# Introduction to Python

## Getting Started With Python

Rafael Lott
rafaellott@

# Agenda

1. Introduction
    a. History - Where it all began?
    b. Python code - Cool things in Python
    c. Types - (bool, list, tuple, dict, set, ...)
2. Python Features
    a. String manipulation
    b. File I/O
    c. What's new in Python 3.6
3. Packaging and distributions project
    a. Pip/setuptools/

# Introduction

# History

Guido, python creator (wikipedia)

- Invented in the Netherlands, early 90s by Guido van Rossum
- Conceived in the late 1980s and its implementation was started in December 1989
- Named after ‘Monty Python’s Flying Circus’, a surreal sketch comedy series created by and starring the comedy group Monty Python.
- Open sourced from the beginning
  - CPython source code: https://github.com/python/cpython

Monty Python group (BBC Archive)

# Types

Built-ins types:

- bool [boolean]
- str [string] / unicode / bytes
- list (Only value)
- tuple (A list unchangeable)
- dictionary (Key and value)
- set (Same as list, but with unique values)

Show the code:

- Creating list, dict, tuple and set
- Accessing data in list, dict, tuple and set

# Features

- Dynamically Typed

- No need to declare the variable type, as the type of a variable is checked during run-time.
- You you can assign a different type value to a already declared variable.

```
In [1]: myvar = "Hello Python"

In [2]: type(myvar)
Out[2]: str

In [3]: myvar = 1

In [4]: type(myvar)
Out[4]: int
```

- Strongly Typed

```
In [5]: 1 + '2'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-a11677f986f5> in <module>()
----> 1 1 + '2'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- A type of a value doesn't change in unexpected ways.
- Every change of type requires an explicit conversion:

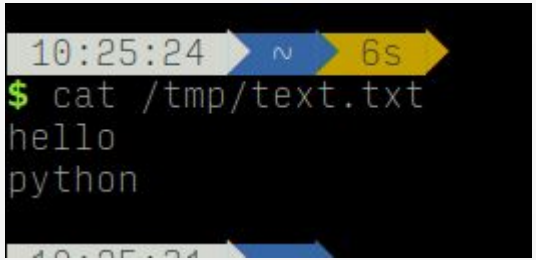
```
>>> age = “10”        # This is a string
>>> age = int(age)    # This is a string casting to integer
```

# Features

- **<u>Write less code and do more (Context Manager)</u>**

**Python** provides an easy way to manage resources: **Context Managers**. The *`with`* keyword is used.

This way you can use and release a resource “automatically”, as the Context Manager will be responsible to close the resource.




```
In [6]: with open('/tmp/text.txt') as f:
   ...:     print(f.read())
   ...:
hello
python
```

# Features

- **String manipulation**
  - %-formatting
    - Exists since the beginning of language.
    - Isn't good to use, as you increase variable, it is hard to read the code.
  - Formatting - str.format()
    - Introduced in Python 2.6
    - Replacement fields are marked by curly braces, than can be empty, numeric or named.
    - Create to use with dictionary
  - F-String - The new way
    - Introduced in Python 3.6
    - Simple Syntax
    - Arbitrary Expressions

# Hands on

- **What is `id()`?**

`id` is a built-in function in Python. It gives us the ability to check the unique identifier of an object. Basically show the memory identifier.

Show the code:

  - Id of integer and string.
  - Cloning list

- **What is `type()`?**

An object has 3 things, id, type, and value. This methods show us the variable type, that can be int, str, list, set, dict, method.

Show the code:

  - Type of integer, string
  - Method type

# Hands on

- **What are mutable objects?**

  - A mutable object is a changeable object and its state can be modified after it is created.

    `list, dict, set`

Show the code

- **What are immutable objects?**

  - An immutable object is an object that is not changeable and its state cannot be modified after it is created.

    `integer, float, string, tuple, bool, frozenset`

# Hands on

- **Why that matters?**

Números, sequências de caracteres e tuplas são imutáveis. Listas, dicionários e conjuntos são mutáveis, assim como a maioria dos novos objetos que você desenvolver com classes.

Show the code:

- Difference of copying list, dict
- Normal copy, shallow copy and deep copy (next slides)

# Hands on

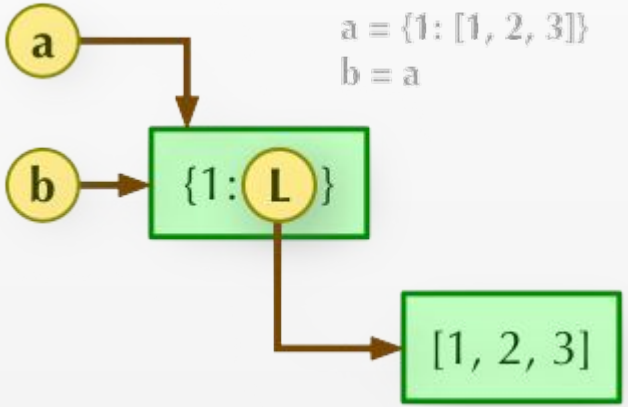
- Normal copy

