Metodika analýzy dat: Od základů po aplikace metod strojového učení

Úvod do programování a jazyk Python 2. část

Jakub Steinbach, Jan Vrba

Ústav matematiky, informatiky a kybernetiky
VŠCHT

1.10.2024

Obsah slajdů I

Moduly a balíčky

Moduly a balíčky

- **modul** je soubor obsahující definice a příkazy v Pythonu
- jeho obsah lze používat v jiných modulech po jeho importu
- import se provádí pomocí klíčového slova **import** a jména modulu (případně v kombinaci s klíčovým slovem **as** a uvedením aliasu)
- jméno modulu je dáno názvem souboru (bez přípony) a je dostupné v rámci modulu jako proměnná *__name__*

V souboru my_module.py máme kód

```
def dummy_function():
    print("I am so dummy")
```

V souboru out_of_module.py importujeme my_module.py a zavoláme funkci dummy_fuction()

```
import my_module as mm
mm.dummy_function()
print(mm.__name__)
```

Příkaz import

- příkaz **import** realizuje dva kroky:
  1. nalezení modulu (prohledává všechny adresáře specifikované v **sys.path**), jeho nahrání a inicializaci
  2. zpřístupnění obsah modulu v rámci kódu ve kterém je **import** zavolán (funkce a proměnné jsou dostupné pod jmény definovanými v modulu)
- pokud není adresář dostupný, lze jej přidat pomocí metody **sys.path.append(path)**

soubor module_test.py:

```
def dummy_function():
    print("dummy function")
a = 7
```

soubor import_test.py ve stejném adresáři:

```
import module_test
module_test.dummy_function()
print(module_test.a)
print(module_test.__name__)
```

Modul - import vybraných funkcí

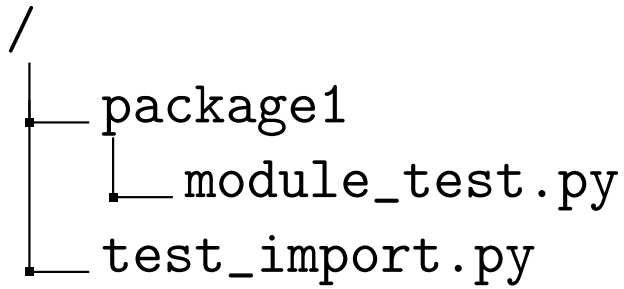
- z modulu lze importovat pouze vybrané funkce (dostupné pod aliasem)
- z modulu lze importovat všechny funkce, které jsou pak dostupné pod jejich názvem

```
from my_module import dummy_function
dummy_function()

from my_module import dummy_function as df
df()

from my_module import dummy_function as df1, \
dummy_function2 as df2
df1()

from my_module import *
dummy_function()
```

Import z adresáře

- moduly lze strukturovat do adresářů a podadresářů
- jednotlivé moduly jsou pak pro import dostupné pomocí tečkové notace (**adresář.modul**)

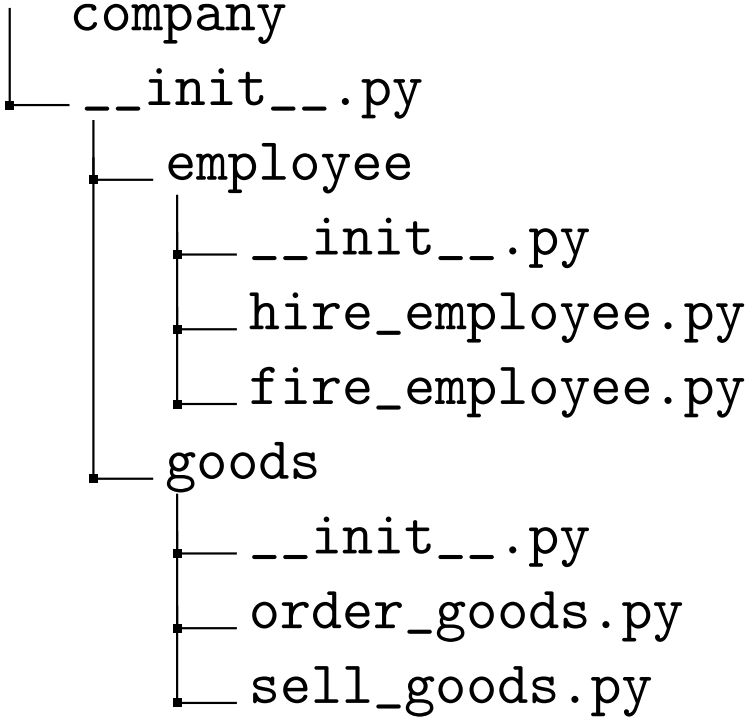
Mějme adresářovou strukturu




```
import package1.module_test

package1.module_test.dummy_function()
print(package1.module_test.a)
print(package1.module_test.__name__)
```

