

You can order print and ebook versions of *Think Python 3e* from [Bookshop.org](https://bookshop.org) and [Amazon](https://www.amazon.com).

```
In [163]: 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');
download('https://github.com/ramalho/jupyter/turtle/versions/latest/download/2024-03/jupyterlab-4.0.0.tar.gz')

import thinkpython
```

The history saving thread hit an unexpected error (OperationalError('attempt to write a readonly database')). History will not be written to the database.

Conditionals and Recursion

The main topic of this chapter is the `if` statement, which executes different code depending on the state of the program. And with the `if` statement we'll be able to explore one of the most powerful ideas in computing, **recursion**.

But we'll start with three new features: the modulus operator, boolean expressions, and logical operators.

Integer division and modulus

Recall that the integer division operator, `//`, divides two numbers and rounds down to an integer. For example, suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

```
In [2]: minutes = 105
minutes / 60
```

Out[2]: 1.75

But we don't normally write hours with decimal points. Integer division returns the integer number of hours, rounding down:

```
In [3]: minutes = 105
```

```
hours = minutes // 60
hours
```

Out[3]: 1

To get the remainder, you could subtract off one hour in minutes:

```
In [4]: remainder = minutes - hours * 60
remainder
```

Out[4]: 45

Or you could use the **modulus operator**, `%`, which divides two numbers and returns the remainder.

```
In [5]: remainder = minutes % 60
remainder
```

Out[5]: 45

The modulus operator is more useful than it might seem. For example, it can check whether one number is divisible by another -- if `x % y` is zero, then `x` is divisible by `y`.

Also, it can extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly, `x % 100` yields the last two digits.

```
In [6]: x = 123
x % 10
```

Out[6]: 3

```
In [7]: x % 100
```

Out[7]: 23

Finally, the modulus operator can do "clock arithmetic". For example, if an event starts at 11 AM and lasts three hours, we can use the modulus operator to figure out what time it ends.

```
In [8]: start = 11
duration = 3
end = (start + duration) % 12
end
```

Out[8]: 2

The event would end at 2 PM.

Boolean Expressions

A **boolean expression** is an expression that is either true or false. For example, the following expressions use the equals operator, `==`, which compares two values and produces `True` if they are equal and `False` otherwise:

```
In [9]: 5 == 5
```

```
Out[9]: True
```

```
In [10]: 5 == 7
```

```
Out[10]: False
```

A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` assigns a value to a variable and `==` compares two values.

```
In [11]: x = 5  
y = 7
```

```
In [12]: x == y
```

```
Out[12]: False
```

`True` and `False` are special values that belong to the type `bool`; they are not strings:

```
In [13]: type(True)
```

```
Out[13]: bool
```

```
In [14]: type(False)
```

```
Out[14]: bool
```

The `==` operator is one of the **relational operators**; the others are:

```
In [15]: x != y           # x is not equal to y
```

```
Out[15]: True
```

```
In [16]: x > y           # x is greater than y
```

```
Out[16]: False
```

```
In [17]: x < y           # x is less than to y
```

```
Out[17]: True
```

```
In [18]: x >= y          # x is greater than or equal to y
```

```
Out[18]: False
```

```
In [19]: x <= y          # x is less than or equal to y
```

Out[19]: True

Logical operators

To combine boolean values into expressions, we can use **logical operators**. The most common are `and`, `or`, and `not`. The meaning of these operators is similar to their meaning in English. For example, the value of the following expression is `True` only if `x` is greater than 0 *and* less than 10.

```
In [20]: x > 0 and x < 10
```

Out[20]: True

The following expression is `True` if *either or both* of the conditions is true, that is, if the number is divisible by 2 *or* 3:

```
In [21]: x % 2 == 0 or x % 3 == 0
```

Out[21]: False

Finally, the `not` operator negates a boolean expression, so the following expression is `True` if `x > y` is `False`.

```
In [22]: not x > y
```

