Outline

1. Formal Languages
   - syntax and semantics
   - Backus-Naur Form

2. I/O Formats
   - raw_input() and input()
   - the % operator

3. Shared References
   - a name refers to an object
   - side effect of assigning composite objects

4. Summary + Assignments

MCS 260 Lecture 7
Introduction to Computer Science
Jan Verschelde, 10 September 2008
1. **Formal Languages**
   - syntax and semantics
   - Backus-Naur Form

2. **I/O Formats**
   - raw_input() and input()
   - the % operator

3. **Shared References**
   - a name refers to an object
   - side effect of assigning composite objects

4. **Summary + Assignments**

Outline
Programming Languages

classified into 4 generations

1. **machine language**
   instructions are encoded as bit sequences

2. **assembly language**
   a mnemonic system for representing programs

3. **high level programming languages**
   similar to writing algorithms in pseudo code
   early examples: FORTRAN, Cobol, C

4. **problem solving environments**
   environment helps in discovery of algorithms
   mostly limited to one specific problem area
Programming Languages
classified into 4 generations

1. **Machine language**
   instructions are encoded as bit sequences

2. **Assembly language**
   a mnemonic system for representing programs

3. **High level programming languages**
   similar to writing algorithms in pseudo code
   early examples: FORTRAN, Cobol, C

4. **Problem solving environments**
   environment helps in discovery of algorithms
   mostly limited to one specific problem area
Programming Languages
classified into 4 generations

1. machine language
   instructions are encoded as bit sequences

2. assembly language
   a mnemonic system for representing programs

3. high level programming languages
   similar to writing algorithms in pseudo code
   early examples: FORTRAN, Cobol, C

4. problem solving environments
   environment helps in discovery of algorithms
   mostly limited to one specific problem area
Formal Languages
syntax and semantics

Every language is defined through syntax and semantics.

- Syntax states rules how to compose valid sentences.
- Semantics define the meaning of the sentences.

Syntax and semantics of the assignment:

Syntax: an assignment consists of a variable, followed by $=$, followed by an expression.

Semantics: the assignment stores the value of the expression at the right of $=$ into the object referred to by the variable at the left.

Programming languages have a context-free grammar.
Every language is defined through syntax and semantics.

- Syntax states rules how to compose valid sentences.
- Semantics define the meaning of the sentences.

Syntax and semantics of the assignment:

**syntax** an assignment consists of a variable, followed by \(=\), followed by an expression.

**semantics** the assignment stores the value of the expression at the right of \(=\) into the object referred to by the variable at the left.

Programming languages have a *context-free grammar*.
Every language is defined through syntax and semantics.

- Syntax states rules how to compose valid sentences.
- Semantics define the meaning of the sentences.

Syntax and semantics of the assignment:

**syntax**  an assignment consists of a variable, followed by \( = \), followed by an expression.

**semantics**  the assignment stores the value of the expression at the right of \( = \) into the object referred to by the variable at the left.

Programming languages have a *context-free grammar*. 
Formal Languages
syntax and semantics

Every language is defined through syntax and semantics.

- Syntax states rules how to compose valid sentences.
- Semantics define the meaning of the sentences.

Syntax and semantics of the assignment:

**syntax** an assignment consists of a variable, followed by $=$, followed by an expression.

**semantics** the assignment stores the value of the expression at the right of $=$ into the object referred to by the variable at the left.

Programming languages have a *context-free grammar*. 
Outline

1. Formal Languages
   - syntax and semantics
   - Backus-Naur Form

2. I/O Formats
   - raw_input() and input()
   - the % operator

3. Shared References
   - a name refers to an object
   - side effect of assigning composite objects

4. Summary + Assignments
The Backus-Naur Form (or BNF) is a formal notation to define the syntax of programming languages. The BNF of a language consists of

**an alphabet** is a finite set of symbols, containing terminal and nonterminal symbols. Keywords are special terminal symbols.

**rules** of type $A ::= \alpha$ where $A$ is nonterminal, $ ::= \alpha$ is a reserved symbol (in BNF) and $\alpha$ is a string of terminal and nonterminal symbols.

