# Improving our first program

JAN 28, 2015

# Our first program

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

# Revisiting while-loops

```
Line 1
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

# Body of while loop

- 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

```
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

```
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

```
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

```
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

```
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?

# Improving the output

- 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**!

| Expression | Value |
|---|---|
| "0" + "1001" | "01001" |
| "1" + "1001" | "11001" |

# 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.

| Output | Suffix |
|---|---|
| 1 | "" |
| 1 | "1" |
| 1 | "11" |
| 0 | "111" |
| 0 | "0111" |
| 1 | "00111" |
| | "100111" |

# Improved program

```
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?

```
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

```
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)
```