# Using Python in (Occultation) Astronomy

Mike Kretlow · IOTA/ES · Lauenbrück · Germany · mike@kretlow.de

**ABSTRACT:** Python is one of the most popular programming languages and is a great tool for scientific computing. Thus, it is well suited for almost any kind of astronomical applications. In the last few years many large packages for astronomical and astrophysical work have been developed or have matured, in many cases actively maintained by professional institutions. A first, brief introduction into Python is given here and the use of Python for occultation work is demonstrated in practical examples.

## Introduction

Python [1] is a high-level, general-purpose programming language, created by the Dutch programmer Guido van Rossum and first released in 1991. It is very well suited for scripting, interactive work and quick prototyping, while being powerful enough to write large applications in it. Python is open-source, and its license makes it freely usable and distributable, even for commercial use. Python runs on all major platforms like *Microsoft Windows*, *Linux* and *Mac OS*, and on various hardware.
The language was not designed (especially) for scientific computing, which was at that time the domain of high-level compiled languages like Fortran and C/C++ (later also Java) or domain-specific systems like Matlab, IDL, R, etc. But over time the language evolved, matured (and got faster), and with the availability of more and more packages (libraries) its popularity increased and vice versa. In 2005 the NumPy [2] package was released which boosted the usage of Python in numeric computing[1].

[1] Predecessor were the packages Numeric and Numarray, dating back to 1995.

Complemented by packages like SciPy, Matplotlib and Pandas, a comprehensive, easy to use and professional level tool box for scientific computing is available to the community.
Since 2018 Python is among the first three most popular programming languages according to the TIOBE index (Figure 1). Python is widely used in any kind of web applications and services, artificial intelligence (AI), machine learning (ML), computer vision, scientific computing, data science, and as a general scripting language. Many (large) companies and organizations use Python, like Wikipedia, Google, Facebook, Dropbox, CERN, NASA, and much more.
Python's design and philosophy has influenced other programming languages - for example Go, Ruby and Julia.
This article isn't meant to be a complete introduction nor a tutorial to Python. You will find plenty of free introductions and (video) tutorials on the web, probably even in your native language, as well as books on the market.

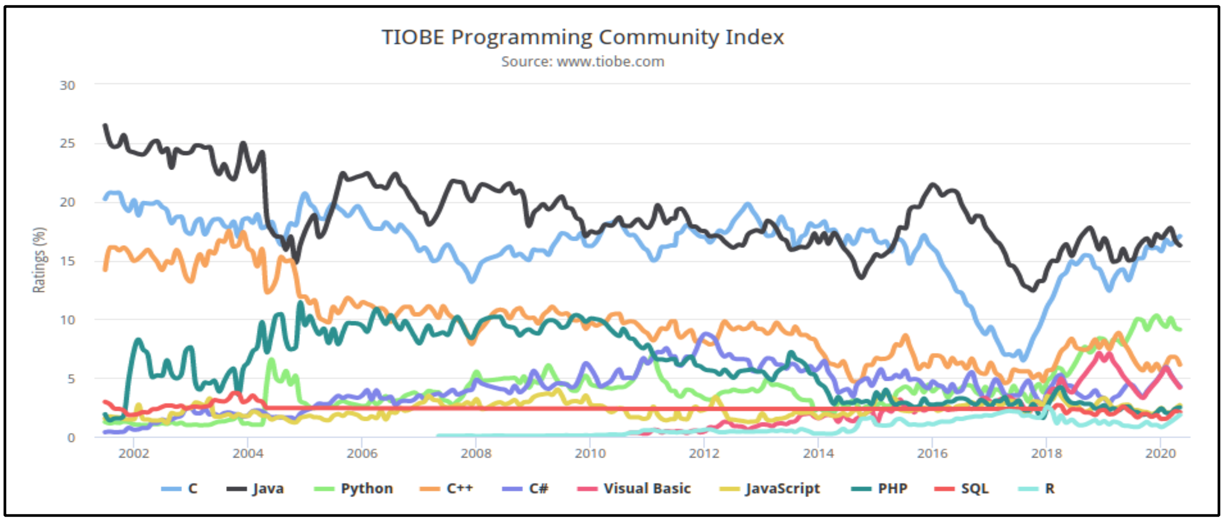


*Figure 1. The TIOBE programming community index is a measure of popularity of programming languages. The rating is based on a formula that assesses searches related to programming languages in 25 search engines including Google, Baidu, Yahoo, Wikipedia, Bing.*

