Outline

1. Formal Languages
   - syntax and semantics
   - Backus-Naur Form
2. Strings, Lists, and Tuples
   - composite data types
   - building data structures
   - the % operator
3. Shared References
   - a name refers to an object
   - side effect of assigning composite objects
4. Summary + Assignments

MCS 260 Lecture 6
Introduction to Computer Science
Jan Verschelde, 25 January 2016

languages, strings, lists, and tuples

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
   object oriented languages: Ada, C++

4. framework languages, problem solving environments
   environment helps in discovery of algorithms,
   mostly limited to one specific problem area,
   Python comes with batteries included.

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*.

languages, strings, lists, and tuples

Backus-Naur Form

named after John Backus and Peter Naur

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 ::=` α where `A` is nonterminal, `::=` is a reserved symbol (in BNF) and α 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.

BNF of a Sum

Let us define the syntax of a sum of natural numbers.

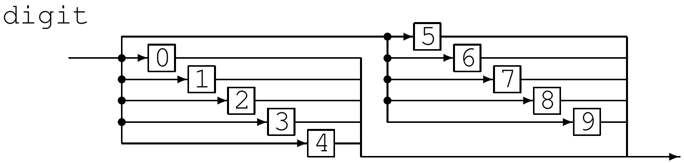
Denote the axiom by `S` (initial or start symbol).

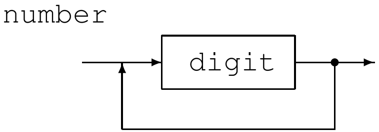
Nonterminal symbols are enclosed by $<$ and $>$.

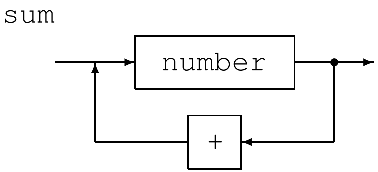
Vertical bars | indicate choice, they mean *or*.

```
S ::= < number >|< number > + < sum >

< number > ::= < digit >|< digit >< number >

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

< sum > ::= < number >|< number > + < sum >
```

Syntax Diagrams







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.

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

Syntax error?

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

languages, strings, lists, and tuples

Strings as Objects – assigning string constants

String constants are sequences of characters,
enclosed between right quotes.

```
>>> x = 'hello'
>>> x
'hello'
>>> id(x)
406912
>>> x = 'I wrote \"hello\"'
>>> print(x)
I wrote "hello"
>>> id(x)
421920
```

- different addresses mean different objects
- use \ to include special characters

computing with strings

With + and * we can calculated with strings:

- + is concatenation
- * is duplication.

```
>>> w = 'we have '
>>> f = 'fun '
>>> w + 3*f
'we have fun fun fun '
```

manipulating strings – selecting from strings

Selection: taking slices of strings

```
>>> w = 'hello world'
>>> len(w)
11
>>> w[4]
'o'
>>> w[0:5]
'hello'

>>> w[-5:]
'world'
```

- negative indices to count backwards
- an omitted index means first or last

lists for compounding data

Lists are a versatile compound data type:

- elements in a list can be of different type
- the length of a list is variable.

```
>>> L = ['Brian', 4 ]
>>> K = ['life' , 'of']
>>> K+L
['life', 'of', 'Brian', 4]
```

deleting an element for a list:

```
>>> N = _
>>> del(N[3])
>>> N
['life', 'of', 'Brian']
```

string ↔ list

Reversing the characters in a string:

```
>>> s = "monday"
>>> L = [a for a in s]
>>> L
['m', 'o', 'n', 'd', 'a', 'y']
>>> L.reverse()
>>> L
['y', 'a', 'd', 'n', 'o', 'm']
>>> t = ''.join(L)
>>> t
'yadnom'
```

tuples: lists of fixed length

Tuples are very similar to lists,
except that their length remains fixed.
Good for storing coordinates of points or vectors.

```
>>> A = (1,2,3)
>>> A
(1, 2, 3)
```

Notice the round brackets. We can make longer tuples:

```
>>> B = (A[0],A[1],A[2],'go')
>>> B
(1, 2, 3, 'go')
```

Notice: `A[:]  = (A[0],A[1],A[2])`.

application of tuples: swapping elements

Suppose we want to swap two variables `x` and `y`.
We have that `x` refers to 4 and `y` refers to 6.

