Temat: Wybrane biblioteki Pythona.

Cel: zapoznanie się z podstawowymi bibliotekami do przetwarzania danych w Pythonie.

Spis treści

1 *NumPy*

1.1 Wprowadzenie

- *NumPy* to skrót od *Numerical Python* (numeryczny Python)
- biblioteka ta jest podstawą obliczeń numerycznych w Pythonie
- **większość pakietów obliczeniowych oferujących jakieś funkcje naukowe używa obiektów tablicowych *NumPy* jako uniwersalnego środka wymiany danych**
- główne elementy pakietu *NumPy*:
    - ***ndarray*** – wydajna implementacja tablic wielowymiarowych umożliwiająca szybkie wykonywanie tablicowych operacji arytmetycznych i elastyczne **rozgłaszanie** (ang. *broadcasting*)
    - funkcje matematyczne przeznaczone do wykonywania szybkich operacji na całych tablicach danych bez potrzeby tworzenia pętli
    - narzędzia przeznaczone do zapisu i odczytu danych tablicowych z plików umieszczonych na dysku, a także w mapowanym obszarze pamięci
    - obsługa algebry liniowej, generowania liczb losowych i transformacji Fouriera
    - interfejs programistyczny C przeznaczony do łączenia pakietu *NumPy* z bibliotekami napisanymi w języku C, C++ lub Fortran
- biblioteka *NumPy* posiada prosty w użyciu interfejs programistyczny obsługujący język C, co umożliwia przesyłanie danych do zewnętrznych bibliotek napisanych w językach niskiego poziomu oraz zwracanie z tych bibliotek danych w postaci tablic *NumPy*
- możliwość wykorzystywania starych baz kodów C, C++ i Fortrana zapewniła Pythonowi dużą popularność
- pakiet *NumPy* nie zawiera funkcji przeznaczonych do wykonywania obliczeń naukowych czy modelowania danych, ale zrozumienie obsługi tablic *NumPy* ułatwia wykorzystywanie innych narzędzi obsługujących składnię zorientowaną tablicowo
- wśród najważniejszych operacji tablicowych, które mają zastosowanie przy użyciu *NumPy* można wskazać:
    - szybkie operacje przeprowadzane na wektoryzowanych tablicach, które przydają się podczas obróbki danych, tworzenia podzbiorów, filtrowania i przekształcania danych
    - standardowe algorytmy tablicowe takie jak operacje sortowania, znajdowania elementów niepowtarzalnych i tworzenie zestawień
    - wydajne generowanie parametrów statystycznych, a także agregacja i podsumowywanie danych
    - wyrównywanie danych i relacyjne operacje na danych służące do łączenie heterogenicznych zbiorów danych
    - tworzenie logicznych operacji warunkowych bezpośrednio na tablicach bez potrzeby tworzenia zagnieżdżeń if-elif-else
    - grupowe operacje na danych: agregacja, transformacja, stosowanie funkcji
- biblioteka *NumPy* jest zaprojektowana z myślą o wydajnym wykonywaniu obliczeń na dużych tablicach danych, co wynika z takich charakterystyk jak:
    - wewnętrzne pakiet *NumPy* przechowuje dane w stykających się ze sobą blokach pamięci, niezależnie od innych wbudowanych obiektów Pythona
    - algorytmy pakietu *NumPy* napisane w języku C mogą wykonywać operacje na tym obszarze pamięci bez sprawdzania typów i innego narzutu
    - ponadto, tablice *NumPy* zajmują o wiele mniej pamięci niż wbudowane sekwencje Pythona
    - operacje pakietu *NumPy* mogą przetwarzać w sposób złożony całe tablice bez potrzeby tworzenia w Pythonie pętli for (różnica w prędkości działania nawet 100-krotna)
- podstawą mocy biblioteki NumPy jest *wektoryzacja* i *rozgłaszanie*
- **wektoryzacja**:
    - opisuje brak pętli, indeksowania itp. w kodzie (te rzeczy dzieją się "za kulisami" w zoptymalizowanym i prekomilowanym kodzie C
    - zwektoryzowany kod ma wiele zalet, między innymi:
        - kod wektorowy jest bardzo zwięzły i łatwiejszy do odczytania
        - mniej linii kodu oznacza ogólnie mniej błędów
        - kod przypomina bardziej standardową notację matematyczną
        - wektoryzacja daje w rezultacie bardziej "pythoniczny" kod

        - bez wektoryzacji kod byłby zaśmiecony nieefektywnymi i trudnymi do odczytania pętlami for
- **rozgłaszanie**:
    - to termin używano do opisania niejawnego zachowania operacji element po elemencie
    - w NumPy wszystkie operacje, nie tylko arytmetyczne, ale logiczne, bitowe, itp. zachowują się w ten niejawny sposób element po elemencie, tj. rozgłaszają
- NumPy w pełni obsługuje podejście zorientowane obiektowo, zaczynając od *ndarray*
- *ndarray* to klasa posiadająca wiele metod i atrybutów

