Język programowania – Python
Wybrane narzędzia analityki danych

Marzena Nowakowska

Wydział Zarządzania i Modelowania Komputerowego
Politechnika Świętokrzyska

Budynek C, p. 3.21

spimn@tu.kielce.pl

Spyder

Spyder (https://www.spyder-ide.org/) to IDE napisany w Pythonie i przeznaczony do obsługi programowania w tym języku. Podobnie jak Jupyter, wchodzi w skład pakietu Anaconda. Można jednak zainstalować Spydera bez konieczności instalacji Anacondy

Środowisko Spydera

- Menu główne i pasek narzędzi.
- Lewe okno. Sekcja edycji kodu (edytor tekstowy z analizą kodu i stylu w czasie rzeczywistym); tworzenie skryptów w Pythonie.
- Prawe górne okno. Zakładki dolne: Help, Variable Explorer (eksplorator zmiennych), Plots (wykresy utworzone z wykorzystaniem biblioteki mathplotlib), Files (pliki i foldery).
- Prawe dolne okno. Ipython (konsola) - interaktywne uruchamianie poleceń, przydatne: przywoływanie pomocy na wybrany temat (*help(słowo_kluczowe)*), *History* (historia poleceń wydanych w konsoli).

Każde okno posiada w prawym górny rogu menu kontekstowe ☰ – wykaz opcji menu zależy od zawartości okna.

Tools/Preferences → Wybrać z lewego panelu Working directory.

Tryby pracy: interaktywny (konsola), skryptowy (edytor kodu).

Standardowe typy danych Pythona - 1

Wszystkie dane w Pythonie są obiektami, co oznacza, że właściwie wszystkie zmienne (i stałe) posiadają metody (funkcje), które mogą wykonywać różne operacje na rzecz obiektu. W treści programu wielkość liter ma znaczenie.

Python jest językiem typowanym dynamicznie, tzn. że typ danej zależy od wartości do niej przypisanej.

Liczba całkowita (**int**)

Stała: 12; -100

Zmienna: a = 12; a, b = 12, 100

Liczba zmiennoprzecinkowa (rzeczywista) (**float**)

Stała: 12.5; -100.78; 12.6e-3; 12.

Zmienna: aa = 12.5; aa, bb = 12.6, 13.

Liczba zespolona (część rzeczywista + urojona) (**complex**)

Stała: 2.6+6j

Zmienna: a = 2.6+6j

Typ logiczny (**bool**). Jest podtypem typu całkowitego.

Stała: True; False

Zmienna: a = 12.5 > 18 (wartość a: False)

Łańcuch (napis) (**str** – string) - ciąg znaków między cudzysłowami lub apostrofami. Wielolinijkowy napis umieszcza się między potrójnymi cudzysłowami.

Stała:

```
"Miał pod nosem czarny wąs"
 "Moja mama mówiła: 'Ostatnia umiera nadzieja'"
""" Wszystko to co mam
To ta nadzieja, że życie mnie poskleja"""
```

Zmienna: poczatek = "Miał pod nosem czarny wąs"

Standardowe typy danych Pythona - 2

Lista (**list**). Służy do przechowywania uporządkowanej kolekcji obiektów (danych) a poszczególne elementy zmiennej tego typu mogą podlegać modyfikacji (dodawanie, podmiana i usuwanie elementów).

Stała: ['Python', 'WZiMK', 123, 98.45, 3.6e-3, 2.6+6j]
Zmienna: a = ['Python', 'WZiMK', 123, 98.45, 3.6e-3, 2.6+6j]

Krotka (**tuple**). Służy do przechowywania uporządkowanej kolekcji obiektów (danych) a poszczególne elementy zmiennej tego typu nie mogą podlegać modyfikacji (brak możliwości edycji „w miejscu").

Stała: ('Python', 'WZiMK', 123, 98.45, 3.6e-3, 2.6+6j)
Zmienna: a = ('Python', 'WZiMK', 123, 98.45, 3.6e-3, 2.6+6j)

Słownik (**dict**). Tablica asocjacyjna, w której klucz jest kojarzony z przypisaną mu wartością, przy czym klucze muszą być unikalne. Słownik zapisuje się w postaci par *klucz*: *wartość*, rozdzielonych przecinkami.