Balíčky

- balíčky umožňují vytvářet hierachii modulů a podmodulů
- pokud je v adresáři přítomný soubor __init__.py je adresář považován za balíček a při importu se vykoná obsah souboru __init__.py
- soubor __init__.py nemusí obsahovat žádný kód



Import z balíčků

- import jednotlivých modulů

```
from company.employee import hire_employee
hire_employee.new_employee()

import company.employee.hire_employee
company.employee.hire_employee.new_employee()
```

- import konkrétních funkcí

```
from company.employee.hire_employee import \
    new_employee
new_employee()
```

- import všech funkcí modulu

```
from company.employee.hire_employee import *
new_employee()
```

Práce se soubory

Práce se soubory

- soubory jsou zásadní v okamžiku kdy potřebujeme pracovat daty, která nejsou potřebná pouze při aktuálním běhu programu -> potřeba data uložit, nebo načíst
- k souborům lze přistupovat pomocí funkce **open(filename, mode)** v režimech:
  1. read - pro čtení - defaultní přístup, vrátí chybu pokud soubor neexistuje ("r")
  2. write - pro zápis, vytvoří soubor pokud neexistuje ("w")
  3. append - přidání dalších dat do souboru, vytvoří soubor pokud neexistuje ("a")
  4. create - vytvoří soubor, v případě že existuje, vrátí chybu ("x")
- dále je možné specifikovat jestli jde o soubor textový, nebo binánrí
  1. text - defaultní přístup, informace v souboru jsou jako text ("t")
  2. binary - informace v souboru jsou zpracovávány po bajtech ("b") - např. pickle soubory, obrázky, všechny soubory co neobsahují text
- funkce **open()** vrací objekt který reprezentuje otevřený soubor

Otevření souboru - open()

- otevření souboru v textovém módu pro čtení

```
f = open("file.txt")
```

- otevření souboru v binárním módu pro zápis

```
f = open("file.txt", "wb")
```

- otevření souboru v textovém módu pro přidání dat

```
f = open("file.txt", "a")
```

- vytvoření nového souboru

```
f = open("file.txt", "x")
```

- otevření binárního souboru pro čtení

```
f = open("file.txt", "rb")
```

Čtení dat z textového souboru - open()

- čtení obsahu celého textového souboru

```
f = open("file.txt")
content = f.read()
```

- načtení požadovaného počtu znaků

```
f = open("file.txt")
content = f.read(10)
```

- čtení prvních dvou řádků

```
f = open("file.txt")
print(f.readline())
print(f.readline())
```

- procházení obsahu souboru po řádcích

```
f = open("file.txt")
for line in f:
    print(line)
```

Zápis do souboru

- pro zápis do souboru je potřeba soubor otevřít v módu pro zápis nebo přídání (write/append)
- při otevření souboru pro zápis je původní obsah přepsán novým

```
f = open("file.txt", "w")
f.write("content1\n")
f.close()
f = open("file.txt", "a")
f.write("content2\n")
f.close()
```

Uzavření souboru - close()

- při otevření souboru je v OS vytvořen tzv. deskriptor souboru a ostatní procesy k němu tak nemůžou přistoupit (ověřit!!!)
- po ukončení práce se souborem je potřeba soubor uzavřít a tím umožnit dalším procesům k němu přistupovat
- pro uzavření souboru slouží funkce **close()**

```
f = open("file.txt", "x")
data = f.read()
f.close()
```

