[Software Development]

Python (Part A)

Davide Balzarotti

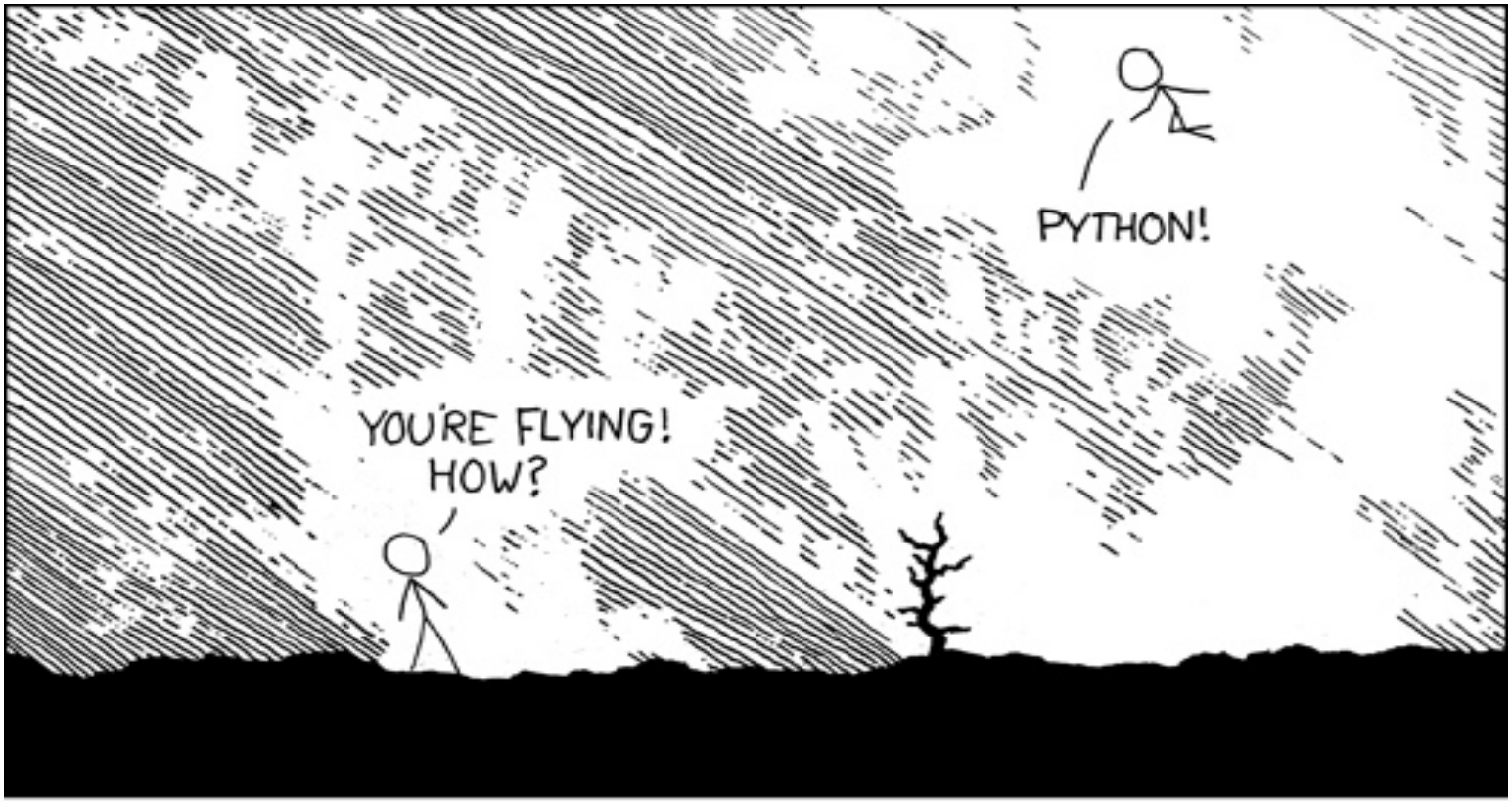
Eurecom – Sophia Antipolis, France

Homework Status

- 83 registered students
  - 41% completed at least one challenge
  - 5 command line ninjas
  - 0 python masters
  - 0 development-fu
- Time to register till the end of the week!!
- 265 Submissions
  - 25% of which were correct

