Brief introduction to Python

Daniel Maldonado

February 2022

Contents

1 Introduction

1.1 Why MedPCPy

Within behavior analysis a common limiting factor is the inability to process data efficiently. Although given enough time most researchers would probably find ways to manage their data with more or less swiftness, many still struggle with manually converting their files to a more manageable format and making manual counts. Excel macros are a great tool for this, but they still require a great deal of engagement, and given that they require human interaction they are prone to errors.

The motivation for making this library was to provide a free and accessible tool that allowed both new and seasoned researchers to quickly gather the data they need without needing to spend time learning to code. This tool aims to be easy to learn and use, less error prone than regular scripting, and above all free (as in *gratis* and as in *libre*) for everyone. Med provides their own software for a similar purpose, but it is prohibitively costly. Those who develop open tools share the belief that one cannot put a price neither on knowledge nor on the tools to produce it.

1.2 About this package

The team behind MedPCPy has extensively worked to make sure that the library works in all conditions. Plenty of bugs and special cases have been found, and all have been fixed to the best of our abilities. However—as is the case with anything programming-related—something is bound to go wrong at some point. It is likely that more bugs will be found, either by us or by users. We encourage you to try and test the library in different cases, and to let us know when things go wrong so that we can improve the library and make it more useful for everyone.

If you find that something does not work as expected, or if you have any ideas on how to improve the library, please feel free to either open an issue on the project's GitHub page, or email us.

2 Brief introduction to Python

Python is a general purpose, interpreted programming language. Although it is not as fast as other languages, such as *C*, its simple syntax makes it ideal for purposes as machine learning and artificial intelligence, among many others. It is easy to learn and very powerful. For those reasons it was chosen to develop this library.

2.1 Installing Python

Python can be installed on any machine by downloading it from its official page, from the Microsoft Store, or from any Linux repository. Several tutorials on YouTube will be of help if any problems arise during the installation.

2.2 Integrated Development Environments

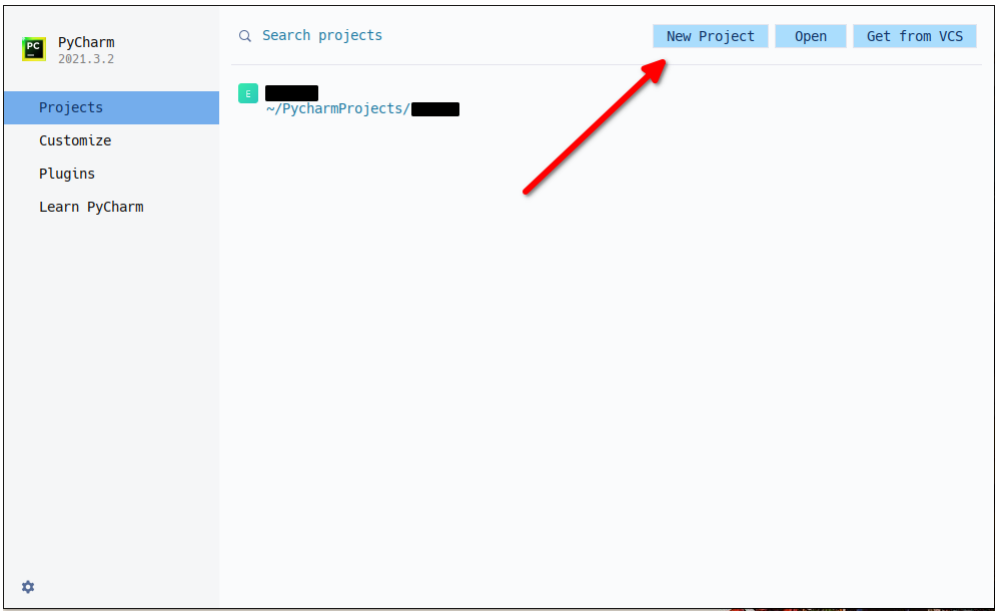
For beginner users and those who wish a working out-of-the-box experience, several IDEs are available. An IDE is a program which provides services to facilitate computer programming, such as syntax highlighting, code autocompletion, easy documentation access, and more. Personally, I recommend the community edition of PyCharm because it is free and can automatically manage virtual environments. But any IDE or text editor will be more than enough.

