Extending Python

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

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

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

4. Exercises and Summary

MCS 275 Lecture 2
Programming Tools and File Management
Jan Verschelde, 11 January 2017
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

\[
\text{print(‘%d = ’, n)
}
\]

is equivalent to

\[
s = ‘%d = ’ % n
\]

\[
\text{print(s)
}
\]

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

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

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

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

4. Exercises and Summary
Flowchart for Factoring in Primes

```
n = input('give n : ')
d = 2; s = ’%d = ’ % n

\[d < n?\]  \hspace{1cm} False \hspace{1cm} \rightarrow s = s + ’%d’ % n

True \hspace{1cm} \rightarrow (q,r) = \text{divmod}(n,d)

\[r == 0?\]  \hspace{1cm} False \hspace{1cm} \rightarrow d = d + 1

True \hspace{1cm} \rightarrow s = s + ’%d’ % d
\hspace{1cm} \hspace{1cm} \rightarrow n = q; d = 2
```
the script \texttt{facnums.py}

```python
"""
factor a number into product of primes, writing the result to a string
"""

\begin{verbatim}
n = eval(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)
\end{verbatim}
```
Extending Python

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

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

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

4. Exercises and Summary
Timing Programs – measuring performance

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

```
$ time python3 facnums.py < input
give a natural number n : 121121 = 7 11 11 11 11 13
```

real 0m0.039s
user 0m0.027s
sys 0m0.009s

It takes 39 milliseconds total,
of which 27 milliseconds are spent on the user program,
there are 9 milliseconds of time spent by the system.
Python versus C on 1000000007
On MacBook Pro, Early 2015, 3.1 GHz Intel Core i7.

$ time python3 facnums.py < input_prime
give a natural number n : 1000000007 = 1000000007

real  8m13.203s
user  7m58.106s
sys   0m 3.166s

The C program facnum0.c is compiled 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  0m2.848s
user  0m2.786s
sys   0m0.015s

C is 173 times faster than Python!
Extending Python

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

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

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

4. Exercises and Summary
/* L-2 MCS 275 : 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 }

\[ d = 2; \]
\[ \text{while}(d < n) \]
\[ \{ \]
\[ \quad r = n \% d; \]
\[ \quad \text{if}(r == 0) \]
\[ \quad \quad \{ \]
\[ \quad \quad \quad \text{printf}(" %d", d); \]
\[ \quad \quad \quad n = n/d; \]
\[ \quad \quad \quad d = 2; \]
\[ \quad \quad \} \]
\[ \quad \text{else} \]
\[ \quad \quad d = d + 1; \]
\[ \} \]
\[ \text{printf}(" %d\n", n); \]
\[ \]
\[ \text{return} \ 0; \]
Printing to Strings in C

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

Therefore, the C code will print to a string.

\[
\text{printf("%d = ",n)}
\]

is equivalent to

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

Adding a factor \(d\) to the result \(s\):

\[
\text{char f[10]; /* 10 characters for factor */}
\text{sprintf(f," %d",d);}
\text{strcat(s,f); /* string concatenation */}
\]
Extending Python

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

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

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

4. Exercises and Summary
using a function factor – refactoring code

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

int factor ( int n, char *s );
/* writes the prime factors of n to 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",s); /* print result */

    return 0;
}
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 be in a module `NumFac`.

```python
>>> 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.
1. Factorization in Primes
   - flowchart and Python code
   - timing: Python versus C

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

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

4. Exercises and Summary
Python.h and test()

Append to the file with the C code:

```c
#include "Python.h"
```

In the makefile, we need to define the location of Python.h:

```makefile
PYTHON3=/Library/Frameworks/Python.framework/Versions/3.6/include/python3.6m
```

```bash
NumFac.so:
gcc -dynamiclib -undefined dynamic_lookup \\
-o NumFac.so -I$(PYTHON3) numfac.c
```

Typing `make NumFac.so` at the command prompt in a Terminal window will then make the shared object `NumFac.so`.

Note: in the makefile, the `gcc` is preceded by one tab.
It is good practice to keep the main program as an interactive test program:

```c
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;
}
The registration table contains the documentation strings for the functions the module exports:

```c
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 } ,
};
```

The documentation strings show when the user does `help()`. 
Module Initialization

At the end of the file, we initialize the module:

```c
static struct PyModuleDef NumFacModule = {
    PyModuleDef_HEAD_INIT,
    "NumFac",
    "prime number factorization",
    -1,
    NumFacMethods
};

PyMODINIT_FUNC
PyInit_NumFac()
{
    return PyModule_Create(&NumFacModule);
}
```
Extending Python

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

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

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

4. Exercises and Summary
Compiling and Installing

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

To create a \textit{shareable object} (file with extension .so), we create a script \texttt{setup.py}:

\begin{verbatim}
from distutils.core import setup, Extension
# for using numfac.c :
MOD = 'NumFac'
setup(name=MOD, ext_modules=[Extension(MOD, 
    sources=['numfac.c'])])
\end{verbatim}

and do \texttt{python3 setup.py build}

Instead of \texttt{python3 setup.py install},
do \texttt{cp build/lib*/NumFac.so}.
(copy the shareable object to the current directory).
In this lecture we covered

1. some elements of C programs, see the Chapter 6 in *computer science, an overview*;
2. an introduction to extending Python, for more: https://docs.python.org/3/extending/index.html.
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`:

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

   Build a Python extension from equivalent C code.