Principles of programming languages

Lecture 9

Natalia Silvis-Cividjian
e-mail: nsilvis@few.vu.nl

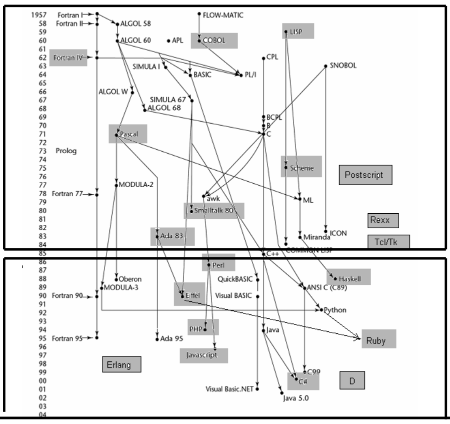
vrije Universiteit *amsterdam*

From last week

- Q: Multiparadigm languages? A: Yes
- Alma-0 combines logic and imperative programming
- Jinni 2006 = Java based Prolog compiler
- See **3rd International Workshop on**

Multiparadigm Constraint Programming Languages

- http://uebb.cs.tu-berlin.de/MultiCPL04/



30 nov

7 dec

Today

Part I. Object orientation

Part II. Scripting languages. First look at Python

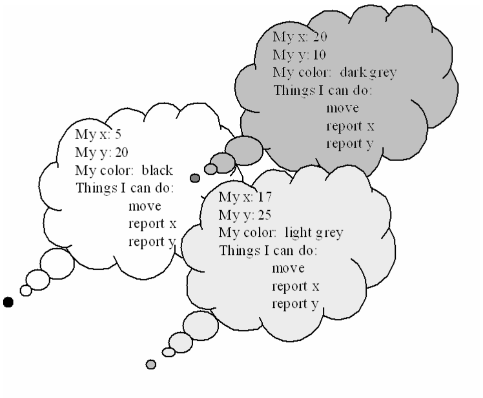
Abstraction in programming

- Subroutines and control abstraction (from early days)
- Object orientation and data abstraction (1980s)

- You have a programming problem
- You can solve a problem in different ways :
  - Procedural
  - Object oriented
  - Aspect oriented

Object-Oriented Style

- Solve problems using **objects**: little bundles of data that know how to do things to themselves
- Not *the computer knows how to move the point*, but rather *the point knows how to move itself*
- Object-oriented languages make this way of thinking and programming easier



OO languages evolution

Simula, mid 60s

Clu, Modula, Euclid, 70s

Smalltalk, 80s

Eiffel, C++, Modula-3, Ada95, Sather, Python, Ruby, Java, PHP, C# , CLOS, Objective C, 90s and 00s

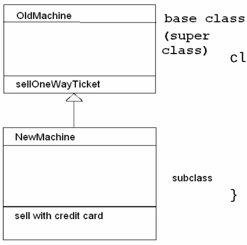
OO languages features

- Encapsulation
- Inheritance
- Dynamic method binding

Some OO highlights

- Multiple inheritance
- Polymorphism
- Dynamic method binding

Inheritance



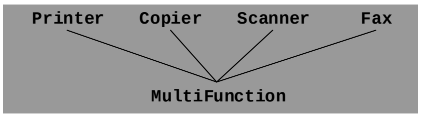
```
class NewMachine extends OldMachine {

     NewMachine(){
     }

     void SellWithCreditCard() {
     ……
     }

     /* other methods */
}
```

Inheritance Questions

- One universal base class?
  - A class from which all inherit: Java's **Object**
  - No such class: C++
- More than one base class allowed?
  - Single inheritance: Smalltalk, Java
  - Multiple inheritance: C++, CLOS, Eiffel
- Forced to inherit everything?
  - Java: derived class inherits all methods, fields
  - Sather: derived class can rename inherited methods (useful for multiple inheritance), or just undefine them

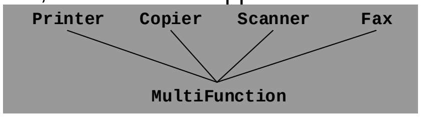
Multiple Inheritance

- In some languages (such as C++) a class can have more than one base class
- For example: a multifunction printer



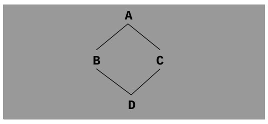
Collision Problem

- The different base classes are unrelated, and may not have been designed to be combined
- **`Scanner`** and **`Fax`** might both have a method named **`transmit`**
- When **`MultiFunction.transmit`** is called, what should happen?



