Introduction to programming in Python

Ivan Marković Matko Orsag Damjan Miklić
(Srećko Jurić-Kavelj)

University of Zagreb, Faculty of Electrical Engineering and Computing,
Departement of Control and Computer Engineering

2015

University of Zagreb
Faculty of Electrical Engineering
and Computing

What is Python?

Python

A powerful dynamic programming language, useful in a wide variety of application domains.

- dynamic
- interpreted
- object-oriented
- extensive ecosystem of 3rd party libraries
- extensible, easily integrated with C
- portable
- developed by Guido van Rossum (a mathematician)

Bottom line

Faster code development, easier maintenance.

Who uses Python and why?

Python users

- Google (Search, Gmail, YouTube,...)
- NASA (Integrated planning system)
- IBM
- Autodesk (Maya)

What is Python good for?

- Scripting, "Glue logic", prototyping
- Scientific and Numeric Computing (NumPy, SciPy)
- Network programming
- Web development
- Games (Sid Meyer's Civilization IV)
- In short: Everything :)

Installing Python

- On Linux, Python is already installed :)
- Binary installers exist for Windows

Python 2.7 or 3.x

- 3.x is actively developed (but still not supported by all libraries)
- 2.7 is the status quo

Using Python interactively

Starting an interactive Python session:

```
user@host:~$ python
>>> 5+7
12
>>>
```

The interactive shell

Python is interpreted, so we can try things out interactively.

Numbers and booleans

- numbers

```
>>> a = 3
>>> 3**a
>>> 3/2; 3.0/2
>>> b = (a+2)*7
```

- booleans

```
>>> b = -7
>>> a > b
>>> a | True
>>> not True
```

Strings

Strings in Python are a fundamental data type.

```
>>> s1 = 'feeble '; s2="humans"
>>> greeting = s1+s2
>>> len(greeting)
>>> s1*5
>>> greeting.replace('a','HAHAHAHA')
>>> greeting
>>> shout = greeting.upper()
```

Useful information

- Everything in Python is an object
- Objects have functions[1] that operate on their data

```
>>> shout.lower()
```

- Listing all functions belonging to an object

```
>>> dir(shout)
```

- Getting help on any function

```
>>> help(shout.lower)
```

- Objects can be mutable or immutable ("constant")

```
>>> shout[3] = 'c'
```

[1] functions belonging to objects are sometimes called methods

String formatting

Formatting method calls (recommended):

```
>>> "Six by {0}. Fourty {1}".format('nine', 2)
```

Formatting expressions (legacy):

```
>>> "The %s of life is %d" % ('meaning', 42)
```

Exercise

Create the variables `name`, `surname`, age, containing your respective personal information, with all small letters. Using the variables `name` and `surname` and appropriate functions, create a new variable `full_name` which contains your full name, correctly capitalized. Using a formatting method call and the variables `full_name` and age, create the string `hello` with a sentence that introduces you, e.g. `"Hello, I'm Arthur Dent and I'm 42 years old"`.

Dynamic typing and references (Part I)

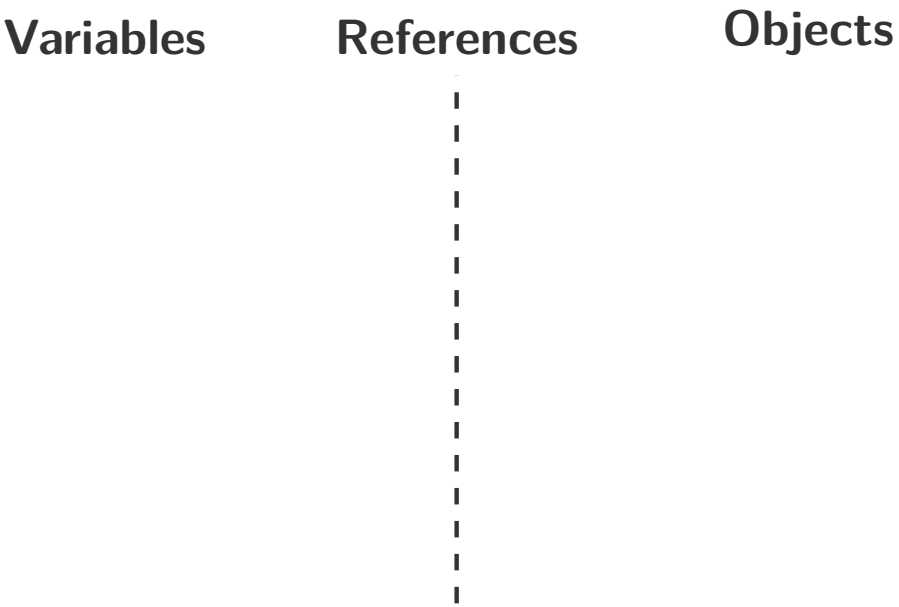
Variables are only named references to objects!

```
>>> a = 3
>>> b = a
>>> a = 'spam'
```

Dynamic typing and references (Part I)

Variables are only named references to objects!

