FS12

Week 02
Python Basics II

Mikhail Masyagin
Daniil Devyatkin

Hello, everyone!

Artur Chakhvadze
@norpadon

Друзья, если вы делаете первые шаги в программировании и вам приходится выбирать между десятком каких-то непонятных инструментов, устанавливать их по запутанным инструкциям, а потом рвать на себе волосы о того что ничего не запускается, я спешу вас ободрить:

ДАЛЬШЕ БУДЕТ ХУЖЕ

04:01 · 24.08.2023 из: Earth · Просмотров: **16K**

Programming paradigm

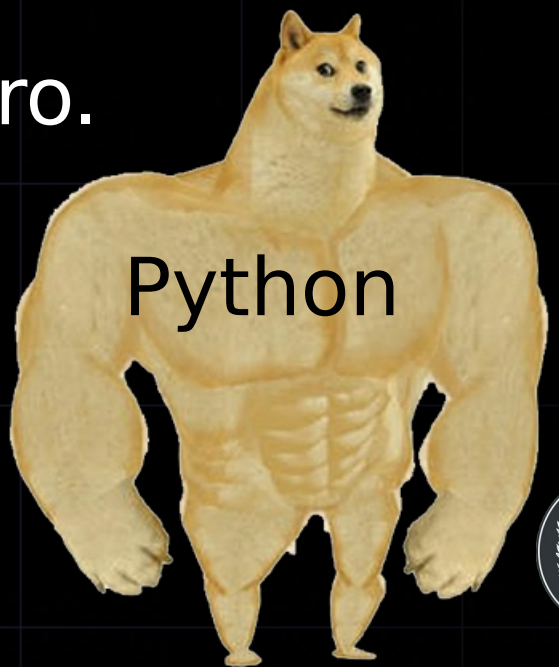
- Procedural Programming (PP);
- Functional Programming (FP);
- Object Oriented Programming (OOP) [next lesson];
- etc.

Functional programming

- FP is a programming paradigm that focuses on using functions the building blocks of a program and avoiding mutable data and state;
- Programs are built from the composition of functions, which return result;
- Variable is immutable, no cycles, recursion;
- FP has a good mathematicians basis. (lambda calculus)

FP in Python: # True or False?

- Python is multilateral language and it can FP;
- Base standard function for FP: `map`, `reduce`, ‘filter’, and lambda functions;
- You can use functools library for research more functions;
- But before lambda functions intro.



FP in Python: lambda functions

- You can define function in this way:

```
>>> def func(a, b, c): pass
```

- But you can use lambda (anonymous) functions for clean development. Syntax:

```
>>> lambda <args>: <one string logic>
```

- Example:

```
>>> (lambda x, y: x + y)(2, 3) # 5
>>> t = [(2, “v”), (1, “d”), (5, “a”)]
>>> sorted(t, key = lambda x: x[1])
# [(5, “a”), (1, “d”), (2, “v”)]
```

FP in Python: map

- `map` is function, which apply some function to iterable collecton, return collection like input;
- `map(<func>, <iterable>...)`;
- Examples:

```
>>> map(int, [“1”, “2”, “3”]) # [1, 2, 3]
>>> map(lambda x: x.lower(), [“ABCD”, “S”])
# [“abcd”, “s”]
>>> map(lambda x, y: x + y, [1, 2, 3], [4, 5, 6])
# [5, 7, 9]
```

FP in Python: reduce

- `reduce` function applies the some function to the elements of a sequence, reducing it to a single value, return value;
- `reduce(<func>, <iterable>, <init>)`;
- Examples:

```
>>> lst = [1, 24, 17, 14, 9, 32, 2]
>>> cond = lambda a,b: a if (a > b) else b
>>> reduce(cond, lst, 0) # 32
```

FP in Python: filter

- `reduce` function filter a sequence be some condition;
- `reduce(<func>, <iterable>)`, func should return bool value;
- Examples:

```
>>> lst = [1, 24, 17, 14, 9, 32, 2]
>>> cond = lambda x: bool((x + 1) % 2)
>>> list(filter(cond, lst)) # [24, 14, 32, 2]
```

Modules & Packages I: Modules

- A **module** in Python is a file with .py extensions.

  It just encapsulation method for readability code;
- Example of module import:

```
>>> import math
>>> math.sqrt(4) # 2
```

- Example of alias:

```
>>> import math as  m
>>> m.sqrt(4) # 2
```

Modules & Packages I: Modules

- Example of import concrete function:

```
>>> from math import sqrt
>>> sqrt(4) # 2
```

- Also you can use this set up:

```
>>> from math import sqrt as s
>>> s(4) # 2
```

- Import all:

```
>>> from math import *
```

- You can write self-module and import it like others.

Modules & Packages II: Packages

- If you want write self-library, you should know about packages;
- A **package** in Python is a directory that includes other directories, modules and contains an *__init__.py* file;
- Example:

```
helloworld-project
|---- helloworld
|       |-- __init__.py
|       |_ core.py
|-- setup.py
```

Modules & Packages II: Packages

- When importing a package (import package, from my_cool_lib import package) only *__init__.py* is imported (everything that is written inside is performed);
- Constructor for package, configurate something before import.

```
from ._dict_vectorizer import DictVectorizer
from ._hash import FeatureHasher
from .image import img_to_graph, grid_to_graph
from . import text

__all__ = ['DictVectorizer', 'image', 'img_to_graph', 'grid_to_graph', 'text',
           'FeatureHasher']
```

Modules & Packages II: Packages

- But, what is magic *setup.py* file in **helloworld-project**?;
- A *setup.py* is special service file for package manager (pip) and install package;
- What *setup.py* file (can) contain:
  - Version;
  - Required packages;
  - Python version;
  - License.