Diamond Problem

- A class may inherit from the same base class through more than one path



- If **`A`** defines a field **`x`**, then **`B`** has one and so does **`C`**
- Does **`D`** get two of them?

Solvable, But…

- A language that supports multiple inheritance must have mechanisms for handling these problems
- Not all that tricky
- The question is, is the additional power worth the additional language complexity?
- Java's designers did not think so

Some OO highlights

- Multiple inheritance
- Polymorphism
- Dynamic method binding

Polymorphism

- Found in many languages, not just OO ones
- Special variation in many OO languages:
  - When different classes have methods of the same name and type, like a stack class and a queue class that both have an **add** method
  - When language permits a call of that method in contexts where the class of the object is not known statically

What means in Java

```
Person x ;
```

Subtype Polymorphism

`Person x` does not always declare **x** to be a reference to an object of the **Person** class

the *type* **Person** may include references to objects of other classes

Java has 2 forms of subtype polymorphism:

§ interfaces
§ class extending

Interfaces

- An interface in Java is a collection of method prototypes

```
public interface Drawable {
  void show(int xPos, int yPos);
  void hide();
}
```

Implementing Interfaces

- A class can declare that it implements a particular interface
- Then it promises to provide **`public`** method definitions that match those in the interface

Example

```
public class Icon implements Drawable {
  public void show(int x, int y) {
    … method body …
  }
  public void hide() {
    … method body …
  }
  …more methods and fields…
}


public class Square implements Drawable, Scalable {
 … all required methods of all interfaces implemented …
}
```

Polymorphism with interfaces

```
public class Window implements Drawable …
public class MousePointer implements Drawable …
public class Oval implements Drawable …
```

```
Drawable d;
d = new Icon("i1.gif");
d.show(0,0);
d = new Oval(20,30);
d.show(0,0);
```

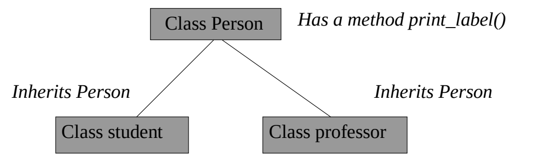
d is a polymorphic variable

```
static void flashoff(Drawable d, int k) {
  for (int i = 0; i < k; i++) {
    d.show(0,0);
    d.hide();
  }
}
```

flashoff is a polymorphic method

Some OO highlights

- Multiple inheritance
- Polymorphism
- Method binding(dispatch)



Both classes `student` and `professor` override the `print_label()` definition and make their own customized print_label methods.

Imagine: you work in a library and made a list of persons (student and professors) and you want to print an address label for all persons who forgot to bring the books back in time.

Which `print_label` method will be used?

2 options are possible:

1. The print_label of the base class Person
- This is **static method binding (dispatch)**
2. The print_label of each derived class, according to its type.
   This is **dynamic method binding (dispatch)**

Static vs. dynamic method binding

- Static binding denies the derived class control over the consistency of its own state
- Dynamic binding imposes run-time overhead, for small routines it is not really necessary
- Smalltalk, Objective-C, Python, Ruby – always dynamic
- Java, Eiffel – by default dynamic, can be overruled (`final, frozen`)
- C++, C# - static by default, dynamic when desired (`virtual`)

Note! Dynamic method binding is not the same with dynamic binding

Is language x object oriented?

- Things are not black and white
- Fundamental OOP concepts: encapsulation, inheritance, dynamic method binding
- Different languages support these concepts to different degrees

OO Discussion

- Some people say a **pure** OO language should make it impossible to write programs that are not OO.
- It means that each data type is a class, every variable is a reference to an object and every subroutine is a object method.
- Smalltalk and Ruby come close to this ideal.
- Ada 95 and Modula -3 are the best examples of imperative languages that permit programmer to write OO if desired.
- C++ has a lot of OO features (multiple inheritance, generics) but also problematic wrinkles: subroutines outside classes, weak type checking, no garbage collection.
- Conclusion: C++ provides all necessary OO tools but requires discipline on the part of the programmer to use them correctly.

Conclusion

- Object-oriented programming style is not the same as programming in an object-oriented language
- Object-oriented languages are not all like Java.