Stała: {"klasa_a": 4.5, "klasa_b": 4, "klasa_c": 3.5, "klasa_d": 4.5, "klasa_e": 5 }
Zmienna: sl = {"klasa_a": 4.5, "klasa_b": 4, "klasa_c": 3.5, "klasa_d": 4.5, "klasa_e": 5 }
osoba = {"imię": "Dawid", "nazwisko": "Podsiadło", "rok_ur": 1993}
roma = {1: "I", 2:"II", 3:"III", 4:"IV", 5:"V", 6:"VI", 7:"VII"}

Zbiór (**set**). Nieuporządkowana kolekcja prostych obiektów. Można je kojarzyć ze zbiorami w matematyce. Cechą wyróżniającą zbiór jest unikalność elementów, to znaczy, że elementy w zbiorze nie mogą się powtarzać

Stała: {'bulgaria', 'mexico', 'poland', 'usa'}
Zmienna: VMW = {'bulgaria', 'mexico', 'poland', 'usa'}

Instrukcje języka

Elementy niebieską czcionką określa projektant

```
zmienna = wyrazenie
```

```
if warunek1:
    ciąg_instrukcji1_w bloku_if
[else:
    ciąg_instrukcji1_w bloku_else ]
```

```
for el in obiekt_iterowalny:
    ciąg_instrukcji1_w bloku_for
[else:
    ciąg_instrukcji2_w_bloku_else]
```

```
while wyrażenie_logiczne:
    ciąg_instrukcji1_w bloku_while
[else:
    ciąg_instrukcji2_w_bloku_else]
```

Sformułownie ciąg_instrukcji … oznacza kolejno instrukcje pisane jedna pod drugą z odpowiednim wcięciem (indetancją)

Wybrane operatory

- Operator warunkowy (trójargumentowy):

  wynik_gdy_prawda if *warunek* else *wynik_gdy_fałsz*

  Przykład: wynik = a-b if a >= b else b-a
- Operatory arytmetyczne:

  +, -, *, /, % (modulo), // (dzielenie całkowite), ** (podniesienie do potęgi)
- Operatory porównania: <, <=, >, >=, !=, == (dwa znaki = obok siebie)

  a < b < c to samo co *a < b and b < c*

  a >= b > c to samo co *a >= b and b > c*

  a ==b == c to samo co *a == b and b == c*

  Przykład: 5 < 6 < -1 → False
- Operator przypisania – znak =

  Stosowany w najprostszej instrukcji języka – przypisania

Mutowalność

Dana **niemutowalna** to taka, której wartości nie można zmienić „w miejscu” jej położenia (na stosie). Należą do nich: **liczby**, **wartości logiczne**, **łańcuchy**, **zakresy (*range*)** i **krotki**.

Dane prawie wszystkich innych typów są mutowalne. Dana **mutowalna** to taka, której wartość można zmienić „w miejscu” jej położenia (na stosie). Są to np.: **listy**, **słowniki**, **zbiory**, **obiekty** zdefiniowane przez projektanta (programistę).

Kontenery

Kontener to obiekt takiego typu, który składa się z elementów; przechowuje dowolną liczbę innych obiektów. Kontenery zapewniają sposób **dostępu do zawartych w nich obiektów** i **iterowanie po nich**. Rozmaite typy kontenerów różnią się sposobem dostępu do elementów, modyfikowalnością oraz optymalizacją pod względem różnych zastosowań.

Kontenery

Sekwencje (dostęp do elementu poprzez indeks):
- Łańcuchy
- Krotki
- Listy

Nie sekwencje:
- Słowniki (dostęp do wartości poprzez klucz)
- Zbiory (brak dostępu wprost do elementu, tylko poprzez operacje na zbiorze)

Dostęp do elementów kontenera

- Sekwencje: taki sam dostęp dla listy, krotki i łańcucha

  a = ['Python', 'WZiMK', 123, 98.45, 3.6e-3, 2.6+6j]

  0 in a → False

  a[0] → 'Python'

  a[1::2] → ['WZiMK', 98.45, 2.6+6j]

  Wycinanie [*początek* : *koniec* : *krok*]

  Wartości domyślne: *początek* = 0, *koniec* = nr ostatniego+1, *krok* = 1

  a < ['Python', 'WMiBM'] → False

  Cykliczny dostęp do kolejnych wartości (organizacja pętli) – po indeksach (sekwencja) lub poprzez iterator.

