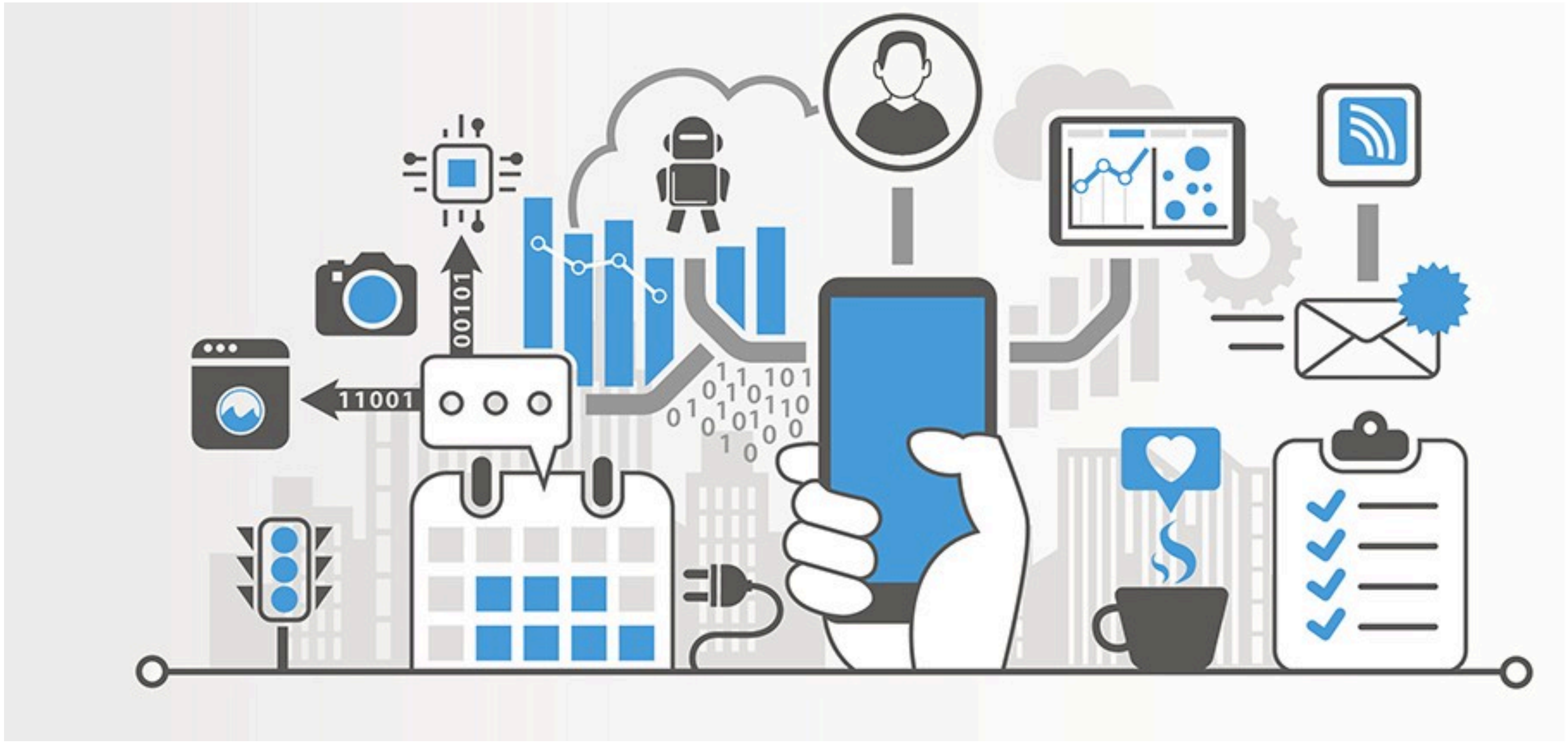
Programming for IoT Applications

Edoardo Patti

Lecture 2



Python programming

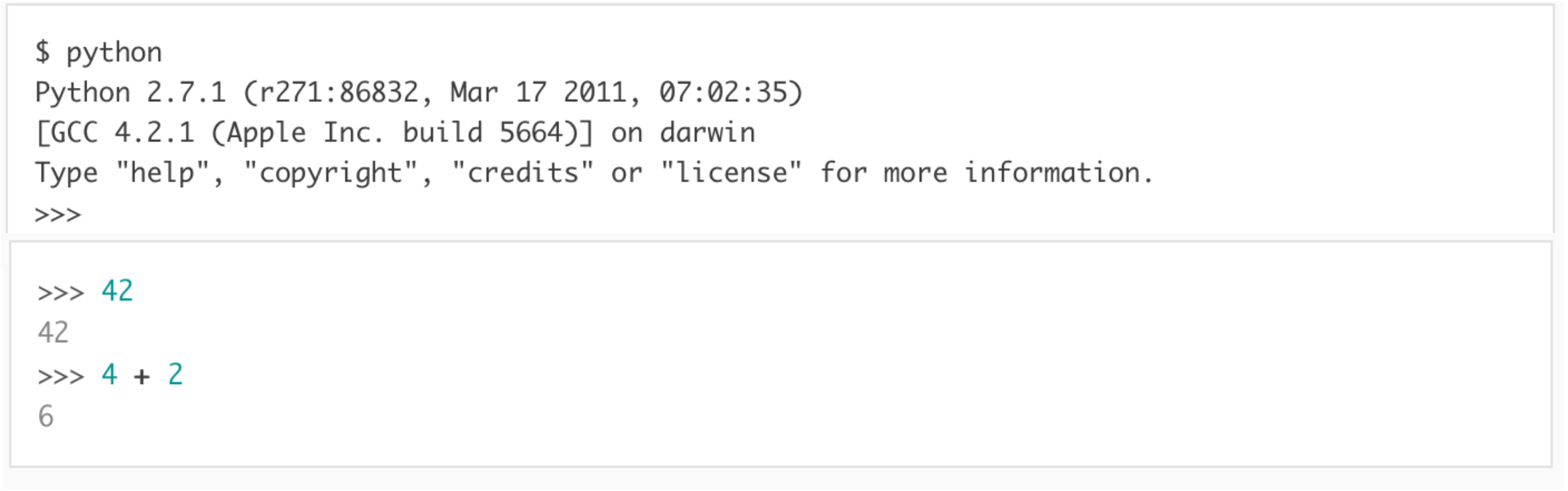
PYTHON BASICS

Python setup

- Download python 3
- Download an editor (e.g. Sublime Text 2 and command line)

Python

- Using python:
  - Interactive interpreter



```
$ python
Python 2.7.1 (r271:86832, Mar 17 2011, 07:02:35)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
>>> 42
42
>>> 4 + 2
6
```

  - Writing python scripts

Code Organization

- Python emphasizes code readability, using **indentation** and **whitespaces to create code blocks**.

In Python:

```
if __name__ == “__main__”:
   print (“hello world”)
```

Code Organization

- Python emphasizes code readability, using **indentation** and **whitespaces to create code blocks**. This makes it simpler than C or Java, where curly braces and keywords are scattered across the code.

In Python:

```
if __name__ == "__main__":
   print ("hello world")
```

In C:

```
int main (int argc, char *argv[])
{
   printf ("hello world");
   return 0;
}
```

Code Organization

- Python emphasizes code readability, using **indentation** and **whitespaces to create code blocks**. This makes it simpler than C or Java, where curly braces and keywords are scattered across the code.

In Python:

```
if __name__ == “__main__”:
   print (“hello world”)
```

In C:

```
int main (int argc, char *argv[])
{
   printf (“hello world”);
   return 0;
}
```

- Python is high-level, which allows programmers to create logic with fewer lines of code.

Variables

- No need to declare

Variables

- No need to declare
- Need to assign (initialize)
    - use of uninitialized variable raises exception

Variables

- No need to declare