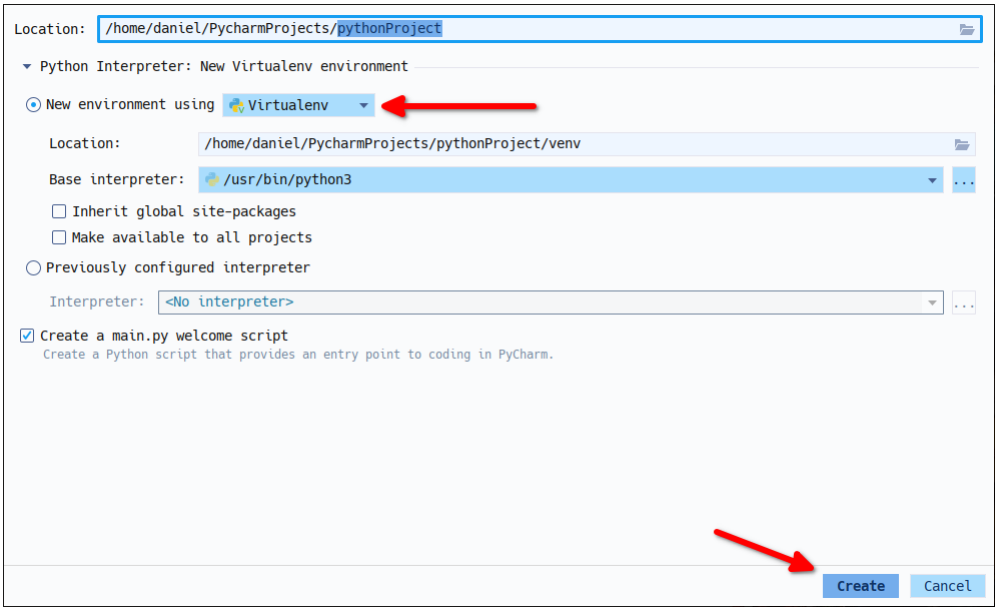
2.3 Virtual Environments

Python is a modular language. Any number of third-party libraries can be installed on a system. This allows the use of an unlimited amount of tools created by the community for all sorts of tasks. That, however, can give rise to conflicts if, for example, a project in which someone is working requires version 2 of a library, but a different project requires version 3. In order to solve this issue, as of Python 3.3 the *virtualenv* module is included with every standard Python installation.

The *virtualenv* module allows the creation of *virtual environments*: containers isolated from the system-wide install which have their own Python binary and can have their independent set of installed Python packages. These packages are accessible only from within the virtual environment, and thus avoid version conflicts as long as projects with different version requirements are hosted in their own separate environments.

The creation of virtual environments is simple. And in fact the PyCharm IDE makes it almost automatic. To create a virtual environment in PyCharm one needs only to select the "New Project" button, choose a directory in which to create it, select "New environment using: Virtualenv", and then click on "Create":





To create a virtual environment without an IDE it is necessary to use the terminal or command line. On windows it can be accessed by pressing WindowsKey + R, and writing "cmd" on the prompt. Clicking "Ok" will then open the terminal.

On Mac, the terminal can be opened by clicking on the Launchpad icon, typing "terminal" and clicking on the Terminal program.

On Linux, I trust you.

Once the terminal is open, a virtual environment can be created by running the command

```
python -m venv [directory]
```

where [directory] is the path to the directory in which you will create the environment. After the virtual environment is created, it can be activated by running a script installed by venv. On windows, this is done with

```
myenv\Scripts\activate.bat
```

on either Mac or Linux this is done with

```
source myenv/bin/activate
```

where "myenv" is the name of the virtual environment directory just created.

After the environment is activated, any packages that are installed will only be so in the local environment. Python scripts that are run while inside the environment will be able to use any packages installed in it.

