Extending Python

Factorization in Primes
- flowchart and Python code
- timing: Python versus C

Writing a C program
- one main program
- using a function

Extending Python
- add Python.h and wrappers
- compiling and installing

Exercises and Summary

MCS 275 Lecture 2
Programming Tools and File Management
Jan Verschelde, 16 January 2008

Factorization in Primes

example of lecture 1

Input/Output specification:

input	:	a natural number n
output	:	a string $n = p_1\, p_2\, \cdots\, p_k$, $k > 0$
		where $n = p_1 \times p_2 \times \cdots \times p_k$
		and every p_i is prime, $i = 1, 2 \ldots, k$.

Running at the command prompt $:

```
$ python facnums.py
give a natural number n : 121121
121121 = 7 11 11 11 13
$
```

Factorization in Primes

example of lecture 1

Input/Output specification:

input	:	a natural number n
output	:	a string $n = p_1\, p_2\, \cdots\, p_k$, $k > 0$
		where $n = p_1 \times p_2 \times \cdots \times p_k$
		and every p_i is prime, $i = 1, 2 \ldots, k$.

Running at the command prompt $:

```
$ python facnums.py
give a natural number n : 121121
121121 = 7 11 11 11 13
$
```

Writing to Strings

delayed printing

If n is a number, then

```
print '%d = ', n
```

is equivalent to

```
s = '%d = ' % n
print s
```

The `'%d'` is a format string and the `%` following the string is the conversion operator.

We can update the string s as `s = s + '%d' % n` and postpone the printing of s till the end of the program.

Writing to Strings

delayed printing

If n is a number, then

```
print '%d = ', n
```

is equivalent to

```
s = '%d = ' % n
print s
```

The `'%d'` is a format string and the `%` following the string is the conversion operator.

We can update the string s as `s = s + '%d' % n` and postpone the printing of s till the end of the program.

Writing to Strings

delayed printing

If n is a number, then

```
print '%d = ', n
```

is equivalent to

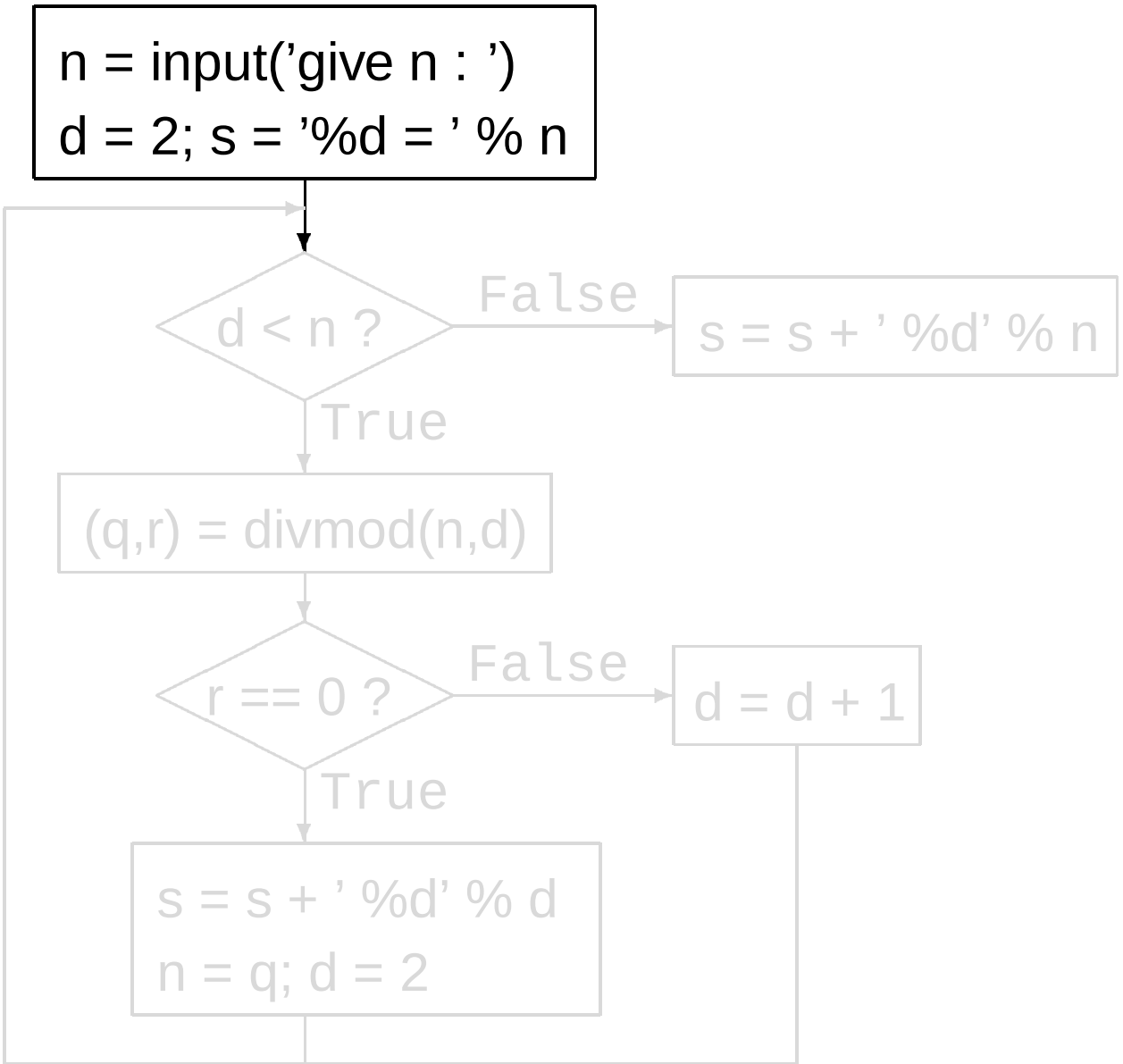
```
s = '%d = ' % n
print s
```

The `'%d'` is a format string and the `%` following the string is the conversion operator.