## Technical Aspects

Python is well structured and dynamically typed. Object-oriented programming and structured programming are fully supported, and many of its features support functional programming. Python is meant to be an easily readable language. Its formatting is visually uncluttered and it uses white-space indentation, rather than curly brackets or keywords, to delimit logical code blocks. So, indentation is part of the syntax. Beside the comprehensive standard library, which provides functions suited for many tasks, Python was designed to be highly extensible by third-party modules (packages), which is a main reason for its popularity (see the section "Ecosystem").

The Python language (standard) says nothing about whether the language is compiled or interpreted. It depends on the specific implementation. Compiled languages use a compiler which translates the language code directly into (hardware-specific) machine code which can be executed on the processor.

An interpreted language is any programming language that isn't already in machine code prior to run time. Translation from source code into machine code occurs at the same time as the program is being executed, this process costs execution speed. In the (official) Python implementation, CPython, the source code (.py) is compiled into a much simpler form called byte code (.pyc), which is then interpreted. In that case, Python (CPython) is neither a true compiled nor a pure interpreted language. Other Python implementations like PyPy or Python packages like Numba provide a (partly) Just-In-Time (JIT) compilation of the Python source code (or specific parts like numerical functions in the case of the Numba package).

## Getting Started

First you have to install Python[2]. This can be done by downloading a Python installation package from [1] or, if you prefer another popular implementation / distribution, Anaconda Python [3] would be a good choice. *Linux* users might use their package manager for installing Python, but usually it is already installed on modern mainstream *Linux* distributions.

After installation it is possible to start a Python program with the shell command **python filename.py** or you can invoke the interactive shell (REPL, Read Eval Print Loop) by just entering **python** on the command shell. You can use the REPL for your first steps or simply as a sophisticated calculator:

Another great toolbox is the Jupyter software [4], which allows you to run various programming languages (like Python, Julia or R) inside your web browser, providing you with a sophisticated, interactive and graphical environment. This approach is very common among scientists or in education and we will use these so-called Jupyter notebooks in this article from now on. The examples presented here are available as notebooks on [5]. You can download them from GitHub or clone the repository, and run it on your local hardware. You can even run the notebooks directly on the web using Binder [6], providing the GitHub URL given before, so you just need a browser and installation work is zero.

For a more comprehensive and step-by-step description of the installation of Python, Jupyter (if you do not use Binder) and Python packages, please refer to the installation documentation and guides given on the web sites from where you got your installation packages. And by searching the web for beginner guides you will also find a lot of help going beyond the scope of this article.

## Hello World

Most tutorials and books start with a very simple first program ("Hello World"). The following first example is a little bit more extensive in order to introduce some basic entities and features of the Python language at one glance. Python has different data types to store numerical values, strings, Boolean values (true and false), etc. It has sequences like lists, tuples (which are similar to lists, but the elements are immutable) and dictionaries (key-value pairs). Because of the dynamical typing you do not have to declare the data type of a variable and you can change it freely. All the features which are common to programming languages you will find in Python as well, like comparing operators and conditional statements. Some functions (like the print() function) are built-in, other functions need to be imported before they can be used. The **log10** function used in the following example is defined inside the **math** module, which is part of the Python standard-library. The hash symbol # indicates a single comment line (multi-line comments exist as well). Python has different kinds of loops. In this example we iterate over the content of a list (**smag**).

```
linux> python
Python 3.8.3 (default, May 17 2020, 18:15:42)
[GCC 10.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 17+4*4
33
>>> from math import *
>>> acos(0)
1.5707963267948966
>>> deg2rad = 180/pi
>>> deg2rad
57.29577951308232
```

[2] Throughout the paper Python 3 is used interchangeably with Python. Python 2 has reached its EOL (end-of-life).