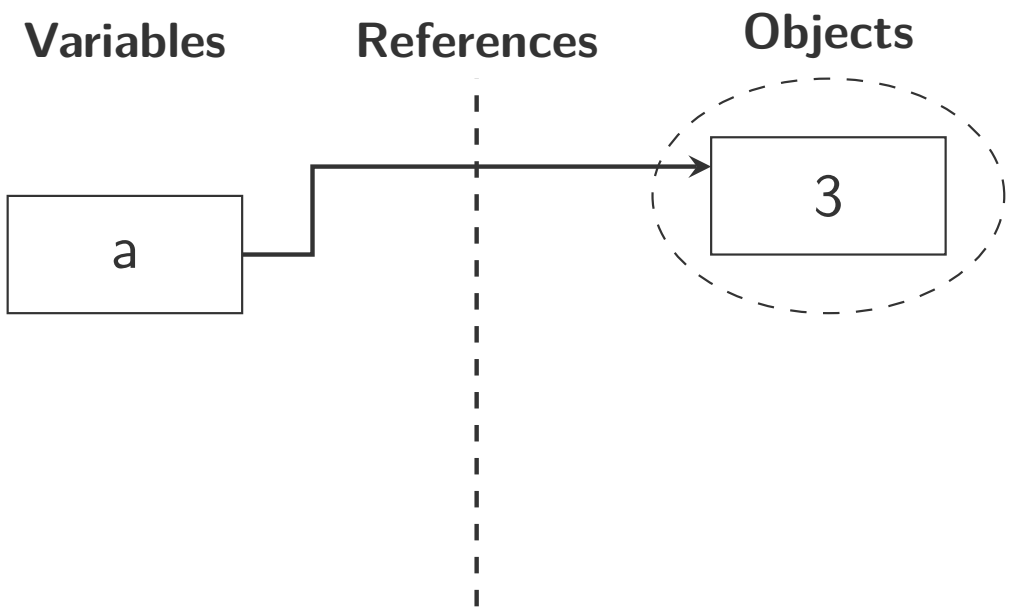
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

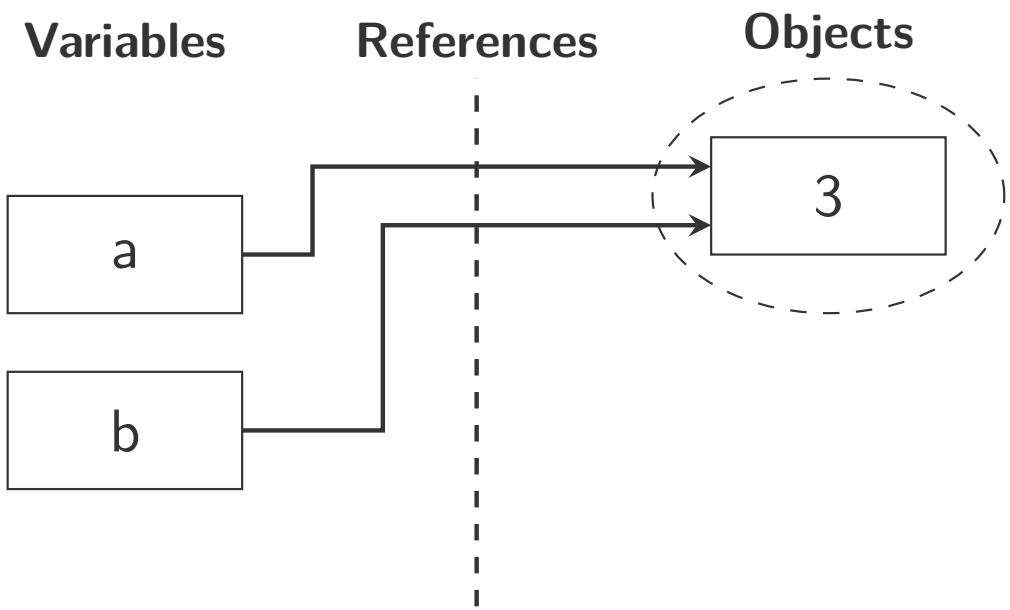
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