- Słownik: dostęp jak w tablicy asocjacyjnej – przez klucz (klucze się nie powtarzają)

  sl = {"klasa_a": 4.5, "klasa_b": 4, "klasa_c": 3.5, "klasa_d": 4.5, "klasa_e": 5 }

  sl["klasa_a"] → 4.5 (element słownika o kluczu "klasa_a")

  Cykliczny dostęp do kolejnych elementów poprzez poprzez iterator i: do wartości – metoda .values(), do kluczy – metoda .keys() a do pary (klucz, wartość) – metoda .items(). Domyśla iteracja – po kluczach.

- Zbiór: nie ma bezpośredniego dostępu do elementu, dlatego zbiór przekształca się na sekwencję

  VMW = {'bulgaria', 'mexico', 'poland', 'usa'}; VMW_lst = **list**(VMW)

  VMW_lst[0] → 'bulgaria'

  Cykliczny dostęp do kolejnych wartości poprzez iterator.

Praca z kontenerami

Organizacja pętli przez iterator

```
kontener=[1,2,3,4,5]
# kontener="abcdef"
# kontener={11,22,33,"ala", "ola"}
# kontener, np. typu: str, tuple, set, dict
print("Iterator domyślnie w pętli for")
for val in kontener:
    print(val)
```

Organizacja pętli po indeksach

```
kontener=[1,2,3,4,5]
print("Działanie iteratora i indeksu w pętli for")
n=len(kontener)
for i in range (0,n,1):
    print(kontener[i])

# range(0,n,1) → 0, 1, 2, …, n-1
```

Wybrane metody dla kontenerów standardowych

- Lista (**list**); append(), copy(), count(), extend(), remove(), reverse(), sort()
- Krotka (**tuple**): count(), index()
- Łańcuch (**str**): capitalize(), count(), find(), index(), join(), lower(), upper()
- Słownik (**dict**): clear(), copy(), items(), keys(), pop(), values()
- Zbiór (**set**): add(), copy(), issuperset(), remove()

Wybrane pożyteczne funkcje

len() range() slice(), sum() min() max() zip()

Typ zwracanego wyniku zależy od funkcji i jej argumentów.

Złożenie (*comprehension*) w Pythonie

Pyton umożliwia tworzenie nowego obiektu iterowalnego przy wykorzystaniu podanej sekwencji (jednej lub kilku). Taka operacja nazywa się **złożeniem** (*comprehension*).
Jest to zasadniczo sposób napisania zwięzłego bloku kodu w celu wygenerowania obiektu, który może być listą, słownikiem, zbiorem lub generatorem. Może obejmować wiele etapów konwersji różnych typów sekwencji. Składnia złożenia:

Nawias el_definiowany **for** el_definiujący **in** ob_iterowalny **Nawias**

gdzie Nawias jest znakiem właściwym dla danego kontenera.

- Złożenie dla list: ls1 = [i for i in range(5)] ; ls2 = [j**2 for j in ls1]
- Złożenie dla krotek: ls11 = tuple(i for i in range(5)) ; ls22 = tuple(j**2 for j in ls1)
- Złożenie dla zbiorów: ls111 = {i for i in range(5)} ; ls222 = {j**2 for j in ls111}
- Złożenie dla słownika:

```
s1 = [nr for nr in range(1, 8,1)] ;
s2 = ['Pn','Wt','Śr','Czw','Pt', 'Sb', "Nd"]
dni_tyg ={nr: nazwa for (nr, nazwa) in zip(s1, s2)}
```

```
# Jakie obliczenia wykonuje poniższy kod?
x = [5, 1, 10, 1, 15, 1, 20, 1, 25, 1, 30, 1, 35]
suma=sum([el if el%5==0 else 0 for el in x])
print("Suma liczb jakich?:", suma)
```

Biblioteka Pandas

Moduł (pakiet, biblioteka) **Pandas** pracuje głównie ze strukturami tabelarycznymi, które mogą zawierać dane różnych typów. Należy go zaimportować, aby móc korzystać z jej zasobów, np:

```
import pandas as pd        # częstsze, odwołanie: pd.funkcja()
```

Pandas zapewnia dostęp do danych (wykorzystywanych w analityce) o typach:

