Fundusze Europejskie dla Rozwoju Społecznego
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską

Politechnika Świętokrzyska
Wydział Zarządzania i Modelowania Komputerowego

Kierunek studiów:
Inżynieria danych

Paweł Stąpór

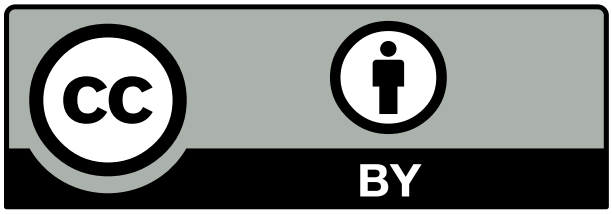
Materiały dydaktyczne do przedmiotu

JĘZYK PROGRAMOWANIA PYTHON

opracowane w ramach realizacji Projektu
„Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki”
FERS.01.05-IP.08-0234/23

Kielce, 2025

Spis treści



1. Uwagi wstępne

Materiały te adresowane są przede wszystkim do studentów uczących się języka programowania Python w ramach przedmiotu *Język programowania Python*. To zbiór notatek na temat Pythona, w szczególności dotyczących korzystania z wybranych pakietów takich jak Numpy, Pandas i Matplotlib.

2. Język Python

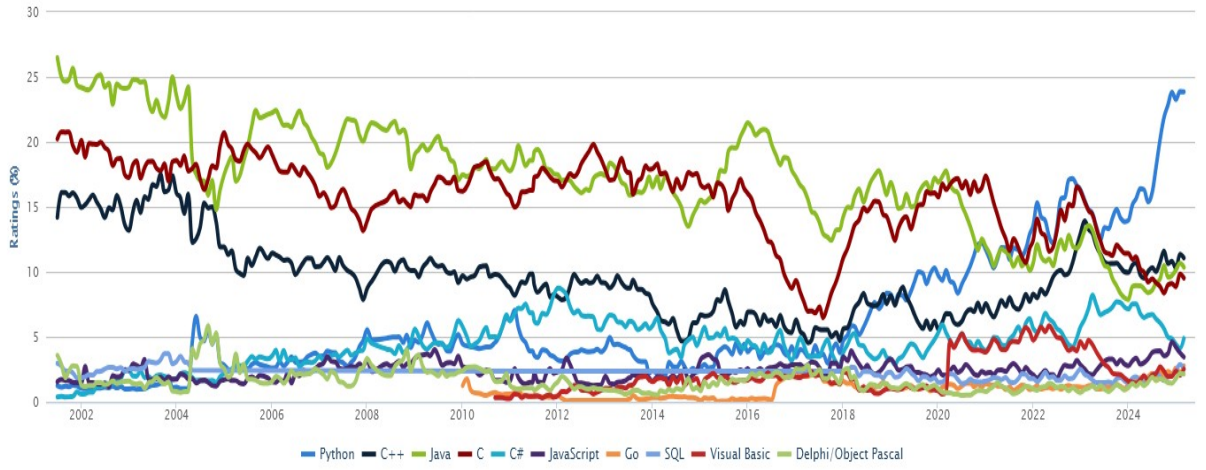
Python jest prawdopodobnie najłatwiejszym do nauczenia i najprzyjemniejszym w użyciu językiem programowania w powszechnym użyciu. Kod Pythona jest przejrzysty w czytaniu i pisaniu, a także zwięzły, bez popadania w zawiłości. Python jest językiem bardzo ekspresyjnym, co oznacza, że zazwyczaj możemy napisać znacznie mniej linii kodu Pythona niż byłoby to wymagane w przypadku równoważnej aplikacji napisanej, powiedzmy, w C++ lub Javie. Python jest językiem wieloplatformowym: zasadniczo ten sam program w Pythonie można uruchomić w systemie Windows i systemach uniksopodobnych, takich jak Linux, BSD i Mac OS X, po prostu kopiując plik lub pliki tworzące program na maszynę docelową, bez konieczności „rekompilowania”. Możliwe jest tworzenie programów w Pythonie, które wykorzystują funkcje specyficzne dla danej platformy, ale rzadko jest to konieczne, ponieważ prawie cała standardowa biblioteka Pythona i większość bibliotek zewętrznych jest w pełni wieloplatformowa. Jedną z największych zalet Pythona jest to, że zawiera bardzo kompletną bibliotekę standardową – pozwala nam to wykonywać takie czynności, jak pobieranie plików z internetu, rozpakowywanie skompresowanych plików archiwum lub tworzenie serwera WWW, a wszystko to za pomocą zaledwie jednej lub kilku linijek kodu. Oprócz biblioteki standardowej dostępne są tysiące bibliotek zewnętrznych, z których niektóre oferują bardziej zaawansowane funkcje niż biblioteka standardowa – na przykład biblioteka sieciowa Twisted czy biblioteka numeryczna NumPy – podczas gdy inne oferują funkcjonalność, która jest zbyt wyspecjalizowana, aby znaleźć się w bibliotece standardowej – na przykład pakiet symulacyjny SimPy. Większość bibliotek zewnętrznych jest dostępna w indeksie pakietów Pythona (pypi.python.org/pypi). Python może być używany do programowania proceduralnego, obiektowego i, w mniejszym stopniu, funkcyjnego, chociaż w gruncie rzeczy Python jest językiem obiektowym.

Python jest interpretowalnym językiem programowania wysokiego poziomu stworzonym przez holenderskiego programistę Guido van Rossuma w 1990 roku. W zasadzie można go określić mianem “pomysłodawcy” tego języka programowania, bo w jego rozwój zaangażowanych było znacznie więcej osób. Nazwa języka wywodzi się z programu telewizyjnego BBC „Latający cyrk Monty Pythona”. Obecnie

Python rozwijany jest jako projekt Open Source, zarządzany przez Python Software Foundation, będącą organizacją non-profit.

W ciągu paru dekad język Python stał się jednym z najważniejszych i najpopularniejszych narzędzi programowania (rys. 1) według indeksu TIOBE (https://www.tiobe.com/tiobe-index).

[Tekst alternatywny. Wykres dwuwymiarowy z kilkoma seriami. Opis osi x: kolejne lata od 2002 do 2025; opis osi y: popularność w procentach poszczególnych języków programowania, takich jak: Python, C++, Java, C#, JavaScript, Go, SQL, Visual Basic, Delphi/Object Pascal.]



Rys. 1. Popularność języków programowania

3. Biblioteka NumPy

NumPy (skrót od Numerical Python) to fundamentalna, otwarto-źródłowa biblioteka języka Python służąca do obliczeń numerycznych, oferująca wydajne, wielowymiarowe obiekty tablicowe (*ndarray*) oraz szeroką gamę funkcji matematycznych, statystycznych i logicznych. Stanowi podstawę dla wielu innych bibliotek do analizy danych i uczenia maszynowego, takich jak Pandas, SciPy czy Scikit-learn.

Wiele obliczeń matematycznych wykorzystuje wektory, macierze i inne tablice liczb. Na pierwszy rzut oka listy i krotki w Pythonie wyglądają jak wektory, ale nie da się na nich wykonać np. „dodawania" takiego jakbyśmy chcieli w przypadku wektorów. Dlatego potrzebujemy typu obiektu, który jest tablicą liczb tego samego typu, którą można manipulować jak wektorem lub macierzą. W języku Python nie ma odpowiedniej funkcji do tego celu, ale Python umożliwia dodanie funkcji za pomocą modułów i pakietów, a jednym z nich jest moduł *numpy*: zapewniający odpowiednie

tablice liczbowe poprzez obiekty typu *ndarray* i udostępnia narzędzia do pracy z nimi, takie jak funkcja *array()* do tworzenia tablic z list lub krotek. (NumPy udostępnia również duży zbiór innych narzędzi do obliczeń numerycznych, jak zobaczymy później).

3.1. Instalacja pakietu numpy

Dwa główne narzędzia instalujące pakiety Pythona to *pip* i *conda*. Ich funkcjonalność częściowo się pokrywa (np. oba mogą zainstalować numpy), jednak mogą również działać razem. Omówimy tutaj główne różnice między pip a conda – jest to ważne, jeśli chcemy efektywnie zarządzać pakietami.

Pierwsza różnica polega na tym, że conda jest wielojęzykowa i może zainstalować Pythona, podczas gdy pip jest instalowany dla konkretnego Pythona w systemie i instaluje inne pakiety tylko w tej samej instalacji Pythona. Oznacza to również, że conda może instalować biblioteki i narzędzia inne niż Python, których możemy potrzebować (np. kompilatory, CUDA, HDF5), podczas gdy pip nie.

Druga różnica polega na tym, że pip instaluje z *Python Packaging Index* (PyPI), podczas gdy conda instaluje z własnych kanałów (zazwyczaj „defaults” lub „conda-forge”). PyPI to zdecydowanie największy zbiór pakietów, jednak wszystkie popularne pakiety są dostępne również dla conda.

Trzecią różnicą jest to, że conda to zintegrowane rozwiązanie do zarządzania pakietami, zależnościami i środowiskami, podczas gdy w przypadku Pip może być potrzebne inne narzędzie (jest ich wiele) do obsługi środowisk lub złożonych zależności.

- Conda: Jeśli używasz conda, możesz zainstalować numpy z domyślnych kanałów lub kanałów conda-forge:

```
conda create -n my-env
conda activate my-env
conda install numpy
```

Listing 1. Instalacja pakietu numpy za pomocą conda

- Pip:

```
pip install numpy
```

Listing 2. Instalacja pakietu numpy za pomocą pip

Po zainstalowaniu NumPy należy zweryfikować instalację, uruchamiając poniższe polecenie w powłoce Pythona lub skrypcie:

```
import numpy as np
print(np.__version__)
```

Listing 3. Weryfikacja instalacji pakietu numpy

3.2. Klasa ndarray

Głównym obiektem NumPy jest jednorodna tablica wielowymiarowa. Jest to tabela elementów (zazwyczaj liczb), wszystkie tego samego typu, indeksowana krotką nieujemnych liczb całkowitych. W NumPy wymiary nazywane są osiami. Na przykład, współrzędne punktu w przestrzeni trójwymiarowej [1, 2, 1] mają jedną oś. Oś ta zawiera 3 elementy, więc mówimy, że ma długość 3. W poniższym przykładzie tablica ma 2 osie. Pierwsza oś ma długość 2, a druga oś ma długość 3.

```
[[ 1., 0., 0.],
[ 0., 1., 2.]]
```

Listing 4. Przykład tablicy wielowymiarowej

Klasa tablicowa NumPy nazywa się *ndarray*. Znana jest również pod aliasem array. Należy pamiętać, że numpy.array to nie to samo, co klasa array.array ze standardowej biblioteki Pythona, która obsługuje tylko tablice jednowymiarowe i oferuje mniej funkcjonalności. Ważniejsze atrybuty obiektu ndarray to:

- *ndarray.ndim* – liczba osi (wymiarów) tablicy.
- *ndarray.shape* – wymiary tablicy. Jest to krotka liczb całkowitych wskazująca rozmiar tablicy w każdym wymiarze. Dla macierzy o n wierszach i m kolumnach, rozmiar będzie wynosić (n, m). Długość krotki równa się zatem liczbie osi, ndim.
- *ndarray.size* – całkowita liczba elementów tablicy. Jest ona równa iloczynowi rozmiarów tablicy.
- *ndarray.dtype* – obiekt opisujący typ elementów tablicy. Można tworzyć lub określać typy danych za pomocą standardowych typów Pythona. Dodatkowo NumPy udostępnia własne typy. Przykładami są numpy.int32, numpy.int16 inumpy.float64.
- *ndarray.itemsize* – rozmiar każdego elementu tablicy w bajtach. Na przykład tablica elementów typu float64 ma rozmiar 8 elementów (=64/8), a tablica typu complex32 ma rozmiar 4 elementów (=32/8). Jest to odpowiednik ndarray.dtype.itemsize.

Tworzenie tablic ndarray

Istnieje kilka sposobów tworzenia tablic. Na przykład, można utworzyć tablicę ze zwykłej listy lub krotki Pythona za pomocą funkcji *array*. Typ wynikowej tablicy jest wywnioskowywany z typu elementów w sekwencjach.

```
>>> import numpy as np
```

```
>>> a = np.array([2,3,4])
>>> a
    array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

Listing 5. Przykład tworzenia tablicy ndarray

Metoda array przekształca sekwencje sekwencji w tablice dwuwymiarowe, sekwencje sekwencji sekwencji w tablice trójwymiarowe itd.

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5, 2. , 3. ],
       [ 4. , 5. , 6. ]])
```

Listing 6. Tworzenia tablicy wielowymiarowej

Podstawowe operacje

Operatory arytmetyczne na tablicach działają element po elemencie. W wyniku tworzona jest nowa tablica z wynikiem operacji.

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
    array([0, 1, 2, 3])
>>> c = a-b
>>> c
    array([20, 29, 38, 47])
>>> b**2
    array([0, 1, 4, 9])
>>> 10*np.sin(a)
    array([ 9.12945251, -9.88031624, 7.4511316 , -2.62374854])
>>> a<35
    array([ True, True, False, False])
```

Listing 7. Przykład operacji arytmetycznych

W przeciwieństwie do wielu języków macierzowych, operator iloczynu * działa element po elemencie w tablicach NumPy. Iloczyn macierzowy można wykonać za pomocą operatora @ (w Pythonie >=3.5) lub metody *dot*.

```
>>> A = np.array( [[1,1],
... [0,1]] )
>>> B = np.array( [[2,0],
... [3,4]] )
>>> A * B # iloczyn element po elemncie
array([[2, 0],
[0, 4]])
>>> A @ B # iloczyn macierzowy
```

```
array([[5, 4],
[3, 4]])
>>> A.dot(B) # iloczyn
array([[5, 4],
[3, 4]])
```

Listing 8. Operatory * i @

Funkcje uniwersalne

NumPy udostępnia znane funkcje matematyczne, takie jak sin, cos i exp. W NumPy nazywane są one „funkcjami uniwersalnymi” (ufunc). W NumPy funkcje te operują na elementach tablicy, generując tablicę jako wynik.