Przykład wektoryzacji:
załóżmy, że *a* i *b* są dwuwymiarowymi tablicami *NumPy*

kod w języku C:
```
for (i = 0; i < rows; i++): {
  for (j = 0; j < columns; j++): {
    c[i][j] = a[i][j]*b[i][j];
  }
}
```

kod w Pythonie:
```
c = a * b
```

1.2 Typ tablicowy NumPy - *ndarray*

- rdzeniem w *NumPy* są struktury danych w postaci homogenicznych wielowymiarowych tablic
- homogeniczność oznacza, że wszystkie elementy w tablicy są tego samego typu
- głównym obiektem *NumPy* jest jednorodny wielowymiarowy obiekt tablicowy ***ndarray***
- obiekt *ndarray* jest szybkim i uniwersalnym kontenerem przeznaczonym dla dużych zbiorów danych
- poza danymi w tablicach przechowywane są także metadane, np. kształt, rozmiar, typ danych
- tablice *ndarray* pozwalają na wykonywanie operacji matematycznych na całych blokach danych za pomocą składni podobnej do składni definiującej operacje na elementach skalarnych
- tablica *ndarray* jest indeksowana krotką nieujemnych liczb całkowitych
- w *NumPy* wymiary tablicy są nazywane *osiami*

Przykład.
Współrzędne punktu w przestrzeni 3D [1, 2, 1] mają jedną oś. Ta oś ma 3 elementy, więc ma długość 3.
Natomiast tablica:
```
[[1., 0., 0.],
 [0., 1., 2.]]
```
ma dwie osie, pierwsza ma długość 2, druga oś ma długość 3.

- poniżej mamy przykład wygenerowania małej tablicy oraz pomnożenia jej przez skalar i dodania jej samej do siebie
- najpierw instrukcja importu biblioteki w standardowej konwencji:

```
import numpy as np
dane = np.random.randn(2, 3)
dane
```

```
array([[-0.23066027, -1.26541584,  0.98759334],
       [-0.44826661,  0.20159121, -2.37584918]])
```

```
dane * 10
```
```
array([[ -2.30660269, -12.6541584 ,   9.87593341],
       [ -4.48266614,   2.01591205, -23.75849183]])
```

```
dane + dane
```
```
array([[-0.46132054, -2.53083168,  1.97518668],
       [-0.89653323,  0.40318241, -4.75169837]])
```

Podstawowe atrybuty klasy *ndarray*

Atrybut	**Opis**
`shape`	krotka zawierająca liczbę elementów dla każdej osi tablicy
`size`	łączna liczba elementów w tablicy
`ndim`	liczba osi
`nbytes`	liczba bajtów wykorzystywana do przechowywania danych
`dtype`	typ danych przechowywanych w tablicy

1.2.1 Typy danych

- W obliczeniach numerycznych najczęściej są wykorzystywane typy int dla liczb całkowitych i float dla liczb zmiennoprzecinkowych.
- Każdy z tych typów występuje w różnych rozmiarach: `int32`, `int64`, itd.
- Zwykle nie ma potrzeby jawnego wybierania bitowej długości reprezentacji poza określeniem liczb całkowitych lub zmiennoprzecinkowych.
- Obsługiwane są także typy complex i nie liczbowe jak łańcuchy znaków, obiekty, typy złożone zdefiniowane przez użytkownika
- W specjalnym obiekcie *dtype* znajduje się informacja o typie danych zawartych w tablicy *ndarray*
- Dzięki temu obiektowi Python wie jak interpretować określony obszar pamięci zawierający dane:

```
tab1 = np.array([1, 2, 3], dtype=np.float64)
tab2 = np.array([1, 2, 3], dtype=np.int32)
tab1.dtype
```
```
dtype('float64')
```

```
tab2.dtype
```
```
dtype('int32')
```

- nazwy numerycznych typów danych składają się z deskryptora takiego jak np. `float` lub `int` i numeru określającego liczbę bajtów zajmowanych przez dany element
- standardowa wartość zmiennoprzecinkowa o podwójnej precyzji, którą Python używa do wewnętrznie do zapisu obiektów `float`, zajmuje do 8 bajtów (64 bity) – jest to typ `float64`
- wszystkie typy danych obsługiwane przez tablice *ndarray* przedstawia tabela:

Typ	Kod typu	Opis
int8, uint8	i1, u1	8-bitowa liczba całkowita (1 bajt) ze znakiem lub bez
int16, uint16	i2, u2	16-bitowa liczba całkowita ze znakiem lub bez
int32, uint32	i4, u4	32-bitowa liczba całkowita ze znakiem lub bez
int64, uint64	i8, u8	64-bitowa liczba całkowita ze znakiem lub bez

float16	f2	Liczba zmiennoprzecinkowa o połowicznej precyzji
float32	f4 lub f	Standardowa liczba zmiennoprzecinkowa o pojedynczej precyzji, kompatybilna ze zmienną typu `float` języka C
float64	f8 lub d	Standardowa liczba zmiennoprzecinkowa o podwójnej precyzji, kompatybilna ze zmienną typu `double` języka C i obiektem `float` Pythona
float128	f16 lub g	Liczba zmiennoprzecinkowa o rozszerzonej precyzji
complex64	c8	Liczby zespolone złożone z dwóch wartości zmiennoprzecinkowych o długości 32, 64 lub 128 bitów
complex128	c16	
complex256	c32	
bool	?	Wartości logiczne `True` lub `False`
object	O	Typ obiektu Pythona. wartość może być dowolnym obiektem Pythona
string_	S	Łańcuch znaków ASCII o określonej długości, każdy znak zajmuje 1 bajt pamięci. W celu utworzenia łańcucha o długości równej 10 należy skorzystać z typu danych 'S10'
unicode_	U	Łańcuch znaków Unicode o określonej długości (liczba bajtów zależy od platformy). Semantyka specyfikacji jest identyczna jak w przypadku typu string_ , np. 'U10'