**the axiom** is the initial symbol.

Sentences are derived by starting with the axiom. Then we apply the rules, replacing the axiom by strings until the final string only consists of terminal symbols.
The Backus-Naur Form (or BNF) is a formal notation to define the syntax of programming languages. The BNF of a language consists of

- an alphabet is a finite set of symbols, containing terminal and nonterminal symbols. Keywords are special terminal symbols.
- rules of type \( A ::= \alpha \) where \( A \) is nonterminal, \( ::= \) is a reserved symbol (in BNF) and \( \alpha \) is a string of terminal and nonterminal symbols.

the axiom is the initial symbol.

Sentences are derived by starting with the axiom. Then we apply the rules, replacing the axiom by strings until the final string only consists of terminal symbols.
The Backus-Naur Form (or BNF) is a formal notation to define the syntax of programming languages. The BNF of a language consists of

- **an alphabet** is a finite set of symbols, containing terminal and nonterminal symbols. Keywords are special terminal symbols.

- **rules** of type $A : ::= \alpha$ where $A$ is nonterminal, $::= $ is a reserved symbol (in BNF) and $\alpha$ is a string of terminal and nonterminal symbols.

- the **axiom** is the initial symbol.

Sentences are derived by starting with the axiom. Then we apply the rules, replacing the axiom by strings until the final string only consists of terminal symbols.
The Backus-Naur Form (or BNF) is a formal notation to define the syntax of programming languages. The BNF of a language consists of

an alphabet is a finite set of symbols, containing terminal and nonterminal symbols. Keywords are special terminal symbols.

rules of type $A ::= \alpha$ where $A$ is nonterminal, $::=$ is a reserved symbol (in BNF) and $\alpha$ is a string of terminal and nonterminal symbols.

the axiom is the initial symbol.

Sentences are derived by starting with the axiom. Then we apply the rules, replacing the axiom by strings until the final string only consists of terminal symbols.
Let us define the syntax of a sum of natural numbers.

Denote the axiom by $S$ (initial or start symbol).

Nonterminal symbols are enclosed by `<` and `>`. Vertical bars `|` indicate choice, they mean or.

$$S ::= \langle \text{number} \rangle | \langle \text{number} \rangle + \langle \text{sum} \rangle$$

$$\langle \text{number} \rangle ::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{number} \rangle$$

$$\langle \text{digit} \rangle ::= 0|1|2|3|4|5|6|7|8|9$$

$$\langle \text{sum} \rangle ::= \langle \text{number} \rangle | \langle \text{number} \rangle + \langle \text{sum} \rangle$$
BNF of a Sum

Let us define the syntax of a sum of natural numbers.
Denote the axiom by $S$ (initial or start symbol).
Nonterminal symbols are enclosed by < and >.
Vertical bars $|$ indicate choice, they mean or.

$$S ::= <\text{ number }>|<\text{ number }>+<\text{ sum }>$$

$$<\text{ number }> ::= <\text{ digit }>|<\text{ digit }><\text{ number }>$$

$$<\text{ digit }> ::= 0|1|2|3|4|5|6|7|8|9$$

$$<\text{ sum }> ::= <\text{ number }>|<\text{ number }>+<\text{ sum }>$$
BNF of a Sum

Let us define the syntax of a sum of natural numbers. Denote the axiom by $S$ (initial or start symbol).

Nonterminal symbols are enclosed by < and >.

Vertical bars $|$ indicate choice, they mean or.

$S ::= <\text{number}> | <\text{number}> + <\text{sum}>

<\text{number}> ::= <\text{digit}> | <\text{digit}><\text{number}>

<\text{digit}> ::= 0|1|2|3|4|5|6|7|8|9

<\text{sum}> ::= <\text{number}> | <\text{number}> + <\text{sum}>
BNF of a Sum

Let us define the syntax of a sum of natural numbers. Denote the axiom by $S$ (initial or start symbol).

Nonterminal symbols are enclosed by < and >.

