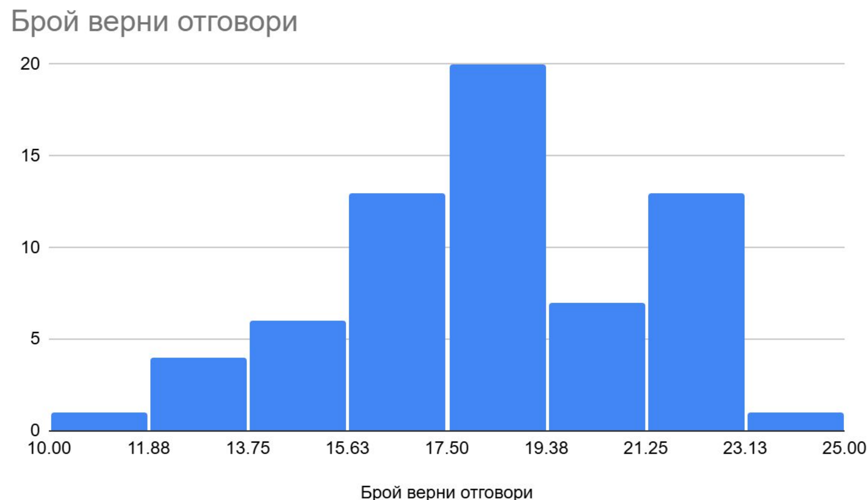
10. Модули и среди

14 ноември 2024

Малко за контролното

- Първо, някой хвана ли хинт-а в новината?
- Някои от вас вече си разбраха резултатите
- Останалите - ще могат след малко
- А като статистика:



Грешка №1

Да си припомним

Истина и лъжа

В контекста на булевите операции като лъжа се интерпретират следните стойности:

- `False`
- `None`
- числото `0` независимо от типа числа (например `0, 0.0, 0j`)
- празният низ
- празни контейнери (`tuple, list, dict, set`)
- наши типове могат да дефинират как да бъдат оценявани като булеви променливи

Всички останали стойности се интерпретират като истина.

Малко за предизвикателството

- Няколко чести грешки:
  - Излишни променливи
  - Кофти “инициализиране” на “нулеви” стойности
  - *Излишни проверки за тип*

Излишни променливи

```
position = exc.position
exp_type = exc.expected
args[position - 1] = exp_type()
```

Излишно, следното е също толкова четимо:

```
args[exc.position - 1] = exc.expected()
```

Изключение е ако името на променливата дава важна информация:

```
parsed_error_logs = prettify('\n'.join(self.filter_log(self.log_data)))
with open('log.txt') as log_file:
    log_file.write(parsed_error_logs)
```

Нулеви стойности

```python
def __generate_value_of_type__(self, expected_type):
    if expected_type is int:
        return 0
    elif expected_type is float:
        return 0.0
    ...
```

Да де, обаче...

```python
bytes, memoryview, bytearray, object, type(...)
# (<class 'bytes'>, <class 'memoryview'>, <class 'bytearray'>, <class 'object'>, <class 'ellipsis'>)
```

А може просто:

```python
int(), float(), bytearray(), type('asd')()
# (0, 0.0, bytearray(b''), '')
```

Модули 101

- Всяко нещо в Python е част от някой модул
- Дори нещата, които пишем в интерактивната конзола, са в модул
- Всеки файл с разширение *.py* е модул. Името на модула е името на файла (без *.py*)
- Всяка директория може да бъде модул
- Файлът, от който е стартирана програмата, е с име `__main__`
- Дори и в интерпретатора?!
- Същото е:

```
>>> print(__name__) # __main__
```

__name__

- Името на модула, duh:

```
>>> print(__name__) # __main__
```

```
>>> import unittest
>>> print(unittest.__name__) # unittest
```

- Помните ли това:

```
if __name__ == "__main__": …
```

- Е, целта е да се подсигурите, че текущият файл е изпълнен, а не импортиран

import

- Всеки модул се импортира само по веднъж
- При импортирането му се изпълнява целият файл!
- Важно е!

