Ceng 111 – Fall 2015
Week 7a

Basic data

**Credit:** Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.
Concurrent Paradigm

- Programming using multiple CPUs concurrently.
- The task is to assign the overall flow & data to individual CPUs.
- With the bottleneck in CPU power, this paradigm is going to be the trend in the future.
Event-Driven Paradigm

- A program is composed of events and what to do in case of events.
- The task is to decompose a problem into a set of events and the corresponding functionalities that will be executed in case of events.
- Suitable for Graphical User Interface design.
The hyperspace of languages

Figure 1.4: The hyperspace of programming (only 3 axes displayed)
Zoo of Programming Languages

- Around 700 programming languages!

- But, why do we not have a programming language that serves all paradigms/purposes/requirements?
Choosing a PL

Ex: Moving soil with a shovel and a grader

Previously on CENG 111!
Factors that affect choosing a PL

- Domain & Technical Nature of the Problem

  a) Finding the pixels of an image with RGB value of [202,130,180] with a tolerance of 5.4% in intensity.

  b) A proof system for planar geometry problems.

  c) A computer game platform which will be used as a whole or in parts and may get extended even rewritten by programmers at various levels.

  d) Payroll printing.
Factors that affect choosing a PL

- Personal taste and preference
- Circumstance-imposed constraints
  - e.g., time limit.
- Current trend
Current trend

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
How are languages implemented

COMPILATIVE APPROACH

SOURCE CODE

int alice = 1;
int bob = 456;
int carol;
main(void)
{
carol = alice*bob;
printf("%d", carol);
}

COMPILER

OBJECT CODE

OBJECT CODE

LINKER

EXECUTABLE CODE

PREVIOUSLY ON CENG 111

CEng 111

S. Kalkan & G. Ucoluk & A. Cosar

2015

LIBRARY
How are languages implemented

```c
int alice = 123;
int bob = 456;
int carol;
main(void)
{
    carol = alice*bob;
    printf("%d", carol);
}
```

**COMPILATIVE APPROACH**

Previously on CENG 111

2015 S. Kalkan & G. Ucoluk & A. Cosar - CEng 111
How are languages implemented

INTERPRETIVE APPROACH

USER TYPES

int alice = 123;

INTERPRETER REPLIES

OK. alice is 123.

USER TYPES

int bob = 456;

INTERPRETER REPLIES

OK. bob is 456.

USER TYPES

print bob * alice;

INTERPRETER REPLIES

56088

Previously on CENG 111
How is a program written?

- Modular & Functional Breakdown

- For example:
  - User interface module
  - Database module
  - Control module
Testing

Top-down Testing

http://sce.uhcl.edu/whiteta/sdp/subSystemIntegrationTesting.html
Testing

Bottom-up Testing

Bottom-Up Integration Testing

SystemX.1
  - SystemX.1.1
  - SystemX.1.2

SystemX.2
  - SystemX.2.1
  - SystemX.2.2

SystemX
  - SystemX.1
  - SystemX.2

http://sce.uhcl.edu/whteta/sdp/subSystemIntegrationTesting.html
Black-box Testing

Testing

Requirements

Input

The System

Output

Previously on CENG 111!
Bugs, Errors

Syntax Errors

Area = 3.1415 * R * R

Area = 3.1415 x R x R

Run-time Errors

```python
>>> def SqrtDelta(a,b,c):
    >>> return sqrt(b*b - 4*a*c)
    >>>
>>> print SqrtDelta(1,3,1)
2.2360679774997898
>>> print SqrtDelta(1,1,1)
ValueError: math domain error
```
Bugs, Errors

- **Logical Errors**

\[ root_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]

>>> root1 = (-b + sqrt(b*b - 4*a*c)) / 2*a

- **Design Errors**

\[ x^3 + ax^2 + bx + c = 0 \]

\[ root_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]
An interpretive/scripting PL that:

- Longs for code readability
- Ease of use, clear syntax
- Wide range of applications, libraries, tools

Multiple Paradigms:

- Functional
- Imperative
- Object-oriented
Started at the end of 1980s.

- V2.0 was released in 2000
  - With a big change in development perspective: Community-based
  - Major changes in the facilities.

- V3.0 was released in 2008
  - Backward-incompatible
  - Some of its features are put into v2.6 and v2.7.
Where does the name come from?

