Introduction to programming in Python

Ivan Marković Matko Orsag Damjan Miklić

Automation and Robotics
Robot Programming and Simulation

2021

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?

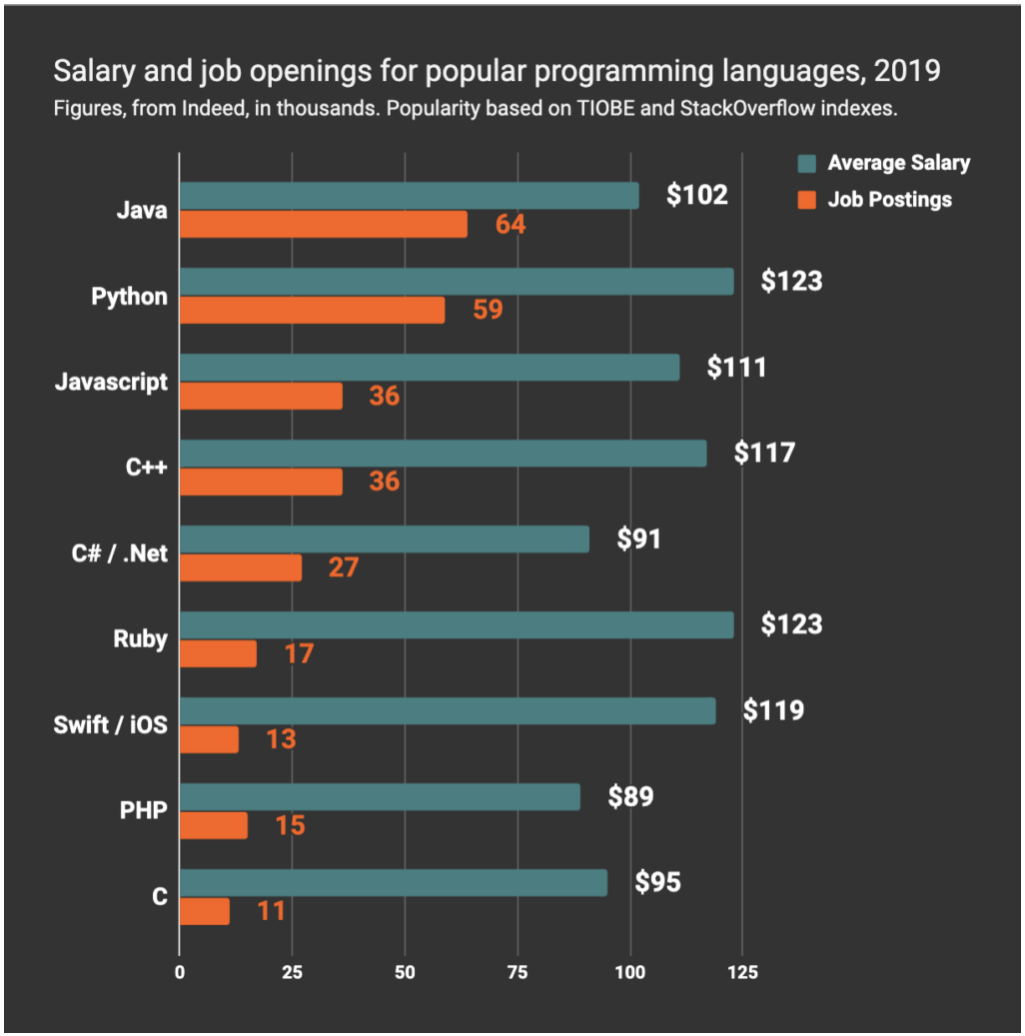
Python users

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

What is Python good for?

- Scripting, "Glue logic", prototyping
- Scientific and Numeric Computing (NumPy, SciPy)
- Machine learning and AI (scikit-learn, PyTorch)
- Network and web programming (Django, Flask)
- Games (Sims4, World of Tanks)

Why should I bother learning Python?



