Python for Dummies

Sebastian Hahn

Department of Geodesy and Geoinformation (GEO), TU Wien
http://www.geo.tuwien.ac.at/

Topics

- Introduction
- Development Environment
- Python Data Types
- Control Structures and Functions
- Summary
- Live Demo

INTRODUCTION TO PYTHON

Language overview

Central website: http://www.python.org/

- Created by Guido van Rossum and first released in 1991
- Python has a design philosophy that emphasizes code readability
  - notably using significant whitespace
- General purpose
  - can write anything from websites (Youtube, Reddit…) to scientific code
- High Level
  - Strong abstraction from inner workings of the computer: e.g. no memory management
- Interpreted
  - executes instructions directly, no compilation
- Multi paradigm
  - Object oriented, functional, imperative or procedural styles are supported

Language overview

- Dynamically typed
  - Variable types are checked during execution
- Modular
  - Python modules must be imported to be used
- Multi Platform
  - Works on Linux, Mac OS, Windows, …
- Batteries included
  - Powerful standard library (e.g. file reading, URL parsing…)

Different Main Versions and Implementations

- Currently Python 2.7 and 3.7 are the latest versions
  - **Python 2.7 will not be maintained past 2020**
    - https://docs.python.org/3/howto/pyporting.html
- Python 3 introduced some incompatible changes
  - Nowadays, most 3rd party packages either work with Python 3 or both versions.
- Reference implementation: CPython, download on python.org
- Others: IronPython, Jython, PyPy, Stackless Python,…
- Open Source
  - Often, there are several packages available that serve the same goal

Python Enhancement Proposals (PEPs)

- https://www.python.org/dev/peps/
  - PEP001 PEP Purpose and Guidelines
  - PEP004 Deprecation of Standard Modules
  - PEP005 Guidelines for Language Evolution
  - PEP006 Bug Fix Release
  - PEP007 Style Guide for C Code
  - **PEP008 Style for Python Code**
  - PEP010 Voting Guidelines
  - …

DEVELOPMENT ENVIRONMENT

Development environment

- Theoretical minimum:
  - Python installation
  - Text editor
- Suggested IDE (Integrated Development Environment) for Win/Mac/Linux:

- Different versions: Professional (89€/year, free for students), Community (free), Edu (free)
- https://www.jetbrains.com/pycharm/download/

Development environment

- Other IDE
  - Spyder (written in Python)
  - Pydev (Plugin for Eclipse)
  - Pyscripter (Windows only)
  - Komodo
  - …
- IDE helps with
  - Syntax highlighting
  - Code refactoring
  - Version control
  - Debugging
  - Code search
  - …

Python installation

- Python 2.7/3.6 with Anaconda
  - Available for Win/Mac/Linux
- Multiple installations of python on the same computer (e.g. different versions, …)
  → “environment”
- Two ways to install new packages and to create new environments:
  - GUI (“Anaconda Navigator”)

  - Command line (“Anaconda prompt”)

```
conda create --name projektX python=2.7
activate projectX
conda install matplotlib
deactivate
```

Miniconda installation

- **wget** https://repo.continuum.io/miniconda/Miniconda2-4.5.11-Linux-x86_64.sh -O miniconda.shbash
- **miniconda.sh** -b -p $HOME/hsaf_conda
- **export** PATH="$HOME/hsaf_conda/bin:$PATH„
- **conda** create -n work_env -c conda-forge numpy scipy pandas matplotlib rasterio geopandas netCDF4 pyflakes statsmodels cartopy basemap basemap-data-hires cython h5py jupyter pybufr-ecmwf pykdtree pygrib pyresample python=2
- **source** activate work_env
- **pip** install ascat pytesmo

pip vs. conda

- The choice between pip and conda can be a confusing one, but the essential difference between the two is this:
  - pip installs python packages in any environment
  - conda installs any package in conda environments

