Ceng 111 – Fall 2015
Week 9a

Functions

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.
Expressions in Python
- Involving **arithmetic** operators -

<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>Unary</td>
<td>Positive 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>
Expressions in Python
- Involving **arithmetic** operators -

**Precedence & Associativity of arithmetic operators.**

**What is precedence?**
- The expression “3 + 4 * 5” has two different interpretations:
  - (3+4)*5
  - 3 + (4*5)

**What is associativity?**
- The expression “3.02 + 4.1 + 5.24” has two different interpretations:
  - (3.02+4.1)+5.24
  - 3.02+(4.1+5.24)
Expressions in Python
- Involving arithmetic operators -

- Precedence & Associativity of arithmetic operators.
- Top: highest precedence.
- Bottom: lowest precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Associativity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Binary</td>
<td>Right-to-left</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>+, -</td>
<td>Unary</td>
<td>Right-to-left</td>
<td>Positive, negative</td>
</tr>
<tr>
<td>*, /, //, %</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Multiplication, Division, Remainder, Modulo</td>
</tr>
<tr>
<td>+, -</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Addition, Subtraction</td>
</tr>
</tbody>
</table>

Floor division (fraction part of the result is removed)
Expressions in Python
- Involving container operators -

- Concatenation (+)
  - “a” + “b” → “ab”

- Repetition (*)
  - “a” * 3 → “aaa”

- Membership (in, not in):
  - “a” in “Mathematics” → True
  - “a” not in “Mathematics” → False

- Indexing ([])

2015 S. Kalkan & G. Ucoluk & A. Cosar - CEng 111
Expressions in Python - Involving *container operators*

Precedence and associativity of container operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Associativity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Indexing</td>
</tr>
<tr>
<td>**</td>
<td>Binary</td>
<td>Right-to-left</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>+, -</td>
<td>Unary</td>
<td>Right-to-left</td>
<td>Positive, negative</td>
</tr>
<tr>
<td>*, /, //, %</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Multiplication &amp; Repetition, Division, Remainder, Modulo</td>
</tr>
<tr>
<td>+, -</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Addition, Subtraction, Concatenation</td>
</tr>
<tr>
<td>in, not in</td>
<td>Binary</td>
<td>Right-to-left</td>
<td>Membership</td>
</tr>
</tbody>
</table>
Expressions in Python
- Involving relational operators -

- **Equality (==)**
  - Two data are equivalent if they represent the same value/information!
  - “Ali” == “Ali” → True

- **Less-than (<)**:
  - A numerical data is less than another if the value of the first is less than that of the second:
    - 3 < 4.5 → True
  - A string is less than another if it is lexicographically (i.e., in ASCII value) less than the second.
    - “abc” < “def” → True
  - A tuple/list is less than another tuple/list if the first different items satisfy the less-than relation.
Expressions in Python
- Involving relational operators -

- **Less-than-or-equal (<=)**
  - `<=` → `(<=)` or `(==)`

- **Greater-than (>)**
  - `>` → `not (<=)`

- **Greater-than-or-equal-to (>=)**
  - `>=` → `not (<)`

- **Not-equal (!=)**
  - `!=` → `not (==)`

Note that in Python, relational operators can be chained. In other words, `a RO b RO c` (where `RO` is a relational operator) is interpreted as:

```
(a RO b) and (b RO c).
```

In most other programming languages, `a RO b RO c` is interpreted as `(a RO b) RO c.`
Expressions in Python

- Involving relational operators -

Precedence & Associativity

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Associativity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Indexing</td>
</tr>
<tr>
<td>**</td>
<td>Binary</td>
<td>Right-to-left</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>+, -</td>
<td>Unary</td>
<td>Right-to-left</td>
<td>Positive, negative</td>
</tr>
<tr>
<td>*, /, //, %</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Multiplication &amp; Repetition, Division, Remainder, Modulo</td>
</tr>
<tr>
<td>+, -</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Addition, Subtraction, Concatenation</td>
</tr>
<tr>
<td>in, not in, &lt;, &lt;=, &gt;, &gt;=, ==, !=</td>
<td>Binary</td>
<td>Right-to-left</td>
<td>Membership, Comparison</td>
</tr>
</tbody>
</table>
Expressions in Python

Involving **logical** operators -

Logical operators manipulate truth values:

- **and** operator
  - \( A \text{ and } B \) \( \rightarrow \) True iff \((A \text{ is True}) \& (B \text{ is True})\)

- **or** operator
  - \( A \text{ or } B \) \( \rightarrow \) True iff either \((A \text{ is True})\) or \((B \text{ is True})\)

- **not** operator
  - \( \text{not } A \) \( \rightarrow \) True iff \(A\) is False
Expressions in Python - Involving **logical** operators -