To deactivate or exit a virtual environment one only needs to run "deactivate" in the terminal. PyCharm automatically opens and closes virtual environments when opening and closing projects.

2.4 Python basics

Providing an introductory course to Python is not the purpose of this guide. However, aiming to be as accessible as possible, the very basics needed to understand and use the MedPCPy library will be explained.

2.4.1 Data types

Several data types are available in Python. The most common ones are `strings`, `integers`, `floats`, `lists`, `tuples`, `dictionaries`, and `booleans`.

Name	Type	Description	Examples
Strings	str	Ordered sequence of characters	"Hello, world" "123"
Integers	int	Whole numbers	0 399 1996
Floating points	float	Numbers with a decimal point	0.0 3.1415
Lists	list	Ordered sequence of objects	[1, 3.6, "one"]
Tuples	tup	Non-modifiable list	(10, "hello", 1.4)
Dictionaries	dict	Ordered Key:Value pairs	{"name": "Giorno", "age": 15}
Boolean	bool	Logical `True`/`False` values	`True` `False`

2.4.2 Variable assignment

All data types can be assigned to variables to make them more manageable.

Python

```python
my_string = "Hello, world!"
my_int = 745
my_float = 100.0
my_list = [my_string, my_int, my_float]
my_tuple = (my_string, my_int, my_float, my_list)
my_dict = {
    "name": "Joseph",
    "nationality": "english",
    "age": 18
}
my_bool = True
```

As you can see, having named variables instead of crude data is more readable. It also allows the use of a previously declared variable on the declaration of another one.

2.4.3 Nesting

Lists, tuples, and dictionaries allow nesting other lists, tuples or dictionaries within them with no limit to how deep the nesting can go.

Python

```python
list_1 = [1, 2, 3, 4]
list_2 = [5, 6, 7, 8]
super_list = [list_1, list_2]
```

In this example the variable `super_list` would contain two items, both of which would be lists. I.e., the code above is equivalent to:

Python

```python
super_list = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

And the same is true for either tuples or dictionaries:

Python

```python
super_tuple = (("a", "b", "c"), "d")
dictionary = {
    "names": ["Jonathan", "Joseph", "Giorno"],
    "nationalities": ["english", "american", "italian"]
}
```

This is also not limited to the same type of variable. That is, lists can contain tuples and dictionaries, tuples can contain lists and dictionaries, and so on.

Dictionaries can even be nested inside other dictionaries:

Python

```
planets = {
    "earth": {"color": "blue", "population": "lots"},
    "mars": {"color": "red", "population": "none"}
}
```

The analysis list used in the MedPCPy library is a list comprised of dictionaries with other, nested dictionaries:

Python

```
analysis_list = [
    # Measure 1
    {"fetch": {"cell_row": 1,
               "cell_column": 1,
               "sheet": "Measure1",
               "summary_column_dict": measure1_cols,
               "offset": 0,
               }},
               ...
                ]
```

Be mindful that the use of space in the declaration of lists, tuples or dictionaries has no syntactic purpose. The distribution is only to increase readability.

2.4.4 Functions

Functions are blocks of reusable code which perform actions that are needed frequently. They are declared only once but can be executed any time they are called. They help increase the readability and modularity of code.

Functions are declared with the `def` keyword, followed by the function name, optionally the arguments in parentheses, and a semicolon. They can either only perform an action, or return a value which can then be assigned to a variable.

Python

```
def hello_function():
    print("Pizza Mozzarella")

def my_sum_function(a, b):
    return a + b
```

Then they are called with their name and the necessary arguments:

Python

```python
hello_function()

x = 20
y = 30
result = my_sum_function(x, y)
print(result)
```

This would result in printing the string `"Pizza Mozzarella"`, assigning the value `50` to the `result` variable, and then printing that variable as well:

Output

```
Pizza Mozzarella
50
```

Functions can have positional arguments (identified by their position) and keyword arguments (identified by a keyword). Additionally, if a default value for an argument is provided when the function is defined, then that argument becomes optional and, if it is not declared when the function is called, it will take the default value.

Python

