Lab 03 - Python development

Objectives

- Using virtual environments and **pip**
- Debugging scripts
- Understanding public APIs

Contents

Tasks

Proof of Work

Today we're picking up where we left off last time. By now you should already know the basics of working with *Python*. Developing a project in *Python* however, requires more than interacting with a shell or editing some scripts. In this lab, you will (hopefully) learn to manage isolated virtual environments and debug errors in your scripts. But probably most important, you will learn to consult API documentations.

As a more tangible goal, you will have to write your very own discord [https://discord.com/] music bot! Exciting stuff, right? As always, in addition to the script itself, remember to put together a *.pdf* explaining your approach to solving the problem. Once finished, upload both to the appropriate moodle [https://curs.upb.ro/2021/course/view.php?id=5793] assignment.

Tasks

01. [40p] Python environment

Different *Python* projects usually employ not only different modules, but even different versions of the same module. The reason is that module developers push out new versions very rapidly and project maintainers may be slow on the uptake. Ideally, updates to any module (or package) should not break tools using previous implementations. Each update should add bug fixes and extend the API with new features, but not change it! However, programs are sometime built around some anomalous behavior of an API and bug fixes actually break said program.

Click GIF to maximize.

As you work on different *Python* projects, you may need different versions of the same module. Probably even a module that you already have installed system-wide. For this, we use virtual environments [https://docs.python.org/3/library/venv.html]. These environments allow you to install specific module versions in a local directory and alters your shell's environment to prioritize using them. Switching between environments can be as easy as **source**ing another setup script.

The problem with virtual environments is that they don't mesh well with **apt**. In stead of **apt**, we will use a *Python* module manager called **pip3**. Our suggestion is to use **pip** only in virtual environments. Yes, it can also install modules system-wide, but most modules can be found as **apt** packages anyway. Generally, it is not a good idea to mix package managers!

[10p] Task A - Dependency installation

First things first, we need **python3**, the **venv** module, and **pip**. These, we can get with **apt**

```
$ sudo apt install python3 python3-venv python3-pip
```

[10p] Task B - Creating the environment

Assuming that you are in your project's root directory already, we can set up the virtual environment:

```
$ python3 -m venv .venv
```

The `-m` flag specifies to the *Python* interpreter a module. **python3** searches its known install paths for said module (in this case, **venv**) and runs it as a script. `.venv` is the script's argument and represents the name of the storage directory. Take a look at its internal structure:

```
$ tree -L 3 .venv
```

Notice that in *.venv/bin/* we have both binaries and activation scripts. These scripts, when sourced, will force the current shell to prioritize using these. The modules you install will be placed in *.venv/lib/python3.*/site-packages/.* Try to activate your environment now. Once active, you will have access to the **deactivate** command that will restore your previous environment state:

```
$ source .venv/bin/activate
$ deactivate
```

If you are still using the setup from the first lab, you may get an ugly `(.venv)` prompt that looks nothing like that in the GIF above. Add this to your *.zshrc*:

```
VIRTUAL_ENV_DISABLE_PROMPT="yes"
```

The display function depends on your selected theme. For *agnoster*, you can fiddle with the **prompt_virtualenv()** function in the *agnoster.zsh-theme* source file.

[10p] Task C - Fetching modules with pip

Same as **apt**, **pip** used to have a search function for modules. Unfortunately, they removed this feature due to a high number of queries. Now, to search for modules, you will need to use the web interface [https://pypi.org/project/pip/].

Let us install the modules needed for this laboratory. After that, let us also check the versions of all modules (some will be added implicitly). Can you also find their installation path in the *.venv/* directory?

```
$ sudo apt install libffi-dev libnacl-dev python3-dev

$ pip3 install 'py-cord[voice]' PyNaCl ipython
$ pip3 list
```

So, what are these, exactly?

- **py-cord[voice]**: a *Python* module that interacts with discord [https://discord.com/]. The previous discord.py [https://discordpy.readthedocs.io/en/stable/] project has been discontinued and pycord [https://docs.pycord.dev/en/master/index.html] remains the best alternative. We'll be using this in the following exercise to write a discord bot.
- **PyNaCl**: wrapper of the NaCl [https://nacl.cr.yp.to/] library. This library offers a high-level interface for networking and cryptographic operations. Note that the library must be installed using **apt** and that which we install with **pip** is only a wrapper.
- **ipython**: a more interactive *Python*. See the next task for some details, but there's really not much to it.

