Introduction to Python

A cookbook

Marcelo Garcia

KAUST Library

Why a Cookbook

- There are good introductory courses to Python available on the Internet, like the "Software Carpentry Foundation," but...
- it can be difficult to find practical information: "Now what??"
- Also to share what I learned with developing code for the library.

The Plan

The presentation is organized as follows:

- Origins of Python.
- Installing Python.
- The "Zen of Python", your guide to *pythonic* code.
- A brief review of the basic elements of Python.
- Reading configuration files.
- Passing environment variables (like passwords) to Python scripts.
- Path separator in Windows and Linux (and MacOS)[1]
- How execute other commands inside your script.
- Working with dates.
- Using templates.

[1] In this text "MacOS" will be treated like Linux.

Origins

- created by Guido Van Rossum in late 1980s for the experimental Amoeba Operating System at "Vrije Universiteit Amsterdam."
  - Interesting note: Guido was working with Andrew S. Tanenbaum, who co-authored the book "MINIX book. Operating systems: Design and Implementation," that would inspire Linus Torvalds to create the Linux kernel, as he wrote in his initial post in 1991: "Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones(...)"
- Version 1.0 released in 1994, version 2.0 released in 2000, and version 3.0 released 2008.
- The name is from the BBC TV show "Monty Python's Flying Circus."

Installing

- For *nix and MacOS most probably will be already installed.
  - On Ubuntu systems, it's a good idea to install the package `python3-venv`, and if you don't plan to use Python 2, then install the package `python-is-python3`, so the command "python" starts `python3` instead of an error that Python 2 is missing.
- For Windows, it's possible to install via the binary provided by Python org, but, it's better to install one of the Python binaries available on Microsoft Store. There are several versions available.

The Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Basic Elements

Python has all the expected elements of a modern programming language: numbers, strings, functions, objects, etc.

```
>>> c = 300_000_000_000 # speed of light in vacum in SI units (m/s).
>>> ff = 5.678 # float number
>>> zz = 3 + 4j # complex number
>>> hello = "hello world!"
>>> hello.upper() # Basic string functions
'HELLO WORLD!'
>>> hello[2:4] # String slicing.
'll'
>>>
>>> if vv > 10:
...     print('more than 10')
... else:
...     print('not more than 10')
...
not more than 10
>>>
```

Basic Elements (cont.)

```
>>> for ii in range(5):
...     print(f"{ii}", end=",")
...
0,1,2,3,4,>>>
>>>
>>> numbers = range(1, 16) # Create a list from 1 to 15
>>> list(numbers)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>>
>>> [ nn*nn for nn in numbers] # List comprehension
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
```

Configuration Files

Create a configuration file to control the behaviour of your script

```
[ACCESSION]
accession_id = 000_000_0000

[BAGGER]
source_dir = C:\Users\joe\Work\boat_trip_pictures
dest_dir = C:\Users\joe\Work\${ACCESSION:accession_id}

[CLAMAV]
(...)
quarantine_days = 30
run_it = no
(...)
```

Reading the Configuration Files

Reading configuration file with `ConfigParser`

```
import configparser
   (...)
   # Creating a configparser object
    config = configparser.ConfigParser(
        interpolation=configparser.ExtendedInterpolation()
    )
    config.read(config_file)
   (...)

    # Reading values
    acc_number = config['ACCESSION']['accession_id']

    # Using a convenience function to read a boolean value
    if config['CLAMAV'].getboolean('run_it'):
        (...)
```

Dot Env File

- Reading environment variables, like data base passwords, without putting them on the code.
- Good for GitHub, but don't forget to add to the `.gitignore`.
- On Linux or MacOS environment variables are easy to use, but on Windows not so much.
- The `.env` file offers a uniform way for all platforms

Consider the file with some credentials

```
me@myserver:~/Work/repo$ cat .env
USERNAME="joe.doe@example.com"
PASSWORD="abc123"
```

Reading the Dot Env File

Load the `.env`, and read the variable from the environment variable:

```
import os
from dotenv import load_dotenv
(...)
    load_dotenv()

    api_passwd = os.environ['MY_API_PW']
```

Path Separator

Python provides libraries to use the correct path separator: `os.path.join` and `pathlib.join`. Example of how Linux and Windows handle path: see the `droid_exec_path`

```
#
# Linux
>>> import os
>>> droid_dir = "/opt/LibraryApps/droid-binary-6.5-bin-with-jre"
>>> droid_bin = "droid.sh"
>>> droid_exec_path = os.path.join(droid_dir, droid_bin)
>>> print(f"droid_exec_path = '{droid_exec_path}'")
droid_exec_path = '/opt/LibraryApps/droid-binary-6.5-bin-with-jre/droid.sh'
#
# Windows
>>> import os
>>> droid_dir = "C:\LibraryApps\droid-binary-6.5-bin-win32-with-jre"
>>> droid_bin = "droid.bat"
>>> droid_exec_path = os.path.join(droid_dir, droid_bin)
>>> print(f"droid_exec_path = '{droid_exec_path}'")
droid_exec_path = 'C:\LibraryApps\droid-binary-6.5-bin-win32-with-jre\droid.bat'
>>>
```

Executing Commands

Executing a program or a script inside your script using the subprocess module

```
import subprocess
(...)
    # 1. Prepare the parameters for the command
    droid_exec_path = os.path.join(droid_config['droid_dir'], droid_config['droid_bin'])
    droid_bag_path = os.path.join(bag_path, "data")

    # 2. Assemble the command line to be executed
    droid_cmd = f"{droid_exec_path} -a {droid_bag_path} --recurse -p {acc_number}.droid"
    (...)

        # 3. Execute the program.
        result = subprocess.run(droid_cmd, stdout=subprocess.PIPE, stderr=subprocess.STDOUT,
            text=True)
        result.check_returncode()
    (...)
```

Explaining

Explaining the parameters in the `subprocess.run`

- "`stdout=subprocess.PIPE`" means to capture the output of the command.
- "`stderr=subprocess.STDOUT`" redirects the output of `stderr` to `stdout`. These are the standard error and standard output devices. They are created automatically by the operating system.
- "`text=True`" convert the output to string (text) instead of a byte sequence.

