Ceng 111 – Fall 2015
Week 8b

Actions

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.
Variables in Python

- ‘=’ means “Change the content of the variable with the value at the right-hand side”.
  - Assignment!
- The left-side of the assignment should be a valid variable name:
  - Ex:   a+2 = 5  →  NOT VALID!

- We don’t need to define a variable before using it.
- We don’t need to specify the type of a variable.

```
>>> a = 4
>>> b = 3
>>> c = a + b
>>> a
4
>>> b
3
>>> c
7
```
Variable Naming in Python

- Variable names are case sensitive. So, the names a and A are two different variables.
- Variable names can contain letters from the English alphabet, numbers and an underscore _.
- Variable names can only start with a letter or an underscore. So, 10a, $a$, and var$ are all invalid whereas _a and a_20, for example, are valid names in Python.

Variable names cannot be one of the keywords in Python:

```
and      del      from      not      while
as       elif     global    or       with
assert   else     if        pass     yield
break    except   import    print
class    exec     in        raise
continue finally is        return
def      for      lambda    try
```
More on Variables in Python

- Typing of variables:
  - Python is dynamically typed:

```python
>>> a = 3
>>> type(a)
<type 'int'>
>>> a = 3.4
>>> type(a)
<type 'float'>
```

- Using variables:

```python
>>> a = (1, 2, 3, 'a')
>>> type(a)
<type 'tuple'>
>>> a[1]
2
>>> a[-1]
'a'
```
Variables, Values and Aliasing in Python

Every data (whether constant or not) has an identifier (an integer) in Python:

```python
>>> a = 1
>>> b = 1
>>> id(1)
135720760
>>> id(a)
135720760
>>> id(b)
135720760
```

• If the type of the data is mutable, there is a problem!!!

```python
>>> a = ['a', 'b']
>>> b = a
>>> id(a)
3083374316L
>>> id(b)
3083374316L
>>> b[0] = 0
>>> a
[0, 'b']
```

This is called Aliasing.
Now

- We start another ingredient of a program:
  - Actions!
What are actions?

Actions in a PL are the *things* that we can do with the data. **What could they be?**

- Create data or modify data
- Interact with the external environment
Actions for creating/modifying data

- Evaluating a mathematical expression
  - But there are differences to the expressions in Mathematics

- Working with structured data

- Storing results of computations (in another data)

- Making a decision about how to proceed with the computation
  
  - if x*y < 3.1415 then \textit{do some action}
  
  - if "ali" in class_111_list then \textit{do some action}
  
  - if tall("ali") then \textit{do some action}
Interaction-type actions

“Interaction” means Input/Output.

Why interact with the environment? Why do we have Input/Output actions?

- To react on a change in the external environment
- To produce an effect in the external environment
Action Types in High-Level Languages

- Expression evaluation
  - $3 + 4 \times 5 / 2$

- Statement execution
  - `del L[2:4]`
An expression is a calculation which has a set of operations.

- Operations have operators and operands.
- Example: \( 3 + 4 \)
  - \(+\) → operator
  - \(3, 4\) → operands
Today

Actions

- Different operators
- Expression evaluation
- Dijkstra’s shunting yard algorithm

Reminder:

- Tentative midterm date: 9 December at 17:40.
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^{*<em>y} = x^y))</em></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 ([])
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 (\(<=\))
  - \(<= \rightarrow (\<) \text{ or } (==)\)

- Greater-than (\(>\))
  - \(> \rightarrow \text{ not } (\<=)\)

- Greater-than-or-equal-to (\(>\=\))
  - \(>\= \rightarrow \text{ not } (\<)\)

- Not-equal (\(!=\))
  - \(!= \rightarrow \text{ not } (==)\)

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:

\((a \text{ RO } b) \text{ and } (b \text{ RO } c)\).

In most other programming languages, a RO b RO c is interpreted as \((a \text{ RO } b) \text{ RO } c\).
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 and B → True iff (A is True) & (B is True)

**or** operator
- A or B → True iff either (A is True) or (B is True)

**not** operator
- not A → 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.
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:

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

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
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

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.
    Push $t$ on the stack.

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

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.

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

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

```python
>>> print "I am %f tall, %d years old and have %s eyes" % (1.86, 20, "brown")
I am 1.860000 tall, 20 years old and have brown eyes
```

- `%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 <code>str()</code> function</td>
</tr>
<tr>
<td>r</td>
<td>Using the <code>repr()</code> function</td>
</tr>
<tr>
<td><code>%</code></td>
<td>The <code>%</code> 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:

  if $\beta$ then $\sigma$

  if $\beta$ then $\sigma_1$ else $\sigma_2$

  while $\beta$ do $\sigma$

  for $\nu = 1$ to 5 do print $\nu$, $\nu \ast (\nu - 1)$
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 \( \langle \text{action} \rangle \)

\( \alpha : \text{ some actions that follow the if } \)
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

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

\(<\text{exp-1}> \text{ if } \text{<cond-exp> else } \text{<exp-2>}\)

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.

```java
1  void main()
2  {
3      hello();
4
5      // Some execution here
6      hello();
7  }
8
9  void hello()
10  {
11     ...
12     // I am looong function involving lots and lots of statements
13     ...
14  }
15
```
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
Functions in Python

```python
1 def function-name(parameter-1, ..., parameter-N):
  statement-1
  ...
  ...
  statement-M
```

- Syntax is important!
- Indentation is extremely important