Práce se souborem - jde to i lépe?

```
with open("file.txt", "x") as f:
    data = f.read()
```

- použití klíčového slova **with** umožní vykonání bloku kódu pomocí metod definovaných příslušným *context managerem*
- v tomto případě se automatické zavolání funkce **close()** po dokončení načtení souboru
- zavolání funkce **close()** proběhne v i případě, že dojde při otevírání souboru nebo načtení dat k výjimce

Práce se souborem - funkce readlines()

- funkce **readlines(sizehint)** načte všechny řádky textového souboru jako seznam
- pokud specifikujeme parametr *sizehint* je načten pouze omezený počet bytů zaokrouhlený k nejbližší hodnotě velikosti vnitřního bufferu

```
with open("foo.txt", "r") as f:
    lines = f.readlines()
    print(lines)
    f.seek(0, 0)
    line = f.readlines(10)
    print(line)
```

Práce se souborem - funkce writelines()

- funkce **writelines(list)** zapíše do souboru položky seznamu

```
with open("foo.txt", "w") as f:
    lines = ["a\n", "b", "c"]
    f.writelines(lines)

with open("foo.txt", "a") as f:
    lines = ["a\n", "b", "c"]
    f.writelines(lines)
```

Cesta k souborům - pathlib

- pro správu souborů a práci s cestou lze použít modul os nebo modul pathlib (doporučeno)
- modul pathlib umožňuje efektivně:
  - zjistit obsah adresářů a podadresářů
  - vybírat soubory podle mask (pattern matchitg)
  - vytvářet a mazat adresáře
  - mazat soubory a adresáře
  - kopírovat, přesouvat a přejmenovávat soubory a adresáře
  - pracovat s cestami bez ohledu na platformu

Cesta k adresáři, souboru

- modul pathlib obsahuje třídu **Path** která umožňuje pracovat s Windows i Posix cestama k adresářů/souborům
- pomocí **Path** je možné používat relativní i absolutní cesty

```
from pathlib import Path
path = Path("C:/", "Users") # path to C:\Users
path = Path("C:/Users") # path to C:\Users
path = Path(".") # path current directory
path = Path("/usr") # Posix path to /usr folder
path = Path(".").resolve().parent
```

Zjištění obsahu adresáře

- pro zjištění obsahu adresáře lze použít funkce **iterdir()** nebo **glob()** třídy **Path**
- funkce **iterdir()** vrací generátor obsahující názvy podadresářů/souborů v adresáři
- funkce **glob(mask)** vrací generátor obsahjující názvy podadresářů/souborů v adresáři které vyhovují požadované masce (*mask*)

```
for item in Path(path_to_folder).iterdir():
    print(item)
```

```
for item in Path(path_to_folder).glob('*'):
    print(item)
```

Výběr souborů pomocí masky

- funkce **glob(mask)** třídy **Path** umožňuje vybírat soubory a adresáře, které odpovídají zadané masce

```
for item in Path(path_to_folder).glob('*.py'):
    print(item)
for item in Path(path_to_folder).glob('**/*'):
    print(item)
for item in Path(path_to_folder).glob('????.txt'):
    print(item)
```

Pozor, ** může být v případě velkého počtu podadresářů časově náročné!

Vytvoření a přejmenování adresáře

- pro vytvoření adresáře lze použít funkci **mkdir(parents=True, exist_ok=True)** třídy **Path**
- parametr *parents=True* zajistí vytvoření rodičovský adresář, v případě že neexistuje
- parametr *exist_ok=True* ignoruje *FileExistsError* v případě, že adresář již existuje
- pro přejmenování adresáře/souboru slouží funkce **rename(new_name)** třídy **Path**

```
filepath=Path("./temp", "temp2")
filepath.mkdir(parents=True, exist_ok=True)
Path("./temp").rename("renamed_temp")
```

