:
```
# Solution

def hypot(x, y):
    return x

print(hypot(3, 4))
```

```
3
```

In [41]:
```
# Solution

def hypot(x, y):
    return x+y
```

```
print(hypot(3, 4))
```

```
7
```

In [42]:
```
# Solution

def hypot(x, y):
    return x**2

print(hypot(3, 4))
```

```
9
```

In [43]:
```
# Solution

def hypot(x, y):
    return x**2 + y**2

print(hypot(3, 4))
```

```
25
```

In [44]:
```
# Solution

def hypot(x, y):
    return math.sqrt(x)

print(hypot(3, 4))
```

```
1.7320508075688772
```

In [45]:
```
# Solution

def hypot(x, y):
    return math.sqrt(x+y)

print(hypot(3, 4))
```

```
2.6457513110645907
```

In [46]:
```
# Solution

def hypot(x, y):
    return math.sqrt(x**2)

print(hypot(3, 4))
```

```
3.0
```

In [47]:
```
# Solution

def hypot(x, y):
    return math.sqrt(x**2+y**2)

print(hypot(3, 4))
```

```
5.0
```

6.12.3 Exercise

Write a boolean function, `is_between(x, y, z)`, that returns `True` if $x < y < z$ or if $z < y < x$, and `False` otherwise.

In [57]:
```
# Solution

def is_between(x, y, z):
    if x < y or y < z:
        return True
    elif x > y or y > z:
        return True
    else:
        return False
```

You can use these examples to test your function.

In [58]:
```
is_between(1, 2, 3)  # should be True
```

Out[58]: True

In [59]:
```
# Solution

def is_between(x, y, z):
    if x < y < z:
        return True
    elif x > y > z:
        return True
    else:
        return False
```

In [60]:
```
is_between(3, 2, 1)  # should be True
```

Out[60]: True

In [61]:
```
# Solution

def is_between(x, y, z):
    return x < y < z or x > y > z
```

In [62]:
```
is_between(1, 3, 2)  # should be False
```

Out[62]: False

In [63]:
```
is_between(2, 3, 1)  # should be False
```

Out[63]: False

In [64]:
```
is_between(3, 2, 1)  # should be True
```

Out[64]: True

6.12.4 Exercise

The Ackermann function, $A(m, n)$, is defined:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Write a function named `ackermann` that evaluates the Ackermann function. What happens if you call `ackermann(5, 5)` ?

Was having to run stuff in VSCode there for a while, and it turns out the KaTex markdown stuff doenst work quite the same - the rendering below uses /text{} instead of /mbox{} as above! :-P

6.12.4 Exercise

The Ackermann function, $A(m, n)$, is defined:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

In [2]:
```python
# Solution

import math

def ackermann(m, n):
    """Computes the Ackermann function using the numbers m & n.

    Args:
    m: first number
    n: second number

    Returns:
    Ackerman result
    """
    ## THIS IS YUCK print(f"Parameter m: {m}, Parameter n: {n}")

    if m < 0 or n < 0:
        return "Ackermann function only defined for positive integers."
    elif type(m) != int or type(n) != int:
        return "Ackermann function only defined for positive integers!"
    elif m == 0:
        return n+1
    elif m >0 and n == 0:
        return ackermann(m-1,1)
    elif m > 0 and n > 0:
        return ackermann(m-1, ackermann(m, n-1))
```

```
    else:
        return "All is lost..."
```

You can use these examples to test your function.

In [3]:
```
ackermann(3, 2)  # should be 29
```

Out[3]: 29

In [4]:
```
ackermann(3, 3)  # should be 61
```

Out[4]: 61

In [5]:
```
ackermann(3, 4)  # should be 125
```

Out[5]: 125

If you call this function with values bigger than 4, you get a `RecursionError`.