joey.py:

```
print("Hey, how you doin?")
```

Интерактивен интерпрататор:

```
>>> import joey
"Hey, how you doin?"
```

В този ред на мисли

Преди: `import __hello__`

Сега: `$ python -m __hello__`

Защото...

```
def main():
    print("Hello world!")

if __name__ == '__main__':
    main()
```

Какво става при импортиране на модул?

```
import random

print(random) # <module 'random' from '<some_path>'>
```

- На името `random` се присвоява стойност - обект от тип модул
- Всички “глобални имена“ в този модул стават атрибути на обекта
    - достъпват се с точка (`random.shuffle`)
- Имената са “наистина глобални“ само ако сте в `__main__`
    - ...иначе стоят зад името на модула, от който идват
        - освен ако не импортирате само част от модула, но за това след малко
- Разделянето на кода ни в модули ни позволява
    - хем да преизползваме код
    - хем да имаме еднакви имена, зад които стоят различни обекти

Още за import

```
>>> from random import shuffle
```

```
>>> import itertools as it
>>> it
<module 'itertools' (built-in)>
>>> it.__name__
'itertools'
```

```
>>> from matplotlib import pyplot as plt
```

Можете да бъркате навътре в модулите

- Пример:

```
from module.submodule.deeper.even_deeper.oh_god.please_stop import
oh_this_is_actually_not_so_bad
```

Всичко е просто namespace

- Вариант 1:

```
from django import db
my_model = db.models.Model()
```

- Вариант 2:

```
from django.db.models import Model
my_model = Model()
```

Кога какво?

- Ако ще използвате само едно-две-три неща от целия модул:
```
from module import what_i_need, and_this_other_thing
```
- Ако ще използвате много неща, очевидно:
```
import module
```
- Ако ще използвате едно нещо на много места:
```
from module import what_i_need_a_lot
```
- Следното е напълно валидно:
```
import module
from module import what_i_need
```

Можете и да изсипете всичко с лопатата

```
from panda import *
```

- По този начин всичко от panda ще бъде "изсипано" в текущия скоуп
    - Това не включва имена, започващи с една или две долни черти
- Нямате добра причина да правите това извън интерактивна конзола или unit тестове
- Не го правете

dir()

```
import itertools
dir(itertools)
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', '_grouper',
'_tee', '_tee_dataobject', 'accumulate', 'chain', 'combinations',
'combinations_with_replacement', 'compress', 'count', 'cycle', 'dropwhile',
'filterfalse', 'groupby', 'islice', 'pairwise', 'permutations', 'product',
'repeat', 'starmap', 'takewhile', 'tee', 'zip_longest']
```

Search Order

- "вградените модули"
- текущата директория
- *sys.path / PYTHONPATH*

```
>>> import sys
>>> sys.path
['', '/usr/lib/python310.zip', '/usr/lib/python3.10',
'/usr/lib/python3.10/lib-dynload',
'/home/vbechev/.local/lib/python3.10/site-packages',
'/usr/local/lib/python3.10/dist-packages', '/usr/lib/python3/dist-packages']
```

Пакети

- Модули, изградени само от файлове, е твърде плоско решение
- Можем да групираме няколко файла (модули) в един свръх-модул (у-у-у-у)
- За да направим една директория модул, трябва да създадем *__init__.py* файл вътре
- Това е инициализаторът на свръх-модула (пакета) и се изпълнява преди всичко останало

```
fingers/
    __init__.py
    thumb.py
    index.py
    middle.py
    ...
```

__all__

- Можем в *__init__.py* да дефинираме списък от стрингове `__all__`
- Само имената в него ще бъдат импортнати, ако се използва `*`
    - `също то можем да направим и в модул от само един файл`

Absolute vs relative

- Ако пишем в пакет, можем да пропуснем търсенето в *sys.path* и директно да импортнем нещо от текущата директория:

```
from . import index
```