Out[22]: True

Strictly speaking, the operands of a logical operator should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as `True`:

```
In [23]: 42 and True
```

Out[23]: True

This flexibility can be useful, but there are some subtleties to it that can be confusing. You might want to avoid it.

if statements

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
In [24]: if x > 0:
          print('x is positive')
```

x is positive

`if` is a Python keyword. `if` statements have the same structure as function definitions: a header followed by an indented statement or sequence of statements called a **block**.

The boolean expression after `if` is called the **condition**. If it is true, the statements in the indented block run. If not, they don't.

There is no limit to the number of statements that can appear in the block, but there has to be at least one. Occasionally, it is useful to have a block that does nothing -- usually as a place keeper for code you haven't written yet. In that case, you can use the `pass` statement, which does nothing.

```
In [25]: if x < 0:
          pass          # TODO: need to handle negative values!
```

The word `TODO` in a comment is a conventional reminder that there's something you need to do later.

The `else` clause

An `if` statement can have a second part, called an `else` clause. The syntax looks like this:

```
In [26]: if x % 2 == 0:
          print('x is even')
          else:
          print('x is odd')
```

`x is odd`

If the condition is true, the first indented statement runs; otherwise, the second indented statement runs.

In this example, if `x` is even, the remainder when `x` is divided by `2` is `0`, so the condition is true and the program displays `x is even`. If `x` is odd, the remainder is `1`, so the condition is false, and the program displays `x is odd`.

Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called **branches**.

Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**, which includes an `elif` clause.

```
In [27]: if x < y:
```

```
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

x is less than y

`elif` is an abbreviation of "else if". There is no limit on the number of `elif` clauses. If there is an `else` clause, it has to be at the end, but there doesn't have to be one.

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the `if` statement ends. Even if more than one condition is true, only the first true branch runs.

Nested Conditionals

One conditional can also be nested within another. We could have written the example in the previous section like this:

```
In [28]: if x == y:
        print('x and y are equal')
        else:
            if x < y:
                print('x is less than y')
            else:
                print('x is greater than y')
```

x is less than y

The outer `if` statement contains two branches. The first branch contains a simple statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, **nested conditionals** can be difficult to read. I suggest you avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. Here's an example with a nested conditional.

```
In [29]: if 0 < x:
        if x < 10:
            print('x is a positive single-digit number.')
```

x is a positive single-digit number.

The `print` statement runs only if we make it past both conditionals, so we get the same effect with the `and` operator.

```
In [30]: if 0 < x and x < 10:
```

```
print('x is a positive single-digit number.')
```

x is a positive single-digit number.

For this kind of condition, Python provides a more concise option:

```
In [31]: if 0 < x < 10:  
        print('x is a positive single-digit number.')
```

x is a positive single-digit number.

Recursion

It is legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. Here's an example.

```
In [32]: def countdown(n):  
        if n <= 0:  
            print('Blastoff!')  
        else:  
            print(n)  
            countdown(n-1)
```

If `n` is 0 or negative, `countdown` outputs the word, "Blastoff!" Otherwise, it outputs `n` and then calls itself, passing `n-1` as an argument.

Here's what happens when we call this function with the argument `3`.

```
In [33]: countdown(3)
```

```
3  
2  
1  
Blastoff!
```

The execution of `countdown` begins with `n=3`, and since `n` is greater than `0`, it displays `3`, and then calls itself...

The execution of `countdown` begins with `n=2`, and since `n` is greater than `0`, it displays `2`, and then calls itself...

The execution of `countdown` begins with `n=1`, and since `n` is greater than `0`, it displays `1`, and then calls itself...

The execution of `countdown` begins with `n=0`, and since `n` is not greater than `0`, it displays "Blastoff!" and returns.

The `countdown` that got `n=1` returns.

The `countdown` that got `n=2` returns.

The countdown that got `n=3` returns.