**Precedence & Associativity**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Associativity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Indexing</td>
</tr>
<tr>
<td>**</td>
<td>Binary</td>
<td>Right-to-left</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>+, -</td>
<td>Unary</td>
<td>Right-to-left</td>
<td>Positive, negative</td>
</tr>
<tr>
<td>*, /, //, %</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Multiplication &amp; Repetition, Division, Remainder, Modulo</td>
</tr>
<tr>
<td>+, -</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Addition, Subtraction, Concatenation</td>
</tr>
<tr>
<td>in, not in, &lt;, &lt;= &gt;, &gt;=, ==, !=</td>
<td>Binary</td>
<td>Right-to-left</td>
<td>Membership, Comparison</td>
</tr>
<tr>
<td>not</td>
<td>Unary</td>
<td>Right-to-left</td>
<td>Logical negation</td>
</tr>
<tr>
<td>and</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>Binary</td>
<td>Left-to-right</td>
<td>Logical OR</td>
</tr>
</tbody>
</table>
Expressions in Python

- Assignment, **not** an operator -

**Single assignment:**
- \( a = 4 \)

**Multiple assignment:**
- \( a = b = c = 4 \)

**Combined assignment:**
- \( a = a + 4 \)  \( \rightarrow \)  \( a += 4 \)
- +=, *=, -=, /=, etc.

```
>>> b += 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```
```
>>> b = 5
>>> b **= 2
>>> b
25
```
Church-Rosser Property

- Evaluation of a mathematical expression is said to have the Church-Rosser Property:
  - A reduction/re-writing system has the Church-Rosser Property if the set of rules always lead to the same results independent of the order of application of the rules.

- A simple example:
  - “If both ends of a string are consonants, remove one”
Church-Rosser Property

How about expressions in programming languages? Do they have Church-Rosser Property?

Answer it yourself considering these:

- Limitations due to fixed size representations of numbers: Remember that \( a+(b+c) \) may not be equivalent to \((a+b)+c\)?
- Side-effects in evaluating some operations and function calls
  - \( f(2) + x \)

LESSON: A programmer has to know the order an expression is evaluated!
So, how are expressions evaluated in HLPL?

Consider these:

- $2 - 3 \times 4 / 8 + 2 \times 4 \times 5 \times 1 \times 8$
- $4 + 2 - 10 / 2 \times 4 \times 2$
- $3 / 3 \times 3 \times 3$

or these:

a) not a == b + d < not a
b) a == b <= c == True
c) True <= False == b + c
d) c / a / b
Dijkstra’s Shunting-Yard Algorithm

Previously on CEng 111!
Algorithm 1 Dijkstra’s Shunting-yard algorithm.

Get next token $t$ from the input queue

if $t$ is an operand then
    Add $t$ to the output queue
endif

if $t$ is an operator then
    while There is an operator $\tau$ at the top of the stack, and either $t$ is left-associative and its precedence is less than or equal to the precedence of $\tau$, or $t$ is right-associative and its precedence is less than the precedence of $\tau$ do
        Pop $\tau$ from the stack, to the output queue.
    endwhile
    Push $t$ on the stack.
endif

if $t$ is a left parenthesis then
    Push $t$ on the stack.
endif

if $t$ is a right parenthesis then
    Pop the operators from the stack, to the output queue until the top of the stack is a left parenthesis.
    Pop the left parenthesis.
endif

if No more tokens to get then
    Pop the operators on the stack, if any, to the output queue.
Dijkstra’s Shunting-Yard Algorithm: Example

\[ A + \frac{B^{CD} \times (E - (F - G - H))}{K} \]

<table>
<thead>
<tr>
<th>FINAL</th>
<th>INITIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ A - B \land C \land D \land (E - (F \land G \land H)) / K ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD\land EFG\land H\land K\land \land \land \</td>
</tr>
</tbody>
</table>
Postfix Evaluation

1. Go from left to right

2. When you see an operator:
   a) Apply it to the last two operands
   b) Remove the last two operands and put the result in place of the operator.
Now

- We have seen expressions/actions changing data

- Now, actions for I/O
Output in Python

%f → Data identifier

We have the following identifiers in Python:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>Integer</td>
</tr>
<tr>
<td>f, F</td>
<td>Floating point</td>
</tr>
<tr>
<td>e, E</td>
<td>Floating point in exponent form</td>
</tr>
<tr>
<td>s</td>
<td>Using the str() function</td>
</tr>
<tr>
<td>r</td>
<td>Using the repr() function</td>
</tr>
<tr>
<td>%</td>
<td>The % character itself</td>
</tr>
</tbody>
</table>
Output in Python

```python
>>> print "I am {0} tall, {1} years old and have {2} eyes".format(1.86, 20, "brown")
I am 1.86 tall, 20 years old and have brown eyes
```

- `{0}`, `{1}`, `{2}` → Data fields
- Instead of numbers, we can give names to the fields:

```python
>>> print "I am {height} tall, {age} years old and have {color} eyes".format(height=1.86, age=20, color="brown")
I am 1.86 tall, 20 years old and have brown eyes
```

- We can re-use the fields