```
>>> a = "python"
>>> b = a
>>> id(a)
>>> id(b)
```

Reference assignment, Make a and b points to the same object.

Coping dicts and lists:

```
>>> a = {"first_name": "rafael"}
>>> b = a
>>> b["last_name"] = "lott"
>>> id(a)
>>> id(b)
```



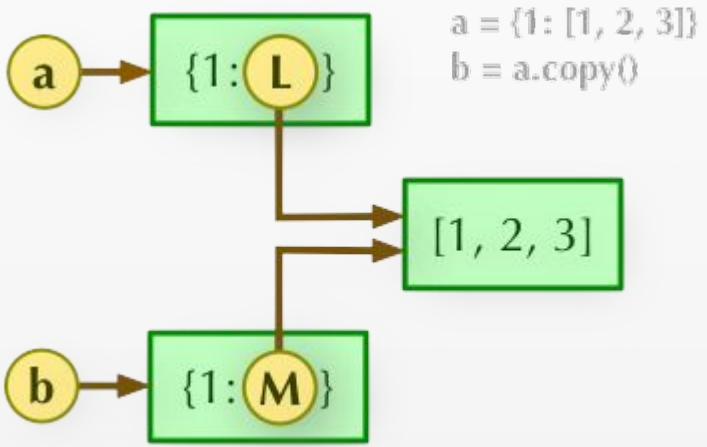
# Hands on

- Shallow copy

Shallow copying, a and b will become two isolated objects, but their contents still share the same reference

Coping dicts and lists:

```
>>> a = {"first_name":
"rafael"}
>>> b = a.copy()
>>> b["last_name"] = "lott"
>>> print(a)
>>> print(b)
>>> id(a)
>>> id(b)
```



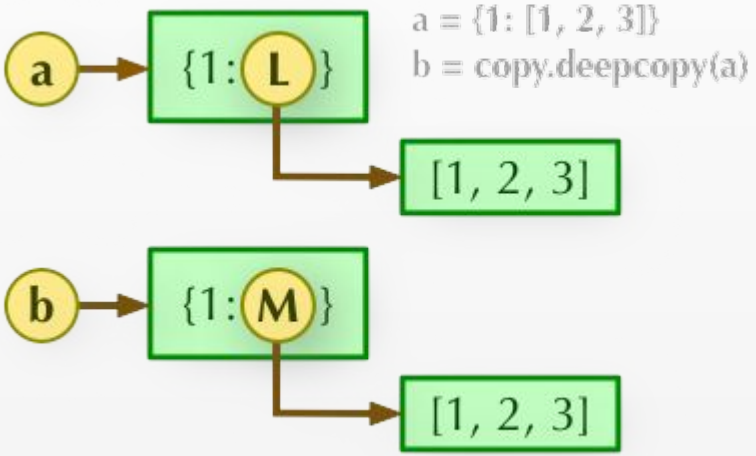
# Hands on

- Deep copy

Deep copying, a and b's structure and content become completely isolated.

Coping dicts and lists:

```
>>> import copy
>>> a = {"first_name": "rafael"}
>>> b = copy.deepcopy(a)
>>> b["last_name"] = "lott"
>>> print(a)
>>> print(b)
>>> id(a)
>>> id(b)
```



# Python Virtual Environment

- Virtualenv