1.2.2 Reprezentacja danych tablicowych w pamięci

- z uwagi na wydajność obliczeniową tablice wielowymiarowe są przechowywane w pamięci w sposób ciągły
- można wybrać dowolne rozmieszczenie elementów tablicy w segmencie pamięci
- dla dwuwymiarowej tablicy możliwe są dwa warianty przechowywania danych w postaci sekwencji kolejnych wartości:
  - zapisywanie wierszy jeden pod drugim – tzw. format wierszowy (tak zapisuje język C)
  - zapisywanie kolumn jedna po drugiej – tzw. format kolumnowy (tak zapisuje Fortran)
- sposób reprezentacji tablicy można ustalić w trakcie jej tworzenia lub zmiany kształtu za pomocą argumentu `order`
- wartość `order='C'` tworzy reprezentację wierszową (jest to domyślna reprezentacja)
- wartość `order='F'` tworzy reprezentację kolumnową

1.3 Tworzenie tablic *ndarray*

- tablice *ndarray* najprościej jest utworzyć przy pomocy funkcji `array`, która przyjmuje dowolny obiekt będący sekwencją (w tym inne tablice) i generuje nową tablicę *NumPy* zawierającą przekazane dane:

```
dane1 = [5, 6.5, 7, 0, 1]
tab1 = np.array(dane1)
tab1
```

```
array([5. , 6.5, 7. , 0. , 1. ])
```

- zagnieżdżone sekwencje, np. listy list o równej długości zostaną zamienione na tablice wielowymiarowe:

```
dane2 = [[1, 2, 3], [4, 5, 6]]
tab2 = np.array(dane2)
tab2
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

- ponieważ obiekt *dane2* był listą list, więc tabela *NumPy tab2* ma dwa wymiary:

```
tab2.ndim
```
```
2
```

```
tab2.shape
```
```
(2, 3)
```

- jeżeli podczas tworzenia tabeli nie zostanie określony typ danych w sposób jawny, to `np.array` próbuje określić samodzielnie dobry typ danych dla tworzonej tablicy
- informacje o typie danych są przechowywane w specjalnym obiekcie metadanych o nazwie *dtype*:

```
tab1.dtype
```
```
dtype('float64')
```

```
tab2.dtype
```
```
dtype('int32')
```

- tabela z wykazem wybranych funkcji przeznaczonych do tworzenia tablic ndarray

Funkcja	Opis
`array`	Konwertuje dane wejściowe (listę, krotkę, tablicę lub inny sekwencyjny typ danych) na tablicę *ndarray*, określając samodzielnie typ danych lub korzystając z jawnie zdefiniowanego typu danych. Dane wejściowe są domyślnie kopiowane.
`asarra`	Konwertuje dane wejściowe na tablicę *ndarray*, ale nie kopiuje danych wejściowych, jeżeli zostały one już wcześniej umieszczone w tablicy *ndarray*
`arange`	Działa jak wbudowane funkcja *range*, ale zwraca tablicę *ndarray* zamiast listy
`ones, ones_like`	Generuje tablicę wypełnioną samymi jedynkami o określonym kształcie i typie danych. Funkcja `ones_like` przyjmuje inną tablicę na wejściu i generuje wypełnioną jedynkami tablicę o takim samym kształcie i typie danych jak tablica wejściowa.
`zeros, zeros_like`	Funkcje działają jak `ones` i `ones_like` ale zwracają tablicę zer.
`empty, empty_like`	Tworzy nowe tablice, alokując nową pamięć, ale w przeciwieństwie do funkcji `ones` i `zeros` nie wypełnia tablic żadnymi wartościami.
`full, full_like`	Generuje tablicę o danym kształcie i typie danych, a wszystkie elementy tablicy przyjmują określoną wartość. Funkcja `full_like` przyjmuje inną tablicę i generuje wypełnioną tablicę o takim samym kształcie i typie danych.
`eye, identity`	Tworzy kwadratową macierz tożsamościową NxN (jedynki umieszczane są na głównej przekątnej, a pozostałe elementy przyjmują wartość równą zero).

- przykłady tworzenia tablic za pomocą funkcji `zeros, empty, arange`:

```
np.zeros(10)
```
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
np.zeros((3, 6))
```
```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

```
np.empty((2, 3, 4))
```
```
array([[[0.2 , 0.2 , 0.6 , 1.  ],
        [0.  , 0.6 , 1.  , 1.  ],
        [0.  , 0.8 , 0.4 , 1.  ]],