Source: Code Platoon

Installing Python

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

Python 2.7 or 3.x

- 3.x is actively developed, default in Ubuntu Focal/ROS Noetic
- 2.7 is officially EOL, but still used in Ubuntu Bionic/ROS Melodic

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

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

f-Strings (recommended):

```
>>> f"The {'meaning'} of life is {42}"
```

Formatting method calls:

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

Formatting expressions (legacy):

```
>>> "Six by %s. Fourty %d" % ('nine', 2)
```

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 all three of the above methods 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)

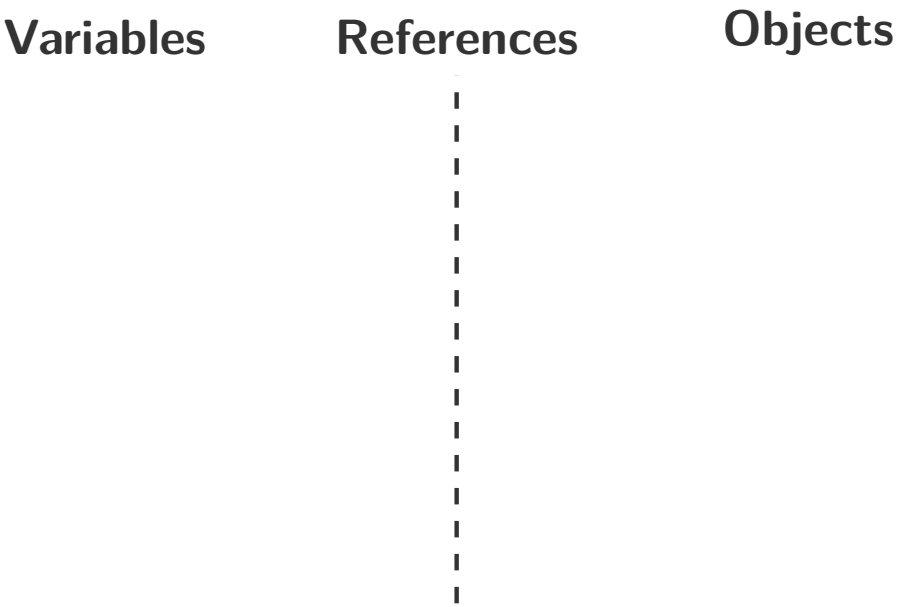
Variables are only named references to objects!

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

Dynamic typing and references (Part I)

Variables are only named references to objects!

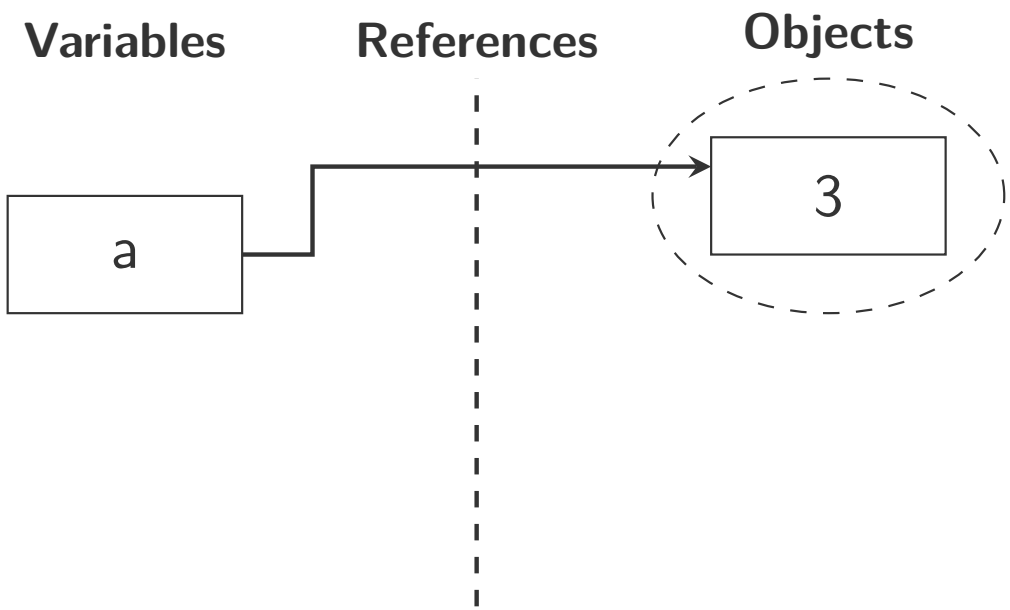
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