```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([ 1. , 2.71828183, 7.3890561 ])
>>> np.sqrt(B)
array([ 0. , 1. , 1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([ 2., 0., 6.])
```

Listing 9. Przykłady funkcji universalnych

Indeksowanie, iterowanie i tworzenie wycinków

Tablice jednowymiarowe można indeksować, dzielić i iterować, podobnie jak listy i inne sekwencje języka Python. Tablice wielowymiarowe mogą mieć jeden indeks na oś. Indeksy te są podawane w krotce, rozdzielonej przecinkami:

```
>>> def f(x,y):
... return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0, 1, 2, 3],
[10, 11, 12, 13],
[20, 21, 22, 23],
[30, 31, 32, 33],
[40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1] # każdy wiersz w drugiej kolumnie b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1] # to samo co powyżej
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ] # każda kolumna w drugim i trzecim wierszu b
array([[10, 11, 12, 13],
```

```
[20, 21, 22, 23]])
```

Listing 10. Przykłady indeksowania i wycinków

Iterowanie po tablicach wielowymiarowych odbywa się względem pierwszej osi:

```
>>> for row in b:
... print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

Listing 11. Przykłady iteracji po tablicy wielowymiarowej

Jeśli jednak chcemy wykonać operację na każdym elemencie tablicy, możemy użyć atrybutu *flat*, który jest iteratorem po wszystkich elementach tablicy:

```
>>> for element in b.flat:
... print(element)
...
0
1
2
3
10
11
12
13
```

Listing 12. Przykłady iteracji po tablicy wielowymiarowej element po elemencie

Zmiana rozmiaru

Tablica ma rozmiar określony przez liczbę elementów wzdłuż każdej osi:

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a
array([[ 2., 8., 0., 6.],
[ 4., 5., 1., 1.],
[ 8., 9., 3., 6.]])
>>> a.shape
(3, 4)
```

Listing 13. Sprawdzenie rozmiaru tablicy

Rozmiary tablicy można zmienić za pomocą różnych poleceń. Należy pamiętać, że wszystkie trzy poniższe polecenia zwracają zmodyfikowaną tablicę, ale nie zmieniają oryginalnej tablicy:

```
>>> a.ravel() # zwraca tablice "spłaszczoną"
```

```
array([ 2., 8., 0., 6., 4., 5., 1., 1., 8., 9., 3., 6.])
>>> a.reshape(6,2) # zmienia wymiary tablicy
```

Listing 14. Zmiana wymiarów tablicy

Kopiowanie, referencje i widoki

Podczas obsługi i manipulowania tablicami ich dane są czasami kopiowane do nowej tablicy, a czasami nie. Często jest to źródłem nieporozumień dla początkujących. Istnieją trzy przypadki.

Proste przypisania nie tworzą kopii obiektów tablicowych ani ich danych.

```
>>> a = np.arange(12)
>>> b = a # nie tworzony jest nowy obiekt
>>> b is a # aa i b są referencjami do tego samego obiektu ndarray
True
>>> b.shape = 3,4 # zmienia również rozmiar a
>>> a.shape
(3, 4)
```

Listing 15. Referencje do tablic

Różne obiekty tablicowe mogą współdzielić te same dane. Metoda *view* tworzy nowy obiekt tablicowy, który analizuje te same dane.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a # c jest widokiem danych zawartych w tablicy a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = 2,6 # wymiary tablicy a nie ulegają zmianie
>>> a.shape
(3, 4)
>>> c[0,4] = 1234 # dane w tablicy a też zostają zmienione
>>> a
array([[ 0, 1, 2, 3],
[1234, 5, 6, 7],
[ 8, 9, 10, 11]])
```

Listing 16. Tworzenie widoków tablic

Tworzenie wycinków tablicy zwraca ich widok:

```
>>> s = a[ : ,1:3]
>>> s[:] = 10 # s[:] jest widokiem s.
>>> a
array([[ 0, 10, 10, 3],
[1234, 10, 10, 7],
```

```
[ 8, 10, 10, 11]])
```

Listing 17. Tworzenie widoków przez wycinki

Kopiowanie

Metoda *copy* tworzy kompletną kopię tablicy i jej danych

```
>>> d = a.copy() # nowy obiekt tablicy z danymi jest tworzony
>>> d is a
False
>>> d.base is a # d nie ma nic wspólnego z a
False
>>> d[0,0] = 9999
>>> a
array([[ 0, 10, 10, 3],
[1234, 10, 10, 7],
[ 8, 10, 10, 11]])
```

Listing 18. Tworzenie kopii tablicy

Czasami po wycięciu należy wywołać funkcję *copy*, jeśli oryginalna tablica nie jest już potrzebna. Na przykład, załóżmy, że *a* jest ogromnym wynikiem pośrednim, a wynik końcowy *b* zawiera tylko niewielką część *a*, podczas konstruowania *b* za pomocą wycięcia należy wykonać kopię:

```
>>> a = np.arange(int(1e8))
>>> b = a[:100].copy()
>>> del a # pamięć ``a`` może być zwolniona.
```

Listing 19. Tworzenie kopii z wycinka

Jeśli zamiast tego użyto by *b = a[:100]*, *b* jest widokiem *a* i będzie przechowywane w pamięci nawet po wykonaniu polecenia *del a*.

Przegląd funkcji i metod

value

To właściwość obiektu *DataFrame* modułu *Panda*s (o którym będzie mowa później). Zwraca ona bezpośrednio dwuwymiarową tablicę NumPy zawierającą wartości z DataFrame.

```
>>> import pandas as pd

>>> data = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
>>> df = pd.DataFrame(data)

>>> arr = df.values

>>> print(arr)  # [[1 4]
```

```
                # [2 5]
                # [3 6]]
```

Listing 20. Tworzenie tablicy z ramki DataFrame

zeros(size)

Tworzy tablicę wypełnioną zerami.

```
>>> np.zeros(5) # [0. 0. 0. 0. 0.]
```

Listing 21. Tworzenie tablicy wypełnionej zerami

ones(size)

Tworzy tablicę wypełnioną jedynkami.

```
>>> np.ones(5) # [1. 1. 1. 1. 1.]
```

Listing 22. Tworzenie tablicy wypełnionej jedynkami

eye(size)

Tworzy macierz jednostkową.

```
>>> np.eye(3) # [[1. 0. 0.]
              #  [0. 1. 0.]
              #  [0. 0. 1.]]
```

Listing 23. Tworzenie macierzy jednostkowej

arange(start, stop, step)

Tworzy tablicę z równomiernie rozmieszczonymi wartościami w określonym zakresie.

```
>>> np.arange(5, 10, 2) #[5 7 9]
```

Listing 24. Tworzenie z równomiernie rozmieszczonymi wartościami

linspace(start, stop, count)

Tworzy tablicę zawierającą określoną liczbę wartości równomiernie rozmieszczonych pomiędzy wartością początkową i końcową.

```
>>> np.linspace(0, 5, 5) #[0. 1.25 2.5 3.75 5. ]
```

Listing 25. Tworzenie tablicy zawierającą określoną liczbę wartości

random.randint(low, high, size)

Tworzy tablicę wypełnioną losowymi liczbami całkowitymi z określonego zakresu od najniższego do najwyższego.

```
>>> np.random.randint(1, 7, size=(2, 3)) # [[4 3 5]
                                         # [2 6 3]]
```

Listing 26. Tworzy tablicę wypełnioną losowymi liczbami całkowitymi

random.random(size)

Tworzy tablicę wypełnioną losowymi liczbami zmiennoprzecinkowymi z zakresu od 0 do 1.

```
>>> np.random.random(3) # [0.84957714 0.7538379 0.42278136]
```

Listing 27. Tworzy tablicę wypełnioną losowymi liczbami z zakresu 0-1

unique(<np-array>, <axis>)

Zwraca unikalne elementy tablicy.

```
>>> arr = np.array([1, 2, 2, 3, 4, 4, 5])
>>> print(np.unique(arr))  # [1 2 3 4 5]
```

Listing 28. Unikalne elementy tablicy

argmax(<np-array>, <axis>)

Znajduje indeksy wartości maksymalnych wzdłuż określonej osi w tablicy.

axis: liczba całkowita określająca oś, wzdłuż której ma zostać znalezione maksimum. 0 dla wierszy, 1 dla kolumn itd.

Zwraca tablicę indeksów o tym samym kształcie co tablica wejściowa, z usuniętym wymiarem wzdłuż określonej osi..

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

>>> # Znajduje indeks elementu max ze wszystkich osi
>>> print(np.argmax(arr))  # 5 (indeks 6 w "spłaszczonej" tablicy)
>>> print(np.argmax(arr, axis=0))  # [0 1 1] (indeksy wartości
maksymalnych w kolumnach)
>>> print(np.argmax(arr, axis=1))  # [2 2] (indeksy wartości
maksymalnych w wierszach)
```

Listing 29. Indeksy wartości maksymalnych

argmin(<np-array>, <axis>)

Znajduje indeksy wartości minimalnych wzdłuż określonej osi w tablicy.

where(<condition>, <true-return- value>, <false-return-value>)

Wybiera elementy z tablicy na podstawie zadanego warunku.

condition: tablica wartości logicznych o tym samym kształcie co tablica wejściowa.
true-return- value: wartość zwracana dla elementów, dla których warunek jest

spełniony, *false-return-value*: wartość zwracana dla elementów, dla których warunek jest nie spełniony.

Zwraca tablicę o tym samym kształcie co tablica wejściowa, w której każdy element jest zastępowany na podstawie warunku.

```
>>> print(np.where(arr > 3, arr * 2, arr / 2))  # zamienia elementy, dla których warunek jest prawdziwy, na dwukrotnie większy, a jeśli fałszywy na dwukrotnie mniejszy
```

Listing 30. Wybór elementu z tablicy na podstawie zadanego warunku

3.3. Przykładowe ćwiczenia

Ćwiczenie 3.1. Konwertuj tablicę 1D na tablicę 2D z 2 wierszami.

Ćwiczenie 3.2. Pomnóż macierz 5x3 przez macierz 3x2.

Ćwiczenie 3.3. Wyodrębnij wszystkie liczby nieparzyste z tablicy 1-10.

Ćwiczenie 3.4. Zamień wszystkie liczby nieparzyste w tablicy 1-10 na -1.

Ćwiczenie 3.5. Konwertuje tablicę 1D na tablicę wartości logicznych, w której wszystkie wartości dodatnie stają się wartościami True.

Ćwiczenie 3.6. Zastąp wszystkie liczby parzyste w tablicy 1D liczbami ujemnymi.

Ćwiczenie 3.7. Utwórz losową macierz 3x3 i znormalizuj ją.

Ćwiczenie 3.8. Oblicz sumę elementów diagonalnych macierzy 3x3.

Ćwiczenie 3.9. Znajdź indeksy elementów niezerowych z [1,2,0,0,4,0].

Ćwiczenie 3.10. Odwróć tablicę 1D (pierwszy element staje się ostatnim).

Ćwiczenie 3.11. Utwórz macierz jednostkową 3x3.

Ćwiczenie 3.12. Przekształć tablicę 1D w tablicę 2D z 5 wierszami i 2 kolumnami.

Ćwiczenie 3.13. Pobierz wspólne elementy dwóch tablic.

4. Biblioteka Pandas

Biblioteka Pandas to darmowa biblioteka języka programowania Python, używana do analizy i przetwarzania danych w sposób tabelaryczny. Umożliwia łatwe wczytywanie, czyszczenie, manipulowanie i eksplorację zbiorów danych, oferując struktury takie jak *Series* i *DataFrame*, a także funkcje do wykonywania operacji matematycznych i statystycznych. Pandas jest powszechnie stosowana w nauce o danych i została stworzona przez Wesa McKinneya w 2008 roku.

Pandas doskonale nadaje się do analizy wielu różnych rodzajów danych:

- danych tabelarycznych z kolumnami o niejednorodnym typie, np. w tabeli SQL lub arkuszu kalkulacyjnym Excel,
- uporządkowanych i nieuporządkowanych (niekoniecznie o stałej częstotliwości) szeregów czasowych,
- danych macierzowych o dowolnym typie (jednorodnym lub niejednorodnym) z etykietami wierszy i kolumn,
- dowolnych innych form zbiorów danych obserwacyjnych/statystycznych. Dane nie muszą być w ogóle oznaczone, aby można je było umieścić w strukturze danych Pandas.

4.1. Instalacja pakietu pandas

- Instalacja za pomocą Minicondy

Dla użytkowników doświadczonych w Pythonie zalecanym sposobem instalacji Pandas jest użycie Minicondy. Miniconda pozwala na stworzenie minimalnej, autonomicznej instalacji Pythona oraz użycie menedżera pakietów Conda do zainstalowania dodatkowych pakietów i utworzenia wirtualnego środowiska dla instalacji. Następnym krokiem jest utworzenie nowego środowiska Conda. Środowisko Conda jest podobne do środowiska wirtualnego, które pozwala na określenie konkretnej wersji Pythona i zestawu bibliotek:

```
conda create -c conda-forge -n name_of_my_env python pandas
```

Listing 31. Instalacja pakietu pandas za pomocą conda

- Instalacja z poziomu PyPI

Pandas można również zainstalować za pomocą pip z poziomu PyPI.