- Need to assign (initialize)
    - use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```

Variables

- No need to declare
- Need to assign (initialize)
  - use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```

- ***Everything*** is a "variable":
  - Even functions, classes, modules

Numbers

- Representations
  - `12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5`

Numbers

- Representations
  - `12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5`
- C-style shifting & masking
  - `1<<16, x&0xff, x|1, ~x, x^y #XOR`

Numbers

- Representations
  - `12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5`
- C-style shifting & masking
  - `1<<16, x&0xff, x|1, ~x, x^y #XOR`
- Integer division truncates :-(
  - `1/2 -> 0  # 1./2. -> 0.5, float(1)/2 -> 0.5`
  - Will be fixed in the future

Numbers

- Representations
  - `12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5`
- C-style shifting & masking
  - `1<<16, x&0xff, x|1, ~x, x^y #XOR`
- Integer division truncates :-(
  - `1/2 -> 0 # 1./2. -> 0.5, float(1)/2 -> 0.5`
  - Will be fixed in the future
- Long (arbitrary precision), complex
  - `2L**100 -> 1267650600228229401496703205376L`
    - In Python 2.2 and beyond, 2**100 does the same thing
  - `1j**2 -> (-1+0j)`

Strings

- `"hello"+"world"  "helloworld"     # concatenation`
- `"hello"*3             "hellohellohello" # repetition`
- `"hello"[0]            "h"                   # indexing`
- `"hello"[-1]      "o"                    # (from end)`
- `"hello"[1:4]     "ell"                  # slicing`
- `len("hello")     5                      # size`
- `"hello" < "jello"    1                      # comparison`
- `"e" in "hello"       1                      # search`
- `"escapes: \n etc, \if etc”`