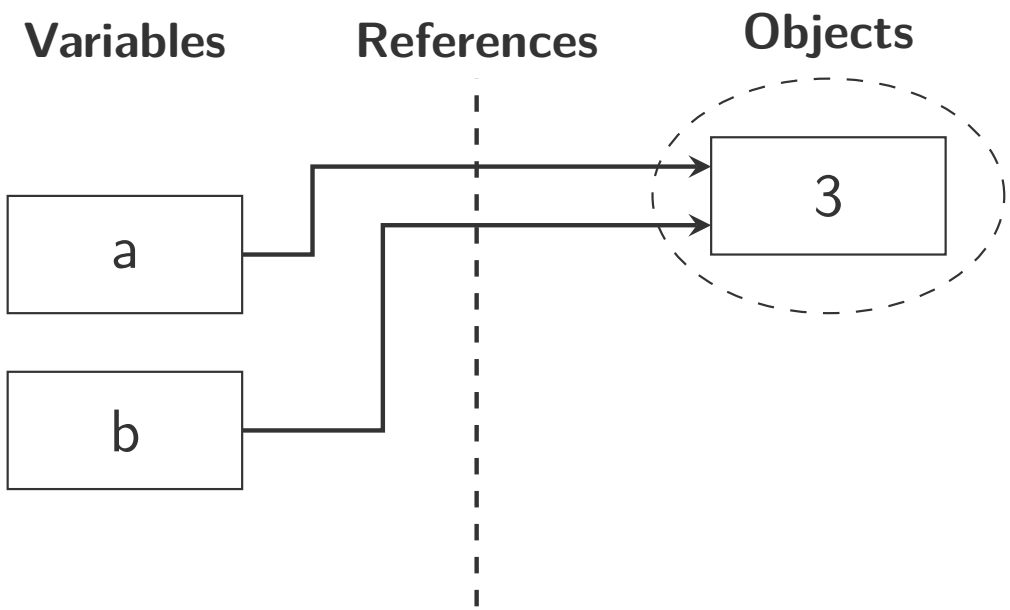
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

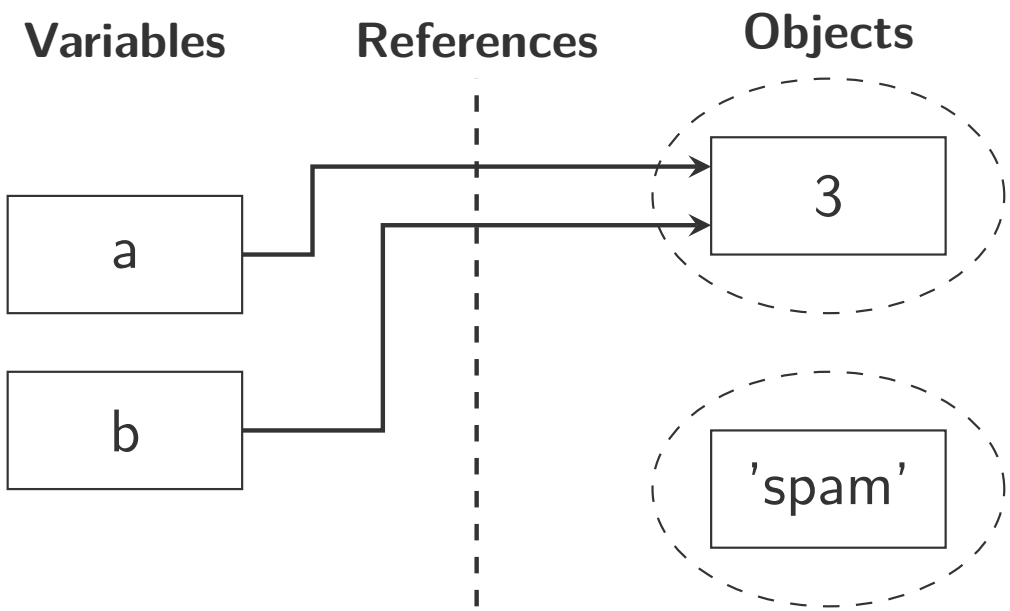
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