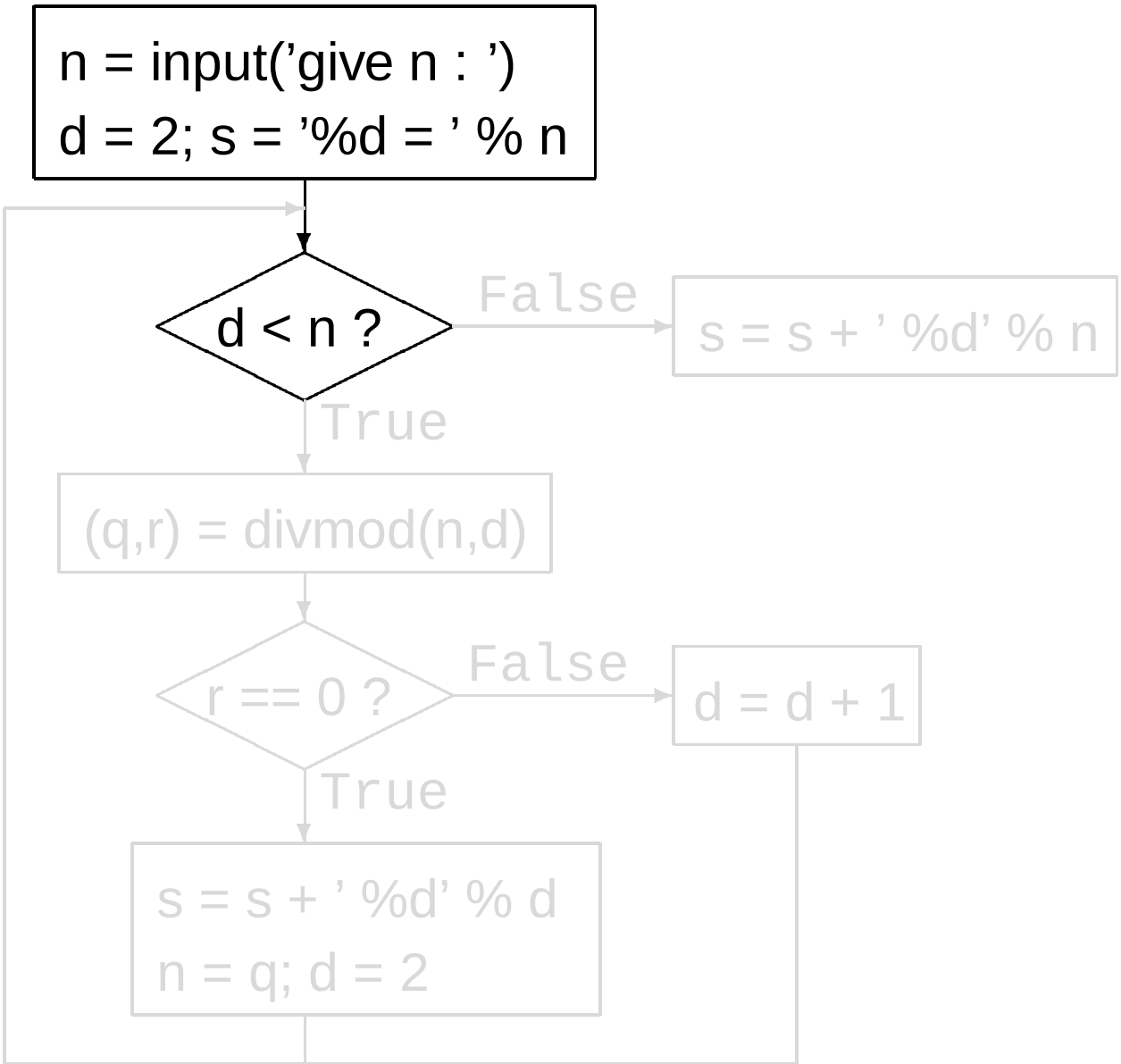
We can update the string `s` as `s = s + ' %d' % n` and postpone the printing of `s` till the end of the program.