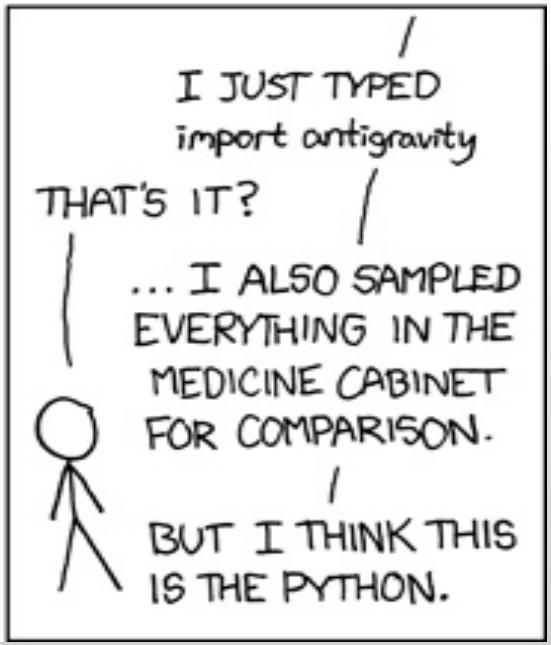
Why Python (a bit of Propaganda)

- Because it is excellent for beginners, yet superb for experts
- Because it is suitable for large projects as well as for everyday tasks
- Because it allows for very short development times
- Because it let you focus on the problem
- Because you can write code nearly as fast as you can type
- Because Perl code seems to be unreasonably difficult to read and grasp even for the authors of the code themselves
- Because I still have to look up how to open a file every time I do it in Java


PYTHON!
YOU'RE FLYING! HOW?


I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"


I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?


I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Language Characteristics

- Automatic memory management
- Small, highly extensible core language with a large standard library
- Support for multiple programming paradigms
  - Object oriented
  - Imperative/Procedural
  - ~ Functional
- Both strongly and dynamically typed
  - The type strictly determines the set of allowed operations
  - The interpreter keeps track of all variables types but rarely uses what it knows to limit variable usage

	Weak	Strong
Static	C	Java
Dynamic	Perl	Python

Running Python

- Python programs are run by passing them to the interpreter or by writing them directly in an interactive session
- The Python interpreter compiles statements to byte code and execute them on a virtual machine
  - Python scripts have the extension "`.py`"
  - Compiled bytecode have the extension "`.pyc`"
- Compilation occurs automatically the first time the interpreter reads code which imports modules, and subsequently upon modification
  - Standalone scripts are recompiled when run directly
- Compilation and execution are transparent to the user

History

- Guido van Rossum started developing the language in 1989
  - The name comes from the television series “Monty Python's Flying Circus”

- First release in 1991
- Python 1.0 was released in January 1994
- Python 2.0 was released in October 2000
- Python 3.0 (or Py3K) was released in December 2008
  - It was intentionally backwards incompatible with python 2.*
  - Many features backported to 2.6 and 2.7
  - 2to3 tool to convert old code to python 3k

Which Python

- It is likely that you will find a mix of Python 2.7 and Python 3K
- You can try some features from Python 3k in Python 2.7 using:

```
from __future__ import print_function, division, ...
from future_builtins imports zip, …
```

- 2.7 is intended to be the last major release in the 2.x series
  - Current version: 2.7.10
- For now we can ignore the version number
  - We will discuss python 3k when we will know the language better

Indentation matter (and no, it's not that bad)

- The language has no analog of the C and Perl brace syntax. Instead, changes in indentation delimit groups of statements
  - A group of instructions cannot be empty.
    To simulate an empty body you can use the `pass` instruction (it's a NOP)
  - The first physical line in a source file must have no indentation
- The end of a physical line marks the end of a statement
  - Unless it is terminated by a (\) character
  - if an open parenthesis ((), brace ({), or bracket ([) has not yet been closed, lines are joined by the compiler
- Best practice
  - 4 spaces per indentation level
  - No hard tabs (set your editor to “retab”)
  - Never mix tabs and spaces

