Outline

1. Digital Systems
   half adders
   adder circuits

2. Looping Constructs
   the while loop
   the for loop

3. Designing Loops
   Euclid's algorithm

4. Summary + Assignments

MCS 260 Lecture 11
Introduction to Computer Science
Jan Verschelde, 19 September 2008
adder circuits
while and for loops

1 Digital Systems
   half adders
   adder circuits

2 Looping Constructs
   the while loop
   the for loop

3 Designing Loops
   Euclid’s algorithm

4 Summary + Assignments
Half Adders
adding two bits

A half adder takes on input two bits $b_1$ and $b_2$, and returns the sum $S$ and the carry over $C$.

As logical functions, they are:
- $C = b_1 \AND b_2$
- $S = (\NOT b_1 \AND b_2) \OR (b_1 \AND \NOT b_2)$

<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$C$</th>
<th>$\NOT b_1 \AND b_2$</th>
<th>$b_1 \AND \NOT b_2$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Does $S$ look familiar? $S = \text{xor} = \text{exclusive or}$
Half Adders
adding two bits

A half adder takes on input two bits $b_1$ and $b_2$, and returns the sum $S$ and the carry over $C$

As logical functions, they are

$C = b_1 \text{ AND } b_2$

$S = (\text{ NOT } b_1\text{ AND } b_2) \text{ OR } (b_1\text{ AND NOT } b_2)$

<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$C$</th>
<th>NOT $b_1$ AND $b_2$</th>
<th>$b_1$ AND NOT $b_2$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Does $S$ look familiar? $S = \text{ xor } = \text{ exclusive or}$
Half Adders
adding two bits

A half adder takes on input two bits \( b_1 \) and \( b_2 \), and returns the sum \( S \) and the carry over \( C \).

As logical functions, they are:

\[
C = b_1 \text{ AND } b_2 \\
S = (\text{ NOT } b_1 \text{ AND } b_2) \text{ OR } (b_1 \text{ AND NOT } b_2)
\]

<table>
<thead>
<tr>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( C )</th>
<th>( \text{NOT } b_1 \text{ AND } b_2 )</th>
<th>( b_1 \text{ AND NOT } b_2 )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Does \( S \) look familiar? \( S = \text{xor} = \text{exclusive or} \)
Half Adders
adding two bits

A half adder takes as input two bits \( b_1 \) and \( b_2 \), and returns the sum \( S \) and the carry over \( C \).

As logical functions, they are:

\[
C = b_1 \text{ AND } b_2 \\
S = (\text{ NOT } b_1 \text{ AND } b_2) \text{ OR } (b_1 \text{ AND NOT } b_2)
\]

<table>
<thead>
<tr>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( C )</th>
<th>( \text{NOT } b_1 \text{ AND } b_2 )</th>
<th>( b_1 \text{ AND NOT } b_2 )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Does \( S \) look familiar? \( S = \text{ xor } = \text{ exclusive or} \)
Realization of a Half Adder
with logic gates: 2 NOTs, 3 ANDs, and 1 OR

\[ C = b_1 \text{ AND } b_2 \]
\[ S = ( \text{NOT } b_1 \text{ AND } b_2 ) \text{ OR } ( b_1 \text{ AND } \text{NOT } b_2 ) \]

exercise: realize with NAND & NOR
Realization of a Half Adder
with logic gates: 2 NOTs, 3 ANDs, and 1 OR

C = b_1 \ AND \ b_2
S = ( NOT b_1 \ AND \ b_2 ) \ OR \ ( b_1 \ AND \ NOT \ b_2 )

exercise: realize with NAND & NOR
Realization of a Half Adder

with logic gates: 2 NOTs, 3 ANDs, and 1 OR

\[ C = b_1 \text{ AND } b_2 \]
\[ S = ( \text{NOT } b_1 \text{ AND } b_2 ) \text{ OR } ( b_1 \text{ AND NOT } b_2 ) \]

exercise: realize with NAND & NOR
Realization of a Half Adder
with logic gates: 2 NOTs, 3 ANDs, and 1 OR

\[ C = b_1 \text{ AND } b_2 \]
\[ S = ( \text{NOT } b_1 \text{ AND } b_2 ) \text{ OR } ( b_1 \text{ AND NOT } b_2 ) \]

