flatlib Documentation

Release 0.2.0

João Ventura

October 11, 2016

Contents

1 Contents 3
1.1 Installation . . . 3
1.2 Tutorials . . . 5
1.3 Frequently Asked Questions . . . 13

Flatlib is a Python 3 library for Traditional Astrology.:

```
>>> date = Datetime('2015/03/13', '17:00', '+00:00')
>>> pos = GeoPos('38n32', '8w54')
>>> chart = Chart(date, pos)

>>> sun = chart.get(const.SUN)
>>> print(sun)
<Sun Pisces +22:47:25 +00:59:51>
```

CHAPTER 1

Contents

1.1 Installation

The following instructions will install flatlib from the source files. In the future, binaries may be made available and the instructions will be updated accordingly.

1.1.1 Windows

If you don't have Python 3 installed on your system, download and install the latest Python 3.4 for Windows from https://www.python.org/downloads/. You can check if the interpreter was correctly installed by executing `py` on the command line.

Open a Windows command prompt (or exit the Python interactive shell) and install flatlib using `py -m pip install flatlib`.

If you get an error such as **Microsoft Visual C++ 10.0 is required (Unable to find vcvarsall.bat)**, you will have to install a C compiler. The compiler is required to build *pyswisseph* - the Python port of the Swiss Ephemeris.

There are several C compilers for Windows, such as Cygwin and MinGW, but Visual C++ 2010 is the most used for compiling Python 3 extensions on Windows. Download Microsoft Visual C++ 2010 Express from http://go.microsoft.com/?linkid=9709949 (it may require the creation of a free Windows Developer account). After the installation, execute the following on the command line:

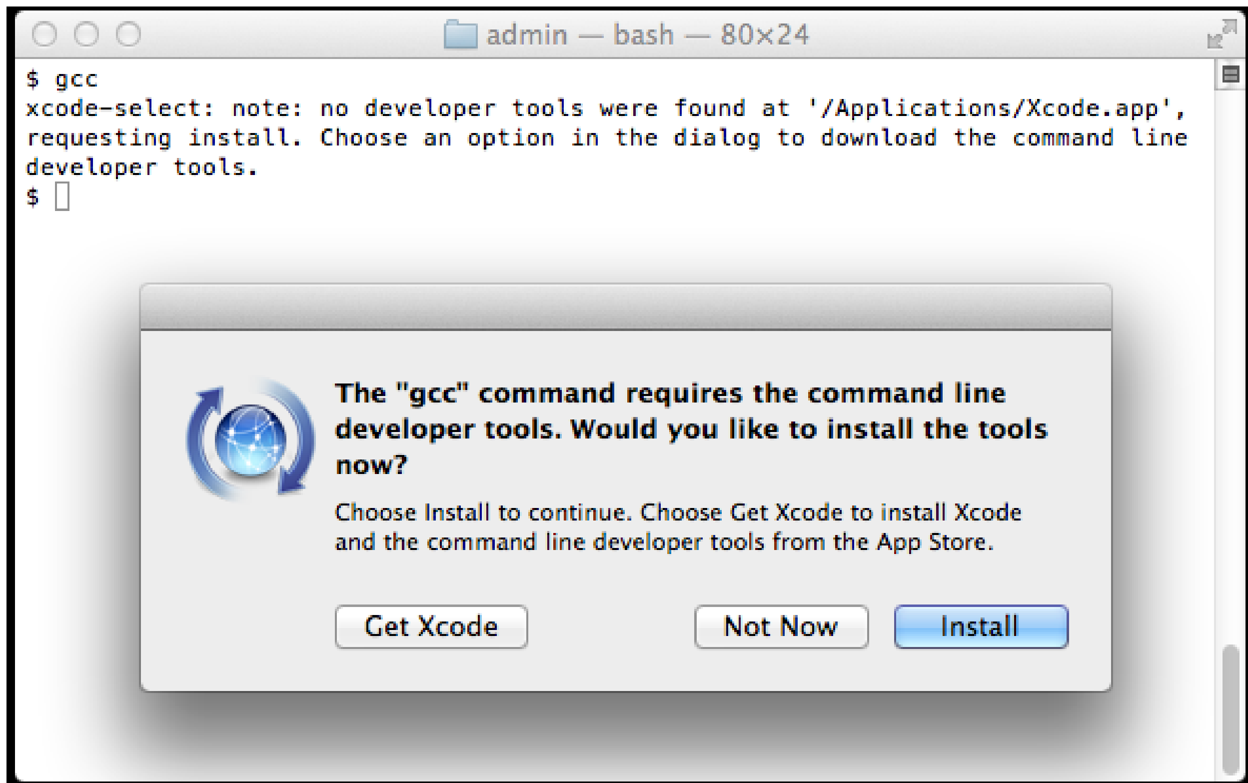
```
set CL=-DWIN32
py -m pip install flatlib
```