       [[1.  , 1.  , 0.6 , 1.  ],
        [0.5 , 0.36, 0.33, 1.  ],
        [1.  , 1.  , 1.  , 1.  ]]])
```

Uwaga: jak widać z przykładu funkcja `empty` nie zawsze zwraca tablicę wypełnioną zerami.

- wypełnianie tablic wartościami rosnącymi: `arange` i `linspace`
- funkcja `arange` jest odpowiednikiem wbudowanej funkcji Pythona `range`, która jest przeznaczona do pracy z tablicami
- funkcja `arange`: dwa pierwsze parametry to wartość początkowa i końcowa (bez tej wartości), a trzeci parametr to skok inkrementacji:

```
np.arange(12)
```
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
np.arange(2,12)
```
```
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
np.arange(2,12,3)
```
```
array([ 2,  5,  8, 11])
```

- funkcja `linspace`: dwa pierwsze parametry to wartość początkowa i końcowa (z tą wartością), a trzeci parametr to łączna liczba elementów jaka ma się znaleźć w tablicy:

```
np.linspace(1, 50)
```
```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13.,
       14., 15., 16., 17., 18., 19., 20., 21., 22., 23., 24., 25., 26.,
       27., 28., 29., 30., 31., 32., 33., 34., 35., 36., 37., 38., 39.,
       40., 41., 42., 43., 44., 45., 46., 47., 48., 49., 50.])
```

```
np.linspace(0, 10, 11)
```
```
Out[10]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

- metoda `astype` obiektu *ndarray* pozwala na jawną konwersję (rzutowanie) jednego typu danych w inny; w poniższym przykładzie dokonano rzutowania wartości stałoprzecinkowych (`int`) na wartości zmiennoprzecinkowe (`float`):

```
tab = np.array([1, 2, 3, 4, 5])
tab.dtype
```
```
dtype('int32')
```

```
float_tab = tab.astype(np.float64)
float_tab.dtype
```
```
dtype('float64')
```

- podczas rzutowania odwrotnego, ułamek dziesiętny jest odcinany od wartości całkowitoliczbowej:

```
tab = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
tab
```
```
array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
```

```
tab.astype(np.int32)
```
```
array([ 3, -1, -2,  0, 12, 10])
```

1.3.1 Działania matematyczne z tablicami *NumPy*

- na tablicach *NumPy* można wykonywać operacja wsadowe, tzn. bez konieczności tworzenia pętli *for*, co nazywane jest **wektoryzacją**
- wszystkie operacje arytmetyczne wykonywane na tablicach o równych rozmiarach są przeprowadzane element po elemencie (ang. *element-wise*):

```
tab = np.array([[1., 2., 3.], [4., 5., 6.]])
tab
```
```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

```
tab * tab
```
```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

```
tab - tab
```
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

- operacje arytmetyczne z wartościami skalarnymi są wykonywane na każdym elemencie tablicy:

```
1/tab
```
```
array([[1.        , 0.5       , 0.33333333],
       [0.25      , 0.2       , 0.16666667]])
```

- porównanie dwóch tablic o takich samych rozmiarach generuje tablicę wartości logicznych:

```
tab2 = np.array([[0., 4., 1.], [7., 2., 12.]])
tab2
```
```
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])
```

```
tab2 > tab
```
```
array([[False,  True, False],
```

```
       [ True, False,  True]])
```

- Uwaga: operacje pomiędzy tablicami o różnych rozmiarach określa się mianem rozgłaszania (ang. *broadcasting*)

1.3.2 Podstawy indeksowania tablic i zakresy

- istnieje kilka sposobów wybierania pojedynczych elementów lub ich podzbioru z tablic *NumPy*
- w przypadku tablic jednowymiarowych operacje wykonuje się tak jak na listach Pythona:

```
tab = np.arange(10)
tab
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
tab[5]
```

```
5
```

```
tab[5:8]
```

```
array([5, 6, 7])
```

```
tab[5:8] = 12
tab
```

```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

- w ostatnim przykładzie przypisano skalar do wycinka, wartość ta ulega propagacji tzn. jest rozgłaszana i umieszczana w całym wybranym obszarze
- Uwaga: warto zwrócić szczególną uwagę na różnicę pomiędzy wbudowanymi w Pythona listami a tablicami NumPy – przechwycone wycinki tablic są widokami elementów oryginalnej tablicy, co oznacza, że dane nie są kopiowane a zatem modyfikacja wycinków spowoduje wprowadzenie zmian w tablicy źródłowej!!!

```
tab_wyc = tab[5:8]
tab_wyc
```

```
array([12, 12, 12])
```

```
tab_wyc[1] = 12345
tab
```

```
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,     9])
```

```
tab_wyc[:] = 64
tab
```

```
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

- do kopiowania wycinka tablicy *ndarray* używana jest funkcja `copy`
- w przypadku tablicy dwuwymiarowej elementy znajdujące się pod każdym z indeksów nie są skalarami lecz tablicami jednowymiarowymi:

```
tab2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
tab2d[2]
```

```
array([7, 8, 9])
```

- dostęp do poszczególnych elementów można uzyskać na dwa sposoby:

```
tab2d[0][2]
```

```
3
```

lub

```
tab2d[0, 2]
```

```
3
```

- Schemat indeksowania tablicy dwuwymiarowej:

		Oś 1		
		0	1	2
Oś 0	0	0, 0	0, 1	0, 2
	1	1, 0	1, 1	1, 2
	2	2, 0	2, 1	2, 2

- w przypadku pominięcia kolejnych indeksów tablic wielowymiarowych zwracany jest obiekt będący tablicą *ndarray* o mniejszej liczbie wymiarów; zawiera on wszystkie dane zapisane w wyższych wymiarach:

```
tab3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
tab3d
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Obiekt tab3d jest tablica 2x3:

```
tab3d[0]
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

Do tab3d[0] mogą być przypisane wartości skalarne lub tablice:

```
old_values = tab3[0].copy()
tab3d[0] = 42
tab3d
```

```
array([[[42, 42, 42],
        [42, 42, 42]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

```
tab3d[0] = old_values
tab3d
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

- zastosowanie składni tab3d[1, 0] spowoduje zwrócenie wartości, których indeksy zaczynają się od (1, 0) w formie jednowymiarowej tablicy:

```
tab3d[1, 0]
```

```
array([7, 8, 9])
```

- Inne metody indeksowania:
  - za pomocą wycinków
  - za pomocą kontroli spełnienia warunków logicznych
  - za pomocą indeksowania specjalnego

1.3.3 Transponowanie tablic

- transpozycja jest specjalną formą przekształcania macierzy, która zwraca widok danych bez kopiowania ich
- tablice dysponują metodą `transpose` oraz specjalnym atrybutem **T**:

```
tab = np.arange(15).reshape((3, 5))
tab
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
tab.T
```

```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

- operacja transponowania jest często wykonywana podczas działań na macierzach, np. mnożenia dwóch macierzy za pomocą funkcji np.dot:

```
tab = np.random.randn(6, 3)
tab
```

```
array([[-0.84851427,  0.43902386,  1.16554201],
       [-0.36278313, -1.33342008,  2.87299612],
       [-0.27017324, -0.22865247,  1.42388204],
       [ 0.94031488,  1.45025643,  0.26349599],
       [-1.62889485,  0.62270187, -0.05850375],
       [-0.52556509, -2.28805216,  1.14946622]])
```

```
np.dot(tab.T, tab)
```

```
array([[ 4.73829078,  1.72490223, -2.67700202],
       [ 1.72490223,  9.74921704, -5.92911642],
       [-2.67700202, -5.92911642, 13.03416038]])
```

1.3.4 Przegląd funkcji uniwersalnych

- funkcja uniwersalna to funkcja, która wykonuje operacje na poszczególnych elementach danych tablic *ndarray* i można je postrzegać jako szybkie wektorowe obudowy prostych funkcji, które przyjmują przynajmniej jedną wartość skalarną i generują wynik w postaci przynajmniej jednej wartości skalarnej
- przykłady funkcji uniwersalnych jednoargumentowych to:
    - abs, fabs – oblicza wartość bezwzględną element po elemencie dla wartości zmiennoprzecinkowych, stałoprzecinkowych i liczb zespolonych
    - sqrt – oblicza pierwiastek kwadratowy każdego elementu (ekwiwalent arr**0.5)
    - square – oblicza wartość każdego elementu podniesionego do kwadratu (ekwiwalent arr**2)
    - ceil – oblicza najmniejszą wartość całkowitą większą lub równą danej wartości
    - floor – oblicza najmniejszą wartość całkowitą mniejszą lub równą danej wartości
    - cos, cosh, sin, sinh, tan, tanh – regularne i hiperboliczne funkcje trygonometryczne
- przykłady funkcji uniwersalnych dwuargumentowych:
    - add – dodaje odpowiadające sobie elementy tablic
    - substract – odejmuje elementy drugiej tablicy od elementów pierwszej tablicy
    - multiply – mnozy elementy tablicy
    - divide, floor_divide – dzielenie i dzielenie bez reszty (reszta jest pomijana)
    - power – podnosi elementy pierwszej tablicy do potęg określanych przez wartości umieszczone w drugiej tablicy

2 *Pandas*

2.1 Przeznaczenie biblioteki *pandas*

- zawiera struktury danych i narzędzia przeznaczone do przetwarzania danych, które ułatwiają i przyspieszają wstępną obsługę (np. oczyszczanie) danych i ich analizę w Pythonie
- biblioteka *pandas* jest często używana w połączeniu z innymi narzędziami przeznaczonymi do przetwarzania danych numerycznych, takimi jak *NumPy* i *SciPy*, bibliotekami analitycznymi, takimi jak *scikit-learn* i *statsmodels*, a także bibliotekami przeznaczonymi do wizualizacji danych, np. *matplotlib*
- pakiet *pandas* jest podobny do pakietu *NumPy* ponieważ także przetwarza tablice i oferuje wiele funkcji operujących na tablicach i umożliwia przetwarzanie danych bez pętli for
- **podstawowa różnica jest taka, że *NumPy* działa na homogenicznych tablicach danych liczbowych, a *pandas* może przechowywać i przetwarzać dane heterogeniczne**
- zwyczajową konwencją do importu biblioteki *pandas* jest klauzula:

```
import pandas as pd
```

- zatem wszędzie gdzie w kodzie występuje zapis `pd.` dany fragment odwołuje się do biblioteki *pandas*
- ponieważ bardzo często używa się modułów *Series* oraz *DataFrame* to można je także załadować do lokalnej przestrzeni nazw:

```
from pandas import Series, DataFrame
```