- Можем да търсим и в "горния" възел:

```
from .. import toes
```

- Или:

```
from ..toes import big_toe
```

- Правете го пестеливо

.pyc файлове

- *.pyc* са прекомпилирани версии на *.py* файловете ни
- Генерират се при импортиране на файла
- Питонска оптимизация
- В Python 3 стоят в директорията *__pycache__*
- Можем да ги деактивираме генерално
- Не мислим за тях
- (освен при version control)

Модули, които си струва да споменем

- os
- logging
- re
- sys
- json
- random
- itertools
- datetime
- unittest

За всички фенове на Java

```
>>> from __future__ import braces
SyntaxError: not a chance
```

Питонския зен

```python
import this
```

10. Част 2 - среди

Въпроси дотук?

Средата е важна

"С какъвто се събереш, такъв си си бил!"

~ А. Наков

Програмата ви има нужда от правилната среда за да работи:

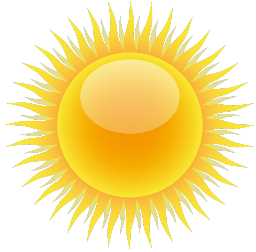
- Операционна система
- Файлова система
- Интерпретатор
- Библиотеки
- Връзка с интернет
- ...

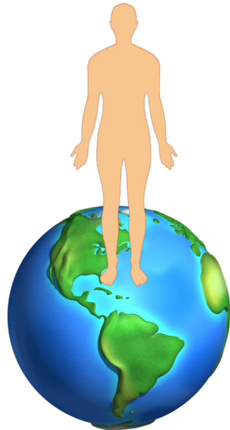
По-лесно е да контролираме стабилна средата, отколкото да преправим програмата така, че да работи във всяка среда.

Как да подготвя средата?

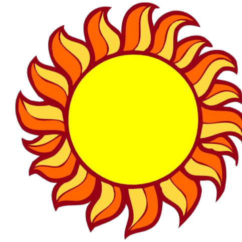
- Желязо
- Виртуална машина
- Докер
- Виртуална среда

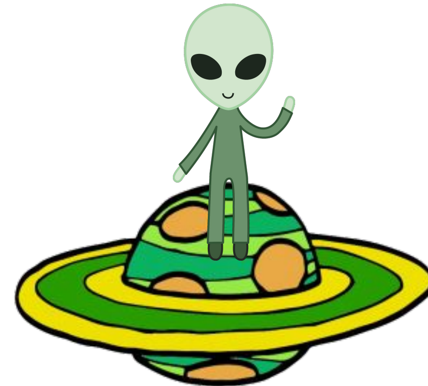
Да поиграем на Господ





- Различните програми имат нужда от различни среди
- Различните форми на живот също





Ако искаме да предоставим нова среда, можем...

- просто да създадем нова Вселена

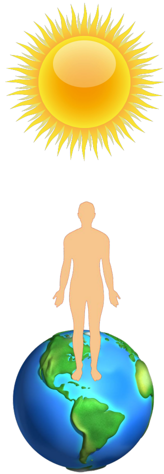
Което е съпоставимо с това да...

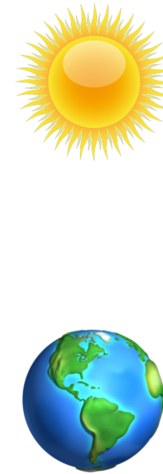
- осигурим ново желязо

- но това е доста скъпо решение

Ако искаме да предоставим нова среда, можем...

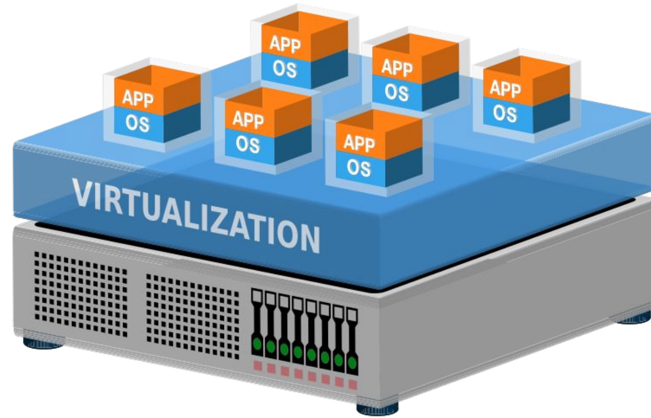
- просто да създадем ново Слънце и нова планета





Което е съпоставимо с това да...