You should now have flatlib installed in your system.

1.1.2 OS X

Latest versions of OS X are bundled only with Python 2. The preferred way to install Python 3 in OS X is using the homebrew package manager (http://brew.sh/). Install homebrew and then install Python 3 using `brew install python3`.

Before you install flatlib, you must have a C compiler in your system. This is because flatlib depends on the Swiss Ephemeris which is implemented in C. Fortunatelly, Apple provides the *Xcode Command Line Tools* which bundles a C compiler. To install it, open the terminal (Applications/Utilities/Terminal) and execute `gcc`. You'll see an alert box if you don't have a compiler installed:



If you don't need the entire Xcode (about 2.5GB) just press **Install**..

Finally, to install flatlib use `pip3 install flatlib`.

1.1.3 Linux

Python 3 is already included on most of the newer distributions. The simplest way to test for the existence of Python 3 is to open the terminal and execute `python3` to start the interactive python interpreter. If the interpreter is not found, you will have to install it from your distribution's repo.

To install flatlib, use `pip3 install flatlib`. It may require you to install *pip* and other python 3 development libraries.

If you get a Permission Denied error, execute the previous command with `sudo`.

1.1.4 Testing the installation

Start the python3 interactive interpreter (`python3` on Linux and Mac, and `py` on Windows) and execute the following:

```
>>> import flatlib
>>> flatlib
<module 'flatlib' from '/usr/local/lib/python3.4/dist-packages/flatlib/__init__.py'>
```

If you don't get an import error, flatlib is installed in your system.

1.1.5 Upgrading from a previous version

To upgrade from a previous version, run:

- `pip3 install flatlib --upgrade` in Linux and Mac.
- `py pip install flatlib --upgrade` in Windows.

1.1.6 Uninstalling

Just do `pip3 uninstall flatlib` on Linux and Mac or `py pip uninstall flatlib` on Windows.

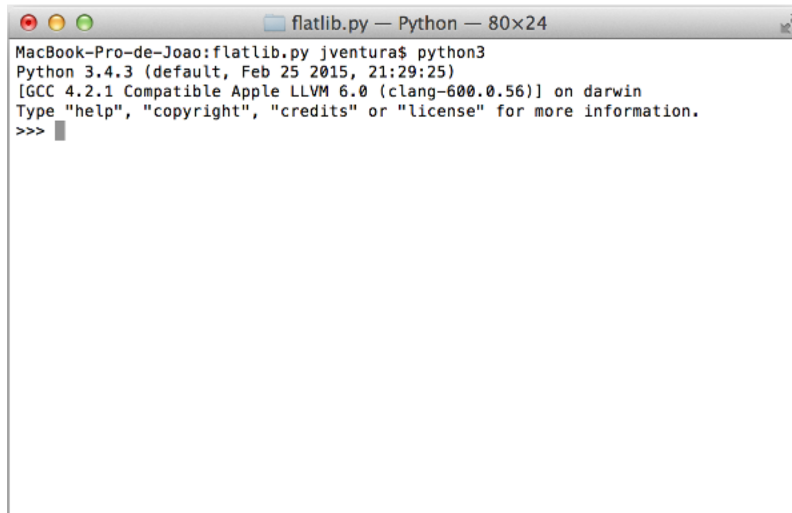
1.2 Tutorials

1.2.1 Introduction to Python

The purpose of this tutorial is to show you how you can quickly start working with the Python programming language. It assumes that you have already succesfully installed Python in your computer. If you already know how to use Python, you can skip this tutorial.