Outline

Part I. Object orientation

➡ **Part II. Scripting languages. First look at Python**

Scripting languages

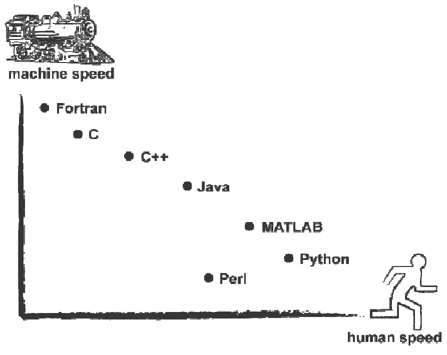
J. Ousterhout's (designer of Tcl) article:

"Scripting: Higher Level Programming for the 21th century"

(from http://www.tcl.tk/doc/scripting.html)

Defines 2 categories of languages:

- System programming languages (Fortran, C/C++, Java, etc)
- Scripting languages (Perl, Python, PHP, etc)



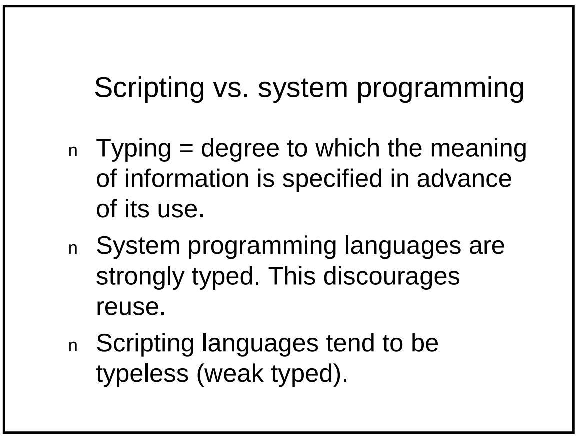
[from Greg Wilson's Software carpentry course]

Scripting vs. system programming

"**Scripting languages** are designed for different tasks than system programming languages, and this leads to fundamental differences in the languages.

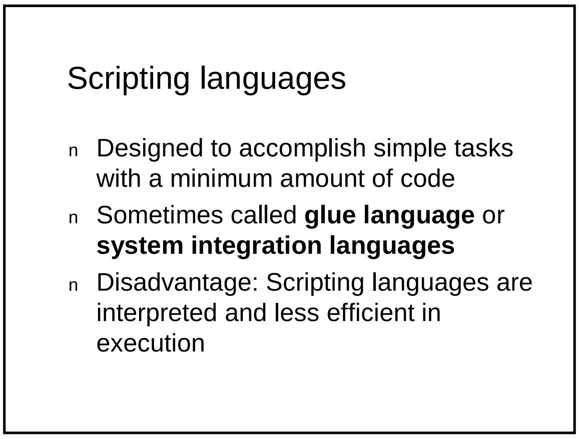
System programming were designed for building data structures and algorithms from scratch. In contrast, scripting languages are designed for gluing. They assume the existence of a set of powerful components and are intended primary for connecting (plugging) components together".

(from Ousterhout's article)



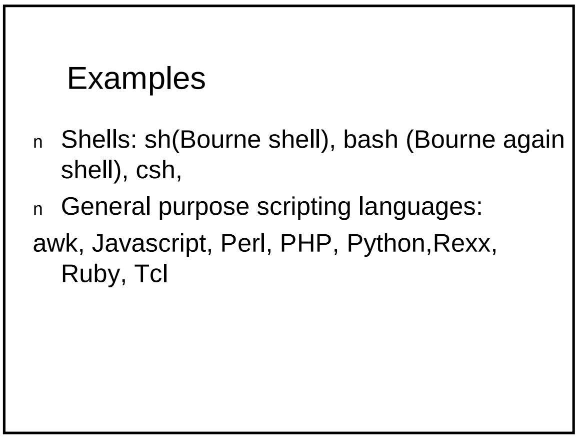
Scripting vs. system programming

- Typing = degree to which the meaning of information is specified in advance of its use.
- System programming languages are strongly typed. This discourages reuse.
- Scripting languages tend to be typeless (weak typed).