Strings

- `"hello"+"world"  "helloworld"     # concatenation`
- `"hello"*3            "hellohellohello" # repetition`
- `"hello"[0]           "h"                   # indexing`
- `"hello"[-1]      "o"                   # (from end)`
- `"hello"[1:4]     "ell"                 # slicing`
- `len("hello")     5                     # size`
- `"hello" < "jello"    1                     # comparison`
- `"e" in "hello"       1                     # search`
- `"escapes: \n etc, \if etc”`

Difference between single, double, triple quote:

- '...' and "..." are equivalent. If you have an apostrophe in the string, it is easier to use "..." so you do not have to escape the apostrophe. If you have quotes in the string, it is easier to use '...' so you do not have to escape the quotes.

Strings

- `"hello"+"world"  "helloworld"     # concatenation`
- `"hello"*3            "hellohellohello" # repetition`
- `"hello"[0]           "h"                    # indexing`
- `"hello"[-1]      "o"                    # (from end)`
- `"hello"[1:4]     "ell"                  # slicing`
- `len("hello")     5                      # size`
- `"hello" < "jello"    1                      # comparison`
- `"e" in "hello"       1                      # search`
- `"escapes: \n etc, \if etc”`

Difference between single, double, triple quote:

- '...' and "..." are equivalent. If you have an apostrophe in the string, it is easier to use "..." so you do not have to escape the apostrophe. If you have quotes in the string, it is easier to use '...' so you do not have to escape the quotes.

- Triple quotes (both varieties, """ and ''' are permitted) allow the string to contain line breaks. These are commonly used for docstrings (and other multi-line comments) and for embedded snippets of other computer languages such as HTML and SQL.

Lists

- Flexible arrays
  - `a = [99, "bottles of beer", ["on", "the", "wall"]]`

Lists

- Flexible arrays
  - `a = [99, "bottles of beer", ["on", "the", "wall"]]`
- Same operators as for strings
  - `a+b, a*3, a[0], a[-1], a[1:], len(a)`

Lists

- Flexible arrays
  - `a = [99, "bottles of beer", ["on", "the", "wall"]]`
- Same operators as for strings
  - `a+b, a*3, a[0], a[-1], a[1:], len(a)`
- Item and slice assignment
  - `a[0] = 98`

Note that 1:2 means element #2 excluded, so just one element

Lists

- Flexible arrays
  - `a = [99, "bottles of beer", ["on", "the", "wall"]]`

- Same operators as for strings
  - `a+b, a*3, a[0], a[-1], a[1:], len(a)`

- Item and slice assignment
  - `a[0] = 98`
  - `a[1:2] = ["bottles", "of", "beer"]`

```
-> [98, "bottles", "of", "beer", ["on", "the", "wall"]]
```

Using the slicing notation 1:2 istructs the interpreter to insert elements as separate elements in the list.
To insert as a sublist you can do instead:

```
a[1] = ["bottles", "of", "beer"]
```

Lists

- Flexible arrays
  - `a = [99, "bottles of beer", ["on", "the", "wall"]]`
- Same operators as for strings
  - `a+b, a*3, a[0], a[-1], a[1:], len(a)`
- Item and slice assignment
  - `a[0] = 98`
  - `a[1:2] = ["bottles", "of", "beer"]`

    `-> [98, "bottles", "of", "beer", ["on", "the", "wall"]]`
  - `del a[-1]    # -> [98, "bottles", "of", "beer"]`

More List Operations

```
>>> a = range(5)      # [0,1,2,3,4]
>>> a.append(5)       # [0,1,2,3,4,5]
>>> a.pop()         # [0,1,2,3,4]
5
>>> a.insert(0, 42)     # [42,0,1,2,3,4]
>>> a.pop(0)          # [0,1,2,3,4]
5.5
>>> a.reverse()       # [4,3,2,1,0]
>>> a.sort()          # [0,1,2,3,4]
```

Dictionaries

- Hash tables, "associative arrays"
  - `d = {"duck": "eend", "water": "water"}`

Dictionaries

- Hash tables, "associative arrays"
  - `d = {"duck": "eend", "water": "water"}`

- Lookup:
  - `d["duck"] -> "eend"`
  - `d["back"] # raises KeyError exception`

Dictionaries

- Hash tables, "associative arrays"
    - `d = {"duck": "eend", "water": "water"}`
- Lookup:
    - `d["duck"] -> "eend"`
    - `d["back"] # raises KeyError exception`
- Delete, insert, overwrite:
    - `del d["water"] # {"duck": "eend", "back": "rug"}`
    - `d["back"] = "rug" # {"duck": "eend", "back": "rug"}`
    - `d["duck"] = "duik" # {"duck": "duik", "back": "rug"}`