This list of dependencies can be exported (together with the exact version) in a way that allows another user to install the exact same modules in their own virtual environments. The file holding this information in named by convention *requirements.txt*:

```
$ pip3 freeze > requirements.txt
$ pip3 install -r requirements.txt
```

[10p] Task D - Testing that it works, with ipython

In the previous lab, you used the **python3** interpreter in interactive mode. Now, we upgrade to **ipython**. This new interpreter offers an enhanced user experience by means of color coding, tab completion, multi-line editing, etc. For scripting purposes, **python3** should remain your interpreter of choice. But for prototyping, we suggest using this. For now, let's see if the **discord** module in the **py-cord[voice]** package is available to import.

```
$ ipython
Python 3.9.7 (default, Oct 10 2021, 15:13:22)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.29.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import discord

In [2]: discord?

In [3]: discord??

In [4]: help(discord)
```

Note how in stead of **help**, in **ipython** we can `?` or `??` to something to access a brief / more complete documentation.

If you can't import the **discord** module, try to source the activation script again after installing the packages with **pip**. Some versions of **venv** / **pip** might act up.

02. [60p] Discord bot

Let's move on to the good part. In this exercise, you will write your very own **discord** bot. In case you've been living under a rock, **discord** is pretty much like **teamspeak** from back in the day, but you can create ad-hoc servers for you and your friends. These servers have both text and voice channels and can be joined via a referral link.

Each account (provisional or fully registered) is allocated a token. This token is a string of characters that uniquely identifies you to the **discord** servers. If you use this token with **pycord** (see the Client class [https://docs.pycord.dev/en/stable/api/clients.html]), you will be able to automate a human account. This is fine, if you want to write a terminal **discord** client for yourself. However, as a developer, you are also able to create bot applications under your account. These bots belong to you, but are issued separate tokens.

While human accounts are added to a server by receiving an invite link, bot owners give out join links to server admins. Once the admin approves the join request for your bot (reviewing a list of permissions that the bot needs to function), the bot will appear as a normal user in that respective server. When you run your *Python* script and authenticate yourselves with the bot's token, the bot will come online. When you stop your script and the connection is lost, the bot goes offline in **discord**. Simple, right?

To get you started, we prepared a code skeleton for you:

bot-skel.py

```
#!./.venv/bin/python

import discord      # base discord module
```

```
import code         # code.interact
import os           # environment variables
import inspect      # call stack inspection
import random       # dumb random number generator

from discord.ext import commands    # Bot class and utils

################################################################################
############################### HELPER FUNCTIONS ###############################
################################################################################

# log_msg - fancy print
#   @msg   : string to print
#   @level : log level from {'debug', 'info', 'warning', 'error'}
def log_msg(msg: str, level: str):
    # user selectable display config (prompt symbol, color)
    dsp_sel = {
        'debug'   : ('\033[34m', '-'),
        'info'    : ('\033[32m', '*'),
        'warning' : ('\033[33m', '?'),
        'error'   : ('\033[31m', '!'),
    }

    # internal ansi codes
    _extra_ansi = {
        'critical' : '\033[35m',
        'bold'     : '\033[1m',
        'unbold'   : '\033[2m',
        'clear'    : '\033[0m',
    }

    # get information about call site
    caller = inspect.stack()[1]

    # input sanity check
    if level not in dsp_sel:
        print('%s%s[@] %s:%d %sBad log level: "%s"%s' % \
            (_extra_ansi['critical'], _extra_ansi['bold'],
             caller.function, caller.lineno,
             _extra_ansi['unbold'], level, _extra_ansi['clear']))
        return

    # print the damn message already
    print('%s%s[%s] %s:%d %s%s%s' % \
        (_extra_ansi['bold'], *dsp_sel[level],
         caller.function, caller.lineno,
         _extra_ansi['unbold'], msg, _extra_ansi['clear']))

################################################################################
############################### BOT IMPLEMENTATION #############################
################################################################################

# bot instantiation
intents = discord.Intents.all()
bot = commands.Bot(command_prefix='!', intents=intents)
```

```
# on_ready - called after connection to server is established
@bot.event
async def on_ready():
    log_msg('logged on as <%s>' % bot.user, 'info')

# on_message - called when a new message is posted to the server
#   @msg : discord.message.Message
@bot.event
async def on_message(msg):
    # filter out our own messages
    if msg.author == bot.user:
        return

    log_msg('message from <%s>: "%s"' % (msg.author, msg.content), 'debug')

    # overriding the default on_message handler blocks commands from executing
    # manually call the bot's command processor on given message
    await bot.process_commands(msg)

# roll - rng chat command
#   @ctx     : command invocation context
#   @max_val : upper bound for number generation (must be at least 1)
@bot.command(brief='Generate random number between 1 and <arg>')
async def roll(ctx, max_val: int):
    # argument sanity check
    if max_val < 1:
        raise Exception('argument <max_val> must be at least 1')

    await ctx.send(random.randint(1, max_val))

# roll_error - error handler for the <roll> command
#   @ctx     : command that crashed invocation context
#   @error   : ...
@roll.error
async def roll_error(ctx, error):
    await ctx.send(str(error))

################################################################################
############################## PROGRAM ENTRY POINT ##############################
################################################################################

if __name__ == '__main__':
    # check that token exists in environment
    if 'BOT_TOKEN' not in os.environ:
        log_msg('save your token in the BOT_TOKEN env variable!', 'error')
        exit(-1)

    # launch bot (blocking operation)
    bot.run(os.environ['BOT_TOKEN'])
```

