

# Stack

This lesson will introduce you to the stack data structure and its implementation in Python which we'll use in the problems throughout this chapter.

## We'll cover the following



- What is a stack?
- Stack Operations
  - Push
  - Pop
  - Peek

In this lesson, we are going to consider the stack data structure and its implementation in Python.

In subsequent lessons, we'll provide specific problems where a stack is particularly useful. We'll be using the implementation that we develop in this lesson to solve those problems.

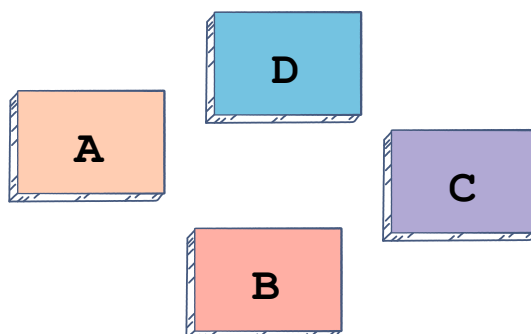
## What is a stack? #

First of all, let me describe what a stack is. Based on the name, it should be a relatively familiar concept.

Let's assume that we have four books with the following riveting titles:

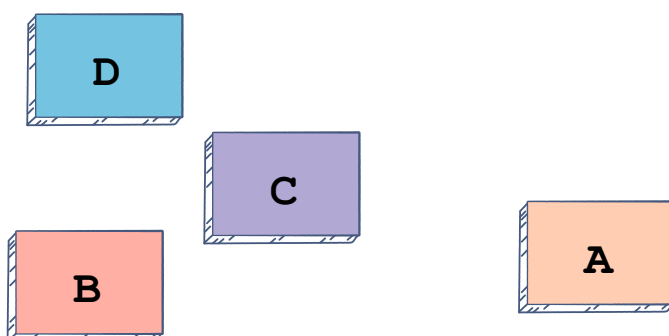
- A
- B
- C
- D

At the moment, these books are strewn out all over the floor and we want to stack them up neatly.



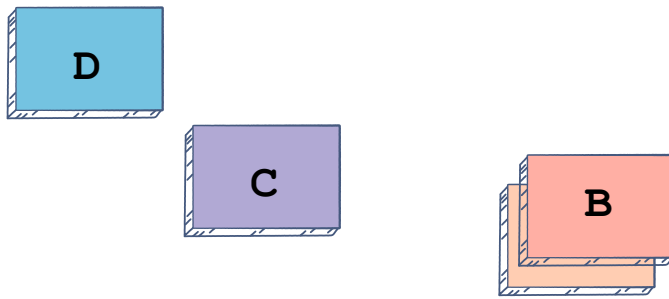
Four books are lying all over the place.

1 of 5



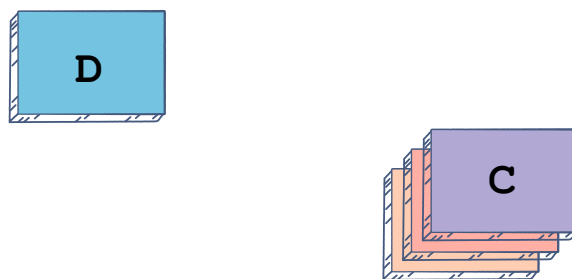
We take Book A and set it down.

2 of 5



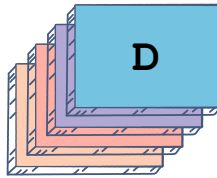
We go back to our pile of books, take Book B and put it on the top of the stack.

3 of 5



We take Book C, put it on B and we're almost done cleaning up!

4 of 5



Now we take Book D and place it on top of Book C.

5 of 5



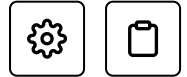
Now we have a nice neat stack of books! If we want to retrieve a book from this stack, we can take the book on top. Taking a book from the bottom is a bit precarious and we don't want to topple the entire stack. Therefore, we'll take down the top book on the stack and read it or do whatever we want to do with it.

Let's say we want to take **Book A**. Right now, it is at the bottom of the stack, so we need to take **Book D**, put it down, then do the same for **Book C** and **Book B**, and then we can access **Book A**.

This is the main idea of a stack. The data structure stack is very similar to a physical stack that you'd most likely be familiar with. The stack data structure allows us to place any programming artifact, variable or object on it, just as our example stack allowed us to put books in it.

## Stack Operations #

## Push #



The operation to insert elements in a stack is called **push**. When we *push* the book on a stack, we put the book on the previous *top* element which means that the new book becomes the *top* element. This is what we mean when we use the *push* operation, we *push* elements onto a stack. We insert elements onto a stack and the last element to be pushed is the new *top* of the stack.

## Pop #

There is another operation that we can perform on the stack, popping. Popping is when we take the top book of the stack and put it down. This implies that when we remove an element from the stack, the stack follows the *First-In, Last Out* property. This means that the top element, the last to be inserted, is removed when we perform the pop operation.

Push and Pop are two fundamental routines that we'll need for this data structure.

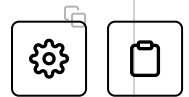
## Peek #

Another thing that we can do is view the top element of the stack so we can ask the data structure: "What's the top element?" and it can give that to us using the *peek* operation. Note that the *peek* operation does not remove the *top* element, it merely returns it.

We can also check whether or not the stack is empty, and a few other things too, that will come along the way as we implement it.