One goal of Python: “fun to use”

- The origin of the name is the comedy group “Monty Python”
- This is reflected in sample codes that are written in Python by the original developers.
Previously on CENG 111!

```
skalkan@divan:~$ python
Python 2.5.2 (r252:60911, Jan 24 2010, 17:44:40)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```
Today

- Basic data

- Reminder:
  - Tentative midterm date: 9 December at 17:40.
Design of a solution
What is data?

• **Data**: Information to be processed to solve a problem.
• Identify the data for the following example problems:
  
  - Find all wheat growing areas in a terrestrial satellite image.
  
  - Given the homework, lab and examination grades of a class, calculate the letter grades.
  
  - Alter the amplitude of a sound recording for various frequencies.
  
  - Extrapolate China’s population for the year 2040 based on the change in the population growth rate up to this time.
  
  - Compute the launch date and the trajectory for a space probe so that it will pass by the outermost planets in the closest proximity.
  
  - Compute the layout of the internals of a CPU so that the total wiring distance is minimized.
  
  - Find the cheapest flight plan from A to B, for given intervals for arrival and departure dates.
  
  - Simulate a war between two land forces, given (i) the attack and the defense plans, (ii) the inventories and (iii) other attributes of both forces.
What is data?

- CPU can only understand two types of data:
  - Integers,
  - Floating points.

- The following are not directly understandable by a CPU:
  - Characters (‘a’, ‘A’, ‘2’, …)
  - Strings (“apple”, “banana”, …)
  - Complex Numbers
  - Matrices
  - Vectors

- But, programming languages can implement these data types.
Basic Data Types

- Integers
  - Full support from the CPU
  - Fast processing

- Floating Points
  - Support from the CPU with some precision loss
  - Slower compared to integer processing due to “interpretation”
Problems due to precision loss

- What has precision loss is misleading: 0.9 has precision loss whereas 0.9375 does not!
  - \((0.5 + 0.25 + 0.125 + 0.0625)\)
  - \((\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16}) = 0.1111\)
- Imprecise representations are round-offs!
- If you have a lot of round-offs, you might be in trouble!
Problems due to precision loss

- Since the floating point representation is based on shifting the bits in the mantissa, the following are not equivalent in a PC:
  - $1.0023 - 1.0567$
    - Result: $-0.054400000000000004$
  - $1000.0023 - 1000.0567$
    - Result: $-0.054399999999986903$

- $\pi = 3.1415926535897931\ldots$
  - $\sin(\pi)$ should be zero
  - But it is not: $1.2246467991473532 \times 10^{-16}$
Problems due to precision loss

- **Associativity in mathematics:**
  - \((a+b) + c = a + (b + c)\)
  - This does not hold in floating-point arithmetic:

  \[
  \text{set } a = 1234.567, \quad b = 45.67834 \quad \text{and } \quad c = 0.0004:
  \]
  
  \((a + b) + c \text{ will produce } 1280.2457399999998,\)
  \(a + (b + c) \text{ will produce } 1280.2457400000001.\)
So what to do with precision loss, then?

- If you can transform the problem to the integer domain, do so.
  - Refrain from using floating points as much as you can.
- Use the most precise type of floating point of the high level language you are using.
  - Use only less precision floating points if you are short in memory.
So what to do with precision loss, then?

- If you have two addends that are magnitude-wise incomparable, you are likely to lose the contribution of the smaller one. That will yield unexpected results when you repeat the addition in a computational loop where the looping is so much that the accumulation of the smaller is expected to become significant. It will not.

- The contrary happens too. Slight inaccuracies accumulate in loops to significant magnitudes and yield non-sense values.

- You better use well known, decent floating point libraries instead of coding floating point algorithms by yourself.
Integers in Python

- Python provides `int` type.

```python
>>> type(3)
<type 'int'>
>>> type(3+4)
<type 'int'>
>>> 
```

- For big integers, Python has the `long` type:

```python
>>> type(3L)
<type 'long'>
>>> type(3L+4L)
<type 'long'>
>>> 
```

`int` is limited by the hardware whereas `long` type is unlimited in Python.
Floating Points in Python

- Python provides the float type.

