Ceng 111 – Fall 2015
Week 7a

Container 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.
Design of a solution

Previously on CENG 111!
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: $-0.054399999999986903$

- $\pi = 3.1415926535897931...$
  - $\sin(\pi)$ 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:

  \[
  \begin{aligned}
  a &= 1234.567, \\
  b &= 45.67834, \\
  c &= 0.0004
  \end{aligned}
  \]

  \((a + b) + c\) will produce 1280.2457399999998,
  \(a + (b + c)\) 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.
Python provides \texttt{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>
<tr>
<td></td>
<td></td>
<td>(Ex: $x**y = x^y$)</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>SYN</th>
<th>ETB</th>
<th>CAN</th>
<th>EM</th>
<th>SUB</th>
<th>ESC</th>
<th>FS</th>
<th>GS</th>
<th>RS</th>
<th>US</th>
<th>SPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010110</td>
<td>00010111</td>
<td>00011000</td>
<td>00011001</td>
<td>00011010</td>
<td>00011011</td>
<td>00011100</td>
<td>00011101</td>
<td>00011110</td>
<td>00010000</td>
<td>00010001</td>
</tr>
<tr>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>01000010</td>
<td>01000011</td>
<td>01000100</td>
<td>01000101</td>
<td>01000110</td>
<td>01000111</td>
<td>01001000</td>
<td>01001001</td>
<td>01001010</td>
<td>01001011</td>
<td>01001100</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>(</td>
<td>\</td>
<td>)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>n</td>
</tr>
<tr>
<td>01011000</td>
<td>01011001</td>
<td>01011010</td>
<td>01011011</td>
<td>01011100</td>
<td>01011101</td>
<td>01011110</td>
<td>01011111</td>
<td>01000000</td>
<td>01000001</td>
<td>01000010</td>
</tr>
<tr>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>01001111</td>
<td>01010000</td>
<td>01010001</td>
<td>01010010</td>
<td>01010011</td>
<td>01010100</td>
<td>01010101</td>
<td>01010110</td>
<td>01010111</td>
<td>01011000</td>
<td>01011001</td>
</tr>
<tr>
<td>z</td>
<td>{</td>
<td>}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DEL</td>
</tr>
<tr>
<td>01111100</td>
<td>01111101</td>
<td>01111110</td>
<td>01111111</td>
<td>01111112</td>
<td>01111113</td>
<td>01111114</td>
<td>01111115</td>
<td>01111116</td>
<td>01111117</td>
<td>01111118</td>
</tr>
</tbody>
</table>

A portion of the ASCII 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:

- $3 > 4$
- $125 \neq 1000/8$
- $3 \leq 12345.34545/12324356.0$

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.

Try this:

```python
>>> True == bool(2)
False
>>> True == 1
True
>>> False == 0
True
>>> False == 1
False
```

`not` operator:

```python
not True
not 4 > 3
```
Today

- Container/structured data
  - Approaches for storing structured data
  - Strings
  - Tuples
  - Lists

- Reminder:
  - Tentative midterm date: 9 December at 17:40.
Structured Data

- If you have lots and lots of one type of data (for example, the ages of all the people in Turkey):
  - You can store them into memory consecutively (supported by most PLs)
    - This is called arrays.
  - Easy to access an element. Nth element:
    - <Starting-address>+ (N-1)*<Word Width>
    - Ex: 2nd element is at 128 + (2-1) * 4 = 132
Structured Data

- What if you have to make a lot of deletions and insertions in the middle of an array?
- Then, you have to store your data in blocks/units such that each unit has the starting address of the next unit/block.
Containers

Containers of basic data:
- Strings
- Tuples
- Lists
Strings

- Sequence of characters:
  - Ex: “Book”, “Programming”, “Python”

- How can they be represented?
  1. Put a set of characters one after the other and end them with a non-character value.
  2. At the beginning of the characters, specify how many characters follow.

- Both have advantages and disadvantages.
Strings in Python

Python provides the `str` data type for strings:

```python
>>> "Hello?"
'Hello?'
>>> type("Hello?")
<type 'str'>
```

**Simplest operation with a string:**

```python
>>> len("Hello?")
6
```
Strings in Python

- Accessing elements of a string
  "Hello?"[0] → 1\textsuperscript{st} character (i.e., "H")
  "Hello?"[4] → 5\textsuperscript{th} character

- Indexing starts at 0!!!

- What is the last element then?
  - "Hello?"[len("Hello?") - 1]

- Negative indexing possible:
  - Last element: "Hello?"[-1] → "?"