- If you already have a Python installation that you're using, then the choice of which to use is easy:
  - If you installed Python using Anaconda or Miniconda, then use conda to install Python packages. If conda tells you the package you want doesn't exist, then use pip (or try conda-forge, which has more packages available than the default conda channel)
  - If you installed Python any other way (from source, using pyenv, virtualenv, etc.), then use pip to install Python packages

- Finally, because it often comes up, never use **sudo pip install**

PYCHARM IDE

Getting started with PyCharm

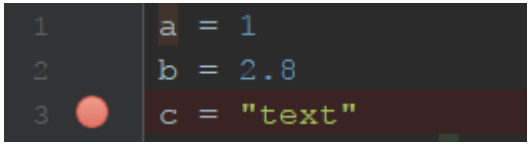
- pyCharm organizes code in “Projects”
- This whole class can be a project
- Anaconda creates a “root” environment by default
- pyCharm may select this automatically, otherwise it may be set at
  `File → Settings … → Project → Project Interpreter`
- New packages can also be installed there
  (however, the Anaconda Navigator is the suggested way)
- The environment can also be set for each file at
  `Run → Run configurations → Python interpreter`
- To run the current file: Right-Click into the editor and press 

- To run again: [Ctrl]+[F5], or 

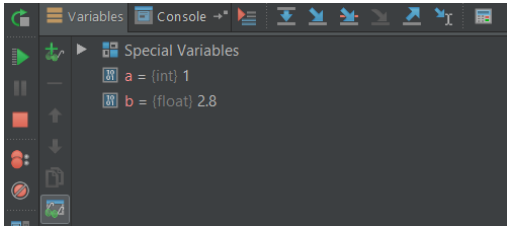
Debugging

- De-bug: remove bugs
- pyCharm: Instead of Run, press the Debug button 

- Code will run like normal, until a breakpoint is encountered
  - Set breakpoints by clicking left of code
  - Remove by clicking again



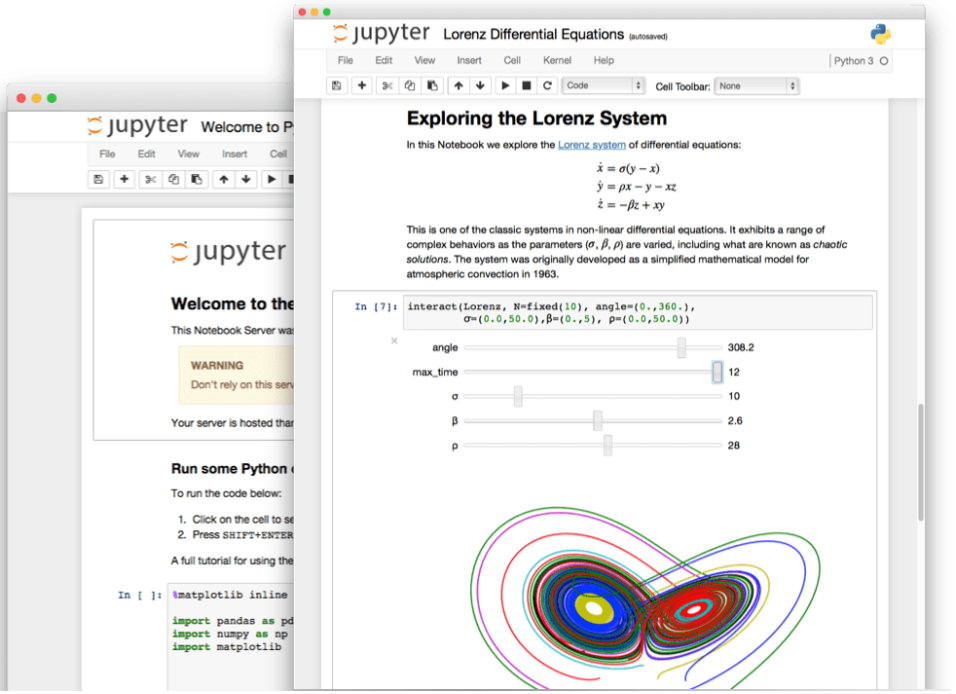
- Before the line containing the breakpoint is evaluated, the execution is halted and values of variables are shown