exercise: realize with NAND & NOR
Realization of a Half Adder
with logic gates: 2 NOTs, 3 ANDs, and 1 OR

\[
C = b_1 \text{ AND } b_2 \\
S = ( \text{ NOT } b_1 \text{ AND } b_2 ) \text{ OR } ( b_1 \text{ AND NOT } b_2 )
\]

exercise: realize with NAND & NOR
Realization of a Half Adder
with logic gates: 2 NOTs, 3 ANDs, and 1 OR

\[ C = b_1 \text{ AND } b_2 \]
\[ S = ( \text{ NOT } b_1 \text{ AND } b_2 ) \text{ OR } ( b_1 \text{ AND NOT } b_2 ) \]

exercise: realize with NAND & NOR
adder circuits
while and for loops

1. Digital Systems
   half adders
   adder circuits

2. Looping Constructs
   the while loop
   the for loop

3. Designing Loops
   Euclid’s algorithm

4. Summary + Assignments
Adder Circuits
adding sequences of bits

To add sequences of bits, we need a more complex circuit, an **adder** which has three inputs:

1. the two bits to add at the current position;
2. the carry over from the previous position.

To speedup the addition of bit sequences, circuits are enabled with **carry look ahead**.

Other operations that run fast on bit sequences are

- **rotate**: 1101 $\rightarrow$ 1011
- **shift**: 1101 $\rightarrow$ 1010
Adder Circuits

adding sequences of bits

To add sequences of bits, we need a more complex circuit, an adder which has three inputs:

1. the two bits to add at the current position;
2. the carry over from the previous position.

To speed up the addition of bit sequences, circuits are enabled with carry look ahead.

Other operations that run fast on bit sequences are:

- rotate: 1101 → 1011
- shift: 1101 → 1010
Adder Circuits

adding sequences of bits

To add sequences of bits, we need a more complex circuit, an **adder** which has three inputs:

1. the two bits to add at the current position;
2. the carry over from the previous position.

To speed up the addition of bit sequences, circuits are enabled with **carry look ahead**.

Other operations that run fast on bit sequences are:

- rotate: $1101 \rightarrow 1011$
- shift: $1101 \rightarrow 1010$
Adder Circuits
adding sequences of bits

To add sequences of bits, we need a more complex circuit, an \textit{adder} which has three inputs:

1. the two bits to add at the current position;
2. the carry over from the previous position.

To speedup the addition of bit sequences, circuits are enabled with \textit{carry look ahead}.

Other operations that run fast on bit sequences are

- \textit{rotate}: $1101 \rightarrow 1011$
- \textit{shift}: $1101 \rightarrow 1010$
To add sequences of bits, we need a more complex circuit, an **adder** which has three inputs:

1. the two bits to add at the current position;
2. the carry over from the previous position.

To speed up the addition of bit sequences, circuits are enabled with **carry look ahead**.

Other operations that run fast on bit sequences are

- **rotate**: $1101 \rightarrow 1011$
- **shift**: $1101 \rightarrow 1010$
Adder Circuits
adding sequences of bits

To add sequences of bits, we need a more complex circuit, an adder which has three inputs:

1. the two bits to add at the current position;
2. the carry over from the previous position.

To speedup the addition of bit sequences, circuits are enabled with carry look ahead.

Other operations that run fast on bit sequences are

- rotate: $1101 \rightarrow 1011$
- shift: $1101 \rightarrow 1010$
Bitwise Operators in Python

Multiplying numbers with a power of two is just shifting bits to the left, padding with zeroes to the right.

Using the bitwise operator `<<` we efficiently create large numbers, e.g. $2^{100}$:

```python
>>> 1 << 100
1267650600228229401496703205376L
```

Shifting $k$ bits to the right, dividing by $2^k$ is done via `>>`.

A summary of bitwise operators:

<table>
<thead>
<tr>
<th>operator</th>
<th>example</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>~ $a$</td>
<td>not $a$</td>
</tr>
<tr>
<td>&amp;</td>
<td>$a$ &amp; $b$</td>
<td>$a$ and $b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a$</td>
</tr>
<tr>
<td>^</td>
<td>$a$ ^ $b$</td>
<td>$a$ xor $b$</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>$a$ &lt;&lt; $b$</td>
<td>$a \times 2^b$</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>$a$ &gt;&gt; $b$</td>
<td>$a / 2^b$</td>
</tr>
</tbody>
</table>
Bitwise Operators in Python

Multiplying numbers with a power of two is just shifting bits to the left, padding with zeroes to the right.