```python
>>> type(3.4)
<type 'float'>

>>> 3.4 + 4.3
7.7

>>> 3.4 / 4.3
0.79069767441860461
```
Simple Operations with Numerical Values in Python

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Binary</td>
<td>Addition of two operands</td>
</tr>
<tr>
<td>-</td>
<td>Binary</td>
<td>Subtraction of two operands</td>
</tr>
<tr>
<td>-</td>
<td>Unary</td>
<td>Negated value of the operand</td>
</tr>
<tr>
<td>*</td>
<td>Binary</td>
<td>Multiplication of two operands</td>
</tr>
<tr>
<td>/</td>
<td>Binary</td>
<td>Division of two operands</td>
</tr>
<tr>
<td>**</td>
<td>Binary</td>
<td>Exponentiation of two operands</td>
</tr>
</tbody>
</table>

- **abs**(*x*): Absolute value of *x*.
- **round**(*Float*): Rounded value of float.
- **int**(*Number*), **long**(*Number*), **float**(*Number*): Conversion between numerical values of different types.
We need characters to represent textual data:

Computer uses the ASCII table for converting characters to binary numbers:

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>01000001</td>
</tr>
<tr>
<td>B</td>
<td>01000010</td>
</tr>
<tr>
<td>C</td>
<td>01000011</td>
</tr>
<tr>
<td>D</td>
<td>01000100</td>
</tr>
<tr>
<td>E</td>
<td>01000101</td>
</tr>
<tr>
<td>F</td>
<td>01000110</td>
</tr>
<tr>
<td>G</td>
<td>01000111</td>
</tr>
<tr>
<td>H</td>
<td>01001000</td>
</tr>
<tr>
<td>I</td>
<td>01001001</td>
</tr>
<tr>
<td>J</td>
<td>01001010</td>
</tr>
<tr>
<td>K</td>
<td>01001011</td>
</tr>
<tr>
<td>L</td>
<td>01001100</td>
</tr>
<tr>
<td>M</td>
<td>01001101</td>
</tr>
<tr>
<td>N</td>
<td>01001110</td>
</tr>
<tr>
<td>O</td>
<td>01001111</td>
</tr>
<tr>
<td>P</td>
<td>01010000</td>
</tr>
<tr>
<td>Q</td>
<td>01010001</td>
</tr>
<tr>
<td>R</td>
<td>01010010</td>
</tr>
<tr>
<td>S</td>
<td>01010011</td>
</tr>
<tr>
<td>T</td>
<td>01010100</td>
</tr>
<tr>
<td>U</td>
<td>01010101</td>
</tr>
<tr>
<td>V</td>
<td>01010110</td>
</tr>
<tr>
<td>W</td>
<td>01010111</td>
</tr>
<tr>
<td>X</td>
<td>01011000</td>
</tr>
<tr>
<td>Y</td>
<td>01011001</td>
</tr>
<tr>
<td>Z</td>
<td>01011010</td>
</tr>
<tr>
<td>_</td>
<td>01011011</td>
</tr>
<tr>
<td>`</td>
<td>01011100</td>
</tr>
<tr>
<td>a</td>
<td>01100001</td>
</tr>
<tr>
<td>b</td>
<td>01100010</td>
</tr>
<tr>
<td>c</td>
<td>01100011</td>
</tr>
<tr>
<td>d</td>
<td>01100100</td>
</tr>
<tr>
<td>e</td>
<td>01100101</td>
</tr>
<tr>
<td>f</td>
<td>01100110</td>
</tr>
<tr>
<td>g</td>
<td>01100111</td>
</tr>
<tr>
<td>h</td>
<td>01101000</td>
</tr>
<tr>
<td>i</td>
<td>01101001</td>
</tr>
<tr>
<td>j</td>
<td>01101010</td>
</tr>
<tr>
<td>k</td>
<td>01101011</td>
</tr>
<tr>
<td>l</td>
<td>01101100</td>
</tr>
<tr>
<td>m</td>
<td>01101101</td>
</tr>
<tr>
<td>n</td>
<td>01101110</td>
</tr>
<tr>
<td>o</td>
<td>01101111</td>
</tr>
<tr>
<td>p</td>
<td>01110000</td>
</tr>
<tr>
<td>q</td>
<td>01110001</td>
</tr>
<tr>
<td>r</td>
<td>01110010</td>
</tr>
<tr>
<td>s</td>
<td>01110011</td>
</tr>
<tr>
<td>t</td>
<td>01110100</td>
</tr>
<tr>
<td>u</td>
<td>01110101</td>
</tr>
<tr>
<td>v</td>
<td>01110110</td>
</tr>
<tr>
<td>w</td>
<td>01110111</td>
</tr>
<tr>
<td>x</td>
<td>01111000</td>
</tr>
<tr>
<td>y</td>
<td>01111001</td>
</tr>
<tr>
<td>z</td>
<td>01111010</td>
</tr>
<tr>
<td>{</td>
<td>01111100</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>01111110</td>
</tr>
<tr>
<td>~</td>
<td>01111111</td>
</tr>
</tbody>
</table>
Characters in Python

- Python does not have a separate data type for characters!
- However, one character strings can be treated like characters:
  - `ord(One_Char_String)`: returns the ASCII value of the character in `One_Char_String`.
    - Ex: `ord(“A”)` returns 65.
  - `chr(ASCII_value)`: returns the character in a string that has the ASCII value `ASCII_value`:
    - Ex: `chr(66)` returns “B”
Boolean Values

The CPU often needs to compare numbers, or data:

- $3 > 4$
- $125 = \frac{1000}{8}$
- $3 \leq \frac{12345.34545}{12324356.0}$

We have the truth values for representing the answers to such comparisons:

- If correct: TRUE, True, true, T, 1
- If not correct: FALSE, False, false, F, 0
Boolean Values in Python

- Python provides the `bool` data type for boolean values.
- `bool` data type can take True or False as values.

```python
>>> 3 > 4
False
>>> type(3 > 4)
<type 'bool'>
```

```python
>>> True == bool(2)
False
>>> type(True == bool(2))
<type 'bool'>
```

**not** operator:
- not True
- not 4 > 3

Try this:
- ```python
  >>> True == bool(2)
  False
  >>> True == 1
  True
  >>> False == 0
  True
  >>> False == 1
  False
  ```