JUPYTER

Jupyter notebook

- Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.
- http://jupyter.org/



Data types, collections, control structures

PYTHON LANGUAGE

Built-in primitive data types (immutable)

- Numeric types:
  - Integers (“int”): `-1, 0, 1, 2, 3, …`
  - Floats (“float”): `-120.0, 3.141, 1.5e12, 3.00e8`
  - Complex (“complex”): `1.0j, 1+1j, 3e8+0j`
- Binary types:
  - Boolean (“bool”): True/False
    - `bool(0)` → `False; bool(1)` → `True`
    - `bool("0")` → `True, bool("")` → `False`
- String types:
  - String (“str”): `'Hallo', "Hallo", "“`
- NoneType
  - `x = None`

Collections

- Collections can be used when we have more than one value
- Python provides the following collections built-in (more are in the Collections package):
  - Lists
    - ordered
  - Sets
    - only unique entries are allowed
    - unordered
  - Tuples
    - are immutable (cannot be changed)
    - ordered
  - Dictionaries
    - pairs of key and value
    - unordered (generally)

```
a_list = [1, 2, 3, 3, ,four']
```

```
a_set = set(a_list)
>>> a_set
{‘four', 1, 2, 3}
```

```
a_tuple = (1, 2, 3, 3, ,four')
```

```
a_dictionary = {
1: 'one',
2: 'two',
3: 'three',
4: 'four'
}
```

Accessing Collections

- Ordered Collections (not sets) can be accessed by index:
  - `a_list[2]` → `3  # python indices start with 0`
- To change values of a collection, just overwrite the value:
  - `a_list[3] = 'three'`
- This cannot be done with tuples:
  -
```
>>> a_tuple[2] = 'three'
    Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
    TypeError: 'tuple' object does not support item
assignment
```
- Add items to a list using .append(item):
  - `a_list.append(5)`
- To combine two lists, add them together:
  - `b_list = [6,7,8]`
  - `lists = a_list + b_list ->[1, 2, 3, 'three', 'four', 5, 6, 7, 8]`

References in Python

- Imagine a list that contains other variables:

```
a = 2
b = 3
my_list = [a, b]
```

- What happens if you change a afterwards?

```
a = a + 1  # or a += 1
print(my_list) → [2, 3]
```

- This only works for immutable types (int, float, string, …). If we have mutable types (e.g. lists, dicts, sets), we get a different behaviour:

```
list1 = [1, 2]
list2 = list1
list3 = [1, 2]  # new list
list1.append(3)
list2 → [1, 2, 3]
list3 → [1, 2]  # not changed
```

References in Python

- The use of id(..) helps to clear this up

```
a = 5
id(a) → 1362872176
a += 1
id(a) → 1362872208

my_list = [1,2,3]
id(my_list) → 2188976983112
my_list.append(4)
id(my_list) → 2188976983112
```

- Mutable types do not change id, immutable ones do!
- For reference:

https://codehabitude.com/2013/12/24/python-objects-mutable-vs-immutable/

Standard operators (Python as a calculator)

- Addition, Subtraction, Multiplication: +, -, *
- Exponentiation: a**b = a^b
- Square root: a**(1/2)
- Division: Float vs. Integer division (New behaviour in python 3.x)
  - 1 / 2 → 0.5
  - 1 // 2 → 0
  - 1.0 / 2.0 → 0.5
  - 1.0 // 2.0 → 0.0
- % is the modulo operator: division remainder
  - Useful to decide if a number is even or odd!
- `in` checks if a value is within a collection:

```
names = ['john', 'smith']
'john' in names → True
```

- Case sensitive!