Using the bitwise operator `<<` we efficiently create large numbers, e.g. $2^{100}$:

```python
>>> 1 << 100
1267650600228229401496703205376L
```

Shifting $k$ bits to the right, dividing by $2^k$ is done via `>>`. A summary of bitwise operators:

<table>
<thead>
<tr>
<th>operator</th>
<th>example</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>~ $a$</td>
<td>not $a$</td>
</tr>
<tr>
<td>&amp;</td>
<td>$a$ &amp; $b$</td>
<td>$a$ and $b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a$</td>
</tr>
<tr>
<td>^</td>
<td>$a$ ^ $b$</td>
<td>$a$ xor $b$</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>$a$ &lt;&lt; $b$</td>
<td>$a \times 2^b$</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>$a$ &gt;&gt; $b$</td>
<td>$a/2^b$</td>
</tr>
</tbody>
</table>
Bitwise Operators in Python

Multiplying numbers with a power of two is just shifting bits to the left, padding with zeroes to the right.

Using the bitwise operator `<<` we efficiently create large numbers, e.g. $2^{100}$:

```
>>> 1 << 100
1267650600228229401496703205376L
```

Shifting $k$ bits to the right, dividing by $2^k$ is done via `>>`. A summary of bitwise operators:

<table>
<thead>
<tr>
<th>operator</th>
<th>example</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>~ a</td>
<td>not a</td>
</tr>
<tr>
<td>&amp;</td>
<td>a &amp; b</td>
<td>a and b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>^</td>
<td>a ^ b</td>
<td>a xor b</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>a &lt;&lt; b</td>
<td>$a \times 2^b$</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>a &gt;&gt; b</td>
<td>$a/2^b$</td>
</tr>
</tbody>
</table>
adder circuits
while and for loops

1. Digital Systems
   half adders
   adder circuits

2. Looping Constructs
   the while loop
   the for loop

3. Designing Loops
   Euclid’s algorithm

4. Summary + Assignments
Yearly Balance of an Investment

script `yieldbal.py` of lecture 4 revisited

The balance $B$ of an investment at rate $r$ grows as

$$B = B(1 + r).$$

To look at the annual growth of the balance of an investment, consider:

**Input**: amount to invest, annual interest rate (as %), the number of years.

**Output**: starting at year 0, the balance is written to screen for each year.

**Algorithm**: compute $B \times (1 + r/100)$ as many times as the number of years.
Yearly Balance of an Investment

script yieldbal.py of lecture 4 revisited

The balance $B$ of an investment at rate $r$ grows as

$$B = B(1 + r).$$

To look at the annual growth of the balance of an investment, consider:

**Input:** amount to invest,
annual interest rate (as %),
the number of years.

**Output:** starting at year 0, the balance is written to screen for each year.

**Algorithm:** compute $B \times (1 + r/100)$
as many times as the number of years.
Yearly Balance of an Investment

script `yieldbal.py` of lecture 4 revisited

The balance $B$ of an investment at rate $r$ grows as

$$B = B(1 + r).$$

To look at the annual growth of the balance of an investment, consider:

**Input:** amount to invest, annual interest rate (as %), the number of years.

**Output:** starting at year 0, the balance is written to screen for each year.

**Algorithm:** compute \( B \times (1 + r/100) \) as many times as the number of years.
Yearly Balance of an Investment

script `yieldbal.py` of lecture 4 revisited

The balance $B$ of an investment at rate $r$ grows as

$$B = B(1 + r).$$

To look at the annual growth of the balance of an investment, consider:

**Input:** amount to invest,
annual interest rate (as %),
the number of years.

**Output:** starting at year 0, the balance is written to screen for each year.

**Algorithm:** compute $B \times (1 + r/100)$
as many times as the number of years.
Running the Program

Running `showbal.py` at the command prompt `$`:

```
$python showbal.py
```

Calculation of the annual balance
Give amount to invest : 1000
Give annual interest rate : 3.14
Give number of years : 2
At year 0 : Balance = $1000.00
At year 1 : Balance = $1031.40
At year 2 : Balance = $1063.79