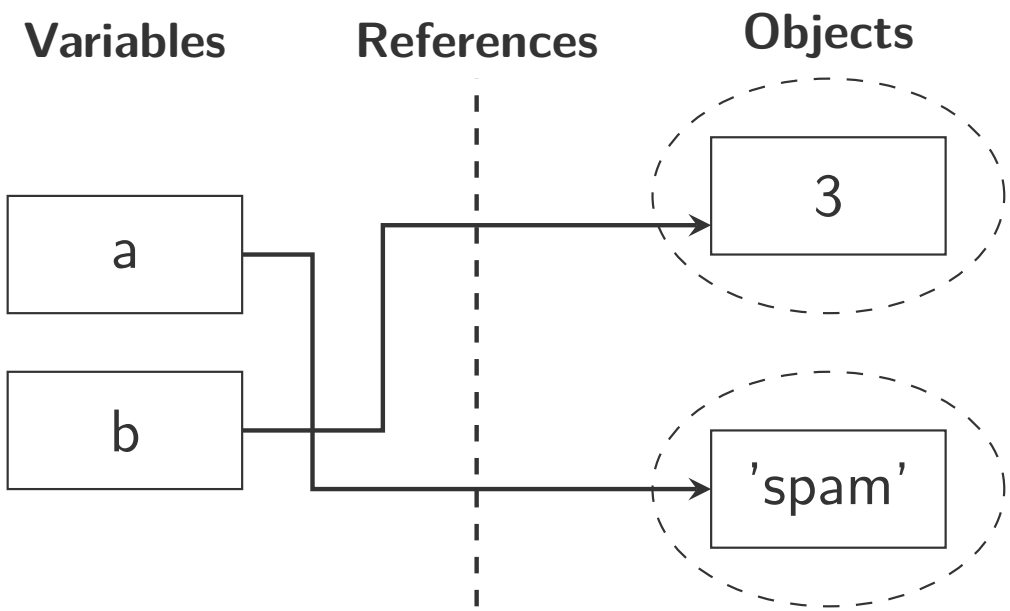
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

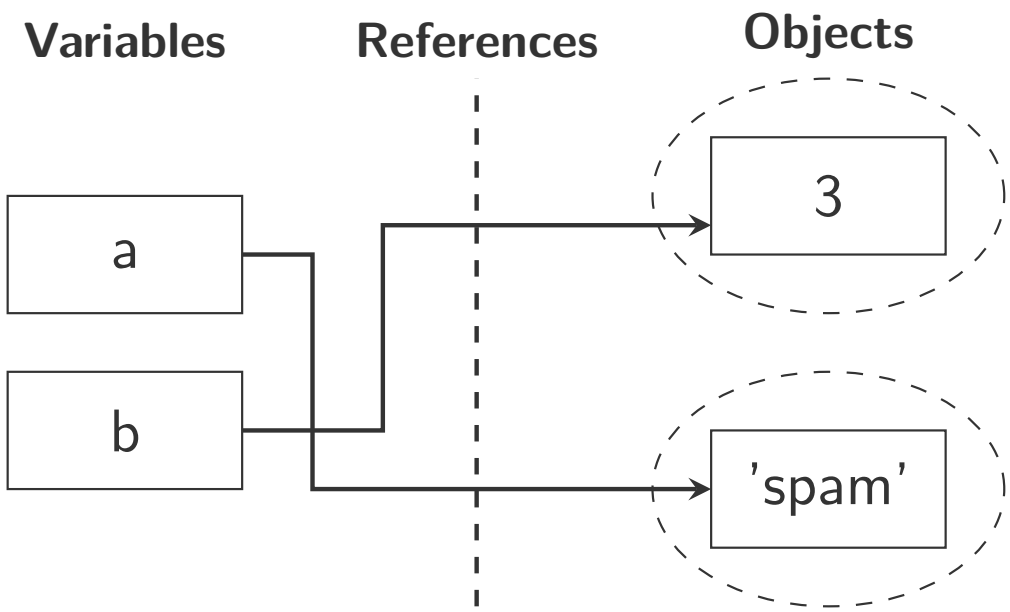
```
>>> a = 3
>>> b = a
>>> a = 'spam'
```



Dynamic typing and references (Part I)

Variables are only named references to objects!