Vertical bars $|$ indicate choice, they mean or.

$$S ::= \langle \text{number} \rangle | \langle \text{number} \rangle + \langle \text{sum} \rangle$$

$$\langle \text{number} \rangle ::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{number} \rangle$$

$$\langle \text{digit} \rangle ::= 0|1|2|3|4|5|6|7|8|9$$

$$\langle \text{sum} \rangle ::= \langle \text{number} \rangle | \langle \text{number} \rangle + \langle \text{sum} \rangle$$
Let us define the syntax of a sum of natural numbers. Denote the axiom by $S$ (initial or start symbol). Nonterminal symbols are enclosed by < and >. Vertical bars | indicate choice, they mean or.

$S ::= <\text{number}> | <\text{number}> + <\text{sum}>

<\text{number}> ::= <\text{digit}> | <\text{digit}><\text{number}>

<\text{digit}> ::= 0|1|2|3|4|5|6|7|8|9

<\text{sum}> ::= <\text{number}> | <\text{number}> + <\text{sum}>
Syntax Diagrams

digit

number
digit

sum

number

+
Syntax Diagrams

digit

number

digit

sum

number
Syntax Errors

Interpreters and compilers always first *parse* the statements and check for syntactical correctness.

A *syntax error* means that the statement does not belong to the language.

In the second phase, for valid statements, the interpreter or compiler checks the semantics.

```python
>>> float(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Syntax error?

```python
>>> flot(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'flot' is not defined
```
Syntax Errors

Interpreters and compilers always first *parse* the statements and check for syntactical correctness.

A *syntax error* means that the statement does not belong to the language.

In the second phase, for valid statements, the interpreter or compiler checks the semantics.

```python
>>> float(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Syntax error?

```python
>>> flot(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'flot' is not defined
```
Syntax Errors

Interpreters and compilers always first *parse* the statements and check for syntactical correctness.

A *syntax error* means that the statement does not belong to the language.

In the second phase, for valid statements, the interpreter or compiler checks the semantics.

```python
>>> float(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Syntax error?

```python
>>> flot(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'flot' is not defined
```
Syntax Errors

Interpreters and compilers always first parse the statements and check for syntactical correctness.

A syntax error means that the statement does not belong to the language.

In the second phase, for valid statements, the interpreter or compiler checks the semantics.

```python
>>> float(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Syntax error?

```python
>>> flot(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'flot' is not defined
```
Syntax Errors

Interpreters and compilers always first *parse* the statements and check for syntactical correctness.

A *syntax error* means that the statement does not belong to the language.

In the second phase, for valid statements, the interpreter or compiler checks the semantics.

```python
>>> float(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

```python
Syntax error?

>>> flot(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'flot' is not defined
```
Outline

1. Formal Languages
   syntax and semantics
   Backus-Naur Form

2. I/O Formats
   raw_input() and input()
   the % operator

3. Shared References
   a name refers to an object
   side effect of assigning composite objects

4. Summary + Assignments
Getting User Input
unformatted or of specific type

Two ways to prompt the user for input:

1. \( s = \text{raw_input}('Give a number : ') \)
2. \( n = \text{input}('Give a number : ') \)

In both cases, the user sees Give a number :

The difference is in the type of the object returned

1. \text{raw_input()} returns always a (raw) string
2. \text{input()} applies Python’s \textit{dynamic typing}

Why two ways?

1. \text{raw_input()} is suitable for any general input
2. \text{input()} returns objects ready for computations
Getting User Input
unformatted or of specific type

Two ways to prompt the user for input:

1  \( s = \text{raw_input}('\text{Give a number : }') \)
2  \( n = \text{input}('\text{Give a number : }') \)

In both cases, the user sees Give a number :

The difference is in the type of the object returned

1  \( \text{raw_input}() \) returns always a (raw) string
2  \( \text{input}() \) applies Python’s \textit{dynamic typing}

Why two ways?

1  \( \text{raw_input}() \) is suitable for any general input
2  \( \text{input}() \) returns objects ready for computations
Getting User Input
unformatted or of specific type

Two ways to prompt the user for input:

1. \( s = \text{raw\_input}('\text{Give a number : }') \)
2. \( n = \text{input}('\text{Give a number : }') \)

In both cases, the user sees \text{Give a number :}

The difference is in the type of the object returned

1. \text{raw\_input()} returns always a (raw) string
2. \text{input()} applies Python’s \textit{dynamic typing}

Why two ways?

1. \text{raw\_input()} is suitable for any general input
2. \text{input()} returns objects ready for computations
Examples of Input

difference between `raw_input()` and `input()`