Dictionaries

Given: d = `{"duck": "duik", "back": "rug"}`

- Keys, values, items:
  - `d.keys() -> ["duck", "back"]`
  - `d.values() -> ["duik", "rug"]`
  - `d.items() -> [("duck","duik"), ("back","rug")]`

Dictionaries

Given: d = `{"duck": "duik", "back": "rug"}`

- Keys, values, items:
  - `d.keys() -> ["duck", "back"]`
  - `d.values() -> ["duik", "rug"]`
  - `d.items() -> [("duck","duik"), ("back","rug")]`
- Presence check:
  - `d.has_key("duck") -> 1; d.has_key("spam") -> 0`

Dictionaries

Given: d = `{"duck": "duik", "back": "rug"}`

- Keys, values, items:
  - `d.keys() -> ["duck", "back"]`
  - `d.values() -> ["duik", "rug"]`
  - `d.items() -> [("duck","duik"), ("back","rug")]`
- Presence check:
  - `d.has_key("duck") -> 1; d.has_key("spam") -> 0`
- Values of any type; keys almost any

  -
    ```
    {"name":"Guido", "age":43, ("hello","world"):1,
      42:"yes", "flag": ["red","white","blue"]}
    ```

Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values

Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

Tuples

- Tuples are *immutable* versions of lists

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Tuples

- Tuples are *immutable* versions of lists

- One strange point is the format to make a tuple with one element:

  ‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Tuples Examples

- `key = (lastname, firstname)`
- `point = x, y, z  # parentheses optional`
- `x, y, z = point   # unpack`
- `lastname = key[0]`
- `singleton = (1,)   # trailing comma!!!`
- `empty = ()     # parentheses!`

Reference Semantics

- Assignment manipulates references
    - x = y **does not make a copy** of y
    - x = y makes x **reference** the object y references

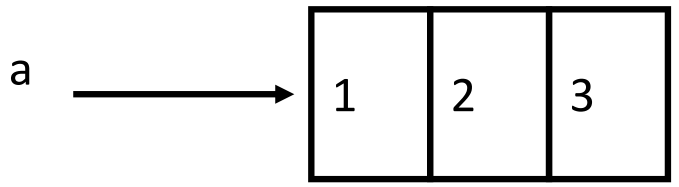
Reference Semantics

- Assignment manipulates references
    - x = y **does not make a copy** of y
    - x = y makes x **reference** the object y references
- Very useful; but beware!
- Example:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print (b)
[1, 2, 3, 4]
```

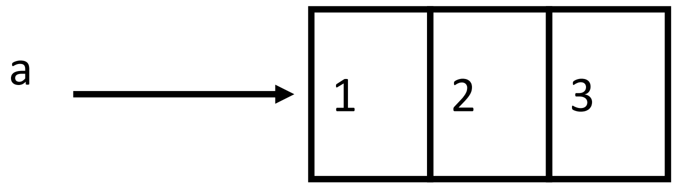
Changing a Shared List

```
a = [1, 2, 3]
```