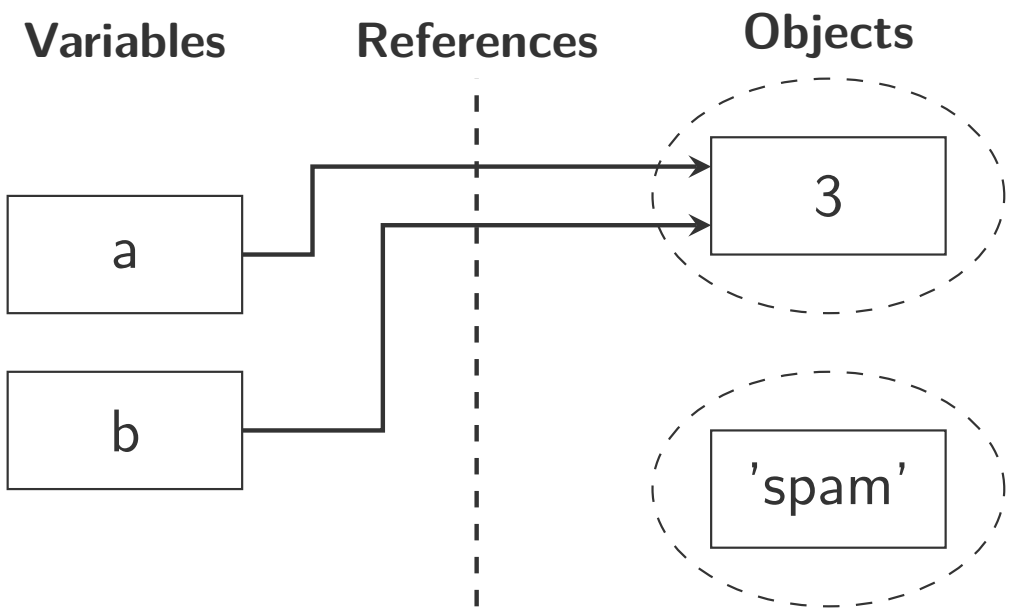
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

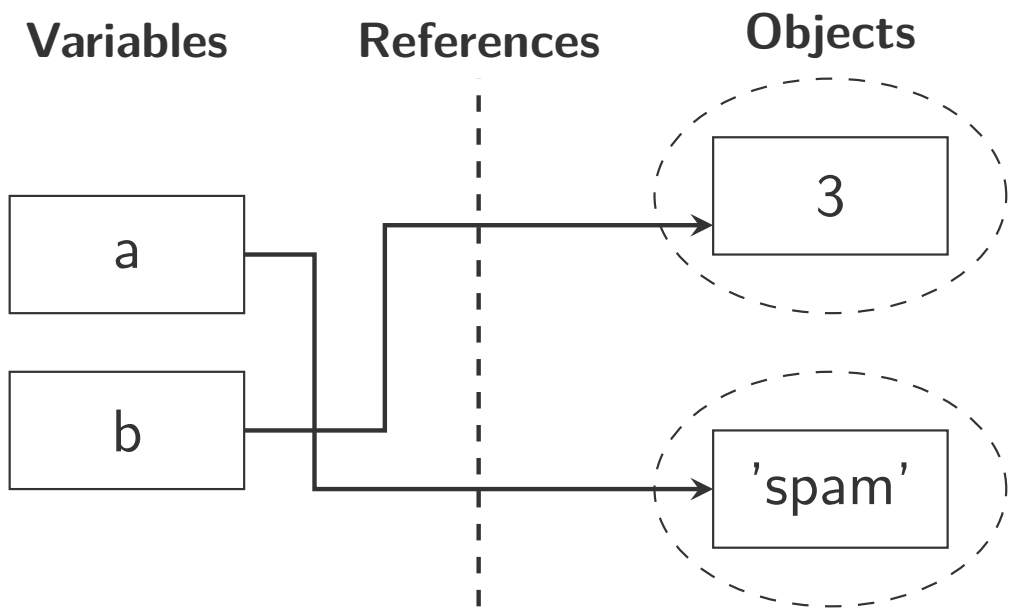
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