```
$ sudo pip install virtualenv                                  # Install virtualenv on global pip
$ virtualenv <env_name> --python=<python_version>   # Create a new virtualenv in current folder
$ cd <folder_where_installed_virtualenv>                       # Activate virtualenv part 1
$ source <env_name>/bin/activate                               # Activate virtualenv part 2
$ rm <env_name>/                                               # Remove virtualenv by deleting the folder
```

# Python Virtual Environment

- Virtualenvwrapper - Requires virtualenv

**Setup**

```
$ sudo pip install virtualenvwrapper        # Install virtualenvwrapper
```
Add the executable .sh to .bashrc file  # Only require once

**Using**

```
$ mkvirtualenv <env_name> --python=<python_version>      # Create virtualenv
$ rmvirtualenv <env_name>                                # Remove a virtualenv
$ workon <env_name>                                      # Activate the virtualenv
```

# Python Virtual Environment

- Pipenv
  - `$ sudo pip install pipenv` # Install Pipenv

  # Create a new environment, go to folder where is Pipfile
  - `$ pipenv install [--dev] --python <python_version_only_number>`

  - `$ pipenv shell` # Activate environment
  - `$ pipenv sync [--dev]` # Sync virtual environment with Pipfile
  - `$ pipenv --rm` # Remove environment

# Why use Pipenv

Problems that Pipenv Solves

- Dependency Management with requirements.txt

Solution is set all 3rd party dependency

```
click==6.7
Flask==0.12.1
itsdangerous==0.24
Jinja2==2.10
MarkupSafe==1.0
Werkzeug==0.14.1
```

But now you are responsible to update those version.
If a security hole discovered in Werkzeug==0.14.1, you manually have to update it to 0.14.2

`requirements.txt`

```
Flask
```

*pinning* a dependency

`requirements.txt`

```
Flask==0.12.1
```

But *Flask* has a dependency of `Werkzeug==0.14`

The build isn't deterministic

# Why use Pipenv

- Dependency Resolution

Let's say we have two packages

`package_a`
`package_b`

And both requires `package_c`
`package_a` requires `package_c>=1.0`
`package_b` requires `package_c<=2.0`

Let's say `package_c` is in version 2.1.

Step-by-step made by `pip install -r requirements.txt`

- See `package_a` in requirements
- Download `package_a` and see that has dependency of `package_c>=1.0`
- Install `package_c` version 2.1, since fulfill that requirements
- See `package_b` in requirements
- Download `package_a` and see that has dependency of `package_c<=2.0`
- See that `package_c` is installed, but version 2.1
- Breaks install and exit

# Magic Methods

Dunder or magic methods in Python are the methods having two prefix and suffix underscores in the method name. Dunder here means “Double Under (Underscores)”. These are commonly used for operator overloading. Few examples for dunder methods are: __init__, __add__, __len__, __repr__, etc.

**Show the code:**

- Creating a new String class
- Adding __repr__
- Doing str + int, by overlapping __add__()

# Other points

- Creating methods with def t(s=[]) (See notes for code)

- The built-in method `dir()`, show all methods available for the object.