Extending Python

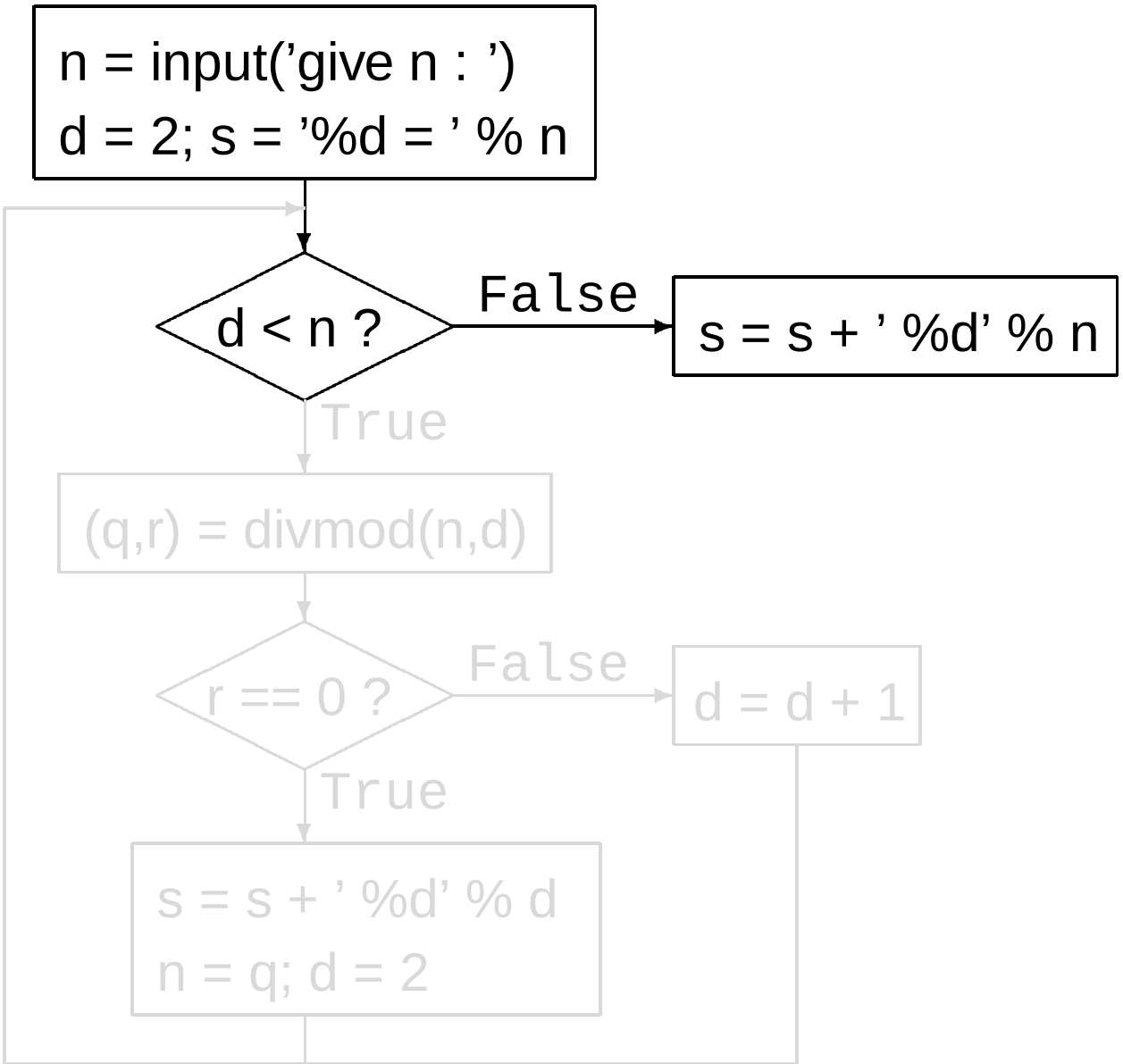
Flowchart for Factoring in Primes



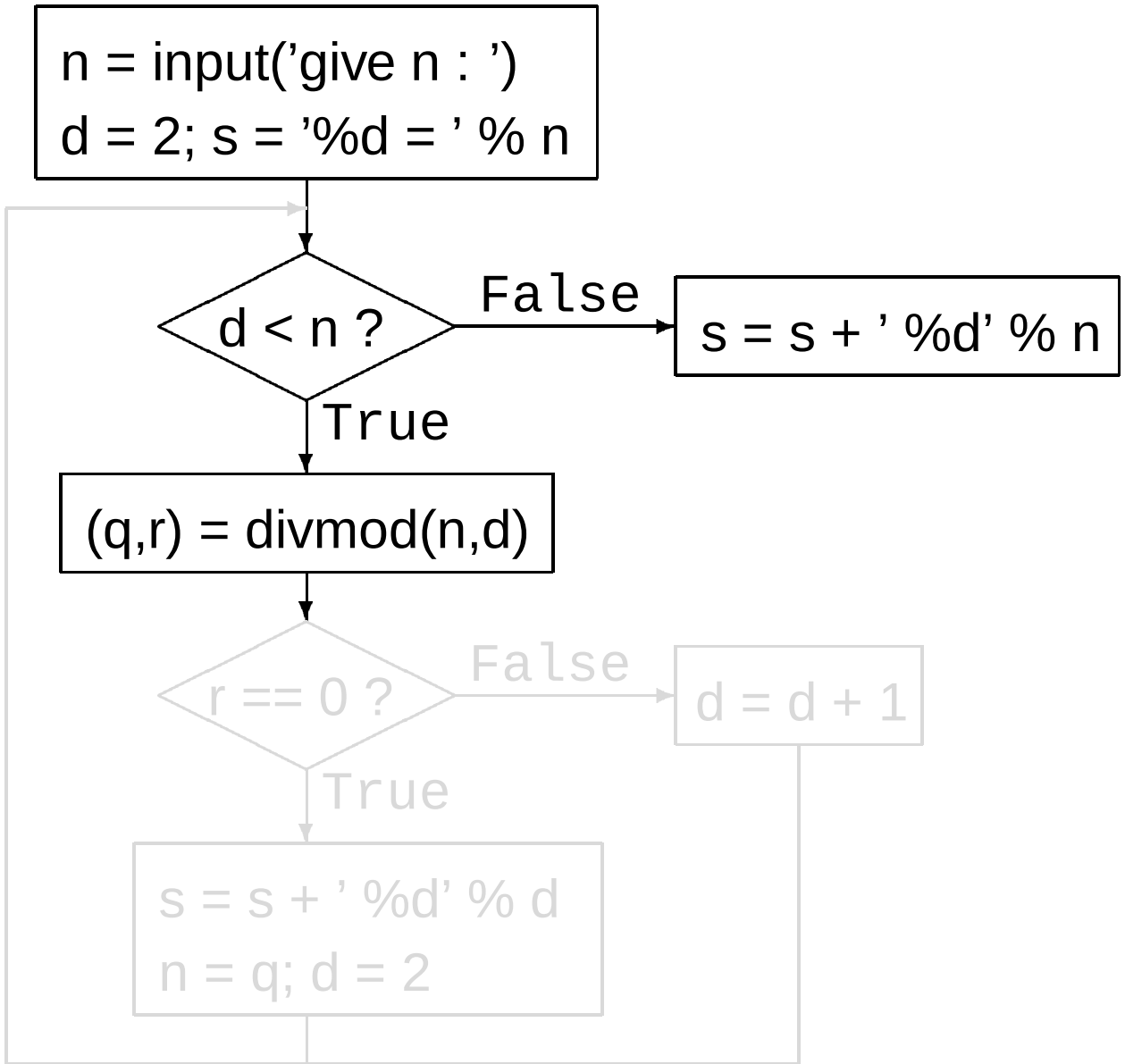
Flowchart for Factoring in Primes



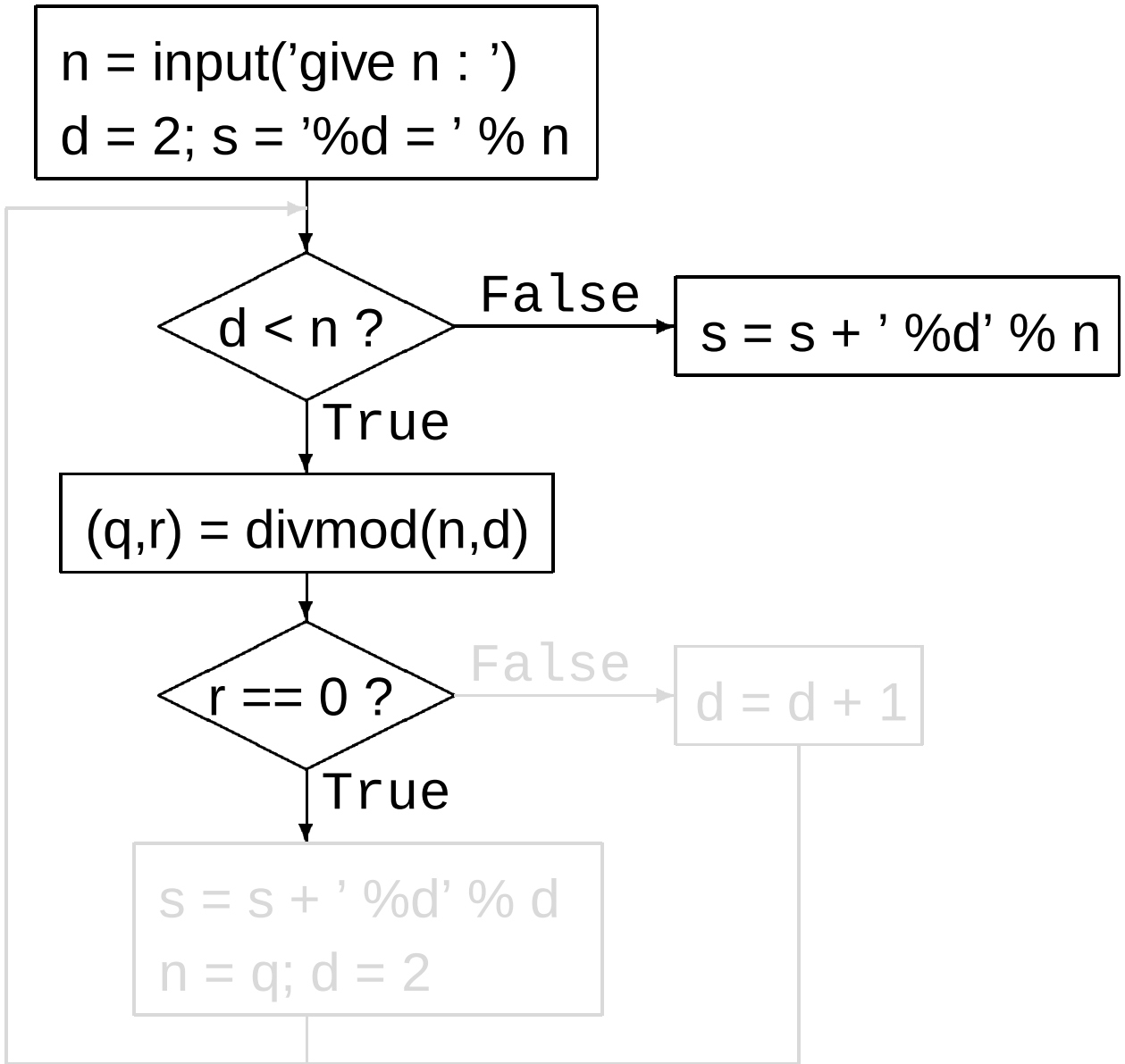
Flowchart for Factoring in Primes



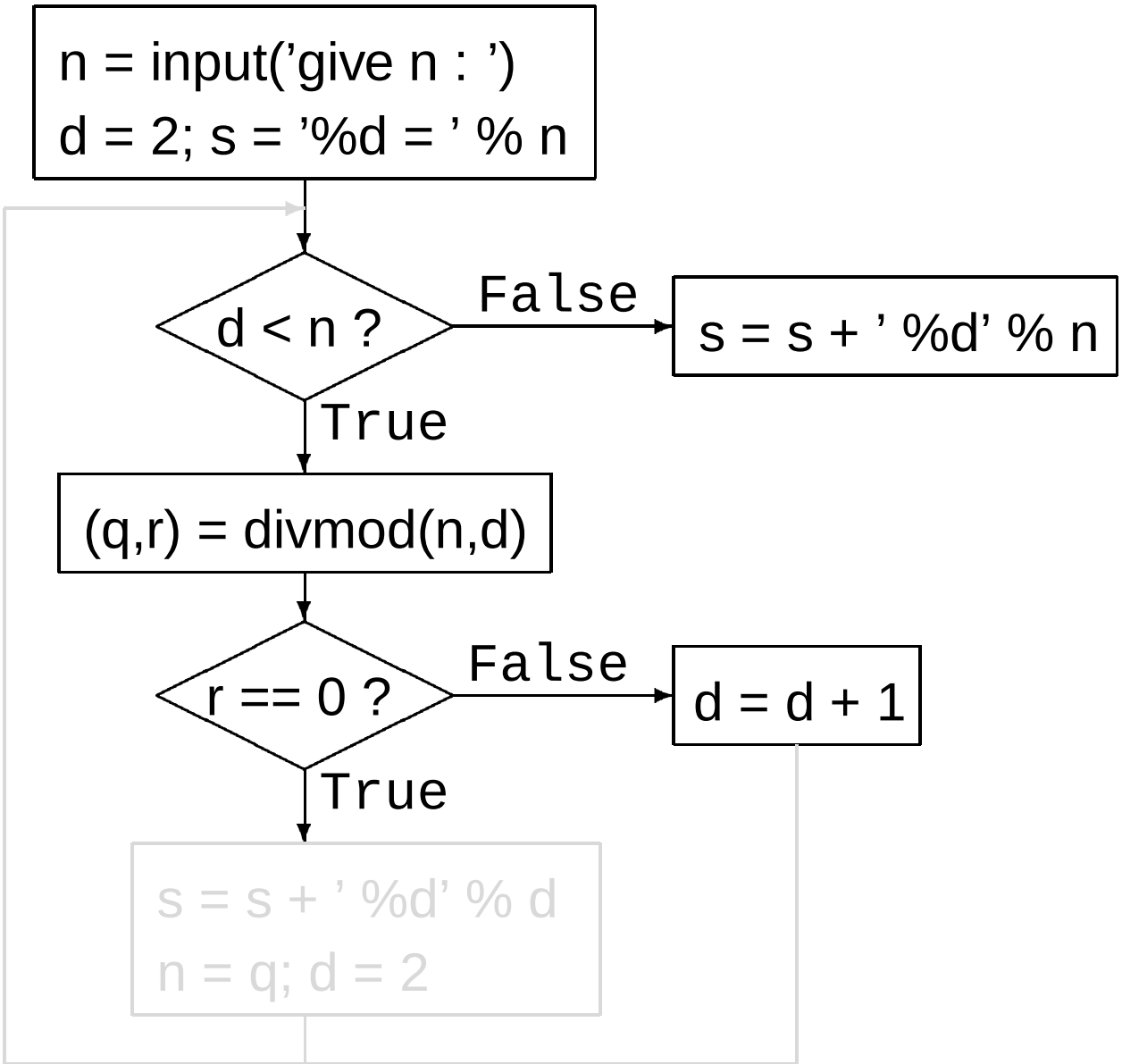
Flowchart for Factoring in Primes



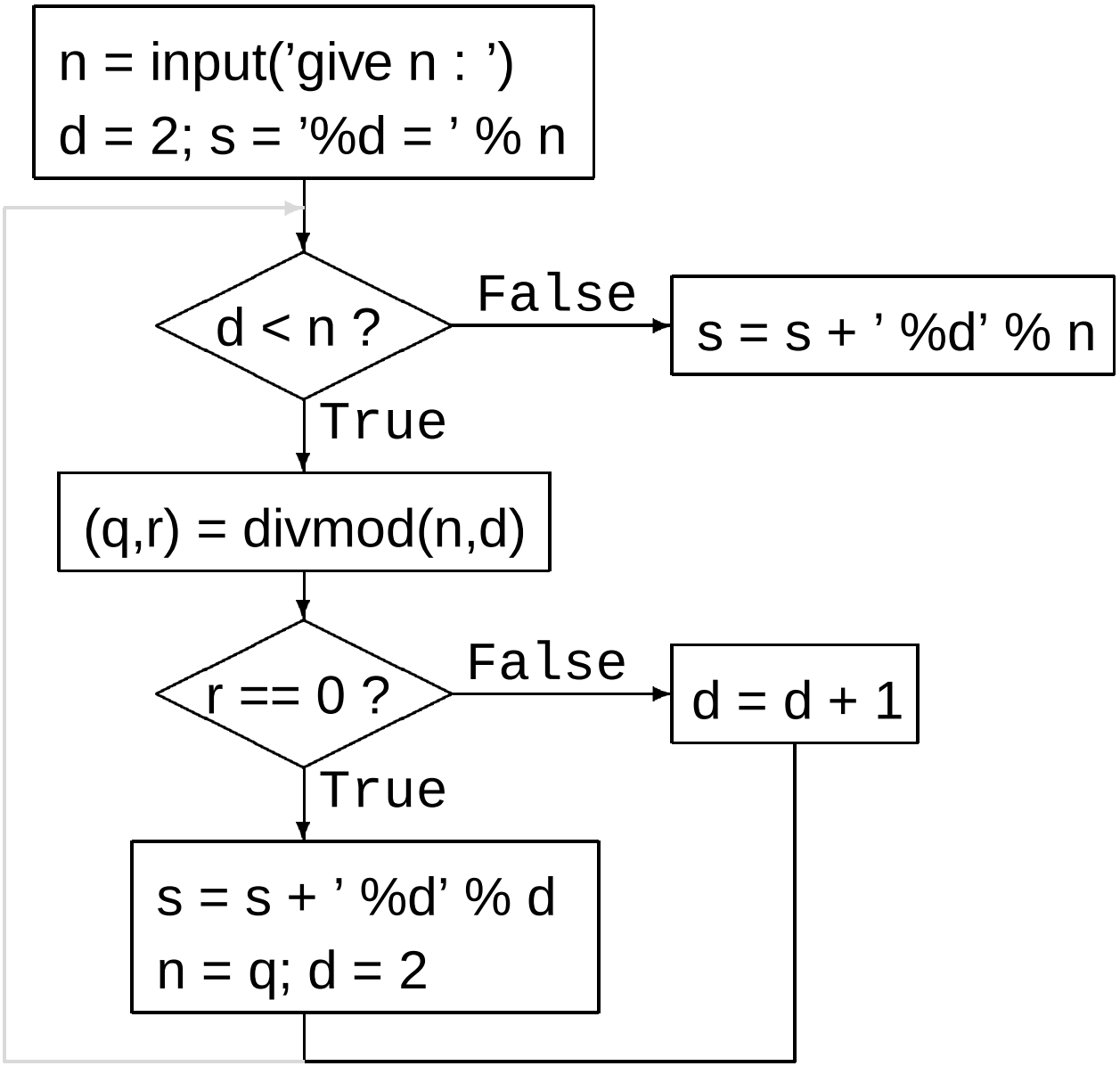
Flowchart for Factoring in Primes



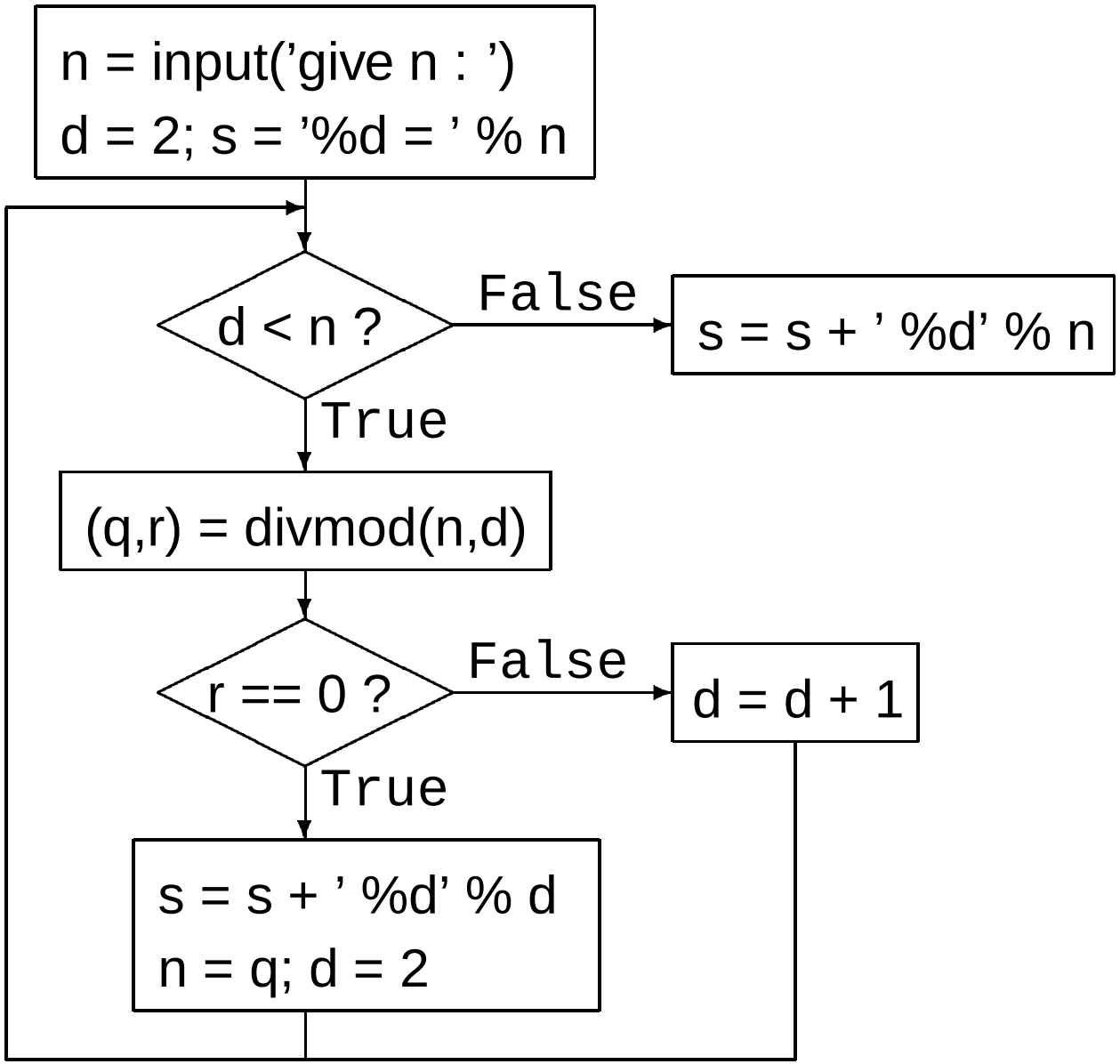
Flowchart for Factoring in Primes



Flowchart for Factoring in Primes



Flowchart for Factoring in Primes



the script facnums.py

```
# MCS 275 L-2 Wed 14 Jan 2008 : facnums.py
# factor a number into product of primes,
# writing the result into a string

n = input('give a natural number n : ')
d = 2; s = '%d =' % n
while(d < n):
    (q,r) = divmod(n,d)
    if(r == 0):
        s = s + ' %d' % d
        n = q; d = 2
    else:
        d = d + 1
s = s + ' %d' % n
print s
```

Extending Python

Timing Programs

measuring performance empirically

- easiest: use `time` command
- slow typers take long time, so redirect input
- if the file `input` contains `121121`:

```
$ time python facnums.py < input
give a natural number n : 121121 = 7 11 11 11 13

real    0m0.022s
user    0m0.011s
sys     0m0.011s
```

It takes 22 milliseconds total,
