Improving our first program
Our first program

```python
n = int(input("Enter a positive integer:"))
while n > 0:
    print(n % 2)
    n = n // 2
```
Revisiting **while**-loops

Line 1

```python
while boolean expression:
    Line 2
    Line 3
Line 4
```

- **while**-loops affect the *flow* of the program, i.e., the order in which program statements are executed.

- For the above example the flow of the program is:

  Line 1, bool expr (True), Line 2, Line 3, bool expr (True), Line 2, Line 3, bool expr (False), Line 4
• Lines 2 and 3 form the *body* of the while loop

• Python uses indentation to identify the lines following the while statement that constitute the body of the while loop.
Our first program

```python
n = int(input("Enter a positive integer:"))
while n > 0:
    print(n % 2)
    n = n // 2
```

- Suppose \( n \) has value 35 initially.
- Then the sequence of values that \( n \) takes on is:
  
  35, 17, 8, 4, 2, 1, 0.

- When the value of \( n \) becomes 0, then the boolean expression in the while-statement becomes false and the while-loop ends.
while-loops example 2: Counting up

```python
n = int(input("Please type a positive integer: "))

count = 0  # Initialization. It is easy to forget this.
while count < n:
    print(count)
    count = count + 1

print("Done")
```

- What is the output if the user types 10 in response to the prompt?
while-loops example 3: Counting down

```python
n = int(input("Please type a positive integer: "))

while n > 0:
    print(n)
    n = n - 1

print("Done")
```

- What is the output if the user types 10 in response to the prompt?
while-loops example 4: Accumulating a sum

```python
n = int(input("Please type a positive integer: "))

total = 0  # Initially the total has value 0
while n > 0:
    total = total + n
    n = n - 1

print(total)
```

- What is the output if the user types 10 in response to the prompt?
while-loops example 4: Accumulating a product

```python
n = int(input("Please type a positive integer: "))
product = 1  # Initially the product has value 1
while n > 0:
    product = product * n
    n = n - 1
print(product)
```

- What is the output if the user types 10 in response to the prompt?
The current program generates bits one by one in the wrong order!

How can we put together the bits we generate, in the correct order, to construct the binary equivalent?

**String concatenation!**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>“0” + “1001”</td>
<td>“01001”</td>
</tr>
<tr>
<td>“1” + “1001”</td>
<td>“11001”</td>
</tr>
</tbody>
</table>
Algorithmic idea

- After $i$ iterations of the while loop we have generated the right most $i$ bits of our answer.

- Call this the *length*-i suffix.

- We want to maintain a string that grows as:
### Example

- Input is 39.

<table>
<thead>
<tr>
<th>Output</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;11&quot;</td>
</tr>
<tr>
<td>0</td>
<td>&quot;111&quot;</td>
</tr>
<tr>
<td>0</td>
<td>&quot;0111&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;00111&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;100111&quot;</td>
</tr>
</tbody>
</table>
n = int(input("Enter a positive integer:"))
suffix = ""
while n > 0:
    suffix = str(n % 2) + suffix
    n = n // 2
print(suffix)
Further improvement

Now suppose that we want a more informative output message:
The binary equivalent of 39 is 100111

Will this work?

```python
n = int(input("Enter a positive integer:"))
suffix = ""
while n > 0:
    suffix = str(n % 2) + suffix
    n = n // 2
print("The binary equivalent of ", n, " is ", suffix)
```
Here is what works

```python
n = int(input("Enter a positive integer:"))
suffix = ""
originalN = n
while n > 0:
    suffix = str(n % 2) + suffix
    n = n // 2
print("The binary equivalent of", originalN, "is", suffix)
```