Boolean operators

- `==` returns True if two variables have the same value
- `!=` returns True if the do not have the same value
- `>=, <=, <, >` check for inequalities
- `is` returns True if the two variables refer to the same object (not a copy)
  - Careful! This leads to interesting results with immutables:

```
a = 10; b = 10
a is b → True
l1 = [10]
l2 = [10]
l1 is l2 → False (but l1 == l2→ True)
```

String formatting, list slicing

- Strings represent text
- Strings can be concatenated (stitched together) by + or +=:

```
my_string = 'Welcome'
my_string += ' to this class! '
```

- Line breaks can be represented as `' \n'` (new line)
- Strings are lists of characters:

```
my_string[2] → ‘l’
```

- Strings (and lists) also support *slicing*:

```
my_string[start:stop:step=1]
my_string[0:1] → 'W'
my_string[5:] → 'me to this class!'
my_string[::2]  → 'Wloet hscas'
my_string[::-1] → '!ssalc siht ot emocleW’
my_string[:-5] → 'Welcome to this c’
```

String formatting

- Strings have the .format() – Function which allows to pass variables of other types
- For this, the string must contain {}-Braces at the positions

```
name1 = “John"
name2 = "Doe"
"Hello, {} {}!".format(name1, name2)
```

- Alternatively, the string may contain %-Characters and a % (value1, value2, …) at the end (this is the old way to do it)

```
"Hello, %s %s" % (name1, name2)
```

- Or prepend the string with “f” and put the variables to print directly into the placeholders:

```
f”Hello {name1} {name2}!”
```

- Or pass the arguments by name, or use the indices to access them

```
"Hello, {first} {last}".format(first=name1, last=name2)
"Hello, {0} {1}!".format(name1, name2)
```

- For a full reference, see: https://pyformat.info/

String formatting

- When converting values (float, integer, …) to string, additional options may be given:
- Padding adds spaces left and/or right of the value (useful for tables):

```
"{:10}".format("Hi.") → 'Hi.       '
"{:>10}".format("Hi.") → '       Hi.'
"{:^10}".format("Hi.") → '   Hi.    '
```

- Print numbers with a certain precision:

```
"{:5.3f}".format(3.14159265) → '3.142'
```

(5 digits total, 3 after the comma)

- Leading zeros are possible:

```
"{:06.1f}".format(3.14159265)  → '0003.1'
```

- Integers can be converted; with or without leading sign

```
"{:03d}".format(42) → '042'
"{:+03d}".format(42) → '+42'
```

CONTROL STRUCTURES AND FUNCTIONS

Conditional statements

- General idea: Do something only if something else is true.

```
if condition:
    statement1
    statement2
    ...
```

„Block“

- Indentation is optional in many programming languages, but not in python!
- Suggested indentation: **4 spaces**
- Block ends, where indentation ends, e.g.

```
a = 2
a += 1
if a == 3:
    print("increasing a by 1")
    a += 1
print(a)
```

Execute anyway

Execute only if the value of a is equal to 3

Execute anyway

if / elif / else

- An `if`-Statement may contain multiple `elif` and one `else`-Block:

```
if condition1:
    do something
elif condition2:
    do something else
elif ...
    ...
else:
    do something if none of the other cases have occurred
```

- This prevents long lists of `if`-Statements

pass

- When writing a program, some options are left for later implementation
- Python needs an indented block after every if/elif/else-Line
- Use `pass` as a placeholder. `pass` does nothing:

```
if a > 0:
    print("a is bigger than zero")
elif a == 0:
    print("a is zero")
else:
    pass  # we'll do that later
```

  - This is also useful if you want to have clearer conditional expressions:

```
if type(a) == str or type(a) == int:
    pass
else:
    print("A is neither a string nor an integer!")
```

- **„Readability counts.“** - `import this` (Easter Egg)