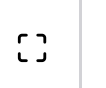
Now I'm going to create a stack class, and the constructor of the class is going to initialize a Python list. The Python list has a lot of things that we're after, and it's going to make it easier for us to tweak Python's implementation of the list to be adapted to what we would expect to see in a stack, namely, push/pop.

```
1 """
2 Stack Data Structure.
3 """
4 class Stack():
5     def __init__(self):
6         self.items = []
```



I'm defining a class variable that I'm calling `items`, and I'm assigning it to an empty list. `self.items` is created when we create a stack object so now let's create the `push` method:

```
1 """
2 Stack Data Structure.
3 """
4 class Stack():
5     def __init__(self):
6         self.items = []
7
8     def push(self, item):
9         self.items.append(item)
```



`push` is going to be a member of this class, and it's going to take an `item` as an argument. In our example, that item is the book. We are putting or pushing the name of the book onto the top of the stack.

`append` is a built-in method for a Python list that adds the `item` to the end of the list. This is just what we want to do for our stack in the `push` method.

```
1 """
2 Stack Data Structure.
3 """
4 class Stack():
5     def __init__(self):
6         self.items = []
7
8     def push(self, item):
```



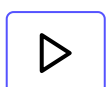
```
9         self.items.append(item)
10
11     def pop(self):
12         return self.items.pop()
```



The other method we need to implement is `pop`. Since we're basing this off a list, Python makes this very easy as well. There is an implicit `pop` method that returns the last element inserted to the list.

Another thing that we're going to do is add a few more methods, but before that, I'm going to do a test drive.

```
1  """
2  Stack Data Structure.
3  """
4  class Stack():
5      def __init__(self):
6          self.items = []
7
8      def push(self, item):
9          self.items.append(item)
10
11     def pop(self):
12         return self.items.pop()
13
14     def get_stack(self):
15         return self.items
16
17 myStack = Stack()
18 myStack.push("A")
19 myStack.push("B")
20 print(myStack.get_stack())
21 myStack.push("C")
22 print(myStack.get_stack())
23 myStack.pop()
24 print(myStack.get_stack())
```



We make a method called `get_stack()`. This will return the `items` list, which forms the core of the stack. Then I define a stack object, `myStack`, on **line 17** and push `A` and `B` onto the stack, **lines 18-19**. After these operations, I can print `myStack` and the output is `['A', 'B']`. `A` is at the bottom of the stack, and `B` is on the top. Then we push `C` on **line 21** and the output is `['A', 'B', 'C']`, which implies `C` is the top.

We understand the core fundamentals of the stack. A few of the other methods we can include in this are helpful, namely, we could have a method called `is_empty`. It will return whether or not the stack is empty.

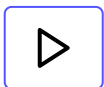
```
1  """
2  Stack Data Structure.
3  """
4  class Stack():
5      def __init__(self):
6          self.items = []
7
8      def push(self, item):
9          self.items.append(item)
10
11     def pop(self):
12         return self.items.pop()
13
14     def is_empty(self):
15         return self.items == []
16
17     def get_stack(self):
18         return self.items
19
20 myStack = Stack()
21 myStack.push("A")
22 print(myStack.is_empty())
```



We get `True` here because there's nothing on the stack at this point. If we push something and call `is_empty()`, we should get `False`. You can try it out yourself.



```
1  """
2  Stack Data Structure.
3  """
4  class Stack():
5      def __init__(self):
6          self.items = []
7
8      def push(self, item):
9          self.items.append(item)
10
11     def pop(self):
12         return self.items.pop()
13
14     def is_empty(self):
15         return self.items == []
16
17     def peek(self):
18         if not self.is_empty():
19             return self.items[-1]
20
21     def get_stack(self):
22         return self.items
23
24 myStack = Stack()
25 print(myStack.peek())
26 myStack.push("A")
27 myStack.push("B")
28 myStack.push("C")
```



Now we have the peek operation on **line 17** which tells us the topmost element of the stack.

If peek is called on the stack in the above code, it should return D . peek checks if the stack is not empty on **line 18** and if it isn't, it returns the last element of the list, which in this case, is the top element of the stack. We return `self.items[-1]` on **line 19** which is the last item in the list for Python. Hence, at first, we get D as the element that Python thinks is on the top of the stack.

Note that you can also check whether the stack is empty or not in the methods implemented previously. However, to keep things simple, this step has been omitted from the other methods.



Go ahead and create a stack object. You don't have to limit yourself to letters or strings or anything like that, you could also push numbers. Just play around with this data structure and get a sense of how it works. I'll see you in the next lesson.

[← Back](#)[Next →](#)

What is this course about?

Determine if Brackets are Balanced

 Completed



Report an  
Issue



Ask a Question

([https://discuss.educative.io/tag/stack\\_\\_stack\\_\\_data-structures-and-algorithms-in-python](https://discuss.educative.io/tag/stack__stack__data-structures-and-algorithms-in-python))



# Determine if Brackets are Balanced

This lesson will teach us how to determine whether or not a string has balanced usage of brackets by using a stack.

## We'll cover the following



- Examples of Balanced Brackets
- Examples of Unbalanced Brackets
- Algorithm
- Special Case
- Explanation
- Explanation

In this lesson, we're going to determine whether or not a set of brackets are balanced or not by making use of the stack data structure that we defined in the previous lesson.

Let's first understand what a balanced set of brackets looks like.

A balanced set of brackets is one where the number and type of opening and closing brackets match and that is also properly nested within the string of brackets.

## Examples of Balanced Brackets #

- {}
- {} {}
- (({[]}))

## Examples of Unbalanced Brackets #

- `(( ))`
- `{{{}}} ]`
- `[] [] [] ]`



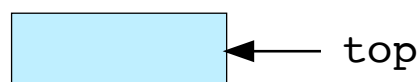
## Algorithm #

Check out the slides below to have a look at the approach we'll use to solve this problem:

### Balanced Example

`( [ ] )`

We have a balanced example of brackets and an empty stack.