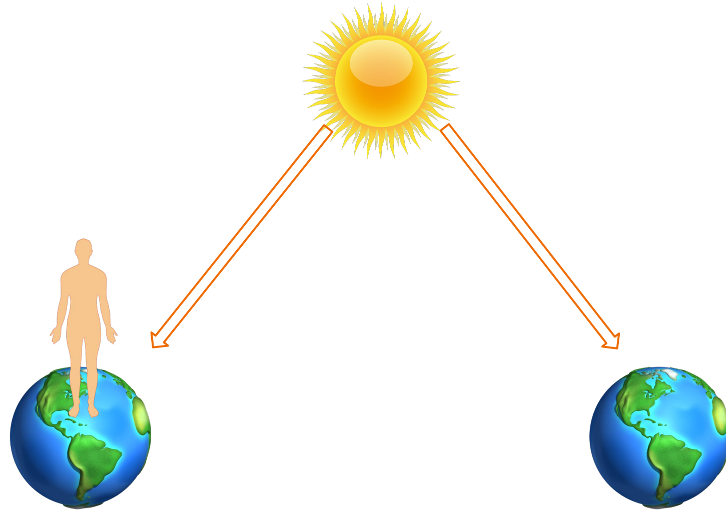
- създадем виртуална машина във вече съществуващо желязо



- няма нужда от нов хардуер, но всяка виртуална машина притежава своя собствена операционна система, което изисква ресурс

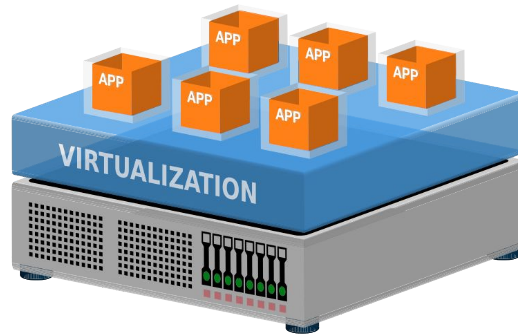
Ако искаме да предоставим нова среда, можем...

- просто да създадем нова планета, използвайки същото слънце



Което е съпоставимо с това да...

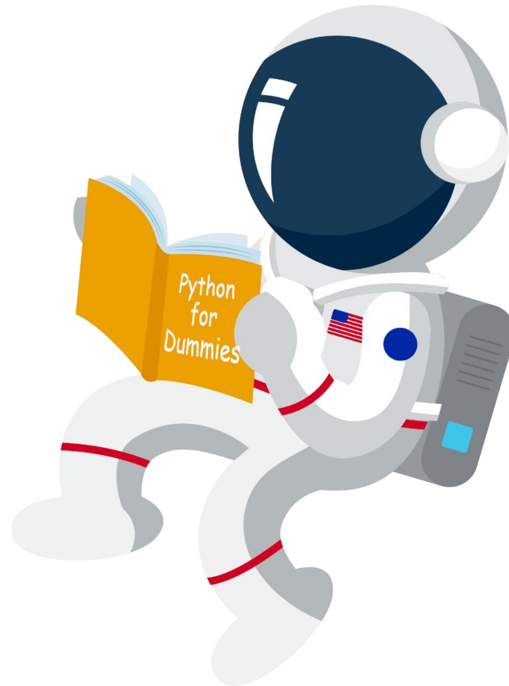
- използваме докер контейнер



- няма нужда от нов хардуер и няма нужда от отделна операционна система за всеки контейнер, но докерът притежава собствена файлова система, процеси, потребители, мрежа...

Ако искаме да предоставим нова среда, можем...

- просто да връчим на човека един скафандър и да го пращаме навсякъде



Което е съпоставимо с това да...