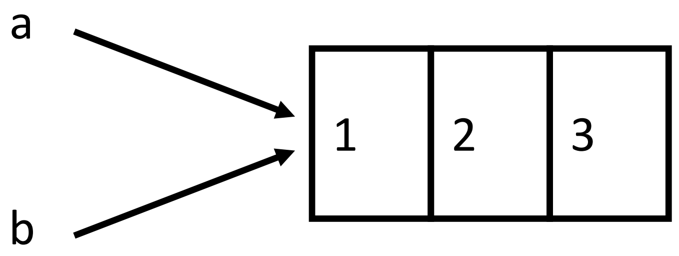


Changing a Shared List

```
a = [1, 2, 3]
```

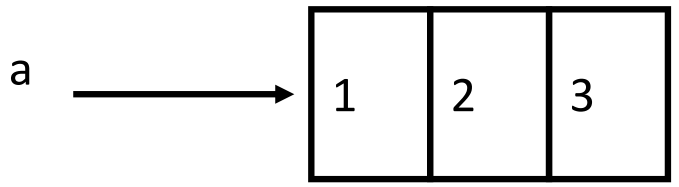


```
b = a
```

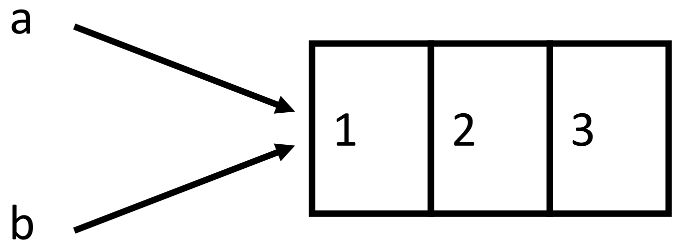


Changing a Shared List

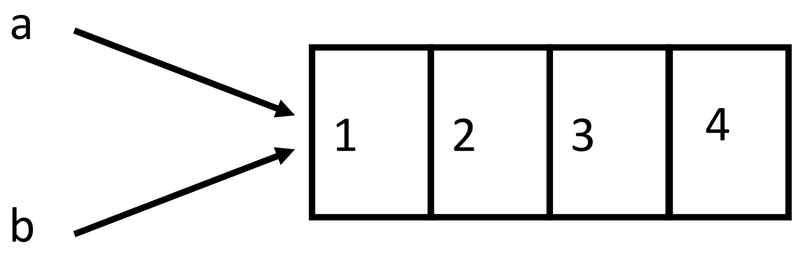
```
a = [1, 2, 3]
```



```
b = a
```

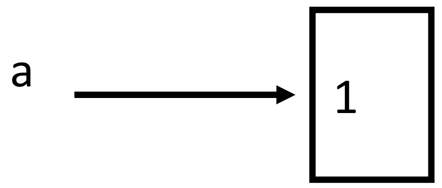


```
a.append(4)
```



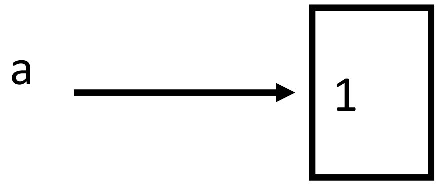
Changing an Integer

```
a = 1
```