- **Series** (szereg, seria) dla danych jednowymiarowych
- **DataFrame** (ramka danych) dla wielowymiarowych

Uwaga: może istnieć potrzeba instalacji biblioteki, co można zrealizować z wykorzystaniem programu pip w oknie konsoli Spydera:

```
pip install pandas
```

Dostęp do danych *Series* w Pandas

Przykłady tworzenia serii danych

```
import pandas as pd
s1 = pd.Series([1, 3.2, 0.1, 2.])
s2 = pd.Series([[1, 3.2, 0.1, 2]])
s3 = pd.Series({'f':1, 'g':5, 'h':3, 5:[1,2,3]})
```

s1 - Series

Index	0
0	1
1	3.2
2	0.1
3	2

s1[0] → 1

s2 - Series

Index	0
0	[1, 3.2, 0.1, 2.0]

s2[0] → [1, 3.2, 0.1, 2]

s3 - Series

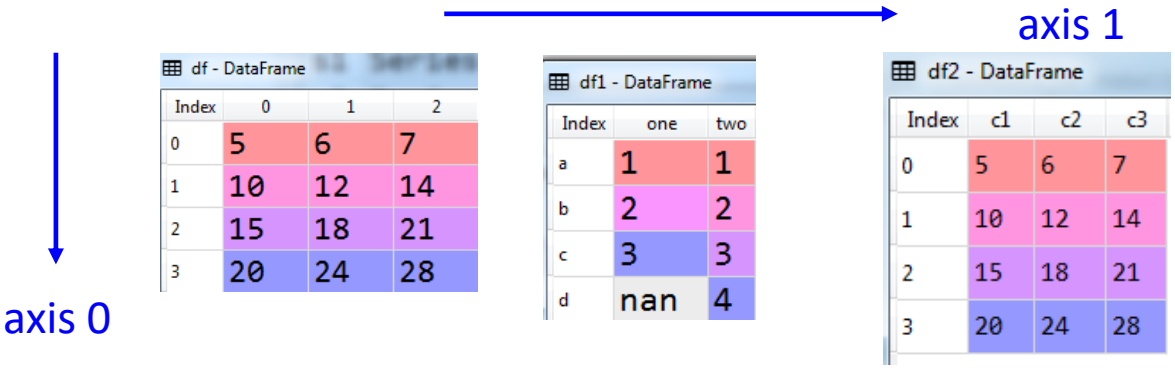
Index	0
f	1
g	5
h	3
5	[1, 2, 3]

s3[‘g'] → 5

Dostęp do danych *DataFrame* w Pandas

Przykłady tworzenie ramek danych

```
import pandas as pd
df=pd.DataFrame([[ 5,  6,  7], [10, 12, 14], [15, 18, 21], [20, 24, 28]])
d = { "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]), "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c", "d"]) }; df1=pd.DataFrame(d)
df2=pd.DataFrame(([[ 5,  6,  7], [10, 12, 14], [15, 18, 21], [20, 24, 28]]), columns=("c1", "c2", "c3"))
```



Dostęp do danych ramki DF:

DF[nr_kolumny]

DF["nazwa_kolumny"]

Dostęp do wiersza – za pomocą atrybutu iloc i indeksu lub loc i etykiety:

DF.iloc[nr_wiersza]

DF.loc["nazwa_wiersza"]

Dostęp do elementu ramki (są różne sposoby dostępu):

DF["nazwa_kolumny", indeks_wiersza]

Odczyt pliku Excela do ramki danych
Wybrane parametry

pandas.read_excel(*file, sheet_name=0, header=0, usecols=None*)

Odczyt pojedynczego arkusza lub listy arkuszy Excela do obiektu typu odpowiednio *DataFrame* lub słownika arkuszy. Parametry:
file – wartość typu *str* definiująca plik do odczytu
sheet_name – łańcuch (np. *'Arkusz1'*, liczba całkowita (np. 1) lub lista (np. [0, 1, *'ArkuszXX'*] określające arkusz (lub arkusze) do wczytania
header – liczba całkowita lub lista liczb całkowitych. Wykaz numerów wierszy arkusza, których zawartość ma być wykorzystana jako nazwy kolumn ramki danych.
usecols – łańcuch. Określa, które kolumny arkusza mają być wczytane do ramki danych.