2.2 Opis struktur danych biblioteki *pandas*

- główne struktury danych biblioteki *pandas* to **serie** (*Series*) i **ramki danych** (DataFrame)

2.2.1 Obiekty *Series*

- **Seria** to jednowymiarowy obiekt przypominający tablicę
- składa się z sekwencji wartości oraz z **index**-u który jest tablicą etykiet odnoszących się do danych
- obiekt typu *Series* można porównać do uporządkowanych słowników o określonej długości, ponieważ w obu strukturach mamy do czynienia z przypisaniem wartości indeksu do wartości danych
- zatem obiekty *Series* można stosować w wielu kontekstach w których używa się słowników
- serię można utworzyć przez tablicę danych:

```
seria1 = pd.Series([1, 5, -8, 3])
seria1
```

```
0    1
1    5
2   -8
3    3
dtype: int64
```

- domyślne indeksy w postaci liczb od 0 pokazane są po lewej stronie, a po prawej odpowiadające im wartości serii
- aby wyświetlić oddzielnie wartości i indeksy korzysta się z metod `values` oraz `index`:

```
seria1.values
```

```
array([ 1,  5, -8,  3], dtype=int64)
```

```
seria1.index
```
```
RangeIndex(start=0, stop=4, step=1)
```

- indeks obiektu *Series* może być modyfikowany za pomocą operacji przypisania:

```
seria1
```
```
0    1
1    5
2   -8
3    3
dtype: int64
```

```
seria1.index = ['Jan', 'Jacek', 'Jurek', Jarek']
seria1
```
```
Jan      1
Jacek    5
Jurek   -8
Jarek    3
dtype: int64
```

- można tworzyć obiekt *Series* z indeksem identyfikującym każdy element serii za pomocą etykiety:

```
seria2 = pd.Series([1, 5, -8, 3], index=['a', 'b', 'c', 'x'])
seria2
```
```
a    1
b    5
c   -8
x    3
dtype: int64
```

```
seria2.index
```
```
Index(['a', 'b', 'c', 'x'], dtype='object')
```

- aby wybrać pojedynczą wartość lub zbiór wartości można wykorzystać etykiety umieszczone w indeksie:

```
seria2['x']
```
```
3
```

```
seria2[['x', 'c', 'a']]
```
```
x    3
c   -8
a    1
dtype: int64
```

- należy zauważyć, że indeksy nie muszą być liczbowe, mogą być łańcuchami
- dane w formie słownika można przekształcić do postaci serii:

```
sdata = {'WNiUO': 1000, 'WME': 2000, 'WNHiS': 3000, 'WDiOM': 4000}
seria3 = pd.Series(sdata)
seria3
```
```
WNiUO    1000
```

```
WME      2000
WNHiS    3000
WDiOM    4000
dtype: int64
```

- indeksy można przygotować wcześniej w postaci listy i wykorzystać w metodzie *Series*:

```
wydzialy = ['WNiUO', 'WME', 'IDSZ', 'WNHiS']
seria4 = pd.Series(sdata, index=wydzialy)
seria4
```

```
WNiUO    1000.0
WME      2000.0
IDSZ        NaN
WNHiS    3000.0
dtype: float64
```

ponieważ nie znaleziono wartości klucza 'IDSZ' to przypisano mu wartość *NaN* – nie liczba, natomiast index WDiOM nie został uwzględniony na liście wydziałów więc nie znalazł się w obiekcie wyjściowym

- brakujące wartości lub inaczej brakujące dane można określić przy pomocy funkcji `isnull` oraz `notnull`:

```
pd.isnull(seria4)
```

```
WNiUO    False
WME      False
IDSZ      True
WNHiS    False
dtype: bool
```

```
pd.notnull(seria4)
```

```
WNiUO     True
WME       True
IDSZ     False
WNHiS     True
dtype: bool
```

- obiekty typu *Series* automatycznie wyrównują indeksy podczas operacji arytmetycznych:

```
seria3
```

```
WNiUO    1000
WME      2000
WNHiS    3000
WDiOM    4000
dtype: int64
```

```
seria4
```

```
WNiUO    1000.0
WME      2000.0
IDSZ        NaN
WNHiS    3000.0
dtype: float64
```

```
seria3 + seria4
```

```
IDSZ        NaN
```

```
WDiOM       NaN
WME      4000.0
WNHiS    6000.0
WNiUO    2000.0
dtype: float64
```

jest to coś na kształt operacji `join` w języku `sql`

2.2.2 Obiekty *DataFrame*

- obiekt *DataFrame* (ramka danych) jest **prostokątną tabelą** danych z **uporządkowanym zbiorem kolumn**
- w każdej kolumnie może znajdować się wartość innego typu, tzn. liczba, łańcuch znaków, wartość logiczna
- ramka danych posiada indeksy wierszy i kolumn
- można ją postrzegać jako słownik obiektów *Series* współdzielących ten sam *index*
- Python nie przechowuje ramek danych w formie listy, słownika ani zbioru jednowymiarowych tablic, natomiast dane są przechowywane w formie dwuwymiarowych bloków (i to powinno wystarczyć do zrozumienia struktury)
- wynika z tego, że obiekt *DataFrame* ma charakter dwuwymiarowy, ale może być używany do reprezentowania danych o większej liczbie wymiarów (przez indeksowanie hierarchiczne)
- obiekty *DataFrame* mogą być tworzone na wiele różnych sposobów, ale najczęściej generuje się je na podstawie słownika list o równej długości lub tablic *NumPy*:

```
dane = {'wydział': ['WNiUO', 'WNiUO', 'WNiUO', 'WME', 'WME', 'WME'], \
        'rok': [2019, 2020, 2021, 2019, 2020, 2021], \
        'studenci': [1000, 1100, 1200, 1050, 1150, 1250]}
ramka = pd.DataFrame(dane)
```