You will need these parameters to capture the output of your program as a text so you can save to file.

A Complete Example

Let's present a "complete" example of running a script with `subprocess.run`

```
# A very simple script:
# PS C:\Users\garcm0b\Work> cat .\hello_python.py
# print('hello from a python script')

# First we build the command line, and here, how to set the full path to the
# script: home_dir (~) + Work + hello_python.py
>>> import os
>>> import subprocess
>>> ppp = os.path.join(os.path.expanduser('~'), 'Work', 'hello_python.py')
>>> ppp
'C:\\Users\\garcm0b\\Work\\hello_python.py'
>>>
>>> my_cmd = "python.exe " + ppp
>>> subprocess.run(my_cmd, stdout=subprocess.PIPE, stderr=subprocess.STDOUT, text=True)
CompletedProcess(args='python.exe C:\\Users\\garcm0b\\Work\\hello_python.py', returncode=0, stdout='hello from a python script\n')
>>>
# Running the command again, and capturing the result.
>>> result = subprocess.run(my_cmd, stdout=subprocess.PIPE, stderr=subprocess.STDOUT, text=True)
>>> print(result.returncode)
0
>>>
>>> print(result.stdout)
hello from a python script

>>>
```

Working with Dates

Python has objects for date, time, and datetime

```
>>> import datetime as DT
>>> moon_landing = DT.date(1969, 7, 20)
>>> moon_landing.isoformat()
'1969-07-20'
>>> moon_landing.strftime("%a, %d %B %Y")
'Sun, 20 July 1969'
>>> DT.date.today().strftime("%a, %d %B %Y")
'Mon, 20 February 2023'
>>> DT.datetime.now().strftime("%H:%M:%S")
'15:55:33'
>>> dead_line = DT.datetime(2023, 4, 21, 17, 00)
>>> dead_line.strftime("%a, %H:%M, %d %B %Y")
'Fri, 17:00, 21 April 2023'
>>>
```

Manipulating Dates

```
>>> expire_date = DT.date.fromisoformat('2023-04-24')
>>> today = DT.date.today()
>>> good_days = (expire_date - today).days
>>> if good_days > 0:
...     print(f"It's valid for '{good_days}' {good_days == 1 and 'day' or 'days'}")
...
It's valid for '63' days
>>> in_3weeks = today + DT.timedelta(weeks=3)
>>> in_3weeks.strftime("%a, %d %B %Y")
'Mon, 13 March 2023'
>>> today.strftime("%a, %d %B %Y")
'Mon, 20 February 2023'
>>>
>>> av_run_date = DT.datetime.today().strftime("%Y%m%d")
>>> av_log_file = f"av_scan_{av_run_date}.txt"
>>> print(av_log_file)
av_scan_20230220.txt
>>>
```

Templates with Jinja2

- Jinja2 is a very powerful template system available in Python.
- What is a template system?

```
>>> import jinja2 as J2
>>> environment = J2.Environment()
>>> template = environment.from_string("Hi {{ name }}, how are you?")
>>> template.render(name="Joe")
'Hi Joe, how are you?'
>>>
```

How does Jinja2 work

- A list of values as input source.

```
(venv) PS C:\Users\garcm0b\Work\Introduction_Python\examples\templates\in> cat .\hosts.txt
atta, 192.168.56.10
flik, 192.168.56.11
hopper, 192.168.56.12
```

- A template of the output, like a text, or a configuration file.

```
(venv) PS C:\Users\garcm0b\Work\Introduction_Python\examples\templates\temps> cat .\hostconfig.j2
/**
* Simple Icinga host configuration file
*/

object Host "{{ hostname }}" {
    check_command = "hostalive"
    address = "{{ ip_addr }}"
}
(venv) PS C:\Users\garcm0b\Work\Introduction_Python\examples\templates\temps>
```

How does Jinja2 work (cont.)

- A Python script

```
(venv) PS C:\Users\garcm0b\Work\Introduction_Python\examples\templates> cat .\hostsconfig.py
import jinja2 as J2

def main():
    environment = J2.Environment(loader=J2.FileSystemLoader("temps/"))
    template = environment.get_template("hostconfig.j2")
    with open("in/hosts.txt", "r") as fin:
        text = fin.readlines()
    hosts = [line.strip().split(",") for line in text]
    for host in hosts:
        hostname,ip_addr = host[0], host[1]
        content = template.render(hostname=hostname, ip_addr=ip_addr)
        with open(f"out/{hostname}.cfg", "w") as fout:
            fout.write(content)
if __name__ == "__main__":
    main()
(venv) PS C:\Users\garcm0b\Work\Introduction_Python\examples\templates>
```

How does Jinja2 work (cont.)

- Run the script

```
(venv) PS C:\Users\garcm0b\Work\Introduction_Python\examples\templates> python .\hostsconfig.py
```

- Check the output

```
(venv) PS C:\Users\garcm0b\Work\Introduction_Python\examples\templates\out> cat .\atta.cfg
/**
* Simple Icinga host configuration file
*/

object Host "atta" {
    check_command = "hostalive"
    address = " 192.168.56.10"
}
```