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



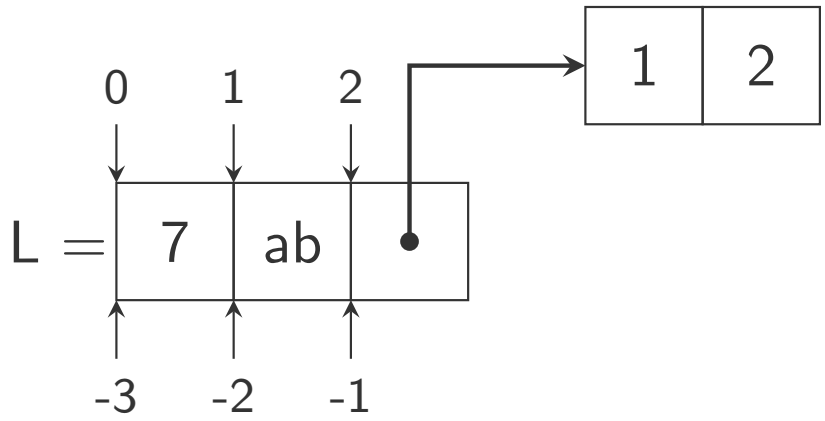
Note

- 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(7)
L.extend([-3,22,-0.1])
```



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:

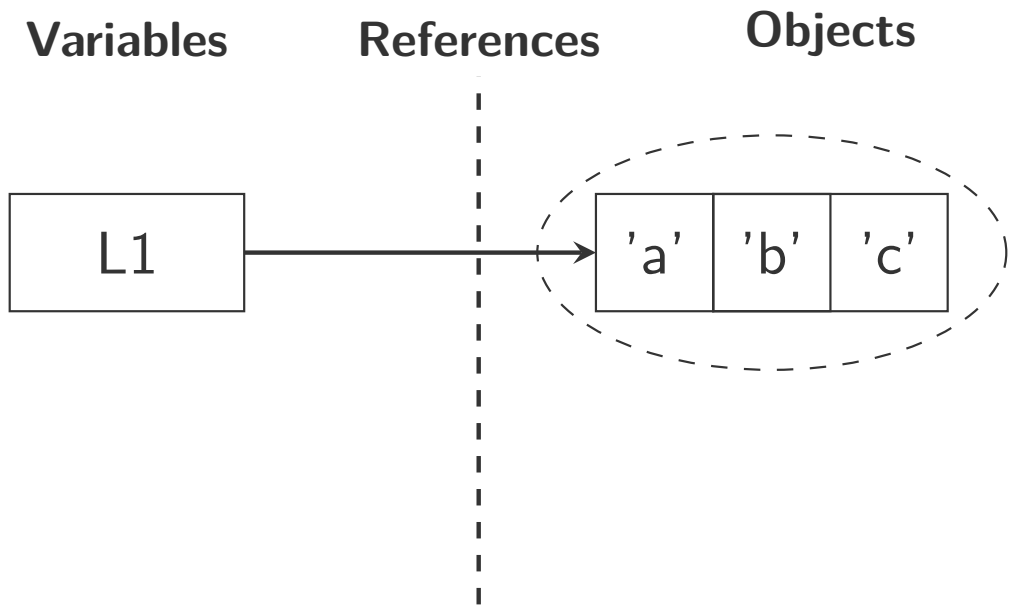
```
>>> L1 = ['a','b','c']
>>> L2 = L1
>>> L2[1] = 17
>>> print(L1)
```

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)
```