```
pip install pandas
```

Listing 32. Instalacja pakietu pandas z pomocą pip

Zwyczajowo moduł pandas importujemy w następujący sposób:

```
import pandas as pd
```

Listing 33. Importowanie modułu pandas

4.2. Podstawowe struktury danych w Pandas

Dwie podstawowe struktury danych pandas, *Series* (jednowymiarowa seria danych) i *DataFrame* (dwuwymiarowa ramka danych), sprawdzają się w większości typowych przypadków użycia w finansach, statystyce, naukach społecznych i wielu dziedzinach inżynierii. Najlepszym sposobem na postrzeganie struktur danych Pandas jest traktowanie ich jako elastycznych kontenerów dla danych o niższym wymiarze. Na przykład DataFrame to kontener dla serii, a seria to kontener dla skalarów.

Wszystkie struktury danych Pandas są zmienne pod względem wartości (wartości, które zawierają, można modyfikować), ale nie zawsze są zmienne pod względem rozmiaru. Długość serii nie może być zmieniona, ale na przykład kolumny można wstawiać do ramki danych. Jednak zdecydowana większość metod generuje nowe obiekty i pozostawia dane wejściowe nienaruszone.

Tworzenie serii i ramki danych

Serie możemy tworzyć poprzez przekazanie listy wartości, co pozwala pandasowi utworzyć domyślny indeks:

```
>>> s = pd.Series([1, 3, 5, np.nan, 6, 8])
```

Listing 34. Tworzenie serii z domyślnym indeksem

Poniższy przykład przedstawia tworzenie obiektu DataFrame poprzez przekazanie tablicy NumPy z indeksem *datetime* za pomocą *date_range()* i oznaczonymi kolumnami:

```
>>> dates = pd.date_range("20130101", periods=6)

>>> dates

DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

>>> df = pd.DataFrame(np.random.randn(6, 4), index=dates,
columns=list("ABCD"))

>>> df

                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
```


Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki” nr FERS.01.05-IP.08-0234/23

```
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

Listing 35. Tworzenie ramki danych z indeksem *datetime*

Inny sposób to tworzenie ramki danych poprzez przekazanie słownika obiektów, w którym kluczami są etykiety kolumn, a wartościami są wartości kolumn:

```
>>> df2 = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20130102"),
        "C": pd.Series(1, index=list(range(4)), dtype="float32"),
        "D": np.array([3] * 4, dtype="int32"),
        "E": pd.Categorical(["test", "train", "test", "train"]),
        "F": "foo",
    }

)

>>> df2
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

Listing 36. Tworzenie ramki danych ze słownika

Przeglądanie danych

Aby przeglądać dane możemy użyć metod *DataFrame.head()* i *DataFrame.tail()*, aby wyświetlić odpowiednio górny i dolny wiersz ramki:

```
>>> df.head()
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401

>>> df.tail(3)
                   A         B         C         D
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

Listing 37. Wyświetlanie początkowych i końcowych wierszy ramki danych

Kolejny przykład przedstawia wyświetlanie indeksów i nazw kolumn za pomocą metod *DataFrame.index* or *DataFrame.columns*:

```
>>> df.index
 DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
                '2013-01-05', '2013-01-06'],
               dtype='datetime64[ns]', freq='D')
>>> df.columns
 Index(['A', 'B', 'C', 'D'], dtype='object')
```

Listing 38. Wyświetlanie indeksów i nazw kolumn

Metoda *describe()* pozwala na szybkie podsumowanie statystyk danych:

```
>>> s = df.describe()

              A         B         C         D
count  6.000000  6.000000  6.000000  6.000000
mean   0.073711 -0.431125 -0.687758 -0.233103
std    0.843157  0.922818  0.779887  0.973118
min   -0.861849 -2.104569 -1.509059 -1.135632
25%   -0.611510 -0.600794 -1.368714 -1.076610
50%    0.022070 -0.228039 -0.767252 -0.386188
75%    0.658444  0.041933 -0.034326  0.461706
max    1.212112  0.567020  0.276232  1.071804
```

Listing 39. Tworzenie szybkich statystyk

Do transponowania danych służy metoda *T*:

```
>>> df.T

   2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-
06
A    0.469112    1.212112   -0.861849    0.721555   -0.424972
-0.673690
B   -0.282863   -0.173215   -2.104569   -0.706771    0.567020
0.113648
C   -1.509059    0.119209   -0.494929   -1.039575    0.276232
-1.478427
D   -1.135632   -1.044236    1.071804    0.271860   -1.087401
0.524988
```

Listing 40. Transponowanie danych

Metoda *DataFrame.sort_index()* sortuje według wybranej osi:

```
>>> df.sort_index(axis=1, ascending=False)

                   D         C         B         A
2013-01-01 -1.135632 -1.509059 -0.282863  0.469112
2013-01-02 -1.044236  0.119209 -0.173215  1.212112
2013-01-03  1.071804 -0.494929 -2.104569 -0.861849
```

```
2013-01-04  0.271860 -1.039575 -0.706771  0.721555
2013-01-05 -1.087401  0.276232  0.567020 -0.424972
2013-01-06  0.524988 -1.478427  0.113648 -0.673690
```

Listing 41. Sortowanie według osi

Meoda *DataFrame.sort_values()* sortuje według wybranej kolumny:

```
>>> df.sort_values(by="B")

                   A         B         C         D
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
```

Listing 42. Sortowanie według kolumny

Wybór danych

W przypadku DataFrame przekazanie pojedynczej etykiety powoduje wybranie kolumny i wygenerowanie serii odpowiadającej *df.A*:

```
>>> df["A"]
Out[24]:
2013-01-01    0.469112
2013-01-02    1.212112
2013-01-03   -0.861849
2013-01-04    0.721555
2013-01-05   -0.424972
2013-01-06   -0.673690
Freq: D, Name: A, dtype: float64
```

Listing 43. Wybór kolumny

W przypadku obiektu DataFrame przekazanie wycinka powoduje wybranie odpowiadających wierszy:

```
>>> df[0:3]
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804

>>> df["20130102":"20130104"]
                   A         B         C         D
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
```

Listing 44. Wybór wierszy

Za pomocą metody *loc* możemy wybierać wiersze odpowiadające etykiecie:

```
>>> df.loc[dates[0]]

A    0.469112
B   -0.282863
C   -1.509059
D   -1.135632
```

Listing 45. Wybór wierszy przez etykiety

Aby wybrać dane z wybranych kolumny możemy użyć wycinka : i nazw kolumn:

```
>>> df.loc[:, ["A", "B"]]
                   A         B
2013-01-01  0.469112 -0.282863
2013-01-02  1.212112 -0.173215
2013-01-03 -0.861849 -2.104569
2013-01-04  0.721555 -0.706771
2013-01-05 -0.424972  0.567020
2013-01-06 -0.673690  0.113648
```

Listing 46. Wybór wierszy dla danych kolumn

Dane wiersze możemy wybierać również poprzez pozycje za pomocą metody *iloc*:

```
>>> df.iloc[3]
A    0.721555
B   -0.706771
C   -1.039575
D    0.271860
Name: 2013-01-04 00:00:00, dtype: float64
```

Listing 47. Wybór wierszy przez pozycje

Dane możemy również wybierać z użyciem warunków. Poniższy przykład przedstawia wybór wierszy, w których w kolumnie A znajdują się wartości większe 0:

```
>>> df[df["A"] > 0]
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
```

Listing 48. Wybór wierszy spełniających warunek

Dodanie nowej kolumny automatycznie przypisuje dane według indeksów:

```
>>> s1 = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range("20130102",
periods=6))

>>> s1
2013-01-02    1
2013-01-03    2
```

```
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64

df["F"] = s1
```

Listing 49. Dodanie nowej kolumny

Zmiana wartości przez podanie etykiet elementu realizowane jest za pomocą metody *at* i *iat*:

```
>>> df.at[dates[0], "A"] = 0
>>> df.iat[0, 1] = 0
```

Listing 50. Zmiana wartości elementu

Zmiana wartości może być realizowana również na podstawie warunku:

```
>>> df2 = df.copy()

>>> df2[df2 > 0] = -df2

>>> df2
                   A         B         C    D    F
2013-01-01  0.000000  0.000000 -1.509059 -5.0  NaN
2013-01-02 -1.212112 -0.173215 -0.119209 -5.0 -1.0
2013-01-03 -0.861849 -2.104569 -0.494929 -5.0 -2.0
2013-01-04 -0.721555 -0.706771 -1.039575 -5.0 -3.0
2013-01-05 -0.424972 -0.567020 -0.276232 -5.0 -4.0
2013-01-06 -0.673690 -0.113648 -1.478427 -5.0 -5.0
```

Listing 51. Zmiana wartości elementu na podstawie warunku

Przykładowe operacje na danych

Obliczanie wartość średniej dla każdej kolumny:

```
>>> df.mean()
Out[61]:
A   -0.004474
B   -0.383981
C   -0.687758
D    5.000000
F    3.000000
dtype: float64
```

Listing 52. Wartość średnia dla kolumn

Obliczanie wartości średniej dla każdego wiersza:

```
>>> df.mean(axis=1)
```

```
Out[62]:
2013-01-01    0.872735
2013-01-02    1.431621
2013-01-03    0.707731
2013-01-04    1.395042
2013-01-05    1.883656
2013-01-06    1.592306
Freq: D, dtype: float64
```

Listing 53. Wartość średnia dla wierszy

Metody *DataFrame.agg()* i *DataFrame.transform()* pozwalają na zastosowanie zdefiniowanej przez użytkownika funkcji, które przetwarzają dane:

```
>>> df.agg(lambda x: np.mean(x) * 5.6)

A    -0.025054
B    -2.150294
C    -3.851445
D    28.000000
F    16.800000
dtype: float64

>>> df.transform(lambda x: x * 101.2)

                    A           B           C      D      F
2013-01-01    0.000000    0.000000 -152.716721  506.0    NaN
2013-01-02  122.665737  -17.529322   12.063922  506.0  101.2
2013-01-03  -87.219115 -212.982405  -50.086843  506.0  202.4
2013-01-04   73.021382  -71.525239 -105.204988  506.0  303.6
2013-01-05  -43.007200   57.382459   27.954680  506.0  404.8
2013-01-06  -68.177398   11.501219 -149.616767  506.0  506.0
```

Listing 54. Wartość średnia dla wierszy

Zliczanie wartości

Metoda *value_counts()* pozwala na zliczanie wartości w serii danych:

```
>>> s = pd.Series(np.random.randint(0, 7, size=10))

s

0    4
1    2
2    1
3    2
4    6
5    4
6    4
7    6
8    4
```

```
9    4
dtype: int64

>>> s.value_counts()

4    5
2    2
6    2
1    1
Name: count, dtype: int64
```

Listing 55. Zliczanie wartości w serii

Operacje na ciągach znaków

Serie są wyposażone w zestaw metod przetwarzania ciągów znaków, które ułatwiają wykonywanie operacji na każdym elemencie tablicy, jak w poniższym fragmencie kodu:

```
>>> s = pd.Series(["A", "B", "C", "Aaba", "Baca", np.nan, "CABA", "dog",
"cat"])
s.str.lower()
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

Listing 56. Metoda lower obiektu str

Łączenie danych

Pandas udostępnia szereg funkcji umożliwiających łatwe łączenie obiektów Series i DataFrame z różnymi rodzajami zbiorów poprzez operacje łączenia/scalania. Przykład przedstawia łączenie obiektów pandas wierszowo za pomocą metody *concat()*:

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df
          0         1         2         3
0 -0.548702  1.467327 -1.015962 -0.483075
1  1.637550 -1.217659 -0.291519 -1.745505
2 -0.263952  0.991460 -0.919069  0.266046
3 -0.709661  1.669052  1.037882 -1.705775
```


Politechnika Świętokrzyska
Kielce University of Technology
Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki” nr FERS.01.05-IP.08-0234/23

```
4 -0.919854 -0.042379  1.247642 -0.009920
5  0.290213  0.495767  0.362949  1.548106
6 -1.131345 -0.089329  0.337863 -0.945867
7 -0.932132  1.956030  0.017587 -0.016692
8 -0.575247  0.254161 -1.143704  0.215897
9  1.193555 -0.077118 -0.408530 -0.862495

# podział ramki df na trzy części

pieces = [df[:3], df[3:7], df[7:]]

pd.concat(pieces)

          0         1         2         3
0 -0.548702  1.467327 -1.015962 -0.483075
1  1.637550 -1.217659 -0.291519 -1.745505
2 -0.263952  0.991460 -0.919069  0.266046
3 -0.709661  1.669052  1.037882 -1.705775
4 -0.919854 -0.042379  1.247642 -0.009920
5  0.290213  0.495767  0.362949  1.548106
6 -1.131345 -0.089329  0.337863 -0.945867
7 -0.932132  1.956030  0.017587 -0.016692
8 -0.575247  0.254161 -1.143704  0.215897
9  1.193555 -0.077118 -0.408530 -0.862495
```

Listing 57. Metoda concat

Uwaga

Dodanie kolumny do obiektu DataFrame jest stosunkowo szybkie. Jednak dodanie wiersza wymaga kopii i może być kosztowne. Zalecane jest przekazanie predefiniowanej listy rekordów do konstruktora obiektu DataFrame zamiast budowania obiektu DataFrame poprzez iteracyjne dołączanie do niego rekordów.

Metoda *merge* umożliwia łączenie danych w stylu SQL wzdłuż określonych kolumn.

```
>>> left = pd.DataFrame({"key": ["foo", "foo"], "lval": [1, 2]})

>>> right = pd.DataFrame({"key": ["foo", "foo"], "rval": [4, 5]})

>>> left

   key  lval
0  foo     1
1  foo     2

>>> right

   key  rval
0  foo     4
```

```
1  foo     5

>>> pd.merge(left, right, on="key")

   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

Listing 58. Metoda merge

Grupowanie

Przez „grupowanie według” rozumiemy proces obejmujący jeden lub więcej z następujących kroków:

- podział danych na grupy na podstawie określonych kryteriów,
- zastosowanie funkcji do każdej grupy niezależnie,
- połączenie wyników w strukturę danych.

Przykład przedstawia grupowanie według etykiety kolumny, wybieranie etykiet kolumn, a następnie stosowanie funkcji *sum()* do powstałych grup:

```
>>> df = pd.DataFrame(

    {

        "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],

        "B": ["one", "one", "two", "three", "two", "two", "one",
"three"],

        "C": np.random.randn(8),

        "D": np.random.randn(8),

    }

)
>>> df

     A      B         C         D
0  foo    one  1.346061 -1.577585
1  bar    one  1.511763  0.396823
2  foo    two  1.627081 -0.105381
3  bar  three -0.990582 -0.532532
4  foo    two -0.441652  1.453749
```

```
5  bar    two  1.211526  1.208843
6  foo    one  0.268520 -0.080952
7  foo  three  0.024580 -0.264610

>>> df.groupby("A")[["C", "D"]].sum()

            C         D
A
bar  1.732707  1.073134
foo  2.824590 -0.574779
```

Listing 59. Metoda *groupby*

Grupowanie według etykiet wielu kolumn tworzy podgrupy:

```
>>> df.groupby(["A", "B"]).sum()

                  C         D
A   B
bar one    1.511763  0.396823
    three -0.990582 -0.532532
    two    1.211526  1.208843
foo one    1.614581 -1.658537
    three  0.024580 -0.264610
    two    1.185429  1.348368
```

Listing 60. Metoda *groupby* dla wielu kolumn

4.3. Przykładowe ćwiczenia

Ćwiczenie 4.1. Napisz program w Pandas, aby utworzyć ramkę danych ze słownika i ją wyświetlić. Przykładowe dane: {'X':[78,85,96,80,86], 'Y':[84,94,89,83,86],'Z': [86,97,96,72,83]}

Oczekiwany wynik:

```
>>> df
    X   Y   Z
0  78  84  86
1  85  94  97
2  96  89  96
3  80  83  72
4  86  86  83
```

Ćwiczenie 4.2. Napisz program w Pandas, który utworzy i wyświetli ramkę danych z określonych danych słownikowych, które zawierają etykiety indeksów.

Przykładowe dane słownikowe Pythona i etykiety list:

exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],

'score': [12,5, 9, 16,5, np.nan, 9, 20, 14,5, np.nan, 8, 19],

'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],

'qualify': ['yes', 'no', 'yes', 'no', 'yes', 'yes', 'no', 'yes', 'yes', 'no', 'no', 'tak']}

etykiety = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

Oczekiwany wynik:

```
>>> df
    attempts       name qualify  score
a          1  Anastasia     yes   12.5
b          3       Dima      no    9.0
....
i          2      Kevin      no    8.0
j          1      Jonas     yes   19.0
```

Ćwiczenie 4.3. Napisz program Pandas, który wybierze określone kolumny i wiersze z podanej ramki danych.

Przykładowe dane słownikowe i etykiety listy w Pythonie:

Wybierz kolumny „name” i „score” w wierszach 1, 3, 5, 6 z poniższej ramki danych.

exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],

'score': [12,5, 9, 16,5, np.nan, 9, 20, 14,5, np.nan, 8, 19],

'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],

'qualify': ['tak', 'nie', 'tak', 'nie', 'nie', 'tak', 'tak', 'nie', 'nie', 'tak']}

labels = ['a', [b], [c], [d], [e], [f], [g], [h], [i], [j]

Oczekiwany wynik:

```
b    9.0      no
d    NaN      no
f   20.0     yes
g   14.5     yes
```

Ćwiczenie 4.4. Napisz program Pandas, który będzie wybierał wiersze, w których liczba podejść do egzaminu jest większa niż 2.

Przykładowe dane słownikowe Pythona i etykiety list:

exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],

'score': [12,5, 9, 16,5, np.nan, 9, 20, 14,5, np.nan, 8, 19],

'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],

'qualify': ['yes', 'no', 'yes', 'no', 'yes', 'yes', 'no', 'yes', 'yes', 'no', 'no', 'tak']}

etykiety = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

Oczekiwany wynik:

```
   name  score  attempts qualify
b     Dima    9.0         3      no
d    James    NaN         3      no
f  Michael   20.0         3     yes
```

Ćwiczenie 4.5. Napisz program Pandas, który doda nowy wiersz „k” do ramki danych z podanymi wartościami dla każdej kolumny. Następnie usuń nowy wiersz i zwróć oryginalną ramkę danych.

Przykładowe dane słownikowe Pythona i etykiety list:

exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],

'score': [12,5, 9, 16,5, np.nan, 9, 20, 14,5, np.nan, 8, 19],

'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],

'qualify': ['yes', 'no', 'yes', 'no', 'yes', 'yes', 'no', 'yes', 'yes', 'no', 'no', 'tak']}

etykiety = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

Wartości dla każdej kolumny będą następujące:

imię: „Suresh”, wynik: 15,5, próby: 1, kwalifikacja: „tak”, etykieta: „k”

Oczekiwany wynik:

```
  attempts       name qualify  score
```

```
a         1  Anastasia     yes   12.5
b         3       Dima      no    9.0
......
j         1      Jonas     yes   19.0
k         1     Suresh     yes   15.5

Delete the new row and display the original  rows:
   attempts       name qualify  score
a         1  Anastasia     yes   12.5
b         3       Dima      no    9.0
........
i         2      Kevin      no    8.0
j         1      Jonas     yes   19.0
```

Ćwiczenie 4.6. Napisz program Pandas, który sortuje DataFrame najpierw według „nazwy” w kolejności malejącej, a następnie według „wyniku” w kolejności rosnącej.

Przykładowe dane słownikowe Pythona i etykiety list:

exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],

'score': [12,5, 9, 16,5, np.nan, 9, 20, 14,5, np.nan, 8, 19],

'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],

'qualify': ['yes', 'no', 'yes', 'no', 'yes', 'yes', 'no', 'yes', 'yes', 'no', 'no', 'tak']}

etykiety = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

Wartości dla każdej kolumny będą następujące:

imię: „Suresh”, wynik: 15,5, próby: 1, kwalifikacja: „tak”, etykieta: „k”

Oczekiwany wynik:

```
#Dane oryginalne
        name  score  attempts qualify
a  Anastasia   12.5         1     yes
b       Dima    9.0         3      no
c  Katherine   16.5         2     yes
d      James    NaN         3      no
e      Emily    9.0         2      no
f    Michael   20.0         3     yes
g    Matthew   14.5         1     yes
h      Laura    NaN         1      no
i      Kevin    8.0         2      no
j      Jonas   19.0         1     yes
```

```
#Dane posortowane
        name  score  attempts qualify
f    Michael   20.0         3     yes
g    Matthew   14.5         1     yes
h      Laura    NaN         1      no
i      Kevin    8.0         2      no
c  Katherine   16.5         2     yes
j      Jonas   19.0         1     yes
d      James    NaN         3      no
e      Emily    9.0         2      no
b       Dima    9.0         3      no
a  Anastasia   12.5         1     yes
```

Ćwiczenie 4.7. Napisz program Pandas, który będzie iterował po wierszach w DataFrame. Przykładowe dane słownikowe i etykiety list w Pythonie:

exam_data = [{'name':'Anastasia', 'score':12.5}, {'name':'Dima', 'score':9}, {'name':'Katherine', 'score':16.5}]

```
import pandas as pd
import numpy as np
exam_data = [{'name':'Anastasia', 'score':12.5},
{'name':'Dima','score':9}, {'name':'Katherine','score':16.5}]
df = pd.DataFrame(exam_data)
for index, row in df.iterrows():
    print(row['name'], row['score'])
```

Listing 61. Rozwiązanie ćw. 4.7.

Ćwiczenie 4.8. Napisz program Pandas, który będzie wybierał wiersze z danej ramki danych na podstawie wartości w niektórych kolumnach.

Przykładowe dane

```
col1  col2  col3
0     1     4     7
1     4     5     8
2     3     6     9
3     4     7     0
4     5     8     1
#Wiersze w których col2 == 4
   col1  col2  col3
1     4     5     8
3     4     7     0
```


Politechnika Świętokrzyska
Kielce University of Technology
Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki” nr FERS.01.05-IP.08-0234/23

5. Biblioteka Matplotlib

Matplotlib to kompleksowa biblioteka do tworzenia statycznych, animowanych i interaktywnych wizualizacji w Pythonie. Pakiet Matplotlib został stworzony przez Johna D. Huntera, jest dostępny jako oprogramowanie open source i można z niego korzystać swobodnie. Pakiet Matplotlib jest napisany głównie w Pythonie, a kilka segmentów w językach C, Objective-C i Javascript dla zapewnienia kompatybilności z innymi platformami.

5.1. Instalacja pakietu Matplotlib

Wersje Matplotlib są dostępne jako pakiety dla systemów macOS, Windows i Linux w PyPI. Można je zainstalować je za pomocą pip:

```
python -m pip install -U matplotlib
```

Biblioteka Matplotlib jest dostępna zarówno poprzez główny kanał Anacondy

```
conda install matplotlib
```

jak i poprzez kanał społecznościowy conda-forge

```
conda install -c conda-forge matplotlib
```

5.2. Podstawowe funkcje pakietu Matplotlib

W tym podpunkcie przedstawiono podstawowe wzorce użytkowania i praktyki, które pozwalają na rozpoczęcie pracy z biblioteką Matplotlib. Aby można było korzystać z modułów pakietu należy je zaimportować, zwyczajowo importujemy je do przestrzeni nazw *plt*:

```
import matplotlib.pyplot as plt
import numpy as np
```

Matplotlib tworzy wykresy danych na rysunkach (np. oknach, widżetach Jupyter itp.), z których każdy może zawierać jedną lub więcej osi – obszar, w którym punkty można określić za pomocą współrzędnych x-y (lub theta-r na wykresie biegunowym, x-y-z na wykresie 3D itp.). Najprostszym sposobem utworzenia rysunku z osiami jest użycie *pyplot.subplots*. Następnie możemy użyć *Axes.plot*, aby narysować dane na osiach i wyświetlić rysunek:

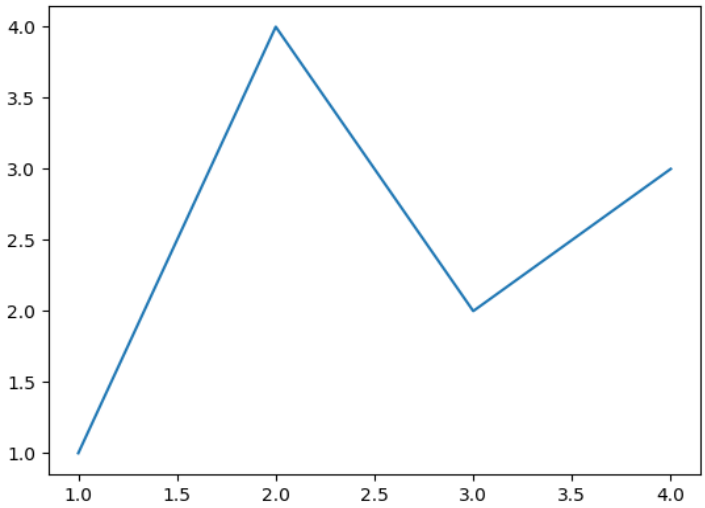
```
fig, ax = plt.subplots()             # Tworzenie okna wykresu
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])  # rysowanie wkresu
plt.show()                           # uwidacznianie okna wykresu
```

Listing 62. Przykład tworzenia okna z wykresem.

W zależności od środowiska pracy, *plt.show()* można pominąć. Dzieje się tak na przykład w notatnikach Jupyter, które automatycznie wyświetlają wszystkie rysunki

utworzone w programie. Efektem powinno być okno zawierające domyślnie sformatowany wykres:

[Tekst alternatywny. Przykład wykresu jednowymiarowego wykonanego za pomocą biblioteki Matplolib.]



Rys. 2. Przykład wykresu jednowymiarowego

Obiekt *figure*

Obiekt *figure* jest kontenerem na wszystkie podrzędne osie, grupę obiektów (tytuły, legendy wykresów, paski kolorów itp.), a nawet zagnieżdżonych pod-wykresów. Zazwyczaj nowy obiekt *figure* tworzy się za pomocą jednej z następujących funkcji:

```
fig = plt.figure()              # puste okno bez osi
fig, ax = plt.subplots()        # okno wykres z jedną osią
fig, axs = plt.subplots(2, 2)  # okno wykresu z czterema osiami
# okno z jedną osią po lewej I dwiema osiami po prawej:
fig, axs = plt.subplot_mosaic([['left', 'right_top'],
                               ['left', 'right_bottom']])
```

Listing 63. Inne spoby tworzenie okna z wykresem.

Metody s*ubplots()* i *subplot_mosaic* to funkcje pomocnicze, które dodatkowo tworzą obiekty osi wewnątrz okna, ale osie można dodać również ręcznie później.

Obiekt *axes*

Osie to obiekty typu „artist” dołączone do rysunku, które zawierają obszar do kreślenia danych i zazwyczaj zawierają dwa (lub trzy w przypadku obiektów 3D) obiekty osi (należy pamiętać o różnicy między obiektami „Axes” i „Axis”), które zapewniają znaczniki i etykiety znaczników, określające skalę danych na osiach.

Każda oś ma również tytuł (ustawiany za pomocą *set_title()*), etykietę x (ustawianą za pomocą *set_xlabel()*) oraz etykietę y ustawianą za pomocą *set_ylabel()*).

Metody osi to podstawowy interfejs do konfigurowania większości elementów wykresu (dodawanie danych, kontrolowanie skali i granic osi, dodawanie etykiet itp.).

Obiekt *axis*

Te obiekty ustawiają skalę i granice oraz generują znaczniki (znaczniki na osi) i etykiety znaczników (ciągi znaków oznaczające znaczniki). Położenie znaczników jest określane przez obiekt *Locator*, a ciągi *ticklabel* są formatowane przez *Formatter*. Połączenie prawidłowego *Locatora* i *Formattera* zapewnia bardzo precyzyjną kontrolę nad położeniem znaczników i etykietami.

Obiekt *artist*

Zasadniczo wszystko, co widać na rysunku, jest tzw. „artystą” (nawet rysunek, osie i obiekty osi). Dotyczy to obiektów tekstowych, obiektów Line2D, obiektów kolekcji, obiektów Patch itp. Po wyrenderowaniu rysunku wszyscy „artyści” są rysowani na płótnie. Większość „artystów” jest powiązana z osiami; takiego „artysty” nie można współdzielić z wieloma osiami ani przenosić między nimi.

Typy danych wejściowych dla funkcji kreślących

Funkcje kreślące oczekują danych wejściowych w postaci *numpy.array*, *numpy.ma.masked_array* albo obiektów, które można przekazać do *numpy.asarray*. Klasy podobne do tablic („tablicowe”), takie jak obiekty danych Pandas i *numpy.matrix*, mogą nie działać zgodnie z oczekiwaniami. Powszechną konwencją jest konwertowanie ich na obiekty *numpy.array* przed kreśleniem. Na przykład, aby przekonwertować *numpy.matrix* należy:

```
b = np.matrix([[1, 2], [3, 4]])
b_asarray = np.asarray(b)
```

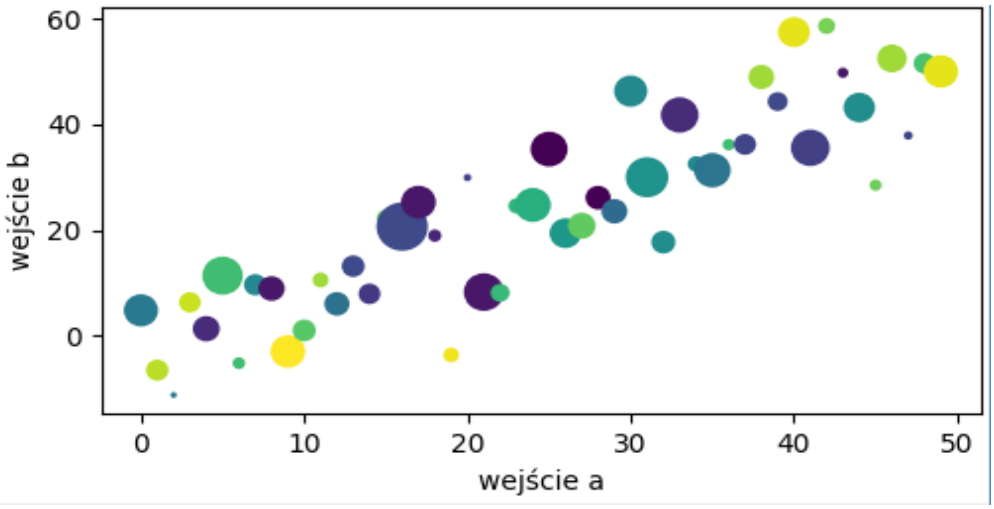
Większość metod analizuje również obiekty indeksowane ciągami znaków, takie jak słownik, ustrukturyzowana tablica numpy lub pandas.DataFrame. Biblioteka Matplotlib pozwala na podanie argumentu słowa kluczowego danych i generowanie wykresów przekazujących ciągi znaków odpowiadające zmiennym x i y.