[5p] Task A - Test server and bot application

Before we dive into the script above, we need to set up a discord [https://discord.com/] account (no need to register), create a server [https://support.discord.com/hc/en-us/articles/204849977-How-do-I-create-a-server-] for testing purposes, and create a bot account [https://docs.pycord.dev/en/master/discord.html].

Normally, for bot permissions, you would want to be as restrictive as possible. People will be reticent to give admin privileges to a random bot. In this case, we want our bot able to view channels and logged users, view their text messages and respond in kind, and also to connect to the voice channel and maybe play some music.

Finish the setup as per the links above and save the bot's token. You'll need it in the following task. If you encounter any problem, ask a friend. If you don't have any, the lab assistant will have to suffice.

[5p] Task B - Code skeleton

Let's take a closer look at the code skeleton and see what each thing does:

- `from discord.ext import commands`: as you guessed, this works pretty much like the regular `import module`. The difference here is that we don't import it as a whole, but bits and pieces such as submodules, functions, etc. Additionally, we don't have to specify the whole module chain (i.e.: `discord.ext.commands`) every time we use it. In stead, we can just say `commands` and the interpreter will know what we're referring to.
- `log_msg()`: this is a helper function that colors your message according to its urgency and also displays the line where it was invoked. Use it if you want, but it's not mandatory. Note however the arguments: `msg: str, level: str`. Since *Python 3.5* you can use typing hints [https://docs.python.org/3/library/typing.html] for variables and function return values. When possible, use these hints for clarity if nothing else.
- `@bot.event`: this is called a decorator. Most likely, you've never seen this before and we can't blame you. Things like this belong on r/iamverysmart [https://www.reddit.com/r/iamverysmart/] and not in programming language specifications. A decorator is basically a function processing function. As the name implies, by specifying a decorator before the declaration of that function, the declared function will be passed to the decorator function for “some processing”. In this case, our **on_ready()** function is passed to the event [https://docs.pycord.dev/en/master/api/clients.html] decorator belonging to our global bot instance (line 58). The “processing” that the **event** decorator does it to assign **on_ready()** as the bot's callback method for the event with the same name. So when the bot finishes its initialization and connects to the server, it raises an *on_ready* event and our method will intercept it and resolve it.
- `async` & `await`: in *Python* there's these things called coroutines [https://docs.python.org/3/library/asyncio-task.html]. Coroutines are functions that can be entered, exited and resumed at different points in their execution. Diving into this subject may prove too difficult right now, so we'll try to keep it simple. Thus, please excuse the hand waving: basically, any event handler can be triggered at any time (even when other handlers are already executing), so we mark them as *asynchronous* with the **async** keyword. When we try to call on asynchronous functions, we need to **await** their execution to finish before continuing on our merry way.
- `@bot.command`: yet another decorator, but this one registers the following function as a command. Commands are identified by the prefix established during the bot's instantiation (in this case, `!`). So, by writing *”!roll 100”* in the **discord** chat, the bot will read the message, recognize **roll** as a command, parse the argument and reply with a random number between 1 and 100.
- `@roll.error`: this decorator defines an error callback routine for the **roll** command. Without this, if the user forgets to specify the argument let's say, the **roll()** function will fail with a nasty error printed to *stdout* but the script will not crash! By specifying the error handler, we can output its message to the discord user in stead of the random number. This way, he isn't greeted by silence, but in stead learns what he did wrong.
- `os.environ['BOT_TOKEN']`: generally, it's a bad idea to hardcode IDs and passwords into the source code. As such, we save our bot's token inside an environment variable in **bash** (or **zsh**). When running a program from your shell, it inherits all your variables. So before you run the skeleton, export the token under the name *BOT_TOKEN*.

