Programowanie w Pythonie

Łukasz Mioduszewski, UKSW 2022

Pliki i wyjątki

Optymalizacja

- Aby szybciej ładować moduły, Python tworzy katalog __pycache__ w którym są one pre-kompilowane do plików binarnych wykonywach przez maszynę wirtualną (CPython pozwala zmienić moduł w taki plik z rozszerzeniem .pyc)
- Wszystko to dzieje się automatycznie, nie trzeba się martwić o to czy uruchamiamy .py czy .pyc, Python sam to optymalizuje…
- … ale jeśli mamy do wyboru użyć polecenia z jakiegoś modułu albo napisać własną implementację z wieloma pętlami for, gotowe polecenie będzie szybsze

Praca z plikami

- Funkcja open(filename, mode='r', encoding=None) zwraca obiekt plikowy
- Możliwe wartości mode:
  - 'r' tylko odczyt
  - 'w' nadpisanie pliku (wszystko co było tam wcześniej zniknie)
  - 'a' append, czyli dodanie nowej zawartości na koniec pliku (stara zostanie)
  - 'r+' odczyt i zapis
  - dodanie litery b na końcu mode otwiera plik w trybie binarnym (nie wpisuj wtedy encoding)

Praca z plikami w trybie tekstowym

- Domyślne encoding nie zawsze jest None, zależy od systemu operacyjnego
- Przy wczytywaniu nowe linie \r\n są zamieniane na \n i odwrotnie – odczyt i zapis pliku binarnego w trybie tekstowym może go zniszczyć!
- Dla UTF-8 wystarczy encoding="utf-8"
- Po skończeniu pracy z obiektem plikowym f należy go zamknąć:
  f.close()
- Jeśli nie uda się zamknąć to mamy problem, dlatego najlepiej użyć…

Konstrukcja with

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

- Wszystko wewnątrz konstrukcji with może źle zadziałać, a obiekt plikowy i tak zamknie plik – dlatego bezpiecznie jej używać (i łatwiej niż konstrukcji try)
- Przy czytaniu plików to nieważne, ale przy zapisie do plików już tak

Obiekt plikowy

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

- Polecenie open(nazwapliku) tworzy nam obiekt plikowy, np. f. Jego metody to:
  - f.read(size) wczytuje size znaków lub bajtów, domyślnie cały plik
    - do stringa w trybie tekstowym, do obiektu bytes w trybie binarnym
    - po odczytaniu size znaków "kursor" zatrzymuje się i kolejne polecenie f.read(size2) wczytuje kolejne size2 znaków. Jeśli to już koniec pliku, f.read() zwróci pusty string
  - f.readline() wczytuje pojedynczą linię (włącznie z \n na końcu, chyba że to koniec)

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Obiekt plikowy

- f.write(string) zapisuje string do pliku i zwraca ile znaków zapisał
- f.tell() zwraca aktualną pozycję kursora licząc od początku pliku
- f.seek(offset, whence=0) przestawia kursor o zadany offset licząc od początku (whence=0), aktualnej pozycji kursora (whence=1) lub od końca (whence=2) pliku
  - Dla binarnych jednostka to 1 bajt
  - Dla tekstowych offset musi być wynikiem metody f.tell() albo zerem,
    whence może być albo 0 albo 2, ale wtedy jedyny dopuszczalny offset to 0

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

Obiekt plikowy

- Obiekt plikowy jest iterowalny w trybie tekstowym (po liniach), jeśli chcemy wszystkie linie w formie listy można użyć f.readlines()

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

Konstrukcja with

- with funkcja() as wynikFunkcji: wykonuje blok instrukcji używający obiektu przechowywanego w zmiennej wynikFunkcji, który nie będzie dostępny po wyjściu z tego bloku. Niektóre klasy (jak klasa obiektu plikowego) mają zdefiniowane destruktory, które automatycznie zamykają co trzeba, więc używając konstrukcji with nie trzeba pamiętać o zamknięciu pliku
- Czy del f też wywołuje destruktor?

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

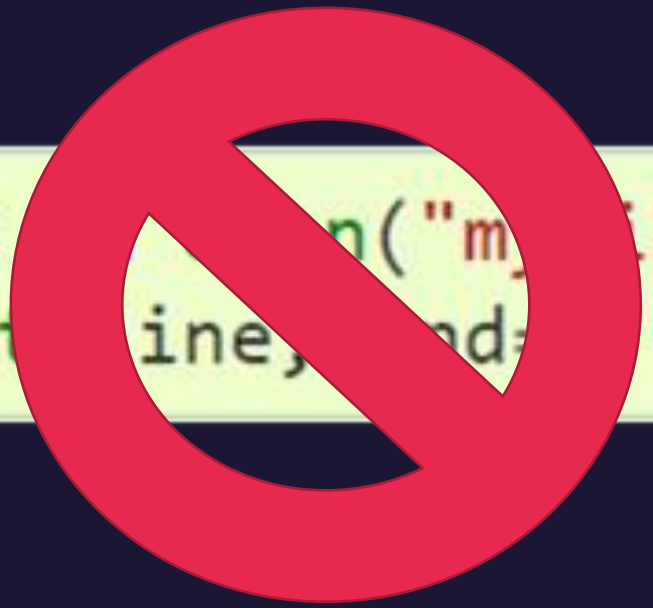
```
for line in open("myfile.txt"):
    print(line, end="")
```

Konstrukcja with

- with funkcja() as wynikFunkcji: wykonuje blok instrukcji używający obiektu przechowywanego w zmiennej wynikFunkcji, który nie będzie dostępny po wyjściu z tego bloku. Niektóre klasy (jak klasa obiektu plikowego) mają zdefiniowane destruktory, które automatycznie zamykają co trzeba, więc używając konstrukcji with nie trzeba pamiętać o zamknięciu pliku
- Czy del f też wywołuje destruktor?