```
np.random.seed(19680801)  # seed the random number generator.
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.scatter('a', 'b', c='c', s='d', data=data)
```

```
ax.set_xlabel('wejście a')
ax.set_ylabel('wejście b')
```

Listing 64. Przykład tworzenia wykresu punktowego.

[Tekst alternatywny. Przykład wykresu jednowymiarowego wykonanego za pomocą biblioteki Matplolib.]



Rys. 3. Przykład wykresu punktowego

Istnieją zasadniczo dwa sposoby korzystania z biblioteki Matplotlib:

- Jawne tworzenie figur i osi oraz wywoływanie na nich metod („styl obiektowy (OO)”).
- Poleganie na pyplot w celu niejawnego tworzenia i zarządzania oknami i osiami oraz używanie funkcji pyplot do tworzenia wykresów.

Można więc używać stylu OO:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2, 100)  # Dane osi x.
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.plot(x, x, label='linear')  # Wykreślanie danych.
ax.plot(x, x**2, label='quadratic')  # Kolejny wykres
ax.plot(x, x**3, label='cubic')  # ... i jeszcze jeden.
ax.set_xlabel('etykieta osi x')  # Tytuł osi x.
ax.set_ylabel('etykieta os y')  # Tytuł osi y.
ax.set_title("Przykładowy wykres")  # Tytuł wykresu.
ax.legend()  # Dodanie legendy.
fig.show()
```

Listing 65. Przykładad tworzenia wykresu w stylu obiektowym.

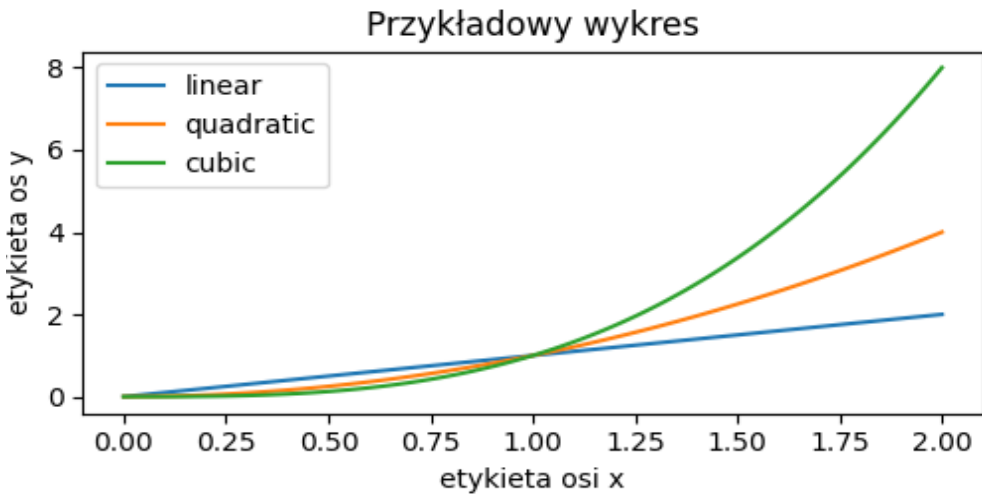
lub stylu pyplot:

```
x = np.linspace(0, 2, 100)  # Dane osi x.

plt.figure(figsize=(5, 2.7), layout='constrained')
plt.plot(x, x, label='linear')  # Wykreślanie danych.
plt.plot(x, x**2, label='quadratic')  # itd.
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Przykładowy wykres")
plt.legend()
```

Listing 66. Przykładad tworzenia wykresu w stylu pyplot.

Efekt jest dokładnie taki sam co przedstawia rysunek poniżej:

[Tekst alternatywny. Przykład wykresu z wieloma funkcjami wykonanego za pomocą biblioteki Matplolib.]



Rys. 4. Przykład wykresu z wieloma funkcjami

Formatowanie wykresu

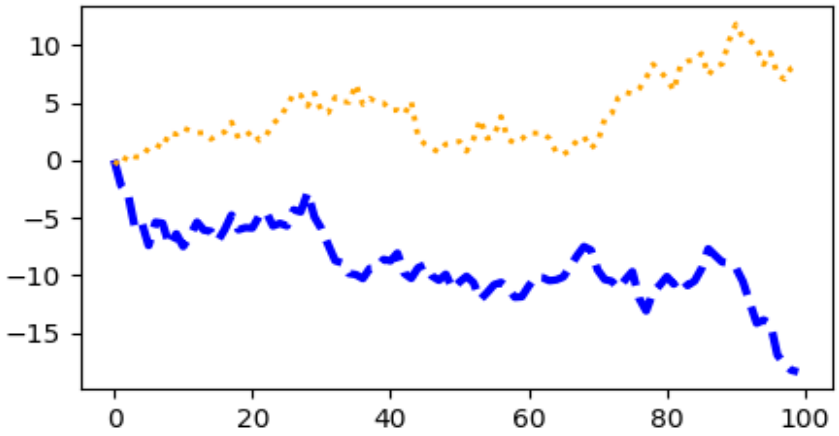
Większość metod kreślenia oferuje opcje formatowania elementów wykresu nazywanych „artystami”, dostępne po wywołaniu metody kreślenia lub za pomocą „settera” w obiekcie typu „artist”. Na poniższym wykresie ręcznie ustawiono kolor, szerokość linii i styl linii wykresów utworzonych przez kreślenie, a styl linii drugiego wiersza ustawiamy za pomocą *set_linestyle*.

```
import matplotlib.pyplot as plt
import numpy as np
data1, data2, data3, data4 = np.random.randn(4, 100)
```

```
fig, ax = plt.subplots(figsize=(5, 2.7))
x = np.arange(len(data1))
ax.plot(x, np.cumsum(data1), color='blue', linewidth=3, linestyle='--')
l, = ax.plot(x, np.cumsum(data2), color='orange', linewidth=2)
l.set_linestyle(':')
```

Listing 67. Przykład formatowania wykresu.

[Tekst alternatywny. Przykład formatowania wykresu wykonanego za pomocą biblioteki Matplolib.]



Rys. 5. Przykład formatowania wykresu

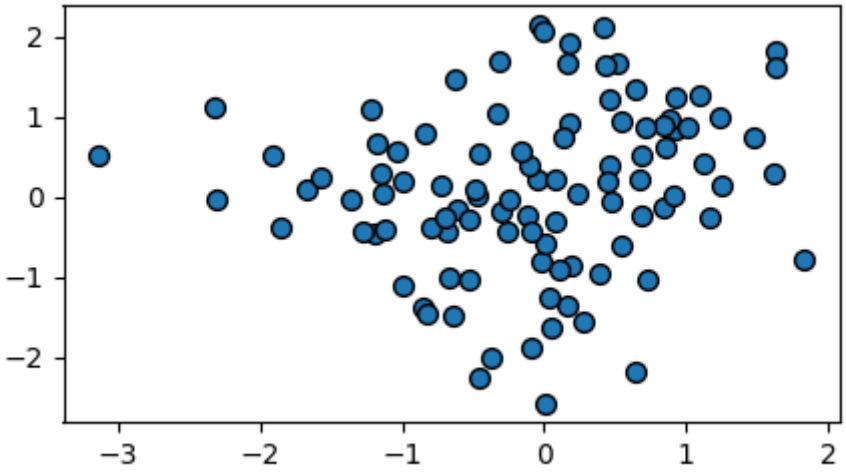
Kolory

Biblioteka Matplotlib oferuje bardzo elastyczną paletę kolorów, akceptowaną przez większość grafików; listę specyfikacji można znaleźć w definicjach dopuszczalnych kolorów. Niektóre elementy wykresu używają wielu kolorów. Np. w przypadku wykresu punktowego krawędzie znaczników mogą mieć inny kolor niż wnętrze:

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.scatter(data1, data2, s=50, facecolor='C0', edgecolor='k')
```

Listing 68. Przykład definicji koloru wykresu.

[Tekst alternatywny. Przykład zmiany kolorów wykresu wykonanego za pomocą biblioteki Matplolib.]



Rys. 6. Przykład zmiany koloru wykresu

Szerokości linii, style linii i rozmiary znaczników

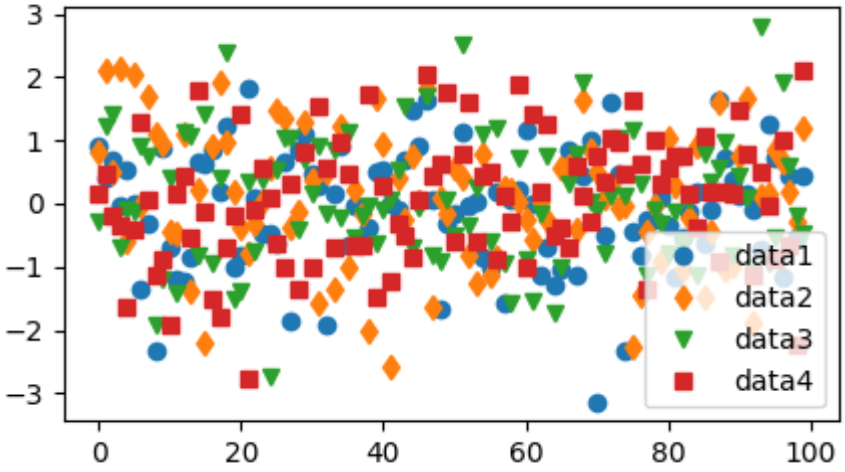
Szerokości linii są zazwyczaj podawane w punktach typograficznych (1 pt = 1/72 cala) i są dostępne dla elementów wykresu, które używają linii obrysowanych. Podobnie linie obrysowane może mieć styl linii.

Rozmiar znacznika zależy od używanej metody. Metoda *plot* określa rozmiar znacznika w punktach i zazwyczaj jest to „średnica” lub szerokość znacznika. *Scatter* określa rozmiar znacznika jako w przybliżeniu proporcjonalny do obszaru wizualnego znacznika. Dostępna jest tablica stylów znaczników w postaci kodów ciągów, a użytkownicy mogą również zdefiniować własny *MarkerStyle*:

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(data1, 'o', label='data1')
ax.plot(data2, 'd', label='data2')
ax.plot(data3, 'v', label='data3')
ax.plot(data4, 's', label='data4')
ax.legend()
```

Listing 69. Przykład definicji znaczników osi.

[Tekst alternatywny. Przykład definicji znaczników osi wykresu wykonanego za pomocą biblioteki Matplolib.]



Rys. 7. Przykład definicji znaczników osi

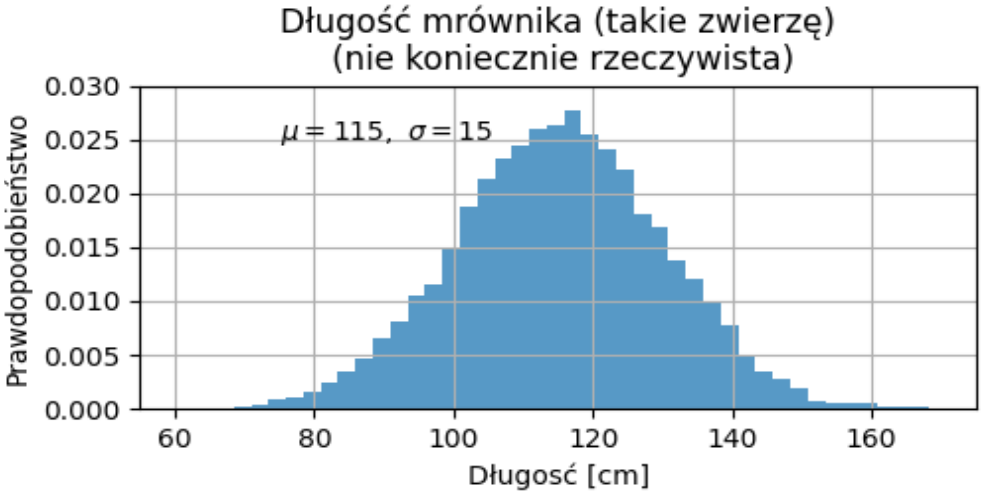
Etykiety osi i tekst

Metody *set_xlabel, set_ylabel* i *set_title* służą do dodawania tekstu we wskazanych miejscach. Tekst można również dodawać bezpośrednio do wykresów za pomocą metody *text*:

```
mu, sigma = 115, 15
x = mu + sigma * np.random.randn(10000)
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
# dane do histogramu
n, bins, patches = ax.hist(x, 50, density=True, facecolor='C0',
alpha=0.75)

ax.set_xlabel('Długosć [cm]')
ax.set_ylabel('Prawdopodobieństwo')
ax.set_title('Długość mrównika (takie zwierzę)\n (nie koniecznie
rzeczywista)')
ax.text(75, .025, r'$\mu=115,\ \sigma=15$')
ax.axis([55, 175, 0, 0.03])
ax.grid(True)
fig.show()
```

Listing 70. Przykład dodania tekstu do wykresu.

[Tekst alternatywny. Przykład dodania tekstu do wykresu wykonanego za pomocą biblioteki Matplolib.]



Rys. 8. Przykład dodania tekstu do wykresu

Używanie wyrażeń matematycznych w tekście

Biblioteka Matplotlib akceptuje wyrażenia *TeX* w dowolnym wyrażeniu tekstowym. Na przykład, aby zapisać wyrażenie w tytule, możesz zapisać wyrażenie TeX $\sigma_i=15$ otoczone znakami dolara:

```
ax.set_title(r'$\sigma_i=15$')
```

gdzie r poprzedzające ciąg tytułu oznacza, że ciąg jest ciągiem surowym i nie należy traktować ukośników odwrotnych jako znaków języka Python. Biblioteka Matplotlib posiada wbudowany parser wyrażeń TeX, a także zawiera własne czcionki matematyczne.

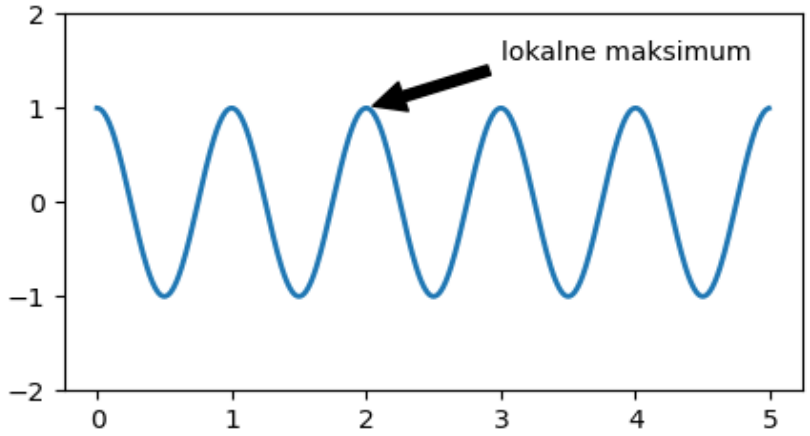
Adnotacje

Możemy również dodawać adnotacje do punktów na wykresie, często łącząc strzałkę wskazującą przez współrzędne *xy* z fragmentem tekstu w punkcie *xytext*:

```
fig, ax = plt.subplots(figsize=(5, 2.7))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2 * np.pi * t)
line, = ax.plot(t, s, lw=2)

ax.annotate('lokalne maksimum', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.set_ylim(-2, 2)
fig.show()
```

Listing 71. Przykład dodania adnotacji ze strzałką do wykresu.