Scripting languages

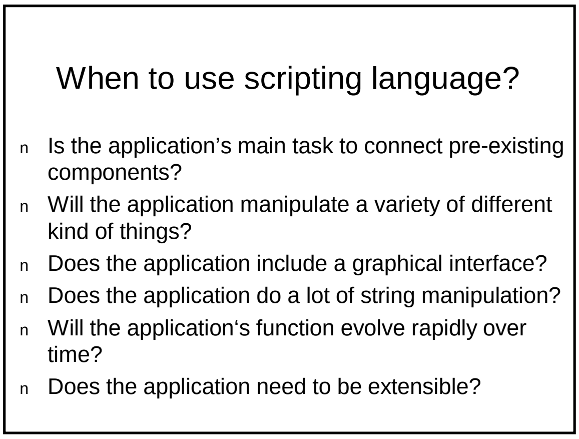
- Designed to accomplish simple tasks with a minimum amount of code
- Sometimes called **glue language** or **system integration languages**
- Disadvantage: Scripting languages are interpreted and less efficient in execution



Examples

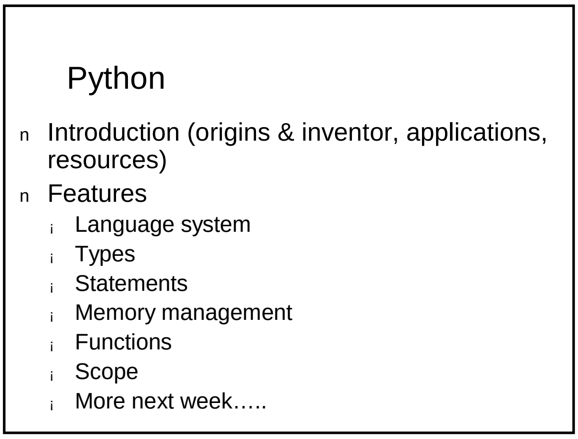
- Shells: sh(Bourne shell), bash (Bourne again shell), csh,
- General purpose scripting languages:

awk, Javascript, Perl, PHP, Python,Rexx, Ruby, Tcl



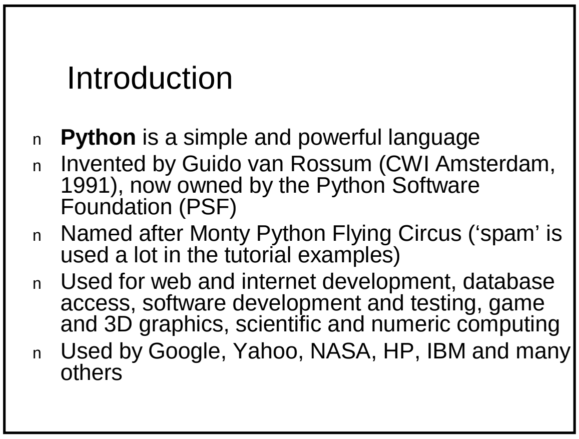
When to use scripting language?

- Is the application's main task to connect pre-existing components?
- Will the application manipulate a variety of different kind of things?
- Does the application include a graphical interface?
- Does the application do a lot of string manipulation?
- Will the application's function evolve rapidly over time?
- Does the application need to be extensible?



Python

- Introduction (origins & inventor, applications, resources)
- Features
  - Language system
  - Types
  - Statements
  - Memory management
  - Functions
  - Scope
  - More next week…..



Introduction

- **Python** is a simple and powerful language
- Invented by Guido van Rossum (CWI Amsterdam, 1991), now owned by the Python Software Foundation (PSF)
- Named after Monty Python Flying Circus ('spam' is used a lot in the tutorial examples)
- Used for web and internet development, database access, software development and testing, game and 3D graphics, scientific and numeric computing
- Used by Google, Yahoo, NASA, HP, IBM and many others

The origins of Python

- The Spam videoclip from Monty Python Flying Circus

- Script from http://bau2.uibk.ac.at/sg/python/Scripts/TheSpamSketch

Why do people use Python?

