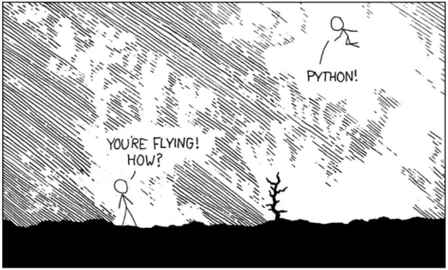
A very informal introduction to Python

Federico Galatolo



Python

- Object Oriented
- Multiple Inheritance
- Dynamically Typed
- Large built-in libraries
- Free (as in freedom) and Open Source

Why Python?

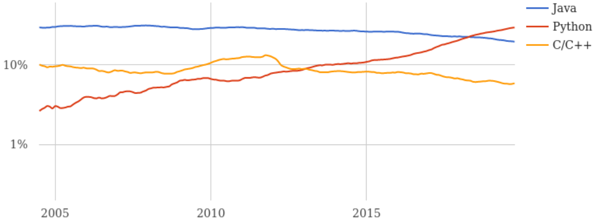


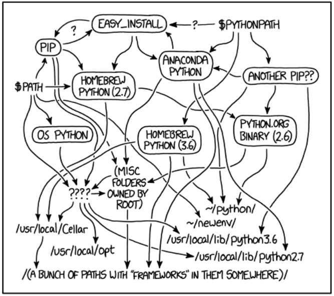
Figure: Languages popularity over time

- High Level
- Portable (Actually write once run everywhere)
- Extendible in C/C++
- Easy to learn and maintain

PyPI: The Python Package Index

- Over 200.000 packages
- Over 400.000 developers
- Portable packages
- Managed by a Non-Profit Organization
- Open Source

Virtual Environments (1)



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

- Method for having project-wide dependencies
- Without it everything become very messy in very little time
- All the dependencies are stored in a folder
- The dependencies folder can be easily snapshotted and retrieved

Virtual Environments (2)

- Create a Virtual Environment with python `X.Y` in folder `env`
  - `virtualenv --python=pythonX.Y env`
- Activate the Virtual Environment
  - `source ./env/bin/activate`
  - `.  ./env/bin/activate`
- Deactivate the Virtual Environment
  - `deactivate`

Managing Packages

- Install package
  - `pip install package`
- Uninstall package
  - `pip uninstall package`
- Snapshot installed packages in `requirements.txt`
  - `pip freeze > requirements.txt`
- Install all packages snapshotted in `requirements.txt`
  - `pip install -r requirements.txt`

Built-in types

Python is **dynamically typed**.
Variables types are determined at **runtime**.
Variables can freely change type during the execution.
In python there are a lot of built-in types, the most notables are:

- Boolean (`bool`)
- Strings (`str`)
- Numbers (`int`, `float`)
- Sequences (`list`, `tuple`)
- Mapping (`dict`)

There is still hope to have a coherent codebase with **dynamic type checking** (last slides)

Variables assignement

As you might expect variables are assigned like this:

```
pi = 3 # problems?
name = "Federico"
```

You can assign multiple variables at once with **iterable unpacking**.

```
pi, name = 3, "Federico"
first, second, third = SomeSequence
```

In python **everything** is stored and passed as **reference** with the only exception of Numbers.

```
a = [1, 2, 3]
b = a
b[0] = 5 # now a = [5, 2, 3]
```

Block syntax: hate it or love it

In python you **don't** surround code blocks with curly brackets
You just use **indentation**

- C like syntax

```
for(int i = 0; i < n; i++){
    int k = i % 3
    if(k == 0){
        // stuff...
    }
}
```

- Python syntax

```
for i in range(0, n):
    k = i % 3
    if k == 0:
        # stuff...
```

Conditional instructions(1)

Simple conditional instruction with the **if** keyword.

```
if someConditions:
    someActions()
    someOtherActions()
```

Python uses **and** and **or** as logical operators instead of **&&** and ||

```
if (C1 and C2) or C3:
    someActions()
    someOtherActions()
```

Conditional instructions(2)

The **else** statement works as you might expect:

```
if Condition:
    someActions()
else:
    someOtherActions()
```

There is no **switch case** statement in python. You can use **if** and **elif**

```
if C1:
    A1()
elif C2:
    A2()
elif C3:
    A3()
else:
    A()
```

Conditional instructions(3): Inline

Inline conditional instructions works as you might expect.
The python syntax is:

```
value if Condition else otherValue
```

For example:

```
pi = 3 if isEngineer else 3.1415
```

While loops

The while loops works as you might expect:

```
while Conditions:
    Stuff()
    otherStuff()
```

There **is no** do-while construct in python.