[Tekst alternatywny. Przykład dodania adnotacji ze strzałką do wykresu wykonanego za pomocą biblioteki Matplolib.]



Rys. 9. Przykład dodania adnotacji ze strzałką do wykresu

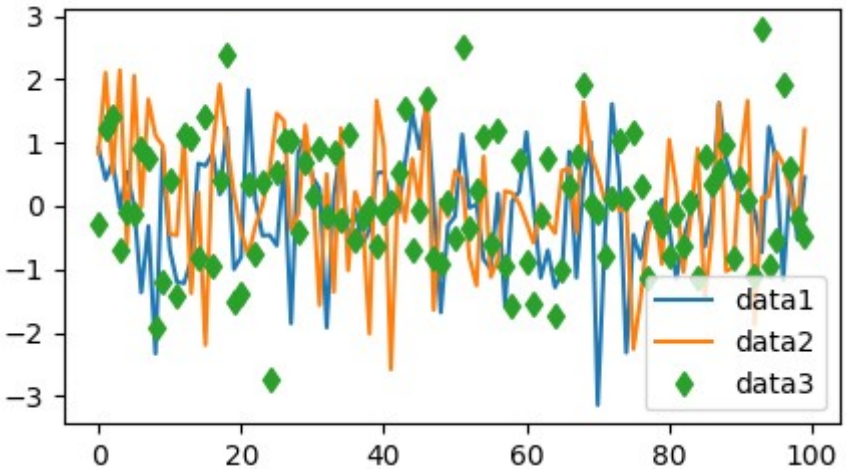
Legendy

Często chcemy identyfikować linie lub znaczniki za pomocą legendy, umożliwia to metoda *Axes.legend*:

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(np.arange(len(data1)), data1, label='data1')
ax.plot(np.arange(len(data2)), data2, label='data2')
ax.plot(np.arange(len(data3)), data3, 'd', label='data3')
ax.legend()
```

Listing 72. Przykład tworzenia legendy wykresu.

[Tekst alternatywny. Przykład tworzenia legendy wykresu wykonanego za pomocą biblioteki Matplolib.]



Rys. 10. Przykład tworzenia legendy wykresu

Skale i znaczniki osi

Każda oś ma dwa (lub trzy) obiekty osi reprezentujące osie x i y. Kontrolują one skalę osi, lokalizatory znaczników i formaty znaczników. Można dołączyć dodatkowe osie, aby wyświetlić kolejne obiekty osi.

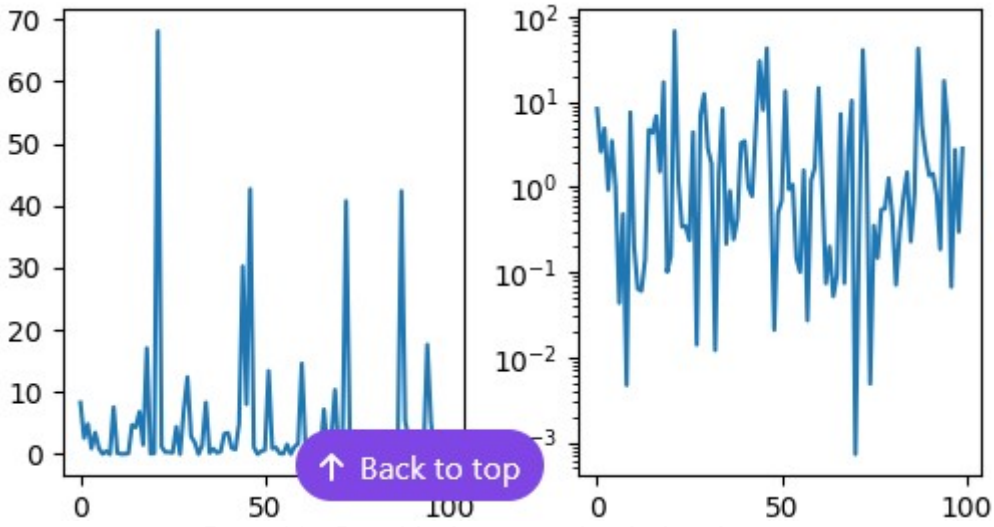
Skale

Oprócz skali liniowej, biblioteka Matplotlib udostępnia skale nieliniowe, takie jak skala logarytmiczna. Ponieważ skale logarytmiczne są tak często używane, istnieją również metody bezpośrednie, takie jak *loglog*, *semilogx* i *semilogy*. W tym przypadku ustawiamy skalę ręcznie:

```
fig, axs = plt.subplots(1, 2, figsize=(5, 2.7), layout='constrained')
xdata = np.arange(len(data1))  # tworzy listę porządkową
data = 10**data1
axs[0].plot(xdata, data)

axs[1].set_yscale('log')
axs[1].plot(xdata, data)
```

Listing 73. Przykład tworzenia skal wykresu.

[Tekst alternatywny. Przykład tworzenia skal wykresu wykonanego za pomocą biblioteki Matplolib.]



Rys. 11. Przykład tworzenia skal wykresu

Skala określa odwzorowanie wartości danych na odstępy wzdłuż osi. Dzieje się to w obu kierunkach i jest łączone w transformację, podobnie jak w przypadku mapowania współrzędnych danych w bibliotece Matplotlib na osie, figury lub współrzędne ekranu.

Lokalizatory i formatery znaczników

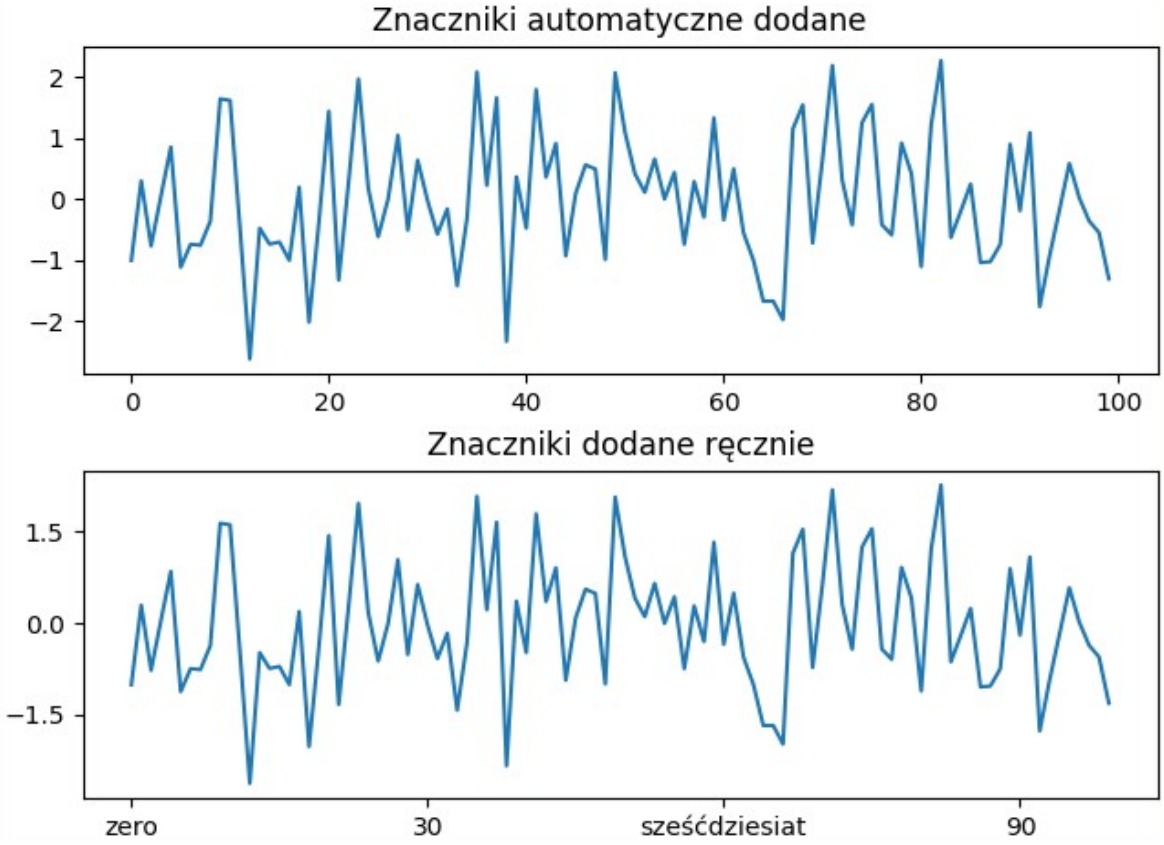
Każda oś ma lokalizator i formater znaczników, które wybierają, gdzie na obiektach osi mają zostać umieszczone znaczniki. Metodą umożliwiającą formatowanie i lokalizowanie znaczników jest *set_xticks*:

```
import matplotlib.pyplot as plt
import numpy as np

data1, data2, data3, data4 = np.random.randn(4, 100)
xdata = np.arange(len(data1))

fig, axs = plt.subplots(2, 1, layout='constrained')

axs[0].plot(xdata, data1)
axs[0].set_title('Znaczniki automatyczne dodane')
axs[1].plot(xdata, data1)
axs[1].set_xticks(np.arange(0, 100, 30), ['zero', '30', 'sześćdziesiat',
'90'])
axs[1].set_yticks([-1.5, 0, 1.5])  # zauważ że nie musimy tu definiować
znaczników
axs[1].set_title('Znaczniki dodane ręcznie')
fig.show()
```

Listing 74. Przykład tworzenia znaczników wykresu.

Politechnika Świętokrzyska
Kielce University of Technology
Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki” nr FERS.01.05-IP.08-0234/23

[Tekst alternatywny. Przykład tworzenia znaczników wykresu wykonanego za pomocą biblioteki Matplolib.]



Rys. 12. Przykład tworzenia znaczników wykresu

Różne skale mogą mieć różne lokalizatory i formatery; na przykład skala logarytmiczna używa metody *LogLocator* i *LogFormatter*.

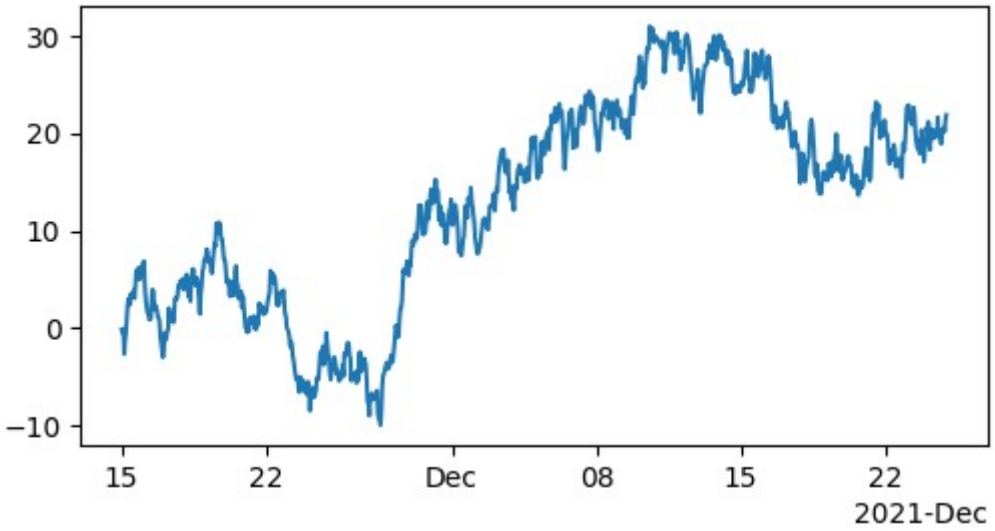
Wykreślanie dat i ciągów znaków

Biblioteka Matplotlib obsługuje wykreślanie tablic dat i tablic ciągów znaków, a także liczb zmiennoprzecinkowych. W razie potrzeby otrzymują one specjalne lokalizatory i formatery, dla dat:

```
from matplotlib.dates import ConciseDateFormatter

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
dates = np.arange(np.datetime64('2021-11-15'), np.datetime64('2021-12-
25'),
                  np.timedelta64(1, 'h'))
data = np.cumsum(np.random.randn(len(dates)))
ax.plot(dates, data)
ax.xaxis.set_major_formatter(ConciseDateFormatter(ax.xaxis.get_major_loc
ator()))
```

Listing 75. Wykreślanie dat.

[Tekst alternatywny. Przykład tworzenia wykresu z osią czasu wykonanego za pomocą biblioteki Matplolib.]



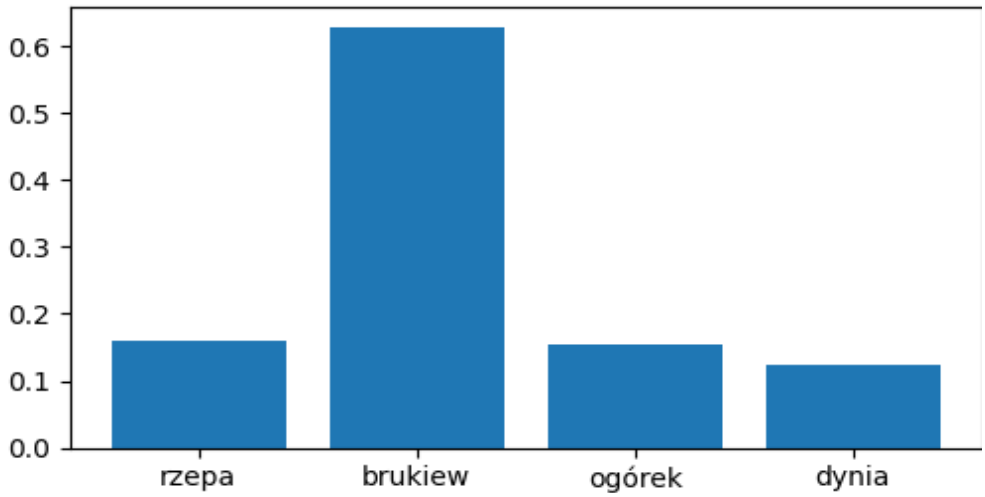
Rys. 13. Przykład tworzenia wykresu z osią czau.

W przypadku ciągów znaków otrzymujemy wykres kategoryczny