- Software quality (readability and OOP)
- Developer productivity (1/3 to 1/5 the size of equivalent C++ or Java code)
- Portability
- Support libraries
- Component integration
- Enjoyment

Features

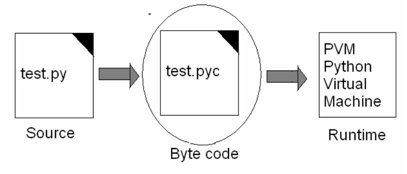
- Simple syntax and easy to learn
- Free and open source
- Very high-level language
- Portable
- Interpreted
- Object-oriented
- Extensible
- Embeddable
- Extensive libraries

Python language system

Execution model: Python interpreter

Source code is compiled to byte code

Byte code is interpreted by the PVM



Python Language system

- You can run Python in interactive mode or by using a file containing the source code
- The source code, for example **spam.py** can be written using any editor (see python site for editors)
- After saving the file, run it by typing :

```
>python spam.py
```

- Or use IDLE = integrated development environment

IDLE (Python GUI).lnk

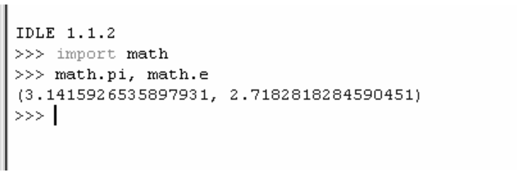
Implementations

Python implementations:

- CPython – standard, coded in ANSI C
- Jython, coded in Java (JPython). Uses Python to script Java applications
- Python.NET
- Psyco JustInTime compiler, speeds Python code

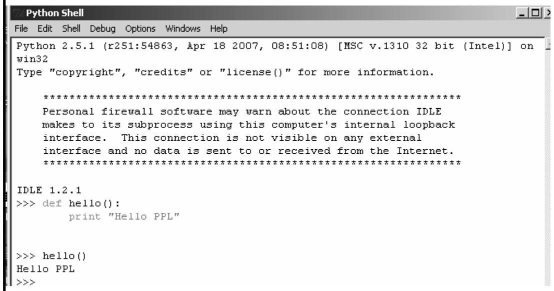
Modules

- Python uses modules. A file is also a module



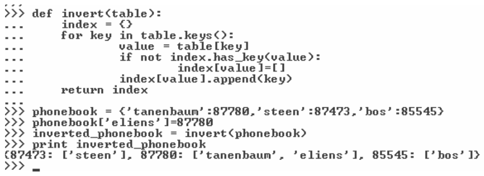
```
IDLE 1.1.2
>>> import math
>>> math.pi, math.e
(3.1415926535897931, 2.7182818284590451)
>>> |
```

“Hello World” Example



```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>> def hello():
        print "Hello PPL"


>>> hello()
Hello PPL
>>>
```

Another example



```
...
>>> def invert(table):
...     index = {}
...     for key in table.keys():
...             value = table[key]
...             if not index.has_key(value):
...                     index[value]=[]
...             index[value].append(key)
...     return index
...
>>> phonebook = {'tanenbaum':87780,'steen':87473,'bos':85545}
>>> phonebook['eliens']=87780
>>> inverted_phonebook = invert(phonebook)
>>> print inverted_phonebook
{87473: ['steen'], 87780: ['tanenbaum', 'eliens'], 85545: ['bos']}
>>> _
```

Big picture

- Programs are composed of modules=packages of names and serve as libraries
- Modules contain statements
- Statements contain expressions
- Expressions create and process objects

Types

- Object is the most fundamental notion in Python
- Everything is an object type in Python and may be processed by Python programs
- In C/C++, Java, a lot of time is spent to implement complicated data structures and their functions (access, insert, search, sort, etc) = it distracts from the real goal

Built-in object types

- Python is a **very high level language**, i.e. its built-in object types are more general and more powerful than in C/C++ or Java
- It is better to use a built in type than to implement your own. A built in type uses algorithms optimized for speed

Types

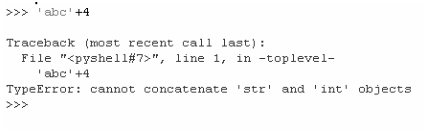
- Formally there are 3 types in Python
  - Numbers
  - Sequences: strings, tuples, lists,
  - Mappings: dictionaries

Numbers

- Integer and floating point numbers: 1234, -24, 3.14e-10, 1.23
- Long integers of unlimited size: 999999999999L
- Octal and hex literals: o177, 0x9ff, 0xFF
- Complex number literals:3+4j, 3.0 + 4.0j

Operators

- Usual operators, + math functions + math modules (NumPy)
- Type conversions similar to C
- Coercion to a superior type (int to long, int to double, etc.)
- But Python does not convert across other type boundaries.

