Announcements

This is CS 180!
C++ versus Python
High level versus low level
Interpreter versus compiler
Dynamic versus static typing
Why learn C++?
- faster
- ubiquitous
- understand low level details
- control
A comparison

Python

def gcd(u, v):
    # we will use Euclid's algorithm
    # for computing the GCD
    while v != 0:
        r = u % v    # compute remainder
        u = v
        v = r
    return u

if __name__ == '__main__':
a = int(input('First value: '))
b = int(input('Second value: '))
print('gcd:', gcd(a, b))

C++:

#include <iostream>
using namespace std;

int gcd(int u, int v) {
    /* We will use Euclid's algorithm
    for computing the GCD */
    int r;
    while (v != 0) {
        r = u % v;    // compute remainder
        u = v;
        v = r;
    }    // end while
    return u;
}    // end gcd function

int main() {
    int a, b;
    cout << "First value: ";
    cin >> a;
    cout << "Second value: ";
    cin >> b;
    cout << "gcd: " << gcd(a, b) << endl;
    return 0;
}
**White space is irrelevant!**

```c
int gcd(int u, int v) { int r; while (v != 0) { r = u % v; u = v; v = r; } return u; }
```

Python uses **returns** & **indentation** to separate commands & loops.

*Parentheses ( ) are important!*  

Please continue to indent.
Executing code

In Python, we could save the code as gcd.py and then type "python gcd.py" to run it.

In C++:
- Save as gcd.cpp
- Type "g++ -o gcd gcd.cpp"
- Type "./gcd"

## Data Types

<table>
<thead>
<tr>
<th>C++ Type</th>
<th>Description</th>
<th>Literals</th>
<th>Python analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>logical value</td>
<td>true, false</td>
<td>bool</td>
</tr>
<tr>
<td>short</td>
<td>integer (often 16 bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>integer (often 32 bits)</td>
<td>39</td>
<td>int</td>
</tr>
<tr>
<td>long</td>
<td>integer (often 32 or 64 bits)</td>
<td>39L</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>integer (arbitrary-precision)</td>
<td></td>
<td>long</td>
</tr>
<tr>
<td>float</td>
<td>floating-point (often 32 bits)</td>
<td>3.14f</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>floating-point (often 64 bits)</td>
<td>3.14</td>
<td>float</td>
</tr>
<tr>
<td>char</td>
<td>single character</td>
<td>'a'</td>
<td></td>
</tr>
<tr>
<td>string*</td>
<td>character sequence</td>
<td>&quot;Hello&quot;</td>
<td>str</td>
</tr>
</tbody>
</table>
Data Types (cont.)

Each integer type can also be unsigned.

Instead of ranging from $-(2^{b-1})$ to $(2^{b-1}-1)$, goes from 0 to $2^b-1$. 
Char versus String

```java
char a;
a = 'a';
a = 'h';
```

```java
String word;
word = "CS 180";
```

Strings are not automatically included! They are standard in most libraries, but need to import that library.
### Strings

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.size()</code></td>
<td>Either form returns the number of characters in string <code>s</code>.</td>
</tr>
<tr>
<td><code>s.length()</code></td>
<td></td>
</tr>
<tr>
<td><code>s.empty()</code></td>
<td>Returns <code>true</code> if <code>s</code> is an empty string, <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>s[index]</code></td>
<td>Returns the character of string <code>s</code> at the given index</td>
</tr>
<tr>
<td><code>s.at(index)</code></td>
<td>(unpredictable when <code>index</code> is out of range).</td>
</tr>
<tr>
<td><code>s == t</code></td>
<td>Returns <code>true</code> if strings <code>s</code> and <code>t</code> have same contents, <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>s &lt; t</code></td>
<td>Returns <code>true</code> if <code>s</code> is lexicographical less than <code>t</code>, <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>s.compare(t)</code></td>
<td>Returns a negative value if string <code>s</code> is lexicographical less than string <code>t</code>, zero if equal, and a positive value if <code>s</code> is greater than <code>t</code>.</td>
</tr>
<tr>
<td><code>s.find(pattern)</code></td>
<td>Returns the least index (greater than or equal to index <code>pos</code>, if given), at which <code>pattern</code> begins; returns <code>string::npos</code> if not found.</td>
</tr>
<tr>
<td><code>s.find(pattern, pos)</code></td>
<td></td>
</tr>
<tr>
<td><code>s.find_first_of(charset)</code></td>
<td>Returns the least index (greater than or equal to index <code>pos</code>, if given), at which a character of the indicated string <code>charset</code> is found; returns <code>string::npos</code> if not found.</td>
</tr>
<tr>
<td><code>s.find_last_of(charset)</code></td>
<td></td>
</tr>
<tr>
<td><code>s.substr(start)</code></td>
<td>Returns a substring from index <code>start</code> through the end.</td>
</tr>
<tr>
<td><code>s.substr(start, num)</code></td>
<td>Returns the substring from index <code>start</code>, continuing <code>num</code> characters.</td>
</tr>
<tr>
<td><code>s.c_str()</code></td>
<td>Returns a C-style character array representing the same sequence of characters as <code>s</code>.</td>
</tr>
</tbody>
</table>
Mutable versus immutable

**Def.** mutable

**Def.** immutable
C++ - Maximum Flexibility

In C++, everything is mutable!

```cpp
string word;
word = "Hello";
word[0] = "J";
```
Arrays

Python has lists, tuples, etc.

C++ only has arrays.
- Size is fixed
- Type is fixed (and homogeneous)

Ex: int numbers [10].
numbers [0] = 56;
numbers [9] = 11;

Numbers [10] = 5;
Creating variables (cont.)

Allowed:

```cpp
int daysInMonth[] = {31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31};
```

Error:

```cpp
int daysInMonth[];
```

Allowed:

```cpp
char greeting[] = "Hello";
```
Creating variables - a few examples

int number;
int a, b;  // creates 2 integers.

int age(40);  
int age (cur Year - birth Year);  
int age(40), zipcode(63116);  
string greeting("Hello");
Forcing things to be immutable:

In some situations, there will be data that we want to be fixed.

To do this, use `const`:

```
const float gravity(9.8);
```