Names of variables:

- Balance of investment: $B$
- Annual interest rate: $r$
- Number of years: $n$
- Counter for the years: $i$
Running the Program

Running `showbal.py` at the command prompt $:$

```bash
$python showbal.py
```

Calculation of the annual balance
Give amount to invest : 1000
Give annual interest rate : 3.14
Give number of years : 2
At year 0 : Balance = $1000.00
At year 1 : Balance = $1031.40
At year 2 : Balance = $1063.79

Names of variables:

- Balance of investment: \( B \)
- Annual interest rate: \( r \)
- Number of years: \( n \)
- Counter for the years: \( i \)
Running the Program

Running `showbal.py` at the command prompt `$`:

```
$python showbal.py
```

Calculation of the annual balance
Give amount to invest : 1000
Give annual interest rate : 3.14
Give number of years : 2
At year 0 : Balance = $1000.00
At year 1 : Balance = $1031.40
At year 2 : Balance = $1063.79

Names of variables:

- Balance of investment: `B`
- Annual interest rate: `r`
- Number of years: `n`
- Counter for the years: `i`
The Flowchart for the Program

B, r, n = input()
print 0, B
i = 1

while i <= n:
    B = B*(1+r/100)
    print i, B
    i = i + 1
The Flowchart for the Program

```
B, r, n = input()
print 0, B
i = 1
while i <= n:
    B = B*(1+r/100)
    print i, B
    i = i + 1
```

The Flowchart for the Program

```python
B, r, n = input()
print 0, B
i = 1

while i <= n:
    B = B*(1+r/100)
    print i, B
    i = i + 1
```

The Flowchart for the Program

```
B, r, n = input()
print 0, B
i = 1

while i <= n:
    B = B * (1 + r / 100)
    print i, B
    i = i + 1
```
The while Loop
continue until condition becomes false

The flowchart of the while loop:

- Initialize variables
- Continue? (branch)
  - True: Body of loop
  - False: Next iteration

The flowchart illustrates the process of initializing variables, checking the condition, and iterating based on the outcome.
The while Loop
continue until condition becomes false

The flowchart of the while loop:

initialize variables

continue? True

body of loop

continue? False
The while Loop
continue until condition becomes false

The flowchart of the while loop:

1. Initialize variables
2. Check continue condition
   - If False, exit loop
   - If True, proceed to body of loop
3. Execute body of loop
4. Return to check continue condition
The while Loop
continue until condition becomes false

The flowchart of the while loop:

initialize variables

continue?

body of loop

True

False
Syntax of the while Statement

A while statement has the format
< initialize variables >
while < continue ? >
    < body of loop >
All statements in the body must be indentated!

The calculations in showbal.py:

```python
i = 1
while i <= n:
    B *= (1 + r/100)
    i += 1
```

forgetting i += 1 results in an infinite loop!

Press the keys `ctrl` and `c` simultaneously to exit.
Also the Unix command `kill` terminates processes.
Syntax of the while Statement

A `while` statement has the format

```
< initialize variables >
while < continue ? >
   < body of loop >
```

All statements in the body must be indentated!

The calculations in `showbal.py`:
```
i = 1
while i <= n:
   B *= (1 + r/100)
   i += 1
```

Forgetting `i += 1` results in an infinite loop!

Press the keys `ctrl` and `c` simultaneously to exit.
Also the Unix command `kill` terminates processes.
Syntax of the while Statement

A while statement has the format
< initialize variables >
while < continue ? >
    < body of loop >
All statements in the body must be indentated!

The calculations in showbal.py:

```python
i = 1
while i <= n:
    B *= (1 + r/100)
    i += 1
```

forgetting i += 1 results in an infinite loop!