```python
>>> print "I am {height} tall, {age} years old. I am {height} tall.".format(age=20,height=1.86)
I am 1.86 tall, 20 years old. I am 1.86 tall.
```
Basic Statements

Examples:

del L[2]

print “this is a string”
Compound Statements

- Involves more than one expression or statement

Example:

\[
\text{if } \beta \text{ then } \sigma \\
\text{if } \beta \text{ then } \sigma_1 \text{ else } \sigma_2 \\
\text{while } \beta \text{ do } \sigma \\
\text{for } v = 1 \text{ to } 5 \text{ do print } v, v \ast (v - 1)
\]
Conditional Statements

\[
\text{if } \langle \text{boolean expression} \rangle\ \text{then } \langle \text{action} \rangle
\]

Translated to:

compute the \langle \text{boolean expression} \rangle, leave the result in the relevant register \( r \)
branch to \( \alpha \) if \( r \neq 0 \)
carry out \langle \text{action} \rangle
\( \alpha : \langle \text{some actions that follow the if} \rangle \)
Conditional Statements

\[
\text{if } \langle \text{boolean expression} \rangle \text{ then } \langle \text{action}_{TRUE} \rangle \\
\text{if } \neg\langle \text{boolean expression} \rangle \text{ then } \langle \text{action}_{FALSE} \rangle
\]

\[
\text{if } \langle \text{boolean expression} \rangle \text{ then } \langle \text{action}_{TRUE} \rangle \\
\text{else } \langle \text{action}_{FALSE} \rangle
\]
Conditional Statements in Python

1. `if <condition-expression>:
2.     <statements-1>
3. else:
4.     <statements-2>

- the syntax is important!
- indentation is extremely important!
- “else”-part can be omitted!

You can indent your Python code using tabs or space. However, it is a good programming practice to use only one of them while indenting your code: i.e., do not mix them!
Multiple If Statements in Python

```python
1 if <condition-expression-1> :
2     <statements>
3 elif <expression-2> :
4     <statements>
5 .
6 .
7 .
8 elif <expression-M> :
9     <statements>
10 else :
11     <statements>
```
Multiple Nested If Statements in Python

```python
1 if <condition-expression-1> :
  <statements-1>
2       if <condition-expression-2>:
3           <statements-2>
4       else:
5           <statements-3>
6 else:
7       <statements-4>
8
```
Conditional Expression in Python

<exp-1> if <cond-exp> else <exp-2>

Note that this is an expression not a statement!!
Functions: Reusable Actions

In programming, we often combine the statements that we use frequently together into functions.

```c
void main()
{
    hello();

    ... // Some execution here

    hello();
}

void hello()
{
    ... 
    // I am looong function involving lots and lots of statements
    ...
}
```
Functions: Reusable Actions (cont’d)

- Functions in programming are similar to functions in Mathematics but there are differences.

- Difference to mathematical functions:
  - A function in programming may not return a value.
  - A function in mathematics only depends on its arguments unlike the functions in programming.
  - A mathematical function does not have the problem of side effects.
Functions: Reusable Actions

Why do we need functions?

- Reusability
- Structure
- Other benefits of the functional paradigm
Today

- Functions

- Reminder:
  - midterm date: 9 December at 17:40.
Functions in Python

Syntax is important!

Indentation is extremely important
Functions in Python

- Write a Python function that reverses a given number
  - Example: If 123 is given, the output should be 321

```python
def reverse(a):
    return int(str(a)[::-1])
```
Default Parameters in Python

```
1 def reverse_num(Number=123):
2     """reverse_num: Reverse the digits in a number"""
3     str_num = str(Number)
4     print "Reverse of", Number, "is", str_num[::-1]
```

- We can now call `reverse_num` as `reverse_num()` in which case `Number` is assumed to be 123.
- If we supply a value for `Number`, that value is used instead.

```
1 def f(Str, Number=123, Bst="Some"):
2     print Str, Number, Bst
```
While we are at it...
let’s have a look at commenting in Python

```python
1 def reverse_num(Number=123):
2     # reverse_num: Reverse the digits in a number
3     str_num = str(Number)
4     print "Reverse of", Number, "is", str_num[::-1]
```

There are two different ways to put comments in Python: (1) You can use # in which case the rest of the line is not interpreted. (2) You can enclose multiple lines like """" <lines of text> """". The comments that are written using the second option are basically documentation strings and available through the help page.
Nested Functions in Python

```python
1 def f(N):
2     Number = N
3     def g():
4         C = 20
5         return N * Number
6     print "Number", N, "and its square:", g()
```

- Function `g()` can access all the local variables as well as the parameters of function `f()`.

- Function `f()` cannot access the local variables of function `g()`!

- Function `g()` cannot be used before it is defined! For example, the second line could not have been `Number = 10 * g(10)`.

- The indentation is extremely important to understand which statement belongs to which function! For example, the last line is part of function `f()` since they are at the same indentation!