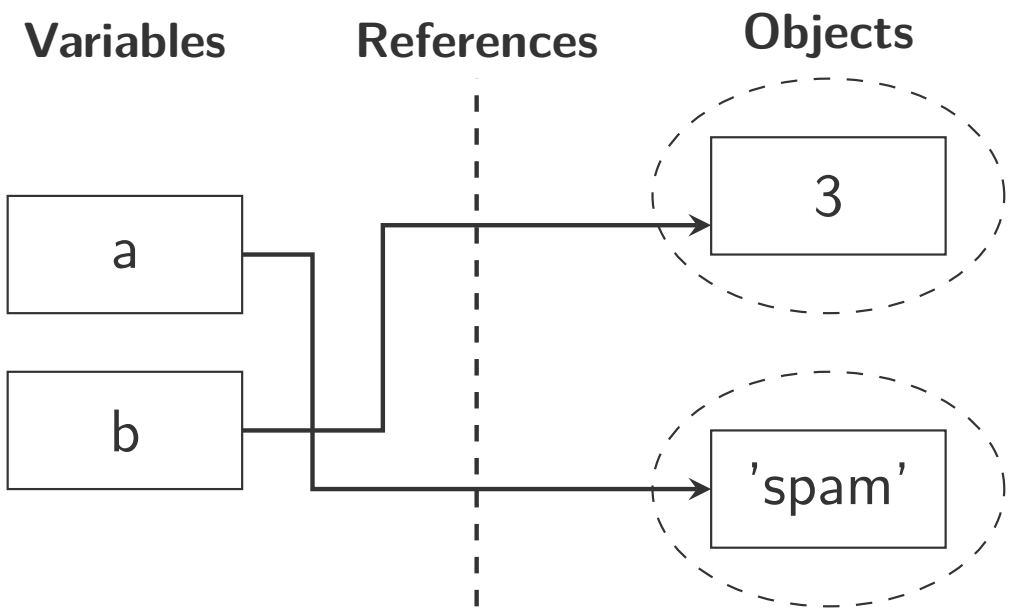
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Note

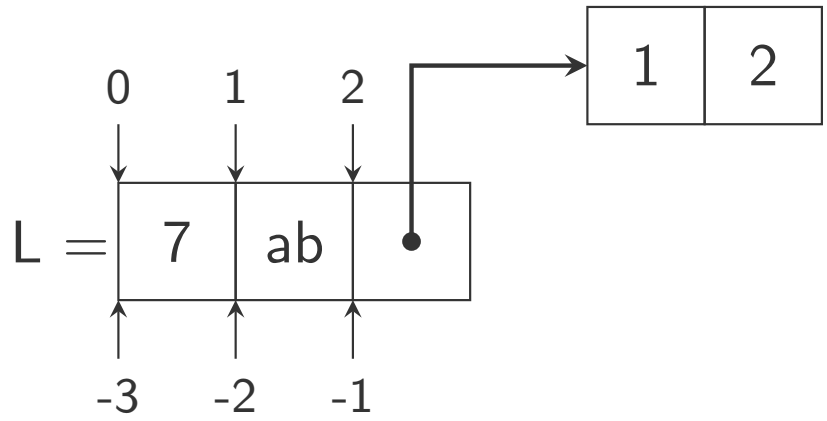
- Variable types are never declared
- Different datatypes can be assigned to the same variable!
- Integers, floats, booleans and strings are immutable types

Lists

- Ordered collections of arbitrary objects, accessed by offset (index)

```
L = [7,'ab',[1,2]]
L[1]; L[-1][0];
L[1:-1]; L[1:] # Slicing!

L[1] = 3.14
len(L)
L.remove(2)
L.extend([-3,22,-0.1])
L.sort()
```



Exericses

1. What effect do arithmetic operators like '+' and '*' have on lists?
2. Try different slicing options, e.g., [:5], [-1:3], ...
3. Insert [0.17, 'c', 12] into L as individual elements.

Dynamic tpying and references (Part II)

Lists are mutable. This, combined with the "variables are references" semantics has non-obvious side-effects.