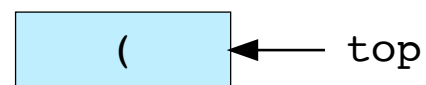
1 of 6



## Balanced Example

( [ ] )

We iterate through the string and push the bracket onto the stack if it's an opening bracket.

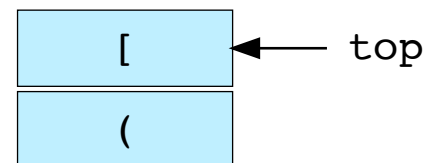


2 of 6

## Balanced Example

( [ ] )

We iterate through the string and push the bracket onto the stack if it's an opening bracket.



3 of 6



## Balanced Example

( [ ] )

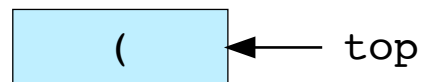
Now we encounter a closing square bracket so we pop off from the stack to check for a match. It is a match so we move on.

The top of the stack should be an opening square bracket to match the closing square bracket.

'[' (popped off)

matches

']'



4 of 6

## Balanced Example

( [ ] )

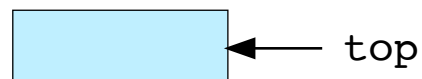
Now we encounter another closing bracket and we check for a match from the popped off element from stack.

The top of the stack should be an opening parenthesis to match the closing parenthesis.

'(' (popped off)

matches

')'



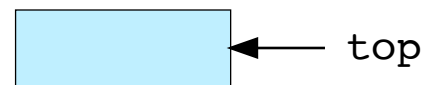
5 of 6



## Balanced Example

( [ ] )

As we have iterated over all the characters of the string and the stack is empty, we conclude that the string has a **balanced usage of brackets**.



6 of 6

— [ ]

As shown above, our algorithm is as follows:

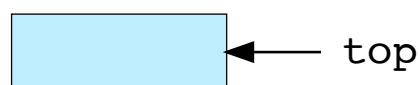
- We iterate through the characters of the string.
- If we get an opening bracket, push it onto the stack.
- If we encounter a closing bracket, pop off an element from the stack and match it with the closing bracket. If it is an opening bracket and of the same type as the closing bracket, we conclude it is a successful match and move on. If it's not, we will conclude that the set of brackets is not balanced.
- The stack will be empty at the end of iteration for a balanced example of brackets while we'll be left with some elements in the stack for an unbalanced example.



## Unbalanced Example

( ( )

We have an unbalanced example of brackets and an empty stack.



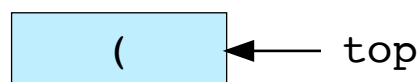
1 of 5

## Unbalanced Example

( ( )



We iterate through the string and push the bracket onto the stack if it's an opening bracket.



2 of 5

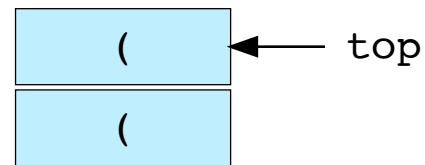




## Unbalanced Example

( ( )  
↑

We iterate through the string and push the bracket onto the stack if it's an opening bracket.



3 of 5

## Unbalanced Example

( ( )  
↑

Now we encounter a closing bracket so we pop off from the stack to check for a match. It is a match so we move on.

The top of the stack should be an opening parenthesis to match the closing parenthesis.

'(' (popped off)

matches

')



4 of 5



## Unbalanced Example

( ( )

We have iterated over the entire string but we still have a non-empty stack. Therefore, we conclude that **brackets are not balanced** in this string.

( ← top

5 of 5

— [ ]

We've covered an example for both the balanced set of brackets and an unbalanced set of brackets, let's move on to a special case.

## Special Case #

**Example: ) )**

What if we encounter a closing bracket but don't have any elements to pop off from the stack? For example, in the case described above, we don't have an opening parenthesis, but we encounter a closing parenthesis. In this case, we immediately know that the string does not have a balanced usage of brackets. Therefore, we need to watch out for an empty stack in our implementation.

Now that you have got a decent idea of the algorithm, we'll go over the implementation of it in Python.

Let's start with the `is_paren_balanced` function:

```
1 def is_paren_balanced(paren_string):
2     s = Stack()
3     is_balanced = True
4     index = 0
5
6     while index < len(paren_string) and is_balanced:
7         paren = paren_string[index]
8         if paren in "({[":
9             s.push(paren)
10        else:
11            if s.is_empty():
12                is_balanced = False
13            else:
14                top = s.pop()
15                if not is_match(top, paren):
16                    is_balanced = False
17            index += 1
18
19    if s.is_empty() and is_balanced:
20        return True
21    else:
22        return False
```



is\_paren\_balanced(paren\_string)

## Explanation #

On **lines 2-4**, we declare a stack, `s` and two variables `is_balanced` and `index`, which are set to `True` and `0`, respectively.

The while loop on **line 6** will execute if the `index` is less than the length of `paren_string` **and** `is_balanced` is equal to `True`. If any of the conditions evaluate to `False`, our program will exit the while loop. In the while loop, we iterate over each character of the `paren_string` by indexing using the `index` variable and save the indexed element in `paren` variable.

We check on **line 8** whether `paren` is any type of the opening brackets and if it is, we push it onto the stack. If it's not any type of the opening brackets, we check if stack `s` is empty and set `is_balanced` to `False`.

This handles our special case, which we discussed in the previous section.

If the stack is not empty, we pop off the top element and check if the current paren, i.e., a closing bracket matches the type of the top element which is supposed to be an opening bracket. If the types don't match, then we update `is_balanced` to `False`.



We increment the `index` for the next iteration. The `while` loop keeps executing until the `index` is equal to or greater than the length of `paren_string` or `is_balanced` equals `False`.

After we exit the `while` loop, on **line 19**, we check if the stack is empty and `is_balanced` is `True`, then we return `True`. Otherwise, we return `False`.