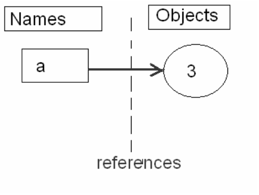


```
>>> 'abc'+4

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in -toplevel-
    'abc'+4
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Assignment

- A variable is created the first time it is assigned. It does not need to be declared, but it has to be assigned before use.

```
>>> a = 3
```

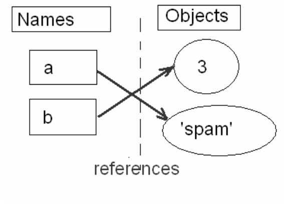


1. Create an object to represent 3
2. Create variable a if it did not exist
3. Bound variable a to object 3

Dynamic typing

- Python uses the **dynamic typing** model = types are determined automatically at runtime
- You have variables (names) and you have objects. Names are entries in a search table, objects are pieces of allocated memory
- Types live with objects, not names

Names and objects



a=3
b=a
a='spam'

Variable name rules

- Syntax: underscore or letter + any number of letters, digits or underscores
- Case sensitive
- Cannot be a reserved word
- __X__ are system defined names

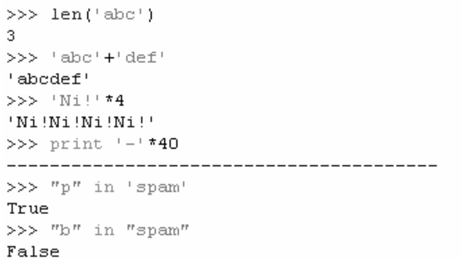
Strings

- Ordered collection of characters
- Examples: ‘spam’, ‘’spa’m’’, ‘’’spam’’’
- Python has powerful string tools
- String is an **immutable sequence**
- Immutable = cannot be changed in place
- Sequence = left to right positional order

Strings

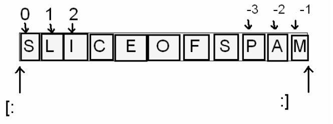
- Escape sequency: for embedding byte code that are difficult to type from keyboard
- Raw string: escape mechanism disabled r”C:\new\test.spm”
- Unicode string: encode larger character sets, support internationalization of applications

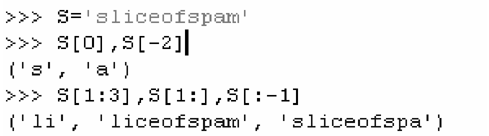
Basic string operations



```
>>> len('abc')
3
>>> 'abc'+'def'
'abcdef'
>>> 'Ni!'*4
'Ni!Ni!Ni!Ni!'
>>> print '-'*40
----------------------------------------
>>> "p" in 'spam'
True
>>> "b" in "spam"
False
```

Indexing and slicing

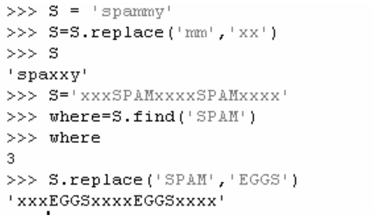




```
>>> S='sliceofspam'
>>> S[0],S[-2]
('s', 'a')
>>> S[1:3],S[1:],S[:-1]
('li', 'liceofspam', 'sliceofspa')
```

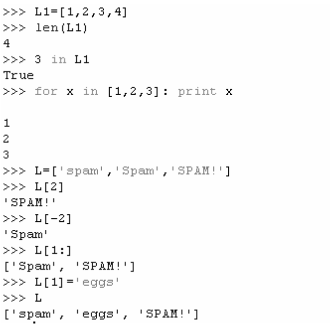
String Methods

- Strings cannot be changed. To change a string make a new one.



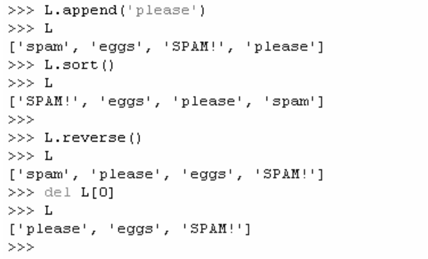
```
>>> S = 'spammy'
>>> S=S.replace('mm','xx')
>>> S
'spaxxy'
>>> S='xxxSPAMxxxxSPAMxxxx'
>>> where=S.find('SPAM')
>>> where
3
>>> S.replace('SPAM','EGGS')
'xxxEGGSxxxxEGGSxxxx'
```

Lists



```
>>> L1=[1,2,3,4]
>>> len(L1)
4
>>> 3 in L1
True
>>> for x in [1,2,3]: print x