Kopírování a přesouvání

- pro kopírování souborů je nutné použít funkci **copy2(src,dst)** modulu **shutil**, která zkopíruje soubor specifikovaný cestou *src* do umístění specifikovaným cestou *dst* včetně metadat
- přesunutí souboru lze realizovat pomocí funkce **rename(new_path_name)** třídy **Path**

```
import shutil
src = Path(".", "renamed_temp", "test.txt")
dst = Path(".", "renamed_temp", "temp2", "test.txt")
shutil.copy2(src, dst)
src.rename(dst)
```

Správa souborů - mazání souboru

- funkce **unlink(missing_ok=False)** vymaže soubor nebo symbolický odkaz
- argument *missing_ok=True* způsobí ignoraci vyjímky FileNotFoundError (od verze 3.8)
- defaultní hodnota argumentu *missing_ok* je False

```
import pathlib
file_to_remove = pathlib.Path("foobar.txt")
file_to_remove.unlink()
```

Správa souborů - mazání adresáře

- funkce **rmdir()** vymaže prázdný adresář
- pokud nás nezajímá jestli je adresář prázdný, můžeme použít funkci funkci **shutil.rmtree(folder)**, kde argument *folder* obsahuje cestu k cílovému adresáři

```
import pathlib
folder_to_remove = pathlib.Path("foo_folder")
folder_to_remove.rmdir()

import shutil
full_folder = pathlib.Path("full_folder")
shutil.rmtree(fullfolder)
```

Serializace - pickle

- převedení libovolné datové struktury (proměnné, objektu..) uloženého v paměti počítače do uložitelné posloupnosti bitů (resp. odeslatelné do sítě) tak, aby mohla být původní datová struktura rekonstruována do původní podoby (deserializace)
- modul pickle umožňuje serializaci v podobě sekvence bytů
- výsledný soubor není člověkem čitelný
- modul pickle není bezpečný (vždy deserializujem jen objekty, kterým věříme)

```
import pickle
with open("data.dat", "wb") as f:
    pickle.dump([1,2,3], f)

with open("data.dat", "rb") as f:
    data = pickle.load(f)
print(data)
```

Serializace - JSON

- formát JSON (JavaScript Object Notation) je jednotný formát pro výměnu dat nezávisle na platformě
- je čitelný člověkem
- při deserializaci nemůže dojít k exekuci

```
import json
with open("data.dat", "w") as f:
    json.dump([1,2,3], f)

with open("data.dat", "r") as f:
    data = json.load(f)
print(data)
```

Vyjímky

Ošetření výjimky

- korektní přístup je ošetřit všechny očekávané vyyjimky
- a ošetřit neočekávané výjimky

```
x = 3
# y = "a"
try:
    print(x / y)
except TypeError as ex:
    print("Could not divide: ", ex)
except ZeroDivisionError as ex:
    print("Zero encountered: " , ex)
except Exception as ex:
    print("Unexpected exception: ", ex)
```

Vyvolání výjimky

- výjimku lze vyvolat i programově pomocí příkazu **raise Exception**
- možné použití při ladění požadované obsluhy výjimky
- některé výjimky je nutné obsloužit (např. zalogovat) a potom je znovu vyvolat
- **raise** bez specifikované výjimky vyvolá poslední vzniklou výjimku

```
try:
    print(x / y)
except ZeroDivisionError as ex:
    print("Zero encountered: " , ex)
except Exception as ex:
    print("Unexpected exception: ", ex)
    raise
```

Blok finally

- klíčové slovo **finally** definuje blok kódu, který se vykoná vždy, bez ohledu jestli vznikla výjimka nebo ne
- je označován jako "clean-up actions"blok kódu
- typické použití: definujeme blok kódu, který se musí vykonat za všech okolností (zavření portů, souboru, smazání objektu, atp.)
- (a místo print() logovat)

```
try:
    if type(to_cypher) is str:
        encrypted_stuff = cypher(to_cypher)
except TypeError:
    print("nothing to cypher")
finally:
    del to_cypher
```

Blok else

- klíčové slovo **else** definuje blok kódu který se vykoná v případě, že nenastala žádná výjimka

```
try:
    y = f(x)
except ValueError:
    logging.error("ValueError in f(x)")
else:
    logging.info("successful evaluation of f(x)")
```