The code given above is a simple implementation of the algorithm you were introduced to.

Let's implement the `is_match` function now:

```
1 def is_match(p1, p2):
2     if p1 == "(" and p2 == ")":
3         return True
4     elif p1 == "{" and p2 == "}":
5         return True
6     elif p1 == "[" and p2 == "]":
7         return True
8     else:
9         return False
```



`is_match(p1, p2)`

## Explanation #

The `is_match` function takes in two characters as `p1` and `p2` and evaluates whether they are a valid pair of brackets. For `p1` and `p2` to match, `p1` has to be an opening bracket while `p2` has to be the corresponding closing bracket. If `p1` and `p2` don't fall in any of the valid conditions, we return `False`.

Easy peasy, right?

The entire implementation of the solution to determine whether a string has a balanced usage of brackets is given below. Feel free to play around with it!



main.py

stack.py

```
17     index = 0
18
19     while index < len(paren_string) and is_balanced:
20         paren = paren_string[index]
21         if paren in "([{":
22             s.push(paren)
23         else:
24             if s.is_empty():
25                 is_balanced = False
26             else:
27                 top = s.pop()
28                 if not is_match(top, paren):
29                     is_balanced = False
30             index += 1
31
32     if s.is_empty() and is_balanced:
33         return True
34     else:
35         return False
36
37     print("String : (((({}))) Balanced or not?")
38     print(is_paren_balanced("(((({})))"))
39
40     print("String : []]] Balanced or not?")
41     print(is_paren_balanced("[]]]"))
42
43     print("String : [][] Balanced or not?")
44     print(is_paren_balanced("[][]"))
```

In the next lesson, we will go over another problem, i.e. reversing a string and solving it using a stack. See you there!

[← Back](#)

Stack

[Next →](#)  
Reverse String☒ Completed[Report an Issue](#)[Ask a Question](#)[https://discuss.educative.io/tag/determine-if-brackets-are-balanced\\_\\_stack\\_\\_data-structures-and-algorithms-in-python](https://discuss.educative.io/tag/determine-if-brackets-are-balanced__stack__data-structures-and-algorithms-in-python)



# Reverse String

This lesson teaches us how to reverse a string using a stack in Python.

We'll cover the following ^

- Algorithm
- Implementation
- Explanation

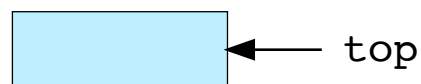
In Python, you can reverse a string very easily. For example,

```
1 input_str = "Educative"  
2 print(input_str[::-1])
```



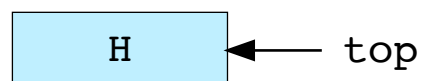
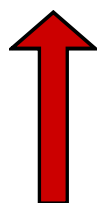
However, if you are required to reverse a string using a stack, you can make use of the following algorithm illustrated in the slides below:

## Algorithm #

**String****Stack**`"H e l l o"`

So we have a string named "Hello" and a stack which is empty for now.

1 of 12

**String****Stack**`"H e l l o"`


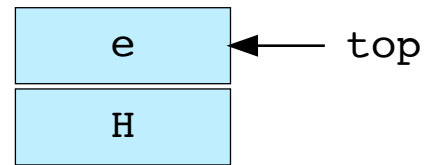
We take 'H' and push it onto the stack.

2 of 12



**String**

"H e l l o"

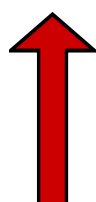
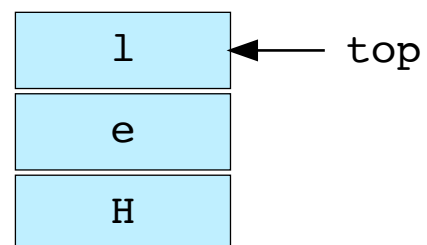
**Stack**

We take 'e' and push it onto the stack.

3 of 12

**String**

"H e l l o"

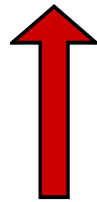
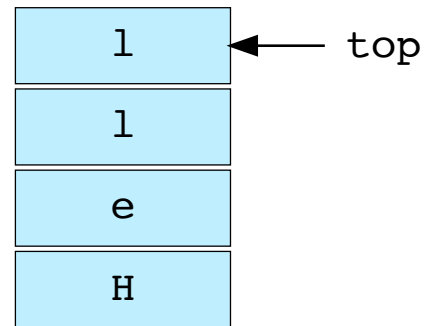
**Stack**

We take 'l' and push it onto the stack.

4 of 12

**String**

"H e l l o"

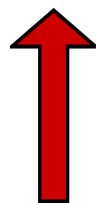
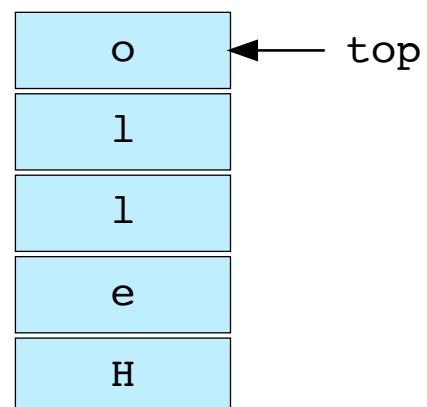
**Stack**

We take 'l' and push it onto the stack.