1
2
3
>>> L=['spam','Spam','SPAM!']
>>> L[2]
'SPAM!'
>>> L[-2]
'Spam'
>>> L[1:]
['Spam', 'SPAM!']
>>> L[1]='eggs'
>>> L
['spam', 'eggs', 'SPAM!']
```

Lists

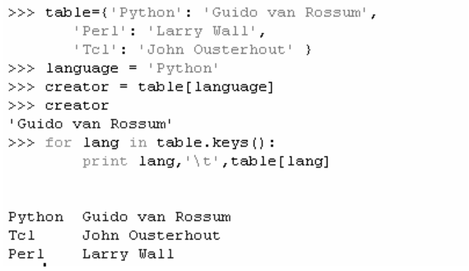


```
>>> L.append('please')
>>> L
['spam', 'eggs', 'SPAM!', 'please']
>>> L.sort()
>>> L
['SPAM!', 'eggs', 'please', 'spam']
>>>
>>> L.reverse()
>>> L
['spam', 'please', 'eggs', 'SPAM!']
>>> del L[0]
>>> L
['please', 'eggs', 'SPAM!']
>>>
```

Dictionaries

- The most flexible built in type in Python
- Unordered collections of data (key,value). Fetch not by position but by key.
- Can replace many of the searching algorithms and data structures manually implemented in lower level languages
- Dictionaries are mutable

Dictionaries

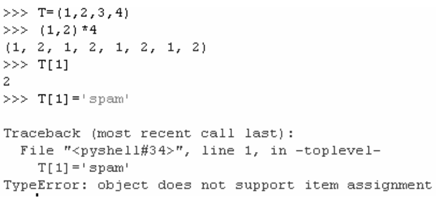


```
>>> table={'Python': 'Guido van Rossum',
       'Perl': 'Larry Wall',
       'Tcl': 'John Ousterhout' }
>>> language = 'Python'
>>> creator = table[language]
>>> creator
'Guido van Rossum'
>>> for lang in table.keys():
        print lang,'\t',table[lang]


Python  Guido van Rossum
Tcl     John Ousterhout
Perl    Larry Wall
```

Tuples

- Tuples are immutable. Have to be used where integrity guarantee is needed, for example as dictionary keys.
- Use lists for collections that might change. Use tuples for the other cases

Tuples



```
>>> T=(1,2,3,4)
>>> (1,2)*4
(1, 2, 1, 2, 1, 2, 1, 2)
>>> T[1]
2
>>> T[1]='spam'

Traceback (most recent call last):
  File "<pyshell#34>", line 1, in -toplevel-
    T[1]='spam'
TypeError: object does not support item assignment
```

Built-in types in Python

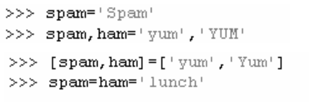
	Category	Mutable
Numbers	Num	No
Strings	Sequence	No
Lists	Sequence	Yes
Dictionaries	Mapping	Yes
Tuples	Sequence	No
Files	Extension	n/a

Statements

- assignment
- calls
- print
- if/elif/else
- for/else
- while/else
- pass
- break,continue
- try/except/finally
- raise
- def, return,yield
- class
- global
- del
- exec
- assert

Assignment

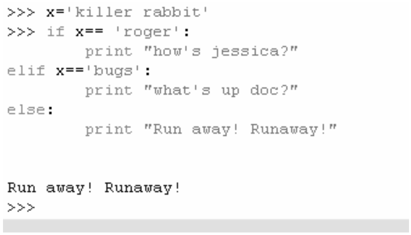
Binding a name to an object



```
>>> spam='Spam'
>>> spam,ham='yum','YUM'
>>> [spam,ham]=['yum','Yum']
>>> spam=ham='lunch'
```

Basic assignment
Tuple and list unpacking assignment
Multiple target assignment

If statement



```
>>> x='killer rabbit'
>>> if x== 'roger':
        print "how's jessica?"
elif x=='bugs':
        print "what's up doc?"
else:
        print "Run away! Runaway!"