- In general:
  - String[start:end:step]
  - Ex: "Hello?"[0:4:2] → "Hl"
  - Ex: "Hello?"[2:4] → "ll"
Creating Strings in Python

1. Enclosing a set of characters between quotes:
   - “ali”, “veli”, “deli”, ...

2. Using the `str()` function:
   - `str(4.5) → "4.5"

3. Using the `raw_input()` function:

```python
>>> a = raw_input("--> ")
--> Do as I say
>>> a
'Do as I say'
>>> type(a)
<type 'str'>
```
Tuples

- Tuple: ordered set of data:
  - (1, 2, 3)
  - ("a", "b", "c")

- May be heterogeneous:
Tuples in Python

Tuples in Python: collection of data enclosed in parentheses, separated by comma.

Accessing elements of a tuple (like strings):

- Positive Indexing: `(1, 2, 3, 4, "a")[2]` returns 3.
- Negative Indexing: `(1, 2, 3, 4, "a")[-1]` returns 'a'.
- Ranged Indexing, i.e., `[start:end:step]`: `(1, 2, 3, 4, "a")[0:4:2]` leads to `(1, 3)`. 
Creating Tuples in Python

1. Enclosing data within parentheses:
   - Ex: (1, “a”, “cde”, 23)

2. Using the tuple() function:
   - Ex: tuple(“ABC”) → (‘A’, ‘B’, ‘C’)

3. Using the input() function:

   ```python
   >>> a = input("Give me a tuple:")
   Give me a tuple:(1, 2, 3)
   >>> a
   (1, 2, 3)
   >>> type(a)
   <type 'tuple'>
   ```
Lists

- Similar to tuples.

- Difference:
  - Tuples are immutable (i.e., not changeable) whereas lists are mutable.
Lists in Python

- Lists in Python: collection of data enclosed in brackets, separated by comma.

- Accessing elements of a list (like strings & tuples):
  - Positive Indexing: \([1, 2, 3, 4, "a"]\)[2] returns 3.
  - Negative Indexing: \([1, 2, 3, 4, "a"]\)[−1] returns ‘a’.
  - Ranged Indexing, i.e., \([start:end:step]\): \([1, 2, 3, 4, "a"]\)[0:4:2] leads to [1, 3].
Creating Lists in Python

1. Enclosing data within brackets:
   - Ex: [1, "a", "cde", 23]

2. Using the list() function:
   - Ex: list("ABC") → ['A', 'B', 'C']

3. Using the range() function:
   - Ex: range(1, 10, 2) → [1, 3, 5, 7, 9]

4. Using the input() function:

```python
>>> a = input("Give me a list:")
Give me a list:[1, 2, "a"]
>>> a
[1, 2, 'a']
>>> type(a)
<type 'list'>
```
Modifying a List in Python

- List[range] = Data
  - Ex:

```
>>> L = [3, 4, 5, 6, 7, '8', 9, '10']
>>> L[:2]
[3, 5, 7, 9]
>>> L[:2] = [4, 6, 8, 10]
>>> L[:2]
[4, 6, 8, 10]
>>> L[]
[4, 4, 6, 6, 8, '8', 10, '10']
```

- Using the append() function:
  - List.append(item)
  - Ex: [1, 2, 3].append(5) → [1, 2, 3, 5]
Modifying a List in Python

Using the extend() function:
- `List.extend(Another_list)`
- `Ex: [1, 2, 3].extend([“a”, “b”]) → [1, 2, 3, “a”, “b”]`

```python
>>> L.extend(["a", "b"])
>>> L
[4, 4, 6, 6, 8, ’8’, 10, ’10’, ’a’, ’a’, ’b’]
```

- Using the insert() function:
  - `List.insert(index, item)`

```python
>>> L=[1, 2, 3]
>>> L
[1, 2, 3]
>>> L.insert(1, 0)
>>> L
[1, 0, 2, 3]
```
Removing Elements from a List in Python

- **del statement**: `del L[start:end]`
  
  ```python
  >>> L
  [1, 0, 2, 3]
  >>> del L[1]
  >>> L
  [1, 2, 3]
  ```

- **L.pop() function**: `L.pop([index])`

  ```python
  >>> L=[1,2,3]
  >>> L.pop()
  3
  >>> L
  [1, 2]
  >>> L.pop(0)
  1
  >>> L
  [2]
  ```

- **L.remove() function**: `L.remove(value)`

  ```python
  >>> L
  [2, 1, 3]
  >>> L.remove(1)
  >>> L
  [2, 3]
  ```
Accessing Data/Containers by Names:

Variables

- Naming:
  - Usually: A combination of letters and numbers
  - Ex: a123, 123a, ...

- Scope & Extent:
  - Scope: Where a variable can be accessed.
  - Extent: The lifetime of a variable.

- Typing:
  - Statically typed: The type of a variable is fixed.
  - Dynamically typed: The type of a variable is variable 😊