Indentation

```
balzarot > python
Python 2.6.2 (release26-maint, Apr 19 2009, 01:56:41)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 2
>>>  b = 1
  File "<stdin>", line 1
    b = 1
    ^
IndentationError: unexpected indent


>>> a = 2
>>> if (a ==2):
... b =1
  File "<stdin>", line 2
    b =1
IndentationError: expected an indented block
```

Everything is an Object

- All data values in Python are objects, and each object has:
  - A type (that cannot be changed) that determines the supported operations
    - To determine the type of an object: `type(obj)`
      - The type of an object is an object itself!
    - To compare the type of an object: `isinstance(obj, type)`

Everything is an Object

- All data values in Python are objects, and each object has:
  - A type (that cannot be changed) that determines the supported operations
    - To determine the type of an object: `type(obj)`
      - The type of an object is an object itself!
    - To compare the type of an object: `isinstance(obj, type)`
  - An identity (that cannot be changed)
    - To get an integer representation of the object identity: `id(obj)`
    - The `is` operator compares the identity of two objects

Everything is an Object

- All data values in Python are objects, and each object has:
  - A type (that cannot be changed) that determines the supported operations
    - To determine the type of an object: `type(obj)`
      - The type of an object is an object itself!
    - To compare the type of an object: `isinstance(obj, type)`
  - An identity (that cannot be changed)
    - To get an integer representation of the object identity: `id(obj)`
    - The `is` operator compares the identity of two objects
  - A value
    - Some types have a Mutable value
    - Some are Immutable and cannot be modified after creation
    - For immutable types, operations that compute new values may return a reference to any existing object with the same type and value

Everything is an Object

```
balzarot > python
Python 2.5.2 (r252:60911, Jul 22 2009, 15:35:03)
[GCC 4.2.4 (Ubuntu 4.2.4-1ubuntu3)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.

>>> a = 1
>>> id(a)
135720760
>>> a = 2
>>> id(a)
135720748
>>> b = 2
>>> id(b)
135720748
>>> a is b
True
>>> type(a)
<type 'int'>
>>> isinstance(a, int)
True
```

Help

- To enumerate the methods and field of an object
  - `dir(object)`
  - And since everything is an object...
- Most of the objects and functions have associated a documentation text accessible through the `__doc__` field
  - `print zip.__doc__`
- Python online documentation is excellent – always keep a copy within reach

Importing Modules

- `import os` (Good)
  - The content of the module is then accessible using `os.name`
  - Use: `os.popen("ls")`

Importing Modules

- `import os` (Good)
  - The content of the module is then accessible using `os.name`
  - Use: `os.popen("ls")`
- `from os import popen, rmdir` (OK)
  - Now `popen` and `rmdir` are part of the current namespace (collisions may happen)
  - Use: `popen("ls")`

Importing Modules

- `import os` (Good)
  - The content of the module is then accessible using `os.name`
  - Use: `os.popen("ls")`
- `from os import popen, rmdir` (OK)
  - Now `popen` and `rmdir` are part of the current namespace (collisions may happen)
  - Use: `popen("ls")`
- `from os import popen as po`
  - Same as before, but also rename the object
  - Use: `po("ls")`

Importing Modules

- `import os` (Good)
  - The content of the module is then accessible using `os.name`
  - Use: `os.popen("ls")`
- `from os import popen, rmdir` (OK)
  - Now `popen` and `rmdir` are part of the current namespace (collisions may happen)
  - Use: `popen("ls")`
- `from os import popen as po`
  - Same as before, but also rename the object
  - Use: `po("ls")`
- `from os import *` (Really BAD)
  - Import everything in the current namespace → likely a mess

The Null Object

- Is the object returned by functions that terminate without returning a value
- It does not support any special operation
- There is exactly one null object, named None (built-in name)

Builtins Types

- Numbers
- Strings
- Lists
- Tuples
- Sets
- Dictionaries

Numbers

- Numbers in Python are immutable
  (so any operation on a number always produces a new object)
- Operations
  - The usual suspects
    - `12, 3.14, 0xFF, 0377, 3.14e-10, abs(x), 0<x<=5`
  - C-style shifting & masking
    - `1<<16, x&0xff, x|1, ~x, x^y`
  - Integer division truncates (fixed in python 3k)
    - `1/2 -> 0  1./2 -> 0.5`
    - `from __future__ import division`
  - Unlimited precision
    - `2**100 -> 1267650600228229401496703205376L`
  - Complex numbers are supported too
    - `x = 7+5j`