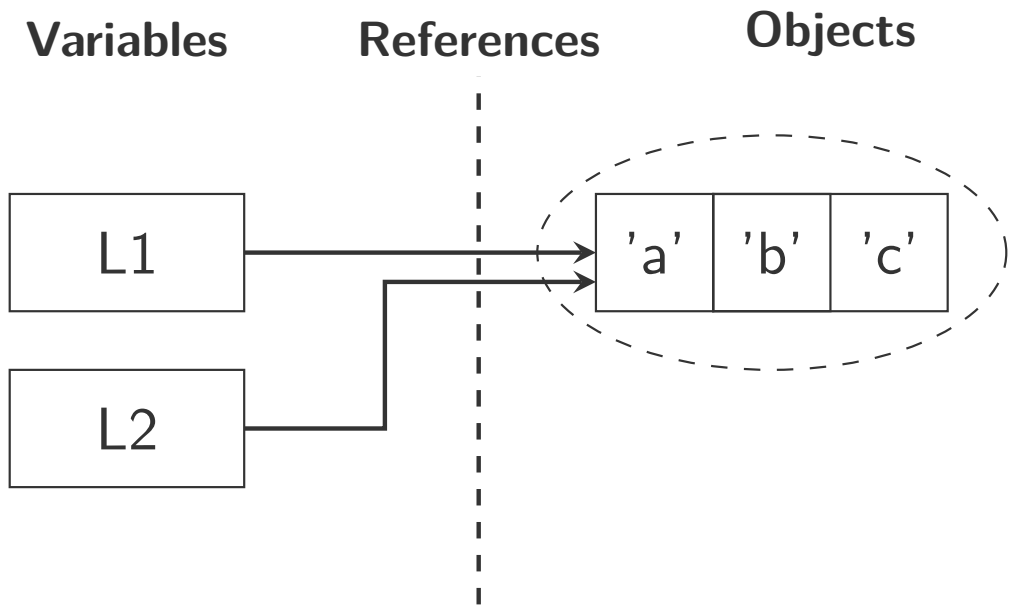


Dynamic tpying and references (Part II)

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)
```

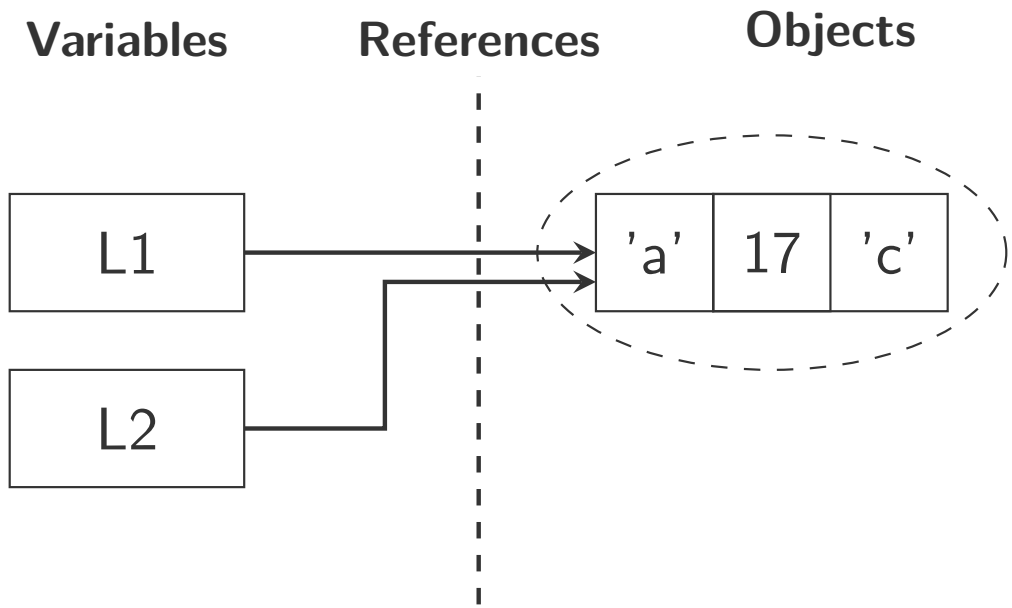


Dynamic tpying and references (Part II)

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)
```