For `y` to refer to the value `x` refers to and for `x` to refer to the value `y` refers to, we must use an auxiliary variable `z` as shown below:

```
>>> x = 4; y = 6
>>> x
4
>>> y
6
>>> z = y
>>> z
6
>>> y = x
>>> x = z
```

We need `z` to hold the value `y` refers to before we execute `y = x`.

swapping with a tuple assignment

With a tuple, we can swap with one assignment:

```
>>> x
6
>>> y
4
>>> (x, y) = (y, x)
>>> x
4
>>> y
6
```

the `in` and `len` operators

We define a string `s`, tuple `t`, and list `L`:

```
>>> s = "hello"
>>> t = ('h','e','l','l','o')
>>> L = ['h','e','l','l','o']
```

The `in` method applies to all three types:

```
>>> 'e' in s
True
>>> 'e' in t
True
>>> 'e' in L
True
```

Similarly: `len(s)`, `len(t)`, and `len(L)`
return the length of a string `s`, tuple `t`, and list `L`.

input() for Composite Types

The `input()` is not restricted to elementary types.
The user may also give tuples and lists:

```
>>> 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.

functions str() and eval()

```
>>> L = range(0,4)
>>> L
[0, 1, 2, 3]
>>> s = str(L)
>>> s
'[0, 1, 2, 3]'
```

Via `str()` we convert an object to its string representation. To revert:

```
>>> K = eval(s)
>>> K
[0, 1, 2, 3]
```

The `eval()` evaluates a string, calling the Python interpreter on the expression.

languages, strings, lists, and tuples

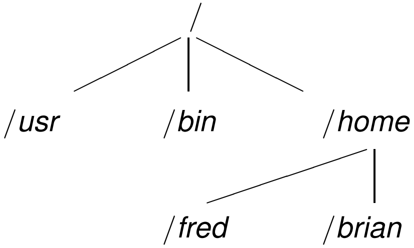
Building Data Structures: nesting tuples

The matrix $\left[\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array}\right]$ can be stored as

```
>>> m = ((1,2),(3,4))
>>> m
((1, 2), (3, 4))
>>> m[0]
(1, 2)
>>> m[1]
(3, 4)
>>> m[0][1]
2
```

Building Data Structures: nesting lists

To store a directory tree



we could use the following nested list:

```
>>> L = ['/usr','/bin',['/home/fred','/home/brian']]
>>> L
['/usr', '/bin', ['/home/fred', '/home/brian']]
```

languages, strings, lists, and tuples

1. Formal Languages
   - syntax and semantics
   - Backus-Naur Form
2. Strings, Lists, and Tuples
   - composite data types
   - building data structures
   - 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.

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

% codes for strings

%c	converts to character, see ASCII code
%s	converts to string, applies `str()`

Every symbol has an ASCII code:

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

Turning numbers to strings:

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

Try also `'%-20s'`.

formatting integers

code	converts to ...
%i	a signed decimal integer
%d	a signed decimal integer
%u	an unsigned decimal integer
%o	an octal integer
%x	an hexadecimal integer, lower case letters
%X	an hexadecimal integer, upper case letters

Hexadecimals are shorthand notations for binary:

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

How many bits?

formatting integers continued ...

Always display the sign:

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

Notice: `'+%d'` $\neq$ `'%+d'`!

Displaying an integer in decimal notation,

- occupying 5 positions, and
- with zeros instead of blanks for empty leading positions

goes like

```
>>> '%05d' % 123
'00123'
```

formatting floats – scientific notation

code	converts to ...
%f	a floating-point real number
%e	scientific notation, using `e`
%E	scientific notation, using `E`
%g	the value shorter of %f and %e
%G	the value shorter of %f and %E

Some examples:

```
>>> the_e = math.exp(1)
>>> '%+8.4f' % the_e
'  +2.7183'
>>> '%+12.4E' % the_e
'  +2.7183E+00'
>>> '%+12.4g' % the_e
'       +2.718'
```

raw strings – strings with special characters

With `\n` we begin a new line:

```
>>> print('hello\nworld')
hello
world
```

Suppose we want to print `\n` as well:

```
>>> 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.

languages, strings, lists, and tuples

Names are References

variable names refer to objects

```
>>> 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.

languages, strings, lists, and tuples

shared references – side effect of assigning

Consider lists of strings:

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

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

List are composite data types, change a component:

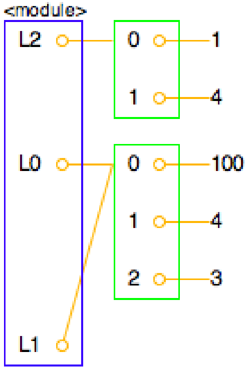
```
>>> L[0] = 'b'
>>> K
['b']
```

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.

visualizing lists

```
from scitools.Lumpy import Lumpy
lumpy = Lumpy()
lumpy.make_reference()
L0 = [1, 4, 3]
L1 = L0
L2 = L1[:-1]
L1[0] = 100
lumpy.object_diagram()
```



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:

```
>>> 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*
- start of chapters 3 & 4 of *Python Programming.*

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.
   ```

5. Compare 2**3**4 with (2**3)**4 and explain.