Press the keys ctrl and c simultaneously to exit.
Also the Unix command kill terminates processes.
The program showbal.py

# L-11 MCS 260 Fri 19 Sep 2008 a while loop
#
# Shows annual growth of an investment.
# The user enters principal, interest rate, and the number of years. After each year
# the balance of the investment are shown.
#
print 'Calculation of the annual balance'
B = input('Give amount to invest : ')
r = input('Give annual interest rate : ')
n = input('Give number of years : ')
print 'At year %d : Balance = $%.2f' % (0,B)
i = 1
while i <= n:
    B *= (1 + r/100)
    print 'At year %d : Balance = $%.2f' % (i,B)
i += 1
The program showbal.py

```python
# L-11 MCS 260 Fri 19 Sep 2008 a while loop
#
# Shows annual growth of an investment.
# The user enters principal, interest rate, 
# and the number of years. After each year 
# the balance of the investment are shown.
#
print 'Calculation of the annual balance'
B = input('Give amount to invest : ')
r = input('Give annual interest rate : ')
n = input('Give number of years : ')
print 'At year %d : Balance = $%.2f' % (0,B)
i = 1

while i <= n:
    B *= (1 + r/100)
    print 'At year %d : Balance = $%.2f' % (i,B)
i += 1
```
The program showbal.py

# L-11 MCS 260 Fri 19 Sep 2008  a while loop  
#  
# Shows annual growth of an investment.  
# The user enters principal, interest rate,  
# and the number of years.  After each year  
# the balance of the investment are shown.  
#  
print 'Calculation of the annual balance'  
B = input('Give amount to invest : ')  
r = input('Give annual interest rate : ')  
n = input('Give number of years : ')  
print 'At year %d : Balance = $%.2f' % (0,B)  
i = 1  
while i <= n:  
    B *= (1 + r/100)  
    print 'At year %d : Balance = $%.2f' % (i,B)  
i += 1
adder circuits
while and for loops

1 Digital Systems
   half adders
   adder circuits

2 Looping Constructs
   the while loop
   the for loop

3 Designing Loops
   Euclid’s algorithm

4 Summary + Assignments
The for Loop
enumerating items

Because enumerating items occurs so frequently, the for statement does automatically

1. the initialization of the counter, and
2. the update of the counter in the body of the loop.

Syntax of the for statement:

```
for < counter > in < sequence > :
    < body of loop >
```

where < sequence > is typically a range of numbers:

```
>>> range(3,11)
[3, 4, 5, 6, 7, 8, 9, 10]
```

Note the ending of the range!

What type is returned by range()?
The for Loop

Because enumerating items occurs so frequently, the `for` statement does automatically

1. the initialization of the counter, and
2. the update of the counter in the body of the loop.

Syntax of the `for` statement:

```python
for < counter > in < sequence > :
    < body of loop >
```

where `< sequence >` is typically a `range` of numbers:

```python
>>> range(3,11)
[3, 4, 5, 6, 7, 8, 9, 10]
```

Note the ending of the range!

*What type is returned by `range()`?*
Using for instead of while

Recall the calculations in `showbal.py`:

```python
i = 1
while i <= n:
    B *= (1 + r/100)
    i += 1
```

With a `for`, fewer lines of code are needed:

```python
for i in range(1, n+1):
    B *= (1 + r/100)
```
Using for instead of while

Recall the calculations in showbal.py:

```python
i = 1
while i <= n:
    B *= (1 + r/100)
    i += 1
```

With a `for`, fewer lines of code are needed:

```python
for i in range(1,n+1):
    B *= (1 + r/100)
```
Showing the Balance with for

# L-11 MCS 260 Fri 19 Sep 2008 a for loop#
#
# Shows annual growth of an investment.
# The user enters principal, interest rate, 
# and the number of years. After each year
# the balance of the investment are shown.
#
print 'Calculation of the annual balance'
B = input('Give amount to invest : ') 
r = input('Give annual interest rate : ') 
n = input('Give number of years : ') 
print 'At year %d : Balance = $%.2f' % (0,B)
for i in range(1,n+1): 
    B *= (1 + r/100)
    print 'At year %d : Balance = $%.2f' % (i,B)
Showing the Balance with for

```python
# L-11 MCS 260 Fri 19 Sep 2008  a for loop
#
# Shows annual growth of an investment.
# The user enters principal, interest rate, and the number of years. After each year
# the balance of the investment are shown.
#
print 'Calculation of the annual balance'
B = input('Give amount to invest : ')
r = input('Give annual interest rate : ')
n = input('Give number of years : ')
print 'At year %d : Balance = $%.2f' % (0,B)
for i in range(1,n+1):
    B *= (1 + r/100)
    print 'At year %d : Balance = $%.2f' % (i,B)
```
Showing the Balance with for