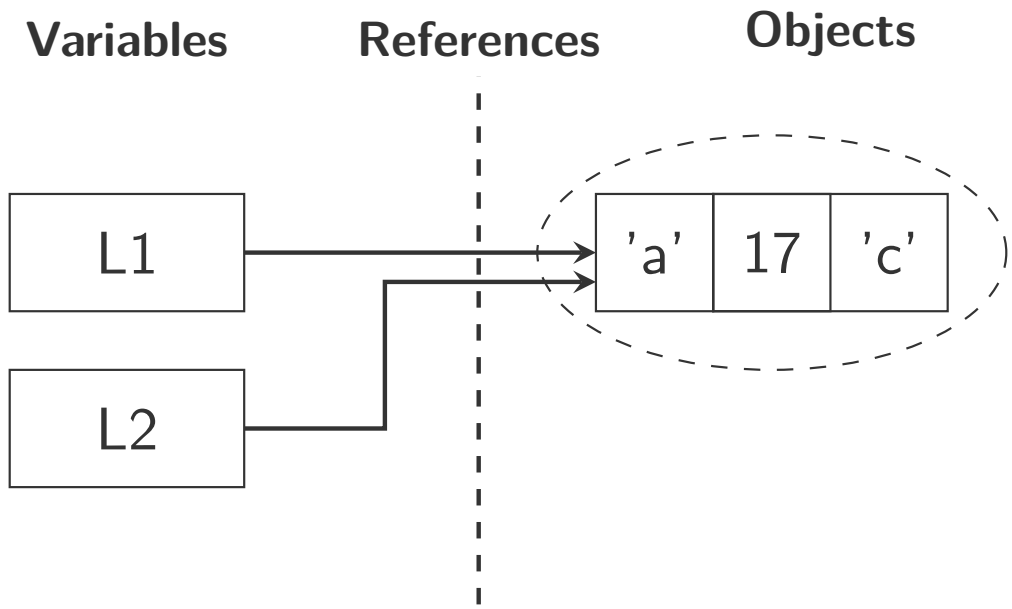


Dynamic tpying and references (Part II)

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[:]
>>> L2 = L1.copy() # Python >= 3.3
>>> import copy
>>> L2 = copy.copy(L); L2[0] = 0; L2[1][0] = 0
>>> L3 = copy.deepcopy(L); L3[0] = 3; L3[1][0] = 'L3'
```

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)

List exercises

Exercise: List indexing

Using the list `L=[1,2,5,6,9,10]`

1. Create a new list, `L2`, containing all the numbers from `1` to `10`, in sequential order, using the `list.insert` method
2. Same as the above, but using list arithmetic (slicing and the `+` operator)
3. Same as the above, but using `list.append` and `list.sort` methods
4. Demonstrate three ways of creating a new list `L3`, containing the first three elements of `L2`

How to run Python programs?

Our first Python program:

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

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

```
$ python helloworld.py
```

Modules

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

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

print("I'll be back!")

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

print('2*{0}={1}'.format(v1,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(str(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 colons :)

Exercises: loops and file I/O

Exercise: User input and writing to a file

Create a script which lets the user input a sequence of numbers, one by one, until the number 0 is entered. Store the numbers in a list. After user input has been finished, compute the sum of the sequence (you can use the built-in `sum` function). Open the file `sequence.txt` for writing and write the original sequence of numbers on the first line, separated by a single space. Write the computed sum on the second line and close the file.

Exercise: Reading from a file and list comprehensions

Open the file `sequence.txt` for reading and read the first two lines. Convert the first line to a list of floating point numbers using the `split` method, `float` function and a list comprehension. Convert the second line to a floating point number. Check if the number on the second line corresponds to the sum of numbers on the first line. Print the result.

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')
```

Arrays as function arguments

```
$ gedit plusone.py &
```

```
def plusone(vin):
        """ Increments the input vector by one """
        for (i,x) in enumerate(vin):
            vin[i] = x+1
        return vin

if __name__ == '__main__':
        v = [1,2,3]
        v1 = plusone(v)
        dv = [y-x for (x,y) in zip(v,v1)]
        print(dv)
```

Arrays as function arguments

```
$ gedit plusone.py &
```

```
def plusone(vin):
        """ Increments the input vector by one """
        for (i,x) in enumerate(vin):
            vin[i] = x+1
        return vin


if __name__ == '__main__':
        v = [1,2,3]
        v1 = plusone(v)
        dv = [y-x for (x,y) in zip(v,v1)]
        print(dv)
```

Passing arrays to functions

Remember, in Python, all objects are passed by reference!