User defined functions are declared by the keyword **def**, followed by the function name and an optional argument list. They can return a value or a list of values (or none).
The first example defines a function to add two magnitudes. In the main part of the script a fixed value `pmag = 14.5` is defined and a list of three secondary magnitudes is stored in a list named **smag**. We loop over all values in the list and compute the combined magnitude of pmag + smag (for all values in smag) by calling our function `addmag()`, which we have defined before. As you see, the formatting is part of the syntax: the loop body is defined by its indentation level.

This simple program gives you a first impression of what Python looks like inside a Jupyter notebook. Each code cell (grey box) contains code which is executed and below the cell the output (if any) of that cell is displayed. If you are already a little bit familiar with Python you can go ahead and study the scripts given in the Example section. Otherwise I recommend you now make your first steps in Python by working through one of the many tutorials about Python you will find on the web (or by using a book).

## Example 1: Adding Magnitudes

```
In [1]: from math import log10

        # This function adds two magnitude values given as arguments x and y.
        # The result res is returned to the caller.
        def addmag(x,y):
            res = -2.5*log10(10**(-x*0.4) + 10**(-y*0.4) )
            return res

        # The main program (script) starts here
        print("Hello world, this is Python\n")

        # A single float value stored in a variable named pmag
        pmag = 14.5

        # A list of float values stored in a variable named smag
        smag = [12.5, 14.5, 15.9]

        # Loop over all values given in the list smag. The current value is assigned to the variable mag.
        for mag in smag:

            # Compare values and execute conditional statements (assign a string to variable cmt)
            if mag < pmag:
                cmt = "star is brighter than planet"
            elif mag > pmag:
                cmt = "star is fainter than planet"
            elif mag == pmag:
                cmt = "star is equal bright as planet"

            # Compute combined magnitude of pmag and the current loop value
            cmag = addmag(pmag,mag)

            # Print result using a Format-String (f-string)
            print(f"pmag = {pmag}, smag = {mag}: combined magnitude is {cmag:0.2f} ({cmt})")

        # Loop body ends here because of indentation level
        print("\nHave a nice day !")
```

```
Hello world, this is Python

pmag = 14.5, smag = 12.5: combined magnitude is 12.34 (star is brighter than planet)
pmag = 14.5, smag = 14.5: combined magnitude is 13.75 (star is equal bright as planet)
pmag = 14.5, smag = 15.9: combined magnitude is 14.24 (star is fainter than planet)

Have a nice day !
```

## Ecosystem and Resources

There is an overwhelming amount of resources and the Python ecosystem is huge, even if we focus on scientific computing and still even if we just consider astronomy within that domain. Many of these libraries / frameworks are mature and have been well tested for 5-10+ years.
In this section some important packages for scientific computing and for astronomical applications are presented.

**Python** and **PyPi**: The main website of Python [1] was already mentioned. Use it as a starting point to learn Python and to navigate to other resources. You will also find there downloads for the installation. The Python Package Index (PyPi) [7] is a repository of software for the Python programming language. Currently about 241000 (!) packages are listed at PyPi. Packages can be simply installed using the Python package installer `pip`, by typing at the command prompt:

```
pip install package_name ...
```

For example, to install NumPy, SciPy, Matplotlib and Pandas you type:

```
pip install numpy scipy matplotlib pandas
```

With `pip list` you get a list of Python packages currently installed on your system.

**NumPy:** As already mentioned, Python was not originally designed for numerical computing, but attracted early the attention of the scientific and engineering community. NumPy [2] adds support for large multi-dimensional arrays and matrices and a large set of mathematical functions to work very fast on these arrays (vectorised). NumPy is the base of almost any scientific application or package for Python.
**SciPy:** SciPy [8] is another major and important package used for scientific and technical computing. It contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE (ordinary differential equations) solvers and other tasks common in science and engineering.
**Pandas:** Pandas [9] is a library for fast, easy and powerful data analysis and manipulation.
**Matplotlib:** Matplotlib [10] is a comprehensive and very popular plotting library for creating static, animated, and interactive visualizations ('plots') in Python. You can use it inside Jupyter (in your browser) and the plots are of high quality ("publication ready") and can also be saved in many formats (images, PDF etc.).
**Astropy:** Astropy [11] is a community Python library for astronomy, containing key functionality and common tools needed for many tasks in astronomy and astrophysics. It is at the core of the Astropy Project, which aims to enable the community to develop a robust ecosystem of affiliated packages, covering a broad range of needs for astronomical research, data processing, and data analysis.