Math

- Set of basic operators
  - `+, -, *, /, %, //, **, +=, -= ...`
  - No `++` and `--`
- The `math` modules provides basic math functionalities (sin, pi, exp..)

```
>>> import math
>>> math.log(2)
0.6931471805599453
```

- External modules to handle more complex functions
  - `numpy` – fast N-dimensional array manipulation
  - `scipy` - user-friendly and efficient numerical routines such as routines for numerical integration and optimization

Strings

Strings are immutable objects that store a sequence of characters

“this ” + 'is a string'	“this is a string”
“fool”*5	“foolfoolfoolfoolfool”
“hello”[2:4]	“ll”
“hello”[-1]	“o”
len(“hello”)	5
“el” in “hello”	True
“\x55”	“U”
r”\x55”	“\\x55”
u"Starsky \u0026 Hutch"	“Starsky & Hutch”
s=”hello”; s[2] = “X”	TypeError: 'str' object does not support item assignment

a = “this is a long string”	“this is a long string”
a.find(“long”)	10
a.replace(“long”, “short”)	“this is a short string”
a.split()	[“this”, “is”, “a”, “long”, “string”]
a.upper()	“THIS IS A LONG STRING”

Slicing

- Operation supported by all sequence types (string, list, tuple)

- Simple: `[start : end]`
  - All elements from start to end-1 (i.e., end is not included)
  - If omitted, the default is from the first to the last (included)
  - end can be larger then the total number of elements in the sequence
  - They can be negative values. In this case the number is interpreted counting backward from the end (e.g., -1 represents the last element)

- Extended: `[start : end : step]`
  - Step specifies the increment between two elements
  - A negative step means moving backward in reverse order

Lists & Tuples

- A list is an ordered, mutable sequence of arbitrary objects (even of different types)

a = [99, "bottles of beer", ["on", "the", "wall"]]	[99, "bottles of beer", ["on", "the", "wall"]]
a[0]	99
a[0] = 98	98
a[1:2] = ["bottles", "of", "beer"]	[99, "bottles", "of", "beer", ["on", "the", "wall"]]
len(a)	3
99 in a	True
del a[0]	["bottles of beer", ["on", "the", "wall"]]
a.index("bottles of beer")	1
a.append("empty")	[99, "bottles of beer", ["on", "the", "wall"], "empty"]
x,y,z = a	x=99; y ="bottles of beer"; z = ["on", "the", "wall"]
a+["old", "empty"]	[99, "bottles of beer", ["on", "the", "wall"], "old", "empty"]

- Tuples are similar, but they are immutable
  - `t = (1, 2, 3, "star")`
  - `t = (1,)`

Dictionaries

- Are associative arrays (or hashtables)
  - The key can be any immutable object (tuples are ok, lists are not)
  - There is no restriction on the values

d = {“foo”: [1,2,3], “bar”: 77 }	{“foo”: [1,2,3], “bar”: 77 }
d[“foo”]	[1,2,3]
d[“eggs”]	KeyError: “eggs”
d.get(“eggs”, 9)	9
“bar” in d	True
d[“bar”] = 0	{“foo”: [1,2,3], “bar”: 0 }
d[“spam”] = “ham”	{“foo”: [1,2,3], “bar”: 0, “spam”:”ham”}
del d[“foo”]	{“bar”: 77}
d.keys()	[“foo”, “bar”]
d.values()	[[1,2,3], 77]
d.items()	[[“foo”, [1,2,3]], [“bar”, 77]]

String Formatting

- `format_string%values`
  - Positional: format_string contains placeholders that are replaced with the corresponding values from the values tuple
    - `"%d is bigger than %d"%(8,2)`
    - Classic placeholders: `%i, %l, %s, %f...`
  - Mapping keys: format_string contains named placeholders that are replaced with the corresponding values from the values dictionary
    - Same placeholders with a name in front
    - `"%(big)d is bigger than %(small)d" % {"small":2, "big":8}`