# L-11 MCS 260 Fri 19 Sep 2008 a for loop
#
# Shows annual growth of an investment.
# The user enters principal, interest rate, 
# and the number of years. After each year 
# the balance of the investment are shown. 
#
print 'Calculation of the annual balance'
B = input('Give amount to invest : ')
r = input('Give annual interest rate : ')
n = input('Give number of years : ')
print 'At year %d : Balance = $%.2f' % (0,B)
for i in range(1,n+1):
    B *= (1 + r/100)
    print 'At year %d : Balance = $%.2f' % (i,B)
List Comprehensions
doing things properly in Python

We noticed already that `range()` returns a list. Combining lists and for loops to show evolving balance:

```python
>>> B = 1000; r = 3.14
>>> L = [ B*(1+r/100)**i for i in range(0,5) ]
```

```python
>>> L
[1000.0, 1031.4000000000001, 1063.7859600000002, 1097.1888391440004, 1131.6405686931218]
```

General syntax of a list comprehension:

```python
[ <function of i> for <i> in range(<a>, <b>) ]
```

Power of Python: all calculations done in one line of code!
List Comprehensions

doing things properly in Python

We noticed already that `range()` returns a list. Combining lists and for loops to show evolving balance:

```python
>>> B = 1000; r = 3.14
>>> L = [ B*(1+r/100)**i for i in range(0,5) ]
>>> L
[1000.0, 1031.4000000000001, 1063.7859600000002, 1097.1888391440004, 1131.6405686931218]
```

General syntax of a list comprehension:

```python
[ <function of i> for <i> in range(<a>, <b>) ]
```

Power of Python: all calculations done in one line of code!
List Comprehensions
doing things properly in Python

We noticed already that `range()` returns a list. Combining lists and for loops to show evolving balance:

```python
>>> B = 1000; r = 3.14
>>> L = [ B*(1+r/100)**i for i in range(0,5) ]
>>> L
[1000.0, 1031.4000000000001, 1063.7859600000002, 1097.1888391440004, 1131.6405686931218]
```

General syntax of a `list comprehension`:

```
[ <function of i> for <i> in range(<a>, <b>) ]
```

Power of Python: all calculations done in one line of code!
adder circuits
while and for loops

1. Digital Systems
   half adders
   adder circuits

2. Looping Constructs
   the while loop
   the for loop

3. Designing Loops
   Euclid’s algorithm

4. Summary + Assignments
The Greatest Common Divisor

**Input:** two positive numbers \(a > b > 0\).

**Output:** the greatest common divisor of \(a\) and \(b\).

An example: \(a = 60, b = 51\)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(b)</td>
<td>(r = a \text{%} b)</td>
</tr>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders \(r = a \text{\%} b\)
- the sequence stops when \(r\) equals zero
The Greatest Common Divisor

**Input:** two positive numbers $a > b > 0$.

**Output:** the greatest common divisor of $a$ and $b$.

An example: $a = 60$, $b = 51$

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$r = a % b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders $r = a \% b$
- the sequence stops when $r$ equals zero
The Greatest Common Divisor

Input: two positive numbers $a > b > 0$.

Output: the greatest common divisor of $a$ and $b$.

An example: $a = 60$, $b = 51$

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$r = a % b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders $r = a \% b$
- the sequence stops when $r$ equals zero
The Greatest Common Divisor

Input: two positive numbers \( a > b > 0 \).

Output: the greatest common divisor of \( a \) and \( b \).

An example: \( a = 60, \ b = 51 \)

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( r = a % b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders \( r = a \% b \)
- the sequence stops when \( r \) equals zero
The Greatest Common Divisor

Input: two positive numbers \( a > b > 0 \).

Output: the greatest common divisor of \( a \) and \( b \).

An example: \( a = 60, b = 51 \)

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( r = a % b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders \( r = a \% b \)
- the sequence stops when \( r \) equals zero
The Greatest Common Divisor

**Input:** two positive numbers $a > b > 0$.

**Output:** the greatest common divisor of $a$ and $b$.

An example: $a = 60$, $b = 51$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
<td>$r = a % b$</td>
</tr>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders $r = a \% b$
- the sequence stops when $r$ equals zero
The Greatest Common Divisor

Input: two positive numbers \(a > b > 0\).

Output: the greatest common divisor of \(a\) and \(b\).