A quick experiment:

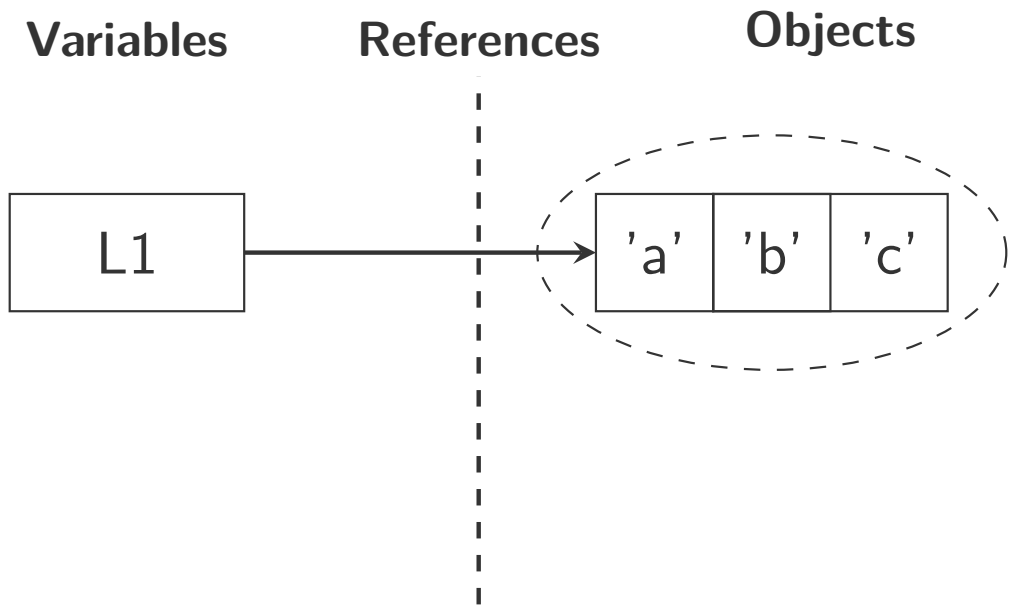
```
>>> L1 = ['a','b','c']
>>> L2 = L1
>>> L2[1] = 17
>>> print(L1)
```

Dynamic tpying and references (Part II)

Lists are mutable. This, combined with the "variables are references" semantics has non-obvious side-effects.

A quick experiment:

```
>>> L1 = ['a','b','c']
>>> L2 = L1
>>> L2[1] = 17
>>> print(L1)
```

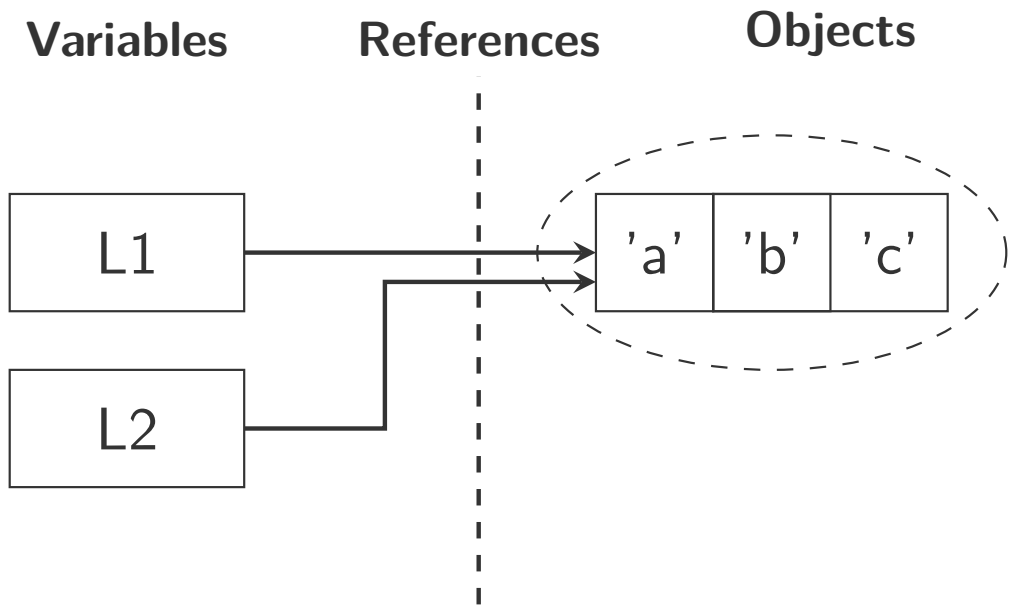


Dynamic tpying and references (Part II)

Lists are mutable. This, combined with the "variables are references" semantics has non-obvious side-effects.