Interactive interpreter

Python is a general purpose programming language that bundles an interactive interpreter. To start the interactive interpreter, open the terminal (command prompt in Windows) and execute `python3` (or `py` in Windows)



In the interactive interpreter, you can enter commands and the Python interpreter will execute those commands and return the answers.

Print command

The *print* command is one of the most useful commands in Python, since it allows us to print the contents of python *variables*. For instance, try the following in your Python interpreter:

```
>>> print('Hello, world!')
Hello, world!
```

You can see that the interpreter will print the exact contents of whatever you put between quotation marks (called string). Try to print other strings.

Variables

In computer languages, variables are reserved memory locations to store values. Basically, when you create a variable, the Python interpreter allocates memory and stores a value in that memory location. The equal sign (=) is used to assign values to variables. Try the following in your interpreter:

```
>>> number = 100
>>> miles = 1000.0
>>> name = "John"
>>> print(number)
100
>>> print(miles)
1000.0
>>> print(name)
John
```

You can assign integer numbers to variables, floating point numbers and strings.

Basic Operations

Since variables may include numbers, strings or other objects, Python allows us to do some basic operations on variables, such as adding, subtracting, multiplying or divinding. Some of those operations make more sense on numbers, but others may be applied in strings as well.

Try the following operations on numbers:

```
>>> num1 = 10
>>> num2 = 20
>>> num1 + num2
30
>>> num1 - num2
-10
>>> num1 * num2
200
>>> num1 / num2
0.5
```

Now let's see how it works for strings:

```
>>> name1 = 'Homer'
>>> name2 = 'Simpson'
>>> name1 + name2
HomerSimpson
>>> name1 - name2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Some operations do not make sense for strings.

Assignment operator

Something very important in programming languages is the assigment operator. We have already discussed it above when we assigned numbers and strings to variables using the equal operator (=). The important thing to notice is that we can assign the result of an operator to a variable like in the following example:

```
>>> num1 = 10
>>> num2 = 20
>>> result = num1 + num2
>>> print(result)
30
>>> result   # This also works in the interpreter
30
```

Python lists and dictionaries

In Python, there are two special data structures called *lists* and *dictionaries*. A list is basically a sequence of variables which can be or not of the same data type. Here is a simple example:

```
>>> mylist = [-5, 0.3, 2.5, 33]
>>> mylist[0]
-5
>>> mylist[1]
0.3
>>> mylist[0] + mylist[1]
-4.7
```

We can access the individual contents of a list by referring to the index number between brackets. For instance, `mylist[1]` returns the contents of *mylist* at index 1. Lists are zero-based.

A dictionary is a data structure somewhat similar to lists but which do not represent sequences of variables. Here is an example for dictionaries:

```
>>> mydict = {'name': 'John Doe', 'age': 32, 'gender': 'male'}
>>> mydict['name']
John Doe
>>> mydict['age']
32
>>> mydict['age'] = 32 * 2
>>> mydict['age']
64
```

Similarly to lists, you can access the individual contents of a dictionary using the index value between brackets. But unlike lists, you can also use strings or other objects as *key*.

Python modules

Python provides different functionalities organized by *modules*. A module is a file containing Python definitions, statements and functions.

Python comes with a library of standard modules which provides many functionalities. To access a module you must explicitly import it using the *import* command. Here is an example of importing the *math* module to use some of its functions:

```
>>> import math
>>> math.factorial(10)
3628800
>>> math.log(20)
2.995732273553991
```

The Python Library Reference describes the standard library that is distributed with Python, and can be found at https://docs.python.org/3.4/library/index.html.

More about Python

The purpose of this tutorial is just to give you enough knowledge on Python to get you started with *flatlib*. There are many more tutorials on the internet which may give you deeper knowledge of the Python programming language.

The Python community provides an official tutorial which can be found at https://docs.python.org/3.4/tutorial/index.html.

1.2.2 Creating a Chart

The goal for this tutorial is to help you create a Chart for a specific date and location. To build a Chart it will be necessary to define:

- the date and time, given by the *Datetime* object.
- the geographic position, given by the *GeoPos* object.

Datetime

The *Datetime* class represents a specific moment in time given by a *Date*, a *Time*, an *UTC offset* and the calendar type. It assumes, by default, the Gregorian calendar.

To create a Datetime object, we must first import it. Here's an example that creates a Datetime object for the 13th of March of 2015 at 5pm, assuming UTC+0:

```
>>> from flatlib.datetime import Datetime
>>> date = Datetime('2015/03/13', '17:00', '+00:00')
>>> date.jd
2457095.2083333335
```

The *jd* attribute (as in `date.jd`) returns the Julian Date.

The time and UTC offset parameters are optional, and the arguments could be given as lists instead of strings. Some alternative ways to build the same date object are:

```
>>> # No UTC Offset argument
>>> date = Datetime('2015/03/13', '17:00')
>>> date.jd
2457095.2083333335

>>> # Build date with date and time lists
>>> date = Datetime([2015,3,13], ['+',17,0,0])
>>> date.jd
2457095.2083333335
```

The Datetime object provides properties and functions which may be useful for some situations:

```
>>> # Print date, time and offset
>>> print(date.date)
<2015/03/13>
>>> print(date.time)
<17:00:00>
>>> print(date.utcoffset)
<00:00:00>

>>> # Other properties
>>> date.date.dayofweek()
5   # 5 is Friday
```

```
>>> date.time.toList()
['+', 17, 0, 0]
```

GeoPos

The *GeoPos* class represents a geographic position on Earth given by a latitude and longitude. To create a GeoPos object, we must first import the class definition and instantiate an object. Here's an example:

```
>>> from flatlib.geopos import GeoPos
>>> pos = GeoPos('38n32', '8w54')
>>> pos.lat
38.53333333333333
>>> pos.lon
-8.9
```

When building the geopos object, the first parameter must be the latitude and the second the longitude. The latitude and longitude properties can be accessed directly (using `pos.lat` and `pos.lon`). Northern latitudes and eastern longitudes have positive values, while southern latitudes and western longitudes have negative values.

Alternative ways to build a Geopos object can be:

```
>>> # Using angle strings
>>> pos = GeoPos('+38:32','-8:54')
>>> pos.lat, pos.lon
(38.53333333333333, -8.9)

>>> # Using angle lists
>>> pos = GeoPos(['+',38,32], ['-',8,54])
>>> pos.lat, pos.lon
(38.53333333333333, -8.9)

>>> # Using the float values
>>> pos = GeoPos(38.53333333333333, -8.9)
>>> pos.lat, pos.lon
(38.53333333333333, -8.9)
```

Chart

The *Chart* class represents an Astrology chart for a specific datetime and geographic position. To create a chart object, we must create the Datetime and GeoPos objects and pass them as arguments to the Chart:

```
>>> # Set datetime and position
>>> from flatlib.datetime import Datetime
>>> from flatlib.geopos import GeoPos
>>> date = Datetime('2015/03/13', '17:00', '+00:00')
>>> pos = GeoPos('38n32', '8w54')

>>> # Finally create the chart
>>> from flatlib.chart import Chart
>>> chart = Chart(date, pos)
```

By default, the chart will include only the Traditional planets (*Sun* to *Saturn*, including *Pars Fortuna* and the Moon nodes) and the *Alcabitius* house system. To create a chart with other parameters, we must first import the **flatlib.const** module (where some things are defined) and pass some arguments in the object constructor:

```
>>> from flatlib import const

>>> # Build a chart with Regiomontanus houses
>>> chart = Chart(date, pos, hsys=const.HOUSES_REGIOMONTANUS)

>>> # Build a chart including modern planets
>>> chart = Chart(date, pos, IDs=const.LIST_OBJECTS)

>>> # Build a chart with only the Sun and Moon
>>> chart = Chart(date, pos, IDs=[const.SUN, const.MOON])
```

In the next tutorials it will be shown how we can access the chart's properties, including objects, houses and angles.

1.2.3 Chart properties and objects

In the previous tutorial it was shown the necessary steps to create a Chart object. In this tutorial it will be shown which properties, objects and functions are accessible on the chart. Specifically, you will learn how to access:

- Objects, houses or angles from the chart
- Fixed stars
- Other chart functions

Let's start by creating a new chart:

```
>>> from flatlib.datetime import Datetime
>>> from flatlib.geopos import GeoPos
>>> from flatlib.chart import Chart

>>> date = Datetime('2015/03/13', '17:00', '+00:00')
>>> pos = GeoPos('38n32', '8w54')
>>> chart = Chart(date, pos)
```

Objects

In *flatlib* an object is a planet, a moon node, the syzygy or pars fortuna. The following example shows how you can access an object from the chart:

```
>>> sun = chart.getObject(const.SUN)
>>> print(sun)
<Sun Pisces +22:47:25 +00:59:51>
```

In this specific example, we use the *getObject* method and say specifically which object we want to access. All objects identifiers are defined in *const.py* (see source code).

Another possibility is to use the generic *get* method, which works for objects, houses, angles and fixed-stars:

```
>>> moon = chart.get(const.MOON)
>>> print(moon)
<Moon Sagittarius +22:22:54 +13:16:01>
```

By default, when we *print* an object, it prints its identifier, the sign, sign longitude and travel speed. However, more information can be accessed from the object. Some of the available properties are:

```
>>> sun.lon
352.7901809551436
>>> sun.lat
```

```
0.00014399505974328042
>>> sun.sign
'Pisces'
>>> sun.signlon
22.790180955143626
>>> sun.lonspeed
0.9976256538994072
```

Some of the available functions are:

```
>>> sun.orb()
15
>>> sun.meanMotion()
0.9833
>>> sun.movement()
'Direct'
>>> sun.gender()
'Masculine'
>>> sun.element()
'Fire'
>>> sun.isFast()
True
```

Most of these properties and functions are self explanatory.

Houses

Similarly to objects, a list of houses is available from the chart. To retrieve an individual house, we can use the *getHouse* method or the generic *get* method:

```
>>> house1 = chart.get(const.HOUSE1)
>>> print(house1)
<House1 Virgo +03:27:30 29.39933122126604>
```

Similarly to objects, we can also access the properties of an house:

```
>>> house1.lon
153.45843823091616
>>> house1.sign
'Virgo'
>>> house1.signlon
3.4584382309161583
>>> house1.size
29.39933122126604
```

or its functions:

```
>>> house1.condition()
'Angular'
>>> house1.gender()
'Masculine'
```

Houses provides also interesting functions to check if an object is in a house, such as:

```
>>> house1.hasObject(sun)
False
```

Angles

In some house systems, such as *Equal* or *Whole sign houses*, there is a clear distinction between the *Ascendant* and *MC*, and the 1st and 10th house cusps, hence the necessity of angles. To retrieve an angle from the chart you can use the *getAngle* method or the generic *get* method:

```
>>> asc = chart.get(const.ASC)
>>> mc = chart.get(const.MC)
>>> print(asc)
<Asc Virgo +03:27:30>
>>> print(mc)
<MC Taurus +29:19:03>
```

Similarly to objects and houses, some properties and functions are also available for angles.

Fixed-stars

To retrieve fixed stars from the chart, we must use the *getFixedStar* method:

```
>>> spica = chart.getFixedStar(const.STAR_SPICA)
>>> print(spica)
<Spica Libra +24:03:34 0.97>
>>> spica.mag  # magnitude
0.97
>>> spica.orb()
7.5
```

The list of avaliable fixed stars are defined in the source code.

Lists

In some cases, instead of retrieving objects, houses or angles one by one, it may be useful to get direct access to their lists. The *chart* object provides the following lists:

- *chart.objects*, with a list of all objects
- *chart.houses*, with a list of all houses
- *chart.angles*, with a list of all angles

The following example uses the `for` command to iterate over all objects in the list of objects:

```
>>> for obj in chart.objects:
...     print(obj)
...
<Moon Sagittarius +22:22:54 +13:16:01>
<Venus Aries +25:30:11 +01:12:41>
<Saturn Sagittarius +04:55:45 +00:00:06>
<Mercury Pisces +00:48:57 +01:29:49>
<North Node Libra +11:08:28 -00:03:11>
<Syzygy Virgo +14:50:23 +11:48:44>
<Sun Pisces +22:47:25 +00:59:51>
<South Node Aries +11:08:28 -00:03:11>
<Pars Fortuna Gemini +03:03:00 +00:00:00>
<Mars Aries +16:32:48 +00:45:18>
<Jupiter Leo +13:38:37 -00:04:45>
```

Lists also provides us with useful functions. For instance, the house list provides a function to retrieve the house where an object is:

```
>>> house = chart.houses.getObjectHouse(sun)
>>> print(house)
<House7 Pisces +03:27:30 29.39933122126604>
```

In this specific case, the sun is in the 7th house. The lists.py file provides a full overview of what is available for each list.

Chart functions

Besides the functions to retrieve objects, houses, angles and fixed-stars, the chart object provides other useful functions:

```
>>> chart.isDiurnal()
True
>>> chart.getMoonPhase()
'Third Quarter'
```

Finally, the chart object also provides a useful function to retrieve the solar return chart for a year:

```
>>> srchart = chart.solarReturn(2020)
>>> print(srchart.date)
<2020/03/12 22:01:59 00:00:00>
```

1.3 Frequently Asked Questions

Can everyone use it?

Flatlib is open-source software so everyone is free to install and use it. However, since it is a programming library, some people may not be particularly inclined to use it since it requires some learning.

So is it not a end-user tool?

Flatlib should really be seen as a traditional astrology software without a graphical user interface. Therefore, it is really powerful, since users can experiment without the "chains" of a graphical user interface.

How can I install it?

Documentation is still scarce, but I hope to improve it in the future. You should install the latest Python 3 (3.4) and grab the flatlib package with `pip3 install flatlib`. This will install flatlib and its dependencies.

Is there a project page?

You can check the code and simple documentation in the github page at https://github.com/flatangle/flatlib.

Are there any sample code?

There's a "recipes" folder with some source code at https://github.com/flatangle/flatlib/tree/master/recipes. You can start with "aspects.py" which is at https://github.com/flatangle/flatlib/blob/master/recipes/aspects.py.

Can I use it on my own work?

Absolutely yes, you are free to use it in your own projects. The flatlib source code is released under an MIT License, which allows it to be used also on commercial projects. There is a caveat though: flatlib uses the swiss ephemeris which is licensed GPL. Therefore, if you want to use flatlib in your commercial projects, you must adhere to the GPL license or buy a Swiss Ephemeris commercial license.

Why are you open-sourcing flatlib?

I really want to help push forward data-driven research in astrology. That is only possible with a strong community of researchers and good tools, so there was no point in keeping flatlib hidden on my hard disk. Flatlib is also a good tool for talking about astrology techniques. Someone can always point to the source code to explain how things can be done.

Can I contribute to the project?

I accept contributions such as code and documentation, although I suggest to wait a while since things are not stable yet. The best contribution for now is to spread the news about the existence of this project.