```
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
categories = ['rzepa', 'brukiew', 'ogórek', 'dynia']

ax.bar(categories, np.random.rand(len(categories)))
fig.show()
```

Listing 76. Wykreślanie ciągów znaków.

[Tekst alternatywny. Przykład tworzenia wykresu z kategoriami wykonanego za pomocą biblioteki Matplolib.]



Rys. 14. Przykład tworzenia wykresu z kategoriami.

Dodatkowe osie układu

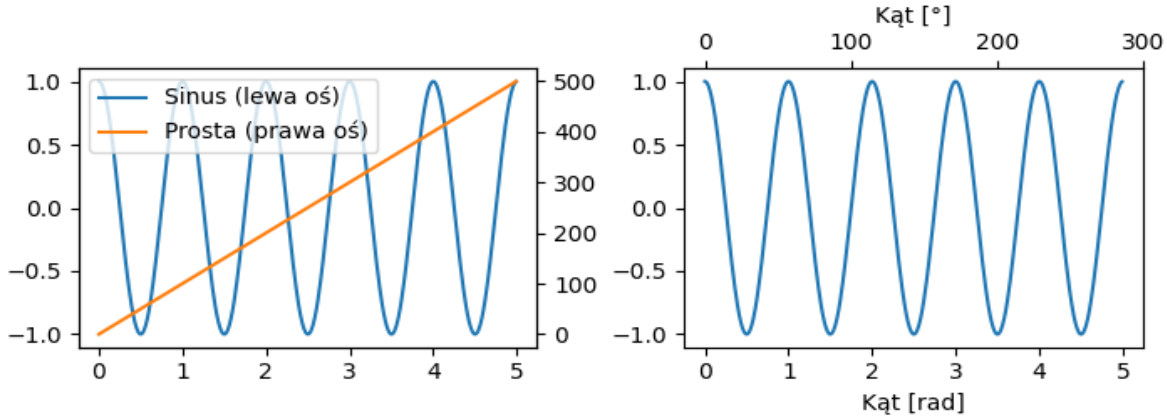
Rysowanie danych o różnej wielkości na jednym wykresie może wymagać dodatkowej osi. Taką oś można utworzyć za pomocą metody *twinx* lub *twiny*, dodając nowe osie z niewidoczną osią X i osią Y.

Podobnie, możemy dodać osie za pomocą metod *secondary_xaxis* lub *secondary_yaxis* o innej skali niż oś główna, aby przedstawić dane w innych skalach lub jednostkach.

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2 * np.pi * t)
fig, (ax1, ax3) = plt.subplots(1, 2, figsize=(7, 2.7),
layout='constrained')
l1, = ax1.plot(t, s)
ax2 = ax1.twinx()
l2, = ax2.plot(t, range(len(t)), 'C1')
ax2.legend([l1, l2], ['Sinus (lewa oś)', 'Prosta (prawa oś)'])

ax3.plot(t, s)
ax3.set_xlabel('Kąt [rad]')
ax4 = ax3.secondary_xaxis('top', (np.rad2deg, np.deg2rad))
ax4.set_xlabel('Kąt [°]')
fig.show()
```

Listing 77. Wykres z dodatkowymi osiami.

[Tekst alternatywny. Przykład tworzenia wykresu z wieloma osiami wykonanego za pomocą biblioteki Matplolib.]



Rys. 15. Przykład tworzenia wykresu z wieloma osiami.

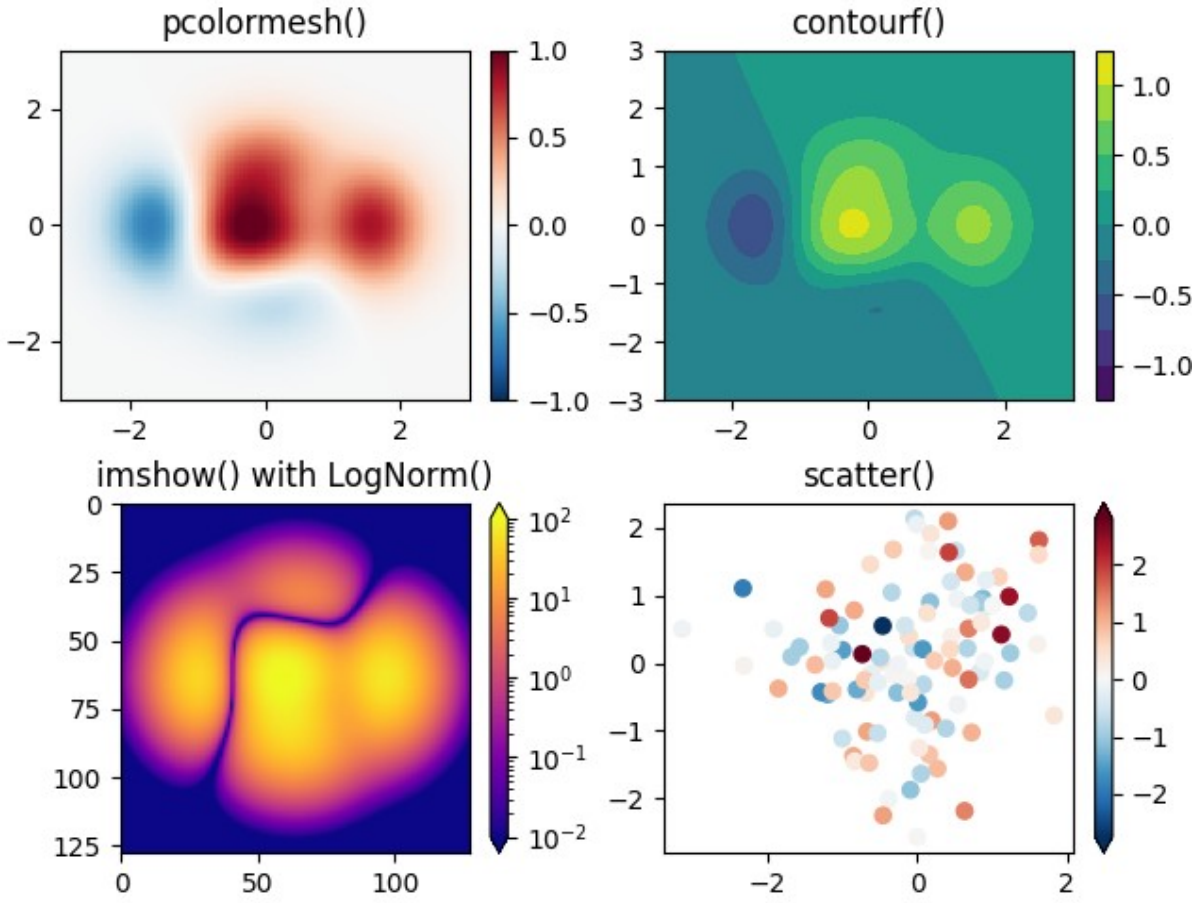
Mapy kolorów

Często chcemy mieć trzeci wymiar na wykresie reprezentowany przez kolory na mapie kolorów. Biblioteka Matplotlib oferuje wiele typów wykresów, które to umożliwiają:

```
from matplotlib.colors import LogNorm

X, Y = np.meshgrid(np.linspace(-3, 3, 128), np.linspace(-3, 3, 128))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

fig, axs = plt.subplots(2, 2, layout='constrained')
pc = axs[0, 0].pcolormesh(X, Y, Z, vmin=-1, vmax=1, cmap='RdBu_r')
fig.colorbar(pc, ax=axs[0, 0])
axs[0, 0].set_title('pcolormesh()')

co = axs[0, 1].contourf(X, Y, Z, levels=np.linspace(-1.25, 1.25, 11))
fig.colorbar(co, ax=axs[0, 1])
axs[0, 1].set_title('contourf()')

pc = axs[1, 0].imshow(Z**2 * 100, cmap='plasma', norm=LogNorm(vmin=0.01,
vmax=100))
fig.colorbar(pc, ax=axs[1, 0], extend='both')
axs[1, 0].set_title('imshow() with LogNorm()')

pc = axs[1, 1].scatter(data1, data2, c=data3, cmap='RdBu_r')
fig.colorbar(pc, ax=axs[1, 1], extend='both')
axs[1, 1].set_title('scatter()')
```

Listing 78. Mapa kolorów.

[Tekst alternatywny. Przykład tworzenia wykresu z mapami kolorów wykonanego za pomocą biblioteki Matplolib.]



Rys. 16. Przykład tworzenia mapy kolorów.

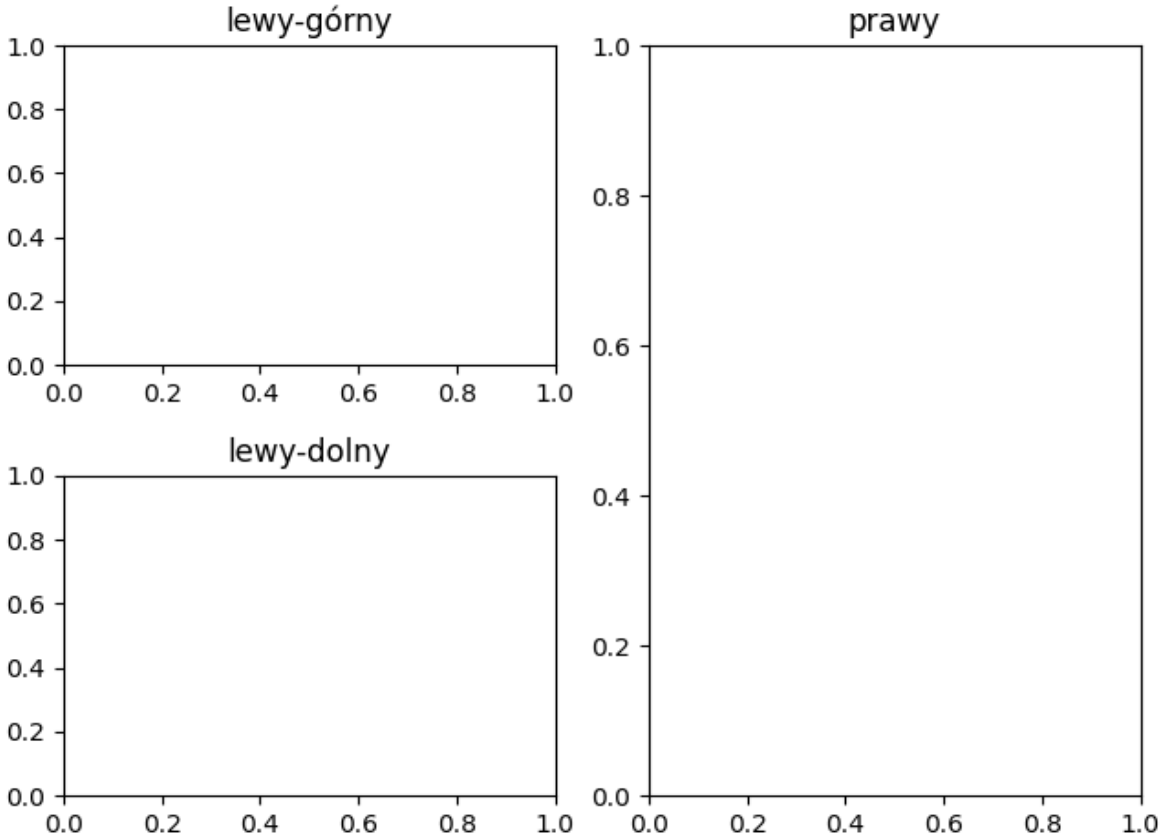
Wiele wykresów w jednym oknie

Wiele okien figur można utworzyć za pomocą wielokrotnego wywołania *fig = plt.figure()* lub *fig2, ax = plt.subplots()*. Zachowując odwołania do obiektów, można formatować każdy obiekt *figure* oddzielnie. Kolejne wykresy w oknie *figure* można dodawać na wiele sposobów, ale najprostszym jest polecenie *plt.subplots()*. Bardziej złożone układy, z obiektami osi obejmującymi kolumny lub wiersze, można uzyskać za pomocą *subplot_mosaic*:

```
fig, axd = plt.subplot_mosaic([['upleft', 'right'],
['lowleft', 'right']], layout='constrained')
axd['upleft'].set_title('lewy-górny')
axd['lowleft'].set_title('lewy-dolny')
axd['right'].set_title('prawy')
fig.show()
```

Listing 79. Podział okna na wiele wykresów

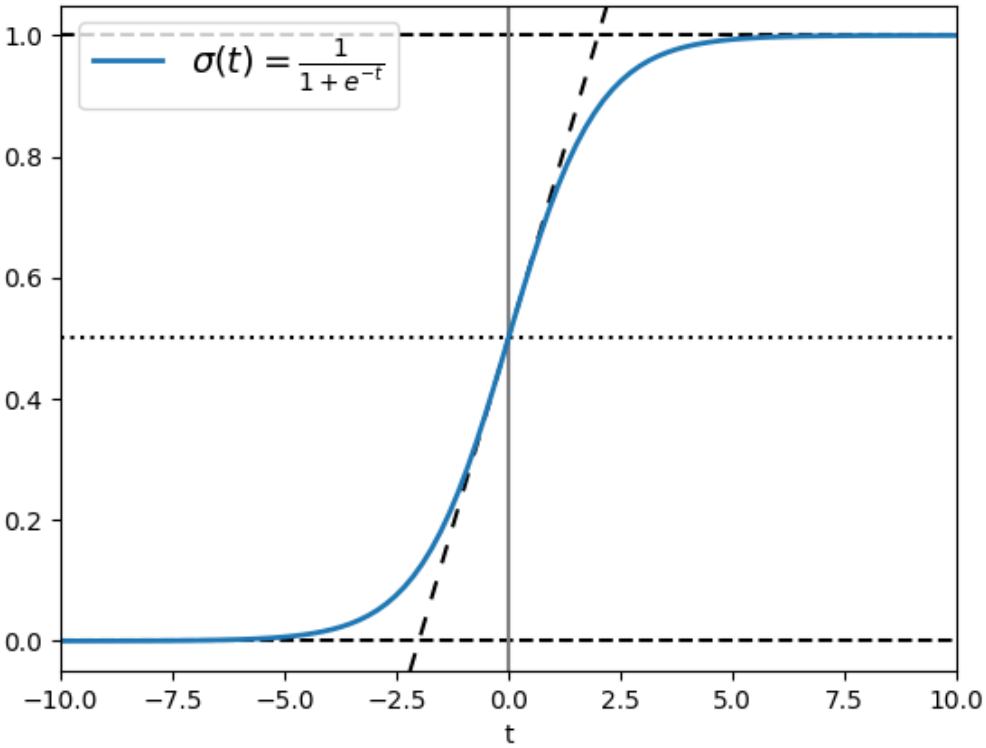
[Tekst alternatywny. Przykład tworzenia okna z wieloma wykresami wykonanego za pomocą biblioteki Matplolib]



Rys. 17. Przykład tworzenia okna z wieloma wykresami.