5 of 12

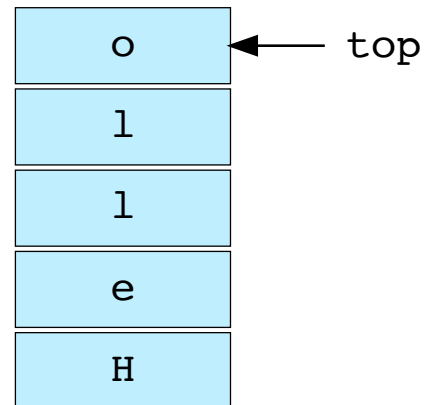
**String**

"H e l l o"

**Stack**

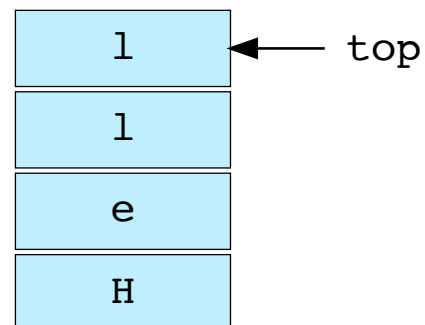
We take 'o' and push it onto the stack.

6 of 12

**String**`"H e l l o"`**Reverse String**`" "`**Stack**

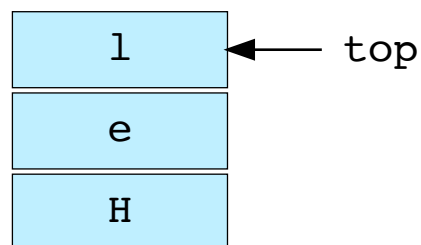
As we have pushed all the characters of the string on to the stack, we initialize our reverse string.

7 of 12

**String**`"H e l l o"`**Reverse String**`"O"`**Stack**

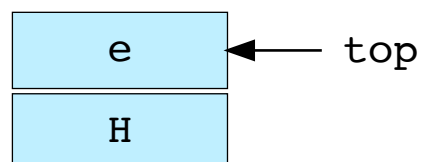
We pop 'o' off the stack and append it to our reverse string.

8 of 12

**String**`"H e l l o"`**Reverse String**`"o l"`**Stack**

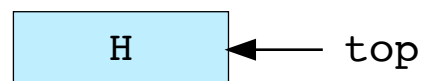
We pop 'l' off the stack and append it to our reverse string.

9 of 12

**String**`"H e l l o"`**Reverse String**`"o l l"`**Stack**

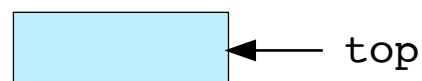
We pop 'l' off the stack and append it to our reverse string.

10 of 12

**String**`"H e l l o"`**Stack****Reverse String**`"o l l e"`

We pop 'e' off the stack and append it to our reverse string.

11 of 12

**String**`"H e l l o"`**Stack****Reverse String**`"o l l e H"`

We pop 'H' off the stack and append it to our reverse string.

12 of 12

Isn't the algorithm pretty straightforward? We push all the characters of the string onto the stack, and due to the *First-In, Last-Out* property of stack, we get all the characters in reverse order when we pop them off the stack.



Now all that is left for us is to code the algorithm shown above. Let's do it!

The stack implementation from the first lesson of this chapter has been provided to you in the file `stack.py`. Check out the code below and feel free to play around with it:

## Implementation #

main.py

stack.py

```
1  from stack import Stack
2  def reverse_string(stack, input_str):
3      for i in range(len(input_str)):
4          stack.push(input_str[i])
5      rev_str = ""
6      while not stack.is_empty():
7          rev_str += stack.pop()
8
9      return rev_str
10
11 stack = Stack()
12 input_str = "!evitacudE ot emocleW"
13 print(reverse_string(stack, input_str))
```

## Explanation #

Let's discuss the `reverse_string` function in the code above. The `for` loop on **line 3** iterates over `input_str` and pushes each character onto the `stack`. Then we initialize `rev_str` as an empty string. We pop off all the elements from the stack and append them to `rev_str` one by one on **line 7** until the stack becomes empty and terminates the `while` loop on **line 6**. We return the `rev_str` on **line 9**.



Easy, right? You can test your understanding and skills in the exercise in the next lesson.

[← Back](#)[Next →](#)[Determine if Brackets are Balanced](#)[Exercise: Convert Decimal Integer to B...](#)

Completed

[Report an Issue](#)[Ask a Question](#)[https://discuss.educative.io/tag/reverse-string\\_\\_stack\\_\\_data-structures-and-algorithms-in-python](https://discuss.educative.io/tag/reverse-string__stack__data-structures-and-algorithms-in-python)



# Exercise: Convert Decimal Integer to Binary

Challenge yourself to solve the problem in this lesson!

We'll cover the following ^

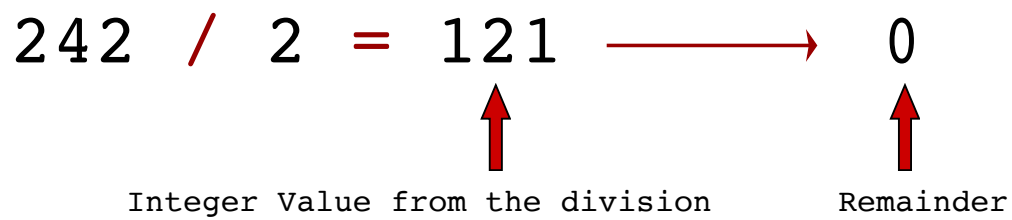
- Division by 2 Method
- Coding Time!

In this coding exercise, you are required to use the stack data structure to convert integer values to their binary equivalent.

## Division by 2 Method #

The slides below show how to use the ***division by 2*** method to compute the binary equivalent for an integer.




$$242 / 2 = 121 \longrightarrow 0$$

Integer Value from the division      Remainder

Divide the number by two.  
Extract the non-fractional part from the answer  
and record the remainder from the division.

1 of 10



$$\begin{array}{rclcl} 242 & / & 2 & = & 121 & \longrightarrow & 0 \\ 121 & / & 2 & = & 60 & \longrightarrow & 1 \end{array}$$

Keep dividing the answer from the previous calculation by two until you reach zero and keep recording the remainders.