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

```
for line
    prin
```



Serializacja w jsonie

- moduł json umożliwia konwersję list i słowników na string w formacie json:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

- Aby zapisać obiekt x w formie stringa json w obiekcie plikowym f, używamy: json.dump(x,f)
- W drugą stronę (deserializacja): x = json.load(f)
- Uwaga: json używa tylko kodowania utf-8

Serializacja w piklach

- moduł pickle umożliwia serializację wszystkiego (w tym obiektów które automatycznie się wykonują i mogą uszkodzić twój komputer)
- Zapisuje i odczytuje pliki w trybie binarnym, są małe ale nieczytelne
- pickle.dump(obj,file) zapisuje obiekt obj do obiektu plikowego file
- pickle.dumps(obj) zwraca obiekt bytes zawierający piklowany obiekt
- pickle.load(file) zwraca zdeserializowany obiekt z pliku piklowego file
- pickle.loads(data) robi to samo, tyle że z obiektu bytes o nazwie data
- Są różne wersje protokołu pickle, w argumencie protocol wybierz właściwy

Błędy i wyjątki

- Jeśli nasz kod jest zły mamy SyntaxError i błąd, ale czasami błędy są do uniknięcia – np. działania x/y nie można wykonać jeśli y wynosi 0, a to ile wynosi y może zależeć od użytkownika itp.
- Wyjątki zwykle zatrzymują działanie programu, ale możemy je łapać konstrukcją try: blok instrukcji except (rodzaje wyjątków): blok instrukcji
  - blok instrukcji po try jest wykonywany (aż się skończy albo wystąpi wyjątek)
  - Gdy wystąpi wyjątek tego typu co w except, wykonywany jest blok instrukcji po except
  - Gdy wystąpi inny wyjątek, jest przekazywany do try wyższego poziomu, jeżeli jest

Konstrukcja try - przykłady

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

- Może być wiele różnych except, każde może obsługiwać wiele wyjątków:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Własne wyjątki

- Wyjątki zgłaszamy poleceniem raise
- raise bez żadnego wyjątku zgłasza ten sam co poprzednio

Wypisuje B, B, B

```
1  class B(Exception):
2      pass
3
4  class C(B):
5      pass
6
7  class D(C):
8      pass
9
10 for cls in [B, C, D]:
11     try:
12         raise cls()
13     except B:
14         print("B")
15     except C:
16         print("C")
17     except D:
18         print("D")
```

Wypisuje B, C, D

```
1  class B(Exception):
2      pass
3
4  class C(B):
5      pass
6
7  class D(C):
8      pass
9
10 for cls in [B, C, D]:
11     try:
12         raise cls()
13     except D:
14         print("D")
15     except C:
16         print("C")
17     except B:
18         print("B")
```

Przekazywanie argumentów wyjątkom

- Konstrukcja except wyjątek as zmienna: pozwala używać wyjątku. Tworząc wyjątek możemy przekazać konstruktorowi argumenty, np. raise NameError('HiThere')

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Klasy wyjątków

- Większość wyjątków pochodzi od Exception, które pochodzi od BaseException
- Exception posiada metodę add_note(string), pozwala dodawać notatki
- W przykładzie po prawej OSError i Exception to klasy, a err to konkretny wyjątek (obiekt z danej klasy)

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

Try except else, gdy wykona się dobrze

- Blok instrukcji po else wykonuje się wtedy, kiedy nie było wyjątku

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Wyjątki przy obsłudze wyjątków

- konstrukcja raise konstruktorWyjątku from wyjątek pozwala zgłosić wyjątek na podstawie istniejącego wyjątku
- Wyjątki zgłaszane w bloku except spowodują wypisanie także tego oryginalnego

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: unable to handle error
```

Wyjątki przy obsłudze wyjątków

- Wyjątki zgłaszane w bloku except spowodują wypisanie także tego oryginalnego, chyba że użyjemy raise from None:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

Finally zawsze się wykona

- Blok instrukcji po "finally:" wykonuje się po tych z try i po tych z except/else
- Wszystkie inne wyjątki zostaną potem ponownie zgłoszone/obsłużone, chyba że finally zawiera break, continue lub return
- Jeśli zarówno try i finally zawierają return, zwrócone zostanie to z finally

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Finally zawsze się wykona

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Zgłaszanie wielu wyjątków naraz

- Argumenty konstruktora obiektu ExceptionGroup to opis błędu i lista wyjątków
- W konstrukcji try łapiemy jeden wyjątek z listy przy użyciu except*

```
>>> def f():
...     raise ExceptionGroup("group1",
...                          [OSError(1),
...                           SystemError(2),
...                           ExceptionGroup("group2",
...                                          [OSError(3), RecursionError(4)])])
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
```

Zgłaszanie wielu wyjątków naraz

- Przykład użycia:

```
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...     raise ExceptionGroup("Test Failures", excs)
...
```

Funkcje to też obiekty

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Klasy

- Klasy też są obiektami
- Można dziedziczyć z klas wbudowanych
- Mają swoją własną przestrzeń nazw (tak jak funkcje)
- """Po nazwie klasy warto dodać string dokumentujący w potrójnym cudzysłowie"""

Zasięg zmiennych

- After local assignment: test spam
- After nonlocal assignment: nonlocal spam
- After global assignment: nonlocal spam
- In global scope: global spam

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Mutowalne zmienne w klasach

- Zagadka: jak to naprawić?

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Elementy klasy z ___

- Podwójny _ na początku zmiennej zmienia jej nazwę na zewnątrz klasy z __zmienna na _nazwaklasy__zmienna, przez co pozwala uniknąć kolizji nazw:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```