- използваме виртуална среда

- единственото новосъздадено нещо е Python, с прилежащите си библиотеки, и конкретни настройки по променливите на средата

Променливи на средата (environment variables)

- Предоставят се от операционната система
- Използват се, за да определят различни свойства на средата
- Потребителят може да ги промени, за да контролира средата
- Всеки процес (програма) получава копие от средата и може да я модифицира за лично ползване
- Това до известна степен ни позволява да използваме различни среди за различни програми

Как изглеждат тези променливи?

Windows - $ set

```
PATH=C:\ProgramFiles;C:\Users\Stamat\AppData\Local\Programs\Python\Python310\;
...
TEMP=C:\Users\Stamat\AppData\Local\Temp
HOMEDRIVE=C:
HOMEPATH=\Users\Stamat
…
```

Как изглеждат тези променливи?

Linux/MacOS - `$ env`

```
PATH=/home/gvkunchev/.local/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:
/usr/games
PWD=/home/gvkunchev/cool_stuff
HOME=/home/gvkunchev
SHELL=/bin/bash
…
```

PATH

- *PATH* е променливата, която се използва при извикване на команда
- Когато отворите терминал и напишете `python`, *shell*-ът търси в *PATH*, за да намери какво искате да пуснете
- Това значи ли, че мога да имам инсталирани две различни версии на Python и да контролирам коя се извиква чрез модификация на *PATH*?
- Анджак!

Да го направим под Windows

```
C:\Users\Stamat>python
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep  5 2022, 14:08:36) [MSC v.1933 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> quit()

C:\Users\Stamat>echo "ECHO I am fake Python" > Desktop\python.bat

C:\Users\Stamat>set PATH=C:\Users\Stamat\Desktop;%PATH%

C:\Users\Stamat>python
I am fake Python
```

Да го направим под Linux

```
gvkunchev@instance-1:~$ python
Python 2.7.18 (default, Jul 14 2021, 08:11:37)
[GCC 10.2.1 20210110] on linux2
Type "help", "copyright", "credits" or "license" for more information.

>>> quit()

gvkunchev@instance-1:~$ mkdir dummy_python

gvkunchev@instance-1:~$ echo "echo 'I am fake python'" > dummy_python/python

gvkunchev@instance-1:~$ chmod +x dummy_python/python

gvkunchev@instance-1:~$ PATH="dummy_python:%PATH%"

gvkunchev@instance-1:~$ python
I am fake python
```

Да не преоткриваме колелото

```
$ python -m pip install virtualenv
```

- *pip* е модул и като всеки друг модул, той може да се изпълни чрез “`python -m <module>`”
- *pip* е мениджъра за пакети на Python и идва заедно с него по подразбиране
- *pip* използва онлайн хранилища, за да намери и инсталира пакета, който ви трябва
- също така се грижи да инсталира и евентуални зависимости на този пакет
- *virtualenv* е пакет, който ви позволява да създавате виртуални среди през *CLI*

```
Collecting virtualenv
  Downloading virtualenv-20.16.7-py3-none-any.whl (8.8 MB)
     -------------------------- 8.8/8.8 MB 9.8 MB/s eta 0:00:00
Collecting distlib<1,>=0.3.6
  Downloading distlib-0.3.6-py2.py3-none-any.whl (468 kB)
     ---------------------- 468.5/468.5 kB 9.8 MB/s eta 0:00:00
Installing collected packages: distlib, virtualenv
Successfully installed distlib-0.3.6 virtualenv-20.16.7
```

Създаване и активиране на виртуална среда

```
$ virtualenv django_venv
```

- създава директория с всичко, което е нужно за изолиране на *Python* от глобално инсталираната версия

```
$ django_venv\Scripts\activate (Windows)
$ source django_venv/bin/activate (Linux)
```

- активира средата, променяйки променливите на средата така, че да използвате Python версията от създадената директория

```
$ where python (Windows) # …\django_venv\Scripts\python.exe
$ which python (Linux) # …/django_venv/bin/python
```