This must have been a lot to take in. So take some time and ask any question that you may have. If you've done everything correctly so far, once you run the script you should see your bot coming online in your test **discord** server. Try interacting with it!

[10p] Task C - Debugging

Notice that our bot is event driven, so debugging it would be kind of annoying. For this, you can use a little hack:

```
code.interact(local=dict(globals(), **locals()))
```

When executed, this line of code will pause the script and open a **python** shell for you to investigate the state of both the local and global variables. The syntax might seem a bit weird. The **interact()** function takes a dictionary as the *local* argument. The *local* argument is used to initialize the variables available in the newly opened shell. Because we want to see both local and global variables, we need to combine them in a single dictionary. **dict** is the dictionary type. When using it to create a new dictionary (in stead of the basic `{ }` syntax), we can specify another dictionary for initialization: **dict(globals())**. But we want to combine two dictionaries. However, we can't just add them with the `+` operator since it doesn't work that way. Also, we don't want to change any of the dictionaries returned by **globals()** and **locals()** since it might get us into trouble with the interpreter later. But wait! dict [https://docs.python.org/3/library/stdtypes.html#dict] can also be initialized from optional keyword arguments, represented by key-value pairs. So ****locals()** can be used to break down the second dictionary in optional keyword arguments for the resulting dictionary's constructor!

Here's an easier example where `x` is basically **globals()** and `c=3, d=4` is ****locals()**.

```
In [1]: x = { 'a' : 1, 'b' : 2 }

In [2]: y = dict(x, c=3, d=4)

In [3]: y
Out[3]: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

For now, use **code.interact()** in the **on_message()** function. Inspect the bot [https://docs.pycord.dev/en/stable/api/clients.html#discord.Bot] and msg [https://docs.pycord.dev/en/stable/api/models.html#discord.Message] variables.

[40p] Task D - Adding features

Up until now, everything was pretty tutorial-ish. From here on out, it's time for you to get your hands dirty and add some new features to the bot! This means looking stuff on the Internet. For yourselves. Scary stuff… I know!
Hint: a good starting point would be the API Reference page [https://docs.pycord.dev/en/stable/api/index.html].

Here is what you have to add:

1. A **play** command that takes a song name (e.g.: danger_zone.mp3 [https://www.youtube.com/watch?v=yK0P1Bk8Cx4]) as argument. Invoking this command while present in a voice channel should cause the bot to connect to that channel and play the song. The song should be loaded from a local file (hint: use any music file that you have on hand, or a YouTube downloader [https://x2convert.com/en129/download-youtube-to-mp3-music]).
2. A **list** command that lists all available songs in the discord chat.
3. A **scram** command that tells the bot to disconnect from the current voice channel immediately.
4. An event handler for **on_voice_state_update** that checks if the bot was left alone in the channel after a user left. If the bot is indeed alone, it should also disconnect.

Hint: check this out for an example: https://brucecodes.gitbook.io/pycord/guide/voice-commands [https://brucecodes.gitbook.io/pycord/guide/voice-commands]

03. [10p] Feedback

Please take a minute to fill in the feedback form [https://forms.office.com/Pages/ResponsePage.aspx?id=usiMLdqNNEOeXPrCCS6brJoxNMaLqNZHpd8YaA7IhDNUNVVLQ0lQV0tJRzBaRjhOQzdNOVhYWklBVC4u] for this lab.