**Astropython:** This website [12] is the starting point if you are using Python for astronomy. It is a community-maintained knowledge base and repository with tutorials, list of astronomical related packages, wiki pages and many other resources.

Often NumPy, Pandas and Matplotlib are used together in a script and you will frequently see an import of these three packages into the main name space using an alias:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Now you can access NumPy functions with `np.function()`, for example `np.sqrt(x)`.

SciPy does not recommend to import the whole package (`import scipy as sp`), but rather to import just the specific modules which are needed, for example
`from scipy import integrate`.

## Examples

In the following sections I demonstrate how I use Python in my astronomical work. These few examples should give you an idea how Python can be used. It covers only a tiny fraction of what is possible. The scripts are not very sophisticated nor should they be considered as 'complete', but my intention was to make them neither too long nor too complicated. Thus, they also do not necessarily represent an elegant Python programming style.

### Occultation Light Curve Analysis

The first example shows the reduction of an occultation by TNO (50000) Quaoar observed on 2019 Sep 26 in Namibia.
The photometry was done using the program *PyOTE* [13] but could also be done with other programs. The content of the light curve data (CSV) file produced with *PyOTE* looks like:

```
R-OTE
FrameNum, Time, Value, Ref1, Ref2, Ref3
0,[18:58:32.9070],2543.818182,4818.000000,7406.636364,5220.333333
1,[18:58:33.2070],1308.000000,3940.166667,7305.384615,5065.909091
2,[18:58:33.5070],2308.666667,4422.000000,6823.333333,4625.333333
3,[18:58:33.8070],2109.615385,4645.545455,7359.000000,5063.500000
…
2031,[19:08:42.2230],2276.000000,5021.000000,7292.846154,4506.000000
```

After two lines of header we have comma separated frame number, timestamp and the photometry of the target star and three reference stars.
For the fitting of a well-depth to the occultation light curve in order to derive the times of D and R of disappearance and reappearance we use the package LMFIT (Non-Linear Least-Squares Minimization and Curve-Fitting for Python) [14], which can be installed using pip with `pip install lmfit`.

The workflow is very simple (the leading number is the number of the corresponding code cell in the corresponding Jupyter notebook):
5: read the photometry from the CSV file into NumPy arrays.
7: make a first plot of the raw occultation light curve (optional step).
8: define the region on the x-axis (unit is frame number) which is outside the occultation. Calculate the intensity mean in the 'outside' region and normalize the light curve (the intensity on the y-axis is now between 0 and 1).
10: fit a rectangle model (well-depth) to the light curve. Among much other information about the performed fit, we get the fit parameter step_center1 and step_center2, which correspond to the point of D and R (in frame units). We also get uncertainties from the covariance matrix. But LMFIT is also able to explicitly calculate the confidence intervals, which is omitted here (cell 11).
12: plot the light curve together with the fit curve, which could be saved and used for publication.
With some additional code we could directly convert frame numbers (and fractions) into times (UT), which is omitted here.
We got the result : D at frame 812.06 +/- 0.23 (1-sigma) and R at frame 1216.00 +/- 0.35 (1-sigma).

## Example 2: Occultation Light Curves and Fits

```
In [1]: %matplotlib inline
```

```
In [2]: # Import packages
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.dates as mdates

        from lmfit.models import LinearModel, StepModel, RectangleModel
```