- Ако искате да излезете от средата:
  ```
  $ deactivate
  ```

Използване на виртуална среда

- Бидейки във виртуална среда, можем да инсталираме различни пакети за различни проекти
  - Например:
    `$ python -m pip install django=2.0.0`
- По този начин изолираме зависимостите на проекта от глобалната среда
- Можем да дефинираме всички пакети, от които се нуждаем, в един *requirements.txt* файл, след което просто създаваме виртуална среда, и в нея пускаме:
  `$ python -m pip install -r requirements.txt`
  - `"$ python -m pip freeze" ще изплюе този файл наготово, спрямо текущото състояние на средата`
- Ако нещо се обърка, просто трием една директория

virtualenv рапър

- Видяхме, че има известни разлики в използването на *virtualenv* под *Windows* и *Linux*
- Също така, активирането на средата е просто source-ване на някакъв скрипт
- На някой това не му е харесало, така че:
  - ```
    $ python -m pip install virtualenvwrapper
    # за Windows е "virtualenvwrapper-win"
    ```
  - ```
    $ mkvirtualenv django_venv
    ```
  - ```
    $ workon django_venv
    ```
- *workon* автоматично намира среди и допълва при таб-ване
- също така може автоматично да изпълнява скриптове преди или след активиране
- и други благинки
    ```

venv

- От известно време *Python* идва с модул за създаване на виртуални среди, наречен “*venv*”
- На повечето места той е готов за използване и се използва точно така, както показахме с *virtuelenv*
- На някои места, обаче, например *Ubuntu*, той самият изисква допълнителни пакети
- Хубавото е, че можете да правите виртуални среди, използвайки Python код

```
import venv

venv.create("/tmp/django_venv")
```

Wheel - стандартът за разпространение на пакети

- Както казахме, “`pip install`“ търси пакети в онлайн хранилище
- Хранилището е https://pypi.org/
- То съдържа *wheel* файлове (**.whl*), които съдържат целия сорс код на даден пакет, плюс мета информация за версии, автори, зависимости и т.н.
- Ако искате да създадете пакет, който да разпространите, това е мястото

Защо Wheel?

- Преди *Wheel* имаше *Egg*, който имаше нужда от подобрения
  - *Egg*, защото… *Monty Python* (*spam and eggs*)
  - а и питоните се раждат от яйца
- *Wheel* е подобрената версия на *Egg* - *Python Packaging* ***reinvented***
- Освен това *wheel* е името на **кутиите за сирене**
  - Отново *Monty Python* и техният скеч за сиренето
- А и това, което *wheel* прави, е "***roll*** *out software*"

Да направим и ние един wheel файл

```
# cool_stuff.py
from coolprint import coolprint

def cool_function():
    coolprint("I am cool.")
```

- cool_project
  - __init__.py
  - cool_stuff.py
- setup.py

```
# setup.py
from setuptools import setup, find_packages

setup(
    name='cool_project',
    version='1.0',
    packages=find_packages(),
    python_requires='>=3.7',
    install_requires=['coolprint']
)
```

Подготовката е направена - да генерираме

- `$ python setup.py bdist_wheel`
- Създава разни директории, но това, което нас ни интересува, е: *dist/cool_project-1.0-py3-none-any.whl*
- {distribution}-{version}(-{build tag})?-{python tag}-{abi tag}-{platform tag}.whl
  - * ABI - Application Binary Interface
- Това е просто *.zip* файл, така че ако искате, можете да го отворите с подходяща програма и да го разгледате

Да направим среда и да инсталираме пакета

- `$ virtualenv coolenv`
- `$ cd coolenv`
- <слагаме .whl файла в директорията>
- `$ python -m pip install cool_project-1.0-py3-none-any.whl`
- …`Successfully installed cool-project-1.0 coolprint-0.0.2`
- Пакетите са инсталирани в *site_packages* на виртуалната среда

```
(coolenv) C:\Users\stamat\Desktop\coolenv>python
>>> from cool_project.cool_stuff import cool_function
>>> cool_function()
```

```
I
```

Въпроси?