Outline

MCS 260 Lecture 11
Introduction to Computer Science
Jan Verschelde, 5 February 2016

adder circuits while and for loops

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.
An adder takes on input b_1, b_2, *and* 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)

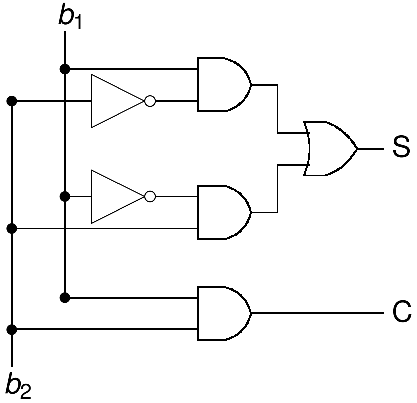
b_1	b_2	C	NOT b_1 AND b_2	b_1 AND NOT b_2	S
0	0	0	0	0	0
1	0	0	0	1	1
0	1	0	1	0	1
1	1	1	0	0	0

Does S look familiar? S = xor = *exclusive or*

Realization of a Half Adder

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

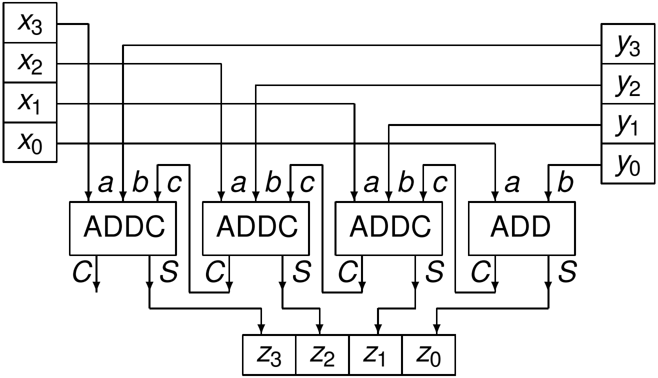
C = b_1 AND b_2
S = (NOT b_1 AND b_2) OR (b_1 AND NOT b_2)



exercise: realize with NAND & NOR

adder circuits
while and for loops

Adding Four Bits



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}:

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

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

operator	example	explanation
$\sim$	$\sim a$	not $a = -a - 1$
&	a & b	a and b
$\mid$	$a \mid b$	a or b
^	a ^ b	a xor b
$<<$	$a << b$	$a \times 2^b$
$>>$	$a >> b$	$a/2^b$

adder circuits while and for loops

Yearly Balance of an Investment

script `yieldbal.py` of lecture 2.5 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

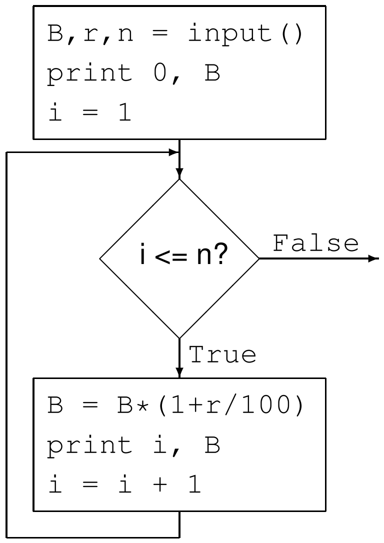
Running showbal.py at the command prompt $:

```
$ 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

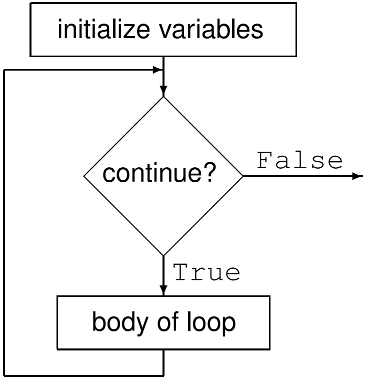
the flowchart for the program



the while loop

continue until condition becomes false

The flowchart of the while loop:



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.

the program showbal.py

```
"""
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 = float(input('Give amount to invest : '))
R = float(input('Give annual interest rate : '))
N = int(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

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:

```
>>> [x for x in 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`:

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

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

```
for i in range(1,n+1):
    B *= (1 + r/100)
```

showing the balance with for

```
"""
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 = int(input('Give amount to invest : '))
R = float(input('Give annual interest rate : '))
N = int(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

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

```
>>> (B, r)  = (1000, 3.14)
>>> L = [ B*(1+r/100)**i for i in range(5) ]
>>> s = ['%.2f' % x for x in L]
>>> s
['1000.00', '1031.40', '1063.79', '1097.19', '1131.64']
```

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

The Greatest Common Divisor

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

Output: the greatest common divisor of a and b.