half of which is spent on the user program,
the other half is time the system spent.

Timing Programs

measuring performance empirically

- easiest: use `time` command
- slow typers take long time, so redirect input
- if the file `input` contains `121121`:

```
$ time python facnums.py < input
give a natural number n : 121121 = 7 11 11 11 13

real    0m0.022s
user    0m0.011s
sys     0m0.011s
```

It takes 22 milliseconds total,
half of which is spent on the user program,
the other half is time the system spent.

Timing Programs

measuring performance empirically

- easiest: use `time` command
- slow typers take long time, so redirect input
- if the file `input` contains `121121`:

```
$ time python facnums.py < input
give a natural number n : 121121 = 7 11 11 11 13

real    0m0.022s
user    0m0.011s
sys     0m0.011s
```

It takes 22 milliseconds total,
half of which is spent on the user program,
the other half is time the system spent.

Python versus C on 1000000007

```
$ time python facnums.py < input_prime
give a natural number n : 1000000007 = 1000000007

real    15m2.126s
user    15m1.454s
sys      0m0.585s
```

In 15 minutes, we wrote the C program `facnum0.c`,
we now compile it with `gcc`:

```
$ gcc -o /tmp/facnum0 facnum0.c
```

and run it on the same input:

```
$ time /tmp/facnum0 < input_prime
give a natural number n : 1000000007 = 1000000007

real    0m5.828s
user    0m5.816s
sys     0m0.012s
```

C is 150 times faster than Python!

Python versus C on 1000000007

```
$ time python facnums.py < input_prime
give a natural number n : 1000000007 = 1000000007

real    15m2.126s
user    15m1.454s
sys      0m0.585s
```

In 15 minutes, we wrote the C program `facnum0.c`, we now compile it with `gcc`:

```
$ gcc -o /tmp/facnum0 facnum0.c
```

and run it on the same input:

```
$ time /tmp/facnum0 < input_prime
give a natural number n : 1000000007 = 1000000007

real    0m5.828s
user    0m5.816s
sys     0m0.012s
```

C is 150 times faster than Python!

Python versus C on 1000000007

```
$ time python facnums.py < input_prime
give a natural number n : 1000000007 = 1000000007

real    15m2.126s
user    15m1.454s
sys      0m0.585s
```

In 15 minutes, we wrote the C program `facnum0.c`, we now compile it with `gcc`:

```
$ gcc -o /tmp/facnum0 facnum0.c
```

and run it on the same input:

```
$ time /tmp/facnum0 < input_prime
give a natural number n : 1000000007 = 1000000007

real    0m5.828s
user    0m5.816s
sys     0m0.012s
```

C is 150 times faster than Python!

Extending Python

Variables and I/O in C

```