2 of 10



$$242 / 2 = 121 \longrightarrow 0$$

$$121 / 2 = 60 \longrightarrow 1$$

$$60 / 2 = 30 \longrightarrow 0$$

3 of 10



$$242 / 2 = 121 \longrightarrow 0$$

$$121 / 2 = 60 \longrightarrow 1$$

$$60 / 2 = 30 \longrightarrow 0$$

$$30 / 2 = 15 \longrightarrow 0$$

4 of 10



$$242 \div 2 = 121 \longrightarrow 0$$

$$121 \div 2 = 60 \longrightarrow 1$$

$$60 \div 2 = 30 \longrightarrow 0$$

$$30 \div 2 = 15 \longrightarrow 0$$

$$15 \div 2 = 7 \longrightarrow 1$$

5 of 10



242	/	2	=	121	→	0
121	/	2	=	60	→	1
60	/	2	=	30	→	0
30	/	2	=	15	→	0
15	/	2	=	7	→	1
7	/	2	=	3	→	1

6 of 10



$$242 / 2 = 121 \longrightarrow 0$$

$$121 / 2 = 60 \longrightarrow 1$$

$$60 / 2 = 30 \longrightarrow 0$$

$$30 / 2 = 15 \longrightarrow 0$$

$$15 / 2 = 7 \longrightarrow 1$$

$$7 / 2 = 3 \longrightarrow 1$$

$$3 / 2 = 1 \longrightarrow 1$$

7 of 10



242	/	2	=	121	→	0
121	/	2	=	60	→	1
60	/	2	=	30	→	0
30	/	2	=	15	→	0
15	/	2	=	7	→	1
7	/	2	=	3	→	1
3	/	2	=	1	→	1
1	/	2	=	0	→	1

8 of 10



LSB (Least Significant Bit)					
242	/	2	=	121	→ 0
121	/	2	=	60	→ 1
60	/	2	=	30	→ 0
30	/	2	=	15	→ 0
15	/	2	=	7	→ 1
7	/	2	=	3	→ 1
3	/	2	=	1	→ 1
1	/	2	=	0	→ 1
MSB (Most Significant Bit)					

You have to read from the bottom of the remainders(MSB) to the top(LSB) to get the binary equivalent of the integer.

9 of 10

LSB (Least Significant Bit)			
242	/ 2 =	121	→ 0
121	/ 2 =	60	→ 1
60	/ 2 =	30	→ 0
30	/ 2 =	15	→ 0
15	/ 2 =	7	→ 1
7	/ 2 =	3	→ 1
3	/ 2 =	1	→ 1
1	/ 2 =	0	→ 1
MSB (Most Significant Bit)			

Therefore the binary equivalent for 242 is:  
11110010

10 of 10

— [ ]

## Coding Time! #

You can build your solution based on **division by 2** method. Your solution should return the correct binary equivalent of `dec_num` as a string from the `convert_int_to_bin(dec_num)` in order to pass the tests.

Make sure that you use stack while solving this challenge. The `stack.py` has been imported to the code. You can make use of the implementation while coding your solution. Remove the `pass` statement if you start implementing your solution.

Good luck!

 Hide Hint

You can use the `//` operator for division if you need the answer to be floored in Python.

main.py

stack.py

```
1 from stack import Stack
2
3 def convert_int_to_bin(dec_num):
4     pass
5
```

 BackNext 

Reverse String

Solution Review: Convert Decimal Inte...

 CompletedReport  
an Issue

Ask a Question

([https://discuss.educative.io/tag/exercise-convert-decimal-integer-to-binary\\_\\_stack\\_\\_data-structures-and-algorithms-in-python](https://discuss.educative.io/tag/exercise-convert-decimal-integer-to-binary__stack__data-structures-and-algorithms-in-python))

# Solution Review: Convert Decimal Integer to Binary

This lesson contains the solution review for the challenge of converting integer values to their binary equivalents.

We'll cover the following ^

- Implementation
- Explanation

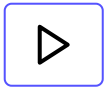
Let's analyze the solution to the exercise in the previous lesson.

## Implementation #

```
1 def convert_int_to_bin(dec_num):
2
3     if dec_num == 0:
4         return 0
5
6     s = Stack()
7
8     while dec_num > 0:
9         remainder = dec_num % 2
10        s.push(remainder)
11        dec_num = dec_num // 2
12
13    bin_num = ""
14    while not s.is_empty():
15        bin_num += str(s.pop())
16
17    return bin_num
18
19 print(convert_int_to_bin(56))
20 print(convert_int_to_bin(2))
21 print(convert_int_to_bin(32))
22 print(convert_int_to_bin(10))
23
```



```
24 print(int(convert_int_to_bin(56),2)==56)
```



## Explanation #

On **line 3**, we cater to an edge case of `dec_num` being equal to `0`. If `dec_num` is equal to `0`, we return `0` on **line 4** as the binary equivalent for the decimal number `0` is `0`.

On **line 6**, we declare a stack and proceed to a `while` loop on **line 8** which executes if `dec_num` is greater than `0`.

As stated in the **division by 2** method, we calculate the remainder of the division of `dec_num` by `2` and push it onto the stack (**lines 9-10**). Then we divide `dec_num` by `2` using the `//` operator to `dec_num` which floors the answer of the division, and we update `dec_num` with the answer (**line 11**). We keep executing the code on **lines 9-11** as long as `dec_num` is greater than `0`. As soon as `dec_num` becomes equal to or less than `0`, the `while` loop terminates.

On **line 13**, `bin_num` is declared as an empty string. The `while` loop on the very next line executes if the stack `s` is *not* empty. If `s` is not empty, we pop a value from `s` and append it to the `bin_num` string on **line 15**. We keep popping elements from `s` until it becomes empty and the `while` loop is terminated.