An example: \(a = 60, b = 51\)

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(r = a%b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders \(r = a\%b\)
- the sequence stops when \(r\) equals zero
The Greatest Common Divisor

Input: two positive numbers $a > b > 0$.

Output: the greatest common divisor of $a$ and $b$.

An example: $a = 60, b = 51$

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$r = a % b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders $r = a \% b$
- the sequence stops when $r$ equals zero
The Greatest Common Divisor

**Input:** two positive numbers \( a > b > 0 \).

**Output:** the greatest common divisor of \( a \) and \( b \).

An example: \( a = 60, b = 51 \)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( r = a % b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders \( r = a \% b \)
- the sequence stops when \( r \) equals zero
The Greatest Common Divisor

**Input:** two positive numbers $a > b > 0$.

**Output:** the greatest common divisor of $a$ and $b$.

An example: $a = 60$, $b = 51$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
<td>$r = a%b$</td>
</tr>
<tr>
<td>60</td>
<td>51</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- the table shows the value of all variables for every stage in the loop
- we compute a sequence of remainders $r = a\%b$
- the sequence stops when $r$ equals zero
Euclid’s Algorithm described in words

To compute the greatest common divisor of $a$ and $b$ we perform the following steps in sequence:

1. Compute the remainder $r$ of $a$ divided by $b$.
2. If $r$ equals zero, then we print $b$ and stop.
3. Assign $b$ to $a$ and $r$ to $b$.
4. Execute the first step again.

Running `gcd.py` at the command prompt $*$:

```
$ python gcd.py
The Greatest Common Divisor
Give a : 60
Give b : 51
gcd(60,51) = 3
```
Euclid’s Algorithm
described in words

To compute the greatest common divisor of \( a \) and \( b \) we perform the following steps in sequence:

1. Compute the remainder \( r \) of \( a \) divided by \( b \).
2. If \( r \) equals zero, then we print \( b \) and stop.
3. Assign \( b \) to \( a \) and \( r \) to \( b \).
4. Execute the first step again.

Running `gcd.py` at the command prompt $:

```
$ python gcd.py
The Greatest Common Divisor
Give a : 60
Give b : 51
gcd(60,51) = 3
```
Euclid’s Algorithm described in words

To compute the greatest common divisor of $a$ and $b$ we perform the following steps in sequence:

1. Compute the remainder $r$ of $a$ divided by $b$.
2. If $r$ equals zero, then we print $b$ and stop.
3. Assign $b$ to $a$ and $r$ to $b$.
4. Execute the first step again.

Running `gcd.py` at the command prompt $\$$:

$ python gcd.py
The Greatest Common Divisor
Give a : 60
Give b : 51
gcd(60,51) = 3
Euclid’s Algorithm
described in words

To compute the greatest common divisor of $a$ and $b$ we perform the following steps in sequence:

1. Compute the remainder $r$ of $a$ divided by $b$.
2. If $r$ equals zero, then we print $b$ and stop.
3. Assign $b$ to $a$ and $r$ to $b$.
4. Execute the first step again.

Running `gcd.py` at the command prompt $\$:

```
$ python gcd.py
The Greatest Common Divisor
Give a : 60
Give b : 51
gcd(60,51) = 3
```
Euclid’s Algorithm
described in words

To compute the greatest common divisor of \(a\) and \(b\) we perform the following steps in sequence:

1. Compute the remainder \(r\) of \(a\) divided by \(b\).
2. If \(r\) equals zero, then we print \(b\) and stop.
3. Assign \(b\) to \(a\) and \(r\) to \(b\).
4. Execute the first step again.

Running \(gcd.py\) at the command prompt $:

$ python gcd.py

The Greatest Common Divisor
Give a : 60
Give b : 51
\(gcd(60,51) = 3\)
Euclid’s Algorithm

described in words

To compute the greatest common divisor of $a$ and $b$ we perform the following steps in sequence:

1. Compute the remainder $r$ of $a$ divided by $b$.
2. If $r$ equals zero, then we print $b$ and stop.
3. Assign $b$ to $a$ and $r$ to $b$.
4. Execute the first step again.