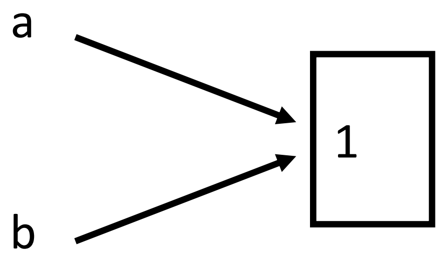


Changing an Integer

```
a = 1
```



```
b = a
```

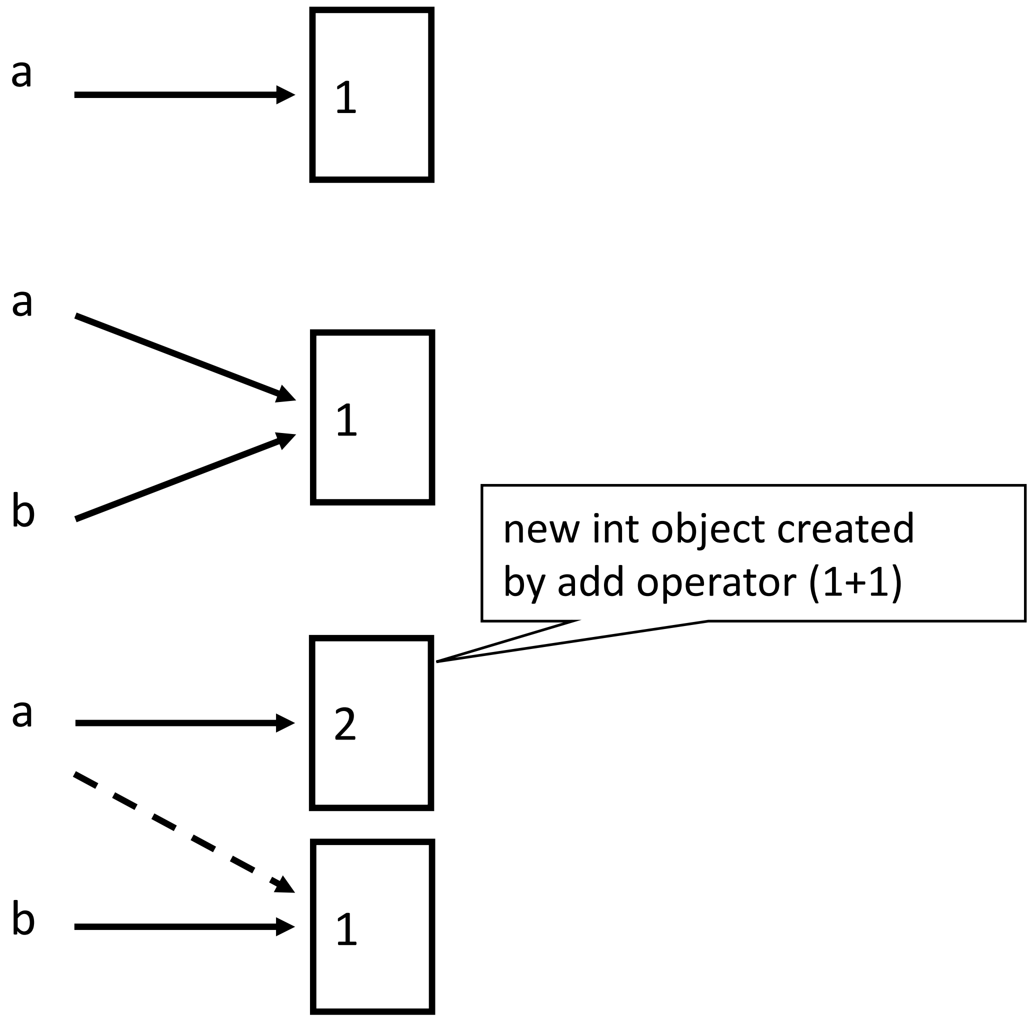


Changing an Integer

```
a = 1

b = a

a = a+1
```

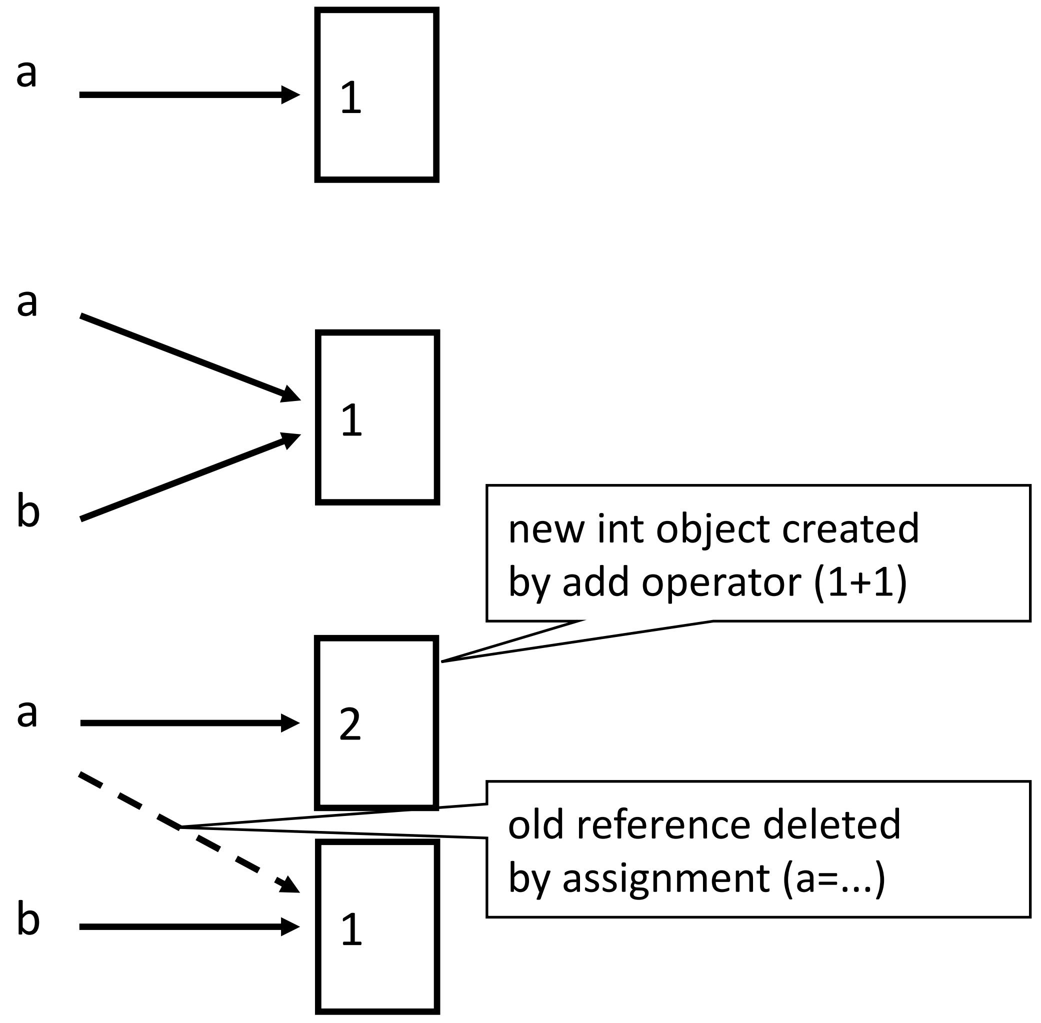


Changing an Integer

a = 1

b = a

a = a+1


a
1
a
1
b
new int object created by add operator (1+1)
a
2
old reference deleted by assignment (a=...)
b
1

In general, any assignment such as a = 2 or b = 2 removes the reference.