A function that calls itself is **recursive**. As another example, we can write a function that prints a string `n` times.

```
In [34]: def print_n_times(string, n):  
        if n > 0:  
            print(string)  
            print_n_times(string, n-1)
```

If `n` is positive, `print_n_times` displays the value of `string` and then calls itself, passing along `string` and `n-1` as arguments.

If `n` is 0 or negative, the condition is false and `print_n_times` does nothing.

Here's how it works.

```
In [35]: print_n_times('Spam ', 4)
```

```
Spam  
Spam  
Spam  
Spam
```

For simple examples like this, it is probably easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

Stack diagrams for recursive functions

Here's a stack diagram that shows the frames created when we called `countdown` with `n = 3`.

```
In [36]: from diagram import make_frame, Stack  
  
frames = []  
for n in [3,2,1,0]:  
    d = dict(n=n)  
    frame = make_frame(d, name='countdown', dy=-0.3, loc='left')  
    frames.append(frame)  
  
stack = Stack(frames, dy=-0.5)
```

```
In [37]: from diagram import diagram, adjust  
  
width, height, x, y = [1.74, 2.04, 1.05, 1.77]  
ax = diagram(width, height)  
bbox = stack.draw(ax, x, y)  
# adjust(x, y, bbox)
```




The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where `n=0`, is called the **base case**. It does not make a recursive call, so there are no more frames.

```
In [38]: from diagram import make_frame, Stack
from diagram import diagram, adjust

frames = []
for n in [2,1,0]:
    d = dict(string='Hello', n=n)
    frame = make_frame(d, name='print_n_times', dx=1.3, loc='left')
    frames.append(frame)

stack = Stack(frames, dy=-0.5)

width, height, x, y = [3.53, 1.54, 1.54, 1.27]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```



Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea. Here's a minimal function with an infinite recursion.

```
In [39]: def recurse():
```

```
recurse()
```

Every time `recurse` is called, it calls itself, which creates another frame. In Python, there is a limit to the number of frames that can be on the stack at the same time. If a program exceeds the limit, it causes a runtime error.

```
In [40]: %xmode Context
```

Exception reporting mode: Context

```
In [41]: %%expect RecursionError
```

```
recurse()
```

RecursionError Traceback (most recent call last)

Cell In[41], line 1

----> 1 recurse()

Cell In[39], line 2, in recurse()

```
1 def recurse():  
----> 2     recurse()
```

Cell In[39], line 2, in recurse()

```
1 def recurse():  
----> 2     recurse()
```

[... skipping similar frames: recurse at line 2 (2957 times)]

Cell In[39], line 2, in recurse()

```
1 def recurse():  
----> 2     recurse()
```

RecursionError: maximum recursion depth exceeded

The traceback indicates that there were almost 3000 frames on the stack when the error occurred.

If you encounter an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call. And if there is a base case, check whether you are guaranteed to reach it.

Keyboard input

The programs we have written so far accept no input from the user. They just do the same thing every time.

Python provides a built-in function called `input` that stops the program and waits for the user to type something. When the user presses *Return* or *Enter*, the program resumes and `input` returns what the user typed as a string.

```
In [42]: # Solution goes here
# NOT SURE WHAT THIS BLOCK IS FOR...??
```

```
In [43]: text = input()
```

Before getting input from the user, you might want to display a prompt telling the user what to type. `input` can take a prompt as an argument:

```
In [44]: # Solution goes here
```

```
In [45]: name = input('What...is your name?\n')
name
```

```
Out[45]: 'General CHeese'
```

The sequence `\n` at the end of the prompt represents a **newline**, which is a special character that causes a line break -- that way the user's input appears below the prompt.

If you expect the user to type an integer, you can use the `int` function to convert the return value to `int`.

```
In [46]: # Solution goes here
```

```
In [47]: prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
speed = input(prompt)
speed
```

```
Out[47]: '2000000'
```

But if they type something that's not an integer, you'll get a runtime error.

