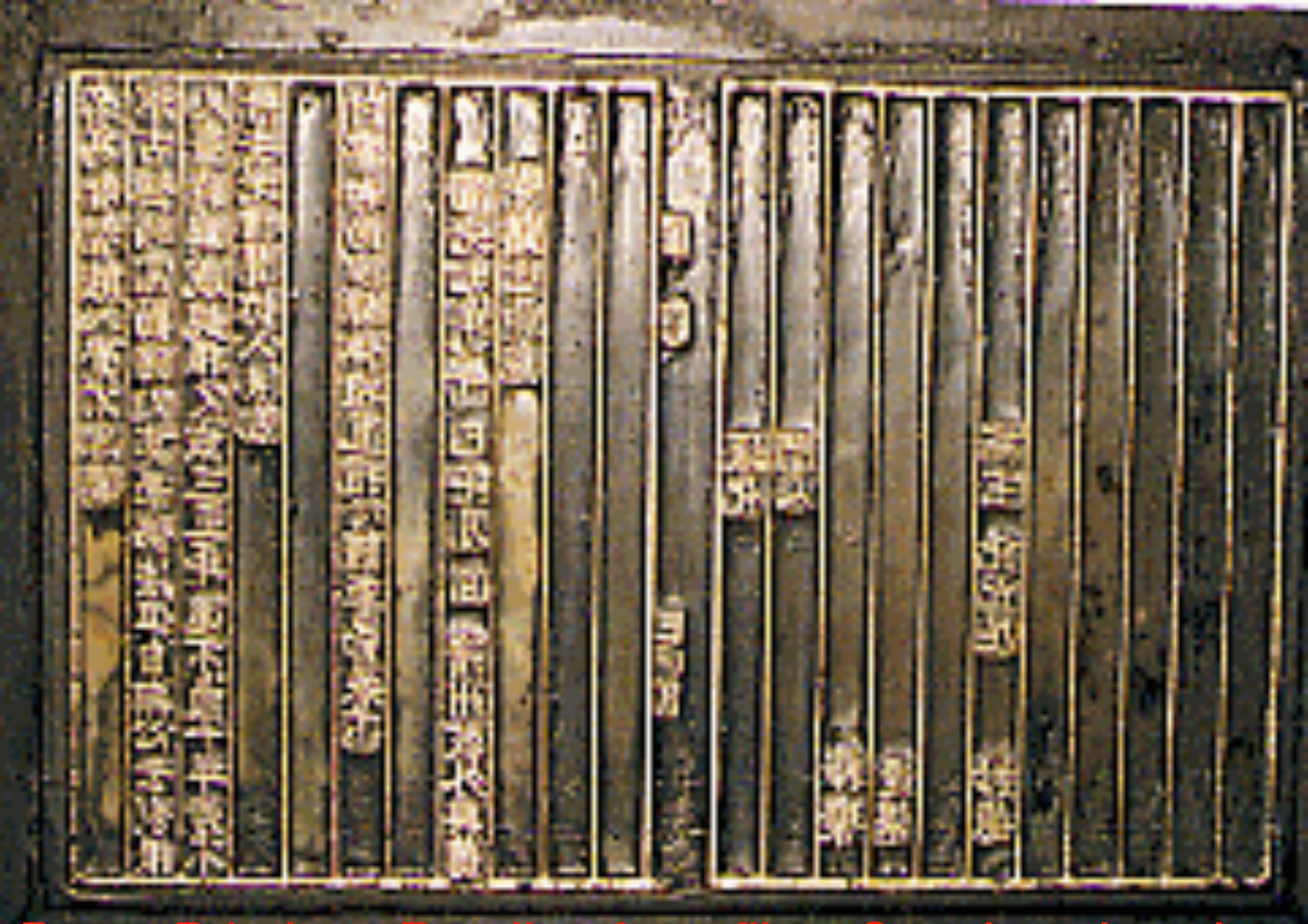

Pretty Printing, Reading from files, Creating nice plots

4.4. **break** and **continue** Statements, and **else** Clauses on Loops¶

The **break** statement, like in C, breaks out of the smallest enclosing **for** or **while** loop.

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the **for** loop, **not** the **if** statement.)

The `continue` statement, also borrowed from C, continues with the next iteration of the loop:

```
>>>
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

```
>>> for x in range(1,11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Basic usage of the **`str.format()`** method looks like this:

```
>>>
>>> print 'We are the {} who say "{}!"'.format('knights',
'Ni')
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the **`str.format()`** method. A number in the brackets refers to the position of the object passed into the **`str.format()`** method.

```
>>>
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

If keyword arguments are used in the **`str.format()`** method, their values are referred to by using the name of the argument.

```
>>>
>>> print 'This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>>
>>> print 'The story of {0}, {1}, and
{other}.'.format('Bill', 'Manfred',
...
other='Georg')
The story of Bill, Manfred, and Georg.
```

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>>
```

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

On Windows, `'b'` appended to the mode opens the file in binary mode, so there are also modes like `'rb'`, `'wb'`, and `'r+b'`. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in `JPEG` or `EXE` files. Be very careful to use binary mode when reading and writing such files. On Unix, it doesn't hurt to append a `'b'` to the mode, so you can use it platform-independently for all binary files.

To read a file’s contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it’s your problem if the file is twice as large as your machine’s memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`""`).

```
>>>
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is efficient, fast, and leads to simple code:

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

`f.write(string)` writes the contents of *string* to the file, returning `None`.

```
>>> f.write('This is a test\n')
```

To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

It is good practice to use the **`with`** keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent **`try`**-**`finally`** blocks:

```
>>>
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True

>>> with open('workfile', 'r') as f:
...     for line in f:
...         read_data.append(line)
>>> f.closed
True
```

exercises: reading data and graphing