```python
>>> n = raw_input('give a number : ')
give a number : 5.6
>>> n
'5.6'
>>> type(n)
<type 'str'>
```

To compute with `n`, we must *convert* it to a float:

```python
>>> f = float(n)
>>> f
5.5999999999999996
```

`input()` determines the type of the given object:

```python
>>> n = input('give a number : ')
give a number : 5.6
>>> type(n)
<type 'float'>
```
Examples of Input

difference between `raw_input()` and `input()`

```python
>>> n = raw_input('give a number : ')
give a number : 5.6
>>> n
'5.6'
>>> type(n)
<type 'str'>

To compute with `n`, we must *convert* it to a float:

```python
>>> f = float(n)
>>> f
5.5999999999999996
```

`input()` determines the type of the given object:

```python
>>> n = input('give a number : ')
give a number : 5.6
>>> type(n)
<type 'float'>
```
Examples of Input

difference between raw_input() and input()

```python
>>> n = raw_input('give a number : ')
give a number : 5.6
>>> n
'5.6'
>>> type(n)
<type 'str'>

To compute with n, we must convert it to a float:

```python
>>> f = float(n)
>>> f
5.5999999999999996
```

input() determines the type of the given object:

```python
>>> n = input('give a number : ')
give a number : 5.6
>>> type(n)
<type 'float'>
```
input() for Composite Types
not restricted to numbers

The user may also give tuples and lists:

```python
>>> p = input('give coordinates : ')
give coordinates : (1,3.4,9)
>>> p
(1, 3.3999999999999999, 9)
>>> type(p)
<type 'tuple'>
```

```python
>>> L = input('list fruit : ')
list food : ['apple', 'orange']
>>> L
['apple', 'orange']
>>> type(L)
<type 'list'>
```

Use upper case L instead of l to avoid confusion with 1.
The user may also give tuples and lists:

```python
>>> p = input('give coordinates : ')
give coordinates : (1,3.4,9)
>>> p
(1, 3.3999999999999999, 9)
>>> type(p)
<type 'tuple'>

>>> L = input('list fruit : ')
list food : [ 'apple', 'orange' ]
>>> L
['apple', 'orange']
>>> type(L)
<type 'list'>
```

Use upper case L instead of l to avoid confusion with 1.
input() for Dictionaries
give a collection of data

A dictionary is a set of **key**: **value** pairs.

```python
>>> h = input('office hours : ')
office hours : { 'Mon':2 , 'Wed':11 , 'Fri':10 }
>>> h
{'Fri': 10, 'Mon': 2, 'Wed': 11}
>>> type(h)
<type 'dict'>
```

Switching types of keys and values:

```python
>>> s = input('your schedule : ')
your schedule : { 1:'class' , 4:'meeting' }
>>> s
{1: 'class', 4: 'meeting'}
```
input() for Dictionaries

give a collection of data

A dictionary is a set of key: value pairs.

```python
>>> h = input('office hours : ')
office hours : { 'Mon':2 , 'Wed':11 , 'Fri':10 }
>>> h
{'Fri': 10, 'Mon': 2, 'Wed': 11}
>>> type(h)
<type 'dict'>
```

Switching types of keys and values:

```python
>>> s = input('your schedule : ')
your schedule : { 1:'class' , 4:'meeting' }
>>> s
{1: 'class', 4: 'meeting'}
```
Outline

1. Formal Languages
   syntax and semantics
   Backus-Naur Form

2. I/O Formats
   raw_input() and input()
   the % operator

3. Shared References
   a name refers to an object
   side effect of assigning composite objects

4. Summary + Assignments
The % Operator

to format output

% turns objects to strings for formatted output
% takes multiple arguments to format

The general syntax is

```
print_string % (convert_arguments)
```

- `print_string` contains the string to be printed as it is, with % codes to format objects
- `convert_arguments` contain the data to be converted, either one single item, tuple or dictionary.