Modules & Packages II: Packages

- Distribution:
  - A project containing setup.py can be built: ‘python setup.py sdist > my_project.tar.gz’;
  - This package can be installed and redistributed:

    ‘pip install my_project.tar.gz’.

Modules & Packages III: __main__

- Python will run all code in a file unlike other languages, because Python don’t have entry point;

  Reminder: `python <some_name>.py`
- When the interpreter runs the file as the main program, it sets the __name__ variable to "__main__";
- If we have some executable code in module, and we need execute it, when this module is file of main program and don’t execute else, we write next condition: [next slide]

Modules & Packages III: __main__

```
# ex1.py
...
<some module logic>
...
if __name__ == “__main__”:
    <some_executable_code>
```

execute if the module will call like a main program:

`python ex1.py`

```
# ex2.py
...
<some module logic>
...

<some_executable_code>
```

always execute

Modules & Packages IV: requirements.txt

- We install package by pip manager;
- A few external packages are required to run the project. In order not to build them with pain every time, it is customary to supply a list of these packages along with the source code. It is customary to place the entire list of required packages in the *requirements.txt* file at the root of the project.

Modules & Packages IV: requirements.txt

- Example *requirements.txt file:*

  django==1.10.2

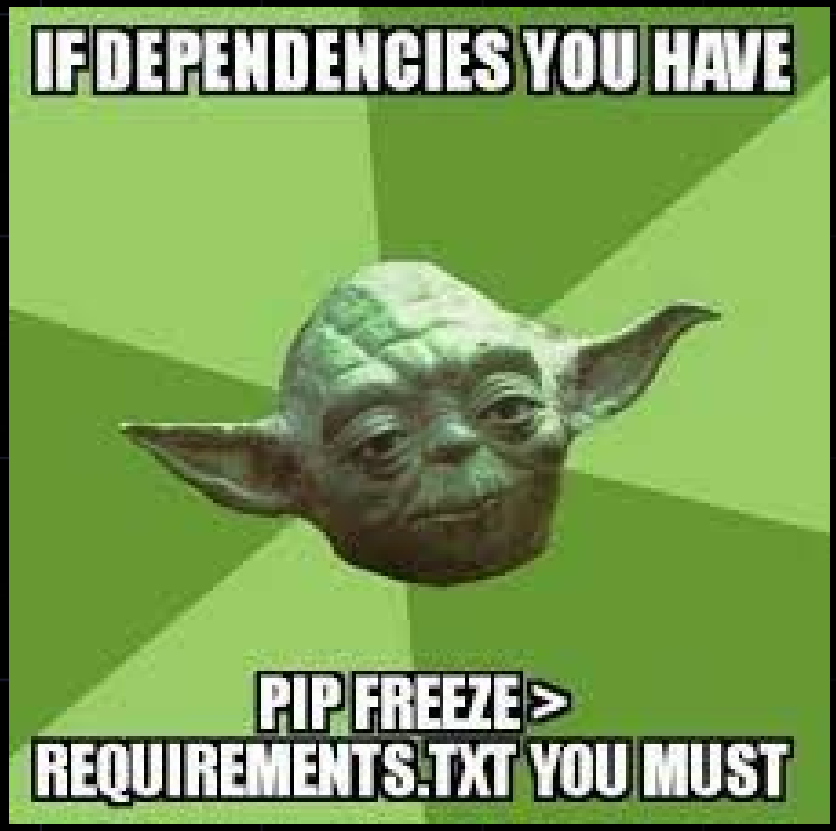
  pillow==3.3.0

  gunicorn==19.6.0

  torch==2.2.3

  sklearn==0.35

  numpy==1.4.2



- `*pip install -r requirements.txt*`

I/O & files I

- What we can do with file?
  - opening operations:
    - open;
    - close;
    - write;
    - read;
  - Operations without opening:
    - rename;
    - copy;
    - etc.

I/O & files I

- When a file is opened, the OS receives a special fd that uniquely determines which file will be used next operation;
- In Python, interaction with files is carried out through a special abstract file object;

  Syntax and args:

  ```
  open(file, mode='r', buffering=-1, encoding=None,
  errors=None, newline=None, closefd=True, opener=None)
  ```

  Return file’s descriptor.

I/O & files II

```
f = open("some.file", "r")
data.write(“some text”)
data = f.read()
f.close()



# better
with open("some.file", "r") as f:
        data.write(“some text”)
        data = f.read()
```

I/O & files II: Mode

Mode	Description
“r”	Read only.
“w”	Write only. the contents of the file are deleted, if the file does not exist, a new one is created.
“rb”	Analogue “r” mode for binary format.
“wb”	Analogue “w” mode for binary format.
“r+”	“r” + “w” mode
“a”	For writing, the information is added to the end of the file.
“x”	To write if the file does not exist, otherwise an exception will be thrown.

I/O & files III

- You can display something in terminal (console) using `print()` function, but this isn’t limited to the terminal.
- Syntax `print()` function:

  print(value, ..., sep='', end='\n', file=sys.stdout, flush=False)
- file is a file-like object (stream). The default is sys.stdout (can change to stdin or stderr). Here you can specify the **file** to which you want to write or add data from the print function. You can save output to file!

I/O & files: meme