/* MCS 275 L-2 Wed 16 Jan 2008 : facnum0.c */

#include <stdio.h>

int main ( void )
{
   int n,d,r;

   printf("give a natural number n : ");
   scanf("%d",&n);

   printf("%d =",n);
```

- static typing: n, d, r are of type int
- stdio.h contains printf and scanf
- &n is the address of the variable n
- code is in between { and }

Variables and I/O in C

```
/* MCS 275 L-2 Wed 16 Jan 2008 : facnum0.c */

#include <stdio.h>

int main ( void )
{
   int n,d,r;

   printf("give a natural number n : ");
   scanf("%d",&n);

   printf("%d =",n);
```

- static typing: n, d, r are of type `int`
- `stdio.h` contains `printf` and `scanf`
- `&n` is the address of the variable n
- code is in between { and }

Variables and I/O in C

```
/* MCS 275 L-2 Wed 16 Jan 2008 : facnum0.c */

#include <stdio.h>

int main ( void )
{
   int n,d,r;

   printf("give a natural number n : ");
   scanf("%d",&n);

   printf("%d =",n);
```

- static typing: n, d, r are of type `int`
- `stdio.h` contains `printf` and `scanf`
- &n is the address of the variable n
- code is in between { and }

Variables and I/O in C

```
/* MCS 275 L-2 Wed 16 Jan 2008 : facnum0.c */

#include <stdio.h>

int main ( void )
{
   int n,d,r;

   printf("give a natural number n : ");
   scanf("%d",&n);

   printf("%d =",n);
```