Running `gcd.py` at the command prompt $\$:$

```bash
$ python gcd.py
The Greatest Common Divisor
Give a : 60
Give b : 51
gcd(60,51) = 3
```
Euclid’s Algorithm
shown as a flowchart

```
a, b = input()
r = a % b
```

- If `r != 0`:
  - `r = a % b`
  - `False print b`
- If `r == 0`:
  - `a = b`
  - `b = r`
  - `r = a % b`
Euclid’s Algorithm
shown as a flowchart

```
a, b = input()
r = a % b
```

```
r != 0?
```

```
True
```

```
a = b
b = r
r = a % b
```

```
False
```

```
print b
```
Euclid’s Algorithm
shown as a flowchart

```python
a, b = input()
r = a % b

if r != 0:
    False print b
else:
    True a = b
    b = r
    r = a % b
```

Euclid’s Algorithm
shown as a flowchart
Euclid’s Algorithm
shown as a flowchart

```
a, b = input()
r = a % b
if r != 0:
    True
else:
    False
    print(b)
a = b
b = r
r = a % b
```

Diagram:
- `a, b = input()`
- `r = a % b`
- `r != 0?`
  - True: `a = b`
  - False: `print(b)`
  - `b = r`
  - `r = a % b`
Euclid’s Algorithm in Python
the script gcd.py

# L-11 MCS 260 Fri 19 Sep 2009  loop for gcd
#
# prints the greatest common divisor of two
# user given positive numbers
#
print 'The Greatest Common Divisor'
a = input('Give a : ')
b = input('Give b : ')
# save numbers for output later
s = 'gcd(%d,%d) = ' % (a,b)
r = a % b  # initialization
while r != 0:  # continue ?
    a = b  # body of loop
    b = r
    r = a % b
print s + '%d' % b  # show result
Euclid’s Algorithm in Python

the script gcd.py

```python
# L-11 MCS 260 Fri 19 Sep 2009 loop for gcd
#
# prints the greatest common divisor of two
# user given positive numbers
#
print 'The Greatest Common Divisor'
a = input('Give a : ')
b = input('Give b : ')
# save numbers for output later
s = 'gcd(%d,%d) = ' % (a,b)
r = a % b # initialization
while r != 0: # continue ?
    a = b # body of loop
    b = r
    r = a % b
print s + '%d' % b # show result
```
Euclid’s Algorithm in Python
the script gcd.py

# L-11 MCS 260 Fri 19 Sep 2009 loop for gcd
#
# prints the greatest common divisor of two
# user given positive numbers
#
print 'The Greatest Common Divisor'
a = input('Give a : ')
b = input('Give b : ')
# save numbers for output later
s = 'gcd(%d,%d) = ' % (a,b)
r = a % b # initialization
while r != 0: # continue ?
    a = b # body of loop
    b = r
    r = a % b
print s + '%d' % b # show result
Euclid’s Algorithm in Python
the script gcd.py

# L-11 MCS 260 Fri 19 Sep 2009 loop for gcd
# prints the greatest common divisor of two user given positive numbers
#
print 'The Greatest Common Divisor'
a = input('Give a : ')
b = input('Give b : ')
# save numbers for output later
s = 'gcd(%d,%d) = ' % (a,b)
r = a % b               # initialization
while r != 0:           # continue ?
    a = b               # body of loop
    b = r
    r = a % b

print s + '%d' % b     # show result
Euclid’s Algorithm in Python

the script gcd.py

# L-11 MCS 260 Fri 19 Sep 2009 loop for gcd
#
# prints the greatest common divisor of two
# user given positive numbers
#
print 'The Greatest Common Divisor'
a = input('Give a : ')
b = input('Give b : ')
# save numbers for output later
s = 'gcd(%d,%d) = ' % (a,b)
r = a % b # initialization
while r != 0: # continue ?
   a = b # body of loop
   b = r
   r = a % b
print s + ' %d' % b # show result
Summary + Assignments

In this lecture we covered more of

- section 1 in *Computer Science, an overview*
- chapter 4 & 5 of *Python Power!*

Assignments:

1. Using logical NOT, AND, OR gates construct a circuit that realizes an adder.

2. Modify the Python program gcd.py so that it prints out the values of all variables for each step.

3. The factorial of a positive numbers $n$ is $n! = n(n - 1)(n - 2) \cdots 1$. Give Python code which takes $n$ on input and has $n!$ as output. Make two versions: one with `while` and the other with `for`.

4. Use `>>` in a loop to determine how many bits it takes to represent any user given number.