For loops(1)

In python the for loop **is** a **for each**.

```
for element in elements:
    doStuff(element)
```

`elements` must be and `Iterable`.

You can use `tuple unpacking` in for loops:

```
for x, y in SequenceOfTuples:
    doStuff(x, y)
```

For loops(2): useful built-ins

With **zip()** you can combine **one-by-one** the elements of two or more `iterables`

```
L1 = [1, 2, 3]
L2 = [4, 5, 6]
for x, y in zip(L1, L2):
    print(x, y)
```

enumerate() will return a list of `(index, element)` tuples:

```
names = ["Federico", "Mario", "Giovanni"]
for i, name in enumerate(names):
    print(i, name)
```

For loops(3): list comprehension

The python equivalent for inline for loop it is called list comprehension. It is **not** a for loop but a way for building a list, the syntax is

```
[someOperation(element) for element in elements]
```

For example:

```
squares = [i**2 for i in range(0, N)]
```

Functions(1)

You can define a new function using the `def` keyword

```
def getCircleArea(r):
    return pi*r**2
```

Default arguments values are indicated with =

```
def getCircleArea(r, isEngineer=True):
    pi = 3 if isEngineer else 3.1415
    return pi*r**2
```

Functions(2): Variable positional arguments

You can define a **variable** number of arguments with the * symbol

```
def sumOfSquares(*args):
    squares = [arg**2 for arg in args]
    return sum(squares)
```

Calling the function like this:

```
result = sumOfSquares(1, 2, 3)
```

Functions(3): Sequence dereference

Likewise you can pass sequences as positional arguments in this way:

```
def norm2D(x, y):
    return math.sqrt(x**2 + y**2)

vec = [2, 3]
norm = norm2D(*vec)
```

Functions(4): Keyword arguments

Besides positional arguments python has **keyword** arguments
You can specify that a function uses keyword arguments with the ** symbol.
You **must** use that symbol as last argument

```
def greet(language = "en", **kwargs):
    if language == "it":
        print("Ciao "+kwargs["name"]+" "+kwargs["surname"])
    else:
        print("Hello "+kwargs["name"]+" "+kwargs["surname"])

greet("it", surname="Galatolo", name="Federico")
greet(name="Mario", surname="Cimino")
```

Functions(5): Dictionary dereference

As you might have guessed you can pass a `dict` of keyword arguments using the symbol `**`

```
def greet(language = "en", **kwargs):
    if language == "it":
        print("Ciao "+kwargs["name"]+" "+kwargs["surname"])
    else:
        print("Hello "+kwargs["name"]+" "+kwargs["surname"])

person = dict(name="Federico", surname="Galatolo")
greet("it", **person)
greet(**person)
```

Bonus: Iterators and Generators

`Iterables` are objects that implement the `__iter__()` to get an `Iterator`
`Iterators` are object that implement the `__next__()` to get the next element
`Generators` are a kind of `Iterators` in which the elements are evaluated **on-the-fly**

You can define an inline generator with the `list comprehension` syntax but using the parenthesis instead of the square brackets.

```
squares = (i**2 for i in range(0, N))
```

Functions(6): yield

Using the `yield` statement stead of the return the function returns a `Generator`.
The elements outputted by the generator are the elements yielded by the function.
The yield **does not** stop the execution flow of the function it just yield a value and go on.

```
def counter(i, end):
    while i < end:
        yield i
        i += 1
```

Functions(7): Inline

You can create inline functions using the `lambda` keyword.
The syntax is

```
lambda comma, separated, arguments : expression
```

For example

```
norm2D = lambda x, y: math.sqrt(x**2 + y**2)
```

Functions(8): Decorators

`Decorators` are a way to dynamical add functionalities to a function. Simple `decorators` can be defined as a function returning a generic wrapper function

```
def prettify(func):
    def wrapper(*args, **kwargs):
        print("%"*50)
        func(*args, **kwargs)
        print("%"*50)
    return wrapper

@prettify
def hello():
    print("Hi there!")
```

Classes(1)

In python classes are defined with the `class` keyword.
Class methods are defined with the `def` keyword.
Every method `must have` one argument.
When a method is called from an instance the first argument is a reference to the caller instance.
Conventionally the name of the first argument is `self`.

```
class Person:
    def getName(self):
        return "Federico"
    def greet(self):
        return "Hi! I am "+self.getName()
```