Sets

- A limitation of list and tuples is that they do not support traditional set operations
- Sets are unordered collection of distinct hashable objects
  - Unordered means that sets do not support indexing, slicing, or other sequence-like behaviors
  - Sets support mathematical operations such as intersection, union, difference, and symmetric difference
    - `set([1,2,3,4]) - set([2,4,9]) -> set([1,3])`

set(iterable)	create a set from an iterable object
set_a <= set_b	Tests weather every element of test_a is in test_b
set_a < set_b	set_a <= set_b AND set_a != set_b
set_a | set_b	union of the two sets
set_a & set_b	intersection
set_a – set_b	difference
set_a ^ set_b	elements in either the set or other but not both

File Objects

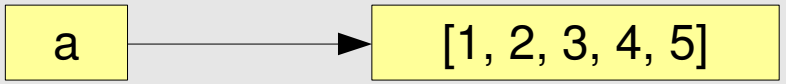
- Reading from files:
  - `f = open("filename")`
  - `f.read()` – read the entire file
  - `f.read(10)` – read 10 bytes
  - `f.readline()` – read a text line from the file
  - `f.readlines()` – return a list of all the file lines
- Writing to files:
  - `f = open("filename","w")`
  - `f.write("hello")`
  - `f.flush()`
  - `close(f)`

Variables

- There is no need to declare them
- But they must be assigned (initialized) before use
  - Use of uninitialized variable raises an exception
- Assignment
  - `a=b=c= 1`
  - `a,b,c = 1,2,3`
  - `a,b = b,a`
- Object assignment manipulates references
  - `x = y` does not make a copy of y
- To make a deep copy
  - ```
    import copy
    x = copy.deepcopy(y)
    ```
    ```

Changing Mutables objects

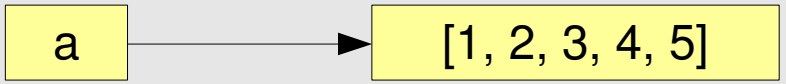
```
> a = [1, 2, 3, 4 , 5]
```



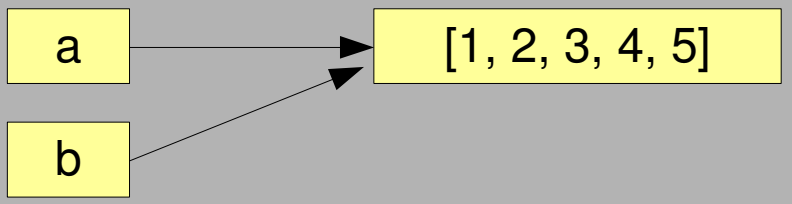

```
> b = a
```

Changing Mutables objects

```
> a = [1, 2, 3, 4 , 5]
```



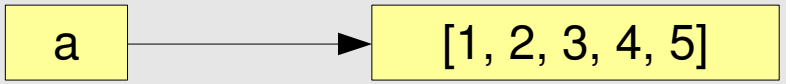

```
> b = a
```



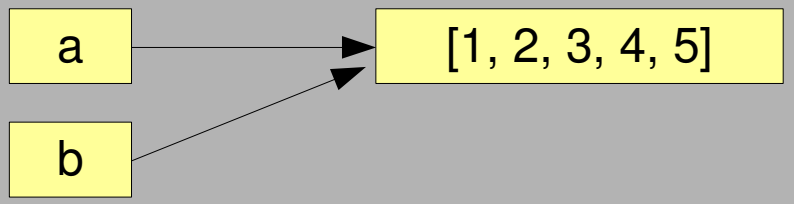

```
> a[2] = 'X'
```

Changing Mutables objects

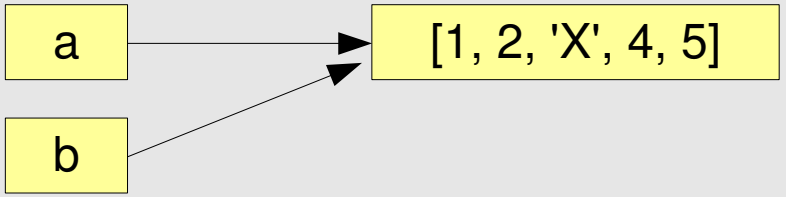
```
> a = [1, 2, 3, 4 , 5]
```