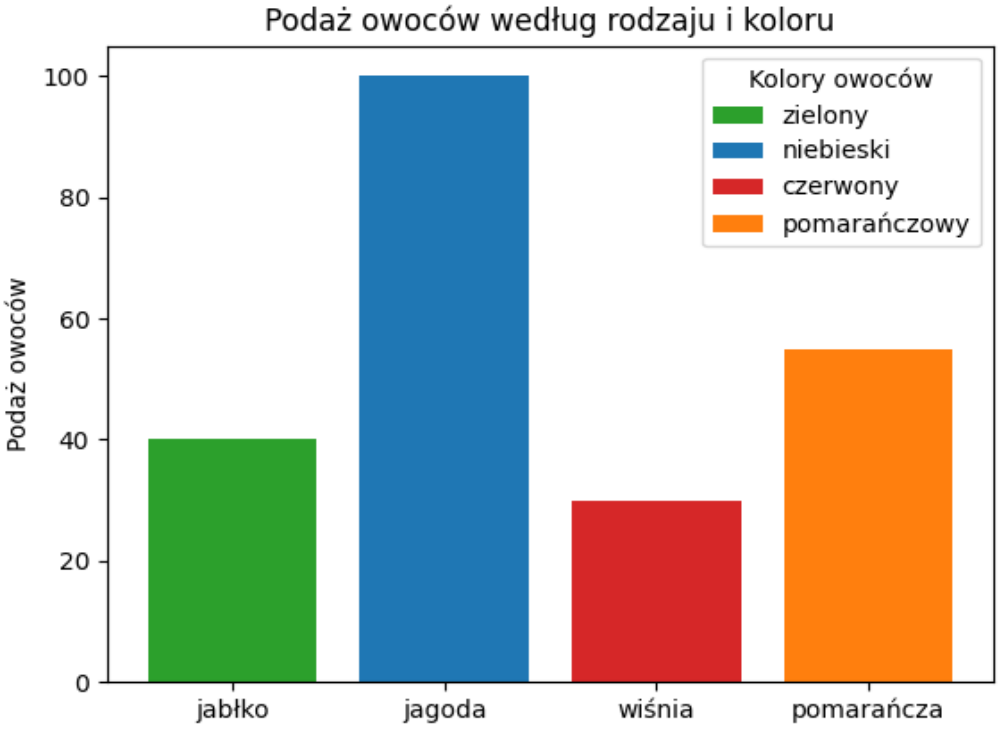
5.3. Przykładowe ćwiczenia

Ćwiczenie 5.1. Narysuj wykres funkcji, do rysowania linii przerywanych można użyć metod *axvline* i *axhline*:

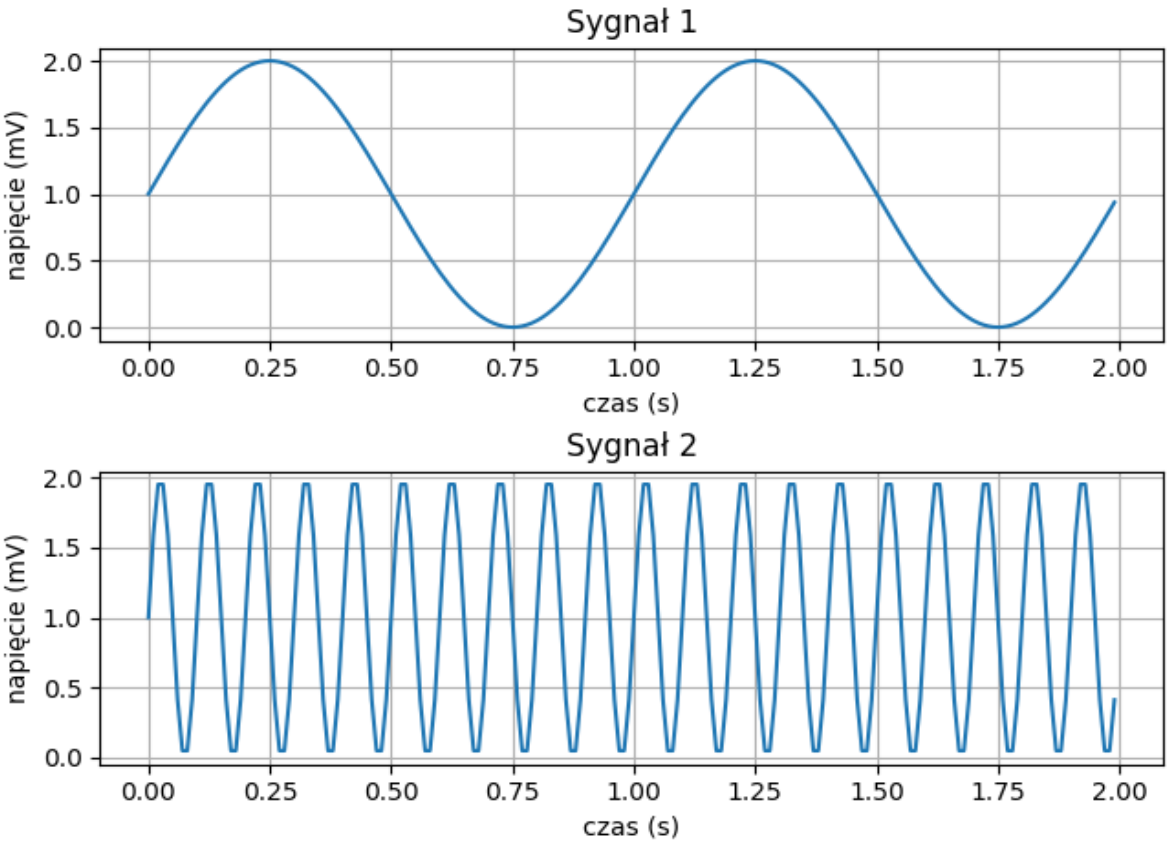


Ćwiczenie 5.2. Dla danych zawartych w listach narysuj wykres słupkowy o określonych kolorach słupków (atrybut *color* metody *bar*):

fruits = ['jabłko', 'jagoda', 'wiśnia', 'pomarańcza']; counts = [40, 100, 30, 55]



Ćwiczenie 5.3. Narysuj wykresy funkcji $f(t)=1+\sin(2*\pi*t)$ w przedziale [0,2] z krokiem 0.01, oraz w drugim wykresie tego samego okna wykres funkcji $f(t)=1+\sin(20*\pi*t)$:



6. Rozwiązania ćwiczeń

```
import numpy as np
arr = np.arange(10).reshape(2, -1)
print(arr)
```

Listing 80. Rozwiązanie ćw. 3.1.

```
mat1 = np.random.random((5, 3))
mat2 = np.random.random((3, 2))
result = np.dot(mat1, mat2)
print(result)
```

Listing 81. Rozwiązanie ćw. 3.2.

```
import numpy as np
arr = np.arange(1, 11)
odd_numbers = arr[arr % 2 != 0]
print(odd_numbers)
```

Listing 82. Rozwiązanie ćw. 3.3.

```
arr = np.arange(1, 10)
arr[arr % 2 != 0] = -1
print(arr)
```

Listing 83. Rozwiązanie ćw. 3.4.

```
arr = np.array([-1, 2, 0, -4, 5])
boolean_arr = arr > 0
print(boolean_arr)
```

Listing 84. Rozwiązanie ćw. 3.5.

```
arr = np.arange(1, 10)
arr[arr % 2 == 0] *= -1
print(arr)
```

Listing 85. Rozwiązanie ćw. 3.6.

```
matrix = np.random.random((3,3))
normalized_matrix = (matrix - np.mean(matrix)) / np.std(matrix)
print(normalized_matrix)
```

Listing 86. Rozwiązanie ćw. 3.7.

```
matrix = np.random.random((3, 3))
diagonal_sum = np.trace(matrix)
print(diagonal_sum)
```

Listing 87. Rozwiązanie ćw. 3.8.

```
arr = np.array([1, 2, 0, 0, 4, 0])
non_zero_indices = np.nonzero(arr)
print(non_zero_indices)
```

Listing 88. Rozwiązanie ćw. 3.9.

```
arr = np.arange(10)
reversed_arr = np.flip(arr)
print(reversed_arr)
```

Listing 89. Rozwiązanie ćw. 3.10.

```
identity_matrix = np.eye(3)
print(identity_matrix)
```

Listing 90. Rozwiązanie ćw. 3.11.

```
arr = np.arange(10)
reshaped_arr = arr.reshape(5, 2)
print(reshaped_arr)
```

Listing 91. Rozwiązanie ćw. 3.12.

```
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([3, 4, 5, 6, 7])
common_items = np.intersect1d(arr1, arr2)
print(common_items)
```

Listing 92. Rozwiązanie ćw. 3.13.

```
import pandas as pd
df = pd.DataFrame({'X':[78,85,96,80,86], 'Y':[84,94,89,83,86],'Z':
[86,97,96,72,83]});
print(df)
```

Listing 93. Rozwiązanie ćw. 4.1.

```
import pandas as pd
df = pd.DataFrame({'X':[78,85,96,80,86], 'Y':[84,94,89,83,86],'Z':
[86,97,96,72,83]});
print(df)
```

Listing 94. Rozwiązanie ćw. 4.2.

```
import pandas as pd
import numpy as np

exam_data  = {'name': ['Anastasia', 'Dima', 'Katherine', 'James',
'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],
        'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
        'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no',
'no', 'yes']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

df = pd.DataFrame(exam_data , index=labels)
print(df)
```

Listing 95. Rozwiązanie ćw. 4.3.

```
import pandas as pd
import numpy as np

exam_data  = {'name': ['Anastasia', 'Dima', 'Katherine', 'James',
'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],
        'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
```

```
        'attempts' : [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no',
'no', 'yes']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

df = pd.DataFrame(exam_data , index=labels)
print("Liczba podejśc większych od 2:")
print(df[df['attempts'] > 2])
```

Listing 96. Rozwiązanie ćw. 4.4.

```
import pandas as pd
import numpy as np
exam_data  = {'name': ['Anastasia', 'Dima', 'Katherine', 'James',
'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],
        'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
        'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no',
'no', 'yes']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
df = pd.DataFrame(exam_data , index=labels)
print("Dane:")
print(df)
print("\nDodaj nowy wiersz:")
df.loc['k'] = [1, 'Suresh', 'yes', 15.5]
print("Print all records after insert a new record:")
print(df)
print("\nUsuń nowy wiersz:")
df = df.drop('k')
print(df)
```

Listing 97. Rozwiązanie ćw. 4.5.

```
import pandas as pd
import numpy as np
exam_data  = {'name': ['Anastasia', 'Dima', 'Katherine', 'James',
'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],
        'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
        'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no',
'no', 'yes']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
df = pd.DataFrame(exam_data , index=labels)
print("Dane:")
print(df)
df = df.sort_values(by=['name', 'score'], ascending=[False, True])
print("Dane posortowane:")
print(df)
```

Listing 98. Rozwiązanie ćw. 4.6.

```
import pandas as pd
import numpy as np
```

```
exam_data = [{'name':'Anastasia', 'score':12.5},
{'name':'Dima','score':9}, {'name':'Katherine','score':16.5}]
df = pd.DataFrame(exam_data)
for index, row in df.iterrows():
    print(row['name'], row['score'])
```

Listing 99. Rozwiązanie ćw. 4.7.

```
import pandas as pd
import numpy as np
d = {'col1': [1, 4, 3, 4, 5], 'col2': [4, 5, 6, 7, 8], 'col3': [7, 8, 9,
0, 1]}
df = pd.DataFrame(data=d)
print("Original DataFrame")
print(df)
print('Rows for colum1 value == 4')
print(df.loc[df['col1'] == 4])
```

Listing 100. Rozwiązanie ćw. 4.8.

```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(-10, 10, 100)
sig = 1 / (1 + np.exp(-t))

fig, ax = plt.subplots()
ax.axhline(y=0, color="black", linestyle="--")
ax.axhline(y=0.5, color="black", linestyle=":")
ax.axhline(y=1.0, color="black", linestyle="--")
ax.axvline(color="grey")
ax.axline((0, 0.5), slope=0.25, color="black", linestyle=(0, (5, 5)))
ax.plot(t, sig, linewidth=2, label=r"$\sigma(t) = \frac{1}{1 + e^{-t}}
$")
ax.set(xlim=(-10, 10), xlabel="t")
ax.legend(fontsize=14)
plt.show()
```

Listing 101. Rozwiązanie ćw. 5.1.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

fruits = ['jabłko', 'jagoda', 'wiśnia', 'pomarańcza']
counts = [40, 100, 30, 55]
bar_labels = ['zielony', 'niebieski', 'czerwony', 'pomarańczowy']
bar_colors = ['tab:green', 'tab:blue', 'tab:red', 'tab:orange']

ax.bar(fruits, counts, label=bar_labels, color=bar_colors)

ax.set_ylabel('Podaż owoców')
```

```
ax.set_title('Podaż owoców według rodzaju i koloru')
ax.legend(title='Kolory owoców')

plt.show()
```

Listing 102. Rozwiązanie ćw. 5.2.

```
import matplotlib.pyplot as plt
import numpy as np

# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s1 = 1 + np.sin(2 * np.pi * t)
s2 = 1 + np.sin(20 * np.pi * t)

fig, (ax0, ax1) = plt.subplots(2, 1, layout='constrained')
ax0.plot(t, s1)

ax0.set(xlabel='czas (s)', ylabel='napięcie (mV)',
       title='Sygnał 1')
ax0.grid()

ax1.plot(t, s2)

ax1.set(xlabel='czas (s)', ylabel='napięcie (mV)',
       title='Sygnał 2')
ax1.grid()

plt.show()
```

Listing 103. Rozwiązanie ćw. 5.3.

7. Literatura

1. Biblioteka numpy. (https://numpy.org/, dostępny 29.06.2025).
2. Przykladowe ćwiczenia numpy (https://www.w3resource.com/python-exercises/numpy, dostępny 29.06.2025)
3. Biblioteka pandas. (https://pandas.pydata.org/, dostępny 29.06.2025).
4. Przykladowe ćwiczenia pandas (https://www.w3resource.com/python-exercises/pandas, dostępny 29.06.2025)
5. Biblioteka matplotlib. (https://matplotlib.org/, dostępny 29.06.2025).

8. Spis rysunków

9. Spis listingów