A quick experiment:

```
>>> L1 = ['a','b','c']
>>> L2 = L1
>>> L2[1] = 17
>>> print(L1)
```

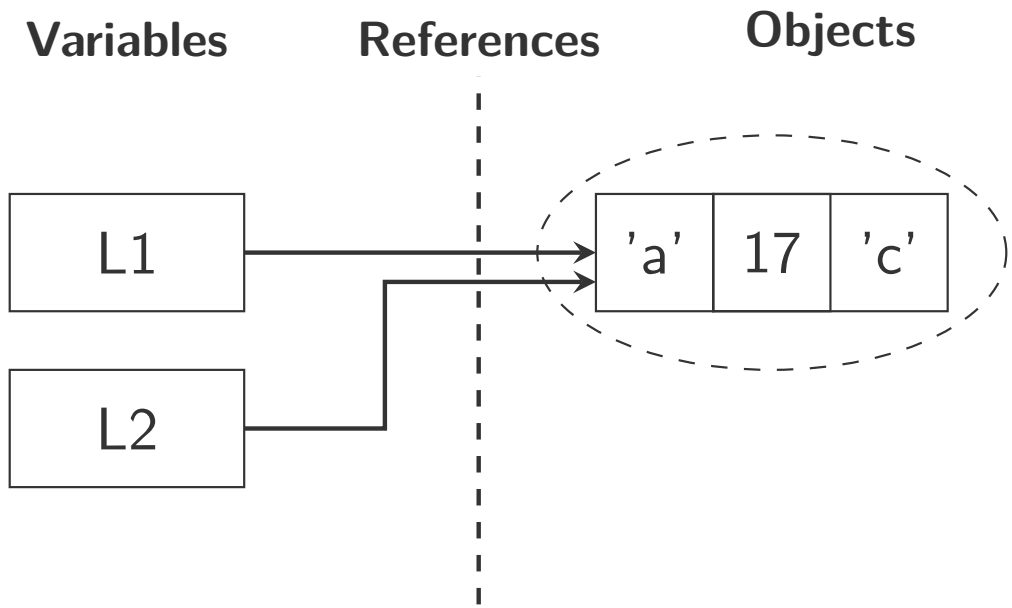


Dynamic tpying and references (Part II)

Lists are mutable. This, combined with the "variables are references" semantics has non-obvious side-effects.

A quick experiment:

```
>>> L1 = ['a','b','c']
>>> L2 = L1
>>> L2[1] = 17
>>> print(L1)
```

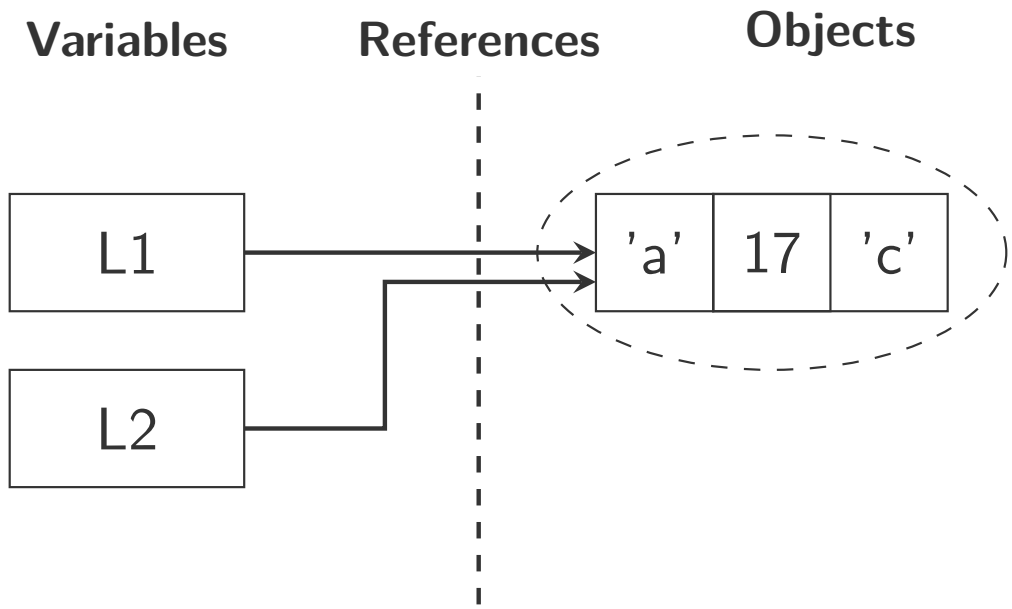


Dynamic tpying and references (Part II)

Lists are mutable. This, combined with the "variables are references" semantics has non-obvious side-effects.