```
> b = a
```



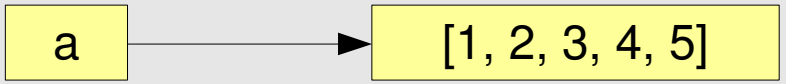

```
> a[2] = 'X'
```



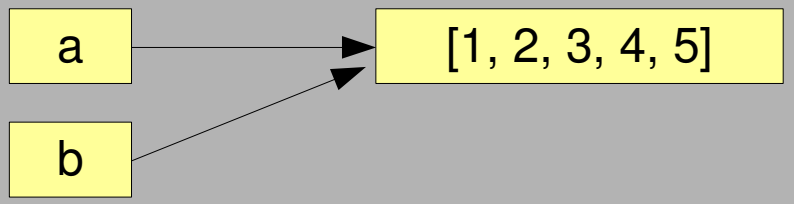

```
> print b
```

Changing Mutables objects

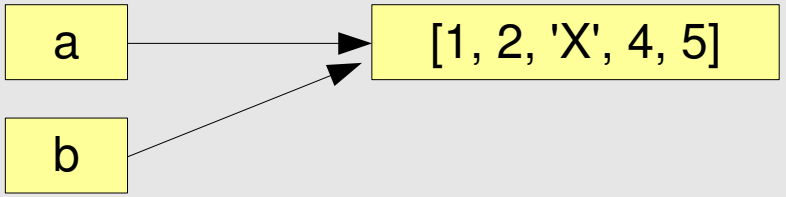
```
> a = [1, 2, 3, 4 , 5]
```



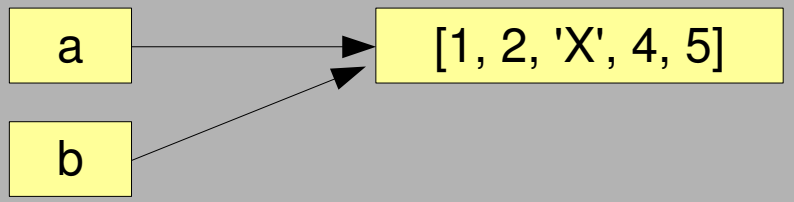

```
> b = a
```




```
> a[2] = 'X'
```

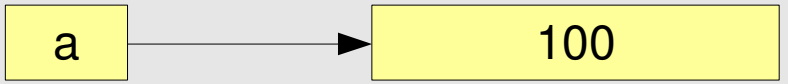



```
> print b
[1, 2, 'X', 4, 5]
```



Changing Immutables objects

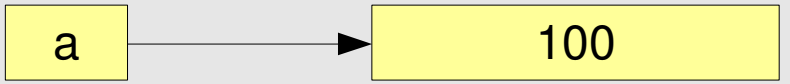
```
> a = 100
```



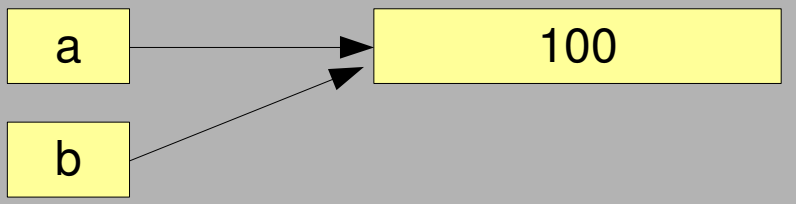