```python
def another_function(name = "Ford Prefect"):
    print(f"Hello, my name is {name}")

another_function("Arthur Dent")
another_function()
```

This would print the following:

Output

```
Hello, my name is Arthur Dent
Hello, my name is Ford Prefect
```

Given that the optional argument `name` took the default value `Ford Prefect` on the second calling of the function.

2.4.5 Classes and objects

Python is an object-oriented programming language, which means that a good deal of its functionality is built around "objects" which contain data and code. In object-oriented programming the code is organized into reusable "blueprints" called `classes`, and those classes can be used to create individual instances of `objects`. Objects have properties (data) and methods (functions) associated to them.

A class is declared with the `class` keyword, the name of the class, and a semicolon. Class attributes are declared in a special function called `__init__`, and methods associated with the class are declared as functions inside of it.

Python

```python
class Dog:
    def __init__(name, age):
        self.dog_name = name
        self.dog_age = age

    def print_name():
        print(f"""
        Hello, my name is {self.dog_name},
        and I am {self.dog_age} years old. Woof!
        """)
```

Afterwards, the class can be used as a blueprint to create individual objects with the general properties of the class.

Python

```python
my_dog = Dog("John", 5)
```

The creation of the object is similar to the calling of a function. In this example the end result would be an object of class `Dog` being assigned to the variable `my_dog`. This object would have two attributes and a method associated. The attributes can be changed with the notation `object.attribute = [new value]`.

Python

```python
my_dog.dog_name = "Comrade"
```

And the method can be used with the notation `object.method()`.

Python

```python
my_dog.print_name()
```

This would result in printing the phrase

Output