Przykłady (uwaga: ustalić folder roboczy)

```
dc = pd.read_excel("vehicles.xlsx", sheet_name=["Old Europe", "New Europe"],
        header=0, usecols="A,H:L") # słownik z ramkami
df1 = d1["Old Europe"] : df2=d1["New Europe"]  # wyodrębnienie ramek
df3 = pd.read_excel("vehicles.xlsx",  sheet_name=3,  header=0, index_col=0,
      usecols=[0, 2, 3,5])  # wczytanie ramiki
```

Zapis ramki danych do pliku Excela
Wybrane parametry

pandas.DataFrame.to_excel(*excel_writer*, *sheet_name='Sheet1'*, *columns=None*, *header=True*, *index=True*)

Zapisuje ramkę danych do pliku *Excela*. Parametry:
excel_writer – ścieżka do pliku lub istniejący obiekt *ExcelWriter*.
sheet_name – nazwa arkusza typu *str*, do którego który zapis ramki danych. Domyślnie *'Sheet1*'.
columns – sekwencja lub lista napisów definiująca kolumny do zapisu
header – dana logiczna informująca, czy w pliku mają być zapisane nazwy kolumn (*True*) czy nie (False)
index – dana logiczna informująca, czy w pliku mają być zapisane nazwy indeksy (*True*) czy nie (False)

Uwaga: pandas.DataFrame.to_excel() nie ma wbudowanego parametru, który automatycznie ostrzega lub blokuje zapis, jeśli plik już istnieje.

Przykład (uwaga: ustalić folder roboczy)

```
df4 = df2.drop([2011, 2012, 2013], axis=1)
df4.to_excel('wynik.xlsx', sheet_name = 'Nowa Europa wybrane')
```

Kontrola zapisu do pliku Excela

pandas.ExcelWriter(path-file_name, engine='openpyxl', mode='w', if_sheet_exists=None)

Klasa (narzędzie) w bibliotece pandas, do tworzenia i zapisywania plików Excel (XLSX, XLS) z danych zawartych w ramkach danych. Działa jak pośrednik między Pandas a plikiem Excel. Parametry:

path-file_name – ścieżka do pliku wraz z jego nazwą.

engine – silnik zapisu

mode – tryb zapisu:

- 'w' (write) nadpisanie pliku
- 'a' (append) dopisanie arkusza do istniejącego skoroszytu.

if_sheet_exists – działanie, jeżeli arkusz o nazwie już istnieje:

- 'error' zgłoszenie błędu (domyślnie)
- 'new' utwórzenie nowego arkusza z unikalną nazwą (np. Sheet1)
- 'replace' — nadpisz istniejący arkusz.

Uwaga: pandas.ExcelWriter nie ma wbudowanego parametru, który automatycznie ostrzega lub blokuje zapis, jeśli plik już istnieje.

Rozwiązanie dopisania ramki danych do pliku Excela - podanie nazwy dołączanego arkusza

```
import pandas as pd
# Nazwa pliku Excela
plik = "Skoroszyt.xlsx"

# Nazwa nowego arkusza
arkusz = "Ramka"

dfs = pd.DataFrame({
    "name": ["Anna", "Bartek", "Celina"],
    "age": [25, 30, 22]
})

# Zapis do nowego arkusza w istniejącym pliku
with pd.ExcelWriter(plik, mode="a", if_sheet_exists="new") as writer:
    dfs.to_excel(writer, sheet_name = arkusz, index=False)
```

Pętla po ramce danych

```
df  = pd.read_excel("titanic.xlsx")
```

Pętla po ramce danych z wykorzystaniem atrybutu *index* (id wierszy)

```
for ind in df.index:
        print(df['name'][ind], df['age'][ind])
```

Pętla po ramce danych z wykorzystaniem metody *iterrows* (zwraca *Series* dla każdego wiersza ramki); iterowanie po wierszach

```
for index, row in df.iterrows():
        print(row["name"], row["age"])
```

metoda df.iterrows() zwraca kolejne pary (index, row), gdzie:

- index → to etykieta (indeks) wiersza z ramki danych
- row → to obiekt typu pandas.Series, który zawiera dane z tego wiersza (analogia do wiersza w tabeli bazy danych; dostęp do elementów tego wiersza jest poprzez nazwy kolumn)