- static typing: n, d, r are of type `int`
- `stdio.h` contains `printf` and `scanf`
- &n is the address of the variable n
- code is in between { and }

Variables and I/O in C

```
/* MCS 275 L-2 Wed 16 Jan 2008 : facnum0.c */

#include <stdio.h>

int main ( void )
{
   int n,d,r;

   printf("give a natural number n : ");
   scanf("%d",&n);

   printf("%d =",n);
```

- static typing: n, d, r are of type `int`
- `stdio.h` contains `printf` and `scanf`
- &n is the address of the variable n
- code is in between { and }

Control Structures

program continued, with Python translation

```
    d = 2;                          | d = 2
    while(d < n)                    | while(d < n):
    {                               |
       r = n % d;                   |    r = n % d
       if(r == 0)                   |    if(r == 0):
       {                            |
          printf(" %d",d);          |       print ' %d' % d
          n = n/d;                  |       n = n/d
          d = 2;                    |       d = 2
       }                            |
       else                         |    else:
          d = d + 1;                |       d = d + 1
    }                               |
    printf(" %d\n",n);              | print ' %d\n' % n

    return 0;
}
```

Printing to Strings in C

The prime factors computed by the C program will be passed to Python in a string.

Therefore, the C code will print to a string.

```
printf("%d = ",n)
```

is equivalent to

```
char s[80]; /* 80 characters for result */
sprintf(s,"%d =",n); /* print to s */
```

Adding a factor d to the result s:

```
char f[10]; /* 10 characters for factor */
sprintf(f," %d",d);
strcat(s,f); /* string concatenation */
```

Printing to Strings in C

The prime factors computed by the C program will be passed to Python in a string.
Therefore, the C code will print to a string.

```
printf("%d = ",n)
```

is equivalent to

```
char s[80]; /* 80 characters for result */
sprintf(s,"%d =",n); /* print to s */
```

Adding a factor d to the result s:

```
char f[10]; /* 10 characters for factor */
sprintf(f," %d",d);
strcat(s,f); /* string concatenation */
```

Printing to Strings in C

The prime factors computed by the C program will be passed to Python in a string.
Therefore, the C code will print to a string.

```
printf("%d = ",n)
```

is equivalent to

```
char s[80]; /* 80 characters for result */
sprintf(s,"%d =",n); /* print to s */
```

Adding a factor d to the result s:

```
char f[10]; /* 10 characters for factor */
sprintf(f," %d",d);
strcat(s,f); /* string concatenation */
```

Printing to Strings in C

The prime factors computed by the C program will be passed to Python in a string.
Therefore, the C code will print to a string.

```
printf("%d = ",n)
```

is equivalent to

```
char s[80]; /* 80 characters for result */
sprintf(s,"%d =",n); /* print to s */
```

Adding a factor d to the result s:

```
char f[10]; /* 10 characters for factor */
sprintf(f," %d",d);
strcat(s,f); /* string concatenation */
```

Extending Python

using a function factor

```
#include <stdio.h>
#include <string.h>

int factor ( int n, char *s );
/* writes the prime factors of n
 * into the string s */

int main ( void )
{
   int n;
   char s[80]; /* string for result */

   printf("give a natural number n : ");
   scanf("%d",&n);
   factor(n,s);
   printf(s); /* print result */

   return 0;
}
```

using a function factor

```
#include <stdio.h>
#include <string.h>

int factor ( int n, char *s );
/* writes the prime factors of n
 * into the string s */

int main ( void )
{
   int n;
   char s[80]; /* string for result */

   printf("give a natural number n : ");
   scanf("%d",&n);
   factor(n,s);
   printf(s); /* print result */

   return 0;
}
```

function definition

```
int factor ( int n, char *s ){
   int d,r;
   char f[10]; /* string for factor */
   sprintf(s,"%d =",n);
   d = 2;
   while(d < n){
      r = n % d;
      if(r == 0){
         sprintf(f," %d",d);
         strcat(s,f);
         n = n/d; d = 2;
      }
      else
         d = d + 1;
   }
   sprintf(f," %d\n",n);
   strcat(s,f);
   return 0;
}
```

Extending Python

The goal is to use the C code from a Python session.

The C code will in a *module* `NumFac`.

```
>>> import NumFac
>>> dir(NumFac)
['__doc__', '__file__', '__name__', 'factor',
'test']
>>> from NumFac import test, factor
>>> test()
give a natural number n : 121121
121121 = 7 11 11 11 13
>>> s = factor(121121,'')
>>> s
'121121 = 7 11 11 11 13\n'
>>> s.split(' ')
['121121', '=', '7', '11', '11', '11', '13\n']
```

After the `split`, the factors are in a list.

Extending Python

The goal is to use the C code from a Python session.

The C code will in a *module* `NumFac`.

```
>>> import NumFac
>>> dir(NumFac)
['__doc__', '__file__', '__name__', 'factor',
'test']
>>> from NumFac import test, factor
>>> test()
give a natural number n : 121121
121121 = 7 11 11 11 13
>>> s = factor(121121,'')
>>> s
'121121 = 7 11 11 11 13\n'
>>> s.split(' ')
['121121', '=', '7', '11', '11', '11', '13\n']
```

After the `split`, the factors are in a list.

Extending Python

The goal is to use the C code from a Python session.

The C code will in a *module* `NumFac`.

```
>>> import NumFac
>>> dir(NumFac)
['__doc__', '__file__', '__name__', 'factor',
'test']
>>> from NumFac import test, factor
>>> test()
give a natural number n : 121121
121121 = 7 11 11 11 13
>>> s = factor(121121,'')
>>> s
'121121 = 7 11 11 11 13\n'
>>> s.split(' ')
['121121', '=', '7', '11', '11', '11', '13\n']
```

After the `split`, the factors are in a list.

Extending Python

The goal is to use the C code from a Python session.

The C code will in a *module* `NumFac`.

```
>>> import NumFac
>>> dir(NumFac)
['__doc__', '__file__', '__name__', 'factor',
'test']
>>> from NumFac import test, factor
>>> test()
give a natural number n : 121121
121121 = 7 11 11 11 13
>>> s = factor(121121,'')
>>> s
'121121 = 7 11 11 11 13\n'
>>> s.split(' ')
['121121', '=', '7', '11', '11', '11', '13\n']
```

After the `split`, the factors are in a list.

Extending Python

The goal is to use the C code from a Python session.