In [6]:
```
#%%expect RecursionError

ackermann(5, 5)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[6], line 3
      1 #%%expect RecursionError
----> 3 ackermann(5, 5)

Cell In[2], line 26, in ackermann(m, n)
     24     return ackermann(m-1,1)
     25 elif m > 0 and n > 0:
---> 26     return ackermann(m-1, ackermann(m, n-1))
     27 else:
     28     return "All is lost..."

Cell In[2], line 26, in ackermann(m, n)
     24     return ackermann(m-1,1)
     25 elif m > 0 and n > 0:
---> 26     return ackermann(m-1, ackermann(m, n-1))
     27 else:
     28     return "All is lost..."

    [... skipping similar frames: ackermann at line 26 (2 times)]

Cell In[2], line 26, in ackermann(m, n)
     24     return ackermann(m-1,1)
     25 elif m > 0 and n > 0:
---> 26     return ackermann(m-1, ackermann(m, n-1))
     27 else:
     28     return "All is lost..."

Cell In[2], line 24, in ackermann(m, n)
     22     return n+1
     23 elif m >0 and n == 0:
---> 24     return ackermann(m-1,1)
     25 elif m > 0 and n > 0:
     26     return ackermann(m-1, ackermann(m, n-1))

Cell In[2], line 26, in ackermann(m, n)
     24     return ackermann(m-1,1)
     25 elif m > 0 and n > 0:
---> 26     return ackermann(m-1, ackermann(m, n-1))
     27 else:
     28     return "All is lost..."

Cell In[2], line 26, in ackermann(m, n)
     24     return ackermann(m-1,1)
     25 elif m > 0 and n > 0:
---> 26     return ackermann(m-1, ackermann(m, n-1))
     27 else:
     28     return "All is lost..."

    [... skipping similar frames: ackermann at line 26 (2967 times)]

Cell In[2], line 26, in ackermann(m, n)
     24     return ackermann(m-1,1)
     25 elif m > 0 and n > 0:
---> 26     return ackermann(m-1, ackermann(m, n-1))
```

```
     27 else:
     28     return "All is lost..."

Cell In[2], line 24, in ackermann(m, n)
     22     return n+1
     23 elif m >0 and n == 0:
---> 24     return ackermann(m-1,1)
     25 elif m > 0 and n > 0:
     26     return ackermann(m-1, ackermann(m, n-1))

RecursionError: maximum recursion depth exceeded
```

To see why, add a print statement to the beginning of the function to display the values of the parameters, and then run the examples again.

Actually, adding a print is a bad idea here, it slows it all the way down and who knows how long it will take to reach a recursive limit...

6.12.4.5 Exercise

A number, a, is a power of b if it is divisible by b and a/b is a power of b. Write a function called `is_power` that takes parameters `a` and `b` and returns `True` if `a` is a power of `b`. Note: you will have to think about the base case.

In [24]:
```
# Solution - so simple - had to have some Copilot help here - been a minute

import math

def is_power(a, b):
    """Determines if a is a power of b

    Args:
    a: number/result
    b: base

    Returns:
    True if a is a power of b
    """

    if a == b:
        return True

    if a%b == 0:
        return is_power(a//b,b)

    else:
        return False
```

You can use these examples to test your function.

In [15]:
```
is_power(65536, 2)   # should be True
```

Out[15]: True

In [16]:
```
is_power(27, 3)  # should be True
```

Out[16]: True

In [17]:
```
is_power(24, 2)  # should be False
```

Out[17]: False

In [18]:
```
is_power(1, 17)   # should be True
```

Out[18]: False

6.12.5 Exercise

The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.

One way to find the GCD of two numbers is based on the observation that if r is the remainder when a is divided by b, then $gcd(a, b) = gcd(b, r)$. As a base case, we can use $gcd(a, 0) = a$.

Write a function called `gcd` that takes parameters `a` and `b` and returns their greatest common divisor.

In [21]:
```
# Solution

def gcd(a,b):
    if b == 0:
        return a
    r = a%b
    return gcd(b,r)
```

You can use these examples to test your function.

In [22]:
```
gcd(12, 8)    # should be 4
```

Out[22]: 4

In [23]:
```
gcd(13, 17)   # should be 1
```

Out[23]: 1

Base case was a bit tricky! --> gcd(a,0) translated to the condition if b==0 !