Faculty of Geodesy
and Cartography
WARSAW UNIVERSITY OF TECHNOLOGY

Informatyka Geodezyjna - wykłady/ćwiczenia, rok akad. 2021-2022

Wyk. 9: Python – operacje na plikach i formatowanie wyrażeń tekstowych

Kinga Wężka
kinga.wezka@pw.edu.pl
Katedra Geodezji i Astronomii Geodezyjnej

Warsaw University
of Technology

1. Czym jest plik - podstawowa struktura
   - Zakończenie linii danych - struktura pliku
2. Kolejność operacji na plikach
   - Otwarcie pliku
   - Tryby (ang. modes) operacji na plikach w Pythonie
   - Otwarcie i odczyt pliku
   - Otwarcie i zapis do pliku
   - Zamykanie pliku
3. Metody formatujące łańcuchy znaków
   - Formatowanie tekstu – znaki specjalne
   - Formatowanie tekstu – kodowanie znaków UTF-8
4. Inne metody odczytu i zapisu do plików
5. Obiekty w bazie danych - serializacja danych – pickle i JSON
6. Pomoc

Struktura plików komputerowych

- **HEADER** : metadane dotyczące zawartości pliku (nazwa, rozmiar, typ itd.).
- **ZAWARTOŚĆ**: zawartość pliku zapisana przez twórcę. To, co reprezentują dane, zależy od zastosowanej specyfikacji formatu, która zwykle jest definiowana przez rozszerzenie. Na przykład plik o rozszerzeniu `.gif` najprawdopodobniej jest zgodny ze specyfikacją formatu Graphics Interchange Format. Istnieją setki (tysiące), rozszerzeń plików, do najpopularniejszych plików formatu ASCII należą .txt lub .csv.
- **END OF FILE (EOF)**: znak specjalny wskazujący koniec pliku

Plik w systemie Windows:

```
1 Imiona: \r\n
2 Ala \r\n
3 Ola \r\n
```

W systemi Linux będzie wyglądał:

```
1 Imiona: \r
2 \n
3 Ala \r
4 \n
5 Ola \r
6 \n
```

- Jednym z problemów często spotykanych podczas pracy z danymi pliku jest **reprezentacja nowej linii** lub **zakończenia linii**. Ma to swoje korzenie w czasach *alfabetu Morsa*, kiedy do oznaczenia końca transmisji lub końca linii użyto specjalnego pro-znaku.
- Zostało to znormalizowane dla drukarek zarówno przez Międzynarodową Organizację Normalizacyjną (**ISO**), jak i American Standards Association (**ASA**). Standard ASA stanowi, że zakończenia linii powinny używać sekwencji znaków powrotu karetki (ang. *Carriage Return*) (`CR` lub `\r`) i znaków przesunięcia linii (ang. *Line Feed*) (`LF` lub `\n`) (`CR + LF` lub `\r \n`). Norma ISO dopuszczała jednak znaki `CR + LF` lub tylko znaki `LF`.
- Windows używa znaków `CR + LF` do wskazania nowej linii, podczas gdy Unix i nowsze wersje Mac używają tylko znaku `LF`. Może to powodować komplikacje podczas przetwarzania plików w systemie operacyjnym innym niż źródło pliku.

Odczyt z pliku

- otwarcie pliku: `open()`
- odczyt pliku: `read()`, `readline()`, `readlines()`
- zamknięcie pliku: `close()`, `flush()`

Zapis do pliku

- otwarcie pliku: `open()`
- odczyt pliku: `write()`, `writelines()`
- zamknięcie pliku: `close()`, `flush()`

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

- `"nazwa_pliku"` – definiuje nazwę zmiennej do której przypiszemy otwarty plik.
- `open(...)` – polecenie otwiera plik, który znajduje się w folderze, w którym jest uruchamiany plik (domyślnie plik otwierany jest tylko do odczytu!)
- `tryb` – tryb otwarcia np. "r" (read) – opcja domyślna – oznacza otwarcie pliku do odczytu, "w" (write) – oznacza otwarcie pliku do zapisu.
- `encoding` – opcja dodatkowo pozwala zdefiniować system kodowania otwieranego pliku (np. polskie znaki, symbole matematyczne itp.: utf-8)
- `errors` – obsługa błędów kodowania i dekodowania (nie dotyczy trybów binarnych).
- `newline` – kontroluje sposób działania trybu nowej linii (dotyczy trybu tekstowego). Może to być `None`, `'\n'`, `'\r'` i `'\r\n'`. Dla streamingu opcja `None`.
- `buffering` – służy do ustawiania zasad buforowania.
- `closefd` i `opener` – opcje dla deskryptorów pliku (identyfikator pliku wykorzystywany przez system operacyjny)

```
1 # zwykłe otwarcie pliku
2 nazwa = open("dane.txt")
3 # otwarcie z trybem do odczytu i kodowaniem znaków
4 nazwa = open("dane.txt", "r", encoding="utf-8"))
5 # otwarcie ze ścieżką bezwzględną DOS(Windows)
6 nazwa = open("C:\\PythonicWorld\\dane.txt", "r", encoding="utf-8")
```