The `bin_num` is returned from the function on **line 17**.

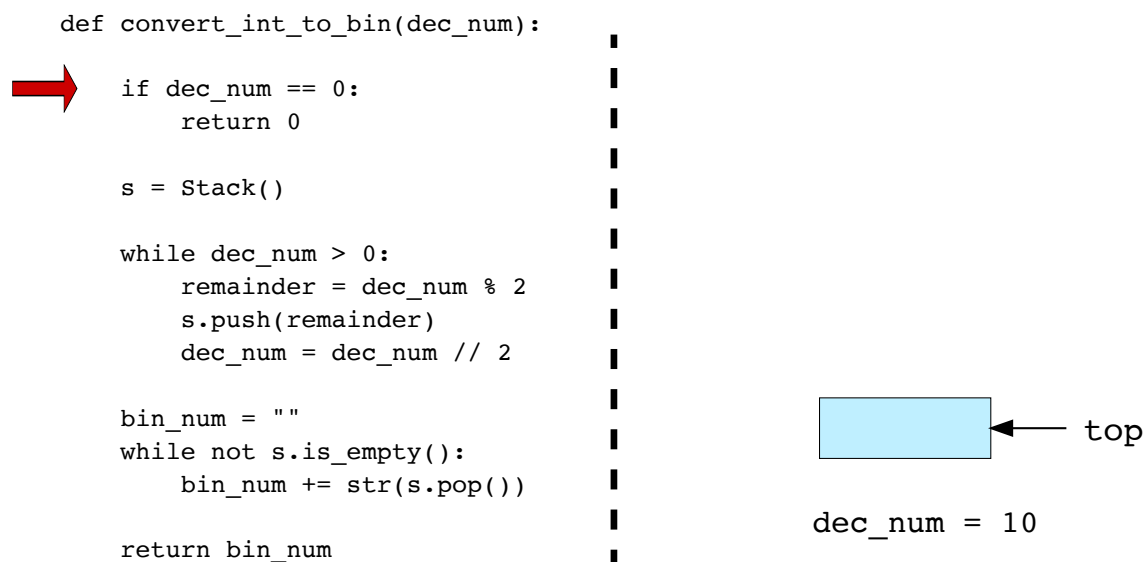
The following code helps us to evaluate whether our implementation is correct or not:

```
print(int(convert_int_to_bin(56),2)==56)
```

The above statement will print `True` if `convert_int_to_bin(56)` returns the correct binary equivalent for 56. We convert the returned value from `convert_int_to_bin(56)` to an integer value by specifying base 2 of the returned value. It will convert to 56 if it's equal to 56 in binary format. Otherwise, the statement will print `False` if we get some number other than 56.

In this problem, the *First-In, Last-Out* property of the stack has enabled us to store the binary bits from the *MSB* (Most Significant Bit) to the *LSB* (Least Significant Bit), although we get the values in reverse order by the *division by 2* method.

Below are some slides that will help you understand the code even better:



```
def convert_int_to_bin(dec_num):
    if dec_num == 0:
        return 0

    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())


    return bin_num
```

dec\_num = 10

1 of 30



```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    → s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```

 ← top

dec\_num = 10

2 of 30

```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    → while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```

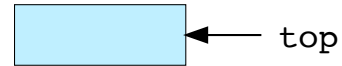
 ← top

dec\_num = 10

3 of 30



```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



dec\_num = 10  
remainder = 0

4 of 30

```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



dec\_num = 10  
remainder = 0

5 of 30





```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



0 ← top

dec\_num = 5  
remainder = 0

6 of 30

```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



0 ← top

dec\_num = 5  
remainder = 0

7 of 30



```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



0 ← top

dec\_num = 5  
remainder = 1

8 of 30

```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



1  
0 ← top

dec\_num = 5  
remainder = 1

9 of 30



```
def convert_int_to_bin(dec_num):

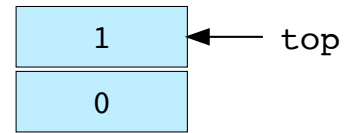
    if dec_num == 0:
        return 0

    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())

    return bin_num
```



```
dec_num = 2
remainder = 1
```

10 of 30

```
def convert_int_to_bin(dec_num):

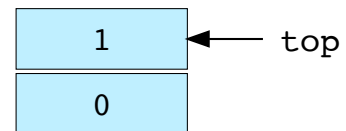
    if dec_num == 0:
        return 0

    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())

    return bin_num
```

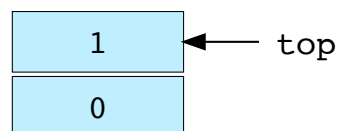


```
dec_num = 2
remainder = 1
```

11 of 30



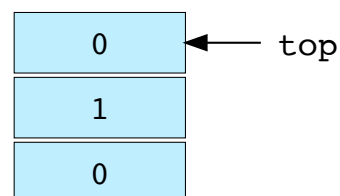
```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



dec\_num = 2  
remainder = 0

12 of 30

```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```

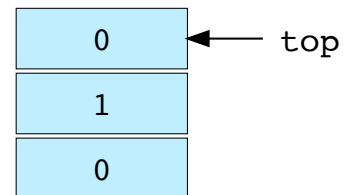


dec\_num = 2  
remainder = 0

13 of 30



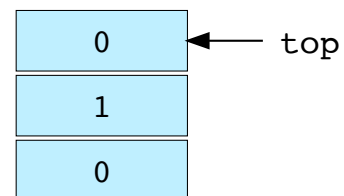
```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



dec\_num = 1  
remainder = 0

14 of 30