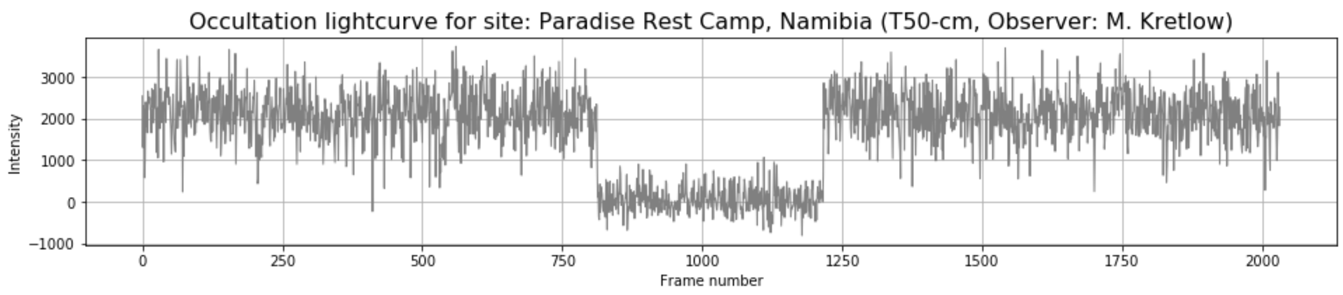
```
In [3]: # Set plot size
        plt.rcParams["figure.figsize"] = (16,6)
```

```
In [4]: # Plot settings: title
        occ_title = "Occultation lightcurve for site: Paradise Rest Camp, Namibia (T50-cm, Observer: M. Kretlow)"
```

```
In [5]: # Read data from CSV file (generated with PYOTE)
        data = np.genfromtxt('OLC_MKretlow_OTE.csv',dtype=None, encoding=None, delimiter=',', comments='#', skip_header=2,
        usecols = (0,1,2), names=['frame', 'time', 'value' ],)
```

```
In [6]: # Get number of frames in the CSV file
        num_of_images = len(data['frame'])
        print("Number of frames / images = ",num_of_images)

        Number of frames / images =  2032
```

```
In [7]: # 1st Plot: raw occultation light curve
        fig = plt.figure()
        ax1 = fig.add_subplot(211)
        ax1.plot(data['frame'], data['value'], color='gray', linestyle='-', linewidth=1, marker='')
        ax1.grid()
        ax1.set_xlabel("Frame number")
        ax1.set_ylabel("Intensity")
        plt.title(occ_title,fontsize=16);
```



In [8]:
```python
# Normalize the occultation light curve

# Define 'outside occultation' region points
l1,l2,r1,r2 = 50,750,1250,2000

xall = data['frame']
yall = data['value']

# Normalize signal using region outside occultation (trim here left and right region)
y_outside = np.concatenate((yall[l1:l2],yall[r1:r2]), axis=0)
yall = data['value'] / np.mean(y_outside)

print("# frames in outside region = ",len(y_outside), ", mean = ",np.mean(y_outside))
```

```
# frames in outside region =  1450 , mean =  2142.316924957931
```

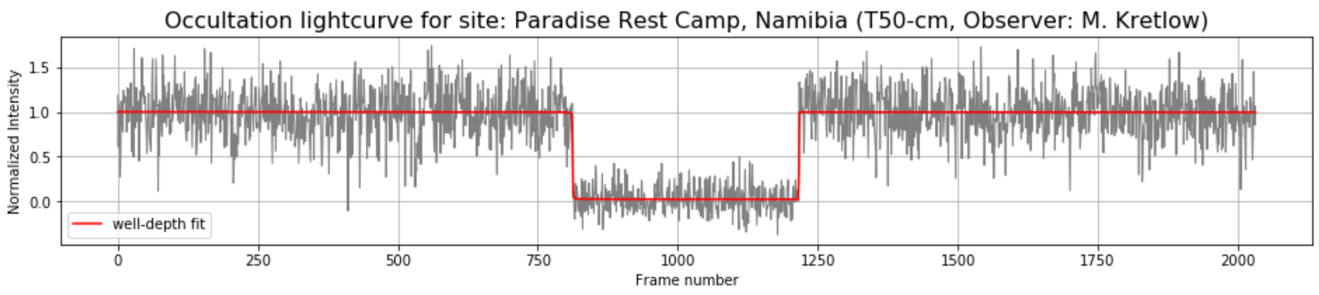
In [9]:
```python
# Slice dataset for the fit (for example just D or R region using step model)
x = xall #[800:2000]
y = yall #[800:2000]
```