Mode	Opis
'r'	Otwiera plik do odczytu. Tryb domyślny.
'w'	Otwiera plik do zapisu. Jeśli plik nie istnieje, zostanie utworzony nowy plik. Jeśli istnieje dane w nim zostają nadpisane.
'x'	Tworzy nowy plik. Jeśli plik istnieje, operacja zwraca błąd.
'a'	Otwiera plik w trybie dopisywania, jeśli plik nie istnieje, zostanie utworzony nowy.
'a+'	Dopisywanie danych na jego końcu, jeśli plik nie istnieje, zostaje utworzony.
't'	Tryb domyślny. Otwiera plik w trybie tekstowym.
'b'	Otwiera plik w trybie binarnym (np. 'rb' odczyt pliku binarnego).
'+'	Otwiera plik do aktualizacji (odczytu 'r+' i zapisu: 'w+').
rb,wb	Otwiera plik do odczytu lub zapisu formatów binarnych.

Metody odczytywania danych z pliku

- `read(size = -1)` – odczytuje plik na podstawie rozmiaru – liczby bajtów. Jeśli nie zostanie przekazany żaden argument lub argument inny niż `None` lub `size =-1`, zostanie odczytany cały plik.
- `readline(size =-1)` – odczytuje linię pliku na podstawie rozmiaru - liczby bajtów. Odczyt trwa do końca linii, a następnie nastepuje zawijanie. Jeśli nie zostanie przekazany żaden argumentub argument inny niż `None` lub `size =-1`, zostanie odczytana cała linia (lub reszta linii). Pusty ciąg znaków jest zwracany tylko w przypadku natychmiastowego napotkania EOF.
- `readlines()` - odczytuje linie z otwartego obiektu pliku i zwraca je jako listę linii.

In[1]:

```
plik  = open('plik1.txt', "r") # otwarcie z opcją do czytania
caly  = plik.read()            # caly plik jako zmienna tekstowa
caly  = caly.split('\n')       # podzieli na linię
plik.close()
```

In[2]:

```
plik  = open('plik1.txt', "r") # otwarcie z opcją do czytania
linie = plik.readline()        # odczyta linie pliku
plik.close()
```

In[3]:

```
plik  = open('plik1.txt', "r") # otwarcie z opcją do czytania
linie = plik.readlines()       # odczyta linie i zapisze do listy
for linia in linie:
    print(linia)               # wyświetli pojedyńczą linię pliku
plik.close()
```

Instrukcja with

- Instrukcja **with** automatycznie zamyka otwarty plik, więc **instrukcja with nie potrzebuje instrukcji close**

In[4]:

```
with open('nazwa_pliku.txt', "r") as plik:
    read_data = plik.read()
```

In[5]:

```
with open('nazwa_pliku.txt') as plik:
    for line in iter(plik.readline, ""):
        process_line(line)
```

In[6]:

```
plik = open('plik1.txt', "r")
try:
    tekst = plik.read()
except IOError: # drugi blok (w razie wyjatku)
    print("Nie ma pliku o nazwie", plik)
else: # trzeci blok opcjonalny (nie było wyjatku)
    print("Plik został otwarty ....")
finally:
    plik.close()
print(tekst)
```

In[7]:

```
1 f = open('plik1.txt','w')
2 f.write('hi there\n') # python zamieni \n to os.linesep
3 f.close()
```

In[8]:

```
1 f = open('plik1.txt', 'w')
2 lines = ['pierwsza\n', 'druga\n', 'trzecia\n']
3 f.writelines(lines)
```

In[9]:

```
with open('plik1.txt', 'a') as p1, open('plik2.txt', "w") as p2:
    p1.write("hello")
    p2.write("hello kinga")
```