- obiekt DataFrame będzie posiadał automatycznie przypisany indeks (podobnie jak dla obiektów Series)

```
ramka
```

```
  wydział   rok  studenci
0   WNiUO  2019      1000
1   WNiUO  2020      1100
2   WNiUO  2021      1200
3     WME  2019      1050
4     WME  2020      1150
5     WME  2021      1250
```

- można określić kolejność ustawienia kolumn obiektu DataFrame:

```
pd.DataFrame(dane, columns=['rok', 'wydział', 'studenci'])
```

```
    rok wydział  studenci
0  2019   WNiUO      1000
1  2020   WNiUO      1100
2  2021   WNiUO      1200
3  2019     WME      1050
4  2020     WME      1150
5  2021     WME      1250
```

- jeżeli podczas tej operacje zostanie przekazana kolumna, której nie ma w słowniku, to zostanie ona dodana do obiektu *DataFrame*, ale zostanie wypełniona wartościami **NaN**; zwróć uwagę na możliwość nadpisania etykiet indeksu:

```
ramka2 = pd.DataFrame(dane, columns=['rok', 'wydział', 'studenci',
'dziekan'], \
index=['raz', 'dwa', 'trzy', 'cztery', 'pięć', 'sześć'])
ramka2
```

```
         rok wydział  studenci dziekan
raz     2019   WNiUO      1000     NaN
dwa     2020   WNiUO      1100     NaN
trzy    2021   WNiUO      1200     NaN
cztery  2019     WME      1050     NaN
pięć    2020     WME      1150     NaN
sześć   2021     WME      1250     NaN
```

- dostęp do kolumny obiektu *DataFrame* można uzyskać za pomocą notacji przypominającej notację słownikową lub za pomocą atrybutu; w obu przypadkach zostanie zwrócony obiekt *Series*

```
ramka2['wydział']
```

```
raz       WNiUO
dwa       WNiUO
trzy      WNiUO
cztery      WME
pięć        WME
sześć       WME
Name: wydział, dtype: object
```

```
ramka2.wydział
```

```
raz       WNiUO
dwa       WNiUO
trzy      WNiUO
cztery      WME
pięć        WME
sześć       WME
Name: wydział, dtype: object
```

Uwagi:

  - korzystając z notacji opartej na atrybucie (`ramka2.wydział`) można korzystać z podpowiedzi przez zastosowanie przycisku *Tab*
  - składnia `obiekt[kolumna]` działa z dowolną nazwą kolumny, a składnia `obiekt.kolumna` działa tylko, gdy nazwa kolumny jest poprawną nazwą zmiennej Pythona
  - zwrócony obiekt *Series* ma ten sam index co obiekt *DataFrame*

- dostęp do wierszy można uzyskać za pomocą pozycji lub nazwy i specjalnego atrybutu *loc*

```
ramka2.loc['trzy']
```

```
rok          2021
wydział     WNiUO
studenci     1200
dziekan       NaN
```

```
Name: trzy, dtype: object
```

- utworzenie nowej kolumny w obiekcie *DataFrame* z jednoczesnym wprowadzeniem określonych wartości:

```
ramka2['rektor'] = 'Jan Nowak'
ramka2
```

```
         rok wydział  studenci dziekan     rektor
raz     2019   WNiUO      1000     NaN  Jan Nowak
dwa     2020   WNiUO      1100     NaN  Jan Nowak
trzy    2021   WNiUO      1200     NaN  Jan Nowak
cztery  2019     WME      1050     NaN  Jan Nowak
pięć    2020     WME      1150     NaN  Jan Nowak
sześć   2021     WME      1250     NaN  Jan Nowak
```

Uwaga:
  - Nowe kolumny nie mogą być tworzone za pomocą składni `obiekt.kolumna`

- usunięcie kolumny z obiektu *DataFrame* za pomocą metody *del*

```
del ramka2['rektor']
ramka2.columns
```

```
Index(['rok', 'wydział', 'studenci', 'dziekan'], dtype='object')
```

2.2.3 Obiekty *index*

- indeksy, czyli obiekty index są używane do przechowywania etykiet osi lub innych metadanych, takich jak np. nazwy osi
- tablica lub inna sekwencja etykiet może zostać użyta podczas tworzenia serii lub ramki danych w celu jawnego zdefiniowania indeksu:

```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
index = obj.index
index
```

```
Index(['a', 'b', 'c'], dtype='object')
```

```
index[1:]
```

```
Index(['b', 'c'], dtype='object')
```

- indeksy są obiektami niemodyfikalnymi, użytkownik nie może ich zmieniać:

```
index[1] = 'd'     # Błąd typu TypeError
```