```
Hello, my name is Comrade,
and I am 5 years old. Woof!
```

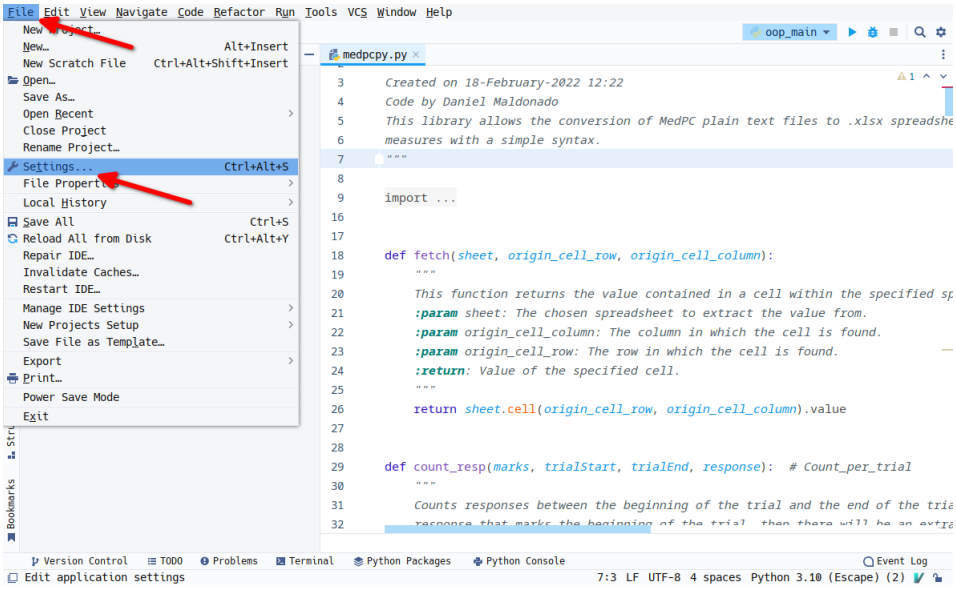
The core of the MedPCPy library is an object of a particular class called `Analyzer`. It has methods to get the sessions associated with a subject, convert files to .xlsx format, create documents, and compute the appropriate responses and latencies.

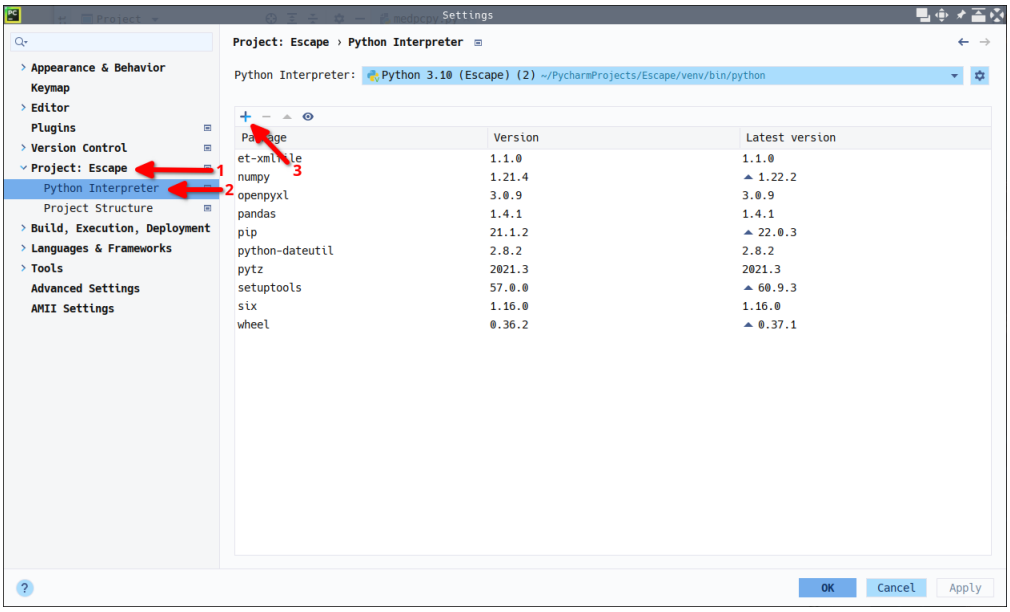
2.4.6 Installing libraries

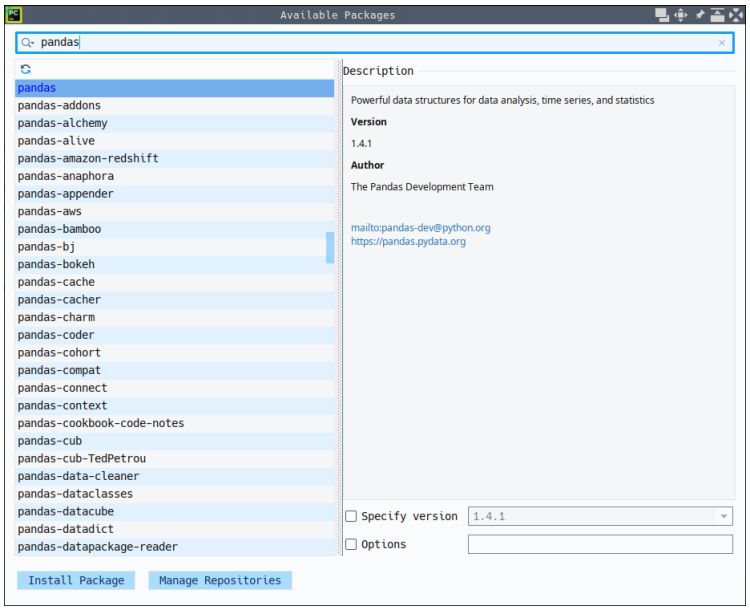
From within the virtual environment libraries can be installed by running the command

