Programming in Python

MMM001 - Data Engineering

Carlos H Brandt
c.brandt@jacobs-university.de

Python: why & what

- As a programming tool: to command computers
  - Data processing
- As a language: to communicate
  - As a broadly spoken language: to communicate broadly

- It's:
  - Interpreted
  - Interactive
  - Multi-paradigm
  - Extensive standard-library
  - Open source and openly developed
  - Extensible

- General-purpose
  - Web-scraping
  - Numerical simulations
- High-level
  - OS abstraction
  - Dynamic typing
  - Garbage-collection

The Zen of Python

- *Readability* is a fundamental aspect of the language
  - Blocks of code are defined by *indentation* levels
- The *pythonic* way of writing code is
  - *Clear*
  - *Concise*
  - *Simple*

The language's informal rules, made up from the decisions and discussions made during the initial development of the language, were caught in a list of aphorisms (by Tim Peters) known as The Zen of Python.

Python Enhancement Proposals

PEPs are the way the language evolves. There are different types of PEPs, covering the most different aspects of the language: from memory management, to standard library inclusion, to documentation standards. Anybody can propose a PEP, which will then be evaluated and discussed by the community to eventually reach an approval or rejection.

PEPs are public and available at:

- https://www.python.org/dev/peps/

Python Package Index

PyPI is a repository for Python software distribution.

- Typically, `pip` will be used to download and install packages from PyPI.
- Anyone can distribute a package through PyPI.

PyPI is available for exploration at:

- https://pypi.org/

The Standard Library

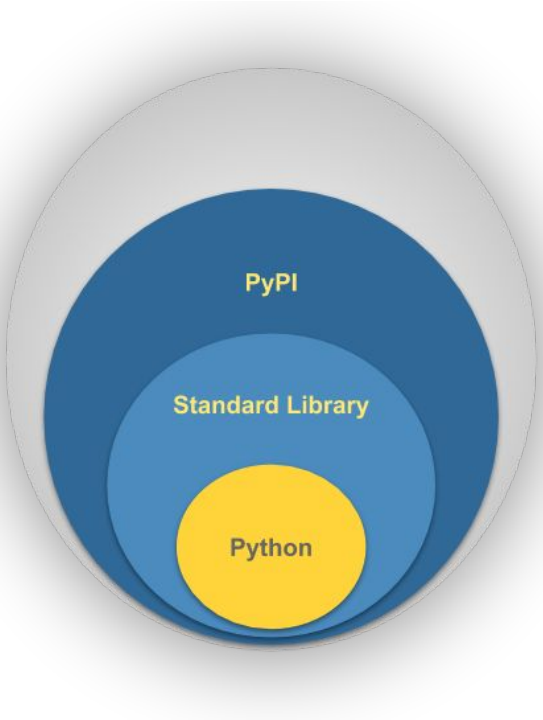
Python comes with an extensive library, distributed together with the interpreter. Among the packages included in the *stdlib* (Standard Library), we will find:

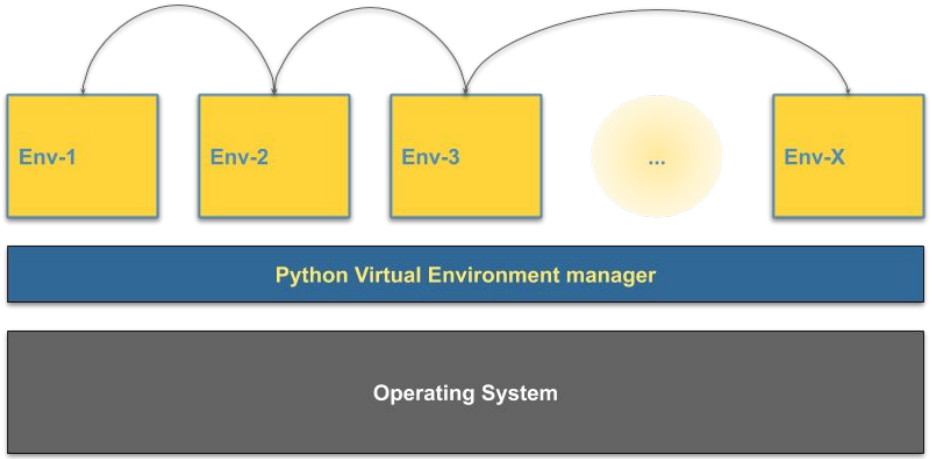
- Text Processing
- Data Types
- Mathematical modules
- File and Directories access
- Generic OS services
- Python Runtime services
- Concurrent Execution
- File Formats
- Structure Markup processing tools
- Internet protocols

Python distributions

- The official python distribution
  - www.python.org
- Anaconda
  - www.anaconda.com
- Canopy
  - www.enthought.com/product/canopy
- ActivePython
  - www.activestate.com/products/activepython

Python environment





The anatomy of Python

```
# A comment.
# In this case, I just would like to say that this is a pseudo-code, it is
# not meant to work but to display what a python source code looks like.
import some_package

a_variable = 'dummy example'

def a_function(arguments):
    """
    With docstring explaining what it does
    """
    out = None
    for arg in arguments:
        # do something
        out = arg
    return out

result = a_function(a_variable)
print(result)
```

Code styling

Code styling is quite a subject in Python, especially if you do pretend to develop collaboratively. The developers/community has actually debated about that long ago (and still do it) and wrote down a set of conventions to serve as guidance. Those conventions are published in PEP-8:

- www.python.org/dev/peps/pep-0008

Nevertheless, independent of the standard you choose for your code, be *consistent* and care about *readability*.