```
In [48]: %xmode Minimal
```

Exception reporting mode: Minimal

```
In [49]: %%expect ValueError
int(speed)
```

```
Out[49]: 2000000
```

We will see how to handle this kind of error later.

Debugging

When a syntax or runtime error occurs, the error message contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was, and

- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Errors related to spaces and tabs can be tricky because they are invisible and we are used to ignoring them.

```
In [50]: %%expect IndentationError
x = 5
y = 6
```

```
Cell In[50], line 2
    y = 6
    ^
```

IndentationError: unexpected indent

In this example, the problem is that the second line is indented by one space. But the error message points to `y`, which is misleading. Error messages indicate where the problem was discovered, but the actual error might be earlier in the code.

The same is true of runtime errors. For example, suppose you are trying to convert a ratio to decibels, like this:

```
In [51]: %xmode Context
```

Exception reporting mode: Context

```
In [52]: %%expect ValueError
import math
numerator = 9
denominator = 10
ratio = numerator // denominator
decibels = 10 * math.log10(ratio)
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[52], line 5
      3 denominator = 10
      4 ratio = numerator // denominator
----> 5 decibels = 10 * math.log10(ratio)
```

ValueError: math domain error

The error message indicates line 5, but there is nothing wrong with that line. The problem is in line 4, which uses integer division instead of floating-point division -- as a result, the value of `ratio` is `0`. When we call `math.log10`, we get a `ValueError` with the message `math domain error`, because `0` is not in the "domain" of valid arguments for `math.log10`, because the logarithm of `0` is undefined.

In general, you should take the time to read error messages carefully, but don't assume that everything they say is correct.

Glossary

recursion: The process of calling the function that is currently executing.

modulus operator: An operator, `%`, that works on integers and returns the remainder when one number is divided by another.

boolean expression: An expression whose value is either `True` or `False`.

relational operator: One of the operators that compares its operands: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

logical operator: One of the operators that combines boolean expressions, including `and`, `or`, and `not`.

conditional statement: A statement that controls the flow of execution depending on some condition.

condition: The boolean expression in a conditional statement that determines which branch runs.

block: One or more statements indented to indicate they are part of another statement.

branch: One of the alternative sequences of statements in a conditional statement.

chained conditional: A conditional statement with a series of alternative branches.

nested conditional: A conditional statement that appears in one of the branches of another conditional statement.

recursive: A function that calls itself is recursive.

base case: A conditional branch in a recursive function that does not make a recursive call.

infinite recursion: A recursion that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

newline: A character that creates a line break between two parts of a string.

Exercises

```
In [53]: # 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

5.14.1 Ask a virtual assistant

- Ask a virtual assistant, "What are some uses of the modulus operator?"

- Python provides operators to compute the logical operations `and` , `or` , and `not` , but it doesn't have an operator that computes the exclusive `or` operation, usually written `xor` . Ask an assistant "What is the logical xor operation and how do I compute it in Python?"

In this chapter, we saw two ways to write an `if` statement with three branches, using a chained conditional or a nested conditional. You can use a virtual assistant to convert from one to the other. For example, ask a VA, "Convert this statement to a chained conditional."

In Python, you can compute the logical XOR operation using the bitwise XOR operator `^`. Here's an example:

```
In [57]: a = True
b = False
result = a ^ b
print(result) # Output: True
```

True

Continuing...

```
In [55]: x = 5
y = 7
```

```
In [56]: if x == y:
          print('x and y are equal')
        else:
          if x < y:
            print('x is less than y')
          else:
            print('x is greater than y')
```

x is less than y

```
In [59]: ##### Chained conditional by Gemini
```

```
In [58]: if x == y:
          print('x and y are equal')
        elif x < y:
          print('x is less than y')
        else:
          print('x is greater than y')
```

x is less than y

Ask a VA, "Rewrite this statement with a single conditional."

```
In [60]: if 0 < x:
          if x < 10:
            print('x is a positive single-digit number.')
```

x is a positive single-digit number.