Control Structures

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

Control Structures

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

```
while condition:
    statements

for var in sequence:
    statements
```

Control Structures

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

```
while condition:
    statements

for var in sequence:
    statements

break
continue
```

If Statements

```
import math

x = 30
if x <= 15 :
   y = x + 15
elif x <= 30  :
   y = x + 30
else :
   y = x
print (‘y = %f‘ % math.sin(y))
```

In file ifstatement.py

```
>>> import ifstatement
y =  0.999911860107
>>>
```

In interpreter

While Loops

```
x = 1
while x < 10 :
   print (x)
   x = x + 1
```

In whileloop.py

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

In interpreter

For Loops

- iterating through a list of values

forloop1.py

```
for x in [1,7,13,2] :
   print (x)
```

For Loops

- iterating through a list of values

forloop1.py

```
for x in [1,7,13,2] :
    print (x)
```

On the shell

```
~: python forloop1.py
1
7
13
2
```

For Loops

- iterating through a list of values

forloop2.py

```
for x in range(5) :
   print (x)
```

For Loops

- iterating through a list of values

forloop2.py

```
for x in range(5) :
   print (x)
```

On the shell

```
~: python forloop2.py
0
1
2
3
4
```

For Loops

- iterating through a list of values

forloop2.py

```
for x in range(5) :
   print (x)
```

range(N) generates a list of numbers [0,1, …, n-1]

On the shell

```
~: python forloop2.py
0
1
2
3
4
```

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

Grouping Indentation

In Python:

```
for i in range(20):
    if i%3 == 0:
        print (i)
        if i%5 == 0:
            print ("Bingo!”)
    print ("---")
```

Grouping Indentation

In Python:

```python
for i in range(20):
    if i%3 == 0:
        print (i)
        if i%5 == 0:
            print ("Bingo!”)
    print ("---")
```

In C:

```c
for (i = 0; i < 20; i++)
{
    if (i%3 == 0) {
        printf("%d\n", i);
        if (i%5 == 0) {
            printf("Bingo!\n");
  }
      }
      printf("---\n");
}
```

Functions, Procedures

```
def name(arg1, arg2, ...):
    """documentation""" # optional doc string
    statements

return              # from procedure
return expression     # from function
```

Example Function

```
def gcd(a, b):
    "greatest common divisor"
    while a != 0:
        a, b = b%a, a    # parallel assignment
    return b

>>> gcd.__doc__
'greatest common divisor'
>>> gcd(12, 20)
4
```

Function Example

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

functionbasics.py

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

In interpreter

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Functions as Parameters

```
def foo(f, a) :
    return f(a)

def bar(x) :
    return x * x
```

funcasparam.py

```
>>> from funcasparam import *
>>> foo(bar, 3)
9
```

Note that the function foo takes two parameters and applies the first as a function with the second as its parameter

Parameters: Defaults

- Parameters can be assigned default values

```
>>> def foo(x = 3) :
...      print (x)
...
>>> foo()
3
```