Run away! Runaway!
>>>
```

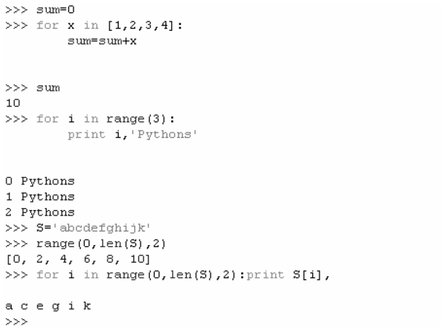
Syntax rules

- Compound statements = header : indented statements
- There are not braces or begin/end delimiters
- A block consists of staments **with the same vertical indentation**
- Blank lines, spaces and comments are ignored

Identation

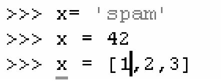
- Indentation is not cosmetical, it is essential for scoping
- Python does not care what you use for indentation
- Use TAB or 2 or 4 spaces but do it consistently

For loops



```
>>> sum=0
>>> for x in [1,2,3,4]:
        sum=sum+x


>>> sum
10
>>> for i in range(3):
        print i,'Pythons'


0 Pythons
1 Pythons
2 Pythons
>>> S='abcdefghijk'
>>> range(0,len(S),2)
[0, 2, 4, 6, 8, 10]
>>> for i in range(0,len(S),2):print S[i],

a c e g i k
>>>
```

Memory management



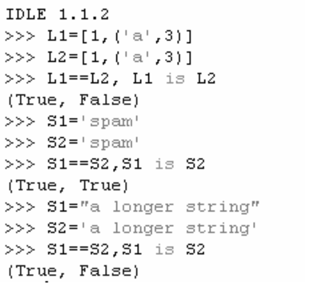
```
>>> x= 'spam'
>>> x = 42
>>> x = [1,2,3]
```

- Python has garbage collection = space occupied by unused objects is automatically reclaimed.
- Actually Python caches and reuses integers and small integers for the case the same object will be generated again (see comparisons example).

Comparisons

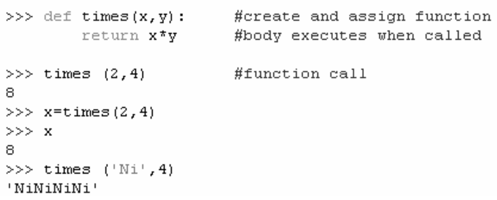
- All Python objects respond to comparison
- Numbers are compared by magnitude
- Strings are compared lexicographically
- Lists and tuples are compared by each component from left top right
- Dictionaries are compared as though comparing sorted (key,value) lists
- == operator tests for value equivalence
- `is` operator tests object identity

Example



```
IDLE 1.1.2
>>> L1=[1,('a',3)]
>>> L2=[1,('a',3)]
>>> L1==L2, L1 is L2
(True, False)
>>> S1='spam'
>>> S2='spam'
>>> S1==S2,S1 is S2
(True, True)
>>> S1="a longer string"
>>> S2='a longer string'
>>> S1==S2,S1 is S2
(True, False)
```

Functions

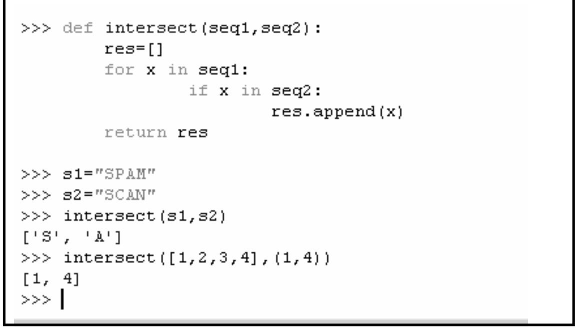


```
>>> def times(x,y):        #create and assign function
        return x*y         #body executes when called

>>> times (2,4)            #function call
8
>>> x=times(2,4)
>>> x
8
>>> times ('Ni',4)
'NiNiNiNi'
```

Polymorphism in Python

- Python is dynamically typed, so polymorphism runs rampant: every operation is a polymorphic operation
- Function time can be applied for any objet type that support * operation
- If types are illegal, Python will detect the error during runtime and raise automatically an exception
- The code does not have to care about specific data types
- A crucial philosophical difference between C/C++ and Java.

Polymorphism



```
>>> def intersect(seq1,seq2):
        res=[]
        for x in seq1:
                if x in seq2:
                        res.append(x)
        return res

>>> s1="SPAM"
>>> s2="SCAN"
>>> intersect(s1,s2)
['S', 'A']
>>> intersect([1,2,3,4],(1,4))
[1, 4]
>>> 
```

- More about Python next week….