- Otwarty w programie plik może powodować problemy. Inne aplikacje próbujące go odczytać nie będą mogły tego zrobić.
- Domyślna implementacja CPython działa tak, że newet jeśli nie zamkniemy pliku to przy końcu programu zostanie on zamknięty automatycznie.
- Jednak nie każda implementacja zamyka plik za nas. Dobrą praktyką jest zamykanie tego samodzielnie.
- Alternatywą dla zamykania pliku jest wykorzystanie instrukcji **with**

Mmetody zamykania pliku

Zapis danych na dysku nie następuje natychmiast, ale dopiero wtedy gdy uzbiera się ich większa ilość. Dlatego na końcu powinno się wykonać operację `close()`.

- `close()` – zapisuje ona wszystkie dane na dysku i kasuje obiekt pliku.
- `flush()` – opróżnienie bufora wyjściowego na dysk bez zamykania pliku. Jeśli chcemy zacząć odczytywać plik bez zamykania musimy się upewnić, że nasze zmiany na prawdę znalazły się na dysku. Gdybyśmy nie użyli `flush()` i nie zamkneli otwartego pliku to nasz plik zostałby zamknięty dopiero po zakończeniu działania programu.
- instrukcja **with** jest strukturą kontroli przepływu, której podstawową strukturą jest:
  `with expression [as variable]: with-block`,
  użycie tej instrukcji automatycznie zamyka otwarty plik, więc **instrukcja with nie potrzebuje instrukcji close**.
  `http://effbot.org/zone/python-with-statement.htm`

Należy pamiętać, że obowiązkiem jest zamknięcie pliku. W większości przypadków po zakończeniu działania aplikacji plik zostanie zamknięty. Nie ma jednak gwarancji, kiedy dokładnie to nastąpi. Może to prowadzić do niepożądanych zachowań, w tym utraty danych.

- Użycie bloku: `try - finally`

In[10]:

```
reader = open('dog_breeds.txt')
try:
    # Further file processing goes here
finally:
    reader.close()
```

- Użycie instrukcji: `with` - która automatycznie dba o zamknięcie pliku, po wyjściu z bloku, nawet w przypadku błędu. Zalecane jest korzystanie z `with` tak często, jak to możliwe.

In[11]:

```
with open('dog_breeds.txt') as reader:
    # Further file processing goes here
```

Metody formatujące łańcuchy znaków:

- **f-strings** - `f"string"`: najnowsza i najszybsza metoda formatowania wyrażeń tekstowych w Pythonie. `https://realpython.com/python-f-strings/#simple-syntax`
- **Metoda format** - `str.format()`: metoda formatowania dodana w Pythonie 2.6 i 3.0. Jest dość unikalna dla Pythona i w znacznym stopniu pokrywa się z możliwościami wyrażeń formatujących.
- **Wyrażenia formatujące** `%-formatting`: Oryginalna technika formatowania łańcuchów znaków dostępna w Pythonie od zawsze. Jest oparta na zasadzie zastosowanej w funkcji sprintf języka C. Przykłady użycia tej metody można znaleźć praktycznie w każdym programie napisanym w Pythonie

- PEP 498 wprowadził nowy mechanizm formatowania ciągów tekstowych znany jako **interpolacja łańcuchów literalnych**, zwanej również **f-String** (ze względu na wiodący znak f poprzedzający ciąg literalny). Ideą **f-String** jest uproszczenie interpolacji ciągów.
- Aby utworzyć **f-String**, poprzedz go ciąg literą „f". Sam łańcuch można sformatować w taki sam sposób, jak w przypadku str.format (). Ciągi F zapewniają zwięzły i wygodny sposób osadzania wyrażeń Pythona w literalnych w celu formatowania.

```
1 f"lancuch znakow do sformatowania {format_val}"
```

In[12]:

```
1 imie, wiek = Olek, 23
2 print(f"Mam na imię {imie} i mam {wiek} lat.")
```

Out[12]:

```
1 Mam na imię Olek i mam 23 lata.
```

- https://geeksforgeeks.org/formatted-string-literals-f-strings-python/
- https://realpython.com/python-f-strings/

```
1 "formatowanie {format_val1}, {format_val1}".format(val1, val2)
```

In[13]:

```
1 war = 75.765367
2 print("Jan ma {0:.3f} procent pizzy!".format(war)) # 75.765
3 print("Jan ma {0:.1f} procent pizzy!".format(war)) # 75.8
4 print("Jan ma {0:.0f} procent pizzy!".format(war)) # 76
```