Parameters: Defaults

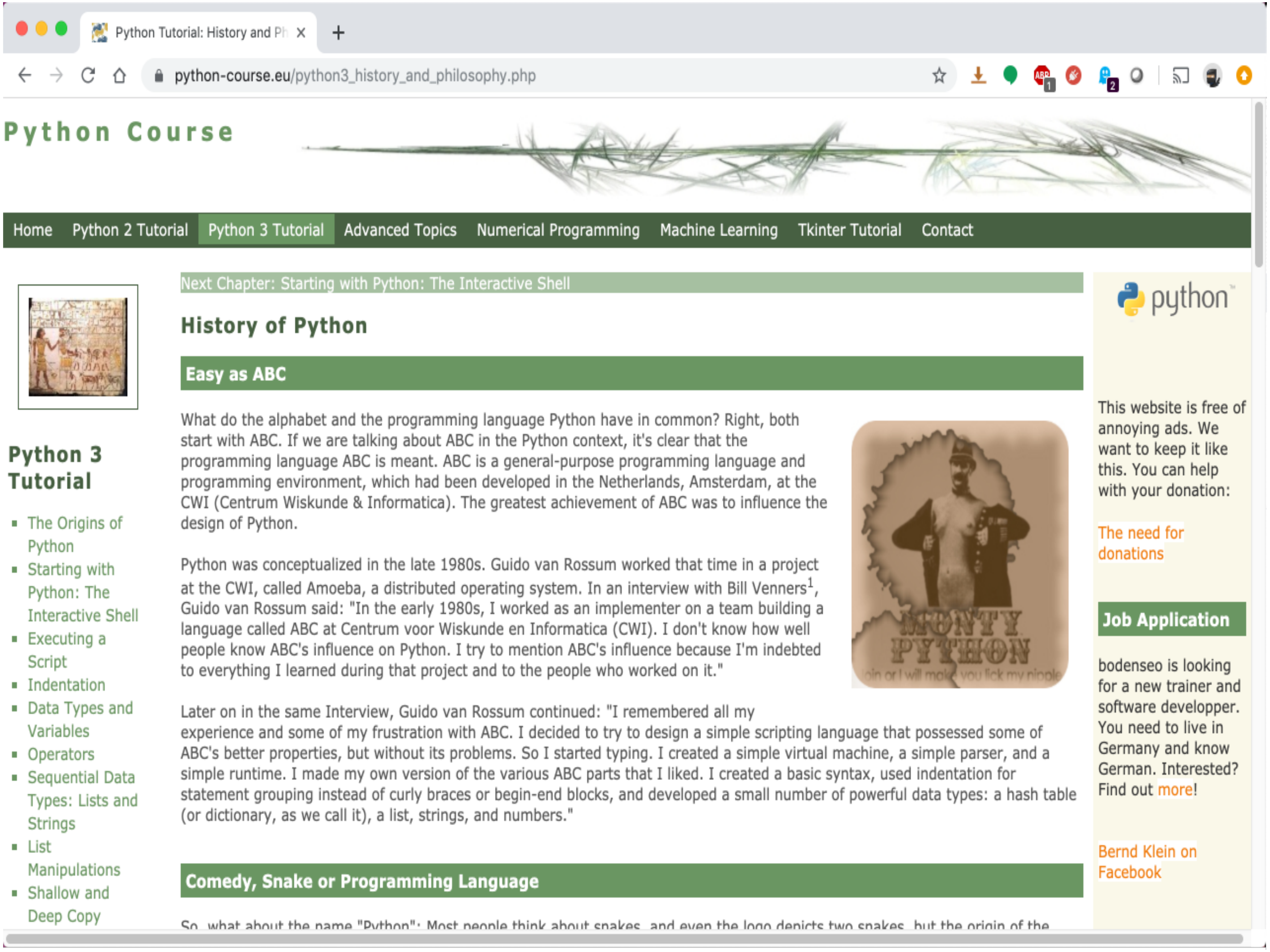
- Parameters can be assigned default values
- They are overridden if a parameter is given for them

```
>>> def foo(x = 3) :
...     print (x)
...
>>> foo()
3
>>> foo(10)
10
```

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default does not limit the type of a parameter

```
>>> def foo(x = 3) :
...     print (x)
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```



Python Course

Next Chapter: Starting with Python: The Interactive Shell

History of Python

Easy as ABC

What do the alphabet and the programming language Python have in common? Right, both start with ABC. If we are talking about ABC in the Python context, it's clear that the programming language ABC is meant. ABC is a general-purpose programming language and programming environment, which had been developed in the Netherlands, Amsterdam, at the CWI (Centrum Wiskunde & Informatica). The greatest achievement of ABC was to influence the design of Python.

Python was conceptualized in the late 1980s. Guido van Rossum worked that time in a project at the CWI, called Amoeba, a distributed operating system. In an interview with Bill Venners[1], Guido van Rossum said: "In the early 1980s, I worked as an implementer on a team building a language called ABC at Centrum voor Wiskunde en Informatica (CWI). I don't know how well people know ABC's influence on Python. I try to mention ABC's influence because I'm indebted to everything I learned during that project and to the people who worked on it."

Later on in the same Interview, Guido van Rossum continued: "I remembered all my experience and some of my frustration with ABC. I decided to try to design a simple scripting language that possessed some of ABC's better properties, but without its problems. So I started typing. I created a simple virtual machine, a simple parser, and a simple runtime. I made my own version of the various ABC parts that I liked. I created a basic syntax, used indentation for statement grouping instead of curly braces or begin-end blocks, and developed a small number of powerful data types: a hash table (or dictionary, as we call it), a list, strings, and numbers."

Comedy, Snake or Programming Language

So, what about the name "Python": Most people think about snakes, and even the logo depicts two snakes, but the origin of the

Python 3 Tutorial

- The Origins of Python
- Starting with Python: The Interactive Shell
- Executing a Script
- Indentation
- Data Types and Variables
- Operators
- Sequential Data Types: Lists and Strings
- List Manipulations
- Shallow and Deep Copy

This website is free of annoying ads. We want to keep it like this. You can help with your donation:

The need for donations

Job Application

bodenseo is looking for a new trainer and software developper. You need to live in Germany and know German. Interested? Find out more!

Bernd Klein on Facebook

References

- https://www.python-course.eu/python3_history_and_philosophy.php