This is not really a single conditional... it's a compound condition.. with two!

```
In [61]: if 0 < x < 10:  
        print('x is a positive single-digit number.')
```

x is a positive single-digit number.

See if a VA can simplify this unnecessary complexity.

```
In [64]: if not x <= 0 and not x >= 10:  
        print('x is a positive single-digit number.')
```

x is a positive single-digit number.

Simplifies to the same as above...

Here's an attempt at a recursive function that counts down by two.

```
In [65]: def countdown_by_two(n):  
        if n == 0:  
            print('Blastoff!')  
        else:  
            print(n)  
            countdown_by_two(n-2)
```

It seems to work.

```
In [66]: countdown_by_two(6)
```

```
6  
4  
2  
Blastoff!
```

But it has an error. Ask a virtual assistant what's wrong and how to fix it. Paste the solution it provides back here and test it.

Gemini:

The primary issue with the provided recursive function is the lack of a base case for negative numbers.

Here's a breakdown of the problem:

1. **Correctness for positive numbers:** The function works as expected for positive numbers, counting down by two until it reaches 0 and prints "Blastoff!".
2. **Incorrectness for negative numbers:** However, when `n` is negative, the function will continue to call itself recursively, subtracting 2 in each call. This creates an infinite loop, leading to a `RecursionError` or potentially crashing the program.

To fix this, we need to add a condition to check if `n` is negative before the recursive

call:

This modified version ensures that the recursion stops when `n` becomes negative, preventing an infinite loop and providing a more robust and correct implementation.

```
In [68]: def countdown_by_two(n):  
         if n >= 0: # Check if n is non-negative  
             if n == 0:  
                 print('Blastoff!')  
             else:  
                 print(n)  
                 countdown_by_two(n-2)  
         else:  
             print("Invalid input: n must be non-negative.")
```

```
In [69]: countdown_by_two(9)  
9  
7  
5  
3  
1  
Invalid input: n must be non-negative.
```

```
In [70]: countdown_by_two(10)  
10  
8  
6  
4  
2  
Blastoff!
```

5.14.2 Exercise

The `time` module provides a function, also called `time`, that returns the number of seconds since the "Unix epoch", which is January 1, 1970, 00:00:00 UTC (Coordinated Universal Time).

```
In [71]: from time import time  
  
now = time()  
now
```

```
Out[71]: 1726105751.7147927
```

Use integer division and the modulus operator to compute the number of days since January 1, 1970 and the current time of day in hours, minutes, and seconds.

You can read more about the `time` module at <https://docs.python.org/3/library/time.html>.


```
In [77]: # Solution
oneDayInSecs = 24*60*60
print(f"One day = {oneDayInSecs} seconds")
daysSinceEpoch = int(now//oneDayInSecs)
print(f"Number of days since Jan 1, 1970: {daysSinceEpoch}")
```

One day = 86400 seconds
Number of days since Jan 1, 1970: 19978

```
In [108... # Solution
currentTimeInSecs = now%oneDayInSecs
print(f"Current UTC time of day in seconds: {currentTimeInSecs}")
```

Current UTC time of day in seconds: 6551.714792728424

```
In [112... oneHourInSec = 60*60
currentUTC_hour = currentTimeInSecs // oneHourInSec
print(f"Current UTC hour: {int(currentUTC_hour)}")

currentUTC_min = (currentTimeInSecs - currentUTC_hour*oneHourInSec)//60
print(f"Current UTC min: {int(currentUTC_min)}")

currentUTC_sec = int(currentTimeInSecs - currentUTC_hour*oneHourInSec - curr
print(f"Current UTC secs: {currentUTC_sec}")
```