Out[13]:

```
1 Janek ma 75.765 procent pizzy!
2 Janek ma 75.7 procent pizzy!
3 Janek ma 75 procent pizzy!
```

- Wiecej informacji na temat formatowania w Pythonie 2.x i 3.x znajduje się tutaj: https://flynerd.pl/2017/01/python-3-formatowanie-napisow.html

```
"formatowany format_val1 lancuch format_val2" % (val1, val2)
```

- Po lewej stronie operatora % podać formatowany łańcuch zawierający jeden lub większą liczbęosadzonych celów konwersji, z których każdy rozpoczyna się od znaku % (na przykład %d).
- Po prawej stronie operatora % udostępnić obiekt (lub obiekty, zapisane w krotce), który Python ma wstawić do formatowanego łańcucha po lewej stronie w miejsce celu lub celów konwersji.

In[14]:

```
'Ten %d %s jest piekny!' % (1, 'ptak')
"%s -- %.2f -- %s" % (42, 3.14159, [1, 2, 3])
```

Out[14]:

```
Ten 1 ptak jest piekny!
42 -- 3.14 -- [1, 2, 3]
```

Typy Łańcuchowe:

```
val = 'one'
f"1 po angielsku to: {val}"          # f-string
"1 po angielsku to: {}".format(val) # metoda format
"1 po angielsku to: %s " % (val)    # wyrażenie formatujące
```

```
# 1 po angielsku to: one
```

Liczby zmiennoprzecinkowe:

```
a, b = 9.8, 'Usain Bolt'
f"Rekord na 100m to {a:.3f} ustanowił go {b}"          # f-string
"Rekord na 100m to {:.3f} ustanowił go {}".format(a, b)# metoda format
"Rekord na 100m to %.3f ustanowił go %s" % (a, b)      # wyr. formatujące
```

```
# Rekord na 100m to 9.800 ustanowił go Usain Bolt
```

Kody stosowane w wyrażeniach formatujących (Lutz, 2011, p.222)

```
s   # Łańcuch znaków (obiekt obsługujący rzutowanie na łańcuch str(X))
r   #To samo co s, jednak wykorzystuje repr, a nie str
c   #Znak
d   #Liczba całkowita
i   #Liczba całkowita
u   #Równoważne d (dawniej wymuszało liczbę całkowitą bez znaku)
o   #Ósemkowa liczba całkowita
x   #Szesnastkowa liczba całkowita
X   #To samo co x, jednak wyświetlane wielkimi literami
e   #Liczba zmiennoprzecinkowa w formacie wykładniczym, małą literą
E   #To samo co e, wyświetlane wielką literą
f   #Zmiennoprzecinkowa liczba w zapisie dziesiętnym
F   #Zmiennoprzecinkowa liczba w zapisie dziesiętnym
g   #Zmiennoprzecinkowe e lub f
G   #Zmiennoprzecinkowe E lub F
```

Wybrane znaki specjalne które mogę się przydać w formatowaniu wyjścia:

Znak specjalny	Znaczenie
`\n`	znak nowej linii, dodanie „entera”
`\t`	dodanie tabulacji
`\'`	apostrof
`\"`	cudzysłów
`\\`	ukośnik

Informowanie Pythona o kodowaniu znaków

O kodowaniu znaków: (Lutz, 2011, p.913)

Wstawiając tą linie kodu na początku każdego pliku programu Python ustawiamy kodowanie znaków danego pliku (modułu), a nie całego programu (program może się składać z wielu plików):

In[15]:

```
1 #-*- coding: utf-8 -*-
```

Jeśli nie zdefiniujemy kodowania znaków, Python nas o tym uprzedzi wyświetlając komunikaty:

```
1 sys:1: DeprecationWarning: Non-ASCII character '\xc5' in file test.py on
     line 5
2 but no encoding declared; see http://www.python.org/peps/pep-0263.html
     for detils
```

Dodawanie znaków specjalnych (ang. *unicode characters*)

Unicode characters in Python:
https://pythonforundergradengineers.com/unicode-characters-in-python.html

In[16]:

```
delta1 = "\N{GREEK CAPITAL LETTER DELTA}" # użycie nazwy znaku
delta2 = "\u0394"                         # użycie 16-bit hex

pi            = '\u03C0'
alapha        = '\u03B1'
beta          = '\u03B2'
degree_sign   = '\u00b0'
degree_celsius = '\u2103'
```