A quick experiment:

```
>>> L1 = ['a','b','c']
>>> L2 = L1
>>> L2[1] = 17
>>> print(L1)
```



Notes

- Lists are mutable!
- Objects in Python are garbage collected!

Safely copying mutable objects

```
>>> L2 = L1[:]
>>> import copy
>>> L2 = copy.copy(L1)
>>> L2 = copy.deepcopy(L1)
```

Safe copying

The slicing operator `[:]` and `copy.copy()` are safe only for "flat" objects. For nested objects (e.g. lists containing lists), use `copy.deepcopy()`.

Quitting the shell:

```
$ exit()
```

or press `Ctrl-D` (EOF)

How to run Python programs?

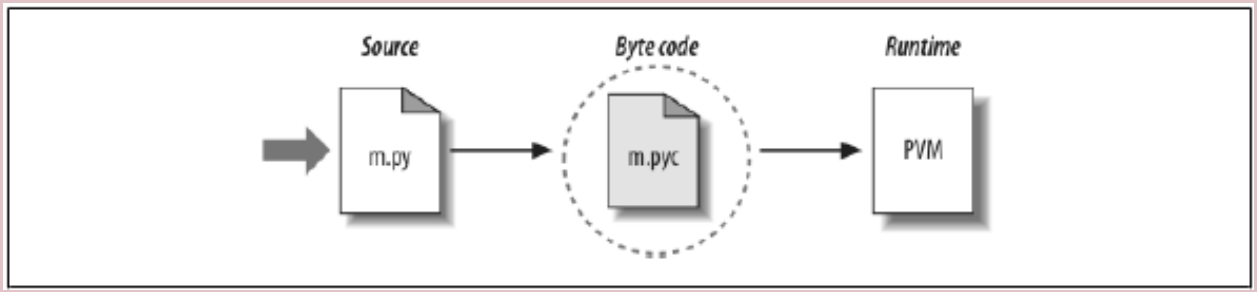
Our first Python program:

```
$ mkdir -p ~/pzros/python
$ cd ~/pzros/python
$ gedit helloworld.py &
```

```
print("I'll be back, baby!")
```

```
$ python helloworld.py
```

The Python interpreter



Portability vs. Speed tradeoff!

Modules

- A text file, with extension .py, containing Python code

```
"""
This is a docstring.
Python will automatically generate documentation from it.
"""

print('Hello beautiful world!')

# This is a block comment. Use comments in your code!
# Below, we will do some vector arithmetic.
v1 = [1,2,3]
v1x2 = 2*v1

print('2*v1={0}'.format(v1x2) ) # Inline comment.
```

for loops

Tip

Set up your editor options to insert spaces instead of tabs!

- Looping over a sequence

```
v1x2 = []
for x in v1:
    v1x2.append(2*x)
```

- Indentation delimits blocks of code (no {})
- Iterator pattern: no need to generate indexes explicitly!
- If we really need indexes[2], there's the `range()` function

```
for i in range(len(v1)):
    v1[i] += 1
```

[2] The only time we really need indexes is when we're modifying the list in-place

List Comprehensions

- Powerful combination of lists and for loops
- List comprehensions are used for generating lists quickly

```
v1pow2 = [x**2 for x in v1]
```

- Much faster than for loops!
- Lists can be combined using the `zip` command

```
v2 = [x+y for (x,y) in zip(v1,v1x2)]
```

- The `(x,y)` object is a tuple, which is an immutable list

Exercise

Implement the dot product of two lists: $\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i$

Files, iterators and for loops

```
$ gedit fileio.py &
```

- Files are elementary data types in Python
- Writing to a text file

```
output = open('myfile.txt', 'w')
output.write('A nice, blank file!\n')
output.write(42)
output.close()
```

- Reading from a text file (iterator pattern, again)

```
for line in open('myfile.txt', 'r'):
    print(2*line)
```

- Read and write methods always work on strings!
- There are safer ways of accessing files using `with`/`as` context managers

while loops, if tests and user input

```
$ gedit volume.py &
```

- Looping over an unknown number of iterations

```
num = 1
while num != 0:
    num = input('Enter the side length: ')
    if num > 1000:
        print('{0} is Too big for me!'.format(num))
    else:
        print('{0}^3 = {1}'.format(num,num**3))
```

- Don't forget the semicolons :)

Functions

```
$ gedit func.py &
```

- The basic tool for code reuse
- Defined with a def statement

```
def add(x, y):
""" Returns x+y """
    return x+y
print(add(5,3))
```