Current UTC hour: 1
Current UTC min: 49
Current UTC secs: 11

```
In [113... # Solution - TIMEZONE adjustment: EST: -5 hrs
tmznOffset = -5*60*60
adjustedNow = now + tmznOffset
localTimeInSecs = adjustedNow%oneDayInSecs

localHour = localTimeInSecs // oneHourInSec
print(f"Local hour: {int(localHour)}")

localMin = (localTimeInSecs - localHour*oneHourInSec)//60
print(f"Current UTC min: {int(localMin)}")

localSec = int(localTimeInSecs - localHour*oneHourInSec - localMin*60)
print(f"Local secs: {localSec}")
```

Local hour: 20
Current UTC min: 49
Local secs: 11

5.14.3 Exercise

If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a test to see if it is possible to form a triangle:

If any of the three lengths is greater than the sum of the other two, then

you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a "degenerate" triangle.)

Write a function named `is_triangle` that takes three integers as arguments, and that prints either "Yes" or "No", depending on whether you can or cannot form a triangle from sticks with the given lengths. Hint: Use a chained conditional.

In [114... *# Solution*

```
def is_triangle(a,b,c):
    """ Three integers as arguments
    Prints "Yes" or "No", depending on whether you can or cannot form a triangle
    Returns true or false respectively

    a: side 1
    b: side 2
    c: side 3
    """

    if a > (b + c):
        print("No")
        return(False)
    elif b > (a + c):
        print("No")
        return(False)
    elif c > (a + b):
        print("No")
        return(False)
    else:
        print("Yes")
        return(True)
```

Test your function with the following cases.

In [115... `is_triangle(4, 5, 6)` *# should be Yes*

Yes

Out[115... True

In [116... `is_triangle(1, 2, 3)` *# should be Yes*

Yes

Out[116... True

In [117... `is_triangle(6, 2, 3)` *# should be No*

No

Out[117... False

In [118... `is_triangle(1, 1, 12)` *# should be No*

No

Out[118... False

5.14.4 Exercise

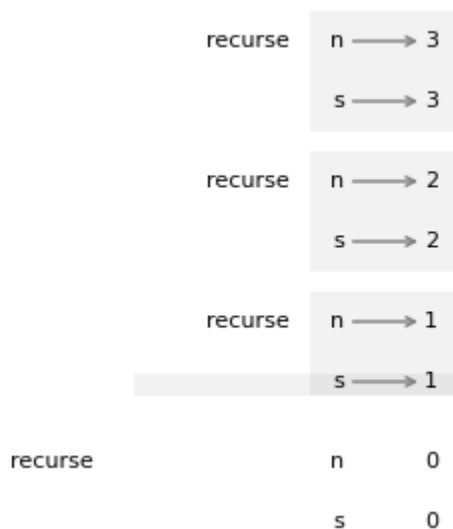
What is the output of the following program? Draw a stack diagram that shows the state of the program when it prints the result.

```
In [119... def recurse(n, s):  
    if n == 0:  
        print(s)  
    else:  
        recurse(n-1, n+s)  
  
recurse(3, 0)
```

6

-- it prints the final cumulative sum at end. In this case: $3+2+1 = 6$

```
In [160... frames = []  
for n in [3,2,1,0]:  
    d = dict(n=n,s=n)  
    frame = make_frame(d, name='recurse', dy=-0.3, loc='left')  
    frames.append(frame)  
  
stack = Stack(frames, dy=-0.7)  
  
width, height, x, y = [1.74, 2.04, 1.05, 1.77]  
ax = diagram(width, height)  
bbox = stack.draw(ax, x, y)  
# adjust(x, y, bbox)
```



In [162... *# Solution - kinda clunky - but, basically there: not worth the mucking arou*

5.14.5 Exercise

The following exercises use the `jupyterturtle` module, described in Chapter 4.