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

a	b	$r = a\%b$
60	51	9
51	9	6
9	6	3
6	3	0

- 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:

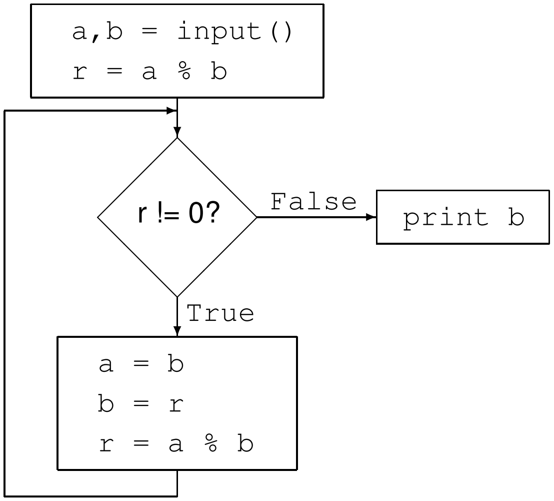
1. Compute the remainder *r* of *a* divided by *b*.
2. If *r* equals zero, then we print *b* and stop.
3. Assign *b* to *a* and *r* to *b*.
4. Execute the first step again.

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

shown as a flowchart

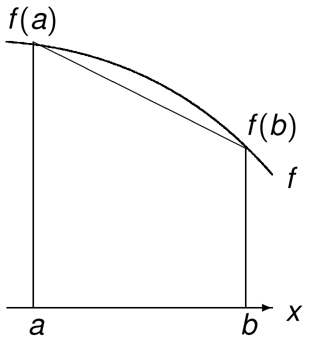


Euclid’s Algorithm in the script gcd.py

```
"""
Prints the greatest common divisor of two
user given positive numbers.
"""
print('The Greatest Common Divisor')
A = int(input('Give a : '))
B = int(input('Give b : '))
# save numbers for output later
OUTCOME = 'gcd(%d, %d) = ' % (A, B)
REST = A % B        # initialization
while REST != 0:
    (A, B) = (B, REST)
    REST = A % B
print(OUTCOME + '%d' % B)
```

adder circuits
while and for loops

the trapezoidal rule



$$\int_a^b f(x)dx \approx \frac{f(a)+f(b)}{2}(b-a)$$

an algorithm to approximate π

π is the area of the unit disk with boundary $x^2 + y^2 - 1 = 0$

$$\begin{aligned}\frac{\pi}{4} &= \int_0^1 \sqrt{1 - x^2}\, dx \\ &\approx \sum_{i=0}^{n-1} \left(\frac{\sqrt{1 - x_i^2} + \sqrt{1 - x_{i+1}^2}}{2} \right) (x_{i+1} - x_i)\end{aligned}$$

Subdividing $[0, 1]$ in n equal intervals: $x_i = i/n$.

translating a formula

$$\sum_{i=0}^{n-1} \left(\frac{\sqrt{1 - x_i^2} + \sqrt{1 - x_{i+1}^2}}{2} \right) (x_{i+1} - x_i)$$

```
from math import sqrt
N = int(input('give the number of samples : '))
S = 0
for i in range(0, N):
    x1 = float(i)/N;   fx1 = sqrt(1-x1*x1)
    x2 = float(i+1)/N; fx2 = sqrt(1-x2*x2)
    base = x2 - x1;    height = (fx1 + fx2)/2
    S = S + base*height
```

Although not optimal, we can verify its correctness.

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.
5. Draw a flowchart for the summation to approximate π.
6. Make the Python code to approximate π more efficient to save calls to $\sqrt{\ }$, without sacrificing accuracy.

Summary

In this lecture we covered *more of*

- section 1 in *Computer Science, an overview*,
- pages 54-55 of *Python Programming in Context*.