- Inherent polymorphism!

```
add('Py', 'thon')
```

Function scoping rules

Scoping rules

Local – Enclosing – Global – Builtin

- Global scope is visible everywhere
- Local scope overrides global scope

```
X = 7; Y = 17 #Global scope
def printer():
    X = 0  #Local scope
    print(X,Y)
```

- Builtin names can be overriden[3]

```
def override(L):
    len = 7
    print(len(L))
override([1,2,3])
```

[3] Which is almost never what you intended to do :)

Advanced function concepts

- Arguments can be passed by name and have defaults

```
def power(x, y = 0):
    """Returns x^y"""
    return x**y
power(y = 3, x = 2)
```

- In Python, everything is an object, including functions
- Like all objects, functions can be assigned (=> Function pointer!)

```
g = add
print(g(2,3))
```

"Function pointer" assignment

Assignment (function pointer)

Write a function that performs simple numerical integration of a single-variable function, using constant function approximation. The function prototype should be `def int(f,xl,xr,dx)`. To test the correctness of your code, use it to compute $\int_2^4 x^2 dx$ and $\int_0^{3.14} \sin(x) dx$ with integration step 0.001; the results should be close to 18.667 and 2 respectively. (Hint: You will also need `from math import sin` and `def sq(x)`.)

Function design concepts

- Use functions :)
- Keep functions as simple as possible (one function, one purpose)
- Don't use global variables
- Use arguments for inputs and return values for outputs
- Watch out for mutable arguments!
- "Black box design"
- Write docstrings!

Module organization

Modules have two use-cases:

- "Direct execution" of code
- Importing of code (like including header files in C)

```
# Class and function definitions
# That can be imported by other modules
def add(x,y):
    """ Returns x+y """
    return x+y

if __name__ == '__main__':
    # This code is not executed
    # When the module is imported
    print(add(5,7))
```

Importing code from modules

- Importing executes the module[4]
- Objects defined within the module become available in the current context
- We can import all objects from a module

```
>>> import func
>>> func.add(12,-3)
```

- Or a specific object

```
>>> from func import add
>>> add(3,4)
```

- Imported modules are not updated automatically when the source changes!
- The help function shows the docstring

```
>>> help(add)
```

[4] Remember, Python is interpreded!

Making python scripts executable

Allows us to execute Python programs as shell scripts.

1. Add the shebang[5] line

   ```
   #!/usr/bin/env python
   # -*- coding: utf-8 -*-
   ```

   (the second line allows us to use non-ascii characters)

2. Make the script executable

   ```
   $ chmod +x func.py
   $ ./func.py
   ```

[5] shebang = hash(#) + bang(!)

Python libraries

Standard library modules

- Mathematical modules: math, cmath, fractions
- Time and date representations: datetime, calendar
- Operating system interface: os, sys
- Interprocess communication: socket, ssl, asyncore
- Dozens of others...

Third party modules

- Scientific computing tools: NumPy, Matplotlib, SciPy
- Graphics, UI, multimedia: PyGame
- Interprocess communication: ZeroMQ
- Thousands of others...

IPython: a user-friendly shell (and more)

- install IPython

```
$ sudo apt-get install ipython
```

- start IPython (a Matlab-like shell)

```
$ ipython
In[1]:
```

- getting help

```
In[2]: ?len
```

- supports *tab completion*, *command history* and much more
- For a Matlab like experince, invoke with the `-pylab` option

```
$ ipython --pylab
```

- for more info, check out the tutorial

Useful links and further reading

Tutorials:

- Google's Python tutorial
- A Byte of Python
- Non-Programmer's Tutorial for Python 2.6 (Wikibook)

Libraries:

- Official website of the Python programming language
- A Matlab-like Python shell
- Scientific computing tools for Python
- A Python game engine

Books on Python:

- M. Lutz, Learning Pyhton 4th Ed., O'Reilly 2009
- M. Lutz, Programming Pyhton 4th Ed., O'Reilly 2010
- Think Python (free online book)

Homework

Assignment 2: The tic-tac-toe game

Write a simple version of the tic-tac-toe game for two human players. Here are some hints:

- Use a list of lists for keeping track of the score
- A handy way for initializing a 3x3 list of lists is the following comprehension [[-1 for j in range(3)] for i in range(3)]
- Take care in structuring your code: use functions
- Display the playing field after each move
- You have to validate every move
- Use docstrings and comments!
- (Optional) Implement an "AI" strategy to enable human players to play against the computer