```
pip install [library-name]
```

PyCharm can handle the installation of libraries in a longer but more intuitive way if one is not accustomed to working in the terminal. To install a new library navigate to `File` →`Settings` →`Project` →`Python interpreter`. There, click on the + symbol, type the name of the library you wish to install, and click "Install".






Available Packages
pandas
pandas
pandas-addons
pandas-alchemy
pandas-alive
pandas-amazon-redshift
pandas-anaphora
pandas-appender
pandas-aws
pandas-bamboo
pandas-bj
pandas-bokeh
pandas-cache
pandas-cacher
pandas-charm
pandas-coder
pandas-cohort
pandas-compat
pandas-connect
pandas-context
pandas-cookbook-code-notes
pandas-cub
pandas-cub-TedPetrou
pandas-data-cleaner
pandas-dataclasses
pandas-datacube
pandas-datadict
pandas-datapackage-reader
Description
Powerful data structures for data analysis, time series, and statistics
Version
1.4.1
Author
The Pandas Development Team
mailto:pandas-dev@python.org
https://pandas.pydata.org
Specify version
1.4.1
Options
Install Package
Manage Repositories

3 This library

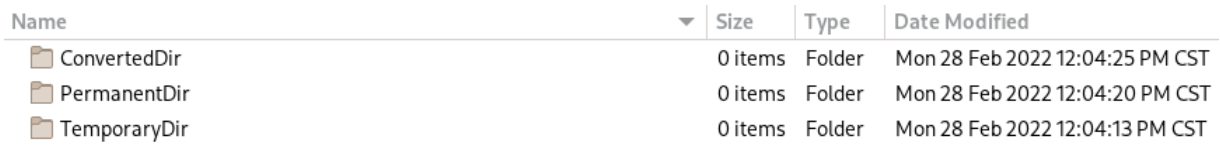
To install this library run the command

```
pip install medpcpy
```

On the terminal, or install it from the interface of your IDE of choice. It will install all its required dependencies by default.

3.1 First steps

The first thing to do is to check that your system is ready to begin the process of file conversion and data extraction. To do this, you must create three separate folders in any part of your system that you desire. These folders will contain the raw files before the analysis, the raw files after the analysis, and the already converted and processed .xlsx files. The reason why two separate folders for raw files are needed is that the library scans the temporary folder and uses its contents to determine the names of the subjects with at least one associated file and the numbers of the sessions to analyze. If all files are contained on the temporary folder, then all files will be processed every time the script is run. That is the reason why only new, unprocessed files must be placed in the temporary folder.



Then, you must check that your files follow the format

```
[subject name][separator][session number]
```

to guarantee that the library is able to read them. The `separator` is a character or group of characters that, as you can guess, separates the subject name and session number. It can be any character or group of characters, but there must be a separator. For example, if the files were named `"Subject_1_session_1"`, `"Subject_1_session_2"`, etc., the `separator` would be `"_session_"`, since that is what is placed between the subject name and the session number.

3.2 Importing the library

The MedPCPy library can be imported to the current working script with the line

Python

```
from medpcpy import *
```

The star "*" makes all the functions and classes available to the current project without the need to write the name of the library every time they are used.

3.3 Declaring variables

Several variables have to be declared before the `Analyzer` object. These are the summary file name, directory paths, subject names, column dictionaries, sheet names, and the analysis list. A full explanation of their requirements (particularly for the column dictionaries and the analysis list, whose names are less intuitive) is provided in the readme file of the GitHub repository. Here only an example declaration is provided:

Python

```
file = "your_summary_filename.xlsx"
temp_directory = "/path/to/your/temporary/directory/"
perm_directory = "/path/to/your/permanent/raw/directory/"
conv_directory = "/path/to/your/converted/directory/"
subjects = ["Subject1", "Subject2", "Subject3"]
measure1_cols = {"Subject1": 2, "Subject2": 4, "Subject3": 6}
sheets = ["Measure1"]

analysis_list = [
    # Measure 1
    {"fetch": {"cell_row": 1,
               "cell_column": 1,
               "sheet": "Measure1",
               "summary_column_dict": measure1_cols,
               "offset": 0,
               }},
    ]
```

After that, the `Analyzer` object can be declared, though it wont yet be ready to analyze data.

Python

```
my_analyzer = Analyzer(fileName=file,
                  temporaryDirectory=temp_directory,
                  permanentDirectory=perm_directory,
                  convertedDirectory=conv_directory
                  subjectList=subjects, suffix="_",
                  sheets=sheets,
                  analysisList=analysis_list,
                  relocate=False)
```

After its declaration, the `Analyzer` object must be used to convert at least one file to .xlsx format with the `.convert()` method.

Python

```
my_analyzer.convert()
```