```
>>> dir(str)
```

# Thanks!

# References


Pipenv Guide:
https://realpython.com/pipenv-guide/

Iterator vs Generators
https://nvie.com/posts/iterators-vs-generators/

Magic Methods
https://www.geeksforgeeks.org/dunder-magic-methods-python/

F-Strings
https://realpython.com/python-f-strings/

Context Manager
https://www.geeksforgeeks.org/context-manager-in-python/

Dict as arguments
https://www.geeksforgeeks.org/python-passing-dictionary-as-keyword-arguments/

Mutable and immutable
https://medium.com/datadriveninvestor/mutable-and-immutable-python-2093deeac8d9

# Containers

Containers are data structures holding elements, and that support membership tests. They are data structures that live in memory, and typically hold all their values in memory, too.

Examples:

- list, deque, …
- set, frozensets, …
- dict, defaultdict, OrderedDict, Counter, …
- tuple, namedtuple, …
- str

# Iterables

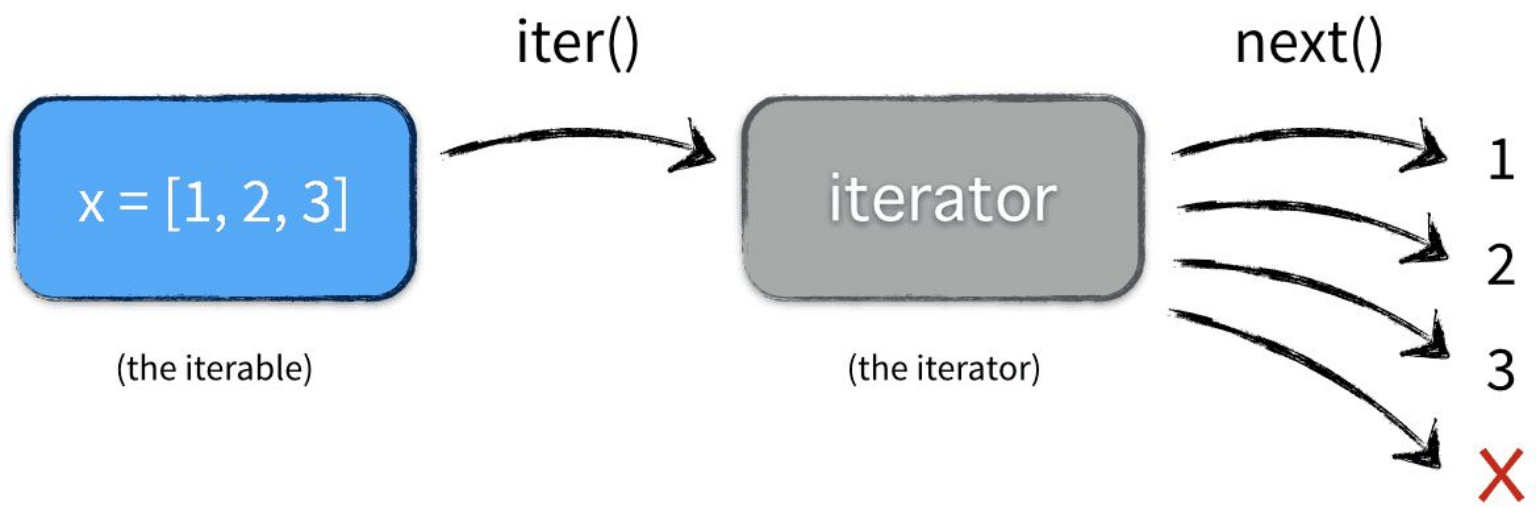
Most containers are also iterable. But many more things are iterable as well. Examples are open files, open sockets, etc. Where containers are typically finite, an iterable may just as well represent an infinite source of data.

**x** is the iterable, while **y** and **z** are two individual instances of an iterator, producing values from the iterable **x**

```
>>> x = [1, 2, 3]
>>> y = iter(x)
>>> z = iter(x)
>>> next(y)
1
>>> next(y)
2
>>> next(z)
1
>>> type(x)
<class 'list'>
>>> type(y)
<class 'list_iterator'>
```

# Iterables

```
x = [1, 2, 3]
for elem in x:
    ...
```



# Iterator

- It's a stateful helper object that will produce the next value when you call next() on it.
- Any object that has a __next__() method is therefore an iterator.
- How it produces a value is irrelevant.

There are countless examples of iterators. All of the `itertools` functions return iterators.

**Infinite sequences:**

```
>>> from itertools import count
>>> counter = count(start=13)
>>> next(counter)
13
>>> next(counter)
14
```

**Infinite sequences:**

```
>>> from itertools import cycle
>>> colors = cycle(['red', 'white', 'blue'])
>>> next(colors)
'red'
>>> next(colors)
'white'
```

Show the code

# Iterator

The code on right, is an example of Iterator, where there is a class that implements the methods __iter__ and __next__.

That is a example of Iterator, where we can call next() on it, and it will print the next value.

```
from itertools import islice

class fib:
  def __init__(self):
    self.prev = 0
    self.curr = 1

  def __iter__(self):
    return self

  def __next__(self):
    value = self.curr
    self.curr += self.prev
    self.prev = value
    return value