Classes(2): Instance attributes

In python you can create, modify and retrieve instance attributes using the dot (.) selector on the instance reference.
You can create and assign an instance attribute everywhere in a class method.

```
class Person:
    def setName(self, name):
        self.name = name
    def greet(self):
        return "Hi! I am "+self.name
```

Classes(3): Class attributes aka static attributes

You can create class attributes specifying them after the class declaration. You can modify and retrieve class attributes using the dot (.) selector on the class reference

```
class Person:
    greeting = "Hi!"
    def setName(self, name):
        self.name = name
    def greet(self):
        return Person.greeting+" I am "+self.name
```

Classes(4): Class methods aka static methods

You can specify class method as a normal class method without the first argument (it make sense if you think about it)

```
class Person:
    greeting = "Hi!"
    def getGreeting():
        return Person.greeting

g = Person.getGreeting()
```

Classes Bonus: It is all about notation

"Nothing is true, everything is permitted"
Python does not know about static/non-static methods, it is all about notation

```
class Person:
    greeting = "Hi!"
    def setName(self, name):
        self.name = name
    def greet(self):
        return Person.greeting+" I am "+self.name

p = Person()
p.greet() # ok
Person.greet(p) # still ok
```

Classes(5): Visibility

In python there **is no** such thing as a **private** method or attribute. Everything is **public**

The naming convention for “private” methods and attributes is to precede their name with the _ symbol.

```
class Person:
    def setName(self, name):
        self._name = name
    def greet(self):
        return "Hi! I am "+self._name
```

Classes(6): Constructor

In python the construct function is named `__init__` and it is called at object instantiation.
You can specify **one or more arguments**.
As for all the python methods the first one is the object instance reference.

```
class Person:
    def __init__(self, name):
        self._name = name
    def greet(self):
        return "Hi! I am "+self._name
p = Person("Federico")
```

Classes(7): Inheritance

You can extend a base class with another specifying the base class between the parenthesis at class definition

```
class Person:
    def __init__(self, name):
        self._name = name
    def greet(self):
        return "Hi! I am "+self._name

class Student(Person):
    def greet(self):
        return "Leave me alone, I have to study"
```

Classes(8): Data model

The are a lot of built-in functions provided by the base class of all the classes `object`.
Each of which provide a specific behavior, a few are:

- `__len__(self)`
  - Returns the "length" of the object (called by `len()`)
- `__str__(self)`
  - Returns the object as a `string` (called by `str()`)
- `__lt__(self, other)`, `__lt__(self, other)`, `__eq__(self, other)`, ...
  - Called when the object is used in a comparison
- `__getitem__(self, key)`, `__setitem__(self, key, value)`,
  - Called in square brackets access

Classes(9): Inheritance done well

When extending a base class you might need to call its construct or its methods.
In order to get the base class class reference you need to use the super() function

```
class Student(Person):
    def __init__(self, name):
        super(Student).__init__(self)

    def greet(self):
        return "Leave me alone, I have to study"
```

Keep in mind that `super(Class)` returns the base class reference.
And that `super(Class, self)` returns the base class instance.
e.g. `super(Class).__init__(self)` is the same as
`super(Class, self).__init__()`

Classes(10): Inheritance tips and tricks

Keyword arguments are usually preferred over positional ones.
Since `kwargs` is a `dict` each construct should pop out its own keys a forward the others.

```
class Person:
    def __init__(self, **kwargs):
        self.name = kwargs.pop("name")

class Student(Person):
    def __init__(self, **kwargs):
        self.grade = kwargs.pop("grade")
        super(Student, self).__init__(**kwargs)
```

Bonus: Arguments subtleties

Default arguments can be passed as keyword arguments

```
def greet(name="Federico", surname="Galatolo"):
    return "Hi "+name+" "+surname

greet(surname="Cimino", name="Mario")
```

Bonus: Dynamic type checking

In python 3 PEP 484 introduced dynamic type checking syntax to python

```
def greet(name: str, isFriend: bool = False) -> str:
    return "Hi "+name if isFriend else "Hello "+name
```

It is **just a syntax**.
If you want to run dynamic type checking at run time you need to run a **type checker** (for example `mypi`).

PEP stays for Python Enhancement Proposal and they are the RFCs of python

That's all folks!

You can find the slides PDF as well as their LaTeX source code on GitHub.

https://github.com/galatolofederico/python-very-informal-introduction

federico.galatolo@ing.unipi.it
@galatolo
galatolo.me
@galatolofederico