Code styling

Some conventions from PEP-8:

- Indentation: Use 4 whitespaces per indentation level;
- Line length: limit it to 72/79 characters,
  - <= 72 columns for docstrings, ~80 columns for general code.
- In multi-line statements, keep indentation consistent and unambiguous.
- Use blank lines; before/after definitions, and on logical blocks. Space is good.
- Encode source files in UTF-8 (default in most editors).
- import modules/packages at the very top of the file.
- Comments: use it in plain english, they don't need to be short but clear.
- Docstrings: write them in all public functions, classes, modules

Features and Components

- Imperative programming: you give orders.
- Python has name binding.
- Variable names are case-sensitive;
  - characters 'a-z', 'A-Z', '0-9', and '_' are allowed.
- Strings can be defined either with single- or double-quotes.
- Pre-defined mutable and immutable objects are provided.
- Garbage collection cleans zero-referenced objects.
- Variable are local to the scope they were defined (lexical scope)
  - Read-only from higher scopes is done transparently
  - Read-write from higher scopes must be flagged with 'global' or 'nonlocal'

Syllabus

- Lesson 2:
  - Variables
  - Data-types
  - Operators
- Lesson 3:
  - Flow-control
  - Data I/O
- Lesson 4:
  - Loops
  - Functions
  - Classes
- Lesson 5:
  - Memory scope
  - Strings
- Lesson 6:
  - Data strctures
  - File I/O

Syllabus

- Lesson 2:
  - Variables
  - Data types
  - Data structures
  - Operators
- Lesson 3:
  - Flow-control
  - Functions
  - Memory scope
- Lesson 4:
  - Classes
  - Strings
- Lesson 5:
  - Data I/O
  - Formating
- Lesson 6:
  - External libs

Laboratory - Setup Python 3

- Install Anaconda
  - www.anaconda.com
  - Python 3.7
- Install the **ipython** package
  - **pip**
  - **conda**
- Run the python interpreter
  - **python**
  - **ipython**
- Run '**import this**' (without the quotes)

Alternatively, *Miniconda* is a lightweight command-line conda manager:

- docs.conda.io/miniconda.html

Laboratory - Setup virtual environments

Run the following commands

- **conda env list**
  - To verify installed virtual environments
- **conda create -n mmm001 python**
  - To create an environment named 'mmm001'
- **conda create -n tmp_py2 python=2.7**
  - To create a python-2 environment
- **conda create -n tmp_jup python jupyter**
  - To create a python environment with jupyter notebook in it
- **conda env list**
  - To verify installed virtual environments

Laboratory - Setup virtual environments

Run the following commands

- **conda activate mmm001**
- **python --version**
  - Verify the version of python interpreter installed in 'mmm001'
- **conda activate tmp_py2**
- **python --version**
  - Verify the version of python interpreter installed in 'tmp_py2'
- **conda deactivate**
  - To exit from any virtual environment you're in (in this case 'tmp_py2')
- **python --version** # your system probably has a default python installed; this is it

Laboratory - Setup virtual environments

Run the following commands

- **conda activate tmp_jup**
- **jupyter notebook**
  - Run Jupyter notebook
  - Then, quit the notebook
- **conda activate mmm001**
- **jupyter notebook**
  - Run Jupyter notebook; an *error* is expected. Why?
- **conda install jupyter**
- **jupyter notebook**
  - Run Jupyter notebook, now successfully.

Laboratory - Setup virtual environments

Run the following commands

- **conda deactivate**
- **conda env list**
- **conda env remove -n tmp_py2**
- **conda env remove -n tmp_jup**
- **conda env list**
    - To see a list with 'base' (default) and 'mmm001' only

Laboratory - Hello World!

Copy-n-paste the following code to a file called '**hello_world.py**':

```
import sys

def greet_user(name):
    return "Hello {}!".format(name.upper())

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: {} <name>".format(sys.argv[0]))
    else:
        res = greet_user(sys.argv[1])
        print(res)
```

Laboratory - Hello World!

Run the code from the command-line:

- **python hello_world.py**
  - The *Usage* message should come out; as requested, give an argument to the script
- **python hello_world.py world**
  - E.g., "**Hello World!**" is printed

Laboratory - Hello World!

Run the code from the interpreter:

- **python** **#** or, if you have installed, '**ipython**'
  - Runs the Python interpreter
- **import hello_world**
  - Import our 'hello_world.py' module
- **help(hello_world)**
  - No (decent) docs, right? Let's add some...

Add docstring to 'greet_user' function. Repeat previous steps, then:

- **hello_world.greet_user('world')**

Laboratory - #Throwback Turtle

Turtle used to be (probably still is) a graphical way to learn, and play, and see programming. The idea is to give directional indications (orders) to the computer and see the result as a drawing of the instructions.

To (play with) turtle we just have to open an interpreter '**python**'/'**ipython**', and:

- **import turtle as t**
- **t.begin_fill()**
- **t.forward(100)**
- **t.right(135)**
- **t.forward(50)**

Laboratory - #Throwback Turtle

- With the help of the docs page -- https://docs.python.org/3.7/library/turtle.html -- let's try to draw a *triangle* circumscribed in a *square*. Mind the initialization and closing commands:
  - **begin_fill()**
  - **end_fill()**
  - **done()**

```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

Homework - StdLib

- Text processing
  - string, re
- Data types
  - datetime, collections,
- Numeric and Mathematical
  - math, random
- Functional programming
  - itertools, functools
- File and Directory access
  - os.path, glob, shutil
- File formats
  - csv, configparser, json
- Generic OS services
  - os, argparse, logging
- Concurrent execution
  - threading, multiprocessing
- Markup processing
  - html, xml
- Internet protocols
  - urllib, http.server
- Development tools
  - docutils, unittest, pydoc
- Runtime services
  - sys