In [10]:
```python
# Fit the model function (well-depth) to the light curve

step_mod = RectangleModel(form='atan', prefix='step_')
line_mod = LinearModel(prefix='line_')

pars = line_mod.make_params(intercept=y.min(), slope=0)
#pars += step_mod.guess(yy, x=x, center=800)
pars += step_mod.guess(y, x=x) #, center1=800,center2=1200)

mod = step_mod + line_mod
out = mod.fit(y, pars, x=x)

# The errors here reported are 1-sigma errors estimates from the covariance matrix of the LS fit
print(out.fit_report())
```

```
[[Model]]
    (Model(rectangle, prefix='step_', form='atan') + Model(linear, prefix='line_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 292
    # data points      = 2032
    # variables        = 7
    chi-square         = 121.189761
    reduced chi-square = 0.05984680
    Akaike info crit   = -5715.05781
    Bayesian info crit = -5675.74038
[[Variables]]
    line_slope:     -2.6298e-06 +/- 9.2671e-06 (352.39%) (init = 0)
    line_intercept:  1.00167922 +/- 0.01122247 (1.12%) (init = -0.3762282)
    step_amplitude: -0.97645174 +/- 0.01419833 (1.45%) (init = 2.120912)
    step_center1:    812.062101 +/- 0.23394510 (0.03%) (init = 507.75)
    step_sigma1:     0.14333769 +/- 0.44418524 (309.89%) (init = 290.1429)
    step_center2:    1216.00038 +/- 0.36659147 (0.03%) (init = 1523.25)
    step_sigma2:     4.7334e-04 +/- 0.44369523 (93736.48%) (init = 290.1429)
    step_midpoint:   1014.03124 +/- 0.22061726 (0.02%) == '(step_center1+step_center2)/2.0'
[[Correlations]] (unreported correlations are < 0.100)
    C(step_center2, step_sigma2)      =  1.000
    C(line_slope, line_intercept)     = -0.839
    C(step_center1, step_sigma1)      =  0.820
    C(line_intercept, step_amplitude) = -0.251
    C(step_amplitude, step_sigma2)    = -0.203
    C(step_amplitude, step_center2)   = -0.203
    C(step_amplitude, step_sigma1)    = -0.202
    C(step_amplitude, step_center1)   = -0.172
```

```
In [11]: # The lmfit confidence module allows you to explicitly calculate confidence intervals for variable parameters. For
         # most models, it is not necessary since the estimation of the standard error from the estimated covariance matrix
         # is normally quite good.
         # But for some models, the sum of two exponentials for example, the approximation begins to fail.
         # For this case, lmfit has the function conf_interval() to calculate confidence intervals directly.
         # This is substantially slower than using the errors estimated from the covariance matrix,
         # but the results are more robust.

         # out.conf_interval()
         # print(out.ci_report())
```

```
In [12]: # Plot light curve together with well-depth fit
         fig = plt.figure()
         ax1 = fig.add_subplot(211)
         ax1.plot(x, y, color='gray', linestyle='-', linewidth=1, marker='')
         ax1.grid()
         ax1.plot(x, out.best_fit, 'r-', label='well-depth fit')
         ax1.set_xlabel("Frame number")
         ax1.set_ylabel("Normalized Intensity")
         ax1.legend(loc='best')
         plt.title(occ_title,fontsize=16);
```



```
In [13]: # Get frame number of disappearance (D) and reappearance (R), which can be converted to times
         print(out.params['step_center1'])
         print(out.params['step_center2'])

         <Parameter 'step_center1', value=812.0621012136145 +/- 0.234, bounds=[-inf:inf]>
         <Parameter 'step_center2', value=1216.0003838355685 +/- 0.367, bounds=[-inf:inf]>
```

## Access Occult Watcher Feeds

The Beautiful Soup [15] package is designed for scraping web pages. It makes it easy to parse HTML tables from web pages into Pandas data frames (DF). Once the data are stored in DFs, you have a lot of possibilities to process these data (from statistics up to visualization). Example 3 shows how we parse some of the existing *Occult Watcher* (OW) feeds and display a list of events summaries. But moreover, we store these data directly into an SQLite database with only some lines of additional code. This DB (a simple file) could be used by another tool or application we could think of, if we add some code to parse the ground tracks as well. A lot of Python packages provide functionality for plotting geographical data on maps etc. That code could be used as core for your own small occultation planning application, written in Python and thus be able to run on different operating systems or even as a web application. Use a tool like the program "DB Browser for SQLite" [16] to open that data base and to browse the data.