```
> b = a
```

Changing Immutables objects

```
> a = 100
```



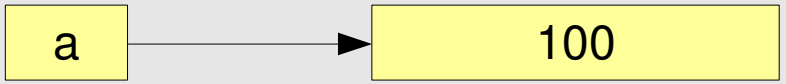

```
> b = a
```



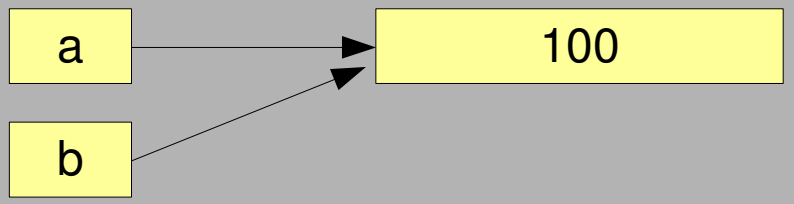

```
> a = 200
```

Changing Immutables objects

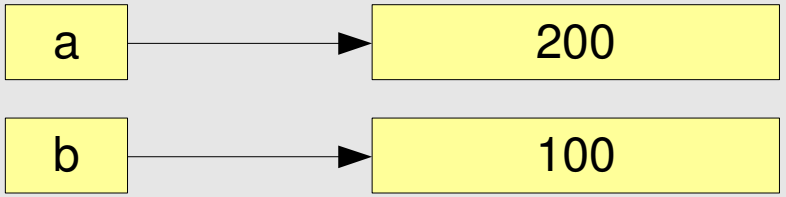
```
> a = 100
```




```
> b = a
```



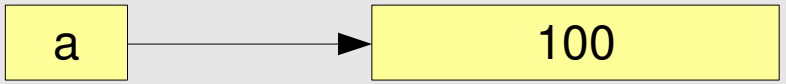

```
> a = 200
```



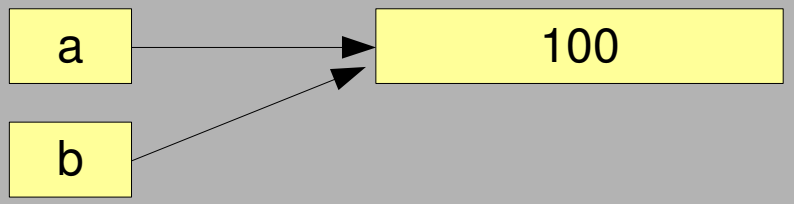

```
> print b
```

Changing Immutables objects

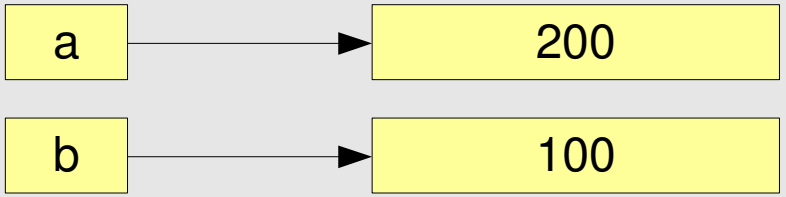
```
> a = 100
```



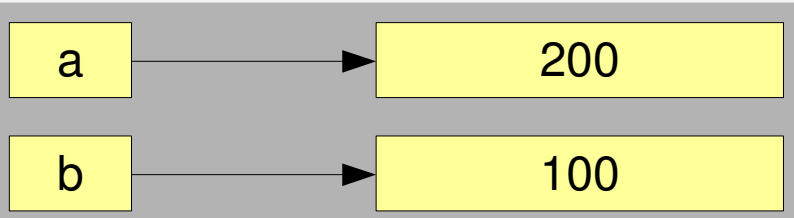

```
> b = a
```




```
> a = 200
```




```
> print b
100
```



Control Structures

IF statement

```
if condition:
   code
elif condition:
   code
else:
   code
```

FOR statement

```
for vars in iterable:
   code
else:
   code
```

WHILE statement

```
while condition:
   code
else:
   code
```


A break statement terminates the loops without executing the else block

A continue statement skips the rest of the block and starts the next iteration

Conditions

- The following values are False:
  - False, None, 0 of any numeric type, empty containers (“”, [], [], {})
- Conditions:
  - `==, <, >, !=, >=, <=`
  - `in` and `not in` check if a value occurs (does not occur) in a sequence
  - `is` and `is not` check if two objects are really the same object
  - Comparisons can be chained
    - `A < B <= C`
  - Comparisons may be combined using the Boolean operators
    - `(A or B) and C`
  - Note that in Python, unlike C, assignment cannot occur inside expressions
    - `if (a=b) -> error`

More on For loops

- Traditional loop over numeric values:

```
for x in range(start,end,step):
```

- Looping and keeping track of the index:

```
for index, value in enumerate(['A', 'B', 'C']):
```

- Looping over two lists at the same time:

```
for a,b in zip(lista, listb):
```

- File objects are iterable too:

```
for line in open("filename"):
```

Fast and memory efficient (does not read all the file in memory)