```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```

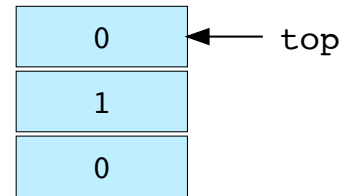


dec\_num = 1  
remainder = 0

15 of 30



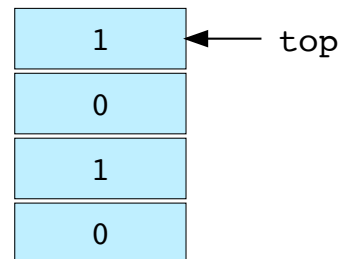
```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



dec\_num = 1  
remainder = 1

16 of 30

```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```

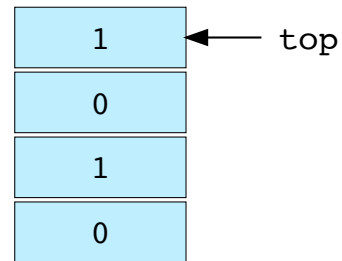


dec\_num = 1  
remainder = 1

17 of 30



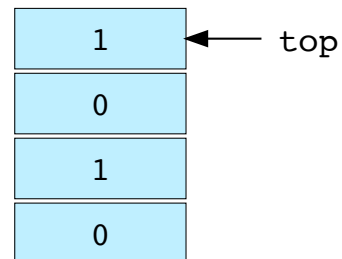
```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



dec\_num = 0  
remainder = 1

18 of 30

```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



dec\_num = 0  
remainder = 1

19 of 30



```
def convert_int_to_bin(dec_num):
```

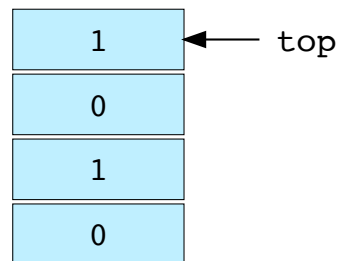
```
    if dec_num == 0:  
        return 0
```

```
    s = Stack()
```

```
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2
```



```
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



```
dec_num = 0  
remainder = 1  
bin_num = ""
```

20 of 30

```
def convert_int_to_bin(dec_num):
```

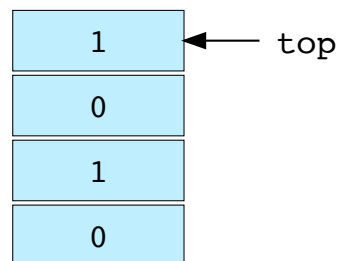
```
    if dec_num == 0:  
        return 0
```

```
    s = Stack()
```

```
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2
```



```
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



```
dec_num = 0  
remainder = 1  
bin_num = ""
```

21 of 30





```
def convert_int_to_bin(dec_num):

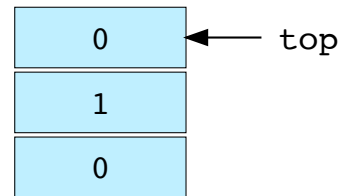
    if dec_num == 0:
        return 0

    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())

    return bin_num
```



```
dec_num = 0
remainder = 1
bin_num = "1"
```

22 of 30

```
def convert_int_to_bin(dec_num):

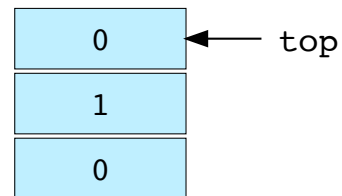
    if dec_num == 0:
        return 0

    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())

    return bin_num
```

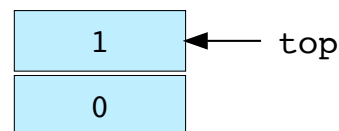


```
dec_num = 0
remainder = 1
bin_num = "1"
```

23 of 30



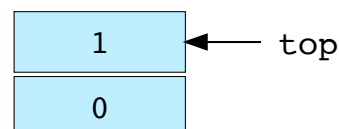
```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



```
dec_num = 0  
remainder = 1  
bin_num = "10"
```

24 of 30

```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
    return bin_num
```



```
dec_num = 0  
remainder = 1  
bin_num = "10"
```

25 of 30



```
def convert_int_to_bin(dec_num):

    if dec_num == 0:
        return 0

    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())

    return bin_num
```



0 ← top

```
dec_num = 0
remainder = 1
bin_num = "101"
```

26 of 30

```
def convert_int_to_bin(dec_num):

    if dec_num == 0:
        return 0

    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())

    return bin_num
```



0 ← top

```
dec_num = 0
remainder = 1
bin_num = "101"
```

27 of 30



```
def convert_int_to_bin(dec_num):

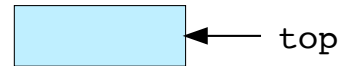
    if dec_num == 0:
        return 0

    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())

    return bin_num
```



```
dec_num = 0
remainder = 1
bin_num = "1010"
```

28 of 30

```
def convert_int_to_bin(dec_num):

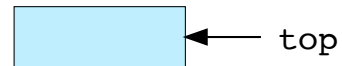
    if dec_num == 0:
        return 0

    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())

    return bin_num
```

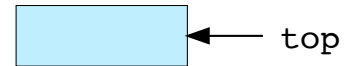


```
dec_num = 0
remainder = 1
bin_num = "1010"
```

29 of 30



```
def convert_int_to_bin(dec_num):  
  
    if dec_num == 0:  
        return 0  
  
    s = Stack()  
  
    while dec_num > 0:  
        remainder = dec_num % 2  
        s.push(remainder)  
        dec_num = dec_num // 2  
  
    bin_num = ""  
    while not s.is_empty():  
        bin_num += str(s.pop())  
  
→ return bin_num
```



```
dec_num = 0  
remainder = 1  
bin_num = "1010"
```

30 of 30