f = fib()
list(islice(f, 0, 100))
```

Outputs:
[1, 1, 2, 3, 5]

# Generator

A generator is a special kind of iterator—the elegant kind.

A generator allows you to write iterators much like the Fibonacci sequence iterator example previous, but in an elegant succinct syntax that avoids writing classes with __iter__() and __next__() methods.

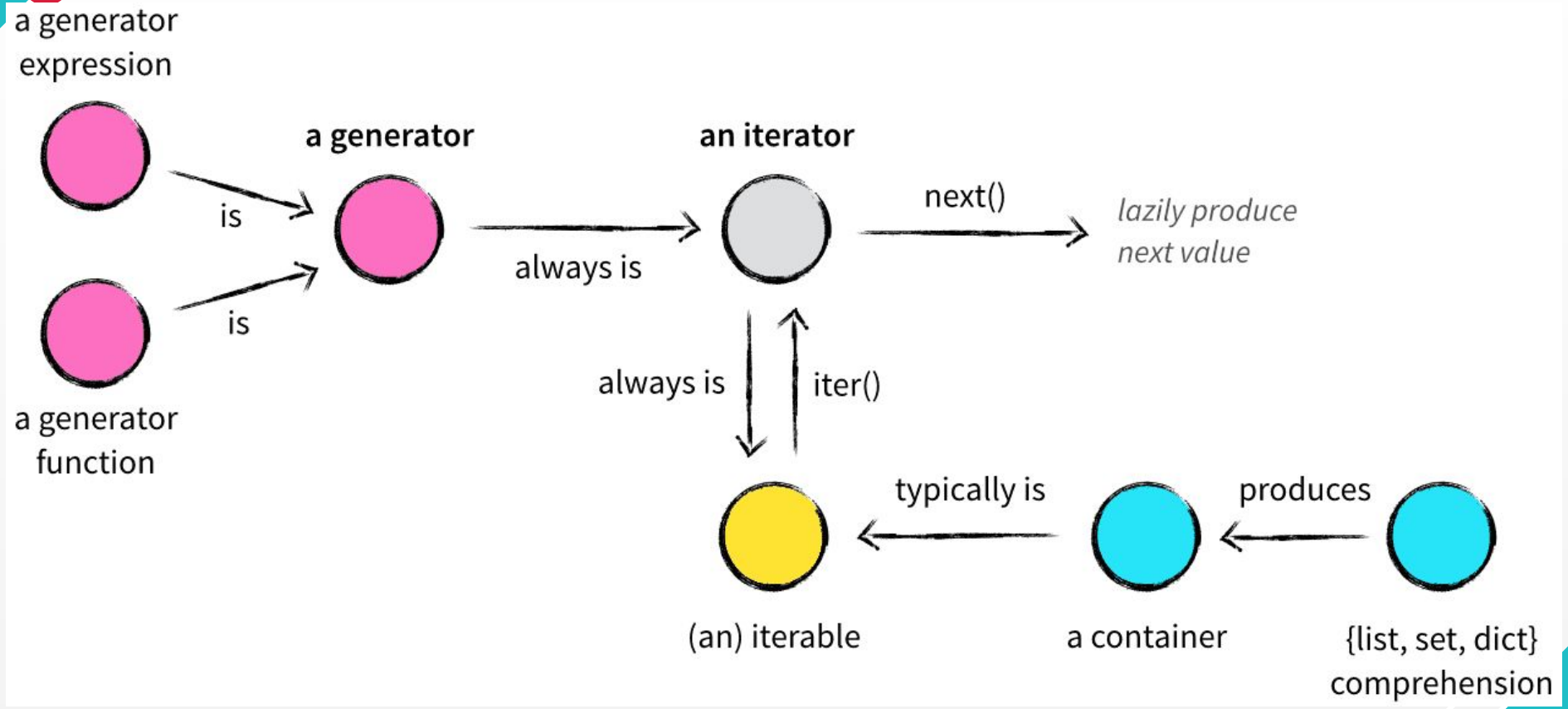
**Let's be explicit:**

- Any generator also is an iterator (not vice versa!);
- Any generator, therefore, is a factory that lazily produces values.

Here is the same Fibonacci sequence factory, but written as a generator

```
>>> def fib():
...     prev, curr = 0, 1
...     while True:
...         yield curr
...         prev, curr = curr,
prev + curr
...
>>> f = fib()
>>> list(islice(f, 0, 10))

  [1, 1, 2, 3, 5, 8, 13,
  21, 34, 55]
```


a generator expression
a generator function
is
is
a generator
always is
an iterator
next()
lazily produce next value
always is
iter()
typically is
produces
(an) iterable
a container
{list, set, dict} comprehension