The C code will in a *module* `NumFac`.

```
>>> import NumFac
>>> dir(NumFac)
['__doc__', '__file__', '__name__', 'factor',
'test']
>>> from NumFac import test, factor
>>> test()
give a natural number n : 121121
121121 = 7 11 11 11 13
>>> s = factor(121121,'')
>>> s
'121121 = 7 11 11 11 13\n'
>>> s.split(' ')
['121121', '=', '7', '11', '11', '11', '13\n']
```

After the `split`, the factors are in a list.

Extending Python

Python.h and test()

append to the file with the C code:

```
#include "/Library/Frameworks/Python.framework
/Headers/Python.h"
```

It is good practice to keep the main program as an interactive test program:

```
static PyObject *NumFac_test
  ( PyObject *self, PyObject *args )
{
   test();
   return (PyObject*)Py_BuildValue("");
}
```

Python.h and test()

append to the file with the C code:

```
#include "/Library/Frameworks/Python.framework
/Headers/Python.h"
```

It is good practice to keep the main program as an interactive test program:

```
static PyObject *NumFac_test
  ( PyObject *self, PyObject *args )
{
   test();
   return (PyObject*)Py_BuildValue("");
}
```

Wrapping the Function factor

```
static PyObject *NumFac_factor
   ( PyObject *self, PyObject *args )
{
   PyObject *result;
   int n;
   char s[80];

   if(!PyArg_ParseTuple(args,"is",&n,&s)) return NULL;

   factor(n,s);

   result = (PyObject*)Py_BuildValue("s",s);

   return result;
}
```

Wrapping the Function factor

```
static PyObject *NumFac_factor
   ( PyObject *self, PyObject *args )
{
   PyObject *result;
   int n;
   char s[80];

   if(!PyArg_ParseTuple(args,"is",&n,&s)) return NULL;

   factor(n,s);

   result = (PyObject*)Py_BuildValue("s",s);

   return result;
}
```

Wrapping the Function factor

```
static PyObject *NumFac_factor
   ( PyObject *self, PyObject *args )
{
   PyObject *result;
   int n;
   char s[80];

   if(!PyArg_ParseTuple(args,"is",&n,&s)) return NULL;

   factor(n,s);

   result = (PyObject*)Py_BuildValue("s",s);

   return result;
}
```

Wrapping the Function factor

```
static PyObject *NumFac_factor
   ( PyObject *self, PyObject *args )
{
   PyObject *result;
   int n;
   char s[80];

   if(!PyArg_ParseTuple(args,"is",&n,&s)) return NULL;

   factor(n,s);

   result = (PyObject*)Py_BuildValue("s",s);

   return result;
}
```

Wrapping the Function factor

```
static PyObject *NumFac_factor
   ( PyObject *self, PyObject *args )
{
   PyObject *result;
   int n;
   char s[80];

   if(!PyArg_ParseTuple(args,"is",&n,&s)) return NULL;

   factor(n,s);

   result = (PyObject*)Py_BuildValue("s",s);

   return result;
}
```

Registration Table and Initialization

The registration table contains the documentation strings for the functions the module exports:

```
static PyMethodDef NumFacMethods[] =
{
   { "factor" , NumFac_factor , METH_VARARGS,
     "factor a natural number into primes" } ,
   { "test" , NumFac_test , METH_VARARGS,
     "interactive test on prime factoring" } ,
   { NULL , NULL, 0, NULL } ,
};

void initNumFac()   /* initialization function */
{
   Py_InitModule("NumFac",NumFacMethods);
}
```

Registration Table and Initialization

The registration table contains the documentation strings for the functions the module exports:

```
static PyMethodDef NumFacMethods[] =
{
   { "factor" , NumFac_factor , METH_VARARGS,
     "factor a natural number into primes" } ,
   { "test" , NumFac_test , METH_VARARGS,
     "interactive test on prime factoring" } ,
   { NULL , NULL, 0, NULL } ,
};

void initNumFac()   /* initialization function */
{
   Py_InitModule("NumFac",NumFacMethods);
}
```

Extending Python

Compiling and Installing

To check whether the extension module is still valid C code, we compile as `gcc -c numfac.c`.

To create a *shareable object* (file with extension `.so`), we create setup.py:

```
from distutils.core import setup, Extension

# for using numfac.c :

MOD = 'NumFac'
setup(name=MOD,ext_modules=[Extension(MOD,\
  sources=['numfac.c'])])
```

and do `python setup.py build`
Instead of `python setup.py install`,
do `cp build/lib*/NumFac.so .`
(copy the shareable object to the current directory).

Compiling and Installing

To check whether the extension module is still valid C code, we compile as `gcc -c numfac.c`.
To create a *shareable object* (file with extension `.so`), we create setup.py:

```
from distutils.core import setup, Extension

# for using numfac.c :

MOD = 'NumFac'
setup(name=MOD,ext_modules=[Extension(MOD,\
  sources=['numfac.c'])])
```

and do `python setup.py build`
Instead of `python setup.py install`,
do `cp build/lib*/NumFac.so .`
(copy the shareable object to the current directory).

Compiling and Installing

To check whether the extension module is still valid C code, we compile as `gcc -c numfac.c`.
To create a *shareable object* (file with extension `.so`), we create setup.py:

```
from distutils.core import setup, Extension

# for using numfac.c :

MOD = 'NumFac'
setup(name=MOD,ext_modules=[Extension(MOD,\
  sources=['numfac.c'])])
```

and do `python setup.py build`
Instead of `python setup.py install`,
do `cp build/lib*/NumFac.so .`
(copy the shareable object to the current directory).

Exercises

1. Take the script `enumdivs.py` from Lecture 1 and write a corresponding C program: `enumdivs.c`. Write the factors to screen instead of into a list.
2. Modify the `enumdivs.c` from the previous exercise so that the factors are written into a string.
3. Use the modified `enumdivs.c` from the previous exercise to build a Python extension EnumDivs.
4. Take a Python function to compute the greatest common divisor of two natural numbers a and b:

```
s = 'gcd(%d,%d) = ' % (a,b)
r = a % b
while r <> 0:
   a = b
   b = r
   r = a % b
print s + '%d' % b
```

   Create a corresponding C program and use it to build a Python extension.

Summary

In this lecture we covered

1. some elements of C programs, see the start of Chapter 4 in *The Art & Craft of Computing*;
2. an introduction to extending Python, for more: `http://docs.python.org/ext/ext.html`.

On Friday we will start `numpy` and `scipy`,
go to `www.scipy.org` to download the code.