Simplifying conditions on collections

- Sometimes it is useful to know if (at least) one element of a list converts to `True`, or if all of them convert to `True`.
- For example, check if there is a number 0 in the list:

```
a_list = [-1, 0, 1, 2, 3]
if all(a_list):
    pass
else:
    print("There is a zero somewhere")
if any(a_list):
    print("There is at least one non-zero element in the list")
```

- Checking general expressions on list items involves list comprehension
(taught in two weeks). For reference:

```
if any(val > 2 for val in a_list):
    print("There is at least one value larger than 2 in the list")
```

Loops

- Code can be carried out multiple times using loops
- 2 types of loops: `while` and `for`

- `while`-loops:
  - Execute a code block until a requirement (boolean expression) is no longer met
  - Typically used when the number of iterations is not clear beforehand

- `for`-loops:
  - Execute a code block for a specific number of times
  - This number is usually known in advance
  - Alternatively: Execute a code block for every element of a list (or any iterable)
  - python for-Loops are always „foreach“-loops!

for - loop

- We want to add up the numbers in the list (without using `sum()`):

```
nums = [-1, 0, 1, 2, 3]
num_sum = 0
for current_number in nums:
    num_sum += current_number
```

- The `range` function creates a generator (for now: like a list) containing whole numbers:

```
range(start, stop, step=1)
range(stop)
range(5) → [0,1,2,3,4]
range(5,7) → [5,6]
range(5,10,2) → [5,7,9]
```

- Use `range` for creating indices in loops:

```
for i in range(10):
    print(i)
```

for loops on dictionaries

- With dictionaries, it is possible to loop over key and value simultaneously:

```
a_dict = {1: 'one', 2: 'two', 3: 'three'}
for (key, val) in a_dict.items():
    print("The word for {} is {}.".format(key, val))
```

- Loop over keys:

```
for key in a_dict: # or: a_dict.keys()
    pass
```

- Loop over values:

```
for val in a_dict.values():
    pass
```

while - loop

- Similar to if-Statements, while-Loops have a boolean expression:

```
while expression:
    do something
```

- The loop is carried out as long as `expression` evaluates to `True`
- Example: Find n prime numbers:

```
n = 10
curr_num = 1
curr_count = 0
while curr_count < n:
    if is_prime(curr_num):
        print(curr_num)
        curr_count += 1
    curr_num += 1
```

Loops: break and continue

- When a loop should be terminated, use `break`
- To skip the current iteration and proceed with the next, use `continue`
- Example for break:

```
name = „John"
for c in name:
    if c == 'h':
        break
    print(c)
```

```
J
o
```

- Example for continue:

```
for c in name:
    if c == 'h':
        continue
    print(c)
```

```
J
o
n
```

Loops: break and else

- Else can be used to check if a break-Statement has terminated the loop
- Let‘s find some primes:

```
for n in range(2, 8):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n / x)
            break # breaks out of (ends) current loop
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

Changing list items while iterating them

- Be careful when trying to change list items while iterating them:
  - If the items are immutable, they are passed by value and cannot be changed
  - If they are mutable (e.g. lists), they are passed by reference

```
list1 = [1,2,3]
list2 = [[1], [2], [3]]

for item in list1:
    item += 1  # list1 remains unchanged

for item in list2:
    item.append(item[0])  # list2 is changed
```

list2 → `[[1, 1], [2, 2], [3, 3]]`

Functions

- A function is a piece of code that can be used multiple times
  - can have input and output, both optional
- Functions are declared using `def`

```
def function_name(arg1, arg2, arg3=default_value, ...):
    do something
    return value
```

- Parameters with defaults must come after those without!
- You can run the function in the „normal“ code

```
a_value = function_name(input1, input2, input3)
a_value = function_name(input1, arg3=input3, arg2=input2)
a_value = function_name(arg3=input3, arg2=input2,
arg1=input1)
a_value = function_name(input1, input2)
a_value = function_name(arg1=input1, input2, input3)
```

- Keyword arguments (kwargs) always come **after** positional arguments!

Function examples

- Example for a function to return the median value of three (pairwise different) values:

```
def median(first, second, third):
    itemlist = [first, second, third]
    itemlist.sort()
    return itemlist[1]

print(median(9.23, 1, 2))
```

- Functions have to be defined **before** they can be used
- When manipulating items within a function, remember:
  - Mutable types get passed by reference (and can be changed within the function)
  - Immutable types get passed by value (changes within the function will not reflect outwards)

Function - *args

- We might want to have a function take an arbitrary amount of arguments (median of n values)
- The special syntax **`*args`** collects all arguments in a tuple called „`args`“

```
def median(*args):
    items = list(args)
    items.sort()
    return items[len(items)//2]

print(median(9.23, 1, 2, 2, 2, 2, 2))
```

- You can also combine „normal“ arguments and the argument list

```
def quantile(quant, *args):
    items = list(args)
    items.sort()
    return items[int(quant * len(items))]
```

Using *args in function calls

- The use of *args is also possible „the other way round“
- E.g. when we want to use a list for a couple of parameters:

```
def add2numbers(number1, number2):
    return number1+number2

numbers = [5, 6]
# expands to add2numbers(numbers[0], numbers[1]
add2numbers(*numbers)
```

- For Geo-Applications very useful when handling (3D) Points:
  - `function(x, y, z)` vs. `function(p)` with `p = [x, y, z]`

Function - **kwargs

- It‘s also possible to get arguments as a dictionary
- **`**kwargs`** (keyword arguments)
- In this context, **`get(key, default)`** is a useful function on dictionaries
  - If the key is in the dict, return the value, otherwise return the default

```
def a_function(**kwargs):
    x_coord = kwargs.get('x', 0.0)
    y_coord = kwargs.get('y', 0.0)
    z_coord = kwargs.get('z', 0.0)
```

- This can also be used „the other way round“, e.g. for string formatting

```
lecturer = {'first': 'John', 'last': 'Smith'}
"Hello, {first} {last}!".format(**lecturer)
```

3RD PARTY LIBRARIES

Important/helpful 3rd party libraries

- numpy - fundamental package for scientific computing with Python containing among other things: a powerful N-dimensional array object
- scipy - Includes modules for graphics and plotting, optimization, integration, special functions, signal and image processing, genetic algorithms, ODE solvers, and other
- pandas - Python Data Analysis Library
- matplotlib - Production quality output in a wide variety of formats
- basemap - Plotting maps (development stops in 2020)
- cartopy - designed for geospatial data processing in order to produce maps and other geospatial data analyses.
- Geopandas - working with geospatial data, combines the capabilities of pandas and shapely
- Rasterio - provides access to geospatial raster data

- dask - provides advanced parallelism for analytics
- xarray - N-D labled arrays and datasets
- scikit-learn - Machine learning in Python
- statsmodels - provides many opportunities for statistical data analysis
- Many more: Plotly, Bookeh, seaborn, scrapy

SUMMARY

Where to go from here?

- Many online tutorials and books
  - https://docs.python.org/3/tutorial/
  - https://www.tutorialspoint.com/python/
  - https://www.learnpython.org/
  - https://wiki.python.org/moin/PythonBooks
  - https://docs.python-guide.org/intro/learning/
  - https://www.youtube.com/results?search_query=python

- Got a Python problem or question?
  - Check the Python FAQs, with answers to many common, general Python questions.
  - Stackoverflow
  - Google

Version control

- Version control is a system that records changes
  - Local Version Control Systems (Copies)
  - Centralized Version Control Systems (CVS, Subversion, …)
  - Distributed Version Control Systems (Git, Mercurial, …)

- Git – book
  - Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities.
    - Speed, simple design, fully distributed, non-linear development, able to handle large projects