- indeksy biblioteki pandas mogą zawierać zduplikowane etykiety:

```
dupl_etykiety = pd.Index['raz', 'raz', 'dwa', 'dwa']
dupl_etykiety
```

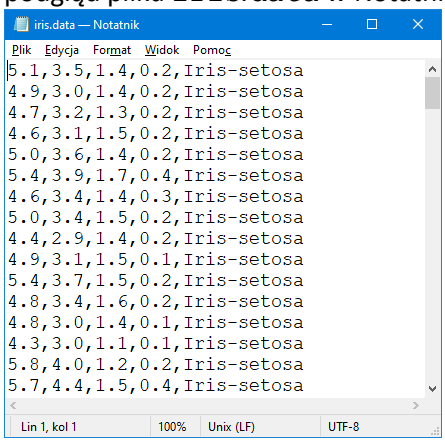
```
Index(['raz', 'raz', 'dwa', 'dwa'], dtype='object')
```

2.3 Wczytywanie i wstępne przetwarzanie danych za pomocą biblioteki *pandas*

- biblioteka *pandas* dostarcza funkcję do wczytywania danych tabelarycznych z pliku lub na podstawie adresu URL
- funkcja ta:
  - zapisuje dane w specjalnej strukturze danych – obiekt jest typu *DataFrame*
  - indeksuje wiersze tabel
  - rozdziela zmienne za pomocą niestandardowych ograniczników
  - ustala typ danych odpowiedni dla każdej kolumny
  - przeprowadza konwersję danych jeśli zachodzi taka konieczność
  - parsuje daty
  - wykonuje obsługę brakujących danych lub wartości błędnych
- przykład wczytania zbioru danych z pliku (csv) `iris.data` do obiektu typu *DataFrame*, jeżeli zbiór danych jest zapisany na komputerze:

```
import pandas as pd
iris_filename = 'iris.data'
iris = pd.read_csv(iris_filename, sep=',', \
decimal='.', header=None, \
names=['sepal_lenght', 'sepal_width', \
'petal_lenght', 'petal_width', 'target'])
iris.head()
```

- podgląd pliku `iris.data` w Notatniku:



- przykład wczytania zbioru danych przy wykorzystaniu adresu URL:

```
import urllib
url = 'http://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data'
set1 = urllib.request.Request(url)
iris_p = urllib.request.urlopen(set1)
iris_other = pd.read_csv(iris_p, sep=',', \
decimal='.', header=None, \
names=['sepal_lenght', 'sepal_width', \
'petal_lenght', 'petal_width', 'target'])
iris_other.head()
```

- funkcje `.head()` i `.tail()`
  - funkcje `.head()` i `.tail()` wywołane na obiekcie typu *DataFrame* wyświetlają domyślnie 5 wierszy początkowych i 5 wierszy końcowych zbioru danych
  - domyślne ustawienia można "nadpisać": `.head(2)` lub `.tail(3)`

- wyświetlenie nazw kolumn:

```
iris.columns
Index(['sepal_lenght', 'sepal_width','petal_lenght', 'petal_width',
'target'], dtype='object')
```

Uwaga: obiekt zwrócony jest typu *Index* a nie typu lista i zgodnie z nazwą określa inseksy nazw kolumn. Klasa *Index* z biblioteki pandas działa jak słownik z indeksem kolumn tabeli. Przykładowo aby pobrać kolumnę `target`, należy wykonać kod:

```
y = iris['target']
y
```

```
0          Iris-setosa
1          Iris-setosa
2          Iris-setosa
3          Iris-setosa
4          Iris-setosa

145    Iris-virginica
146    Iris-virginica
147    Iris-virginica
148    Iris-virginica
149    Iris-virginica
Name: target, Length: 150, dtype: object
```

w przykładzie powyżej obiekt y jest typu *Series* z biblioteki pandas, który można traktować jak jednowymiarowa tablicę
na podstawie listy nazw kolumn można utworzyć obiekt typu *DataFrame*, jak w przykładzie:

```
X = iris[['sepal_lenght', 'sepal_width']]
X
```

```
sepal_lenght  sepal_width
0              5.1          3.5
1              4.9          3.0
2              4.7          3.2
3              4.6          3.1
4              5.0          3.6
..             ...          ...
145            6.7          3.0
146            6.3          2.5
147            6.5          3.0
148            6.2          3.4
149            5.9          3.0

[150 rows x 2 columns]
```

uwagi do powyższego przykładu:

- w pierwszej z dwóch instrukcji powyżej zażądano jednej kolumny, dlatego dane zwrócone są w postaci jednowymiarowego wektora, czyli obiektu *Series* z biblioteki *pandas*

- w drugiej instrukcji zażądano dwóch kolumn, dlatego zwrócony wynik jest macierzą, czyli obiektem *DataFrame* z biblioteki pandas
- jak dostrzec różnice: poprzez zwrócony format danych – zwracany wektor (obiekt *Series*) nie ma nagłówka, a kolumny obiektu *DataFrame* mają nazwy
- po wczytaniu danych zwykle oddziela się cechy od wartości docelowych
- w problemach klasyfikacyjnych wartości docelowe są zwykle w postaci liczb o wartościach dyskretnych lub w postaci pojedynczych znaków (traktowane jako tekst)
- w problemach regresji wartości docelowe są liczbami rzeczywistymi