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)
```

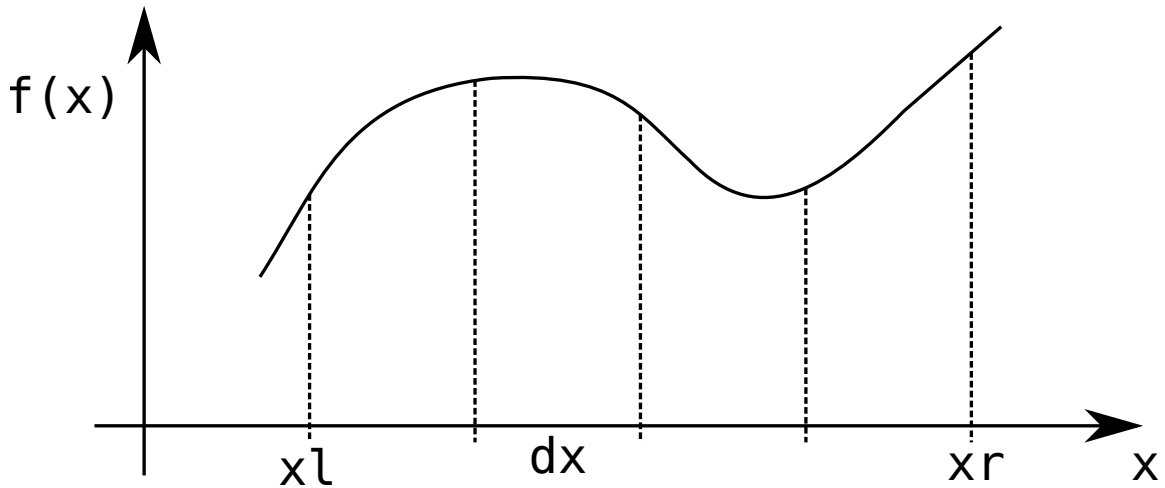
- In Python, everything is an object, including functions
- Like all objects, functions can be assigned (=> Function pointer!)

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

Function "pointer" exercise

Exercise: Function "pointer"

Write a function that performs simple one-dimensional numerical integration, using constant function approximation. The prototype is `def integral(f,xl,xr,dx)`. To test your code, compute $\int_2^4 x^2 dx$ and $\int_0^{3.14} \sin(x) dx$ with step `dx` $= 0.001$; the results should be close to 18.667 and 2 respectively. (Hint: You will 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 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

IPython basics

- Running python code

```
In[3]: run func
```

- All objects from global scope are available in the workspace

```
In[4]: add(4,-3)
```

- Start debugging on error

```
In[5]: pdb on
In[6]: run scoping.py
```

- Reloads modules automatically
- Behavior is configurable through scripts in `/.ipython`

Troubleshooting whitespace issues

Whitespace issues

Python is picky about whitespace. The **Draw Spaces** plugin for the *gedit* editor can help you troubleshoot whitespace issues e.g. when you get some code which has tabs and spaces mixed together.

```
$ sudo apt install gedit-plugins
```

Activatig the **Draw Spaces** plugin

In *gedit* go to `Edit->Preferences->Plugins` check the box next to **Draw Spaces** and click `Close`. Spaces will be indicated by dots and tabs by arrows.

Useful links

Tutorials:

- Google's Python tutorial
- A Byte of Python

Libraries:

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

Further reading

Beginner tutorials:

- E. Matthes, Python Crash Course 2nd Ed, No Starch Press 2019 (project-based)
- A. Scopatz, K.D. Huff, Effective Computation in Physics, O'Reilly 2015 (includes useful material on data visualization, Bash and git)
- M. Lutz, Learning Python 5th Ed., O'Reilly 2013 (very detailed)
- Think Python (free online book)

Homework assignments: Tic-tac-toe

Assignment 1: 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 game state
- 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

Homework assignments: Connect four

Assignment 2: The Connect four game

Write a simple version of the Connect four game for two human players. Here are some hints:

- Use a list of lists for keeping track of the game state
- A handy way for initializing a 6x7 list of lists is the following comprehension [[-1 for j in range(7)] for i in range(6)]
- 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

Homework assignments: Memory

Assignment 3: The Memory game

Write a simple version of the Memory game for two human players. Here are some hints:

- Use a list of lists for keeping track of the score
- Take care in structuring your code: use functions
- On Linux, you can use the `os.system('clear')` call to clear the screen, hiding the previously revealed fields
- You can use numbers and letters as "images"; For a fancier version, you can use "unicode icons", e.g., `print(unichr(0x263a))` prints a smiley
- You have to validate every user selection
- Use docstrings and comments!
- (Optional) Implement an "AI" strategy to enable human players to play against the computer