- csv - odczytuje i zapisuje pliki csv (dane tabelaryczne oddzielone przecinkiem) - **CSV - Comma Separated Values** file (https://docs.python.org/3/library/csv.html)

```
1  # zapis do pliku
2  import csv
3  with open('test_csv.csv', 'w', newline='') as csvfile:
4      spamwriter = csv.writer(csvfile, delimiter=',')
5      spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
6      spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
7
8  # odczyt z pliku
9  with open('test_csv.csv', newline='') as csvfile:
10     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
11     for row in spamreader:
12         print(', '.join(row))
```

- Biblioteka `numpy` posiada wbudowane metody które służą do łatwego i szybkiego wczytywania danych tekstowych:
  - `np.savetxt` - zapis do pliku: https://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html
  - `np.loadtxt` - odczyt z pliku: https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html
  - `np.genfromtxt` - odczyt z pliku z brakującymi wartościami obsługiwanymi zgodnie z opisem: https://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html

- Biblioteka SciPy.io posiada mechanizmy wczytywania (i zapisywania) plików binarnych MATLABA (https://docs.scipy.org/doc/scipy/reference/tutorial/io.html):
  - loadmat(file_name.mat) – wczytuje plik matlaba (.mat).
  - savemat(file_name.mat) – zapisuje tablice do pliku (.mat)
  - whosmat(file_name.mat) – wylistuje zmienne wewnącz pliku (.mat)

```
1 import scipy.io as sio
2 import numpy as np
3 vect = a = np.full((2,3),7) # utworzenie tablicy (2,3) wypełnionej 7
4 sio.savemat('np_vector.mat', {'vect':vect}) # zapis
5 a = sio.whosmat('np_vector.mat') # informacje
6 b = sio.loadmat('np_vector.mat') # odczyt
7 print('dane', b['vect'])          # wyświetlenie danych tablicowych
```

- Ponadto biblioteka SciPy.io posiada mechanizmy pozwalające czytać innego rodzaju formaty: docs.scipy.org/doc/scipy/reference/io.html#module-scipy.io

- `Pandas` również posiada wbudowane metody do odczytu i zapisu plików txt w formacie csv:
  - `pandas.read_csv` - odczytuje plik csv: `pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html`
  - `pandas.DataFrame.to_csv` - zapisuje dane do pliku csv: `pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html`

Trwałość obiektów zaimplementowana została za pomocą trzech modułów biblioteki standardowej, dostępnych w każdej wersji Pythona:

- **pickle** – moduł pickle implementuje binarne protokoły do serializacji i de-serializacji struktury obiektów Pythona. **Pickling** to proces, w którym hierarchia obiektów Pythona jest konwertowana na strumień bajtów, a **unpickling** to odwrotna operacja, w której strumień bajtów (z pliku binarnego lub obiektu podobnego do bajtów) jest ponownie przekształcany w hierarchię obiektów. Pickling (i unpikling) jest alternatywnie nazywane *serializacją*, lub *spłaszczaniem*.(`https://docs.python.org/3/library/pickle.html`)
- **JSON** - biblioteka służy do kodowania i odkodowywania formatu JSON. JSON (JavaScript Object Notation) – jest lekkim formatem wymiany danych inspirowanym składnią dosłowną obiektóœ JavaScript (chociaż nie jest to ścisły JavaScript). Więcej: `https://docs.python.org/3/library/json.html`
- **shelve** - wykorzystuje dwa inne moduły w celu przechowania w pliku obiektów Pythona - dostępnych po kluczu.
- **dbm** (w Pythonie 2.x nosi nazwę anydbm) – implementuje system plików dostępu według klucza, służący do przechowywania łańcuchów znaków.

- Uzupełnieniem wykładów jest Jupyter notebook z przykładami odczytu i zapisu do pliku: `pomoce_podpowiedzi/help_dzialanie_na_plikach_odczyt_zapis.ipynb`
- Uzupełnieniem wykładów jest Jupyter notebook z przykładami wykorzystania **wyrażeń formatujących** oraz **metody format** : `pomoce_podpowiedzi/help_formatowanie_stringow.ipynb`
- Lista znaków specjalnych (unicode) w Pythonie: `pythonforundergradengineers.com/unicode-characters-in-python.html`

M. Lutz. *Python. Wprowadzenie*. Helion, 2011.

Dziękuje za uwagę

Kinga Węzka kinga.wezka@pw.edu.pl