The purpose of this script is not to demonstrate how to parse *Occult Watcher* feeds in order to build a fully functional Python client for it. OW feed format could change at any time without notice and this script could stop working in the future. The purpose of Example 3 is to show how to parse web pages and *Occult Watcher* feed sources were used as they are something with which the reader will be familiar.

## Vizier Queries

In Example 4 we use the packages Astropy [11] and Astroquery [17] to make catalogue queries at VizieR (CDS). Astropy and Astroquery are large and powerful packages, check out the web sites and documentation. In this simple example we make a search around a position in the sky (RA, DE) in order to search for stars in catalogues at this position (i.e. also some kind of cross-reference). Finally, we plot their catalogue positions for comparison. So, you will get an idea of the positional scatter of different catalogues.

Example 3: https://github.com/mkretlow/JOA2020-3/blob/master/Example_3/Example_3.ipynb
4: https://github.com/mkretlow/JOA2020-3/blob/master/Example_4/Example_4.ipynb

## Conclusion

Python is a great tool for data science, for astronomy and for occultation work. It is easy to learn, powerful, and many different working environments are in your hands. Depending on what's best suited for your current tasks, you run Python programs on the console, or you might want to work interactively with Jupyter notebooks or just with the REPL (in that case, IPython is recommended). You can write classical applications with GUIs (graphical user interfaces) or run your code within a web framework like Django [18]. The huge amount and diversity of packages enables you to focus on your specific task, avoiding reinventing the wheel. Because almost all packages are open source you can examine what these packages are doing, and you can copy and change the code for your purposes, if necessary. And this is all for free. Enjoy Python!

Many common concepts in the daily work of (experienced) Python users, as well as data scientists and astronomers using Python are not covered here because it would be far beyond the scope of this article. For example, virtual environments (pyenv, pipenv etc.), JIT implementations (Numba, PyPy), IPython, JupyterLab, many more packages addressing astronomical topics, or object-oriented programming.

The examples are simplified versions of my working copies in order to reduce the amount of content and complexity. For example, they do not handle exceptions (very well). But anyway, they are (hopefully) useful for the reader and with some experience they can serve as a basis or template for your own applications and tasks.

## References

[1] https://python.org
[2] https://numpy.org
[3] https://anaconda.com
[4] https://jupyter.org
[5] https://github.com/mkretlow/JOA2020-3
[6] https://mybinder.org
[7] https://pypi.org
[8] https://scipy.org
[9] https://pandas.pydata.org
[10] https://matplotlib.org
[11] https://astropy.org
[12] https://astropython.org
[13] https://pypi.org/project/pyote
[14] https://lmfit.github.io
[15] https://pypi.org/project/beautifulsoup4/
[16] https://sqlitebrowser.org
[17] https://astroquery.readthedocs.io/en/latest/
[18] https://djangoproject.com

# IOTA Annual Meeting & ESOP XXXIX

Oliver Klös · IOTA/ES · Eppstein-Bremthal · Germany · pr@iota-es.de

**ABSTRACT:** Due to the COVID-19 pandemic the IOTA Annual Meeting 2020 and the 39th European Symposium on Occultation Projects (ESOP) will be held via the web. The worldwide community of occultation observers is invited to join the meetings. The technical aspects and the schedules were still under preparation by the organising teams at the date of publication of this issue of JOA.

## IOTA Annual Meeting 2020

This year's annual meeting of IOTA will be held on 2020 July 25 & 26. At the current status (2020 Jun 29) the participants will meet online using the virtual meeting software *Zoom*.

Pariticipants won't need to preload any software onto their computer. Simply click on the *Zoom* access number when it is presented on the message list [1], and the participant will be asked to install a program that will connect to the meeting. The software deletes itself when you are done with it.

Roger Venable, Vice President of IOTA, has lined up the following presenters and lectures [2]:

David Dunham will talk about the best North American lunar occultations and grazes of the last year and the best North American lunar occultations and grazes coming up during the next year. He will report on his experiences of recruiting new IOTA members and observers online and of observing asteroidal events in Arizona. Steve Preston, President of IOTA, will give an outlook on the best asteroidal occultations observable from North America during the coming year.