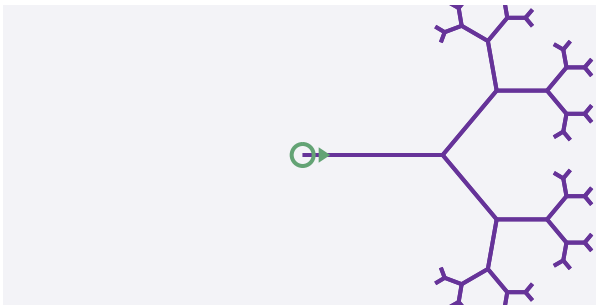
Read the following function and see if you can figure out what it does. Then run it and see if you got it right. Adjust the values of `length`, `angle` and `factor` and see what effect they have on the result. If you are not sure you understand how it works, try asking a virtual assistant.

```
In [3]: from jupyterturtle import forward, left, right, back, make_turtle

def draw(length):
    angle = 50
    factor = 0.6

    if length > 5:
        forward(length)
        left(angle)
        draw(factor * length)
        right(2 * angle)
        draw(factor * length)
        left(angle)
        back(length)
```

```
In [7]: # Solution
make_turtle(delay=0)
draw(70)
```



Cooooo... a little different than i anticipated... but cool.!

5.14.6 Exercise

Ask a virtual assistant "What is the Koch curve?"

To draw a Koch curve with length x , all you have to do is

1. Draw a Koch curve with length $x/3$.
2. Turn left 60 degrees.
3. Draw a Koch curve with length $x/3$.

4. Turn right 120 degrees.
5. Draw a Koch curve with length $x/3$.
6. Turn left 60 degrees.
7. Draw a Koch curve with length $x/3$.

The exception is if x is less than 5 -- in that case, you can just draw a straight line with length x .

Write a function called `koch` that takes x as an argument and draws a Koch curve with the given length.

```
In [8]: # Solution

def koch(x):
    """ Koch curve approx - Fractal!
    """
    if x < 5:
        forward(x)
        return
    koch(x/3)
    left(60)
    koch(x/3)
    right(120)
    koch(x/3)
    left(60)
    koch(x/3)
```

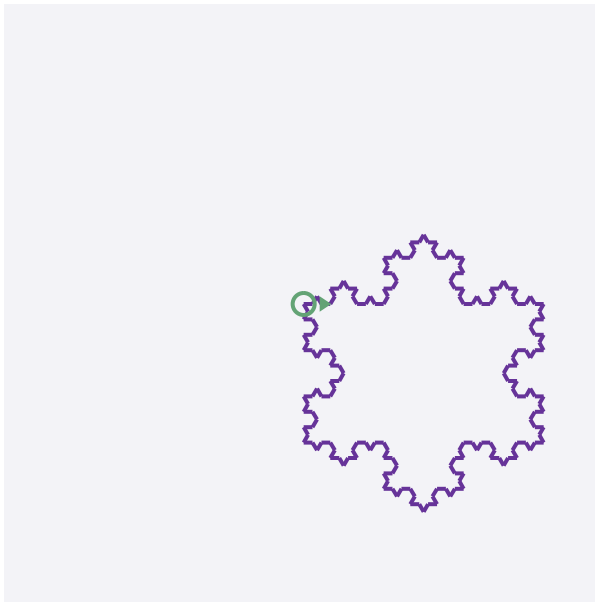
The result should look like this:

```
In [9]: make_turtle(delay=0)
        koch(120)
```



Once you have `koch` working, you can use this loop to draw three Koch curves in the shape of a snowflake.

```
In [10]: make_turtle(delay=0, height=300)
         for i in range(3):
             koch(120)
             right(120)
```



**** Har! Rad!**

5.14.7 Exercise

Virtual assistants know about the functions in the `jupyter_turtle` module, but there are many versions of these functions, with different names, so a VA might not know which one you are talking about.

To solve this problem, you can provide additional information before you ask a question. For example, you could start a prompt with "Here's a program that uses the `jupyter_turtle` module," and then paste in one of the examples from this chapter. After that, the VA should be able to generate code that uses this module.