```python
>>> s = 'The grade of %s is %d.' % ('Brian', 7)
>>> s
'The grade of Brian is 7.'
```
The % Operator

to format output

% turns objects to strings for formatted output
% takes multiple arguments to format

The general syntax is

```
print_string % (convert_arguments)
```

- `print_string` contains the string to be printed as it is, with % codes to format objects
- `convert_arguments` contain the data to be converted, either one single item, tuple or dictionary.

```python
>>> s = 'The grade of %s is %d.' % ('Brian', 7)
>>> s
'The grade of Brian is 7.'
```
The % Operator
to format output

% turns objects to strings for formatted output
% takes multiple arguments to format

The general syntax is

\[
\text{print\_string} \% (\text{convert\_arguments})
\]

- \text{print\_string} contains the string to be printed as it is, with % codes to format objects
- \text{convert\_arguments} contain the data to be converted, either one single item, tuple or dictionary.

```python
>>> s = 'The grade of %s is %d.' % ('Brian', 7)
>>> s
'The grade of Brian is 7.'
```
% Codes for Strings

<table>
<thead>
<tr>
<th>%c</th>
<th>converts to character, see ASCII code</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>converts to string, applies str()</td>
</tr>
</tbody>
</table>

Every symbol has an ASCII code:

```python
>>> ord(';')
59
```

```python
>>> '%c' % 59
';'
```

Turning numbers to strings:

```python
>>> str(343.23)
'343.23'
```

```python
>>> '%20s' % 343.23
' 343.23'
```

Try also '%-20s'.
% Codes for Strings

<table>
<thead>
<tr>
<th></th>
<th>converts to character, see ASCII code</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td></td>
</tr>
<tr>
<td>%s</td>
<td>converts to string, applies str()</td>
</tr>
</tbody>
</table>

Every symbol has an ASCII code:

```python
>>> ord(';')
59
>>> '%c' % 59
';'
```

Turning numbers to strings:

```python
>>> str(343.23)
'343.23'
>>> '%20s' % 343.23
'    343.23'
```

Try also '%-20s'.
% Codes for Strings

<table>
<thead>
<tr>
<th>%c</th>
<th>converts to character, see ASCII code</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>converts to string, applies <code>str()</code></td>
</tr>
</tbody>
</table>

Every symbol has an ASCII code:

```python
given\texttt{ord(’;’) }\quad 59
>>> \texttt{’%c’} \ % \ 59
’;’
```

Turning numbers to strings:

```python
>>> \texttt{str(343.23)}
’343.23’
>>> \texttt{’%20s’} \ % \ 343.23
’ \ 343.23’
```

Try also ’%–20s’.
Formatting Integers

<table>
<thead>
<tr>
<th>code</th>
<th>converts to ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>%i</td>
<td>a signed decimal integer</td>
</tr>
<tr>
<td>%d</td>
<td>a signed decimal integer</td>
</tr>
<tr>
<td>%u</td>
<td>an unsigned decimal integer</td>
</tr>
<tr>
<td>%o</td>
<td>an octal integer</td>
</tr>
<tr>
<td>%x</td>
<td>an hexadecimal integer, lower case letters</td>
</tr>
<tr>
<td>%X</td>
<td>an hexadecimal integer, upper case letters</td>
</tr>
</tbody>
</table>

Always display the sign:

```python
>>> '%+d' % -23
'-23'
>>> '%+d' % 23
'+23'
```

**Notice:** `'+%d' ≠ '%+d'`!

Hexadecimals are shorthand notations for binary:

```python
>>> '%x' % 2007
'7D7'
```

How many bits?
### Formatting Integers

<table>
<thead>
<tr>
<th>code</th>
<th>converts to ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>%i</td>
<td>a signed decimal integer</td>
</tr>
<tr>
<td>%d</td>
<td>a signed decimal integer</td>
</tr>
<tr>
<td>%u</td>
<td>an unsigned decimal integer</td>
</tr>
<tr>
<td>%o</td>
<td>an octal integer</td>
</tr>
<tr>
<td>%x</td>
<td>an hexadecimal integer, lower case letters</td>
</tr>
<tr>
<td>%X</td>
<td>an hexadecimal integer, upper case letters</td>
</tr>
</tbody>
</table>

Always display the sign:

```python
>>> '%+d' % -23
'-23'
```

```python
>>> '%+d' % 23
'+23'
```

Notice: `%+d` ≠ `%+d` !

Hexadecimal are shorthand notations for binary:

```python
>>> '%X' % 2007
'7D7'
```

How many bits?
Formatting Integers

<table>
<thead>
<tr>
<th>code</th>
<th>converts to ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>%i</td>
<td>a signed decimal integer</td>
</tr>
<tr>
<td>%d</td>
<td>a signed decimal integer</td>
</tr>
<tr>
<td>%u</td>
<td>an unsigned decimal integer</td>
</tr>
<tr>
<td>%o</td>
<td>an octal integer</td>
</tr>
<tr>
<td>%x</td>
<td>an hexadecimal integer, lower case letters</td>
</tr>
<tr>
<td>%X</td>
<td>an hexadecimal integer, upper case letters</td>
</tr>
</tbody>
</table>

Always display the sign:

```python
>>> '%+d' % -23
'-23'
```

Notice: `'+%d' != '%+d'`!

Hexadecimals are shorthand notations for binary:

```python
>>> '%X' % 2007
'7D7'
```

How many bits?
Formatting Floats
shortening decimals and scientific notation

<table>
<thead>
<tr>
<th>code</th>
<th>converts to ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f</td>
<td>a floating-point real number</td>
</tr>
<tr>
<td>%e</td>
<td>scientific notation, using e</td>
</tr>
<tr>
<td>%E</td>
<td>scientific notation, using E</td>
</tr>
<tr>
<td>%g</td>
<td>the value shorter of %f and %e</td>
</tr>
<tr>
<td>%G</td>
<td>the value shorter of %f and %E</td>
</tr>
</tbody>
</table>

Some examples:

```python
>>> the_e = math.exp(1)
>>> '%+8.4f' % the_e
' +2.7183'
>>> '%+12.4E' % the_e
' +2.7183E+00'
>>> '%+12.4g' % the_e
' +2.718'
```
With \n we begin a new line:

```python
>>> print 'hello\nworld'
hello
world
```

Suppose we want to print \n as well:

```python
>>> print r'hello\nworld'
hello\nworld
```

The \r before a string converts it into a raw string. A raw string may contain special characters. In a regular string, these special characters are converted into escape characters.
Raw Strings
strings with special characters

With \n we begin a new line:

```python
>>> print 'hello\nworld'
hello
world
```

Suppose we want to print \n as well:

```python
>>> print r'hello\nworld'
hello\nworld
```

The r before a string converts it into a raw string.

A raw string may contain special characters. In a regular string, these special characters are converted into escape characters.
Raw Strings
strings with special characters

With \n we begin a new line:

```python
g>>> print 'hello\nworld'
hello
world
```

Suppose we want to print \n as well:

```python
g>>> print r'hello\nworld'
hello\nworld
```

The r before a string converts it into a raw string. A raw string may contain special characters. In a regular string, these special characters are converted into escape characters.
Outline

1. Formal Languages
   syntax and semantics
   Backus-Naur Form

2. I/O Formats
   raw_input() and input()
   the % operator

3. Shared References
   a name refers to an object
   side effect of assigning composite objects

4. Summary + Assignments
Names are References
variable names refer to objects

>>> s = 'a'
>>> t = s
The variables \texttt{s} and \texttt{t} refer both to the same object.

>>> s = 'b'
>>> t
'a'
>>> s
'b'

Now \texttt{s} and \texttt{t} refer to different objects.

Although strings are sequences of characters, in Python strings are basic data types, like numbers, unlike lists, tuples, dictionaries.
Names are References
variable names refer to objects

```python
>>> s = 'a'
>>> t = s
The variables s and t refer both to the same object.

>>> s = 'b'
>>> t
'a'

>>> s
'b'

Now s and t refer to different objects.

Although strings are sequences of characters, in Python strings are basic data types, like numbers, unlike lists, tuples, dictionaries.
Names are References
variable names refer to objects

```python
>>> s = 'a'
>>> t = s

The variables s and t refer both to the same object.

>>> s = 'b'
>>> t
'a'
>>> s
'b'

Now s and t refer to different objects.

Although strings are sequences of characters, in Python strings are basic data types, like numbers, unlike lists, tuples, dictionaries.
Outline

1. Formal Languages
   syntax and semantics
   Backus-Naur Form

2. I/O Formats
   raw_input() and input()
   the % operator

3. Shared References
   a name refers to an object
   side effect of assigning composite objects

4. Summary + Assignments
Shared References

side effect of assigning composite objects

Consider lists of strings:

```python
>>> L = ['a']
>>> K = L
```

Both \( L \) and \( K \) refer to the same list.

List are composite data types, change a component:

```python
>>> L[0] = 'b'
>>> K
['b']
```

the change in \( L \) also changed \( K \)!

The same list referred to by both \( L \) and \( K \) is composed of items, objects stored only once.
Shared References

side effect of assigning composite objects

Consider lists of strings:

```python
>>> L = ['a']
```

```python
>>> K = L
```

Both $L$ and $K$ refer to the same list.

List are composite data types, change a component:

```python
>>> L[0] = 'b'
```

```python
>>> K
['b']
```

*the change in $L$ also changed $K$!* 

The same list referred to by both $L$ and $K$ is composed of items, objects stored only once.
Consider lists of strings:

```python
>>> L = ['a']
>>> K = L
```

Both \( L \) and \( K \) refer to the same list.

List are composite data types, change a component:

```python
>>> L[0] = 'b'
```

```python
>>> K
['b']
```

*the change in \( L \) also changed \( K \)!*

The same list referred to by both \( L \) and \( K \) is composed of items, objects stored only once.
Avoiding Shared References

Sometimes we may *on purpose* put brackets around an object to create a shared reference.

But we can avoid sharing by copying the content:

```python
>>> L = ['a']
>>> K = [L[0]]
>>> K
['a']
L and K refer to *different* lists with the *same* content.

>>> L[0] = 'b'
>>> K
['a']
>>> L
['b']
```
Avoiding Shared References

Sometimes we may *on purpose* put brackets around an object to create a shared reference.

But we can avoid sharing by copying the content:

```python
>>> L = ['a']
>>> K = [L[0]]
>>> K
['a']
L and K refer to *different* lists with the *same* content.

>>> L[0] = 'b'
>>> K
['a']
>>> L
['b']
```
Avoiding Shared References

Sometimes we may *on purpose* put brackets around an object to create a shared reference.

But we can avoid sharing by copying the content:

```python
>>> L = ['a']
>>> K = [L[0]]
>>> K
['a']
L and K refer to *different* lists with the *same* content.

>>> L[0] = 'b'
>>> K
['a']
>>> L
['b']
```
Summary + Assignments

In this lecture we covered

- section 6.1, 6.4 in *Computer Science. An Overview*
- more of chapter 4 of *Python Power!*

Assignments:

1. Give the BNF of a floating-point number.
2. Extend BNF for a sum to that of a polynomial in one variable, in fully expanded form.
3. Find the BNF of the Python language.
4. Apply the % operator to print rational numbers given by numerator \( n \) and denominator \( d \) as \( n/d \). Use it in

   $ python printrat.py
   Give numerator : 4352
   Give denominator : 234249
   Your number : 4352/234249.$