As an example, ask a VA for a program that draws a Sierpiński triangle. The code you get should be a good starting place, but you might have to do some debugging. If the first attempt doesn't work, you can tell the VA what happened and ask for help -- or you can debug it yourself.

```
In [ ]: # Solution FROM GEMINI (Sept 13, 2024)

import turtle

def sierpinski_triangle(t, size, depth):
    """Draws a Sierpiński triangle.

    Args:
        t: The Turtle object.
        size: The side length of the triangle.
        depth: The depth of recursion.
    """

    if depth == 0:
```

```

    t.begin_fill()
    for _ in range(3):
        t.forward(size)
        t.left(120)
    t.end_fill()
else:
    for _ in range(3):
        sierpinski_triangle(t, size / 2, depth - 1)
        t.forward(size / 2)
        t.left(120)

# Set up the turtle
turtle.speed(0) # Set the turtle speed to the fastest
turtle.hideturtle() # Hide the turtle cursor

# Draw the Sierpiński triangle
sierpinski_triangle(turtle, 300, 6) # Adjust size and depth as needed

turtle.done() # Keep the window open until closed manually

```

In [6]: # 'Debugged' Solution

```

from jupyter_turtle import forward, left, right, back, make_turtle

def draw_sierpinski(size, depth):
    """Draws a Sierpiński triangle.

    Args:
        size: The side length of the triangle.
        depth: The depth of recursion.
    """

    if depth == 0:
        for _ in range(3):
            forward(size)
            left(120)
    else:
        for _ in range(3):
            sierpinski_triangle(size / 2, depth - 1)
            forward(size / 2)
            left(120)

```

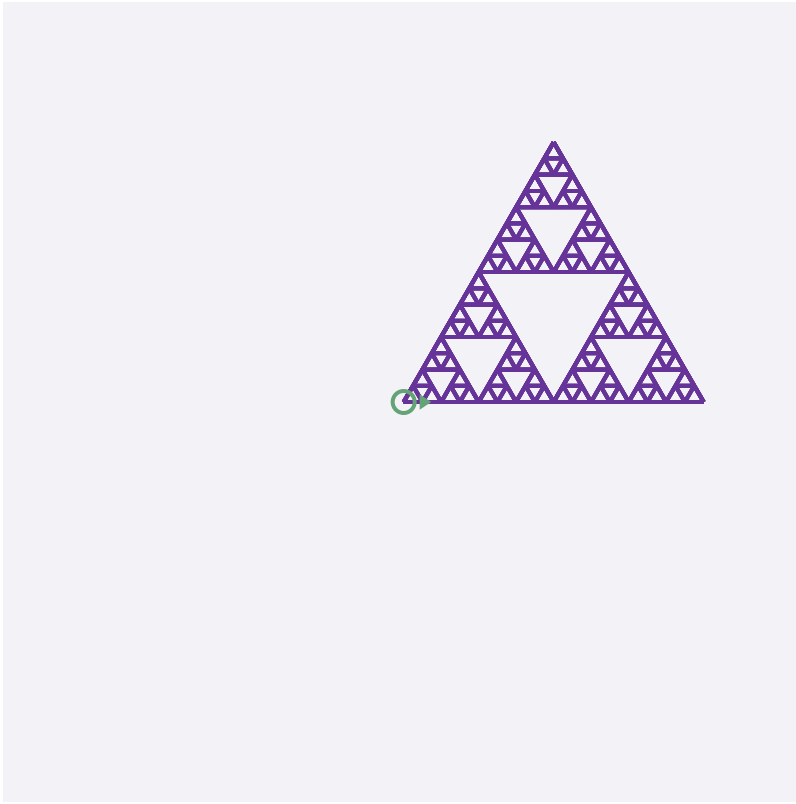
Here's what the result might look like, although the version you get might be different.

```

In [11]: make_turtle(delay=0, height=400, width=400)

draw_sierpinski(300, 5)

```



**** NEAT! :-)**